UNBC

CPSC 320

READINGS PACKAGE

J. Polajnar

Programming Languages

Fall 2004

NOT RETURNABLE

Printed by: UNBC Copy & Publishing Service

sku:10372487

Prentice-Hall International
Series in Computer Science

C. A. R. Hoare, Series Editor

Published

BACKHOUSE, R. C., *Syntax of Programming Languages: Theory and Practice*
de BAKKER, J. W., *Mathematical Theory of Program Correctness*
DUNCAN, F., *Microprocessor Programming and Software Development*
HENDERSON, P., *Functional Programming: Application and Implementation*
JONES, C. B., *Software Development: A Rigorous Approach*
TENNENT, R. D., *Principles of Programming Languages*
WELSH, J. and ELDER, J., *Introduction to PASCAL*
WELSH, J. and MCKEAG, M., *Structured System Programming*

Future Titles

JACKSON, M. A., *System Design*
JOHNSTON, H., *Learning to Program with PASCAL*
NAUR, P., *Studies in Program Analysis and Construction*
REYNOLDS, J. C., *The Craft of Programming*
WELSH, J., ELDER, J., and BUSTARD, D., *Sequential and Concurrent Program Structures*

PRINCIPLES OF PROGRAMMING LANGUAGES

R. D. TENNENT
Department of Computing and Information Science
Queen's University, Kingston, Canada

Prentice/Hall PHI International

ENGLEWOOD CLIFFS, NEW JERSEY LONDON NEW DELHI
SINGAPORE SYDNEY TOKYO TORONTO WELLINGTON

Library of Congress Cataloging in Publication Data

TENNENT, R D 1944-
Principles of programming languages.
Bibliography: p.
Includes index.
1. Programming language (Electronic computers)
I. Title.
QA76.7 T47 001.64'24 80-24271
ISBN 0-13-709873-1

British Library Cataloguing in Publication Data

TENNENT, R D
Principles of programming languages.
1. Programming languages (Electronic computers)
2. Electronic digital computers—Programming
I. Title
001.64'24 QA76.7
ISBN 0-13-709873-1

ISBN 0-13-709873-1

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY., LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE., LTD., *Singapore*
PRENTICE-HALL, INC., *Englewood Cliffs, New Jersey*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

to Liz

CONTENTS

PREFACE

This book is a systematic exposition of the fundamental concepts and general principles underlying programming languages in current use. It may be used as a text for courses in computing science and software engineering programs, and as a reference by advanced programmers, programming theorists, and programming language implementers, describers and designers. Linguists and logicians may also be interested to see how the methods of mathematical logic may be applied to formal languages that are much more complex than the traditional logical calculi.

The material and the presentation have been strongly influenced by the approach to programming language theory founded by Dana Scott and the late Christopher Strachey at Oxford University, particularly the first chapter of *A Theory of Programming Language Semantics* by Robert Milne and Strachey (Chapman and Hall, London, and Wiley, New York). But I have emphasized intuitive concepts, rather than formalism and mathematical theory. I hope that this will help to make their work accessible to a wider audience.

Readers are expected to have enough programming experience to appreciate the basic ideas of programming methodology (importance of program correctness, readability and modularity, as well as efficiency; separation of levels of abstraction; stepwise refinement), and to have a reading knowledge of PASCAL, which is used as a standard example throughout. There are also "case studies" of interesting aspects of several other languages used in practice, but no attempt is made to give complete descriptions of languages, or to discuss experimental languages. The emphasis is on significant differences and similarities between linguistic concepts. The only mathematical prerequisite is a basic knowledge of sets and functions.

Undergraduates with adequate programming experience and mathematical maturity can cover all the material in the order presented in two terms. For students with weaker backgrounds, the "starred" sections

(on the principles underlying Scott's theory of computation) may be omitted. It is also possible to use the final chapter as the outline of an introductory graduate course in formal description of programming languages, referring to material in earlier chapters as needed.

There are exercises, project suggestions and an annotated bibliography at the end of almost every chapter. An additional bibliography of suggested readings on each of the programming languages mentioned in the text is given in an appendix.

I am very grateful to everyone who gave me suggestions and comments on various drafts, particularly Michael Gordon, Robert Milne, Tony Hoare, David Barnard, Mike Jenkins, Molly Higginson, David Leeson, Bill O'Farrell, John Gauch and Bruce Stratton. The remaining errors, obscurities and prejudices are my responsibility. I would also like to thank Michael Levison for his help in preparing the manuscript with his IVI text-editing system and the Natural Sciences and Engineering Research Council of Canada for financial assistance.

R. D. T.

PRINCIPLES OF PROGRAMMING LANGUAGES

The meaning of a sentence must remain unchanged when a part of the sentence is replaced by an expression having the same meaning.

G. FREGE (1892)

1 INTRODUCTION

1.1 PROGRAMMING LANGUAGES

A programming language is a system of notation for describing computations. A useful programming language must therefore be suited both for *describing* (i.e., for human writers and readers of programs), and for *computation* (i.e., for efficient implementation on computers). But human beings and computers are so different that it is difficult to find notational devices that are well suited to the capabilities of both. Languages that favor humans are termed *high-level*, and those oriented to machines *low-level*.

Let us consider some extreme examples of programming languages. In principle, the most "powerful" language for any computer is its machine language, which provides direct access to all of the resources of that computer. However, programs in such a language cannot conveniently be implemented on *other* computers. Furthermore, it is very difficult to write or read machine-language programs. Human beings cannot cope with the complete lack of structure in both programs (sequences of machine instructions) and data representations (sequences of machine words).

It might be thought that "natural" languages (such as English and French) would be at the other extreme. But, in most fields of science and technology, the formalized symbolic notations of mathematics and logic have proved to be indispensable for precise formulation of concepts and principles and for effective reasoning. However, in their full generality the notational devices of mathematics are not even implementable on computers, for deep reasons that will be discussed later.

There is a language called LAMBDA (invented by D. Scott) that has many of the properties of conventional mathematical notations and is as expressive as possible: *all* and *only* the operations that apparently are possible to compute are definable in LAMBDA. These properties make it useful as a specification language and in theoretical studies of computability.

But LAMBDA is so far removed from conventional computers that, though implementable in principle, it would not be practical as a *programming* language.

In short, an ideal programming language would combine the advantages of machine languages and mathematical notations, but achieving this aim has proved to be a very difficult problem. Many existing languages have only managed to combine countless "features" into a jumble that is neither easy to implement nor a pleasure to use.

There are so many programming languages and most are so complex and irregular that it would be nearly impossible and certainly pointless to learn *every* feature of *every* existing programming language (or even of the dozen or so more important ones). Fortunately, there is a great deal of *conceptual* overlap between programming languages, even those that on the surface appear to be quite dissimilar. Almost every practical programming language has mechanisms for dynamically updating storage, introducing symbolic names, transferring control, structuring data, defining procedures, and so on. In every language, these mechanisms are governed by the same general principles.

It is on these fundamental concepts and general principles that this book concentrates. Understanding them will make it easier to use, describe, compare, implement, and design programming languages.

It will be convenient to use a single programming language as a standard example in this book. PASCAL has been chosen because it is widely known and has been one of the most successful at reconciling conflicting design criteria (though it is certainly not the final step in the evolution of programming languages!) The reader is assumed to have a reading knowledge of PASCAL as well as experience in programming with some high-level language. Jensen and Wirth (1974) or a comparable description should be available for reference. Minor variants or extensions of PASCAL will be described and discussed when convenient or necessary to illustrate a point. Several case studies of other well-known languages will provide a broader perspective. Appendix A is a bibliography of suggested readings for each of the programming languages discussed. It should be noted that many of the program fragments used as examples are intended only to illustrate language concepts and do not necessarily exemplify good programming style.

1.2 SYNTAX, SEMANTICS AND PRAGMATICS

It is traditional when dealing with languages of all sorts to try to separate concerns with form, the subject of ***syntax****, from concerns with meaning, the

*Important technical terms are introduced in bold italic face.

field of ***semantics***. Consider the simple "language" of *binary numerals*. Some examples of binary numerals are

```
0
1
101
0101
10011010
```

A communication in this language evidently consists of a finite sequence of characters '0' and '1'. This is just syntax however, and says nothing about what such a communication is intended to *mean*.

The usual interpretation for such numerals is that each numeral denotes a *natural number* (i.e., zero or one of its successors). For example, '101' and '0101' both denote the number five, the fifth successor of zero. Numbers are "abstract" mathematical concepts, whereas the digit strings that appear on paper are numerals, that is to say, symbolic representations or descriptions of numbers. Many other languages have this same set of numbers as their meanings: decimal numerals, Roman numerals, and so on.

In general, then, *syntax* is concerned with only the format, well-formedness, and compositional structure of communications in a language, and *semantics* with their meaning.

The ***pragmatics*** of languages have to do with their origins, uses, and effects. So, the pragmatic aspects of programming languages include language implementation techniques, programming methodology, and the history of programming-language development. In this book, important pragmatic considerations will be pointed out wherever appropriate, but systematic expositions of programming methodology, language implementation, and history are outside its scope.

The criterion for correctness of a language processor is that it implement the syntax and semantics of the language. However, because of pragmatic factors, processors often do not meet their specifications for all possible programs and data. For example, suppose that the language of binary numerals were to be "implemented" by representing numbers in a storage register of fixed size. It is evidently impossible for every numeral in the language to be correctly implemented as specified.

If a processor is unable to meet its specifications for some input, it should signal this with an appropriate warning message. Otherwise, it is termed ***insecure***. Output from an insecure processor must be treated with suspicion unless it can be verified that the program has not breached any of the insecurities.

An important goal of programming language design is to make it easier

for in enters to eliminate insecurities without incurring severe penalties in execution time or storage space. Unfortunately, with most current computer designs, some kinds of programming error cannot be detected economically, so that the goal of eliminating insecurities should also be taken up by computer designers.

1.3 SYNTAX-DIRECTED SEMANTICS

Programmers are encouraged to program in a "structured" way, that is to say, to use the *syntactic* structures of their programming language to help them systematically develop and more clearly express the *semantic* structure of their algorithms. Similarly, languages are best *described* by basing specifications of their semantics on an appropriate syntactic description. Programming languages are so complex that a structured approach is almost essential for conceptual understanding.

As a simple example of syntax-directed semantic description, consider again the language of binary numerals. The syntax of this language may be precisely specified as follows:

(a) Characters '0' and '1' are binary numerals.
(b) If N is a binary numeral, then N with a '0' or a '1' appended to the right of it is also a binary numeral.
(c) These are the only binary numerals.

Rule (a) describes the two *elementary* (i.e., nondecomposable) syntactic forms. Rule (b) describes the two *composite* forms; in this rule, the binary numeral N referred to is an example of what is termed an ***immediate constituent*** (of a composite syntactic form). Rule (c) specifies that the set of binary numerals is to be the *smallest* set meeting requirements (a) and (b).

Note that this syntactic description specifies not only the criteria for well-formedness of a binary numeral, but also its ***phrase structure***, that is to say, how it is analyzed into immediate constituents, and these into their immediate constituents, and so on, until elementary forms are reached. For example, the phrase structure of binary numeral '0101' may be depicted by the tree shown in Fig. 1.1. The process of determining the phrase structure of text is known as ***parsing***.

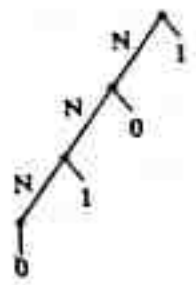


Fig. 1.1

A specification of the meaning of (i.e., the number denoted by) every binary numeral may now be based on the above syntactic description as follows:

(a) Binary numerals '0' and '1' denote numbers zero and one, respectively.
(b) If N is a binary numeral that denotes number n, then (i) N with '0' appended to the right of it denotes number $2\times n$, and (ii) N with '1' appended to the right of it denotes number $2\times n+1$.

For example, consider numeral '0101'. Working from the leftmost character,

'0' denotes zero, using rule (a);
hence, '01' denotes $2\times 0+1=1$, using rule (b), part (ii);
hence, '010' denotes $2\times 1=2$, using rule (b), part (i);
hence, '0101' denotes $2\times 2+1=5$, using rule (b), part (ii).

Each non-terminal node of the phrase structure tree for '0101' may be "labelled" with the semantic object denoted by the corresponding phrase (Fig. 1.2). Thus semantics chases denotation up the syntax tree (with apologies to W. V. Quine).

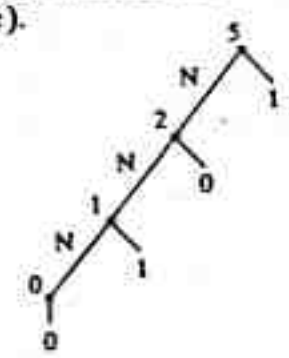


Fig. 1.2

The above description of the syntax and semantics of binary numerals is an example of what is known as the ***denotational*** approach to language description. The general idea is simply to specify the meanings of (i.e., the semantic objects denoted by) elementary forms directly, and the meanings of composites in terms of the meanings of their immediate constituents. This "structured" approach has a long history in logic and linguistics. Subsequent chapters will explain how *programming* languages may be described denotationally.

EXERCISES

1.1 Suggest two "unusual" semantic interpretations for binary numerals.

1.2 Suppose that rule (b) of the definition of the syntax of binary numerals were changed to

(b) If N is a binary numeral, then N *prefixed* by a '0' or a '1' is also a binary numeral.

Define the usual semantics of binary numerals using this syntactic description.

***1.3** Describe the syntax and usual semantics of binary numerals with fractions, such as '101.0101'.

***1.4** Prove that, according to the syntax and semantics given, *every* finite binary numeral has a *unique* meaning, using mathematical induction on the length of the numerals.

*Solutions to starred exercises require a higher level of mathematical maturity.

PROJECT

Write an essay on the history of one of the major programming languages.

BIBLIOGRAPHIC NOTES

There is a large literature on programming language design. Three papers by Hoare [1.6, 1.7, 1.9] are especially recommended. The language LAMBDA was described by Scott [1.15].

The trichotomy between syntax, semantics, and pragmatics was proposed by Morris [1.12, 1.13] and Carnap [1.2]. The history of programming languages is discussed in papers by Knuth and Pardo [1.10] and Hoare [1.8], in a book by Sammet [1.14], and in a conference proceedings [1.18]. There are large literatures on programming methodology and language implementation; see, for example, collections edited by Gries [1.5], and Bauer and Eickel [1.1], respectively.

The denotational approach to language description may be traced back to Frege [1.4], Carnap [1.3], and Tarski [1.17]. Its use for formal description of programming languages was developed by Scott and Strachey [1.16]. Montague [1.11] gave a denotational description of a fragment of a natural language.

1.1 Bauer, F. L. and J. Eickel (eds.) *Compiler Construction, An Advanced Course*, Springer, Berlin (2nd edition, 1976).

1.2 Carnap, R. *Introduction to Semantics*, Harvard University Press, Cambridge (1942).

1.3 Carnap, R. *Meaning and Necessity, A Study in Semantics and Modal Logic*, University of Chicago Press, Chicago (1947, enlarged edition 1956).

1.4 Frege, G. "Über Sinn und Bedeutung"; *Zeitschrift für Philosophie und Philosophisches Kritik* **100**, 25–50 (1892); English translations in *Readings in Philosophical Analysis* (eds., H. Feigl and W. Sellars), pp. 85–102, Appleton-Century-Crofts, New York (1949), and *Translations from the Philosophical Writings of Gottlob Frege* (eds., P. T. Geach and M. Black), pp. 56–78, Blackwell, Oxford (2nd edition, 1960).

1.5 Gries D. (ed.). *Programming Methodology*, Springer, New York (1978).

1.6 Hoare, C. A. R. "Prospects for a better programming language", in *High Level Languages*, Infotech State of the Art Report 7, pp. 325–43, Infotech Ltd., Maidenhead, England (1972).

1.7 Hoare, C. A. R. *Hints on Programming Language Design*, technical report CS–403, Computer Science Dept., Stanford University, Stanford, California (1973).

1.8 Hoare, C. A. R. "High level languages, the way behind", in *High Level Languages, The Way Ahead*, British Computer Society conference proceedings (ed., D. Simpson), pp. 17–25, NCC Publications, Manchester (1973).

1.9 Hoare, C. A. R. "The high cost of programming languages", in *Software Systems Engineering*, pp. 413–29 (papers presented at the European Computing Conference, London, 1976), Online Conferences Ltd., Uxbridge, England (1976).

1.10 Knuth, D. E. and L. T. Pardo. "The early development of programming languages", in *Encyclopedia of Computer Science and Technology*, vol. 7, pp. 419–93, Marcel Dekker, New York and Basel (1977); also technical report CS–562, Computer Science Dept., Stanford University, Stanford, California (1976).

1.11 Montague, R. "English as a formal language", in *Formal Philosophy, Selected Papers of Richard Montague* (ed., R. Thomason), pp. 188–221, Yale University Press, New Haven, Conn. (1974).

1.12 Morris, C. W. *Foundations of the Theory of Signs*; International Encyclopedia of Unified Science, vol. 1, no. 2, University of Chicago Press, Chicago (1938).

1.13 Morris, C. W. *Signs, Language and Behavior*, Prentice-Hall, New York (1946).

1.14 Sammet, J. E. *Programming Languages, History and Fundamentals*, Prentice-Hall, Englewood Cliffs, N.J. (1969).

1.15 Scott, D. S. "Data types as lattices", *SIAM J. on Computing*, 5(3), 522–86 (1976).

1.16 Scott, D. S. and C. Strachey. "Towards a mathematical semantics for computer languages", in *Proc. of the Symposium on Computers and Automata* (ed., J. Fox), pp. 19–46, Polytechnic Institute of Brooklyn Press, New York (1971); also: technical monograph PRG–6, Programming Research Group, University of Oxford (1971).

1.17 Tarski, A. *Logic, Semantics, and Meta-mathematics*; Oxford University Press, Oxford (1956).

1.18 Wexelblat, R. L. (ed.). *Proc. of the History of Programming Languages Conference*, Los Angeles, Aug. 1978, ACM Monograph, Academic Press, New York (1980).

2 SYNTAX

The main aim of this chapter is to analyze the syntactic structure of programming languages. However, the study of syntax cannot be divorced entirely from semantics. Exercise 1.2 demonstrated that a language may have more than one syntactic description, and that one of these may be considerably more convenient for *semantic* specification.

For more complex languages, it is also necessary to *classify* syntactic phrases according to the kinds of meaning they denote, much as "parts of speech" like nouns, adjectives, and verbs are distinguished in the study of natural languages. For example, in the context

```
if g=3 then ···
```

fragment '$g=3$' has a (Boolean) value and is an ***expression***, whereas in the context

```
const g=3;
```

the same fragment gives identifier 'g' a local meaning and is a ***definition***. Other important syntactic classes are ***commands*** (such as '$g:=3$') and ***sequencers*** (such as 'goto 3'). In this chapter, we shall see how such syntactic distinctions are motivatated by semantical considerations. Detailed discussion of semantics will appear in subsequent chapters.

2.1 EXPRESSIONS

2.1.1 Literals

For now, our view of expressions is simply that they are those program phrases that are *evaluated*. The simplest form of expression is the *literal*: an

elementary expression whose value is defined by the language and cannot be changed by the programmer. Literals are also known as "constants" or "denotations". Here are some examples in PASCAL:

```
6349
2.37E-28
'A'
'STRING'
nil
[ ]
```

As always, it is important not to confuse syntactical and semantical entities. Note that syntactically *distinct* literals may have the *same* values (e.g., '4' and '04'). More significantly, a programming language need not provide literals for *all* of the values expressible in it. For example, in PASCAL the two Boolean values are not expressible by literals. (Symbols '*true*' and '*false*' are pre-defined to have these values, but are not literals because a programmer may *re*-define them if he chooses.) Many programming languages omit literals for some of the values expressible in them in order to reduce the amount of specialized notation.

2.1.2 Operators

One way that composite expressions may be formed is by the use of ***operators***, such as '+'. For example, if E_1 and E_2 are expressions, then the construct E_1+E_2 is also an expression with E_1, E_2, and the operator as its immediate constituents. Its value is determined by the operation denoted by operator '+' from the values of operands E_1 and E_2. (For now, we ignore the possibility of illegal uses of operations, such as

```
2+'A'
```

and

```
3 div 0
```

which are "trapped" before or during evaluation.) A construct of the form E_1+E_2 may itself be a sub-expression of a larger expression, and such nesting may in principle be done to arbitrary depth.

Note that the value of E_1+E_2 depends on the *values* of immediate constituents E_1 and E_2, but not on any *other* properties of them, such as whether they are syntactically elementary or composite. Furthermore, the value of E_1+E_2 is independent of the *pragmatic* difficulty of evaluating E_1 and E_2. Expressions '4', '1+3', and '1+1+1+1' are distinct, both syntactically and pragmatically, but are *semantically* equivalent because their values are the same.

2.1.3 Bracketing

In PASCAL, expression '2+3*4' is analyzed as having the phrase structure of Fig. 2.1 rather than that of Fig. 2.2. To obtain the semantic effect of the second of these, a programmer may write '(2+3)*4'. In PASCAL, the value of (E) is that of its immediate constituent, E, but the parentheses may affect the way text is parsed into phrase structures.

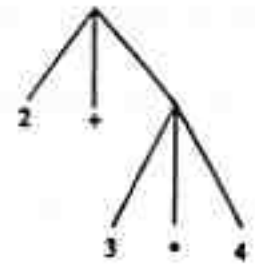


Fig. 2.1

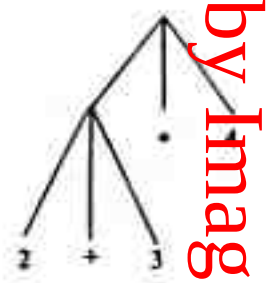


Fig. 2.2

Note that syntactic patterns such as E_1*E_2 and immediate constituent symbols such as E_1 and E_2 stand for *parsed* expressions (i.e., phrase structures), and not merely text. For example, the value of E_1*E_2 may be described as the product of the values of E_1 and E_2. But this would not be accurate if E_1, E_2, and E_1*E_2 stood for text fragments '2+2', '4', and '2+3*4', respectively.

2.1.4 Identifiers

An ***identifier***, like a literal, is a symbolic name, but its value may be defined or re-defined by the programmer, even if it is a pre-defined ("standard") identifier. Because of identifiers, the values of expressions are dependent on their computational context. We shall term this contextual information the ***state*** of the computation.

For now, we are not going to be precise about exactly what a state is or how the value of an identifier occurrence is extracted from it. The significant point is that the possible occurrence of identifiers in expressions now requires us to distinguish between

(a) the *value* of an expression ***relative to*** a particular state and
(b) the *meaning* that an expression denotes.

For example, expression '*x*+1' does not *by itself* have a value, but it does denote a meaning. Roughly, the meaning denoted by an expression yields a value as "output" when given a state as "input". Note however that none of the forms of expression we have discussed so far *change* the state, so that

when evaluating a composite such as E_1+E_2 relative to some state, all of its sub-expressions are evaluated relative to that *same* state.

Expressions are semantically equivalent when they denote the same meaning, that is to say, when they would have equal values relative to *every* possible state. For example, addition is a commutative operation, so that '$x+1$' is equivalent to '$1+x$'.

2.2 COMMANDS

A command is essentially a request for an "irreversible" *change* in the current computational state, and so we speak of ***executing*** a command relative to some state in order to obtain a new state. For example, the effect of executing ***assignment*** command '$x:=x+1$' relative to some state is to produce a new state that differs only in that the value of 'x' is greater by one. Commands are often called "statements", but this term is also applied to phrases that have distinct kinds of meaning, such as declarations and sequencers, so that we shall avoid its use.

Commands are used in programming languages to give a programmer control over storage media that permit over-writing, without requiring him to explicitly mention the states in his programs. If the only practical large-scale storage devices were "read only", there would be little reason to have commands in programming languages.

It is possible for a programming language to have no forms of command. These have been termed *applicative* languages. The sub-language of LISP known as "pure" LISP or as LISP 1.0 is one example, and the language LAMBDA mentioned in Section 1.1 is another. (Of course, *processors* for such languages may still take advantage of conventional storage devices.) Some programming theorists think that it is easier to reason about properties of programs expressed in languages that do not involve the semantic complexities of *imperative* constructs (i.e., assignments and other forms of command).

As with expressions, there are both *elementary* and *composite* forms of command. An example of an elementary form of command is the ***null*** command, which has no effect on the state. In PASCAL it is expressed by the absence of an explicit command, that is to say, by the null text string.

The assignment is an example of a composite command. The two immediate constituents of an assignment are expressions which we shall term its ***source*** and ***target***. In most languages (including PASCAL) the targets of assignments cannot be arbitrary expressions. For example,

$$x+1 := \cdots$$

is not allowed. The class of expressions that may be used as assignment targets will be termed ***l-expressions*** ("*l*" for "left" or "location"). *L*-expressions are often called "variables", but we shall avoid this term to prevent confusion with its use by mathematicians to designate non-constant identifiers (i.e., parameters), and its use by programmers to designate storage as well as *l*-expressions.

Some of the more important composite forms of command are known as ***control structures*** and may be classified as follows:

(1) *Sequential composition*. In this form of command, the effect of the construct is that obtained by executing each of the sub-commands once in a pre-determined sequence. In PASCAL, this is expressed by use of the semi-colon: if C_1 and C_2 are commands, then the effect of composite command C_1; C_2 is that of C_1 followed by that of C_2. Symbol C (possibly with subscripts or primes) will consistently be used to designate arbitrary (parsed) commands, in the same way that E has been used to stand for arbitrary (parsed) expressions.

(2) *Selection*. A ***selective*** (or "conditional") command is a construct whose effect is obtained by executing *one* of its sub-commands (or is null), as determined by the value of a sub-expression. Examples of selective control structures are the **if** and **case** forms of command in PASCAL.

(3) *Repetition*. A ***repetitive*** (or "iterative") command is a construct whose effect is obtained by repeatedly executing a sub-command, with the number of repetitions usually determined by evaluating one or more sub-expressions. Examples of repetitive control structures are the **for**, **while**, and **repeat** forms of command in PASCAL. For the present we ignore the possibility of *non-termination* of execution due to "infinite" looping in repetitive forms.

(4) *Bracketing*. Symbols '**begin**' and '**end**' in PASCAL play the same role for commands as do parentheses '(' and ')' for expressions. The effect of **begin** C **end** is simply that of C itself, but the brackets may be used to control parsing. For example, in PASCAL a command of the form

if E_1 **then if** E_2 **then** C_1 **else** C_2

would be parsed so that the **else** part matches the *second* **if**. If a programmer wanted the **else** part to match the *first* **if**, he would use brackets (and, for readability, indentation) as follows:

```
if E1 then
begin
  if E2 then C1
end
else C2
```

We have seen that several forms of command have expressions as immediate constituents. Programming languages generally also allow commands to be components of expressions in some contexts. For example, suppose that the following form of *expression* were added to PASCAL:

```
begin
  C
result E
```

(A similar construct, with syntax **begin** C ; E **end** is found in the language ALGOL W.) Its value is to be that of sub-expression E evaluated *after* execution of command C. But evaluation of this composite expression may also be expected to change the state, because of the effects of executing C. These state changes are known as the ***side effects*** of the expression evaluation.

For example, the value of

```
begin
  a := sqr(a);
  b := sqr(b)
result a+b
```

is the sum of the squares of the original values of *a* and *b*. But this evaluation *also* has the side effect of squaring *a* and *b*.

Side effects are often confusing to program readers because they are *unexpected*: the familiar expressions of conventional arithmetic and algebra do not have side effects. Furthermore, whenever a composite expression has more than one sub-expression (such as E_1+E_2), it is necessary to take into account the *order* of their evaluation because side effects of one sub-expression may affect the value yielded by another. If expression evaluation cannot change the state, the order of sub-expression evaluations is semantically insignificant (except for possible jumps and error traps). For these reasons, the use of side effects is generally regarded as poor programming practice, except in special circumstances such as during program debugging.

2.3 DEFINITIONS

2.3.1 Definitions and Commands

A simple example of a definition in PASCAL is

```
const i = -j;
```

This seems similar to *assignment*

```
i := -j
```

but the latter is a command and has an *irreversible* effect on the computational state. In contrast, the effect of a definition normally is "undone" later. Consider the composite form

```
D
begin
  C
end
```

in PASCAL, where D stands for a definition. This form of command is termed a ***block***. In block

```
const i = -j;
begin
  ... i ...
end
```

the effect of interpreting the definition is to define identifier '*i*' to denote the value of '$-j$' for the execution of the sub-command. When control leaves the block, some previously defined meaning of '*i*' will be restored.

The localized association of an *identifier* to a *value* established by a definition is termed a ***binding***, and the region of program text in which a binding is effective is known as its ***scope***.

In contrast, the effects of commands (such as '$i := -j$') need not be localized to any pre-determined part of the program. For example, in

```
i := -j;
begin
  ... i ...
end
```

effect of updating *i* will persist until execution of another assignment to it, rather than be localized to the second command. To help maintain the important conceptual distinction between definitions and assignments, the term "binding" should *not* be used for the associations set up by executing assignments.

2.3.2 Environments and Stores

The semantics of blocks and definitions are most conveniently described if the computational state is partitioned into two components: the ***environment***, which is used as a record of identifier bindings, and the ***store***, which is used as a record of the effects of assignments. Expressions, definitions, and commands are all interpreted relative to a complete state, that is to say, *both* an environment *and* a store. But the "outputs" for these three syntactic classes differ:

(a) an expression yields a value (and, if necessary to allow for side effects, a new store);
(b) a command yields a new store; and
(c) a definition yields a new environment.

For example, the meaning of composite command C_1 ; C_2 may be specified by describing the effect of executing it relative to an environment and a store. First, sub-command C_1 is executed relative to the original environment and store; this will yield a new store. Then, sub-command C_2 is executed relative to the *original* environment and the *new* store. This yields another store, which is the overall result of executing the whole composite.

Note that by separating the environment and the store it is not necessary to *explicitly* "undo" any changes to the environment made during the execution of C_1. They are implicitly undone merely by stating that C_2 is executed relative to the *original* environment. It is the responsibility of language processors to manipulate their representations of environments to have the effects specified, but the details of how this is done need not be included in the semantic description.

As a second example, consider execution of a block command

```
D
begin
  C
end
```

relative to an environment and a store. First, definition D is executed relative to the original environment and store. This yields a new environment, and sub-command C is executed relative to this environment and the store. The store yielded by executing C becomes the overall result for the block. As before, the command as a whole has no persistent effect on the environment; however, definition D does have a *local* effect on the environment used for execution of sub-command C.

Finally, consider evaluating an expression of the form

```
begin
  C
result E
```

relative to an environment and store. Command C is executed and yields a new store. Then sub-expression E is evaluated relative to the original environment and this new store. The resulting value and store are the overall results for the whole evaluation.

2.3.3 Declarations

Some forms of definition affect the store as well as the environment. For example, interpretation of ***declaration***

var *x* : *integer*;

in PASCAL has the effect of allocating an area of storage suitable for storing integers. This "new" (i.e., currently unused) storage becomes the local meaning of identifier '*x*', so that assignments to '*x*' in the scope of the declaration will update the contents of this storage area. Thus, the declaration has an effect on the store, because the storage allocated must somehow be marked as being currently "in use". Consequently, interpretation of a declaration has the effect of yielding a new store as well as a new environment. The other forms of definition in PASCAL affect *only* the environment.

Storage allocated by a **var** declaration in a PASCAL block may be *reclaimed* by a processor when execution of the block has terminated, because this storage will no longer be accessible. Such storage reclamation has no semantic effect, but it is an important pragmatic aspect of the language, because it permits a simple and efficient stack-oriented approach to storage management.

2.3.4 Type Expressions and Static Expressions

In declaration

var *x* : *integer*;

the role of identifier '*integer*' is to specify the *type* of the bound identifier, '*x*'. Type information has three purposes. Firstly, it is used to simplify implementation; for example, it is easier and more efficient for an implementation to allocate storage for integers only, rather than for arbitrary values.

The second function of type information is to allow checks of the compatibility of operations and arguments before execution; for example, in the scope of the declaration above, expression '**not** *x*' would be in error because the type of operand '*x*' is not compatible with the type expected by Boolean operator '**not**'.

Finally, type specifications are often useful during program development and can improve program readability.

In declaration

> **var** *x* : *integer*;

'*integer*' is a type *identifier*. PASCAL also allows *composite* type expressions, as in

> **var** *a* : **array**[*Boolean*]**of** *integer*;

in which type construction **array**[T_1]**of** T_2 is used, where T_1 and T_2 are sub-expressions. It is also possible in PASCAL for the programmer to bind an identifier to a type by means of a **type** definition, as in

> **type** *t* = **array**[*Boolean*]**of** *integer*;
> **var** *a* : *t*;

A syntactic class in PASCAL that is closely associated with type expressions is what we will term the ***static expressions***. These are simply expressions that are sufficiently restricted that they may be straightforwardly evaluated "statically", i.e., before program execution. Static expressions are required in certain contexts in PASCAL, such as in subrange type expressions, as case labels, and as the right-hand sides of **const** definitions.

2.4 PROCEDURE INVOCATIONS AND DEFINITIONS

2.4.1 Invocations

The syntactic form

$$I(\cdots, E_i, \cdots)$$

in PASCAL, where I is an identifier, is what will be termed a procedure ***invocation*** (or "call"). It might be an *expression*, in which case the identifier must be a **function** name, or a *command*, in which case the identifier is a procedure name. Sub-expressions E_i are the ***actual parameters*** of the invocation, and their values will be termed the ***arguments*** of the procedure.

How should we describe the meaning of an invocation? The most obvious approach is to express the effect of an invocation in terms of an execution of the program fragment that *defined* the procedure or function name elsewhere in the program. The identifier would therefore be regarded as the name of a *syntactic* entity.

This approach is conceptually unsatisfactory, however, because *distinct* procedure descriptions may have the *same* meanings, just as different numerals may denote the same number. For example, consider how many semantically equivalent array-sorting procedures may be written. Procedures are not merely devices for abbreviating program texts. Programmers are encouraged to use procedures to separate levels of abstraction in the development of large programs. Our explanation of the semantics of procedures should similarly distinguish between the abstract meaning of a procedure (*what* it does) and the concrete text that describes it (*how* to do it).

Let us recall at this point an important mathematical notion: a ***function*** (or "mapping" or "operation") is a correspondence between the elements of two given sets such that for *each* element of one set there corresponds (exactly) *one* element of the other set. If a function *f* maps *a* to *b*, then *b* is termed the ***result*** of ***applying*** *f* to argument *a*. If A is the set of arguments of a function *f*, and B is the set of its possible results, then this may be written $f: A \to B$. Functions are regarded as equal just if they have the same set of arguments, the same set of possible results, and the same result for every argument.

In conventional mathematical terminology, the sets of arguments and possible results of a function are known as its "domain" and "codomain", respectively. In this book, the term ***domain*** will designate any set that is the set of arguments or the set of possible results of a function used in describing a programming language.

Here are some examples of functions over the domain of integers:

(a) For any integer *k*, there is a *constant* function that maps every argument into *k*.
(b) The *squaring* function maps any argument *n* to n^2.
(c) The *identity* function maps any argument *n* to *n*.
(d) The ***negation*** function maps any argument *n* to $-n$.

Now, consider invocation expression

sqr(*i*+*j*)

in PASCAL. Suppose that it is being evaluated in an environment in which identifier '*sqr*' has its pre-defined (standard) meaning. If we assume that the squaring function mentioned above *is* this meaning, then the value of the invocation expression may be described as the result of applying the value of the procedure identifier to the value of the actual parameter. Note that we do *not* need to be concerned with *how* the squaring function is described or implemented. Therefore, we have described the meaning of the invocation *denotationally*, that is, in terms of the meanings of its immediate constituents, just as we did for other composite forms of expression and command.

Of course, this example is unusually simple. In general, the so-called "functions" describable in PASCAL require a more elaborate semantic model to account for side effects of invocations and dependencies on the states at definition and invocation. These issues will all be discussed later.

For the present we consider only the problem of terminology: it seems inappropriate to use the term "function" for entities that in general seem not to behave like mathematical functions. We will therefore adopt the following conventions: the term "function" will *only* refer to the usual mathematical notion (except when reference is made to a syntactic entity as a "**function** definition".) The term ***procedure*** will designate the meaning of identifier I in *all* forms of invocation I(···,E,···). If it is necessary to distinguish between kinds of procedure, they will be termed *expression* procedures, *command* procedures, and so on.

2.4.2 Procedure Definitions and Abstracts

In the previous section, we considered the semantics of invocations and saw that it was possible in a very simple case to describe the meaning of the composite form in terms of the meanings of its immediate constituents. By using a pre-defined procedure it was possible there to avoid issues relating to how procedures may be set up by the programmer. In this section we discuss procedure *definitions*, sometimes called "sub-programs".

Again, complexities of the general case (such as recursion, parameter conventions, and non-local identifiers) may be temporarily avoided by considering the following rather simple example:

```
function sqr(n : integer) : integer;
  begin
    sqr := n*n
  end;
```


We would expect this to define the *same* procedure as the standard meaning for identifier '*sqr*', that is to say, the squaring procedure.

Note that there are really *two* kinds of "definition" involved here. Firstly, there is the "definition" of a certain semantic entity, in this case the squaring procedure. Secondly, this procedure is given a name by means of a definition of identifier '*sqr*'; this is essentially similar to the forms of definition in PASCAL that have been discussed already. In some programming languages, these two aspects are expressed by distinct syntactic features and it will be helpful conceptually to treat them separately.

Consider then the following re-arrangement of the **function** definition:

```
sqr = function(n : integer) : integer;
        n*n;
```

This is not syntactically legal in PASCAL, but similar constructions appear in several other languages. If the phrase on the right-hand side of the '=' symbol is regarded as an *expression* whose value is the squaring procedure, then this construct as a whole may be regarded as an ordinary definition of the form I = E, with the usual effect of locally binding identifier I to the value of expression E. But how can

```
function(n : integer) : integer;
  n*n
```

be regarded as an *expression*? Note in particular that sub-expression '*n***n*' cannot be evaluated when the procedure is *defined* because the value of the *formal parameter*, *n*, is not yet known!

The explanation is that to define the procedure we are only interested in the meaning denoted by '*n***n*', and not its value at the time of definition. Recall that it was suggested that the meaning of an expression could be regarded as yielding a value *when* evaluated relative to a state. Therefore, the value of expression

```
function(n : integer) : integer;
  n*n
```

may be defined in terms of the meaning of '*n***n*', as follows: it is the procedure whose result for any (integer) argument is the value of '*n***n*' relative to a state in which the value of '*n*' is that argument. Of course, this procedure *is* the squaring procedure. Note that three distinct values are involved in this explanation:

(a) the squaring procedure, which is the value of the whole expression,
(b) the argument, *n*, for any application of this procedure, and
(c) the result, *n**n*, for this particular argument.

Again we see that it is possible to specify the meaning of a composite form solely in terms of the meanings of its immediate constituents.

The specification of a procedure by a syntactic description of its result for a typical argument is an example of a linguistic device known in some branches of mathematics as *abstraction*. This term is appropriate because the semantically irrelevant properties of the description are indeed "abstracted away". For example, the PASCAL definition in Fig. 2.3 is syntactically and pragmatically quite different from the definition of '*sqr*' given earlier; but the procedures they define happen to be *semantically* indistinguishable.

```
function sqr(m : integer) : integer;
  var i,s : integer;
  begin
    s := 0;
    for i := 1 to abs(m) do
      s := s+abs(m);
    sqr := s
  end;
```

Fig. 2.3

Expressions like

```
function(n : integer) : integer;
  n*n
```

will be termed *abstracts*. In PASCAL, this form of expression is *implicit* in **function** and **procedure** definitions, but similar constructs appear in other languages and notations. For example, the analogous constructs are

```
proc(int n) int: n*n
```

in ALGOL 68,

```
(LAMBDA (N) (TIMES N N))
```

in LISP, and

$$n \rightarrow n.n$$

in algebra.

It may be helpful to compare the concept of procedural abstraction to a more familiar form of abstraction. In mathematics,

$$\{n \in \mathbf{N} \mid n^2 \leq 2 \times n\}$$

is generally understood to denote the subset of elements of N whose squares are not greater than their doubles. It is an example of a *set* abstract. Its value is a set, determined by the meaning of formula '$n^2 \leq 2 \times n$', just as a procedural abstract has a procedure as its value, determined by the meaning of its body. For example, compare the set abstract above with procedural abstract

```
function(n : natnum) : Boolean;
  sqr(n) ≤ 2*n
```

which defines the characteristic function for the set.

The analog of procedure invocation for sets is set membership testing. For example, if

$$S = \{n \in \mathbf{N} \mid n^2 \leq 2 \times n\},$$

then

$$i \in S$$

would have value *true* just if *i* is an element of the set (i.e., if $i^2 \leq 2 \times i$), and *false* otherwise.

2.5 SEQUENCERS AND LABELS

2.5.1 Sequencers

Conventional computers have the property that *any* location in their fast store may be treated as the next instruction to be executed. Language processors make use of this to implement control structures and procedures. Sequencers are provided in programming languages to allow programmers to use this flexibility more directly. The only sequencers in PASCAL have the form

```
goto N
```

where N stands for a numeral.

The presence of sequencers in a programming language creates two significant semantic difficulties that control structures and procedures alone do not. The first is the problem of explaining the meaning of a construct like '**goto** 13' in terms of the meaning of its immediate constituent, numeral '13'. Evidently the *numerical* significance of the numeral is irrelevant in this context. The numeral is just being used to designate the point in the program that is to be the destination of the "jump". But what is the *semantic* counterpart of a "program point"?

The second difficulty is that, with the semantic model we have been using, an expression evaluation or command execution is expected to yield a value, or an updated store, or both. For example, we defined the effect of

$$C_1;C_2$$

to be the effect of C_2 relative to the store *after* execution of C_1. But now suppose C_1 is a command that contains a **goto**, as in

begin⋯; **goto** 13;⋯**end**; C_2

Then this description would be incorrect because the sequencer may cause a "jump" out of the whole construct and C_2 will not be executed at all. Similarly, "evaluation" of sub-expression E_1 in E_1+E_2 will not yield a value if E_1 contains an invocation of the procedure defined by

```
function escape : integer;
  begin goto 13 end;
```

which results in a jump to a context far removed from the original addition expression. The introduction of sequencers into a language has repercussions on the interpretation of almost all syntactic classes.

We are not going to attempt to solve these semantic problems at this point. But it should be noted that some programming theorists think that sequencers impair the readability and verifiability of programs, and recommend that their use be avoided. Others insist that languages without sequencers would be too inflexible. We shall return to these issues in Chapter 10.

2.5.2 Labels

In PASCAL a command may be labelled by prefixing it by a numeral and a colon, as in

```
13: x:=x+1
```


This establishes a local meaning for the numeral and is therefore similar to a *definition*.

In PASCAL, the procedure definition that encloses a labelled command is also required to contain a "declaration" of the label, as in

```
label 13;
```

But this form of "declaration" does not have any real semantic significance. It is required in PASCAL merely to simplify syntactic analysis of programs in which labels are used before the program points at which they are defined, as in

```
begin
  ⋮
  goto 13;
  ⋮
  13: x:=x+1;
  ⋮
end
```

2.6 FORMAL SYNTAX

It would be possible to specify the syntax of programming languages informally, as was done with the language of binary numerals in Chapter 1. But it is much more convenient to use a specialized formal notation. Notation that is used in describing the syntax or semantics of a language is termed a ***meta-language***. In this section, we shall discuss meta-languages for describing syntax.

2.6.1 Abstract Syntax

The syntactic structure of a language may be conveniently summarized by simply listing all of the possible forms for each of the syntactic classes. This is termed the ***abstract syntax*** of the language. The abstract syntax of PASCAL is specified in Appendix B. The first section lists the syntactic classes along with the symbols that stand for arbitrary elements of the classes. In the second section, the alternatives for each of the non-elementary classes are listed to the right of a '::=' symbol, separated by occurrences of symbol '|'.

2.6.2 Concrete Syntax

An abstract syntax tells us what syntactic structures are available in a

language, but does not specify which strings of characters are well-formed program texts, nor their phrase structures. For example, the abstract syntax of PASCAL tells us that

if E **then** C

and

if E **then** C **else** C

are possible phrase structures for commands, but does not specify whether

if *a* **then if** *b* **then** *p* **else** *q*

is a well-formed command text or, if it is, whether it is to be analyzed so that the **else** part matches the first or the second of the **then** parts. Such issues are settled by a ***concrete syntax*** (or "grammar").

Backus–Naur Formalism (BNF) is a well-known meta-language for specifying concrete syntax. An example is given in Fig. 2.4. The first rule specifies that (the textual representation of) an expression is either a term or an expression followed by an addition operator (*addop*) followed by a term. The interpretation of the other rules is similar.

⟨*expression*⟩	::=	⟨*term*⟩ | ⟨*expression*⟩ ⟨*addop*⟩ ⟨*term*⟩
⟨*term*⟩	::=	⟨*factor*⟩ | ⟨*term*⟩ ⟨*multop*⟩ ⟨*factor*⟩
⟨*factor*⟩	::=	⟨*identifier*⟩ | ⟨*literal*⟩ | (⟨*expression*⟩)
⟨*identifier*⟩	::=	*a* | *b* | *c* | ⋯ | *z*
⟨*literal*⟩	::=	0 | 1 | 2 | ⋯ | 9
⟨*addop*⟩	::=	+ | − | **or**
⟨*multop*⟩	::=	• | / | **div** | **mod** | **and**

Fig. 2.4

These rules specify the *phrase structures* for texts analyzable as expressions, terms, factors, and so on, and not merely *what* texts are recognizable. For example,

a+*b*•*c*

is recognized as an expression whose phrase structure is

⟨*expression*⟩ ⟨*addop*⟩ ⟨*term*⟩

where the term is '*b*•*c*'. Similarly,

a•*b*+*c*

is analyzed as having the phrase structure

⟨*expression*⟩ ⟨*addop*⟩ ⟨*term*⟩

where the term is '*c*'. Therefore, '*a*+*b*•*c*' is equivalent to '*a*+(*b*•*c*)', and '*a*•*b*+*c*' is equivalent to '(*a*•*b*)+*c*'. In short, the syntax specifies that multiplication operators take ***precedence*** over (or "bind more tightly than") addition operators.

Similarly,

a−*b*−*c*

is analyzed as an expression of the form

⟨*expression*⟩ ⟨*addop*⟩ ⟨*term*⟩

where the sub-expression has the same form. Therefore, '*a*−*b*−*c*' is equivalent to '(*a*−*b*)−*c*'. That is, the syntax specifies that the addition operators (and also the multiplication operators) ***associate*** to the left.

A syntactic description is termed ***ambiguous*** if, for any text, it specifies more than one phrase structure. For example, consider replacing the first rule in the syntax above by

⟨*expression*⟩ ::= ⟨*term*⟩ | ⟨*expression*⟩ ⟨*addop*⟩ ⟨*expression*⟩

This would recognize the same set of texts, but for a text like

a−*b*−*c*

there would be two possible phrase structures. Because '(*a*−*b*)−*c*' is not equivalent to '*a*−(*b*−*c*)', these two phrase structures would have distinct meanings. It is certainly undesirable to have an ambiguous syntax, but there is no *general* method possible for testing whether a syntax is ambiguous. However, it is often possible to prove that a *particular* syntax is unambiguous.

BNF is not the only notation in use for specifying formally the concrete syntax of programming languages. Appendix C is a description of the concrete syntax of PASCAL that uses ***syntax diagrams***. Their interpretation should be evident. Every path through a diagram in the direction of the arrows represents a possible analysis for that kind of text.

2.8.3 Context-sensitive Syntax

A significant limitation of notations like BNF and syntax diagrams is that they specify only the *context-free* aspects of the syntax of a language: each rule describes the phrase structures possible solely in terms of their substructures. But contextual information is necessary to specify that texts like

```
const i=32;
begin
    ⋮
  i:=i+1;
    ⋮
end
```

and

```
var i : integer;
begin
    ⋮
  not i
    ⋮
end
```

are ill-formed.

There is no widely accepted notation for specifying such context-sensitive constraints, and they are often expressed informally. Unfortunately, an informal description is often incomplete or imprecise, particularly if it is not based on a formal description. Most of the incompatibilities between and inconsistencies within PASCAL processors are attributable to incomplete and imprecise specification of its syntax. An approach to describing context-sensitive syntax formally will be discussed in Chapter 13.

EXERCISES

2.1 In PASCAL the meanings of operators such as '+' are pre-defined and unchangeable; that is, they are analogous to *literals*. Would it be possible to have operators that are analogous to *identifiers*?

2.2 Give an example of a PASCAL fragment that is semantically equivalent to the null command, yet is syntactically different from it.

2.3 Compare the effects of the following PASCAL blocks:

(a)
```
const a=1;
procedure p;
  const a=2;
  begin
    write(a)
  end {p};
begin
  p;
  write(a)
end
```

(b)
```
var a : integer;
procedure p;
  begin
    a:=2;
    write(a)
  end {p};
begin
  a:=1;
  p;
  write(a)
end
```

2.4 According to Appendix B, none of the forms of expression in PASCAL have command components. How is it possible for an expression to have side effects?

2.5 According to Appendix B, parameter specifiers, Q, have the same syntax as formal parameters, P. Why is it appropriate to distinguish these classes?

2.6 Describe a reasonable semantics for expressions of the form

```
D
begin
  C
result E
```

where D is a definition (possibly a declaration), C is a command, and E is an expression.

2.7 The *composition* of two functions f:A→B and g:B→C is the function $g \circ f$:A→C defined by the rule that a is mapped into $g(f(a))$. Show that for any h:C→D,

$$h \circ (g \circ f) = (h \circ g) \circ f$$

2.8 Does evaluation of an abstract (as described in Section 2.4.2) have side effects?

2.9 A label in PASCAL is analogous to an identifier in that its meaning as a label is definable by the programmer. Is it conceivable for a language to have a *pre-defined* label identifier? What about a *literal* label? Suggest possible applications of these concepts if they are feasible.

2.10 The concrete syntax of COBOL has been described using the following notational conventions:

(a) The vertical bar of BNF is replaced by alternatives arranged vertically and enclosed in braces:

$$\left\{\begin{array}{l}\text{alternative}_1\\ \text{alternative}_2\\ \vdots\\ \text{alternative}_n\end{array}\right\}$$

(b) Optional constructs or optional alternatives are indicated by enclosing them in square brackets:

$$\left[\begin{matrix}\text{option}_1\\ \text{option}_2\\ \vdots\\ \text{option}_n\end{matrix}\right]$$

(c) Repetition of a construct is indicated by appending an ellipsis "...".
(d) Key words that are not underlined have no effect and may be omitted. (They are known as "noise words".)

For example, the syntax of the ADD statement in COBOL may be expressed as follows:

ADD $\left[\begin{matrix}\langle identifier\rangle\\ \langle number\rangle\end{matrix}\right]$ $\left[\begin{matrix}, \langle identifier\rangle\\ , \langle number\rangle\end{matrix}\right]$ ...

TO ⟨*identifier*⟩ [ROUNDED] $\left[, \langle identifier\rangle \text{ [ROUNDED]}\right]$ ...

[; ON SIZE ERROR ⟨*statement*⟩]

Express this with a syntax diagram.

2.11 Part of the concrete syntax given in the 1960 Report on ALGOL 60 is effectively as follows:

⟨*statement list*⟩
::= ⟨*statement*⟩
| ⟨*statement list*⟩ ; ⟨*statement*⟩

⟨*statement*⟩
::= ⟨*assignment*⟩
| **if** ⟨*expression*⟩ **then** ⟨*statement*⟩
| **if** ⟨*expression*⟩ **then** ⟨*statement*⟩ **else** ⟨*statement*⟩
| **begin** ⟨*statement list*⟩ **end**

In the 1963 Revised Report on ALGOL 60, this was changed to

⟨*statement list*⟩
::= ⟨*statement*⟩
| ⟨*statement list*⟩ ; ⟨*statement*⟩

⟨*statement*⟩
::= ⟨*unconditional statement*⟩
| **if** ⟨*expression*⟩ **then** ⟨*unconditional statement*⟩
| **if** ⟨*expression*⟩ **then** ⟨*unconditional statement*⟩
else ⟨*statement*⟩

⟨*unconditional statement*⟩
::= ⟨*assignment*⟩
| **begin** ⟨*statement list*⟩ **end**

(a) Give an example that shows why the original syntax was unsuitable and why the revised syntax is an improvement.
(b) What would be the advantage of the following syntax:

⟨*statement list*⟩
::= ⟨*statement*⟩
| ⟨*statement list*⟩ ; ⟨*statement*⟩

⟨*statement*⟩
::= ⟨*balanced statement*⟩
| ⟨*unbalanced statement*⟩

⟨*balanced statement*⟩
::= ⟨*assignment*⟩
| **begin** ⟨*statement list*⟩ **end**
| **if** ⟨*expression*⟩ **then** ⟨*balanced statement*⟩
else ⟨*balanced statement*⟩

⟨*unbalanced statement*⟩
::= **if** ⟨*expression*⟩ **then** ⟨*statement*⟩
| **if** ⟨*expression*⟩ **then** ⟨*balanced statement*⟩
else ⟨*unbalanced statement*⟩

2.12 In the language APL, all operators have the same precedence and associate to the right. Give a concrete syntax with these properties for expressions whose abstract syntax is

B literals
I identifiers
O operators
E expressions

E ::= B | I | OE | EOE | (E)

PROJECT

Work out an abstract syntax for some other programming language. To facilitate comparison with PASCAL, try to use the same syntactic classes and symbols as in Appendix B.

BIBLIOGRAPHIC NOTES

Programming languages without commands and their use have been described by McCarthy [2.12], Landin [2.11], Burge [2.5], Ashcroft and Wadge [2.1], Backus [2.3], Henderson [2.7], Warren [2.18], Morris et al. [2.15], and others. The *environment* concept was first used for programming-language description by Landin [2.9]. The separation of the "state" into environment and store components was described by Scott and Strachey [2.17]. The term "sequencer" was introduced by Milne and Strachey [2.14]. Arguments pro and con sequencers may be found in Landin [2.10], Dijkstra [2.6], and Knuth [2.8]. The concept of abstract syntax was introduced by Landin [2.9] and McCarthy [2.13]. The notation used in Appendix B is essentially that of Scott and Strachey [2.17]. BNF was introduced by Backus [2.3] and popularized in the reports on ALGOL 60 [2.16]. Much more information on (concrete) syntax and parsing may be found in a book by Backhouse [2.2].

2.1 Ashcroft, E. A., and W. W. Wadge: "LUCID, a non-procedural language with iteration", *Comm. ACM*, **20** (7), 519–26 (1977).

2.2 Backhouse, R. C. *Syntax of Programming Languages, Theory and Practice*, Prentice-Hall International, London (1979).

2.3 Backus, J. "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference", *Proc. Int. Conf. Information Processing*, pp. 125-32, UNESCO, Paris (1959).

2.4 Backus, J. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", *Comm. ACM*, **21** (8), 613–41 (1978).

2.5 Burge, W. H. *Recursive Programming Techniques*, Addison-Wesley, Reading, Mass. (1975).

2.6 Dijkstra, E. W. "Go to statement considered harmful", *Comm. ACM*, **11** (3), 147–8 (1968).

2.7 Henderson, P. *Functional Programming, Application and Implementation*, Prentice-Hall International, London (1980).

2.8 Knuth, D. E. "Structured programming with goto statements", *Comp. Surveys*, **6** (4), 261–301 (1974).

2.9 Landin, P. J. "The mechanical evaluation of expressions", *Comp. J.*, **6** (4), 308–20 (1963).

2.10 Landin, P. J. *Getting Rid of Labels*, Univac Systems Programming Research report, New York (1965).

2.11 Landin, P. J. "The next 700 programming languages", *Comm. ACM*, **9** (3), 157–64 (1966).

2.12 McCarthy, J. "Recursive functions of symbolic expressions and their computation by machine, pt. 1", *Comm. ACM*, **3** (4), 184–95 (1960).

2.13 McCarthy, J. "Towards a mathematical science of computation", in *Information Processing 1962*, Proc. IFIP Congress, 1962, pp. 21–28, North-Holland, Amsterdam (1963).

2.14 Milne, R. E. and C. Strachey. *A Theory of Programming Language Semantics*, Chapman and Hall, London, and Wiley, New York (1976).

2.15 Morris, J. H., E. Schmidt, and P. Wadler. "Experience with an applicative string processing language", *Conf. Record of the 7th ACM Symposium on Principles of Programming Languages*, pp. 32–46 (1980).

2.16 Naur, P. (ed.). "Revised report on the algorithmic language ALGOL 60", *Comm. ACM*, **6** (1), 1–20 (1963); also in *Comp. J.*, **5**, 349–67 (1963), and *Numerische Mathematik*, **4**, 420–52 (1963).

2.17 Scott, D. S. and C. Strachey. "Towards a mathematical semantics for computer languages", in *Proc. of the Symposium on Computers and Automata* (ed., J. Fox), pp. 19–46, Polytechnic Institute of Brooklyn Press, New York (1971); also: technical monograph PRG-6, Programming Research Group, University of Oxford (1971).

2.18 Warren, D. H. D. "Logic programming and compiler writing", *Software Practice and Experience*, **10** (2), 97–125 (1980).

3 DATA

As mentioned in Section 2.4.1, argument and result sets of functions used in describing the semantics of a programming language are termed *domains* (or "data types"). For example, the semantics of binary numerals given in Section 1.3 specifies a mapping having (parsed) binary numerals as its domain of arguments and natural numbers as its domain of possible results.

Every domain has associated with it certain "essential" operations. We shall describe these as the operations with which the domain is *equipped*. For example, the domain of binary numerals is equipped with two "constant" operations whose results are the elementary numerals, '0' and '1', and two operations that construct composite numerals of the forms N0 and N1 from arguments N. Similarly, the domain of natural numbers is equipped with a "constant" operation that produces the number zero, and an operation that constructs the successor of any number. Additional operations on numbers (such as addition and multiplication) may be defined using these basic operations.

The domains that are needed to describe programming languages can be much more complex than the sets of numerals and numbers. Indeed, the diversity and complexity of elementary and structured data in programming languages seem overwhelming at first. PASCAL alone has numbers, characters, Boolean values, enumerations, pointers, arrays, records, sets, files, procedures, and labels. Other languages have S-expressions and lists (LISP), multi-dimensional arrays (APL), strings and patterns (SNOBOL4), and so on. Furthermore, sets of semantic constructs such as environments and stores, and syntactic classes such as expressions, commands, and definitions must all be regarded as domains as well.

Yet we shall see that a remarkably small number of general domain operations will allow us to *construct* domains that model all of these features of programming languages. These domain constructions are described in Section 3.1. Section 3.2 presents several examples of the use of these domain constructions in modelling the "data types" of three well-known programming languages. Certain pragmatic and theoretical limitations on domain construction will be considered in Section 3.3.

3.1 DOMAIN CONSTRUCTIONS

3.1.1 Products of Domains

Consider type expression

```
record
  i : integer;
  c : char
end
```

in PASCAL. It describes a domain that consists of all ordered pairs whose first components are integers and whose second components are characters. In general, for *any* sets **A** and **B**, the set of all ordered pairs, (a,b), with $a \in \mathbf{A}$ and $b \in \mathbf{B}$, is termed the (Cartesian) ***product*** of **A** and **B**, written $\mathbf{A} \times \mathbf{B}$. Note that if **A** is a set with na elements and **B** is a set with nb elements, set $\mathbf{A} \times \mathbf{B}$ contains $na \times nb$ elements.

Domain $\mathbf{A} \times \mathbf{B}$ is equipped with two operations that select the **A** and **B** components, respectively, of arguments in $\mathbf{A} \times \mathbf{B}$. These are termed the ***projection*** functions for the product. In PASCAL the projection operations for record types are expressed by ***field selection*** *l*-expressions of the form

L.I

By using ordered n-tuples, the product construction may be generalized to allow any number n of "factors", as in $\mathbf{A}_1 \times \mathbf{A}_2 \times \cdots \times \mathbf{A}_n$. For any domain **D**, we shall write $\mathbf{D}^n$ for

$\mathbf{D} \times \mathbf{D} \times \cdots \times \mathbf{D}$ (n factors)

i.e., for the domain of all ordered n-tuples of elements of **D**. For $n=0$, we adopt the convention that $\mathbf{D}^0 = \{null\}$, that is to say, the singleton set whose only element is *null*.

3.1.2 Sums of Domains

Consider type expression

```
record case tag : Boolean of
  true : (i : integer);
  false : (c : char)
end
```

in PASCAL. Each of the elements in the domain it describes is *either* an integer *or* a character, together with a Boolean component to differentiate

the two possibilities. In general, for *any* sets **A** and **B**, their ***sum*** (or "disjoint union", or "co-product"), written $\mathbf{A}+\mathbf{B}$, is defined to be the set of ordered pairs $(true, a)$ for all $a \in \mathbf{A}$, and $(false, b)$ for all $b \in \mathbf{B}$; that is,

$$\mathbf{A}+\mathbf{B} = \{(true, a) \mid a \in \mathbf{A}\} \cup \{(false, b) \mid b \in \mathbf{B}\}$$

If sets **A** and **B** contain na and nb elements, respectively, then set $\mathbf{A}+\mathbf{B}$ has $na+nb$ elements, even if **A** and **B** have elements in common.

Domain $\mathbf{A}+\mathbf{B}$ is equipped with two operations that construct elements of $\mathbf{A}+\mathbf{B}$ from arguments in **A** and **B**, respectively. These are termed the ***injection*** functions for the sum. By using n "tags", the sum construction may be generalized to allow any number n of terms, as in $\mathbf{A}_1+\mathbf{A}_2+\cdots+\mathbf{A}_n$.

3.1.3 Function Domains

Consider type expression

array[*char*]**of** *integer*

in PASCAL. Each of the elements in the domain it describes may be regarded as being a *function* mapping characters into integers, because each array determines a *unique* integer "component" for *every* character "subscript". In general, if **A** and **B** are *any* sets, then $\mathbf{A} \rightarrow \mathbf{B}$ is defined to be the set of all functions whose set of arguments is **A** and whose set of possible results is **B**. If sets **A** and **B** contain na and nb elements, respectively, then set $\mathbf{A} \rightarrow \mathbf{B}$ contains nb^{na} elements, because there is a choice among nb possible results for each of the na arguments.

Domain $\mathbf{A} \rightarrow \mathbf{B}$ is equipped with the operation of *application*, which, for any f: $\mathbf{A} \rightarrow \mathbf{B}$ and $a \in \mathbf{A}$, determines the result of applying f to a. In PASCAL, the application operation for array types is expressed by ***subscripting*** *l*-expressions of the form

L[E]

3.1.4 Recursive Definitions of Domains

The fourth (and last) domain construction principle that we need will allow us to construct *infinite* domains. Consider the PASCAL-like definition

```
type string = record case isnull : Boolean of
                true : ( );
                false : (first : char; rest : string)
              end;
```

This is *not* legal in PASCAL (for pragmatic reasons we shall discuss in Section 3.3.1), but the intent should be clear. Each of the elements of the domain it describes is either null, or is not null and consists of a '*first*' component, which is a character, and a '*rest*' component which is a string.

The mathematical counterpart of this **type** definition is the following *equation*:

$$\mathbf{Q} = \{null\} + (\mathbf{H} \times \mathbf{Q}) \tag{3.1}$$

where **H** is the domain of characters. Consider now the set whose elements are

$$(true, null)$$

and

$$(false, (h, (true, null))) \qquad \text{for all } h \in \mathbf{H},$$

and

$$(false, (h_1, (false, (h_2, (true, null))))) \qquad \text{for all } h_1, h_2 \in \mathbf{H},$$

and so on. It may be verified that if this set is termed **Q**, it is a solution to equation (3.1); that is, every element of **Q** is an element of $\{null\} + (\mathbf{H} \times \mathbf{Q})$, and every element of $\{null\} + (\mathbf{H} \times \mathbf{Q})$ is an element of **Q**. In fact, this set is the *smallest* solution of equation (3.1).

In general, we shall regard a domain equation of the form

$$\mathbf{D} = \cdots \mathbf{D} \cdots$$

as *defining* **D** to be the smallest solution of the equation. The construction is termed ***recursive*** because the name of the domain being defined "recurs" on the right-hand side of its definition.

For any domain **D** and $d_1, d_2, \cdots, d_n \in D$, it will be convenient to use the notation

$$\langle d_1, d_2, \cdots, d_n \rangle$$

to stand for

$$(false, (d_1, (false, (d_2, \cdots (false, (d_n, (true, null))) \cdots))))$$

for $n \geq 0$, and regard the solution of

$$\mathbf{S} = \{null\} + (\mathbf{D} \times \mathbf{S})$$

as being the domain of all such finite ***sequences*** of elements of **D**. We shall use $\mathbf{D}^*$ to designate this domain.

If $s = \langle d_1, d_2, \cdots, d_n \rangle$ for $n > 0$, then

$$first(s) = d_1,$$
$$rest(s) = \langle d_2, d_3, \cdots, d_n \rangle,$$

and

$$last(s) = d_n.$$

The important operation of sequence ***concatenation*** will be designated '$\frown$':

$$\langle d_1, d_2, \cdots, d_n \rangle \frown \langle d_{n+1}, d_{n+2}, \cdots, d_{n+m} \rangle = \langle d_1, d_2, \cdots, d_n, d_{n+1}, \cdots, d_{n+m} \rangle.$$

Concatenation of sequences is an ***associative*** operation; that is, $s_1 \frown (s_2 \frown s_3) = (s_1 \frown s_2) \frown s_3$, for all sequences s_1, s_2, and s_3.

As another example of recursive domain definition, consider the equation

$$\mathbf{N} = \{zero\} + \mathbf{N} \tag{3.2}$$

It may be verified that its smallest solution is the set whose elements are

$$(true, zero)$$
$$(false, (true, zero))$$
$$(false, (false, (true, zero)))$$
$$(false, (false, (false, (true, zero))))$$
$$\vdots$$

so that if we let 0 designate $(true, zero)$ and define $successor(i)$ *to be* $(false, i)$, then it is clear that equation (3.2) may be regarded as a definition of the domain of natural numbers (i.e., 0 and its successors).

Further examples of recursive domain definition will be given in Section 3.2 and the exercises. The question of whether solutions to such domain equations always exist will be taken up in Section 3.3.2.

*3.1.5 Approximation and Limit Domains

The infinite domain that is the smallest solution to a recursive domain definition is the *limit* of a sequence of domains that "approximate" it. For example, consider again equation (3.2):

*Starred sections assume a higher level of mathematical maturity and may be skipped without loss of continuity.

$$\mathbf{N} = \{zero\} + \mathbf{N}$$

Now, define a sequence of domains $\mathbf{N}_i$, for $i \geq 0$, as follows:

$$\mathbf{N}_0 = \{\ \} \text{ (i.e., the empty set)}$$

$$\mathbf{N}_{i+1} = \{zero\} + \mathbf{N}_i$$

Note that each domain in the sequence (except the initial one) is defined by using the equation, but with the recursive occurrence of the domain name replaced by the preceding domain in the sequence, so that there is no longer any circularity. Therefore,

$$\begin{aligned}\mathbf{N}_1 &= \{zero\} + \mathbf{N}_0 \\ &= \{(true, zero)\} \\ &= \{0\}\end{aligned}$$

$$\begin{aligned}\mathbf{N}_2 &= \{zero\} + \mathbf{N}_1 \\ &= \{(true, zero),\ (false,\ (true, zero))\} \\ &= \{0,\ successor(0)\}\end{aligned}$$

$$\begin{aligned}\mathbf{N}_3 &= \{zero\} + \mathbf{N}_2 \\ &= \{(true, zero),\ (false, (true, zero)),\ (false, (false, (true, zero)))\} \\ &= \{0,\ successor(0),\ successor(successor(0))\}\end{aligned}$$

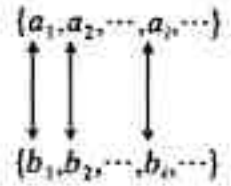
and so on.

The elements of each $\mathbf{N}_i$ are 0 and its successors up to, but not including, the ith. Domains $\mathbf{N}_i$ may be regarded as finite *approximations* to (i.e., subsets of) the desired infinite solution of the equation, so that if we take the *limit* of this sequence to be the union of all of these approximating domains, we obtain the set of all the natural numbers, the desired solution of the equation.

It may be verified that the same technique works with equation (3.1). In subsequent chapters, we shall see several other examples of the use of approximations and limits to construct solutions to recursive definitions, regarded as equations.

*3.1.6 Domain Isomorphism

It is often useful to know that two domains, though not equal as sets, are nonetheless "effectively equivalent" in the sense that they may be interchanged in semantic descriptions. This may be defined rigorously as follows: two sets, **A** and **B**, are *isomorphic*, written $\mathbf{A} \cong \mathbf{B}$, if there exists a one-to-one correspondence between them:



More formally, $\mathbf{A} \cong \mathbf{B}$ just if there are functions $f: \mathbf{A} \to \mathbf{B}$ and $f^{-1}: \mathbf{B} \to \mathbf{A}$ such that

$$f(f^{-1}(b)) = b \text{ for all } b \in \mathbf{B}$$

and

$$f^{-1}(f(a)) = a \text{ for all } a \in \mathbf{A}$$

It may be verified that $\cong$ is an equivalence relation; that is,

(a) $\mathbf{A} = \mathbf{B}$ implies $\mathbf{A} \cong \mathbf{B}$,
(b) $\mathbf{A} \cong \mathbf{B}$ implies $\mathbf{B} \cong \mathbf{A}$, and
(c) $\mathbf{A} \cong \mathbf{B}$ and $\mathbf{B} \cong \mathbf{C}$ imply $\mathbf{A} \cong \mathbf{C}$.

Here are some examples of set isomorphisms involving the product, sum, and function domain constructions:

$$\mathbf{A} \times \mathbf{B} \cong \mathbf{B} \times \mathbf{A}$$
$$(\mathbf{A} \times \mathbf{B}) \times \mathbf{C} \cong \mathbf{A} \times (\mathbf{B} \times \mathbf{C}) \cong \mathbf{A} \times \mathbf{B} \times \mathbf{C}$$
$$\mathbf{A} + \mathbf{B} \cong \mathbf{B} + \mathbf{A}$$
$$(\mathbf{A} + \mathbf{B}) + \mathbf{C} \cong \mathbf{A} + (\mathbf{B} + \mathbf{C}) \cong \mathbf{A} + \mathbf{B} + \mathbf{C}$$
$$(\mathbf{A} \times \mathbf{B}) + (\mathbf{A} \times \mathbf{C}) \cong \mathbf{A} \times (\mathbf{B} + \mathbf{C})$$
$$(\mathbf{A} \times \mathbf{B}) \to \mathbf{C} \cong \mathbf{A} \to (\mathbf{B} \to \mathbf{C})$$
$$(\mathbf{A} + \mathbf{B}) \to \mathbf{C} \cong (\mathbf{A} \to \mathbf{C}) \times (\mathbf{B} \to \mathbf{C})$$

These are valid for all sets **A**, **B**, and **C**. Furthermore, if $\mathbf{0} = \{\ \}$ and **1** is any singleton set, then

$$\mathbf{A} \times \mathbf{1} \cong \mathbf{A}, \qquad \mathbf{A} \times \mathbf{0} \cong \mathbf{0}$$
$$\mathbf{A} + \mathbf{0} \cong \mathbf{A}$$
$$\mathbf{A} \to \mathbf{1} \cong \mathbf{1}, \qquad \mathbf{1} \to \mathbf{B} \cong \mathbf{B}$$

Domain constructions $\times$, $+$, and $\to$ have the following substitution properties with respect to isomorphism: if $\mathbf{A} \cong \mathbf{A}'$ and $\mathbf{B} \cong \mathbf{B}'$, then

$$\mathbf{A} \times \mathbf{B} \cong \mathbf{A}' \times \mathbf{B}'$$
$$\mathbf{A} + \mathbf{B} \cong \mathbf{A}' + \mathbf{B}'$$
$$\mathbf{A} \to \mathbf{B} \cong \mathbf{A}' \to \mathbf{B}'$$

Note that many other operations (such as union and intersection) do *not* have these properties.

3.2 THREE CASE STUDIES OF DOMAIN CONSTRUCTION

3.2.1 S-expressions and Lists in LISP

Values in LISP are termed *S-expressions*, which is a contraction of "symbolic expressions". An S-expression may be either an ***atom***, which is written as a symbol, such as

A
APPLE
PART2

or a ***pair***, written in the form

$(S_1 . S_2)$

where S_1 and S_2 stand for arbitrary S-expressions. Some examples of pairs are

(A . B)
(A . (B1 . B2))
((U . V) . (X . (Y . Z)))

The domain of S-expressions may therefore be defined as the solution of the equation

$$\mathbf{S}=\mathbf{A}+(\mathbf{S}\times\mathbf{S})$$

where **A** is the domain of atoms.

An important subset of the S-expressions consists of what are known as ***lists*** and satisfy the following constraints:

(a) An atom is a list just if it is the atom 'NIL'.
(b) A pair $(S_1 . S_2)$ is a list just if S_2 is a list.

That is, the domain of lists is defined by

$$\mathbf{L}=\{\text{NIL}\}+(\mathbf{S}\times\mathbf{L})$$

for which the solution is $\mathbf{S}^*$ when atom 'NIL' is regarded as the null list.

Therefore, lists in LISP are sequences whose components are S-expressions. The following S-expressions

NIL
(APPLE . NIL)
(A . (B . (C . NIL)))
((APPLE . A2) . NIL)

are all lists.

Lists are more conveniently expressed in "list notation":

$(S_1 \quad S_2 \quad \cdots \quad S_n)$

for $n \geq 0$ is an abbreviation of

$(S_1 . (S_2 . (\cdots (S_n . \text{NIL})\cdots)))$

For example,

(A B C) abbreviates (A . (B . (C . NIL)))
(A) abbreviates (A . NIL)
and ( ) abbreviates NIL

Note the difference between the *pair* '(A . B)', which is not a list, and the *two-component list* '(A B)' which may also be written '(A . (B . NIL))'. Note also that 'A' and '(A)' are not equivalent, because the latter abbreviates the pair '(A . NIL)'.

S-expressions are the *values* manipulated by LISP programs; but LISP *programs* are also S-expressions. For example, the notation for literals in LISP is

(QUOTE S)

where S is an S-expression. Syntactically, this is just a two-component list; the first component is atom 'QUOTE' and the second is S-expression S. Semantically, its *value* (relative to any state) is S-expression S. For example, the value of

(QUOTE APPLE)

is atom 'APPLE', and the value of

(QUOTE (A . B))

is pair '(A . B)'. In LISP an unquoted atom used as an expression is an *identifier* (unless it is 'NIL' or 'T', which, for convenience , are exceptions to this rule, and always denote themselves.)

An expression of the form

$$(E_0\ E_1\ E_2 \cdots E_n)$$

for $n \geq 0$, is an *invocation*. The value of E_0 is the procedure, and the values of E_1 to E_n are its arguments.

LISP provides five primitive operations for constructing, selecting, and testing S-expression values. Procedure 'CONS' constructs a pair from its two arguments. For example, the value of

```
(CONS  (QUOTE  A) (QUOTE  B))
```

is '(A . B)'. of course, the actual parameters of an invocation need not be literals. For example, the value of

```
(CONS  (QUOTE  A)  (CONS  (QUOTE B)  NIL))
```

is '(A B)'.

Procedures 'CAR' and 'CDR' (rhymes with "rudder") require a pair $(S_1 . S_2)$ as an argument and return components S_1 and S_2, respectively. (The names of these procedures come from the machine instructions used to implement them in the first implementaion of LISP.) For example, the values of

```
(CAR (QUOTE (A . B)))
```

and

```
(CDR (QUOTE (A . B)))
```

are 'A' and 'B', respectively.

It is important to understand the results of procedures CONS, CAR, and CDR on list arguments. If the value of E_2 is a list, then the value of

```
(CONS  E1  E2)
```

is that list with the value of E_1 prefixed onto it as its new first component. Therefore, the value of

```
(CONS  (QUOTE  A)  (QUOTE  (B C D)))
```

is list '(A B C D)'.

The result of applying CAR and CDR to a list argument is the *first* component of the list and the *rest* of the list, respectively. For example, the values of

```
(CAR  (QUOTE  (A  B  C  D)))
```

and

```
(CDR  (QUOTE  (A  B  C  D)))
```

are 'A' and '(B C D)', respectively. Note also that the values of

```
(CONS  (QUOTE  A)  NIL)
(CAR  (QUOTE  (A)))
```

and

```
(CDR  (QUOTE  (A)))
```

are '(A)', 'A', and 'NIL', respectively.

The remaining two primitive procedures in LISP are *predicates*, which for LISP means that their result is one of atoms 'NIL' (representing *false*) and 'T' (representing *true*). If the values of E_1 and E_2 are atoms, then the value of

```
(EQ  E1  E2)
```

is 'T' if both arguments are the *same* atom, and 'NIL' otherwise.

The value of

```
(ATOM  E)
```

is 'T' if the value of E is an atom, and 'NIL' if it is a pair.

It is surprisingly easy to program very complex manipulations of hierarchically structured symbolic data with just these five primitive operations on the domain of S-expressions and some additional mechanisms for selective evaluation, procedural abstraction, and recursive definition.

3.2.2 Arrays in APL

All the values in APL are termed ***arrays***. The simplest kind of arrays are termed ***scalars*** and are said to have ***rank*** 0. A scalar is either a number or a character. Here are some examples of scalar-valued literals:

Numbers	*Characters*
8	'A'
2.087	' '

If **R** and **H** are the domains of numbers and character values, respectively, then

$$\mathbf{R}+\mathbf{H}$$

is the domain of all arrays of rank 0.

An array of rank 1 is known as a ***vector*** and is a homogeneous n-tuple of numbers or characters, for $n=0, 1, 2, \cdots$. Some examples of vector-valued literals in APL are given in Table 3.1. Note that there are no literals for numeric vectors of length 0 or 1, or character vectors of length 1; such values are expressible in other ways.

n	Vector of numbers	Vector of characters
0		''
1		
2	3 2.087	'TO'
3	1 2 3	'CAT'

Table 3.1 Vector literals in APL

The domain of all arrays of rank 1 is

$$\sum_n [\mathbf{R}^n+\mathbf{H}^n]$$

where, in general,

$$\sum_i \mathbf{D}_i = \mathbf{D}_0+\mathbf{D}_1+\mathbf{D}_2+\cdots$$

Note that there are *two* vectors with zero components, one from $\mathbf{R}^0$, and one from $\mathbf{H}^0$.

Arrays of higher rank may be obtained in the same manner. A ***matrix*** (array of rank 2) is a "table" of components, consisting of $n_1 \geq 0$ rows, each of length $n_2 \geq 0$ with all $n_1 \times n_2$ components being either numbers or characters. The domain of all arrays of rank 2 is then

$$\sum_{n_1, n_2} [\mathbf{R}^{n_1 \times n_2}+\mathbf{H}^{n_1 \times n_2}]$$

Note that for any n, there are *two* $n \times 0$ matrices, and these are distinguished from the $n' \times 0$ matrices for any n' different from n, and from all the $0 \times m$ matrices, though all have zero components.

In general, an array of rank r is a homogeneous r-dimensional hyperrectangle of numbers or of characters:

$$\sum_{n_1, \cdots, n_r} [\mathbf{R}^{n_1 \times n_2 \times \cdots \times n_r} + \mathbf{H}^{n_1 \times n_2 \times \cdots \times n_r}]$$

The vector whose components are the r numbers $n_1, n_2, \cdots, n_r$ is known as the ***shape*** of such an array. Therefore, the shape of the shape of a value is a vector whose single component is the rank of that value.

The domain of *all* APL arrays is then the sum of the domains of arrays of rank r, for $r=0,1,2,\ldots$:

$$\sum_r \sum_{n_1, \cdots, n_r} [\mathbf{R}^{n_1 \times n_2 \times \cdots \times n_r} + \mathbf{H}^{n_1 \times n_2 \times \cdots \times n_r}]$$

Note that the zero-component vectors are distinguished from the zero-component matrices, and similarly for higher rank arrays. Also, scalars are distinguished from single-component vectors, which are in turn distinguished from single-component matrices and arrays of higher rank.

APL provides many operations for construction, selection, and manipulation of arrays. By using operations on entire arrays it is possible to describe many complex computations without using explicit iterations and subscripting.

3.2.3 Strings and Patterns in SNOBOL4

The language SNOBOL4 is used primarily for describing computations on character strings. If **H** is the domain of character values, the domain of strings is $\mathbf{Q}=\mathbf{H}^*$. Examples of literal strings in SNOBOL4 are

```
''
'A'
' '
'CAT'
```

If a string is an n-tuple of characters, the characters are numbered from 0 to $n-1$, inclusive, that is to say, zero-origin addressing.

Pattern matching is the operation used in SNOBOL4 to *analyze* strings. It can be a very complex operation, possibly involving procedural abstraction, side effects, and jumps. Only a subset of the pattern-matching facilities will be discussed here. This will permit a relatively simple model for patterns.

The most basic kind of pattern value is simply a string. A pattern string of length n is said to *match a subject string from character position i up to* (but

not including) *position* $i+n$ if characters i to $i+n-1$, inclusive, of the subject are equal to the successive characters of the pattern string. The match ***fails*** if the subject is too short or if for some $j<n$ the jth character of the pattern string is not equal to character $i+j$ of the subject.

For example, the pattern string 'GRAM' matches the subject 'PROGRAMMER' from position 3 up to position 7, but fails to match from any other position. Note that the *null* pattern string always matches successfully, and that a null subject string is matched only by the null pattern.

The pattern operation of ***alternation*** produces patterns that can match a subject string in more than one way. If E_1 and E_2 are expressions whose values are the patterns p_1 and p_2, respectively, then expression

$$E_1 \mid E_2$$

has as its value a pattern that matches a subject from some position in all the ways that p_1 or p_2 match from that position. For example, the value of

'EA' | 'E'

is a pattern that matches subject 'BEAD' from position 1 up to positions 3 and 2.

The second important operation on patterns is ***concatenation***, expressed in SNOBOL4 by juxtaposition of pattern-valued expressions separated by at least one blank:

$$E_1 \quad E_2$$

The concatenation of patterns p_1 and p_2 is a pattern that matches a subject from character position i in all the ways that p_1 can match from position i up to position $i+n$ *and* p_2 can match from $i+n$. For example, the value of

'EA' 'D'

matches the subject 'BEAD' from positions 1 up to 4, because 'EA' matches from positions 1 up to 3, and 'D' matches from positions 3 up to 4.

As a more complex example, consider the pattern expressed by

('BE' | 'B') ('ET' | 'AD')

This will match any of the subject strings

'BEET'
'BEAD'
'BET'
'BAD'

from character position 0.

Many other patterns and pattern operations are available in SNOBOL4. For example, if the value of identifier 'N' is the number n, the value of 'POS(N)' is a pattern that matches any subject string in the same way as the null pattern, but only at character position n of the subject.

A convenient way to model patterns mathematically is to regard them as *functions* applicable to arguments (q,i), where q is a subject string and i is the position at which a match is to be attempted. The result of applying a pattern to (q,i) is the *sequence* $\langle j_1,j_2,\cdots,j_m\rangle$ of all positions j_k such that the pattern matches from position i up to (but not including) character j_k of the subject. Positions j_k are in the order that they would be discovered by the pattern-matching algorithm in SNOBOL4 processors. The pattern-match operation itself (not discussed here) only makes use of j_1 (i.e., the first component of such result sequences), but all possible match positions are required to describe the pattern operations of concatenation and alternation. If a match fails, the result will be the null sequence $\langle\rangle$ (i.e., with $m=0$). The domain **P** of patterns is therefore

$$\mathbf{P} = (\mathbf{Q}\times\mathbf{N})\rightarrow\mathbf{N}^*$$

For example, if the pattern derived from the null string is applied to (q,i), where q is a string and i is a position in q, the result is the one-component sequence $\langle i\rangle$. The result of applying the pattern derived from a non-null string such as 'GRAM' to (q,i) is either the one-component sequence $\langle i+4\rangle$ if the substring from positions i to $i+3$, inclusive, of q is equal to 'GRAM', or the null sequence if the match fails.

The operations of alternation (|) and concatenation (^) of patterns may then be formally defined using concatenation of sequences as follows:

$$(p_1 \mid p_2)\,(q,i) = p_1(q,i)^\frown p_2(q,i)$$

and

$$(p_1{}^\frown p_2)\,(q,i) = p_2(q,j_1)^\frown p_2(q,j_2)^\frown\cdots{}^\frown p_2(q,j_m)$$

where

$$\langle j_1,j_2,\cdots,j_m\rangle = p_1(q,i)$$

In the definition of pattern concatenation, if $p_1(q,i)$ is the null sequence ( ), then so is $(p_1 \frown p_2)\ (q,i)$.

For example, let p_1 and p_2 be the values of SNOBOL4 expressions

'BE' | 'B'

and

'ET' | 'AD'

respectively. Then,

p_1('BEAD', 0)=⟨2,1⟩

but

p_2('BEAD', 2)=⟨4⟩ and p_2('BEAD',1)=( )

so that

$(p_1 \frown p_2)$ ('BEAD', 0)=⟨4⟩⌢( )=⟨4⟩

Similarly,

p_1('BET', 0)=⟨2,1⟩

but

p_2('BET', 2)=( ) and p_2('BET',1)=⟨3⟩

so that

$(p_1 \frown p_2)$ ('BET', 0)=( )⌢⟨3⟩=⟨3⟩

3.3 LIMITATIONS ON DOMAINS

3.3.1 Pragmatic Limitations

We have already seen (Section 3.1.4) that it is possible to write PASCAL-like recursive **type** definitions analogous to recursive domain definitions. For example,

```
type natnum = record case iszero = Boolean of
                true : ( );
                false : (pred : natnum)
              end;
```

is the analog of **N** = {zero} + **N**, and

```
type sequence = record case isnull : Boolean of
                  true : ( );
                  false : (first : d ; rest : sequence)
                end;
```


is the analog of **S** = {*null*} + (**D**×**S**). But such definitions are *not* allowed in PASCAL. The difficulty is not conceptual, but pragmatic: the domains described contain an *infinite* number of elements, so that it is not possible to predict before execution how much space will be adequate to represent *any* element of the domain. The same problem arises with all infinite domains.

Several approaches to implementation of infinite domains are possible. One is analogous to the usual treatment of *numerical* domains such as the integers: impose an *a priori* bound on the number of elements that will be represented. If an attempt is made to compute an element outside of this finite subset ("overflow"), the computation should be aborted.

A second approach to implementation is to allocate space as it is required when values are computed. This is more complex for the implementer, but, of course, more convenient for the programmer. The recursively defined domains of LISP, APL, and SNOBOL4 are implemented in this way.

A compromise between these two approaches, using "pointers", is available in many languages, including PASCAL, and will be discussed later.

*3.3.2 Theoretical Limitations

So far, no *theoretical* limitations have been imposed on domains, or on the functions allowable between domains. But if we insist that domain elements and functions be *computable* (i.e., implementable in principle), then this seems unrealistic because a value computable in a finite amount of time can only contain a finite "amount of information". However, this does *not* mean that domains cannot contain values with *infinite* information content. For example, the squaring function is a computable value that specifies an *infinite* number of argument–result correspondences. As another example, it is possible to write a program to compute the decimal representation of the real number $\pi/10$ as a file of decimal digits

31415926535···

and, indeed, to implement arithmetic operations on real numbers to arbitrary precision. The squaring function and the decimal representation of $\pi/10$ are computable values with *unbounded* "information content". We shall term them ***infinitary*** values.

The apparent contradiction between the computability of some *infinitary* values and the constraint of *finite* computation may be resolved by realizing that computable infinitary values must be the *limits* of finitely computable *approximations* to them. For example, in a finite computation only a finite number of the argument–result correspondences of the squaring

function can be determined. But, more and more information about this function may be obtained by doing more and more computation. Furthermore, all such approximation sequences converge to the *same* limit, the squaring function. Similarly, only finite files of digits can be generated in finite time, but it is possible to obtain arbitrarily good approximations to an infinitary value by computing for a sufficiently long time.

The difference between a limit value and a finitely computable approximation to it is known as *truncation error* in connection with numerical computation. But, it is a *general* principle of computation that domains can contain values with unbounded information content only if these values are the limits of finite approximations to them. This limitation on domains explains our use of a special term "domain" for sets of *computable* values.

Now, let us consider functions over domains. Suppose that the domain of arguments of a function f includes an infinitary element d with unbounded information content. In a finite computation d can only be approximated, so that if f is to model a feasible computation it must be possible to approximate $f(d)$ as closely as desired by using some finite approximation to d. Therefore, if $d_0,d_1,d_2,\cdots$ is any sequence of better and better approximations that converge to d, the function f must satisfy

$$\lim_{i\to\infty} f(d_i) = f(\lim_{i\to\infty} d_i) = f(d)$$

It will then be possible to obtain as good an approximation as necessary to $f(d)$ by computing $f(d_i)$ for some *finite* approximation d_i. This is reminiscent of the ϵ–δ definition of continuity in calculus; functions that satisfy this kind of condition are termed ***continuous***. In general, for a function to be computable, it must be continuous at all arguments, that is to say, preserve limits of approximations in domains (where the notion of approximation has to do with the "amount of information" represented by the values).

As an example of a *discontinuous* function, let $\mathbf{T}=\{true,false\}$ and $\mathbf{T}^*$ be the domain of finite and infinite sequences of truth values such that s_1 approximates s_2 just if s_1 is a prefix (initial segment) of s_2; then consider function

$$all: \mathbf{T}^*\to\mathbf{T}$$

such that $all(s_\infty)=true$ for $s_\infty=\langle true,true,true,\cdots\rangle$ (i.e., the infinite sequence whose components are all *true*), and $all(s)=false$ otherwise. But now consider the finite sequences

$$s_i=(true, true,\cdots,true)$$

with i components for $i=0,1,2,\cdots$, whose limit is s_∞. Then,

$$\lim_{i\to\infty} all(s_i)=\lim_{i\to\infty} false=false$$

yet

$$all(\lim_{i\to\infty} s_i)=all(s_\infty)=true$$

so that function *all* is not continuous. But it is evidently not computable either because it requires an implementation to change a result from *false* to *true* after checking an *infinite* number of components.

There are analogs of the product, sum, function domain, and limit constructions and the notions of domain isomorphism and approximation that meet the conditions discussed in this section (that is, infinitary values are elements of domains just if they are limits of finite approximations to them, and functions are continuous). Furthermore, it can be shown that in this framework "smallest" solutions to recursive domain definitions using operations $\times$, $+$ and $\to$ *always* exist. This is not true if *arbitrary* sets and functions are allowed; for example, if $\mathbf{V}\to\mathbf{V}$ must contain *all* the functions from $\mathbf{V}$ to $\mathbf{V}$, then the *only* solution of equation

$$\mathbf{V}=\mathbf{D}+(\mathbf{V}\to\mathbf{V})$$

is a trivial one when $\mathbf{D}$ is the empty set. But if $\mathbf{V}\to\mathbf{V}$ is taken to be the domain of all *continuous* functions, then "smallest" solutions do exist, for *any* domain $\mathbf{D}$. This is a significant result because domain equations such as this are needed to define semantic domains for many programming languages.

In subsequent chapters, we shall see several applications of the general principles discussed in this section.

EXERCISES

3.1 In PASCAL, what are the counterparts for records with variant parts of the injection functions for a domain sum?

3.2 (a) A *binary tree* (with nodes labelled by elements of a domain $\mathbf{D}$) is either null or a 3-tuple (b_1,d,b_2), where $d\in\mathbf{D}$, and b_1 and b_2 are binary trees. Define domain $\mathbf{B}$ of binary trees.

(b) A *tree* (with nodes labelled by elements of **D**) is an ordered pair (d,f), where $d \in \mathbf{D}$ and f is a *forest*, that is, a (finite) sequence of trees. Define domains **T** of trees and **F** of forests.

(c) Show that $\mathbf{B} \cong \mathbf{F}$.

3.3 Define the domain of *list structures* in LISP, that is to say, lists each of whose components is either an atom (other than 'NIL') or a list structure. Assume that **A** is the domain of atoms other than 'NIL'.

3.4 Define in PASCAL-like notation a type that describes the LISP domain of S-expressions.

3.5 For a programmer, how would domain

$$\sum_{r} \sum_{n_1,\cdots,n_r} [\mathbf{H}+\mathbf{R}]^{n_1 \times n_2 \times \cdots \times n_r}$$

differ from the domain of APL arrays? Suggest reasons why this domain was not adopted by the implementers of APL.

3.6 Refer to documentation on SNOBOL4 and define the built-in patterns or pattern-valued operations REM, FAIL, LEN(N), POS(N), TAB(N), and ARB.

3.7 What patterns are the *identities* for pattern concatenation and alternation? i.e., what patterns p_c and p_a have the properties that, for all patterns p,

(a) $p \hat{} p_c = p_c \hat{} p = p$
(b) $p \mid p_a = p_a \mid p = p$

3.8 Show using the definitions of Section 3.2.3 that for all patterns p_1, p_2, and p_3,

(a) $(p_1 \hat{} p_2) \hat{} p_3 = p_1 \hat{} (p_2 \hat{} p_3)$
(b) $(p_1 \mid p_2) \mid p_3 = p_1 \mid (p_2 \mid p_3)$
(c) $(p_1 \mid p_2) \hat{} p_3 = (p_1 \mid p_3) \hat{} (p_2 \mid p_3)$

Also, show by giving counter-examples that the following are not valid in general:

(d) $p_1 \hat{} (p_2 \mid p_3) = (p_1 \hat{} p_2) \mid (p_1 \hat{} p_3)$
(e) $(p_1 \hat{} p_2) \mid p_3 = (p_1 \mid p_3) \hat{} (p_2 \mid p_3)$
(f) $p_1 \mid (p_2 \hat{} p_3) = (p_1 \mid p_2) \hat{} (p_1 \mid p_3)$

3.9 Define a pattern operation *not*(p) that succeeds if p fails and fails if p succeeds. Then, describe the effects of pattern operations defined as follows:

(a) *test*(p)=*not*(*not*(p))
(b) *and*(p_1,p_2)=*test*(p_1)$\hat{}$$p_2$
(c) *butnot*(p_1,p_2)=*not*(p_2)$\hat{}$$p_1$

Is *and*(p_1,*not*(p_2))=*butnot*(p_1,p_2)?

***3.10** Show that $\cong$ is an equivalence relation.

***3.11** If $\mathbf{A}' \cong \mathbf{A}$ and $\mathbf{B}' \cong \mathbf{B}$, show that

(a) $\mathbf{A}' \times \mathbf{B}' \cong \mathbf{A} \times \mathbf{B}$
(b) $\mathbf{A}' + \mathbf{B}' \cong \mathbf{A} + \mathbf{B}$
(c) $\mathbf{A}' \to \mathbf{B}' \cong \mathbf{A} \to \mathbf{B}$

***3.12** Given that **A**, **B**, and **C** are any sets, **0** is the empty set, and **1** is any singleton set, show that

(a) $\mathbf{A} \times \mathbf{1} \cong \mathbf{A}$
(b) $\mathbf{A} \times \mathbf{0} \cong \mathbf{0}$
(c) $(\mathbf{A} \times \mathbf{B}) \times \mathbf{C} \cong \mathbf{A} \times (\mathbf{B} \times \mathbf{C})$
(d) $\mathbf{A} + \mathbf{0} \cong \mathbf{A}$
(e) $(\mathbf{A} + \mathbf{B}) + \mathbf{C} \cong \mathbf{A} + (\mathbf{B} + \mathbf{C})$
(f) $\mathbf{A} \to \mathbf{1} \cong \mathbf{1}$
(g) $\mathbf{1} \to \mathbf{B} \cong \mathbf{B}$
(h) $(\mathbf{A} + \mathbf{B}) \to \mathbf{C} \cong (\mathbf{A} \to \mathbf{C}) \times (\mathbf{B} \to \mathbf{C})$

***3.13** Show that $(\mathbf{A} \times \mathbf{B}) + (\mathbf{A} \times \mathbf{C}) \cong \mathbf{A} \times (\mathbf{B} + \mathbf{C})$ and relate this to the treatment of record types in PASCAL.

***3.14** Show that $(\mathbf{A} \times \mathbf{B}) \to \mathbf{C} \cong \mathbf{A} \to (\mathbf{B} \to \mathbf{C})$ and relate this to the treatment of multi-dimensional arrays in PASCAL.

***3.15** The *powerset* $\mathcal{P}(\mathbf{D})$ of a set **D** is the set of all subsets of **D**, including the empty subset and **D** itself. Show that $\mathcal{P}(\mathbf{D}) \cong \mathbf{D} \to \{false,true\}$. Compare types **set of** T and **array** [T] **of** *Boolean* in PASCAL.

***3.16** Show that $\mathbf{D}^{\mathbf{n}} \cong \sum_{i} \mathbf{D}^{i}$.

***3.17** Show that the set of (parsed) binary numerals, as described by abstract syntax

N binary numerals

N ::= 0 | 1 | N0 | N1

is isomorphic to the domain defined by equation

$\mathbf{Nml} = \mathbf{1} + \mathbf{1} + \mathbf{Nml} + \mathbf{Nml}$

where **1** is any singleton set. Show that it is possible to define a general correspondence between abstract syntax notation and "polynomial" domain equations (i.e., in which only operations × and + are used).

***3.18** (Universality)

(a) Show that for all sets **A** and **B**, $\mathbf{A} \times \mathbf{B} = \{(a,b) \mid a \in \mathbf{A}, b \in \mathbf{B}\}$ and its projection functions $p_{\mathbf{A}}(a,b)=a$ and $p_{\mathbf{B}}(a,b)=b$ have the following

property: for any set **D** and any functions $f : \mathbf{D} \to \mathbf{A}$ and $g : \mathbf{D} \to \mathbf{B}$, there exists a unique function θ: $\mathbf{D} \to \mathbf{A} \times \mathbf{B}$ such that $f(d) = p_{\mathbf{A}}(\theta(d))$ and $g(d) = p_{\mathbf{B}}(\theta(d))$ for all $d \in \mathbf{D}$.

(b) Conversely, show that if **P** is a domain and $p_1 : \mathbf{P} \to \mathbf{A}$ and $p_2 : \mathbf{P} \to \mathbf{B}$ are functions that have the above property, then $\mathbf{P} \cong \mathbf{A} \times \mathbf{B}$.

The significance of these results is that they show that domain construction $\mathbf{A} \times \mathbf{B}$ is the *most general* of its class of domain constructions, and is *uniquely* characterized by this property (up to isomorphism). The sum, function domain, and limit constructions have similar universal properties.

*3.19 An enthusiastic language designer proposes to improve his favorite language CATCHALL by extending the equality operation ("=") to procedures. How would you persuade him that this is unwise?

PROJECT

Design a language that would allow both hierarchical structuring of data, as in LISP, and iterative structuring of data, as in APL.

BIBLIOGRAPHIC NOTES

The four constructions discussed in this chapter were first applied to programming language examples in McCarthy [3.6]; see also Hoare [3.4], Scott [3.7], Strachey [3.12], and Lehmann and Smyth [3.5]. The model of patterns in Section 3.2.3 is due to Gimpel [3.2]. The discussion of approximations, limits, and continuity in Section 3.3.2 is based on work of Scott [3.7–3.10]. For a detailed and rigorous presentation of Scott's theory of computation, see a book by Stoy [3.11]. For the project, see Gull and Jenkins [3.3] and Burge [3.1].

3.1 Burge, W. H. *ISWIM, a Mixture of APL and LISP*, Research Report RC 6967, IBM, Yorktown Heights, N.Y. (1978).

3.2 Gimpel, J. F.: "A theory of discrete patterns and their implementation in SNOBOL4", *Comm. ACM*, **16** (2), 91–100 (1973).

3.3 Gull, W. E. and M. A. Jenkins. "Recursive data structures in APL", *Comm. ACM*, **22** (2), 79–96 (1979).

3.4 Hoare, C. A. R. "Notes on data structuring", in *Structured Programming* (by O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare), pp. 83–174, Academic Press, London (1972).

3.5 Lehmann, D. J. and M. B. Smyth. "Algebraic specification of data types: a synthetic approach", *Math. Systems Theory*, **14** (to appear).

3.6 McCarthy, J. "A basis for a mathematical theory of computation", in *Computer Programming and Formal Systems* (eds., P. Braffort and D. Hirschberg), pp. 33–70, North-Holland, Amsterdam (1963).

3.7 Scott, D. S. "Outline of a mathematical theory of computation", in *Proc. 4th Annual Princeton Conference on Information Sciences and Systems*, Dept. of Electrical Engineering, Princeton University (1970); also technical monograph PRG-2, Programming Research Group, University of Oxford (1970).

3.8 Scott, D. S. "Lattice theory, data types, and semantics", in *Formal Semantics of Programming Languages* (ed., R. Rustin), 2nd Courant Computer Science Symposium (1970), Prentice-Hall, Englewood Cliffs, N.J. (1972).

3.9 Scott, D. S. "Data types as lattices", *SIAM J. on Computing*, **5** (3), 522–86 (1976).

3.10 Scott, D. S. "Logic and programming languages", *Comm. ACM*, **20** (9), 634–41 (1977).

3.11 Stoy, J. E. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass. (1977).

3.12 Strachey, C. "The varieties of programming language", *Proc. Int. Computing Symposium*, pp. 222–33, Cini Foundation, Venice (1972); also technical monograph PRG-10, Programming Research Group, University of Oxford, England (1973).

4 STORAGE

In the preceding chapter, storage was mentioned only in connection with a *pragmatic* problem, representation of infinite domains. Because none of the operations discussed there had "destructive" effects, the use made of storage by processors could be ignored in descriptions of the *semantics*. However, when a language includes commands (such as assignments), or operations with side effects on their arguments, some concept of storage must be introduced into semantic descriptions in order to record and transmit these effects. This is the subject of the present chapter.

4.1 STORES AND LOCATIONS

To describe the effects of assignment commands on the store, we must know what a store is. Fortunately, it will not be necessary to concern ourselves with such details as the word size, data representations, and addressing mechanisms of particular computers in order to describe high-level features of languages. It will be possible to use a much simpler *logical* model of storage.

It might seem from consideration of simple assignments such as

$$n := n+1$$

that a store could be modelled logically as a set of associations between *identifiers* (such as '*n*') and *storable values* (such as the numerical value of '*n*+1'). Then the effect of executing a command of the form

I := E

would be to yield a new store that differs from the old one only in that the target identifier, I, is associated with the value (relative to the old state) of the source expression, E.

However, this simple view of assignment and storage is adequate only for very limited programming languages. One difficulty is that the target of an assignment might be more complex than a simple identifier, as in

```
a[i] := ···
```

This could possibly be regarded as an abbreviation of an assignment to identifier '*a*'. But, consider PASCAL block

```
var p:↑integer;
begin
  ⋮
  new(p);
  p↑:=···
  ⋮
end
```

assignment '*p*↑:=···' illustrates that a target might not contain an identifier with which to associate a new value: this assignment updates the *anonymous variable* created by procedure *new*. In general, the target of an assignment is a (possibly composite) *l-expression*, and not necessarily a simple identifier. Furthermore, *l*-expressions may be used in contexts where there is no accompanying source expression, so that *l*-expressions themselves should be meaningful. But what is the "value" of an *l*-expression when it is being used as an assignment target?

Another factor that makes it difficult to regard a store as an association of *identifiers* with values is that in languages with nested blocks or recursively defined procedures it is possible to have many distinct "variables" with the *same* name. Even more problematical is that it is possible for one "variable" to have several names, so that execution of an assignment having one of them as its target also affects the values of the others, as in the following PASCAL block:

```
var i:integer;
procedure p(var x,y:integer);
  begin
    x:=···;
    ···y···i···
  end;
begin
  ⋮
  p(i,i);
  ⋮
end
```


When the procedure is invoked using '*p*(*i*,*i*)', execution of the assignment to *x* also surreptitiously changes the values of *y* and *i*! This is a form of storage sharing known as ***aliasing***.

These difficulties of semantic description may be overcome by introducing a domain of *semantic* entities, termed ***locations*** (or "references" or "cells") to be intermediaries between identifiers and stored values. Locations are the "logical" counterparts of machine addresses. Unlike physical machine words, locations do not have a "size", nor is it necessary to postulate a linear ordering on the locations. Indeed, all that must be assumed is that it is possible to test equality of locations. Then, a *store* is simply a finite set of associations of *locations* with *storable values*: a location is associated with the value that we want to regard as being currently ***contained in*** (or "stored at") that location.

To see how locations and stores may be used in semantic descriptions, consider an execution of the following PASCAL block:

```
var n:integer;
begin
  n:=0;
  while n<a do
    begin
      n:=n+1;
      ⋮
    end;
  ⋮
end
```

Execution of declaration

```
var n:integer;
```

has the effect of *allocating* some "new" (i.e., currently unused) location; let us designate this location *l*. Therefore, execution of the declaration yields a new store that records the fact that location *l* is now "in use", so that any subsequent allocations cannot claim the same location. The declaration also produces a new environment in which identifier '*n*' is bound to location *l*; that is, throughout the block (the scope of the declaration), '*n*' will denote *l*. Assignments may affect the value *contained* in *l*, but do not change the *binding* of '*n*' to *l*.

Note that allocation of new storage takes place *each time* the block containing a **var** declaration is executed. For the language FORTRAN, it is possible to *implement* all storage allocations for a program before its execution, so that if a declaration is executed more than once, the same location may in fact be used. However, the description of standard FORTRAN does

not *require* this approach to implementation, so that FORTRAN programmers should *not* assume that the initial contents of locations are "left over" from previous executions of the block (unless the special SAVE statement has been used).

Execution of assignment

$n:=0$

initializes *l*. In the store resulting from this execution, the location denoted by '*n*' has become associated with the value of the source expression, literal '0'. All of the other location-value associations are, of course, unchanged.

After initialization of the location, the value of an occurrence of identifier '*n*' in expressions such as '*n*<*a*' and '*n*+1' is the contents of *l* in the current store. This value is obtained by using the *environment* to determine that '*n*' is bound to location *l*, and using the *store* to determine what is currently contained in *l*. On the other hand, an occurrence of '*n*' as the *target* of

$n:=0$

or

$n:=n+1$

denotes *l* and not its contents, and the effect of executing such an assignment is to update the location to contain the value of the source expression.

In general, an occurrence of an identifier that is bound to a location may be "evaluated" in two ways: as the target of an assignment, it denotes the location; in an expression such as '*n*+1', its value is the current contents of that location. These are known as the ***l-value*** and the ***r-value***, respectively, of the identifier ("*l*" for *l*eft or *l*ocation, "*r*" for *r*ight or stored). These two modes of evaluation are also applicable to more complex expressions, but some kinds of expression have *only* an *r*-value; for example, '1' and '*n*+1' do not have *l*-values in PASCAL. An expression that does not have an *l*-value cannot appear as the target of assignment commands (and in other contexts where *l*-values are required). It is possible (though unusual) for expressions to have an *l*-value but not an *r*-value.

After execution of the block, location *l* is no longer accessible from within the program, and this is always true of locations allocated by **var** declarations in PASCAL. Processors take advantage of this by re-claiming such locations for other uses. This will be termed ***disposal*** of a location.

The period of time between an allocation and the subsequent disposal of a location is termed the ***lifetime*** of that "incarnation" of the location. Note the difference between the *scope* of an identifier binding (which is a region of program text), and the *lifetime* of a location. For locations allocated by **var** declarations in PASCAL, these concepts are closely related; in general, this need not be the case.

4.2 VARIATIONS ON ASSIGNMENT

4.2.1 Selective Updating

Consider assignments

```
a:=b;
c:=a[i]
```

in the context of declarations

```
var a,b:array[1 .. n]of char;
    i:1 .. n;
    c:char;
```

These assignments could be explained simply by supposing that *a* and *b* denote *single* locations which can contain arrays of characters. Then the first assignment would update location *a* to contain the contents of *b*, and the second assignment updates *c* to contain the *i*th component of the contents of *a*. In both cases, the effect is to update the *l*-value of the target to contain the *r*-value of the source.

But now, consider assignment

$a[i]:=c$

If we want to continue to describe the effect of assignment commands as that of updating the *l*-value of the target to contain the *r*-value of the source, then we must change our view of the effect of an array declaration, because a *single* location is not decomposable. We therefore suppose that the effect of

```
var a:array[1 .. n]of char;
```

is to bind '*a*' to an *array* of locations, each of which can contain single characters. The *l*-value of '*a*' is then the array of locations it denotes, so that an assignment like

$a:=b$

must be regarded as a multiple update of each of these to contain the corresponding component of array *b*. The *l*-value of '*a*[*i*]' is obtained by using the *r*-value of '*i*' to select a component location of the *l*-value of '*a*' (i.e., the array of locations). So now our general description of assignment applies to examples like '*a*[*i*]:=*c*' as well as to '*a*:=*b*' and '*c*:=*a*[*i*]'.

In PASCAL, arrays, records, and files (but not sets) are ***storage structures*** (and not merely *data* structures) in that they may be *selectively* updated by using *l*-expressions like '*a*[*i*]'. It is evidently more efficient to selectively update a storage structure "in place" than to operate on or copy "large" data structures.

4.2.2 Other Updating Operations

Execution of an assignment of the form

L:=L+E

requires *two* evaluations of *l*-expression L, although the *r*-value of L is obtainable from its *l*-value. Because such updating operations are very common, some languages provide mechanisms that allow them to be expressed so that only a single evaluation is necessary. For example,

ADD E TO L

in COBOL, and

L+:=E

in ALGOL 68 have the effect of updating the *l*-value of L to contain the sum of the *r*-values of L and E, but the *r*-value of L is obtained from its *l*-value, rather than by a separate evaluation. Note that because evaluation of L can have side effects, the number of evaluations is, in general, a semantic (as well as a pragmatic) issue.

4.2.3 Multiple Targets

Some languages allow more than one target *l*-expression in an assignment, so that its execution can update several storage structures to contain the same *r*-value. For example, the assignment command in ALGOL 60 has the general form

$$L_1:=L_2:=\cdots:=L_n:=E$$

It is executed by evaluating the *l*-values of *l*-expressions $L_1, L_2, \cdots$, and L_n and the *r*-value of E, and then updating all of the *l*-values to contain the *r*-value.

4.2.4 Multiple Assignments

In some circumstances, it is desirable to have an assignment involving several *sources* as well as several targets. For example, three conventional assignments and an explicit temporary location are normally required to swap the contents of two locations. With a ***multiple*** (or "simultaneous") assignment of the form

$$L_1,L_2,\cdots,L_n:=E_1,E_2,\cdots,E_n$$

swapping of the contents of, say, *x* and *y* could be expressed more simply and clearly by

$$x,y:=y,x$$

In general, the multiple assignment command is executed by first evaluating the *l*-values of all the *l*-expressions L_i and the *r*-values of all the expressions E_i, and then doing all of the updatings. Note that if there were any sharing among the *l*-values, the order of the updatings could be significant. For example, if $i=j$ the effect of executing

$$a[i],a[j]:=x,y$$

would depend on the order of the updatings (unless the values at *x* and *y* happened to be the same).

4.2.5 Assignment Expressions

In some programming languages, assignments are *expressions* (with side effects), rather than commands. For example, in ALGOL 68

L:=E

is an expression whose value is that of L, and updates storage as a side effect. This allows assignments to be written within expressions, as in

while $(n:=n+1)<a$ **do**···

Such idioms are often difficult to read because expressions are normally expected to be free of side effects.

4.3 POINTERS

We have seen that locations are a kind of semantic entity to which identifiers may be bound by executing **var** declarations and which may be composed into storage structures such as arrays. In some languages, storage locations and structures are not only *denotable* by identifiers, but also *storable* as contents of locations. Storage locations and structures that are themselves stored are known as ***pointers*** (or "references" or "links").

For example, the two blocks in Fig. 4.1 are equivalent in PASCAL, (except that after executing the second, the new location is still accessible via *np*). In the first, execution of the **var** declaration *binds* identifier '*n*' to a new location. In the second, invocation of procedure *new* has the effect of updating its argument, *np*, to *contain* a new location. This may be depicted in a linkage diagram as follows:

```
(i)   var n:integer;
      begin
        n:=0;
        while n<a do
          begin
            n:=n+1;
              ⋮
          end;
        ⋮
      end

(ii)  begin
        new(np);
        np↑:=0;
        while np↑<a do
          begin
            np↑:=np↑+1;
              ⋮
          end;
        ⋮
      end
```

Fig. 4.1.

L-expressions of the form

E↑

are used in PASCAL to access storage indirectly, that is to say, via a pointer value. The *l*-value of such an expression is the storage that is (pointed at by) the *r*-value of sub-expression E. Therefore, the location incremented by executing

$np\uparrow := np\uparrow + 1$

is not *np*, but the location that is (pointed at by) the contents of *np*, that is to say, the *r*-value of expression '*np*'.

A location that can contain pointers can also contain a special value denoted by the literal '**nil**' in PASCAL. This value is used mainly to terminate chains of links and it is an error to attempt to use it to access storage.

In general, storage allocated by an invocation of *new* in PASCAL may still be accessible on exit from the enclosing block, so that there is no automatic disposal. Some processors provide a procedure *dispose* so that the programmer can indicate explicitly when a location will no longer be accessed. In others, inaccessible locations are automatically searched for and

disposed of whenever more storage is needed; this is known as *garbage collection*.

Programming with pointers is notoriously error-prone, and programmers often have to resort to complex linkage diagrams to understand and describe the status of storage structures because the storage subject to updating by an assignment is not identified by name. Other problems are that pointers cannot meaningfully be input or output, and that storage management is much more complex when pointers are used. Why then are pointers available in high-level programming languages? There appear to be three reasons:

(1) *Allocation and disposal*

The **var** declaration in PASCAL and analogous mechanisms in other languages allocate storage only at block entry, and this is disposed of only on block exit. Pointers allow storage to be addressed indirectly, so that it may be allocated and disposed of at arbitrary execution points.

(2) *Sharing*

If a pointer value is contained in *several* locations, then all the locations *share* access to the storage pointed at. For example, after execution of

$p := q$

in the context of

var $p,q:\uparrow t$;

storage $q\uparrow$ is also accessible via p. In contrast, execution of

```
new(p);
p↑:=q↑
```

allocates new storage and copies the contents of $q\uparrow$ into it, so that $p\uparrow$ and $q\uparrow$ may be independently updated. Therefore, when pointers are used, the degree of sharing in a complex storage structure may be controlled by the programmer.

(3) *Structural modifications*

Pointers allow storage to be partially *re-structured* without re-constructing almost the entire structure. For example, insertion of a component into a sequence or removal from a sequence of one component is quite easy when the sequence is implemented as a linked structure of "nodes" in which each node points at the successor and predecessor nodes in the structure.

It would be desirable to have language facilities that achieve these objectives without the use of pointers, at least in some restricted or specialized situations. In the meantime, programmers might be advised to use pointers only as a last resort.

4.3 STORAGE INSECURITIES

An attempt to access the contents of a location *before* its initialization is a serious programming error, because the location will contain something "left-over" from its previous incarnation. Similarly, it is an error to attempt to update or access a location that has been disposed of while it is still accessible, because the location might also be in use in a subsequent incarnation. Storage that has been prematurely disposed of (whether at the explicit request of the programmer or implicitly by a processor) is known as a ***dangling reference***.

Dangling references can sometimes be prevented by careful language design. In PASCAL, for example, pointer values can be created only by procedure *new*, but in some languages it is possible to store *any* *l*-value as a pointer. For example, the following is illegal in PASCAL:

```
var i:integer;
    p:↑integer;
begin
  :
  p:=i;
  :
end
```

but its counterpart in ALGOL 68 is allowed, and has the effect of storing (a pointer to) location *i* in *p*. Note that in such a context it is the *l*-value (and not the *r*-value) of the source expression that is used. Similarly, language PL/I has a procedure ADDR that returns the *l*-value of its actual parameter as a pointer value that may then be stored. If, in such cases, the storage pointed at is automatically disposed of on exit from the block in which it was allocated, it becomes a dangling reference.

To illustrate the danger, suppose that an analogous procedure *addr* were added to PASCAL; then consider the following:

```
var p:↑integer;
procedure q;
  var i:integer;
  begin
    p:=addr(i)
  end;
```


If location *i* is disposed of on exit from the body of procedure *q*, it will be left dangling in *p*:

By restricting pointers to point only at storage allocated by *new*, PASCAL simplifies storage management and eliminates one source of dangling references. However, pointers may be dangling references if they are deallocated after a premature invocation of *dispose*.

Dangling references and uninitialized locations are common insecurities because they are difficult to prevent by language design and their detection by processors can be quite difficult and expensive. Some processors can take advantage of special hardware mechanisms and detect them during execution without too much inefficiency. In some cases, potential errors of this sort can be detected *before* execution by a static analysis of programs. But the most common implementation "solution" is to do nothing, thereby shifting the burden to the programmer.

4.5 TWO CASE STUDIES OF STORAGE STRUCTURING

4.5.1 Selective Updating in LISP

In Section 3.2.1 the domain of S-expressions in LISP was defined by

$$\mathbf{S}=\mathbf{A}+(\mathbf{S}\times\mathbf{S}),$$

where **A** is the domain of atoms. (That section should now be reviewed by readers not familiar with LISP.) This domain was satisfactory for explaining the five primitive operations of pure LISP (CONS, CAR, CDR, ATOM, and EQ), because none of these operations involved "destructive" effects.

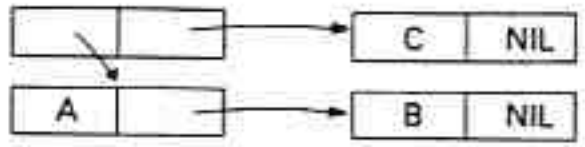
But full LISP provides two procedures, RPLACA ("replace the CAR") and RPLACD ("replace the CDR"), that modify their arguments. To describe the effects of these procedures, the LISP *storage* structures must be described. The domain of representations for S-expressions may be defined as follows:

$$\mathbf{R}=\mathbf{A}+(\mathbf{L}\times\mathbf{L})$$

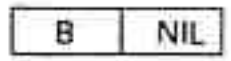
where **L** is the domain of *locations* that may contain elements of **R**. (In fact, even atoms have modifiable sub-structures termed "property lists", but we shall not discuss this.) The value of an expression in full LISP is therefore an element of **R**. When literal S-expressions are evaluated or when procedure CONS is invoked, new location pairs are allocated to represent S-expression pairs. For example, the storage structure that represents S-expression

((A B) C)

may be depicted



Similarly, if the value of identifier 'X' is



then the result of evaluating

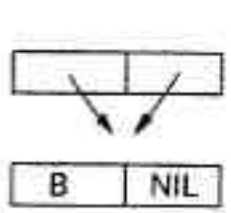
(CONS (QUOTE A) X)

is

Note that if

(CONS X X)

is evaluated, sub-structure X will be *shared*, as follows:



In general, elements of **R** are "unprintable"; when the value of an expression must be output, contents of locations are taken until atoms are reached.

If the value of expression E is the pair of locations (l_1,l_2), the values of

(CAR E) and (CDR E)

are the *contents* in the current store of locations l_1 and l_2, respectively.

It may be verified that with these definitions, the existence of the storage structure is "invisible" to a programmer who uses only the five primitive operations of pure LISP. But the motivation for introducing **R** was to be able to describe the effects of the selective updating operations, RPLACA and RPLACD. If the value of E_1 is the pair of locations (l_1,l_2), the value resulting from invocation

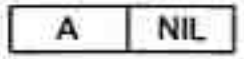
(RPLACA E_1 E_2)

is that of E_1, but it also has the *effect* of updating l_1 to contain the value of E_2. Similarly, the effect of

(RPLACD E_1 E_2)

is to update l_2 to contain the value of E_2.

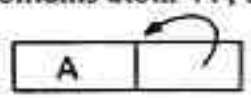
For example, if the value of identifier 'Y' is



the effect of executing

(RPLACD Y Y)

is to make Y a "re-entrant" storage structure, in which the CAR component of Y still contains atom 'A', but the CDR component of Y contains Y itself:



Such values must be used with care because of the danger of non-termination. For example, if an attempt were made to print Y, the system would start to output the *infinite* list

(A A A A ···

Another danger of RPLACA and RPLACD is that they may update the contents of storage structures that are unexpectedly shared at a higher level. Despite these pitfalls, selective updating is essential for efficiency in many applications of LISP.

4.5.2 Files in PASCAL

The design of high-level facilities for using "external" storage devices such as magnetic tapes and disks has been very problematical. Such facilities must be suitable for efficient implementation on a wide variety of hardware devices, and also must be compatible with complex and varied operating and file management systems. Other problems include specification of conversion and formatting for legible input and output, and establishment of correspondences between internal and external names for files. In this section, some aspects of the facilities provided in the language PASCAL for using sequential access devices (such as magnetic tapes, card readers, and line printers) will be described.

The concept underlying file types in PASCAL is the *sequence*. This may be seen most clearly if we begin by regarding a file name '*f*' as denoting a location containing a sequence $\langle x_1, x_2, \cdots, x_n \rangle$ of values. Then the following operations may be defined:

rewrite(*f*)	is equivalent to	$f := \langle\rangle$
write(*f*,*x*)	is equivalent to	$f := f \frown \langle x \rangle$
read(*f*,*x*)	is equivalent to	$x := first(f); f := rest(f)$
eof(*f*)	is equivalent to	$f = \langle\rangle$

These operations can be implemented efficiently because they insert or remove components only at the ends of sequences. There is no need to copy entire sequences, as might be necessary if the operations of assignment and concatenation of sequences were available to the programmer.

However, the above operations do not prevent a programmer from alternating freely between writing to and reading from a file, and this is impractical for implementation on most external storage devices. So an additional operation, *reset*, is introduced. Reading from a file is possible only after invoking *reset*. Furthermore, it is convenient to allow a file to be "re-read"; the *read* operation above does not save the file for subsequent re-reading. This suggests that a file f should be regarded as a *pair* of sequences, $\bar{f}$ and $\hat{f}$; $\bar{f}$ is either the file being written or the subfile already read, and $\hat{f}$ is the unread subfile. (These components are not directly accessible to the programmer.) Then the file operations may be defined as follows:

rewrite(*f*)	is equivalent to	$\bar{f} := \langle\rangle; \hat{f} := \langle\rangle$
write(*f*,*x*)	is equivalent to	$\bar{f} := \bar{f} \frown \langle x \rangle; \hat{f} := \langle\rangle$
reset(*f*)	is equivalent to	$\bar{f} := \bar{f} \frown \hat{f}; \hat{f} := \langle\rangle$
read(*f*,*x*)	is equivalent to	$x := first(\hat{f}); \bar{f} := \bar{f} \frown \langle x \rangle;$ $\hat{f} := rest(\hat{f})$
eof(*f*)	is equivalent to	$\hat{f} = \langle\rangle$

Note that these definitions allow writing to a file (possibly after reading from it) without an intervening invocation of *rewrite*; some file systems would not allow this.

A further complication is introduced by the concept of the *buffer*, $f\uparrow$, of file f. Cyclic buffer areas are an important implementation mechanism for making efficient use of external storage devices. The main motivation for making a "buffer" available to the *programmer* is to allow *selective* use and

updating of file components. Suppose that the components of a file f are "large" values. If '*read*(f,x)' is used, these large values must be copied from the buffer area to x, even if x is never updated or only a "small" part of x is used. Similarly, it is often necessary to construct a "large" value component-by-component before writing it to a file. If only '*write*(f,x)' is available, this must be done in x and the resulting value copied into the actual buffer area for the file. These inefficiencies can be avoided by allowing the programmer to access directly one component, denoted $f\uparrow$, of the physical buffer area.

File operations may be defined (still without reference to the underlying implementation) as follows:

rewrite(*f*)	is equivalent to	$\bar{f} := \langle\rangle; \hat{f} := \langle\rangle; f\uparrow := ?$
put(*f*)	is equivalent to	$\bar{f} := \bar{f} \frown (f\uparrow); \hat{f} := \langle\rangle; f\uparrow := ?$
reset(*f*)	is equivalent to	$\bar{f} := \bar{f} \frown \hat{f}; \hat{f} := \langle\rangle; f\uparrow := first(\hat{f})$
get(*f*)	is equivalent to	$\bar{f} := \bar{f} \frown \langle first(\hat{f}) \rangle; \hat{f} := rest(\hat{f});$ $f\uparrow := first(\hat{f})$
eof(*f*)	is equivalent to	$\hat{f} = \langle\rangle$

where '?' stands for an "undefined" (i.e., unpredictable) value and we define *first*($\langle\rangle$) to be '?'. Derived operations *read* and *write* are now definable in terms of these as follows:

write(*f*,*x*)	is equivalent to	$f\uparrow := x;$ *put*(f)
read(*f*,*x*)	is equivalent to	$x := f\uparrow;$ *get*(f)

Note that the programmer's buffer becomes "undefined" after a *put* operation, for no apparent reason. The explanation is that the programmer's buffer is implemented by using an "internal" pointer into the actual cyclic buffer area. The *put* operation is implemented by moving this internal pointer, and this action makes the contents of $f\uparrow$ unpredictable. It is unfortunate that this implementation aspect is "visible" to the programmer as an apparently unnecessary aspect of the logical model. Despite this, the treatment of sequential files in PASCAL is remarkably clean in comparison to similar facilities in most other programming languages.

EXERCISES

4.1 Is it possible in PASCAL to test equality of locations?

4.2 What are the advantages and disadvantages of the following approaches to the problem of uninitialized locations:

(a) The programmer can be compelled to initialize new locations; for example, the syntax of the **var** declaration might be extended to include an expression whose value becomes the initial contents, as in

var I:T=E;

(b) New locations can be initialized by the implementation to a "dummy" value such that any attempt to use this value will be trapped by run-time tests.

(c) New locations can be initialized by the implementation to a "neutral" value, such as zero for integer locations.

(d) New locations can be initialized by the implementation to an "unlikely" value, such as *maxint* (the largest representable integer).

(e) Programs can be analyzed at compile-time and possible uses of uninitialized locations detected and treated as syntactic errors.

4.3 Give an example to show that in general an ALGOL 60 assignment

$L_1:=L_2:=\cdots:=L_n:=E$

is not equivalent to

$L_n:=E;$
$L_{n-1}:=L_n;$
$\vdots$
$L_1:=L_2$

even when none of the expression evaluations have side effects.

4.4 (a) Describe in detail the effect of executing PASCAL assignment

$a[a[i]]:=a[i]+1$

(b) Is it always the case (in PASCAL) that the *r*-value of L after executing assignment L:=E is equal to the *r*-value of E before executing the command? *Hint*: consider the assignment in part (a).

4.5 Suggest an extension to PASCAL that would make it possible for *sets* to be selectively updated.

4.6 An enthusiastic language designer proposes to extend his favorite language CATCHALL by allowing array expressions and assignments. To simplify the implementation, he suggests that these be interpreted on a component-by-component basis; for example, in the context of


```
var a,b:array[1..n]of real;
    x:real;
```

assignment

$a:=a*x+b$

would be interpreted as

```
for i:=1 to n do
  a[i]:=a[i]*x+b[i]
```

How would you persuade him that this is unwise? *Hint*: consider assignment

$a:=a*a[j]+b$

4.7 Show how analogs of LISP primitives CONS, CAR, CDR, EQ, ATOM, RPLACA, and RPLACD may be implemented in PASCAL using the types defined by

```
type ref=↑sexp;
  sexp=record case isatom:Boolean of
         true:(at:atom);
         false:(car,cdr:ref)
       end;
```

Explain how this approach to representation of infinite domains has characteristics of *both* of the approaches discussed in Section 3.3.1.

4.8 Why, in PASCAL, will the following not work as the programmer probably intended?

```
var f:file of record···;a:char;···end;
begin
  ⋮
  with f↑ do
    while not eof(f) do
    begin
      write(output,a);
      get(f)
    end;
  ⋮
end
```

4.9 Some file systems allow a sequential file to be backspaced so that it may be read from back to front. Define an operation *backspace*(*f*) in terms of the model for PASCAL files.

4.10 Suppose that a file *f* is regarded as a pair of sequences, $\overleftarrow{f}$ and $\overrightarrow{f}$; then define the following operations:

rewrite(*f*) is equivalent to $\overleftarrow{f}:=();\ \overrightarrow{f}:=()$

extend(*f*) is equivalent to $\overleftarrow{f}:=\overleftarrow{f}\frown(?);\ \overrightarrow{f}:=()$

$f\uparrow$	is equivalent to	$last(\tilde{f})$
$reset(f)$	is equivalent to	$\tilde{f}:=\hat{f}^{\frown}\tilde{f}; \hat{f}:=()$
$get(f)$	is equivalent to	$\hat{f}:=\hat{f}^{\frown}(first(\tilde{f}))$; $\tilde{f}:=rest(\tilde{f})$
$eof(f)$	is equivalent to	$\tilde{f}=()$

and the following derived operations:

$write(f,x)$	is equivalent to	$extend(f)$; $f\uparrow:=x$
$read(f,x)$	is equivalent to	$get(f)$; $x:=f\uparrow$

For a programmer, how would this approach to files differ from the approach in PASCAL? Does it have any advantages?

4.11 Why is the following a reasonable definition of a domain of stores:

$$\mathbf{S}=\mathbf{L}\rightarrow(\mathbf{R}+\{unused\})$$

where **L** and **R** are the domains of locations and storable values, respectively?

4.12 An enthusiastic language designer proposes to extend PASCAL by allowing recursive **type** definitions without (explicit) use of pointers, as in

```
type string=record case isnull:Boolean of
                true:();
                false:(first:char;rest:string)
              end;
```

He proposes that this facility be implemented by representing values of such types using "internal" pointers (which would not be accessible to the programmer) to storage that would be allocated dynamically, as needed. For example, if, in the context of

```
var s:string;
```

s contained string 'ABC', then this would be represented by

What problems will arise with this approach if both updating and selective updating of locations like s are allowed? *Hint*: consider

```
var t:string;
begin
  ⋮
  t:=s;
  s.first:='Z';
  write(t.first);
  ⋮
end
```


PROJECTS

4.1 Design extensions to PASCAL that would allow more convenient (but still efficient) text processing.

4.2 Design high-level language facilities for making efficient use of "slow" random-access storage devices, such as disks.

BIBLIOGRAPHIC NOTES

The storage model presented in this chapter is due to Scott and Strachey [4.4, 4.5, 4.6]. The model for PASCAL files presented in Section 4.5.2 is that of Hoare and Wirth [4.2]. The suggestion in Exercise 4.10 is an improvement on the idea presented in Tennent [4.7]. For Exercise 4.12, see Hoare [4.1]. For Project 4.1, see Sale [4.3].

4.1 Hoare, C. A. R. "Recursive data structures", *Int. J. Computer and Information Sciences*, **4** (2), 105–32 (1975).

4.2 Hoare, C. A. R. and N. Wirth. "An axiomatic definition of the programming language PASCAL", *Acta Informatica*, **2**, 335–55 (1973).

4.3 Sale, A. H. J. "Strings and the sequence abstraction in PASCAL", *Software Practice and Experience*, **9** (8), 671–83 (1979).

4.4 Scott, D. S. "Mathematical concepts in programming language semantics", *Proc. 1972 Spring Joint Computer Conference*, pp. 225–34, AFIPS Press, Montvale, N.J. (1972).

4.5 Scott, D. S. and C. Strachey. "Towards a mathematical semantics for computer languages", in *Proc. of the Symposium on Computers and Automata* (ed., J. Fox), pp. 19–46. Polytechnic Institute of Brooklyn Press, New York (1971); also technical monograph PRG-6, Programming Research Group, University of Oxford (1971).

4.6 Strachey, C. "The varieties of programming language", *Proc. Int. Computing Symposium*, pp. 222–33, Cini Foundation, Venice (1972); also technical monograph PRG-10, Programming Research Group, University of Oxford (1973).

4.7 Tennent, R. D. "A note on files in PASCAL", *BIT*, **17**, 362–6 (1977).

5 CONTROL

The preceding chapter was devoted primarily to the semantics of assignment commands. In this chapter, we shall study composite syntactic structures that allow effects of assignments to be combined in useful ways. These are known as *control structures*. Discussion of forms of control sequencing that require more complex semantic models (such as concurrency and sequencers like the **goto**) will be deferred to subsequent chapters.

The *null* command, which makes *no* change to the store, is expressed in PASCAL by a null token string. In order to make null commands "visible" in examples, we shall use comment '{null}'.

5.1 SEQUENTIAL COMPOSITION

Sequential composition of commands C_1 and C_2 is expressed in PASCAL by using the semicolon as a separator:

$C_1;C_2$

Note that the null command expresses the ***identity*** for sequential composition; that is, for all commands C,

{null};C and C;{null}

are both equivalent to C.

In some languages, sequential composition is expressed by the separation (e.g., in FORTRAN) or by textual succession of commands (e.g., in PL/I), rather than by an explicit symbol such as the semicolon. For example, the PASCAL commands

f:=*f*•*i*; *i*:=*i*+1

would be expressed in FORTRAN by

```
F=F•I
I=I+1
```

and in PL/I by

```
F=F*I; I=I+1;
```

The semicolons here terminate (rather than separate) the commands.

5.2 SELECTIVE COMPOSITION

In mathematics, selection is typically expressed as in the following example:

$$d(i,j)=\begin{cases} 0, & \text{if } i\neq j \\ 1, & \text{if } i=j. \end{cases}$$

Programming languages have *selective* control structures, such as the **if** and **case** commands in PASCAL:

if E **then** C_1 **else** C_2

and **case** E **of** $\cdots;K_i:C_i;\cdots$ **end**

The single-alternative form of **if** construct may be regarded as an abbreviation for

if E **then** C **else** {null}

Another abbreviation provided in PASCAL is to allow a list of static expressions as case labels in the **case** construction, so that the corresponding command need not be written out more than once. That is,

```
case E of
  ⋮
  Ki: C;
  ⋮
  Kj: C;
  ⋮
end
```

may be expressed more compactly by

```
case E of
  ⋮
  Ki,Kj: C;
  ⋮
end
```

in which command C is not duplicated.

What should be the effect of a **case** command when the selecting expression has a value that is *not* one of the specified cases? One possibility is

to adopt the null command as the implicit default for all cases that are not specified explicitly, much like the single-alternative form of **if** command. But this is rather undesirable because then erroneous selecting values will not be detected. In PASCAL, such an occurrence is treated as an error which aborts program execution.

The conventional implementation for **case** commands in PASCAL systems is to select a case for execution by using the value of the selecting expression as a "subscript" into an array of commands; that is, program space must be reserved for each value in the range from the smallest to the largest of the case labels. This implementation approach is simple and fast, but, because of space requirements, unsuited to examples like

```
case i of
  1:      ···;
  13:     ···;
  245:    ···;
  1768:   ···;
  32002:  ···;
end
```

in which the case labels are "sparse".

More sophisticated implementation techniques which use search to reduce space requirements are possible. However, if implementation simplicity is an important design criterion (as in PASCAL), it would perhaps have been better to simply forbid "gaps" between cases and verify this before program execution.

Some PASCAL processors provide an extension that allows a command to be specified by the programmer for all otherwise unspecified cases, as in

```
case E of
    K1: C1;
      ⋮
    Kn: Cn
  others Cn+1
end
```

But a similar effect can be obtained in standard PASCAL as follows:

```
if E in [K1,···,Kn] then
  case E of
    K1: C1;
      ⋮
    Kn: Cn
  end
else Cn+1
```

This is equivalent, provided that expression E does not have side effects.

Another abbreviation that is sometimes provided is to accept subrange notation in case labels, as in

```
case E of
   ⋮
  K_i1..K_i2: C_i;
   ⋮
end
```

But then examples like

```
case i of
  -maxint..-1 :···;
            0 :···;
      1..maxint :···
end
```

are inappropriate if the conventional implementation of the **case** construct is used.

In some languages, the **if** construct is terminated syntactically by a "closing bracket", such as '**endif**' or '**fi**':

```
if E
  then C1
  else C2
endif
```

However, this approach to syntax would make a *nested* sequence of selections clumsy to express because of the accumulation of closing brackets, as in:

```
if E1 then C1
  else if E2 then C2
  else if ···
    ⋮
  else if En then Cn
  else Cn+1
endif···endif endif endif
```

Therefore, such languages generalize the **if** form so that a sequence of tests may be expressed with a single construct, as follows:


```
if E1 then C1
  elsif E2 then C2
  elsif···
  ⋮
  elsif En then Cn
  else Cn+1
endif
```

Expressions E_i are evaluated for $i=1,2,\cdots$ until a *true* value is obtained. Then the corresponding C_i is executed, or C_{n+1} if none are *true*.

5.3 ITERATIVE COMPOSITION

Iterative notations such as

$$\sum_{i\in S}\cdots i\cdots$$

are common in mathematics, and these concepts have been adapted to programming languages. We shall distinguish between ***definite*** iterations, for which the number of repetitions is determined *before* any of the executions of the iterated construct (though not necessarily before program execution), and ***indefinite*** iterations, for which the number of repetitions is not predetermined. Note that this is similar to the distinction between definite integration

$$\int_a^b \cdots x \cdots dx$$

and indefinite integration

$$\int \cdots x \cdots dx.$$

5.3.1 Definite Iteration

The simplest approach to determining the number of repetitions is to provide *no* means to terminate the iteration. For example, the construct

```
loop
  C
end
```

might express the (potentially) infinite repetition of C. This is analogous to a definite integral like

$$\int_0^\infty \cdots x \cdots dx.$$

Some *additional* language mechanism (such as a sequencer) would be used to avoid non-terminating executions.

The other approach is to determine the number of iterations by evaluating expressions. This can be made more convenient by allowing the programmer to access the iteration count from within the iterated construct with a "control variable", analogous to iteration index '*i*' in

$$\sum_{i \in S} \cdots i \cdots$$

Furthermore, the "counting" may be made more flexible by allowing both lower and upper count limits and, possibly, an increment to be specified. The PASCAL constructs of this kind have the forms

for I:=E_1 **to** E_2 **do** C

and

for I:=E_1 **downto** E_2 **do** C

The "counting" can be over programmer-defined enumerations, character values, and Boolean values, as well as integers. Although the increments are restricted to those obtained by single uses of *succ* or *pred*, these are the most useful by far, and somewhat simpler to describe and implement than arbitrary increments.

5.3.2 Indefinite Iteration

The indefinite iteration constructs in PASCAL are

while E **do** C

and **repeat** C **until** E

The termination test is done either before or after each execution of C. The construct

loop
  C_1
while E:
  C_2
repeat

has been proposed as a generalization of these. It allows the termination condition to be tested in the "middle" of the loop body. Note that the effect of the **while** form in PASCAL may be obtained by making C_1 the null command, and an effect similar to that of the **repeat** form by making C_2 the null command.

An even more flexible approach would be to allow *several* termination tests, as in

loop
  C_1
when E_1 **exit**
  C_2
when E_2 **exit**
  ⋮
when E_n **exit**
  C_{n+1}
repeat

But then, after the iteration terminated, there would be no way to tell which of the tests caused the termination without re-evaluating them. Therefore, it would be desirable to allow specification of some computation after any successful termination test, as in

loop
  C_1
when E_1 **do** C_1' **exit**
  C_2
when E_2 **do** C_2' **exit**
  ⋮
when E_n **do** C_n' **exit**
  C_{n+1}
repeat

However, the syntax of this construct is rather too intricate for it to be recommended. In Chapter 10, we shall discuss a more flexible and syntactically simpler approach to multiple exits using sequencers.

An interesting special case of the use of more than one termination condition sometimes arises. Consider PASCAL iteration

while $(i \leq n)$ **and** $(a[i] \neq x)$ **do** $i := i+1$

in the context of declaration

var *a* : **array**[1..*n*]**of** *t*;

With most PASCAL processors, evaluation of the composite test is in error when $i=n+1$ because the subscript in '$a[i]$' is out of range.

This problem is avoided if expression

E_1 **and** E_2

is evaluated ***sequentially***, that is to say, if E_2 is only evaluated when the value of E_1 is *true*. When the value of E_1 is *false*, the result is *false* and E_2 is not evaluated at all. In the example, the erroneous subscripting operation is then avoided. Similarly, for expression

E_1 **or** E_2

E_2 is evaluated only if the value of E_1 is *false*.

Another possible approach is to augment the syntax of the **while** and **repeat** constructs to allow *several* tests as immediate constituents, as in

while E_1 **andthen** E_2 **andthen**···**andthen** E_n **do** C

and **repeat** C **until** E_1 **orelse** E_2 **orelse**···**orelse** E_n

and to test them sequentially. For the **while** form, expressions E_i would be evaluated in order until one has the value *false* (and for the **repeat** form, until one has the value *true*); otherwise, command C would be executed. Note that '**andthen**' and '**orelse**' here are separators which are components of the iterative constructs, rather than operators forming composite Boolean expressions.

5.3.3 Iteration in ALGOL 68

Many other iteration constructs are possible, combining in various ways the concepts discussed in the preceding sub-sections. As an example of a relatively complex iteration mechanism, we shall discuss the construct in ALGOL 68. In its most elaborate form, it is

for I **from** E_1 **by** E_2 **to** E_3 **while** E_4 **do** C **od**

Its meaning should be clear from the syntax if it is noted that expressions E_1, E_2, and E_3 are integer-valued and are evaluated only once, and the scope of I is E_4 and C.

In its complete form, as above, the construct is rather complex, but portions of it may optionally be omitted as follows:

(a) if "**for** I" is omitted, the iteration count is not accessible in E_4 or C;
(b) if "**from** E_1" is omitted, the default initial value of I is one;
(c) if "**by** E_2" is omitted, the default increment for I is one;
(d) if "**to** E_3" is omitted, the default limiting value for I is infinity;
(e) if "**while** E_4" is omitted, no termination condition is tested.

Thus the only part of the construct that is never omitted is

do C **od**

In effect, there are thirty-two iterative constructs wrapped into one. This approach is quite different from that of PASCAL, which provides a *small* number of *distinctive* iterative constructions.

One difficulty with the approach in ALGOL 68 is that in general an iteration might be terminated by either the iteration count limit being exceeded or the **while** condition failing, and there is no convenient way of knowing without re-testing the condition. Another is that some simple and useful forms of iteration (such as the **repeat** in PASCAL) are not conveniently expressible.

*5.3.4 Semantics of Iterations

We have so far been rather vague about the semantics of iterations, relying on the reader's programming experience. But the question arises: how can the semantics of a construction like

while E **do** C

be specified? In many language descriptions this is achieved by *translating* the iteration into an equivalent form that uses the **goto** sequencer, such as

```
begin
1: if E then
   begin
     C;
     goto 1
   end
end
```

Although this may be how the **while** loop is actually *implemented*, it does not solve the problem of *semantic* description, because it is quite difficult to describe the semantics of labels and sequencers such as the **goto**. Therefore, the translation approach is conceptually quite unsatisfactory. The reason that the **while** loop is available in programming languages is to provide a semantically *simple* means of expressing a common pattern of control sequencing, so that it is rather inappropriate to describe its effect to the programmer in terms of semantically more complex linguistic devices such as labels and sequencers.

Another approach that has been used, particularly for languages without sequencers, is to specify that the effect of

while E **do** C

is to be equivalent to

```
if E then
begin
  C;                                    (5.1)
  while E do C
end
```

Again, we would certainly expect this equivalence to be true of the semantics of these control structures, but it is not quite adequate as a *definition* of the **while** form because its meaning is given in terms of *itself*, and not in terms of the meanings of its immediate constituents, E and C, alone.

This problem may be solved by a method that is closely analogous to the limit construction for recursively defined domains discussed in Section 3.1.5. Let C_0 be any command whose execution *never* terminates. For example, C_0 might be

```
while 0<1 do {null}
```

Then, define commands C_{i+1} for $i \geq 0$ to be

```
if E then
begin
  C;
  Ci
end
```

C_{i+1} is just command (5.1) with the "recursive" occurrence of the **while** loop replaced by C_i. In general, each of the C_i is a command whose effect is exactly that of the **while** loop, provided that less than i iterations are sufficient. Therefore, the meanings of all the C_i are *approximations* to the intended meaning of

```
while E do C
```

in the sense that whenever they provide any information at all, the same information would be obtained by executing the **while** loop. Furthermore, the sequence of meanings of $C_0, C_1, C_2, \cdots$ is a sequence of better and better approximations to the intended meaning of the **while** loop, in that they represent the effect of more and more computation. Consequently, it is appropriate to *define* the meaning of the **while** loop as the *limit* of this sequence of approximate meanings, making use of the concepts discussed in Section 3.3.2. It can be proved that this limit is equal to the meaning of command (5.1).

Note that each C_{i+1} was defined using only the meanings of immediate constituents E and C of the **while** loop, the "simple" control structures ';' and '**if**', and C_i. It is evident by simple induction on i that all of the approximate

meanings and therefore their limit really depend *only* on the meanings of E and C, the immediate constituents of the **while** loop. The use of selection and sequencing control structures and the initial command, C_0, is just a convenient way of expressing the approximate meanings.

5.4 EXPRESSION CONTROL STRUCTURES

Most of the examples of control structures in the preceding sections have been *commands*; however, most of the concepts discussed also arise with *expressions*. For example, ALGOL 68 has a composite expression form

$$E_1;E_2$$

whose value is that of E_2, evaluated *after* evaluation of E_1 (for its side effect only; the *value* of E_1 is simply discarded). For example, the following is an expression in ALGOL 68:

```
a:=a*a;
b:=b*b;
 a+b
```

(Recall that assignments are *expressions* in ALGOL 68.)

An example of a selection expression is the conditional expression of ALGOL 60:

```
if E then E else E
```

Some languages (such as ALGOL 68) also have **case** expressions.

Iterative expressions are rather rare in programming languages, though they are quite common in mathematics. An example is the "implied DO" of FORTRAN input and output commands. For example,

```
(A(I),I=1,N)
```

is equivalent to

```
A(1), A(2),···,A(N)
```

It would be possible to have a fairly general iterative expression of the form

```
for I=E to E op O on E
```

where O is a binary operator. For example,

```
for i=a to b op+on···i···
```

would be the analog of

$$\sum_{i=a}^{b} \cdots i \cdots$$

5.5 NON-DETERMINATE SELECTION

The description of a language does not have to *fully* specify the result of a computation. It is possible and sometimes desirable to allow implementations to have some degrees of freedom. Language features whose computational results are only partially specified will be termed ***non-determinate*** (or "non-deterministic").

There are several motivations for non-determinacy. One is to allow processors on different machines to use whatever approach is most efficient for each computer, even if results may differ somewhat, as with arithmetic operations on "real" (i.e., floating-point) numbers. Another reason is to allow a programmer to make explicit in his program the fact that there may be several equally acceptable computational paths to a solution, or possibly several acceptable solutions. An important example of non-determinacy that will be discussed in Chapter 11 arises with concurrent (parallel) processing, when the relative speeds of the processors are not determined.

The effect of interpreting a non-determinate construct may be described in terms of a (non-empty) set of *possible* "results", say, values or stores. A *processor* has the freedom to produce *any* element of the set; on the other hand, a *programmer* has the responsibility to ensure that *every* element of the set would be acceptable to him. Program correctness is especially important with non-determinacy, because results of program executions may not be reproducible. This makes conventional debugging extremely difficult and unreliable.

The simplest non-determinate control structure allows a choice of interpreting *either* of two expressions or commands. This might be expressed by

$$E_1 \mid E_2 \quad \text{or} \quad C_1 \mid C_2$$

The set of possible values or stores resulting from interpretation of these constructs would be the *union* of the sets of possible results of the immediate constituents. However, there seems to be no good reason to provide this particular form of non-determinacy in a programming language.

A more useful control structure is non-determinate sequencing of commands, which might be expressed by

$$C_1, C_2$$

The commands are to be executed in sequence, but the order of execution is not determined. This is equivalent to

$$C_1;C_2 \mid C_2;C_1$$

and could be used wherever conventional sequencing would be unnecessarily over-specific.

The following control structure which uses non-determinate selection has been proposed:

```
if E1→C1
 | E2→C2
 ⋮
 | En→Cn
end
```

The implementation is expected to select for execution any of the commands C_i whose "guard" E_i evaluates to *true*. If none are *true*, the computation is in error; if more than one is, the computation may have more than one possible result. For example, the effect of executing

```
if a[i]≤a[j]→m:=i
 | a[i]≥a[j]→m:=j
end
```

is to set m to the index of the smaller of $a[i]$ and $a[j]$. Note the symmetry of the construction: if $a[i]=a[j]$, either of the assignments may be executed, yielding different but presumably equally acceptable results.

Similarly, the following is an *iteration* that uses non-determinate selection:

```
loop E1→C1
   | E2→C2
   ⋮
   | En→Cn
end
```

An implementation is expected to execute any of the commands whose guard evaluates to *true*, and then repeat this until none of the guards are *true*. As before, if more than one guard is *true*, the implementation has a choice of which computational path to pursue.

The kind of non-determinacy that has been discussed so far in this section has been termed *demonic* non-determinacy. A programmer must assume that the implementation will always make the *worst* possible choice at every branch point; otherwise, he could not be certain that the program would *always* execute correctly. For example, the following is *not* suitable for setting i to an arbitrary non-negative integer, because it is possible for an implementation to *always* choose the first guarded command, so that the computation never terminates:

```
i:=0, b:=true;
loop b→i:=i+1
   | b→b:=false
end
```

Another kind of non-determinacy, which might be termed *oracular* non-determinacy, requires a processor to try whenever possible to avoid any "bad" choices, that is to say, branches leading to errors, failure, non-termination, or other kinds of computational disaster. The processor may be thought of as consulting an oracle to guide its choice of computational path into a "safe" branch. Of course, this is really implemented by following *all* possible computational paths until a successful one, if any, is found. This is achieved either by using separate processors in parallel, or by sharing one processor among the alternative branches. Note that such sharing must be *fair* in that computation must not be restricted exclusively to any single branch (unless all of the other branches have already failed), in case it becomes an *infinite* computation that should be avoided. Oracular non-determinacy is more convenient for the programmer than the demonic variety, but is much harder to implement.

EXERCISES

5.1 Some languages have **case** constructions that select by *position*. For example, in the following, if the value of expression E is integer i, then command C_i is selected:

case E **of**
C_1;
C_2;
⋮
C_n
end

What disadvantages does this approach have in comparison with the case construction in PASCAL?

5.2 The **case** command in PASCAL may be thought of as a way of indexing into a (one-dimensional) *array* of commands. Design a *multi-dimensional* generalization of the **case** command.

5.3 Show how to get the effect of the PASCAL form

repeat
C
until E

in ALGOL 68.

5.4 In mathematics, the value of

$$\sum_{i \in S} \cdots i \cdots$$

is zero when S is the empty set. In PASCAL, command

for I:=E_1 **to** E_2 **do** C

has no effect (except for side effects of the evaluations) if the value of E_1 is greater than that of E_2. What general principle governs these conventions? If the value of E_1 is greater than that of E_2, what should be the values of iterative expression

for I=E_1 **to** E_2 **op** O **on** E_3

when O is **'and'**, **'or'**, and '*'? What about '−' and '/'?

5.5 How might the iterative construction in ALGOL 68 be extended so that it could be used as an expression?

5.6 Show how to get the effect of

for I:=E_1 **to** E_2 **do** C

in PASCAL without using **for** or **goto**. *Hint*: consider

for *i*:=*m* **to** *n* **do**···

in the context of declarations

var *m*:*integer*;
i:1..*n*;

5.7 In ALGOL 60, the increments and limit values of the **for** iteration form may be real numbers. What problems can arise from this?

5.8 Why do some PASCAL processors prevent the count identifier for a **for** construct from being updated during execution of the iteration body?

5.9 Design facilities that would allow commands such as the following to be expressed more compactly:

(a)
if *a*[1]=*x*→*i*:=1
| *a*[2]=*x*→*i*:=2
⋮
| *a*[*n*]=*x*→*i*:=*n*
end

(b)
loop *a*[1]=*x*→*b*[1]:=*b*[1]+1; *next*(*x*)
| *a*[2]=*x*→*b*[2]:=*b*[2]+1; *next*(*x*)
⋮
| *a*[*n*]=*x*→*b*[*n*]:=*b*[*n*]+1; *next*(*x*)
end

where *next*(*x*) updates *x*. The control structures are those described in Section 5.5.

5.10 Is there any need for *selective* control structures whose tests are evaluated sequentially, as in the following?

if *i*>*n* **orelse** *a*[*i*]≠*x* **then**···**else**···

***5.11** Describe the semantics of

(a) **repeat** C **until** E (PASCAL)
(b) **loop** C **while** E:C' **repeat** (Section 5.3.2)
(c) **loop**···[E$_i$→C$_i$]···**end** (non-determinate)

as limits of sequences of approximate meanings.

PROJECT

Do a critical study of the iterative construct in the programming language ALGOL 60.

BIBLIOGRAPHIC NOTES

The **loop**···**while**···**repeat** construct was proposed by O.-J. Dahl and discussed by Knuth [5.2]. The non-determinate control structures of Section 5.5 were described by Dijkstra [5.1].

5.1 Dijkstra, E. W. *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J. (1976).

5.2 Knuth, D. E. "Structured programming with **goto** statements", *Computing Surveys*, **6** (4), 261–301 (1974).

6 BINDING

It may be recalled that distinctions were made in Section 2.3 between *binding* (of identifiers) and *updating* (of storage), between *environments* (which associate identifiers with the values they denote) and *stores* (which associate locations with the values they contain), between *definitions* (which yield new environments) and *commands* (which yield new stores), and between *scope* (of identifier bindings) and *lifetime* (of incarnations of locations). Programmers generally feel more at ease with the assignment-related concepts discussed in the preceding two chapters (updating, locations, stores, commands, lifetime) because of their familiarity with the properties of storage devices. But binding and related notions are equally or even more important in programming languages, particularly when *large* programs must be written.

Unlike assignment-related notions, which are features distinctive to programming languages, the terminology and concepts related to identifier binding are derived from mathematical notations. For example, the symbol 'x' in

$$\int_a^b \sin^2(x)\, dx$$

is conventionally known as a ***bound*** variable, and the expression '$\sin^2(x)$' occurring in it is known as the ***scope*** of the binding. The same terminology and concepts arise in programming languages.

In the present chapter, attention is focussed on *where* binding occurs in programs. In the first section, two kinds of identifier occurrences will be distinguished, and the subsequent section will survey the various approaches to binding taken in programming languages. In the final section, the concept of a *free* identifier is introduced.

6.1 BINDING OCCURRENCES AND APPLIED OCCURRENCES

Consider the PASCAL definition

```
const i=-j;
```

It contains two identifier occurrences. The occurrence of '*i*' is a ***binding*** occurrence: the identifier is being defined, and merely stands for itself. The occurrence of '*j*' is an ***applied*** occurrence: it is being used rather than defined, and must have been defined elsewhere. In every form of definition in PASCAL, there is a binding occurrence of the identifier being defined by that definition. Applied occurrences of identifiers are typically in expressions, where the identifier is used to symbolize the value with which it is associated in the current environment.

Note that an occurrence of an identifier as the left-hand side of an assignment is an applied and not a binding occurrence. For example, in

```
var i : integer;
begin
  i :=0;
  ⋮
  i :=i+1;
  ⋮
end
```

the occurrence of '*i*' in the declaration is the *only* binding occurrence. At that point, the identifier is bound to a storage location. All of the other occurrences of '*i*' are applied occurrences, where the identifier denotes that same storage location. The assignment has the effect of updating the location, but does not change the binding of identifier '*i*' to that location.

Usually, the meaning of an applied identifier occurrence is established at a unique and explicit binding occurrence of that identifier. This may be depicted by drawing an arrow in the program text from the applied occurrence to its (circled) binding occurrence, as in

```
type ①= 0..31;
var①, i ;
begin
  i :=0;
  ⋮
  i := i+1;
  ⋮
end
```


Note that if programmers could use such arrows in programs, the identifiers themselves would in principle be redundant.

6.2 APPROACHES TO BINDING

6.2.1 Syntactic Bindings

An important property of a programming language is the ease with which a program reader can *find* binding occurrences in programs, for it is at binding occurrences that identifiers acquire their meaning. This is easiest if binding occurrences are determined by the syntax of the language, in which case the binding may be termed ***syntactic***. Table 6.1 gives the productions of the abstract syntax of PASCAL (from Appendix B) with all of the syntactically determined binding occurrences of identifiers and label numerals indicated by circled symbols ① and Ⓝ.

The binding occurrences in definitions, parameters, and **record** type

L ::= I|L.I|L[E]|E↑

E ::= B|I|OE|EOE|I(···,E,···)|L.I|L[E]|E↑
|[···,E,···,E..E,···]](E)

K ::= B|I|OK

T ::= I|(···,①,···)|K..K|↑I|**set of** T
|**array**[T]**of** T|**file of** T|**record**···;①:T;···**end**
|**record**···;①:T;···**case** ①:I **of**···;K:(···;①:T;···);···**end**

Q ::= I:I|**var** I:I
|**procedure** I(···;Q;···)|**function** I(···;Q;···):I

P ::= ①:I|**var** ①:I
|**procedure** ① (···;Q;···)|**function** ① (···;Q;···):I

D ::= **const** ①=K;|**type** ①=T;|**var** ①:T;
|**procedure** ① (···;P;···);C;|**function** ① (···;P;···):I;C;

S ::= **goto** N

C ::= |L:=E|I|I(···,E,···)|C;C
|**if** E **then** C|**if** E **then** C **else** C
|**case** E **of**···;K:C;···**end**
|**while** E **do** C|**repeat** C **until** E
|**for** I:=E **to** E **do** C|**for** I:=E **downto** E **do** C
|Ⓝ:C|S
|**with** L **do** C|···D···**begin** C **end**|**begin** C **end**

M ::= **program** ①(···,I,···);C.

Table 6.1 Binding Identifier Occurrences in PASCAL

expressions are predictable and easy to locate. Others, such as command labels and enumerated constant identifiers, though syntactically determined, may be "buried" in large type expressions or commands where binding occurrences are not expected, and these are often less easy to find.

6.2.2 Nested Bindings

In PASCAL, a programmer is allowed to *re*-define identifiers. For example, in

```
const i=3;
  ⋮
  ⋮ {here i=3}
  ⋮
procedure p;
  const i=4;
  begin {p}
    ⋮
    ⋮ {here i=4}
    ⋮
  end; {p}
begin
  ⋮
  ⋮ {here i=3}
  ⋮
end
```

the **procedure** definition contains a ***nested*** binding of '*i*'. Note that this creates a "hole" in the scope of the outer binding. The scope of a binding in PASCAL excludes any *nested* scopes of the same identifier.

It might seem desirable to *forbid* nested bindings for the sake of program readability. Yet almost every programming language allows such re-definition. The reason appears to be that this allows a programmer of an "inner" scope to ignore any non-local identifier bindings that are not actually inherited and used in the local scope. If nested bindings were forbidden, he would have to keep in mind *all* of the identifiers that were bound non-locally, or be forced to examine the whole context of that inner scope before introducing any local names. This is so undesirable that nested bindings are tolerated, but as a matter of good programming style, they should be avoided as much as possible. Language processors could help programmers by pointing out nested bindings in program listings with warning messages, and by making it easy to systematically change identifiers in order to remove inadvertent name clashes.

6.2.3 Implicit Bindings

Not all identifier bindings in PASCAL may be attributed to explicit binding occurrences of those identifiers. Execution of command

with L **do** C

involves binding the field names of the type of L to the corresponding components of the *l*-value of L. For example, in

```
var x : record···;i:···;···end;
begin
  ⋮
  with x do···i···;
  ⋮
end
```

applied occurrences of '*i*' in the **with** command denote the '*i*' component of *x*. This is an example of what may be termed ***implicit*** binding, in that at the program point where the identifier is bound, there is no explicit occurrence of it.

If, in

with L **do** C

command C is "large", possibly itself containing implicit bindings, it may become quite difficult for a program reader to find the binding points for applied identifier occurrences, particularly since all other identifier bindings in PASCAL are in procedure headings. Programmers should use this construction only with "small" commands C. This problem with the **with** command in PASCAL would not have arisen if the construct had required programmers to make explicit the bound field names, as in

with (···,I_i,···)=L **do**
  ···I_i···

Processors for languages with implicit binding could improve program readability by making implicitly bound identifiers manifest in program listings. This could be done by inserting "comments" that specify the bound identifiers (and their types) at the points where they are implicitly bound.

6.2.4 Default Bindings

In PASCAL, any applied identifier occurrence that is *not* in the scope of any

binding occurrence of that identifier and is not a pre-defined ("standard") identifier, is in error. However, in some languages (including FORTRAN and PL/1) any applied identifier occurrences that are not *explicitly* bound are assumed to be bound by ***default*** in some way.

Default bindings are intended to be a convenience to the program *writer*; however, there is the significant disadvantage that the program text contains less information for program *readers*, including language processors. Therefore time can be wasted in trying to locate an explicit binding occurrence for an identifier that is actually bound by default. Furthermore, languages with default binding are very susceptible to errors arising from trivial misspellings: a misspelled identifier or keyword will be bound by default. It can be extremely difficult for programmers to find such errors, or even to be aware that they have occurred, unless the program's output is evidently incorrect. The minor "convenience" of being allowed to omit an explicit binding is often paid for by more difficult debugging, incorrect results, or disaster.

Programmers should bind explicitly all identifiers used in their programs, even if the language in use allows default binding, and processors of such languages could help by making any default bindings manifest in program listings with added "comments" or warning messages.

6.2.5 Overloaded Identifiers

Normally, an identifier does not denote more than one semantic entity in any context. But in some languages it is possible for a programmer to ***overload*** an identifier, that is to say, to bind it to more than one meaning. Of course, the value of an applied occurrence of such an identifier would then be ambiguous, so that the context of the applied occurrence must be used to resolve the ambiguity.

An example of overloading in PASCAL is the treatment of **function** names. Within the body of a **function** definition, the identifier denotes a location for returning a value in some *l*-expression contexts, and the procedure itself in other contexts. For example, identifier '*f*' is overloaded in

```
function f(n : integer) : integer;
  begin
    if n=0 then f:=1 else f:=n*f(n-1)
  end
```

Some other languages make more extensive use of overloading. For example, PL/1 allows several procedures to have the same name. The types of the actual parameters of an invocation are used to select the appropriate

procedure. This allows conceptually similar procedures on distinct domains to have the same name, just as literal operators like '+' and '•' may be used with both *integer* and *real* operands. Overloaded procedure names and operators are sometimes termed "generic".

6.2.6 Pseudo-identifiers

Let us suppose that PASCAL were to allow (as do many other languages) use of symbol '**return**' for exiting from a procedure body:

```
procedure I(···;P;···);
    ⋮
  begin
    ⋮
    return
    ⋮
  end;
```

Although '**return**' is evidently not a programmer-defined identifier, its meaning does depend on the context in which it is used. In such a language, procedure definitions may be regarded as (implicitly) *binding* symbol '**return**'. Furthermore, the usual rules of *scope* would apply. The scope of such a binding would be the procedure body, excluding any nested scopes of '**return**'. For example, in

```
procedure p;
  procedure q;
    begin {q}
      ⋮
      return
      ⋮
    end; {q}
  begin {p}
    ⋮
    return
    ⋮
  end; {p}
```

the first occurrence of '**return**' is for exiting from procedure *q*, and the second is for exiting from *p*, which is a scope with a "hole" in it.

Symbols such as '**return**' that have the binding and scope properties of identifiers but are not chosen by the programmer will be termed ***pseudo-***

identifiers. The main disadvantage of pseudo-identifiers is that, unlike normal identifiers, a scope hole cannot be avoided by a systematic change of identifier. For example, in the program fragment above, there would be no way to exit procedure *p* as a whole from within the body of procedure *q*.

6.2.7 Other Variations on Binding

In some languages an identifier occurrence may be both binding *and* applied. For example, in PL/1 an identifier can be declared to be 'EXTERNAL', as in

```
DECLARE (I) EXTERNAL···;
```

The identifier occurrence in such a declaration is an applied occurrence because the meaning of the identifier must be obtained from the "external" environment; but, it is also a binding occurrence with respect to the "internal" or local context.

Another variant is to have more than one binding occurrence associated with a single binding. For example, to allow use of procedure identifiers *before* their actual definitions, PASCAL permits a **forward** "declaration" for the procedure name, as in

```
procedure I(···;P;···); forward;
   ⋮
   {use of I allowed here}
   ⋮
procedure I; C;
   ⋮
```

where both uses of I stand for the same identifier. Both of these binding occurrences of the procedure name are associated with the *same* binding. Note that the parameter list must be omitted in the actual definition of the procedure.

Finally, it is possible for an identifier occurrence to be *neither* binding nor applied, that is to say, unnecessary. For example, the following illustrates the syntax required by some PASCAL processors for specifying procedural parameters:

```
function integral(function f(x:real) : real;
                  a,b,eps:real) : real;
   begin ··· end;
```

The occurrence of identifier '*x*' in parameter specifier '*x:real*' is neither

binding nor applied. The role of the specifier is to indicate the type and kind of parameter expected by procedural arguments *f* to *integral*; identifier '*x*' is irrelevant and could in principle be omitted.

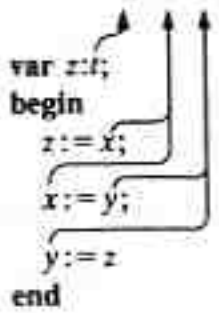
6.3 FREE IDENTIFIERS

An identifier is said to be ***free*** in (or "non-local" to, or "global" to) a construct if it has an applied occurrence in the construct that is not bound in the construct. These are known as the ***free occurrences*** of the identifier in the construct. For example, in PASCAL block

```
var z:t;
begin
  z:=x;
  x:=y;
  y:=z
end
```

(a) '*t*' is not free, because it has no occurrences in the block;
(b) '*z*' is not free, because both of its applied occurrences in the block are in the scope of a binding occurrence in the block;
(c) '*t*', '*x*', and '*y*' are free, because they have applied occurrences that are not bound in the block.

The free occurrences of identifiers in a construct may be thought of as having "binding arrows" that go outside the construct, as in



A construct must be interpreted relative to an environment which determines what is denoted by each of the free identifiers of the construct.

Note that freedom of an identifier and of identifier occurrences depends on the construct under consideration. For example, identifier *z* is free in expression '*z*' and free in command '*z* :=*x*' but not free in block

```
var z : t;
begin
  z := x;
  ⋮
end
```

because all of its applied occurrences are "captured" by the declaration. Also, note that it is possible for a free identifier of a construct to have binding occurrences as well as applied occurrences in the construct; some of the latter must be outside the scopes of all of the former.

EXERCISES

6.1 Which of the identifier occurrences in the following are binding and which are applied?

$$\int_a^b \sin(x)+\left(\int_c^d \cos(y)\,dy\right)dx$$

If possible, draw binding arrows from applied identifier occurrences to the occurrence that binds them. What are the free identifiers of this expression?

6.2 Why might a PASCAL programmer want to use a procedure name before its definition, and therefore need a **forward** declaration?

6.3 What is the disadvantage to program readers of not having the parameter list in the actual definition of a **forward**-declared procedure in PASCAL? Are formal parameter identifiers needed in **forward** "declarations"?

6.4 In PL/I, definitions may appear *anywhere* in a block, and not necessarily before the body of the block. What are the advantages and disadvantages of this flexibility?

6.5 Give an example showing that an identifier which is free in a construct may have binding occurrences in the construct.

6.6 An enthusiastic language designer proposes to extend his favorite language CATCHALL by allowing parameter list definitions of the form

```
param I=(⋯;P;⋯);
```

where P is a formal parameter. For example,

```
param p=(i:integer; var x:real);
  ⋮
procedure q(p); begin⋯i⋯x⋯end;
procedure r(p); begin⋯i⋯x⋯end;
```

would then be equivalent to

```
procedure q(i:integer; var x:real);
  begin⋯i⋯x⋯end;
procedure r(i:integer; var x:real);
  begin⋯i⋯x⋯end;
```

Why should he be discouraged?

7 PROCEDURAL ABSTRACTION

Procedural abstraction is provided in PASCAL by its **procedure** and **function** definition forms. The basic components of a procedural abstraction mechanism are:

(a) a formal parameter part (possibly empty) which contains binding occurrences of the formal parameter identifiers, and
(b) a body: a construct whose interpretation is deferred until the resulting procedure is invoked (by being supplied with appropriate arguments).

For example, in

```
procedure inc(var i:integer);
  begin i:=i+1 end;
```

the formal parameter part is

```
(var i:integer)
```

in which '*i*' has a binding occurrence, and the body is command

```
begin i:=i+1 end
```

In this chapter, the *body* component of procedure definitions and abstracts will be discussed in detail; issues relating to *parameters* will be considered in the following chapter.

7.1 COMMAND PROCEDURES

A **procedure** definition in PASCAL has the syntactic form

```
procedure I(···;P;···); C;
```

To focus attention on abstraction-related aspects, it is convenient to regard this as if it had the form

I=**procedure**(⋯;P;⋯); C;

where abstract

procedure(⋯;P;⋯); C

is an expression whose value is a ***command procedure***. This means that (a) the body, C, of the abstract defining it is a command, and that (b) the procedure it defines is invoked by executing an invocation that is itself a command. For example, the value of the abstract implicit in

procedure *inc*(**var** *i*:*t*);
begin *i*:=*i*+1 **end**;

may be invoked by a command such as

inc(*n*)

What then *is* a command procedure, as a semantic object? Consider the "incrementing" procedure defined above. Its effect when invoked is to increment the integer contained in its argument (a location). This means that the procedure may be regarded abstractly as a *function* mapping a location (its *explicit* argument) and a store (its *implicit* argument) to a new store. The new store is like the old one except that the integer in the location has been incremented.

In general, a command procedure in PASCAL (and most other languages) may be modelled by a function that maps a sequence of argument values and a store into a new store. The explicit arguments are obtained by evaluating the actual parameters of an invocation, and the implicit argument is the state of the store just after these evaluations. The store yielded by executing an invocation is the new store obtained by applying the procedure to its arguments.

Now we may describe precisely what procedure is defined by an abstract of the form

procedure(⋯;P;⋯); C

The value of the abstract is the command procedure that, when applied to a sequence of argument values and a store, yields the store obtained by executing the command C relative to the supplied store. Note that by using mathematical functions to model *what* procedures do, we have been able to "abstract" away from *how* these functions happen to be described by particular **procedure** definitions.

But an important question must still be answered: what *environment* is used for executing C? In other words, what do identifiers denote in C?

Consider abstract

procedure(**var** *i*:*t*);
var *temp*:*t*;
begin
temp:=*i*+*k*;
i:=*temp*
end

Identifier '*temp*' is bound *locally* within the body, so that there is no question about what it denotes there. Identifier '*i*' is a *formal parameter* of the abstract; parameter binding will be discussed in the next chapter, but it is evident that in this case '*i*' must be bound to the *l*-value of the actual parameter of the invocation. Finally, there are identifiers '*k*' and '*t*', which are *free* identifiers of the abstract. Where are free identifiers of abstracts or procedure definitions bound? This question is addressed in the next section.

7.2 FREE IDENTIFIERS OF ABSTRACTS

7.2.1 Static Binding

In PASCAL, free identifier occurrences are bound in the environment of the *abstract*. This is known as ***static*** binding, because the binding occurrence is determined "statically", that is to say, without executing the program; furthermore, the binding occurrence does not change during program execution.

For example, consider definitions

const *k*=3;
type *t*=0..99;
procedure *inc*(**var** *i*:*t*);
begin *i*:=*i*+*k* **end**;

The free identifiers of the **procedure** definition are, as before, '*k*' and '*t*'. The abstract is evaluated in an environment in which '*k*' denotes 3 and '*t*' denotes the subrange from 0 to 99. Therefore, in a language using static binding (such as PASCAL), these are the meanings of the free occurrences of these identifiers in the abstract, no matter what the environment of any *invocation* of the procedure. This may be depicted by binding arrows as follows:

const *k*=3;
type *t*=0..99;
procedure *inc*(**var** *i*:*t*);
begin *i*:=*i*+*k* **end**;

Note that free identifier '*k*' would be bound in the same way even if it denoted a location, rather than a number, as in

```
type (i)=0..99;
var (k):t;
procedure inc(var i:t);
  begin i:=i+k end;
```

In this case the store resulting from an invocation of the procedure depends on the contents in the store (at the time of invocation) of location *k* allocated in the declaration, but *not* on the meaning of identifier '*k*' in the environment of the invocation. In other words, the procedure has a *store* as an implicit argument, but the *environment* for the definition body is determined once and for all when the procedure is defined (except for parameter bindings).

7.2.2 Dynamic Binding

Another approach to free identifiers of abstracts is used in LISP and APL. In these languages, free identifier occurrences of abstracts are bound in the environments of *invocations* of the procedure. Consider the same **procedure** definition as the previous example, but now suppose that the procedure is invoked from an environment in which '*k*' has a different meaning:

```
procedure inc(var i:t);
  begin i:=i+k end
  ⋮
procedure q;
  const (k)=5;
  var n:t;
  begin
    ⋮
    inc(n);
    ⋮
  end;
```

If the environment of the *invocation* were used to determine the meanings of free identifiers of the **procedure** definition, then this invocation of the procedure would increment its argument by 5. Other invocations might have other effects on their arguments, depending on the environments of those invocations.

This approach is known as ***dynamic*** binding, because binding occurrences of free identifiers of abstracts can change during program execution. In languages that use dynamic binding, procedures are functions not only of the stores, but also of the environments of their invocations.

In languages like LISP and APL that are normally implemented by interpreters rather than compilers, dynamic binding is somewhat easier to implement than static binding. But its disadvantages are quite severe. With dynamic binding it is much more difficult for program readers to determine where identifier occurrences are bound and for language processors to verify type compatibility and identifier binding before program execution.

An even more serious disadvantage of dynamic binding is the vulnerability of identifiers accessible in the environment of an invocation. For example, consider the following:

```
var n,k:t;
begin
  ⋮
  p(n);
  ⋮
end
```

In PASCAL and in any language with static binding, execution of the invocation could not affect the contents of *k* because '*k*' is not an actual parameter and the definition of procedure *p* is not in the scope of the declaration of '*k*'. Therefore, the contents of this location cannot be modified or even examined by *p*. But in a language with *dynamic* binding, *k* may be accidentally or maliciously accessed or modified by *p* if '*k*' is a free identifier of the definition of *p*, and there is little that the programmer of the invocation can do to prevent this! With dynamic binding, locally defined or private information is vulnerable to errors or breaches of privacy in *other* program components, and this is quite unacceptable in large systems.

7.3 EXPRESSION PROCEDURES

By analogy with the properties of *command* procedures, it might be expected that (a) an ***expression procedure*** would be invoked by evaluating an invocation *expression*, and that (b) the body of an abstract defining an expression procedure would be an expression. The first of these holds in PASCAL and in most programming languages, and in some languages so does the second. For example, the syntax of a "statement function" definition in FORTRAN is

$$I(\cdots,P,\cdots)=E$$

where the body to the right of the '=' symbol is an expression. However, in PASCAL (b) apparently does not hold, because the syntax for the **function** definition is

function I(···;P;···):I; C;

where the body is a *command*. But conceptually the body of a **function** definition in PASCAL *is* an expression, because it is executed to yield a value. There is a special convention involving the name of the procedure that allows an execution of command C to yield a value.

Let us defer consideration of this somewhat confusing convention for the moment and regard

function (···;P;···):I; E

with expression E as its body as the general form of an expression procedure abstract. Then, the semantics of expression procedure invocation and abstraction may be described by analogy with the treatment of *command* procedures as follows:

(a) An expression procedure may be regarded as a (mathematical) function mapping a sequence of (explicit) argument values and a store (the implicit argument) into a value and a new store.
(b) An invocation expression invokes an expression procedure by applying it to the values of the actual parameter expressions and the current state of the store.
(c) The value and the store yielded by evaluating an invocation expression are those returned by the expression procedure invoked.
(d) The value of abstract

function(···;P;···):I; E

is the expression procedure that, when applied to a sequence of values and a store, yields the value and the store obtained by evaluating E relative to the supplied store.

The environment for the evaluation of E in (d) is established along the same lines as the environment for execution of the body of a command procedure abstract. In a language that uses static binding, free identifiers are bound in the environment of the abstract itself, and not the invocations. In a language that uses dynamic binding, expression procedures are also functionally dependent on the invocation environment.

For example, if abstract

```
function(var a,b:integer):integer;
  begin
    a:=sqr(a);
    b:=sqr(b)
  result a+b
```

is evaluated in an environment in which free identifiers '*integer*' and '*sqr*'

have their standard meanings, then the expression procedure defined by this abstract will return as its value the sum of the squares of the contents in the current store of its (explicit) arguments, and it will also have the side effect of squaring the contents of these locations. (To avoid the side effect, the "value" parameter form may be used.) In a language that uses dynamic binding, the value and effect of the invocation would also depend on the meanings of '*integer*' and '*sqr*' in the environment of the invocation.

Let us now survey the conventions employed in programming languages for returning values when the bodies of expression procedure abstracts are *commands* (syntactically). In PASCAL and ALGOL 60, a location is allocated when an expression procedure is invoked. If the name of the procedure is used as an assignment target in the body of the abstract, it denotes this location, and its contents after executing the body becomes the value returned by the procedure. In contexts other than as assignment targets, the procedure name denotes the procedure itself (recursively), so that the location for the return value is effectively "write-only".

There are two ways this overloading might have been avoided: (a) use a pseudo-identifier like **result** (rather than the name of the procedure) to denote the location for the return value, or (b) allow the programmer to name the location, as in

```
function f(n:t)result:t;
  begin
    ⋮
    result:=···;
    ⋮
  end;
```

Another approach is to avoid a special location and use a sequencer. For example, sequencer

RETURN(E)

is used in PL/I to transfer control to the end of the abstract (and then back to the invocation expression) with the value of E.

It should be pointed out that none of these conventions has essentially to do with procedural abstraction: all could equally well be adapted for use in *arbitrary* expressions whenever it is desired to precede sub-expression evaluation with sub-command execution, as in expression form

begin C **result** E

Generally, language designers have restricted such constructs or conventions to expression procedure abstracts or definitions in order to limit sources of side effects.

7.4 THE PRINCIPLE OF ABSTRACTION AND selector DEFINITIONS

We have seen that, aside from some minor syntactic idiosyncracies, the two kinds of procedure in PASCAL are very closely analogous: a ——— procedure, where "———" is either "command" or "expression", is invoked by a ——— invocation and defined by an abstract whose body is (at least conceptually), a ———. The question now arises: can these concepts be applied to *other* kinds of "———"? The answer is yes: *any* semantically meaningful syntactic class "———" can in principle be used as the body of a form of abstract, and the resulting ——— procedure may be invoked in an invocation that is a ———. This is known as the ***principle of abstraction***. Of course, this is not to suggest that all such procedures are necessarily useful or desirable in programming languages, but it is important for a language designer to be aware that they are *possible*.

A simple first application of the principle of abstraction is the concept of abstraction from *l-expressions*. It may be recalled that an expression is an *l*-expression if it has an *l*-value and so may be used as the target of assignments. So, let us consider ***l-expression procedures***, which must be defined by abstracts whose bodies are *l*-expressions and must be invoked by invocations which are *l*-expressions.

A possible PASCAL-like syntax for definitions of such procedures is

```
selector I(···;P;···):I;
  L;
```

By analogy with ordinary expression procedures, such a procedure returns the *l*-value of L, where the evaluation is relative to the environment of the definition (extended by parameter bindings) and a store from the invocation. For example, consider

```
type stack=record
             a:array[1..n]of t;
             p:0..n
           end
  ⋮
selector top(var s:stack):t;
  s.a[s.p];
  ⋮
```

The **selector** definition defines an *l*-expression procedure *top* that may be invoked in an *l*-expression as follows:

```
top(d):=···
```


where *d* is of type *stack*. This procedure allows the "top of a stack" to be *updated* (as well as fetched). This may be compared to use of the built-in selector notation

```
L↑
```

in PASCAL when the value of L is a file.

Further applications of the principle of abstraction with other syntactic classes will be discussed in subsequent chapters.

EXERCISES

7.1 Give outlines of programs in which

(a) two occurrences of the same **procedure** definition define distinct (i.e., non-equivalent) procedures, and

(b) a single occurrence of a **procedure** definition is interpreted more than once and the two interpretations define distinct procedures.

Assume that the programming language uses static binding.

7.2 In a language with dynamic binding, where is a free identifier of a procedure definition bound if it does not have a binding occurrence in the context of the invocation?

7.3 What should be the *r*-value of an invocation of an *l*-expression procedure?

7.4 What restrictions must be imposed on **selector** definitions if locations allocated by local declarations in the body are disposed of on exit from it?

7.5 Compare *l*-expression procedures to ordinary expression procedures which return pointer values.

7.6 Is it accurate to describe a language with dynamic binding as one in which an applied identifier occurrence is always bound by the *most recent* interpretation of a binding of that identifier?

7.7 Is it accurate to describe a language with static binding as one in which an applied identifier occurrence is always bound by the *most recent* interpretation of a statically enclosing binding of that identifier? *Hint:* consider the occurrence of '*b*' in the definition of '*outputb*' in the following PASCAL block:

```
procedure p(b : Boolean; procedure q);
  procedure outputb;
    begin write (b) end;
  begin {p}
    if b then p (false, outputb) else
    begin
      outputb;
      q
    end
  end; {p}
procedure dummy;
  begin {null} end; {dummy}
begin p(true, dummy) end
```

BIBLIOGRAPHIC NOTES

Abstraction principles have long been important in set theory and logic [7.1, 7.2]. The principle of abstraction for programming languages described here was suggested by Tennent [7.3].

7.1 Church, A. *The Calculi of Lambda Conversion*, Princeton Univ. Press, Princeton, N.J. (1941).

7.2 Quine, W. V. *Set Theory and its Logic*, Harvard Univ. Press, Cambridge, Mass. (1963).

7.3 Tennent, R. D. "Language design methods based on semantic principles", *Acta Informatica*, **8**, 97–112 (1977).

8 PARAMETERS

The basic concepts involved in parameterization are very straightforward: when a programmer-defined procedure is invoked, the body of its definition is interpreted in an environment which binds the formal parameter identifiers to values which are in some way associated with the corresponding *actual* parameters of the invocation. The great variety of parameter mechanisms in programming languages arises primarily from the many answers possible to two questions: *when* and *how* are the actual parameters of an invocation evaluated?

8.1 PARAMETERS IN PASCAL

The language PASCAL has four formal parameter forms:

(a) I:I′
(b) **var** I:I′
(c) **procedure** I(⋯;Q_i;⋯)
(d) **function** I(⋯;Q_i;⋯):I′

In these, I is the bound formal parameter identifier and I′ is an applied occurrence of a type identifier. (For some processors, the list (⋯;Q_i;⋯) of formal parameter specifiers is omitted.)

In PASCAL, actual parameters are always evaluated *before* the procedure is invoked, but different modes of evaluation are used for the various forms of formal parameter. With the ***value*** parameter form (a) the formal parameter identifier is bound to new storage that is initialized to the *r*-value of the actual parameter. Therefore, assignments in the procedure body having the formal identifier as their target update the "private" storage, rather than the *l*-value of the actual parameter.

For the **var** parameter form, the actual parameter must be an

l-expression, and the formal parameter identifier is bound to its *l*-value. This means that assignments having the formal identifier as target update the *l*-value of the actual parameter.

For a **procedure** or **function** parameter, the actual parameter must be the name of a command or expression procedure, respectively, and the formal parameter identifier is bound to that procedure.

Type compatibility constraints for parameters in PASCAL will be discussed in Chapter 12.

8.2 NAME PARAMETERS IN ALGOL 60

The *name* parameter mechanism is described in the ALGOL 60 Report as follows:

> The effect of [a procedure invocation] will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure [invocation]:
>
> Any formal [name parameter identifier] is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing the latter in parentheses if it is an expression but not [an *l*-expression]. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.
>
> Finally, the procedure body, modified as above, is inserted in place of the procedure [invocation] and executed. If the procedure is called from a place outside the scope of any [free identifier] of the procedure body, the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement will be avoided through suitable systematic changes of the latter identifiers.

This algorithm is evidently not a denotational semantic description; that is, it does not specify the meaning of invocations in terms of the meanings of their immediate constituents. In particular, in order to carry out the substitutions, it is necessary to know *how* a procedure is defined.

To illustrate the substitution process, consider PASCAL **procedure** definition


```
procedure square(var i:integer);
  var t:integer;
  begin
    t:=sqr(i);
    i:=t
  end;
```

Now suppose that formal parameter '**var** *i*:*integer*' were to be treated as a name parameter. Then the description above says that an invocation such as

```
square(a[j])
```

is to have the effect of

```
var t:integer;
begin
  t:=sqr(a[j]);
  a[j]:=t
end
```

provided that '*integer*' and '*sqr*', the free identifiers of the procedure definition, have the same meaning in this environment.

Note that each use of a formal name parameter in the definition body results in an evaluation of the corresponding actual parameter expression *during* interpretation of the procedure body. This can be very inefficient and may have surprising effects because even the *l*-value of the identifier can change dynamically.

An example that illustrates the power of name parameters is

```
function sum(a,b:integer; var i:integer; f:real):real;
  var s:real;
  begin
    s:=0;
    for i:=a to b do
      s:=s+f;
    sum:=s
  end;
```

If '*f*' were a name parameter, then in the context of

```
var j:1..n;
    p,q:array[1..n]of real;
```

invocation

```
sum(1,n,j,p[j]*q[j])
```

would determine the inner product of "vectors" p and q, much like

$$\sum_{j=1}^{n} p_j \times q_j$$

Of course it would be clearer and no less convenient to express this using a **function** parameter as follows:

```
function sum(a,b:integer; function f(i:integer):real):real;
  var s:real;
      i:integer;
  begin
    s:=0;
    for i:=a to b do
      s:=s+f(i);
    sum:=s
  end;
```

A typical invocation might then be

```
sum(1,n,function(j:integer):real; p[j]*q[j])
```

where abstract

```
function(j:integer):real; p[j]*q[j]
```

has been used as an actual parameter to express an "anonymous" procedure.

Another use for name parameters is to allow a programmer to define a procedure that can choose to avoid evaluating an actual parameter. For example, the procedure defined by

```
function andthen(x,y:Boolean):Boolean;
  begin
    if x then andthen:=y else andthen:=false
  end;
```

would provide the sequential conjunction of Section 5.3.2 if 'y' were a name parameter.

To see the reasons for the two kinds of "systematic change of identifiers" mentioned in the rules quoted above, consider invocation

```
square(t)
```

in the context of

```
var t:integer;
function sqr(x:integer):integer;
  begin sqr:=··· end;
```


Without changes of identifiers the invocation would be replaced by

```
var t:integer;
begin
  t:=sqr(t);
  t:=t
end
```

which is unsatisfactory for two reasons.

(a) There is a conflict between the occurrence of '*t*' in the actual parameter and local identifier '*t*' of the **procedure** definition body into which the actual parameter is being substituted.

(b) There is a conflict between the occurrence of '*sqr*' in the **procedure** definition body and local identifier '*sqr*' of the context into which the body is being substituted.

The given rules solve these problems by requiring

(a) replacement of all occurrences of '*t*' in the body of the **procedure** definition by any other identifier that is *not* free in the actual parameter, and

(b) replacement of all occurrences of '*sqr*' in the context of the invocation by any other identifier that is *not* free in the modified body of the **procedure** definition.

For example, the result might be

```
var t:integer;
function newsqr(x:integer):integer;
  begin newsqr:=··· end;
    ⋮
  var newt:integer;
  begin
    newt:=sqr(t);
    t:=newt
  end;
    ⋮
```

where all occurrences of '*sqr*' in the ellipses are to be replaced by '*newsqr*'.

There is a much simpler way to describe the semantics of name parameters. An actual name parameter may be treated in the same way as an abstract with a null parameter list. The formal parameter identifier would

then be bound to a *procedure* which would be invoked whenever the identifier was evaluated.

For example,

square(*a*[*j*])

may be regarded as if it were

square(**selector**:*integer*; *a*[*j*])

The actual parameter of *square* is abstract

selector:*integer*; *a*[*j*]

which defines an *l*-expression procedure (see Section 7.4) with no (explicit) arguments. In this way, the meaning of a procedure invocation involving name parameters can be explained solely in terms of the meanings of its immediate constituents without knowing how the procedure is defined.

8.3 OTHER PARAMETER MECHANISMS

PL/I and many FORTRAN processors use a parameter mechanism which combines the effects of the value and **var** forms in PASCAL: the formal parameter identifier is bound to the *l*-value of the actual parameter *if* this is an *l*-expression which has the same type as the formal parameter, and to new storage that is initialized to the *r*-value of the actual parameter otherwise. However, note that any mismatch between formal and actual parameters can lead to unexpected results.

In some implementations of FORTRAN, it is more efficient or convenient to use another approach. A formal parameter identifier is bound to new storage which is initialized to the *r*-value of the actual parameter (i.e., like value parameters in PASCAL), but *after* execution of the procedure body the *l*-value of the actual parameter (if it is an *l*-expression) is then updated by the final contents of the new storage. This is known as the ***value-result*** (or "copy-restore") parameter mechanism.

Actual parameters can also be evaluated before program execution if they are static or type expressions. This possibility will be discussed in Chapter 12. Another approach is to allow an actual parameter to be omitted. The corresponding formal parameter identifier would then be bound to a "default" value specified either by the programmer in the procedure definition, or by the language. A few additional parameter mechanisms will be discussed in the exercises.

8.4 PARAMETER LISTS

Procedures often have more than one (explicit) parameter. This is usually expressed, as in PASCAL, by allowing *lists* of formal and actual parameters. The correspondences between the formal and actual parameters are established by their respective positions in these lists.

If a parameter list is long, such positional correspondence tends to be error-prone and difficult to read, and some languages allow actual parameter expressions to be labelled by the corresponding formal parameter identifier, as in:

```
function integral(function f(x:real):real; a,b,eps:real):real;
  begin
    ⋮
    integral:=⋯;
  end;
begin
    ⋮
  integral(a: −1.0, b: 1.0, eps: 1E−8, f: fct)
    ⋮
end
```

Note that the order of the actual parameters need not be the same as the order of the formal parameters.

Procedures may have a *null* list of formal parameters. In PASCAL and many other languages, such a procedure is invoked by using the name of the procedure by itself. But in some languages it is necessary to distinguish between an invocation with a null actual parameter list and a reference to the procedure itself (or to an identifier bound by default). In FORTRAN, for example, if F is an expression procedure with no formal parameters, it must be invoked by expression 'F()'.

EXERCISES

8.1 When may storage allocated for a value parameter be disposed of in PASCAL without creating dangling references?

8.2 Why do the substitution rules for name parameters require enclosing of actual parameter expressions in parentheses?

8.3 Would substitution rules like those for name parameters be simpler for a language that uses *dynamic* binding for free identifiers of abstracts?

8.4 Devise an example to show that the value-result parameter mechanism is *not* equivalent to the **var** parameter mechanism in PASCAL. Assume that the *l*-value of the actual parameter is evaluated *before* invoking the procedure.

8.5 Is it possible in PASCAL to access a location via both an identifier and a pointer in the same context?

8.6 Define a context in PASCAL such that the three occurrences of identifier '*f*' in

$$f:=g(f,f)$$

have different meanings.

8.7 Consider definitions

```
type matrix=array[1..n,1..n]of real;
  ⋮
procedure transpose(var a:matrix; b:matrix);
  var i,j:1..n;
  begin
    for i:=1 to n do
      for j:=1 to n do
        a[i,j]:=b[j,i]
  end;
```

(a) Suppose that the PL/I parameter mechanism were used for *a* and *b*; would invocation

transpose(*q*,*q*)

have the effect that the programmer probably intended?

(b) Why do PL/I programmers sometimes enclose actual parameter expressions in parentheses?

(c) Many PASCAL programmers use a **var** parameter instead of a value parameter to avoid unnecessary copying of "large" structures. Why is this a dangerous practice?

8.8 Consider definition

```
procedure swap(var x,y:integer);
  var z:integer;
  begin
    z:=x; x:=y; y:=z
  end;
```

and suppose that '*x*' and '*y*' were name parameters.

(a) If *a*[1]=2, *a*[2]=5, and *i*=1, what would be the effect of '*swap*(*a*[*i*],*i*)'

(b) What would be the effect of '*swap*(*i*,*a*[*i*])' in the same circumstances?

(c) In the context of definitions

```
type ind=1..n;
var a:array[ind]of integer;
function min(i:ind):ind;
  var m,j:ind;
  begin
    m:=i;
    for j:=i+1 to n do
      if a[j]<a[m] then m:=j;
    min:=m
  end;
```

the following is intended to sort array *a* by selecting successive minima:

```
for i:=1 to n −1 do
  swap(a[i],a[min(i)])
```

What would in fact be the overall effect on *a*? Explain.

8.9 A book on the design of correct programs claims that in PASCAL

```
while E do C
```

is equivalent to an invocation of the procedure defined by

```
procedure whiledo(e:Boolean; procedure c);
  begin
    if e then
    begin c; whiledo(e,c) end
  end;
```

Why is this incorrect? Suggest a suitable correction to the **procedure** definition.

8.10 The ***result*** parameter mechanism in ALGOL W may be described as follows: the formal parameter identifier is bound to (uninitialized) new storage; *after* execution of the procedure body, the *l*-value of the actual parameter (which must be an *l*-expression) is updated to the final contents of the new storage. Give examples to show that in general this is not equivalent to either the **var** parameter in PASCAL or the name parameter in ALGOL 60.

8.11 The ***lazy evaluation*** parameter mechanism defers evaluation of the actual parameter until the value of the formal parameter is first needed. When (and if) this evaluation takes place, the resulting value is saved for any subsequent evaluations of the formal parameter. Under what conditions would this be equivalent to the name parameter mechanism? What advantage does it have over the name parameter?

PROJECTS

8.1 Pre-defined procedures *read* and *write* in PASCAL illustrate the convenience of variable-length actual parameter lists. Design extensions to PASCAL that would allow programmer-defined procedures to have variable-length parameter lists.

8.2 Devise syntactic (i.e., statically verifiable) constraints that would be sufficient to ensure that a **var** parameter could be implemented like a value-result parameter and the implementation would always produce correct results.

8.3 Devise syntactic constraints that would be sufficient to ensure that a value parameter could be implemented like a **var** parameter and the implementation would always produce correct results.

8.4 Devise syntactic constraints that would be sufficient to ensure that an expression procedure cannot have side effects on storage.

BIBLIOGRAPHIC NOTES

The history of attempts to give a correct definition of substitution between contexts is traced in Church [8.1, pp. 289–90]. The parameter mechanism called "lazy evaluation" by Henderson and Morris [8.2] was invented by Wadsworth [8.4], who termed it "call by need". For projects 8.2 and 8.3, see Hoare [8.3].

8.1 Church, A. *Introduction to Logic*, Vol. 1, Princeton Univ. Press, Princeton, N.J. (1956).

8.2 Henderson, P. and J. H. Morris. "A lazy evaluator", *Conf. Record of the 3rd ACM Symposium on Principles of Programming Languages*, 95–103, ACM, New York (1976).

8.3 Hoare, C. A. R. "Procedures and parameters: an axiomatic approach", in *Symposium on Semantics of Algorithmic Languages* (ed., E. Engeler), *Lecture Notes in Mathematics*, **188**, 102–16, Springer, Berlin (1971).

8.4 Wadsworth, C. P. *Semantics and Pragmatics of the Lambda-calculus*, D.Phil. thesis, University of Oxford (1971).

9 DEFINITIONS AND BLOCKS

The basic concepts underlying definitions and blocks are illustrated by the simple PASCAL block

```
const i = −j;
begin
  write(i)
end
```

The occurrence of identifier '*i*' on the left-hand side of the definition is a binding occurrence. The effect of interpreting the definition is to create a new environment that binds identifier '*i*' to the value of expression '$-j$' for execution of command '*write*(*i*)'.

In this chapter, we shall be studying the relationships between definitions and parameters, the interpretation of recursive definitions, various block constructions, composite definitions, and definition procedures.

9.1 THE PRINCIPLE OF CORRESPONDENCE

Compare the following two PASCAL fragments:

(a)
```
var i : integer;
begin
  i := −j;
  write(i)
end
```

(b)
```
procedure p(i : integer);
  begin
    write(i)
  end;
begin
  p(−j)
end
```

In (a), identifier '*i*' is declared by a **var** declaration. Therefore, it is bound to a new storage location and command '*write*(*i*)' is executed in the scope of this binding, after the location is initialized by the value of expression '$-j$'.

In (b), a procedure *p* is defined and then immediately invoked. Because its formal parameter is a value parameter, identifier '*i*' will be bound to a new storage location. Then '*write*(*i*)' will be executed in the scope of this binding after the location is initialized by the value of actual parameter '$-j$'.

It is evident that the two fragments are equivalent. This is not a coincidence; in general, for *any* identifiers I_1 and I_2, expression E, and command C,

```
var I1 : I2;
begin
  I1 := E;
  C
end
```

and

```
procedure I0(I1 : I2);
  begin
    C
  end;
begin
  I0(E)
end
```

are equivalent in PASCAL (provided that procedure name I_0 is chosen to be any identifier that is not free in C or E). Note that

(i) the declared identifier and its type specification in fragment (a) correspond respectively to the formal parameter identifier and its type specification in (b);
(ii) the body of the block (except for the initializing assignment) in (a) corresponds to the body of the procedure in (b); and
(iii) the initializing expression in (a) corresponds to the actual parameter in (b).

If an initializing expression were part of the **var** declaration in PASCAL, the correspondences would be even more direct:


```
var I1 : I2 = E;
begin
  C
end
```

```
procedure I0(I1 : I2);
  begin
    C
  end;
begin I0(E) end
```

Similar equivalences and correspondences can be established for other forms of parameters and definitions. For example, in ALGOL 68, blocks

```
begin
  T I1=E;
  C
end
```

```
begin
  proc I0=(T I1)void: C;
  I0(E)
end
```

are equivalent (provided that procedure name I_0 is not free in E or C). For example,

```
begin
  int i=−j;
  write(i)
end
```

is equivalent to

```
begin
  proc p=(int i)void: write(i);
  p(−j)
end
```

The parameter mechanism here binds '*i*' directly to the value of the actual parameter; that is, '*i*' denotes an integer, rather than an integer-containing location, and is *not* subject to assignments. Similarly, definition

```
int i=−j
```

binds '*i*' directly to the value of the right-hand side expression '$-j$'. This is just like a **const** definition in PASCAL, but without the restriction that the expression be statically evaluable.

It is not surprising that programming languages have such correspondences: the underlying semantic notions for both parameter and definition mechanisms are simply *expression evaluation* (of an actual parameter or the right-hand side of a definition) and *identifier binding* (of a formal parameter or the left-hand side of a definition.) Therefore all the discussion of parameters in Chapter 8 is also applicable to definitions. For any parameter mechanism, an analogous definition mechanism is possible, and *vice versa*. This is known as the ***principle of correspondence***.

Like the principle of abstraction, the principle of correspondence can be useful to a language designer by pointing out possible inconsistencies and deficiencies. For example, in PASCAL there is no definition mechanism that is the exact analog of the **var** parameter mechanism. (We have already seen that the **var** declaration corresponds fairly closely to the value parameter.) The closest is the rather specialized **with** construction which can only be used for binding record field names to the corresponding components of a record. Otherwise, if it is desired to establish a local name for storage in a PASCAL program, this must be done by defining a procedure with a **var** parameter, and this may be inconvenient or inefficient.

As another example, construct

```
loc int I:=E
```

in ALGOL 68 is a declaration that binds I to new storage and initializes it to the value of E, so that it is similar to the value parameter and the **var** declaration in PASCAL. But there is no analogous *parameter* mechanism in ALGOL 68. An equivalent effect can be achieved either by using an additional local declaration within the procedure, as in

```
proc I0
  =(int I')void:
     begin
       loc int I:=I';
         ⋮
     end;
  ⋮
I0(E)
```

or by putting the storage allocation in the *actual* parameter, as follows:


```
proc I0
  =(ref int I)void:
     begin
       ⋮
     end;
  ⋮
I0(loc int:=E)
```

But neither of these is as convenient or as efficient as the value parameter in PASCAL.

We shall see other applications of the principle of correspondence in later chapters.

9.2 RECURSIVE DEFINITIONS

9.2.1 Basic Concepts

A form of definition will be termed ***recursive*** if free occurrences of the defined identifier on its right-hand side are to denote the value it defines; i.e., the scope of the binding includes the definition. For example, all **procedure** definitions are interpreted recursively in PASCAL, so that in a definition of the form

procedure I(⋯;P;⋯); C

free occurrences of I denote the procedure value it defines. Some languages (such as FORTRAN) do not provide recursive definition forms in order to simplify implementation.

If a **procedure** definition were not interpreted recursively, then the usual rules of scope would apply to occurrences of the name of the procedure in its definition. For example, in a language with static binding, these occurrences would be bound in the environment external to the **procedure** definition, like other free identifiers. This non-recursive interpretation of **procedure** definitions is useful in some circumstances, and some programming languages allow a programmer to specify whether a recursive or a non-recursive interpretation is intended.

*9.2.2 Semantics of Recursive Definitions

Let D stand for definition

```
procedure I;
  begin C end;
```

If D is to be interpreted *recursively*, then its meaning may be specified as discussed in Chapter 7, but with the following crucial difference: the environment for executing C must already include a binding of I to the procedure being defined. The circularity here may be avoided by a limit construction, as in Sections 3.3.2 and 5.3.4.

A convenient way of constructing a sequence of better and better approximations to the meaning of D (interpreted recursively) is suggested by the requirement that D be equivalent to

```
procedure I;
  D                                    (1)
  begin C end;
```

Let D_0 be any definition that defines I to be a parameter-less procedure whose invocations *never* terminate. For example, D_0 might be

```
procedure I;
  begin
    while 0<1 do {null}
  end;
```

Then, for $i \geq 0$, let D_{i+1} be definition

```
procedure I;
  D_i
  begin C end;
```

Note that each D_{i+1} can be interpreted non-recursively (as in Chapter 7), because any free occurrences of I in C will be bound by the *local* definition D_i. Also, notice that D_{i+1} is definition (1) above, with D replaced by D_i.

It is evident that the meanings of D_0, D_1, D_2, ··· are better and better approximations to the intended meaning of D. Each D_i defines a procedure which fails to terminate if the depth of self-activation reaches *i*, but has the same effect as the procedure defined by D whenever its invocation terminates. The meaning of D interpreted recursively may therefore be specified as being the *limit* of this sequence of approximate meanings.

9.3 THE PRINCIPLE OF QUALIFICATION

The principle of correspondence establishes the relationship between *parameters* of abstracts and *definitions*. Similarly, there is a relationship between the *bodies* of abstracts and the *bodies* of blocks, that is to say, the constructs prefixed or "qualified" by definitions. For example, command C in procedure definition

```
procedure p(I : I');
  begin
    C
  end;
```

corresponds to command C in block

```
var I : I';
begin
  C
end
```

In both cases, C is qualified by a local binding of I to new storage.

The principle of abstraction tells us that the body of an abstract does not have to be a command: it can in principle be in *any* semantically meaningful syntactic class. So, it may be concluded that the body of a *block* may similarly be in any meaningful syntactic class. This is known as the ***principle of qualification***.

In a conventional *command* block in PASCAL

```
D
begin
  C
end
```

command C is the block body. An example of an *expression* block is the form

```
begin
  D;
  E
end
```

in ALGOL 68. It is also possible to have *definition* blocks; this is one of the definition structures that will be discussed in Section 9.5.

9.4 OTHER FORMS OF BLOCK

The blocks that have been described so far have consisted of a block body composed with a definition part, as in

```
D
begin C end
```

In this section, some other block-like constructions in programming languages will be described.

One example is command

```
with L do C
```

in PASCAL, in which there are no *explicit* definitions. Nonetheless, command C is executed in the scope of implicit bindings of identifiers to storage.

Another example of a block-like construction is the **for** iteration in some languages (including ALGOL W and ALGOL 68). The body is repeatedly executed in an environment in which the count identifier is *bound* to an integer. In PASCAL, the identifier must already denote a location which is *updated* before every execution of the body. The advantages of the block approach are that (a) the identifier can easily be protected against assignments to it during execution of the body; (b) there is no need to specify a value for the identifier after termination of the iteration; and (c) the programmer does not have to make a separate declaration of the identifier.

In ALGOL 68 there is a block-like *selective* construct that is used for discriminating values of expressions having **union** types. For example, if identifier '*ic*' has type **union(int,char)**, then the following construction will select execution of C_1 if the current value of '*ic*' is an integer, and C_2 if it is a character:

```
case ic of
  (int i): C1,
  (char c): C2
esac
```

Furthermore, execution of either C_1 or C_2 will be qualified by a binding of either '*i*' or '*c*', respectively, to this value.

Finally, a labelled command creates a local binding of the label, so that the part of the program qualified by such a binding is a form of block. Labels and their scope will be discussed in Chapter 10.

9.5 DEFINITION STRUCTURES

In general, a block in PASCAL consists of a block body prefixed by a *sequence* of definitions:

```
D1
D2
 ⋮
Dn
begin
  C
end
```

We shall see that it is desirable to treat the definition part of the block

```
D1
D2
 ⋮
Dn
```

as a ***definition structure*** whose immediate constituents are definitions $D_1, D_2, \cdots, D_n$.

By making a small adjustment in our view of the meaning of a definition, we can treat both simple definitions and structured definitions like this as being in the same syntactic class. From now on, we shall regard a definition as binding *any* number of identifiers. The syntactic class of definitions will then include both the simple definitions considered earlier, which bind *single* identifiers, and definition structures (such as the definition sequence in PASCAL) which may bind *many*. This is analogous to the way that the syntactic class of *commands* includes both simple commands (such as the assignment) and control structures (such as sequential composition $C_1 ; C_2$).

Our task now is to consider how the meanings of definition structures might be composed out of the meanings of simpler constituents.

9.5.1 Definition Structuring in PASCAL

The basic scope rule in PASCAL is that the scope of an identifier defined in a block is the whole block, including all the definitions in the block (but excluding, as usual, nested scopes of the same identifier). But, this simple rule must be hedged and qualified in some respects.

It is unreasonable to require **const** definitions to be interpreted recursively. For example,

```
const i=i;
```

should be syntactically invalid. One way to prevent such circularities is to require applied occurrences of defined identifiers to *follow* their definitions. This is also convenient for language processors. But this requirement excludes useful circular definitions such as

```
type node= record
             info : integer;
             link : ↑node
           end;
```

and

```
procedure p;
begin ··· p; ··· end;
```

On the other hand, a definition such as

```
type sequence=record case isnull : Boolean of
                true : ();
                false : (first : t; rest : sequence)
              end
```

should be illegal, for the pragmatic reasons discussed in Section 3.3.1.

When all of these pragmatic considerations and special cases are taken into account, the definition composition mechanism in PASCAL is quite complex and difficult to describe accurately. In the following sub-sections, we shall study a number of systematic ways of composing definitions out of simpler constituents.

9.5.2 Mutually Recursive Definitions

If a simple definition D is interpreted recursively, then the scope of the binding includes D. There is an obvious generalization of this to *several* definitions. Definitions $D_1, D_2, \cdots, D_n$ are said to be ***mutually recursive*** if the scope of any of the identifiers they bind includes *all* of the D_i. For example, each of the following **procedure** definitions contains applied occurrences of both of the defined identifiers, '*p*' and '*q*':


```
procedure p;
  begin···p···q···end;
procedure q;
  begin···p···q···end;
```

9.5.3 Sequential Definitions

A "definition before use" requirement may be achieved by structuring definitions ***sequentially***, so that each constituent definition is interpreted in the scope of the *preceding* definitions.

For example, suppose that

```
const i=j;
      j=-i;
```

is interpreted as a sequential definition structure. Then, if '*j*' denotes 3 in the original environment, the effect of the definition sequence is to bind '*i*' to 3 and then '*j*' to −3.

9.5.4 Simultaneous Definitions

Another definition composition mechanism is suggested by the principle of correspondence; consider block

```
procedure I0(I1 : I1'; I2 : I2');
  begin C end;
begin
  I0(E1,E2)
end
```

in PASCAL. If we allow initialized **var** declarations and I_0 is not free in C, E_1, or E_2, then the above would be equivalent to

```
var I1 : I1'=E1;
    I2 : I2'=E2;
begin
  C
end
```

if *both* expressions E_1 and E_2 were evaluated in the *original* environment. Therefore, E_2 should not be in the scope of the binding of I_1 set up by the first definition. This composition mechanism will be termed ***simultaneous*** definition.

9.5.5 Definition Blocks

In all of the definition structures considered so far, the names bound in any of the constituent definitions are also bound in the resulting environment. Another way of composing definitions is to have one definition qualify *only* the other. This is the ***definition block*** mentioned in Section 9.3 as an application of the principle of qualification.

Let us extend PASCAL to include such a construct, expressed in the form

private
D_1
within
D_2

The environment it produces is obtained by adding the bindings of D_2 to the original environment; however, the environment used for interpreting D_2 is obtained from the original environment by adding the bindings of D_1.

For example, definition block

```
private
  var a : integer = seed mod d;
within
  procedure draw(var x : real);
    begin
      a :=a*m mod d;
      x :=a/d
    end;
```

defines a procedure for generating random numbers. Storage location *a* is allocated before the procedure is defined and is then accessible *only* from the procedure definition, because the scope of the first definition is restricted to the second definition. Therefore this definition structure is different from the conventional

```
var a : integer;
procedure draw(var x : real);
  begin ··· end;
```

Here, the scope of '*a*' would also include the "users" of procedure *draw*, and this would not be desirable. On the other hand, it would not be possible to place the declaration of '*a*' within the procedure definition, as follows:


```
procedure draw(var x : real);
  var a : integer;
  begin ··· end;
```

because then a new location would be allocated for each *invocation* of the procedure.

9.5.6 Commands in Definitions

Some languages allow initializing expressions in declarations. For example, the ALGOL 68 declaration

loc T I :=E

has the effect of making the value of E the initial contents of the new storage to which identifier I is bound. However, for "large" storage structures, it would be more appropriate to initialize *selectively*. For example, a file in PASCAL is initialized by applying procedure *rewrite* to it, rather than by assignment of an empty file.

One way that such command-oriented initialization can be provided in definitions is to introduce a structure that composes a definition with a *command*. For example, suppose that PASCAL were extended to include a definition structure

```
D
initial
  C
end
```

Command C is executed relative to the new environment (and new store) created by interpreting definition D. Then a file declaration might be composed with an initialization as follows:

```
var f : file of integer;
initial rewrite(f) end
```

Note that this definition structure is syntactically similar to the command block

```
D
begin
  C
end
```

in that the immediate constituents are also a definition and a command; however, the block is a form of *command*, and the identifier bindings of D are *local* to it, whereas the initializing definition structure creates identifier bindings for qualification of constructs which are external to it.

In some applications it is also desirable to allow a programmer to specify a ***finalization*** in a definition; that is to say, computation that is to occur *after* interpretation of the construct qualified by the definition. For example, the following definition structure would check whether the file is non-empty after its use:

```
var f : file of integer;
initial rewrite(f) end
final
  if not eof(f) then leftover(f)
end
```

9.5.7 Discussion

We have seen in the preceding sub-sections that there are several ways of usefully combining simple definitions and commands into definition structures. The composition principles are of course applicable to composite forms as well as to simple ones. For example,

```
private
  const m=32;
  var a : array[1..m]of char;
      p : 0..m;
  initial p:=0 end
  final if p ≠ 0 then error end
within
  procedure push;
    begin p:=p+1 end;
  procedure pop;
    begin p:=p-1 end;
  selector top : char;
    a[p];
```

is a definition structure that defines procedures for using a (bounded) "stack" of characters. A sequence of simple definitions composed with initialization and finalization commands is private to a set of mutually recursive procedure definitions.

Note that in a PASCAL program it would be difficult to bring together these closely related definitions and commands, because of rules on definition ordering and the absence of initialization and finalization mechanisms in definitions. Furthermore, PASCAL does not provide a way to make definitions private to other definitions, so that the representation of the stack (*a* and *p*) would typically be accessible to the part of the program that *uses* the stack procedures *push*, *pop*, and *top*. This would be a serious breach of the principles of program modularity. It would allow the stack discipline to be violated by direct access to the representation. Also, it would be unsafe to modify the representation for the stack without examining all of the code that uses it.

Surprisingly, no widely used programming language has incorporated these notions of definition structuring, though they were developed as long ago as the early 1960s. The usual approach has been to try to find a single "universal" definition structuring mechanism. This is as futile and pointless as trying to find a single "universal" control structure or data structure or parameter mechanism. It is perhaps unnecessary for a language to incorporate *all* of the definition structuring mechanisms discussed here, but it is evident that some, such as private definitions, are definitely lacking in all current languages.

9.6 DEFINITION PROCEDURES AND class DEFINITIONS

According to the principle of abstraction (Section 7.4), it is possible to have procedures whose bodies and invocations are *definitions*. In the language SIMULA, these are known as ***classes***.

The general form of a **class** definition in a PASCAL-like notation might be

```
class I(··· ; P_i ; ···);
  D
end;
```

Here is a small example:

```
class random(seed : integer);
  private
    var a : integer;
    initial a:=seed mod d end
  within
    procedure draw(var x : real);
      begin
        a:=a*m mod d;
        x:=a/d
      end
end;
```

The effect of an invocation of this definition procedure, say

```
random(i)
```

would be to allocate and initialize a new location and bind identifier '*draw*' to a procedure for generating successive elements of a random number sequence.

An invocation of a class is itself a form of definition and could be used wherever a definition can appear, such as in the definition part of a block, as in

```
random(i);   {create a random number generator}
begin
   ⋮
  draw(r);   {update r to the next random number}
   ⋮
end
```

In SIMULA, this is known as ***block prefixing***.

A disadvantage of this notation for class invocation is that it does not permit more than one invocation of any class in the same context, because this would bind identifiers more than once. A simple solution to this problem is to allow class invocations and applied occurrences of the identifiers which they bind to be ***qualified*** by an additional identifier, as in

```
p.random(i);  {create a random number generator p}
q.random(j);  {create a random number generator q}
begin
   ⋮
  p.draw(r);  {update r to the next random number from p}
   ⋮
  q.draw(r);  {update r to the next random number from q}
   ⋮
end
```

The qualifying identifiers '*p*' and '*q*' distinguish between the two ***instances*** of class *random* used in the block.

Such qualification has another important benefit. Note that the binding created by an invocation of *random* is an implicit one (Section 6.2.3), in that identifier '*draw*' which is bound by it is not explicit in the text at that point.

But if applied occurrences of '*draw*' are qualified, as in the above, then the corresponding binding points may easily be found at the binding occurrence of the qualifying identifier ('*p*' or '*q*' in the example).

For these reasons, it is desirable to use the following syntax for invocation of a class I:

$$I'.I(\cdots,E,\cdots)$$

where I' becomes the name of an instance of the class. Any of the identifiers I_i defined by the body of the class would then be accessible in the scope of this invocation by a qualified reference of the form

$$I'.I_i$$

Definition procedures are extremely useful for structuring large programs because they allow definitions to appear in one context and be used in other contexts. Unfortunately, SIMULA is the only widely available programming language that provides classes. In most other languages, the concept must be simulated or approximated.

In PASCAL, the definition of *random* could be simulated as follows:

```
procedure random(seed : integer;
                 procedure inner(procedure draw(var x : real)));
  var a : integer;
  procedure draw(var x : real);
    begin
      a:=a*m mod d;
      x:=a/d
    end;
  begin
    a:=seed mod d;
    inner(draw)
  end;
```

Parameter '*inner*' represents the computation that uses an instance of *random*, when given access to a generating procedure '*draw*'.

An invocation of class *random* and use of that instance of the class may then be simulated as follows:

```
procedure use(procedure draw(var x : real));
  begin
    :
    draw(r);        {update r to the next random number}
    :
  end;
begin
  random(i,use)     {create a random number generator}
end
```

Note that this simulation of class definition and invocation involves only *local* transformations. In particular, a **class** definition can be transformed without knowing how its instances are used, and each of its uses can be transformed without knowing its definition. The simulation therefore retains the modularity of the original, but of course is neither as convenient nor as readable.

EXERCISES

9.1 Devise syntax and describe the semantics of definition forms that would correspond (in the sense of the principle of correspondence) to (a) name parameters (as in ALGOL 60); (b) PL/I parameters; and (c) value-result parameters.

9.2 What procedure would be defined by

```
function eof(f : text) : Boolean;
  begin
    if eof(f) then eof:=true else eof:=f↑=' '
  end;
```

if the definition were interpreted (a) recursively; and (b) non-recursively?

9.3 In an environment in which '*i*' denotes 2 and '*j*' denotes 3, what would be the effect of **const** definitions

```
    i=j;
and
    j=-i;
```

if they were composed (a) sequentially; (b) simultaneously; (c) with the first private to the second; and (d) as in PASCAL?

9.4 In PASCAL, definitions in a block must occur in the order

```
const
type
var
procedure and function
```

Suggest reasons why this order might have been imposed and comment on the success and desirability of this rule.

9.5 In PASCAL, command blocks

```
D begin C end
```

may only appear as the bodies of entire programs or procedures. In most other languages, such blocks may be used in *any* command context. What are the advantages and disadvantages of the approach in PASCAL?

9.6 The storage for an **own** declaration in ALGOL 60 is allocated and initialized just once, *before* program execution, rather than *each* time the declaration is interpreted; however, the *scope* of the binding created by an **own** declaration is the same as for an ordinary declaration. In a PASCAL-like notation, we might express this form of declaration as follows:

```
own I : T = K;
```

where the value of static expression K is used to initialize the storage. (In ALGOL 60, only a default initialization is possible.) For example, the following definition is similar to the random number generator of Section 9.5.5:

```
procedure draw(var x : real);
  own a : integer=seed;
  begin
    a:=a*m mod d;
    x:=a/d
  end;
```

In general, what disadvantages does this approach have in comparison with the definition block approach described in Section 9.5.5?

9.7 Use the simulation method suggested at the end of Section 9.6 to transform into strict PASCAL an example in which more than one instance of a class is used in some context.

9.8 It would be feasible for a language to provide blocks in which the definition part *follows* the block body, as in the following PASCAL-like command block:

```
begin
    C
  where
    D
end
```

The block body C is to be executed in the environment obtained by first interpreting definition D. Discuss the advantages and disadvantages of such constructs.

PROJECT

Explore the feasibility and usefulness of interpreting **class** definitions *recursively*.

BIBLIOGRAPHIC NOTES

Most of the concepts discussed in this chapter are due to Landin [9.1] and Strachey [9.3]. The interpretation of SIMULA classes as definition procedures was given in Tennent [9.4], and the simulation of classes by procedures is from Reynolds [9.2].

9.1 Landin, P. J. "The next 700 programming languages", *Comm. ACM*, **9** (3), 157–64 (1966).

9.2 Reynolds, J. C. "Syntactic control of interference", *Conf. Record of the 5th ACM Symposium on Principles of Programming Languages*, pp. 39–46, ACM, New York (1978).

9.3 Strachey, C. *Fundamental Concepts in Programming Languages*, lecture notes for the International Summer School in Computer Programming, Copenhagen (1967).

9.4 Tennent, R. D. "Language design methods based on semantic principles", *Acta Informatica*, **8**, 97–112 (1977).

10 JUMPS

In preceding chapters it was convenient to assume that evaluation of an expression *always* produces a value, that execution of a command *always* produces a new store, and that interpretation of a definition *always* produces a new environment. These assumptions are unrealistic for non-trivial programming languages.

One problem is that executing an iteration or invoking a recursively defined procedure may result in a *non-terminating* (i.e., infinite) computation and never produce a final value or store or environment. A second problem is that a computation may be "trapped" and aborted because of an error such as an array index being out of range. Finally, control may "jump" out of or into a construct because of language features like the **goto** sequencer.

Because their effects are "global", non-termination and abortion can easily be treated as special cases. For example, interpretation of

$$E_1+E_2$$

can be described as yielding the sum of the values of E_1 and E_2 *if* both of these interpretations terminate without error, and otherwise, as either non-terminating or erroneous. By always "propagating" errors and non-termination in this way, the correct result for an incorrect or non-terminating program will be specified. For convenience, we shall continue to describe expression interpretation as "evaluation", even when it is possible that no value will be produced.

It is more difficult to cope with control jumps. The first section of this chapter introduces the concept of ***continuations***, which will be used in subsequent sections to discuss the semantics of sequencers, labels, sequencer procedures, coroutines in SIMULA, and backtracking in SNOBOL4.

10.1 CONTINUATIONS

The *continuation* for some computation is whatever comes after it, expressed as a function of the results expected for that computation. For example, the result expected for a command execution is a new store. Consequently, the continuation for a command execution is the computation that follows it, as a function of the store resulting from that execution (if it terminates without error, of course).

For example, consider an execution of a composite command of the form

$$C_1;C_2$$

The continuation for the execution of C_1 starts with an execution of C_2 relative to the store resulting from the execution of C_1. The execution of C_2 inherits the continuation of the whole construct as its continuation.

The continuation for an expression evaluation depends on the value and the store that results from that evaluation. For example, in an execution of a command of the form

if E **then** C_1 **else** C_2

the continuation for the evaluation of expression E starts by using the resulting truth value to select execution of either C_1 or C_2. The execution selected inherits the continuation of the whole construct as its continuation.

As another example, consider the iterative construct

loop
  C_1
while E:
  C_2
repeat

described in Section 5.3.2. The continuation for an execution of C_1 starts with an evaluation of E relative to the resulting store. The continuation for this evaluation starts by using the resulting value to select either the execution of C_2 or the continuation of the whole construct. Finally, the continuation for an execution of C_2 starts with a re-execution of C_1.

Similarly, the continuation for interpreting a *definition* is a computation that is functionally dependent on the environment and store the definition produces. For example, in an execution of a PASCAL block of the form

D
begin
  C
end

the continuation for the interpretation of D starts with an execution of C relative to the resulting environment and store.

The semantics of a language that has no sequencers or other jumping mechanisms can be described without mentioning continuations. In the following sections it will be demonstrated that when jumps *are* possible languages can be conveniently described denotationally using continuations.

10.2 SEQUENCERS

Consider the familiar sequencer

goto N

in PASCAL. We want to describe its effect in terms of the meaning of N, its only immediate constituent. Let us assume that a label denotes a command continuation, that of continuing at the program point labelled. For example in block

begin
  ⋮
13: $x:=x+1$;
  ⋮
end

label '13' would denote the continuation that starts with an execution of '$x:=x+1$' and then follows the continuation of this command in the program. Note that the semantic notion of continuation is an appropriate *abstraction* from the syntactic notion of "program point": two distinct program points may represent the same continuation. For example, the two labels in

12:{null}; 13: $x:=x+1;\cdots$

are attached to distinct program points, but denote the same continuation because they are semantically indistinguishable.

The effect of executing **goto** N can now be described as that of "following" the continuation denoted by N, using the current state of the store. Note that the "normal" (contextually determined) continuation for the sequence is ignored. For example, in

goto N; C

the "normal" continuation for the **goto** (which starts by executing C) would be ignored. The characteristic property of sequencers is that they always ignore their contextually determined continuation and follow some other continuation, that is to say, they always cause "jumps".

The **goto** sequencer has two properties not shared by all sequencers. (a) The continuation followed by executing a **goto** is obtained by evaluating an explicit *destination* expression that is an immediate constituent of the sequencer. (b) This continuation is a *command* continuation. There are sequencers in programming languages that have different properties. For example, sequencer

RETURN(E)

in PL/I is used within definitions of expression procedures to terminate their execution. The continuation that is followed as the result of interpreting this sequencer is an *expression* continuation because it depends on a value (the value of expression E) as well as on the store. Furthermore, the continuation that is followed is not the result of evaluating an explicit expression supplied by the programmer. The continuation is implicit in the environment component of the state with respect to which the sequencer is executed.

There are many examples of sequencers that follow command continuations but, unlike the **goto**, do not have explicit destinations. One is RETURN in command procedure definitions in PL/I or FORTRAN. Another is **break** in the language BCPL; this causes termination of the smallest enclosing iteration. However, these cannot be used to return from or exit out of *nested* procedure definitions or iterations.

10.3 LABELS

The familiar **goto** sequencer has been much criticized by some programming theorists. However, the semantic description of the **goto** construct itself simply involves following a command continuation and is no more complex than that of other sequencers, such as RETURN(E). This suggests that some of the criticism of the **goto** might be more appropriately directed against the treatment of *labels* in programming languages, that is, on how the continuations for **goto**s are determined.

Some languages (including FORTRAN, COBOL, and PL/I), allow label values (i.e., command continuations) to be *stored*. The value of the destination expression of a **goto** can then depend on the dynamic history of the computation. This is often detrimental to program readability.

In most languages which have been designed more recently, only *identifiers* (or numerals acting as identifiers) can express label values. This still allows the possibility of label *parameters* to procedures, as in ALGOL 60 or FORTRAN. An example in PASCAL-like notation would be


```
procedure p(···; label lbl;···);
  begin
    ⋮
      goto lbl;
    ⋮
  end;
```

Here the destination of the sequencer depends on the value of the actual parameter corresponding to '*lbl*' in the invocation of the procedure.

In PASCAL, label values are denotable *only* by the numeral that occurs at the destination itself. This makes the destination of every **goto** very evident to program readers.

There is also considerable variation possible in the *scope* rules for labels, that is to say, the language conventions regarding *where* applied occurrences of labels may appear. An obvious constraint on labels is that it must not be possible to jump into a block or procedure body before it has been entered (unless some default bindings are assumed for the parameters or locally defined identifiers). In FORTRAN, ALGOL 60 and PASCAL, this restriction is enforced by letting the scope of a label binding be the innermost block or procedure body that contains that binding occurrence.

The scopes of labels (or their usability) can be restricted further in order to prevent jumping into the subcommands of iterative or selective control structures. In PASCAL, a label is not usable *throughout* its scope; the usable part of a label's scope is the command sequence in which it is bound. In addition, if this compound command is the body of a block, the label is also usable in any procedure definitions in that block. This rule permits jumps within and out of command sequences, as in

```
begin
  ⋮
  goto 13;
  ⋮
13:···
  ⋮
  begin
    ⋮
    goto 13;
    ⋮
  end;
  ⋮
end
```

as well as exits from procedure definitions in a block to the "outer level" of that block, as in

```
label 13;
procedure p;
  begin
    ⋮
    goto 13;
    ⋮
  end;
begin
  ⋮
13: ···
  ⋮
end
```

but it does not allow jumps *into* control structures. For example, the following is illegal:

```
begin
  ⋮
  goto 10;
  ⋮
  while E do
    begin
      ⋮
      10: C;
      ⋮
    end;
  ⋮
end
```

because the label used as the destination of the **goto** is outside the usable part of its scope.

It is possible and perhaps desirable to restrict label scopes even further. If the scope (or usability) of the label binding in a labelled command

N: C

were this command itself, then **goto** N could only be used for re-executing a command that contains it. It might then be appropriate to change the syntax of the sequencer to, for example,

redo N

to emphasize its specialized use.

Suppose that command label N in

N: C

were interpreted as being the name of the *continuation* of this command (instead of the continuation that starts with an execution of C). If, as above, the scope (or usability) of this label binding were restricted to C, then **goto** N could be used only to exit out of commands that contain it. In this case it would be appropriate to change the syntax of the sequencer to something like

leave N

These sequencing disciplines may be simulated in standard PASCAL by using labels only immediately after **begin**s or immediately before **end**s of **begin** ··· **end** constructs, as in

```
begin N:
  C1;
  C2;
  ⋮
  Cn
end
```

or

```
begin
  C1;
  C2;
  ⋮
  Cn;
N: end
```

A jump to N can only occur from within one of the commands C, because of the usability constraints, and would always have the effect of repeating or exiting from the whole construct. With these kinds of restrictions on label scope or usability, the **goto** (and other sequencers with the same semantics) are similar in effect to sequencers such as **break** or RETURN, but are more flexible because they have explicit destinations.

10.4 SEQUENCER PROCEDURES

The "normal" continuation for a command, expression, or definition procedure is the continuation of the *invocation* of the procedure. A ***sequencer procedure*** would be one that *cannot* return to its invocation; that is, both the invocation and the body of such a procedure would effectively be sequencers, that is to say, phrases that ignore their "normal" continuations. This is another illustration of the principle of abstraction.

Consider, for example, definitions of the form

```
exit I(···;P;···); C;
```

which are syntactically similar to conventional **procedure** definitions. However, an invocation of an **exit** procedure would return to the end of the block in which it is defined, rather than to its invocation. Thus, in

```
exit error(n:integer);
  begin
    write('ERROR',n)
  end;
begin
  ⋮
  error(21);
  ⋮
end
```

any invocation of sequencer procedure *error* will result in output of an error message followed by immediate termination of the entire block, regardless of the continuation of the invocation.

It is possible to simulate this in PASCAL as follows:

```
label 999;
procedure error(n:integer);
  begin
    write('ERROR',n);
    goto 999
  end;
begin
  ⋮
  error(21);
  ⋮
999:end
```


However, use of a distinctive syntactic form for exit procedures would make their role more evident to program readers, and might permit a more efficient implementation.

In a language with sequencer procedure definitions, it would also be useful to allow procedures (of all kinds) to have sequencer procedure *parameters*. (Recall the principle of correspondence.) This would allow "exception handlers" for a procedure to be specified at its invocations. Some languages (including PL/I and ADA) achieve this in an ad hoc way: they use dynamic binding (Section 7.2.2) for exception handler names.

10.5 COROUTINE SEQUENCING IN SIMULA

It is sometimes desirable to have a programmer-defined procedure *suspend* its execution and transfer control to another context in such a way that subsequently it can be *re-activated* and will continue executing from the point of suspension. This is termed ***coroutine*** sequencing, because it allows each of several program parts to regard the others as its "subroutines".

In languages with storable label values, coroutine sequencing may be implemented by saving a continuation in a label-valued location before jumping to another context. For example, if a programmer adopts the convention that I_1 and I_2 denote label-valued locations that always contain the resumption continuation for coroutines c_1 and c_2, respectively, then suspension of c_1 and re-activation of c_2 may be expressed by

```
  ⋮
I1:=N;
goto I2;
N:···
  ⋮
```

because label N denotes the resumption continuation for c_1.

But such use of storable label values and **gotos** is not very convenient or clear or efficient. In this section we shall describe the specialized coroutine facilities in SIMULA. For convenience, PASCAL-like syntax will be used.

Suppose that a **coroutine** definition resembles a **class** definition, except that the last **initial** part of its body may contain occurrences of a pseudo-identifier '**detach**', as in

```
coroutine I(···;P;···);
  D
  initial
      ⋮
    detach;
      ⋮
    detach;
      ⋮
  end
end;
```

Interpretation of the body of a **coroutine** definition begins when it is invoked, but suspends when a **detach** is executed. Computation in the coroutine body is resumed when it is re-activated by use of a standard procedure, *call*, as in the following:

```
coroutine c;
    ⋮
  initial
    ⋮
    detach
    ⋮
  end
end;
  ⋮
p.c;              {invocation}
begin
    ⋮
  call(p);        {re-activation}
    ⋮
  call(p);        {re-activation}
    ⋮
end
```

Note that although they are syntactically different, the effects of '**detach**' and '*call*(*p*)' are quite similar: each saves its own "normal" continuation and then follows a previously saved continuation for the other context. They differ only in whether the destination is explicit (*call*) or implicit (**detach**).

A coroutine allowing node-by-node "inorder" traversal of binary trees will illustrate the use of coroutine sequencing. See Fig. 10.1. Parameter *root* is a pointer to the root node of a binary tree. Locations *current* and *more* are for communicating information between the traversing coroutine and its "user".

Execution of the **initial** part of the **coroutine** definition will set *more* to *true* and *current* to the contents of the *key* field of the "leftmost" node of the tree by using recursively defined procedure *traverse*. However, if the tree is empty (i.e., *root*=**nil**), the value of *more* is set to *false*. The coroutine then suspends itself by executing **detach**, and control returns to the point of invocation of the class. A subsequent *call* of the coroutine resumes its execution from the point where it has most recently suspended itself.

```
type ptr=↑node;
     node=record
            key:integer;
            left,right:ptr
          end;
  ⋮
coroutine inorder(root:ptr);
  var current:integer;
      more:Boolean;
  initial
    procedure traverse(ref:ptr);
      begin {traverse}
        if ref ≠ nil then
        begin
          traverse(ref↑.left);
          current:=ref↑.key;
          detach;
          traverse(ref↑.right)
        end
      end; {traverse}
    begin
      more:=true;
      traverse(root);
      more:=false;
      detach
    end {begin}
  end {initial}
end; {inorder}
```

Figure 10.1

Here is how an instance of *inorder* might be created in a block prefix and then used:

```
i.inorder(root);
begin
  while i.more do
  begin
    write(i.current);
    call(i)
  end
end
```

Each execution of '*call*(*i*)' causes execution of this block to be suspended and control to follow the continuation of the most recently executed **detach** in the coroutine. This re-activation either updates *i.current* to the "next" key value in the tree, or changes *i.more* to *false* if the whole tree has been traversed.

Of course, a *complete* traversal such as the above could be programmed just as simply without using coroutine sequencing. To illustrate the usefulness of the *node-by-node* traversal obtainable with the coroutine, consider the problem of testing the equivalence of two binary search trees, that is to say, testing whether they represent the same sorted sequence of keys. Because trees may be equivalent without having the same structure, this is quite inconvenient to program efficiently without using coroutine sequencing.

Here is how this problem could be solved using an *inorder* instance for each tree:

```
var equiv:Boolean;
i1.inorder(root1);
i2.inorder(root2);
begin
  equiv:=true;
  while equiv and i1.more and i2.more do
    if i1.current=i2.current
      then begin call(i1); call(i2) end
      else equiv:=false;
  if i1.more or i2.more then equiv:=false;
    ⋮
end
```


In more complex examples, it is often convenient for a coroutine to resume another without the intermediate step of relinquishing control to the context where they were originally created. In SIMULA, this may be achieved by executing '**resume** *p*' in a coroutine. The effect is equivalent to executing '**detach**', followed immediately by a *call* of *p* in the "parent". For example, the following is the outline of the definition of a game-playing coroutine class:

```
var "game state";
coroutine player(var "opponent");
  var "private move-making data";
  initial
    "initialization";
    detach;
    loop
      ⋮
      "make move";
      resume "opponent";
      if "game over" then detach;
      ⋮
    end {loop}
  end {initial}
end; {player}
```

Each *player* instance resumes its opponent after making its move. When the game is over, a player detaches to the parent. This example illustrates clearly the *symmetry* of coroutine sequencing.

10.6 BACKTRACK SEQUENCING IN SNOBOL4

Backtracking algorithms search for solutions by repeatedly making a choice among alternatives and, if the desired solution is not subsequently found, backing up and "undoing" that choice in order to explore another alternative. Such algorithms are often programmed by using recursively defined procedures having the general form

```
var "solution";
  procedure solve;
  "local definitions"
  begin
    for "each possible choice" do
      if "the choice is feasible" then
        begin
          "record choice";
          if "solution complete"
            then "exit"
            else solve;
          "undo choice"
        end;
    {no more alternatives here; return to a higher level}
  end;
```

The pattern matching operation in SNOBOL is implemented by a similar backtracking approach. This is because a pattern formed by alternation may match in more than one way, and any matching that follows the first possible match may have to be "undone" in order to explore another of the possible matches. For many patterns, this property of the implementation need not be known by the programmer, and the relatively simple model of Section 3.2.3 is sufficient. However, some patterns in SNOBOL have *side effects* during the match, and these require that account be taken of the *sequencing* of pattern matching. For example, if the value of E_1 is a pattern p, then the value of expression

$$E_1 \$ E_2$$

is a pattern that matches like p and, whenever such a match succeeds, also has the side effect of immediately updating the *l*-value of E_2 to be the substring of the subject that is matched by p.

10.6.1 An Example

To illustrate control sequencing during pattern matching, consider matching pattern

('BE'|'B') ('ET'|'AD')

against subject 'BET' at position 0. We know from our previous discussions that the result will be a successful match up to position 3. This result is actually computed as follows. First, 'BE' matches up to position 2. Then, attempts are made to match 'ET' and then 'AD' at this position. But both of these fail, so the matching process backs up to try the most recent unexplored alternative, which is pattern 'B' at position 0 of the subject. This succeeds up to position 1, and now 'ET' matches up to position 3. If subsequent matching causes further backing up, the second alternative, 'AD', will be tried at position 1 but this will fail, and the matching process will then back up to try an even earlier alternative, if any.

*10.6.2 Semantic Description

Such jumps to and fro between pattern components may be described in general by assuming that application of a pattern has *two* possible continuations. One, termed the ***subsequent***, is to be followed whenever a match is (initially) successful. The second, termed the ***alternate***, is followed if the match fails. Thus, if the null string is used as a pattern, the subsequent will always be followed, whereas the value of 'FAIL' in SNOBOL is a pattern that always follows the alternate.

It then turns out that both alternation and concatenation of patterns are just forms of sequential composition, much like the sequential composition of commands expressed by

$$C_1;C_2$$

Alternation is sequential composition with respect to alternate continuations, and concatenation is sequential composition with respect to subsequent continuations. This explains why the patterns denoted in SNOBOL by 'NULL' and 'FAIL' are to pattern concatenation and alternation what the null command is to command composition, that is to say, the identity of the associated operation.

If two patterns p_1 and p_2 are composed by *alternation*, the result is a pattern that matches like p_1, but has as its alternate a match of pattern p_2 at the *same* position in the subject string. For example, in pattern

'BE'|'B'

the alternate for 'BE' is a match of 'B' at the same position in the subject string. The alternate of the whole pattern is inherited by p_2 as its alternate. Therefore, if neither p_1 nor p_2 is successful, control will back up by following this "external" alternate. The subsequent continuation of the whole pattern is also inherited by both p_1 and p_2 as their subsequents, so that it will be followed after a successful match by *either* of them.

If two patterns p_1 and p_2 are *concatenated*, the result is a pattern that matches like p_1 and has p_2 as its subsequent. For example, with pattern

('BE'|'B') ('ET'|'AD')

a match using 'ET'|'AD' is the subsequent for 'BE'|'B'. The subsequent continuation of the whole pattern is inherited by p_2 as its subsequent, so that it will be followed after successful matches by *both* p_1 and then p_2. Similarly, the alternate continuation of the whole pattern is inherited by p_1 as its alternate.

To specify the alternate continuation for p_2 when patterns p_1 and p_2 are concatenated, we must consider what kinds of continuations subsequents and alternates are. An alternate is simply a conventional command continuation; that is, an alternate depends only on the state of the store when it is followed. This means that side effects to the store during pattern matching are *not* undone. A subsequent continuation is more like an expression continuation in that it also requires a value: the position in the subject up to which previous patterns have matched. Furthermore, a subsequent also needs an alternate continuation to fall back to if it fails. In our example, after pattern component 'BE' matches, the subsequent match has 'B' (and then *its* alternate) to fall back to.

Therefore, if p_1 and p_2 are concatenated, the alternate continuation for p_2 is that supplied by p_1 when it followed its subsequent. This means that if backtracking is necessary during the match of p_2, unexplored alternatives in p_1 will be tried before resorting to the alternate for the whole concatenated pattern. This occurs in the example; the alternate that is followed when pattern 'ET'|'AD' fails (after the initial success of 'BE') is a match with 'B' at position 0 of the subject.

In summary, language facilities for backtrack sequencing may be described in terms of two continuations, one, like the subsequent, for "forward" sequencing, and a second, like the alternate, for "backing up".

EXERCISES

10.1 Devise a sequencer that has an explicit destination (like **goto**) but follows an *expression* continuation (like RETURN(E)). Also suggest suitable labelling conventions for use with this sequencer.

10.2 The following iterative construct has been suggested:

```
loop until l1 or l2 or ··· or ln:
  C0
repeat
then
  l1:C1;
  l2:C2;
  ⋮  ⋮
  ln:Cn
end
```


It is executed by repeatedly executing C_0 until a free occurrence of any l_i is executed. Control then transfers immediately to C_i, and then follows the continuation of the whole construct. Show how to simulate the effect of this using **exit** procedures.

10.3 Would you recommend that **exit** definitions be interpreted recursively or non-recursively? Why?

10.4 Design a form of sequencer procedure that follows an expression continuation.

10.5 Show how to simulate in PASCAL the effects of **label** parameters and **exit** parameters.

10.6 Why would it be incorrect to regard the "usable" part of the scope of a label in PASCAL as being its entire scope?

10.7 Built-in operations (such as 'div') do not allow a programmer to provide an "exception handler" for recovery from an error condition (such as for attempted division by zero). What are the advantages and disadvantages of the following design approaches to this problem:

(a) The language can use dynamic binding for the names of **exit** procedures, so that error handlers can be defined in the contexts where the operation is used.

(b) The language can provide predefined procedures that are equivalent to primitive operations but have additional **exit** parameters, such as

```
function xdiv(i,j:integer; exit zerodivide):integer;
  begin
    if j=0 then zerodivide else xdiv:=i div j
  end;
```

A programmer can then supply an error handler as an explicit additional argument.

10.8 The effect of executing to the end of the body of a **coroutine** definition was not specified in Section 10.5. What are some possibilities? Discuss their advantages and disadvantages.

PROJECTS

10.1 Design language facilities for defining and using coroutines that return values when they "detach". (These have been termed "streams", or "generators", or "iterators".)

10.2 Investigate the use of sequencer procedures and sequencer structures (i.e., selective sequencers, sequencer blocks, sequential composition of a command with a sequencer, and so on) as replacements for conventional selective and iterative command structures.

BIBLIOGRAPHIC NOTES

The use of continuations for modelling jumps is due to Strachey and Wadsworth [10.6] and F. L. Morris [10.5]. Sequencer procedures similar to the **exit** mechanism were described by Landin [10.4] (who termed them "program points") and Clint and Hoare [10.2] (who termed them "label procedures"). The binary tree example for coroutines was suggested by Dahl and Hoare [10.3]. The semantics of backtracking is based on Tennent [10.7]. For project 10.2, see Back [10.1].

10.1 Back, R. J. R. "Exception Handling with Multi-exit Statements", in *Programmiersprachen und Programmentwicklung* (ed. H. J. Hoffman), 6. Fachtagung des Fachausschusses Programmiersprachen der GI, Darmstadt, Informatik-Fachberichte **25**, Springer, Berlin–Heidelberg (1980); also Report IW 125, Mathematical Centre, Amsterdam (1979).

10.2 Clint, M. and C. A. R. Hoare. "Program proving: jumps and functions", *Acta Informatica*, **1**, 214–24 (1972).

10.3 Dahl, O.-J., and C. A. R. Hoare. "Hierarchical program structures", in *Structured Programming* (O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare), 175–220, Academic Press, London (1972).

10.4 Landin, P. J. "The next 700 programming languages", *Comm. ACM*, **9** (3), 157–64 (1966).

10.5 Morris, F. L. "The next 700 formal language descriptions", typescript, University of Essex, Colchester (1970).

10.6 Strachey, C., and C. P. Wadsworth. *Continuations, a Mathematical Semantics for Handling Full Jumps*, technical monograph PRG-11, Programming Research Group, University of Oxford (1974).

10.7 Tennent, R. D. *Mathematical Semantics and Design of Programming Languages*, Ph.D. thesis and technical report 59, Computer Science, University of Toronto (1973).

11 CONCURRENT PROCESSES

This chapter discusses linguistic facilities that allow or require *concurrent* (or "parallel") processing. There are two motivations for such facilities. The more obvious is that they may permit more efficient use of systems with multiple *processors*. In most computer systems, peripheral devices contain processing units that operate relatively independently of the central processor. This allows overlapping of lengthy input or output operations with "internal" computations. There are also specialized computer systems known as array processors that contain an array of identical arithmetic processing units. Whenever such a multiple processor configuration is available, it may be possible to reduce execution time and to make more effective use of hardware resources by giving programmers control over the processors by means of appropriate facilities in a programming language.

The other motivation for concurrent programming facilities is that they allow programs to be structured as autonomous (but possibly interacting) *processes*, even if they are actually implemented on a single processing device. Programs organized as interacting processes are particularly useful for operating systems, real-time control systems, simulation studies, and combinatorial search applications.

In this chapter, the issues underlying concurrent programming facilities will be discussed, using typical language features for illustration. We shall not attempt to describe all of the linguistic mechanisms that have been proposed or implemented.

11.1 INTERFERING PROCESSES

Consider extending PASCAL by allowing composite commands of the form

$$C_1 \| C_2$$

The intention is to allow an implementation to execute commands C_1 and C_2 concurrently, either using separate processors or by interleaving the executions on a single processor. What would be approprate as the semantic description of this construct?

Consider the following simple example:

```
var i : integer;
begin
    ⋮
  begin i:=1 ‖ i:=2 end;
    ⋮
end;
```

This example involves ***interfering*** processes, that is to say, processes that share access to storage updated by at least one of them. In this case, the shared storage is location *i*.

If it were possible for these two processes *simultaneously* to update *i*, its final contents would be unpredictable. For example, if integers were represented in binary form, the final value might be 0 or 3, as well as 1 or 2. Of course, this cannot happen in actual computers, so that it is reasonable to assume that implementations of concurrency ensure that primitive operations of interfering processes get ***indivisible*** access to shared storage. That is, if "simultaneous" access is attempted by primitive operations in two processes, the implementation must delay one of these until the other process has completed its operation. If concurrency is implemented by interleaving on a single processor, then the interleaving must not be "finer-grained" than the primitive operations.

So let us assume that updating a shared storage location is in fact an indivisible operation. What then will be the value of *i* after executing '$i:=1$' and '$i:=2$' concurrently? Evidently, this depends on which of the updatings is done last, that is, on the relative execution speeds of the two processes.

In general, specifying *the* result of executing two processes concurrently would require exact and detailed knowledge of their execution times for all possible data. But this is completely infeasible for any but the simplest programs. Consider, for example, the problem of trying to specify the response time of a human operator at an interactive terminal, or the time needed to carry out a mechanical action in an input or output device. Execution times are far too dependent on implementation, data, algorithm, and hardware to play an important role in semantic descriptions.

The only reasonable alternative is to let the execution speed of a concurrent process be *indeterminate* (Section 5.5). Then the *possible* results of executing

$$C_1 \| C_2$$

are the stores resulting from *any* of the interleavings of the sequences of indivisible primitive operations of C_1 and C_2. For example, the result of executing '$i:=1 \| i:=2$' is to set *i* to either 1 or 2.

Note that

```
begin i:=0; i:=i+1 end
```

has the same overall effect on the store as '$i:=1$', but in the presence of concurrency, these commands are *not* semantically equivalent. For example, one of the possible results of executing

```
begin i:=0; i:=i+1 end ‖ i:=2
```

is to set *i* to 3 because of the interleaving

```
i:=0;
i:=2;
i:=i+1
```

When concurrency is possible, the semantics of commands must in general be expressed as sequences of indivisible primitive operations that can be interleaved by operations of other processes, rather than as overall store transformations.

When uncontrolled interference is possible, programming is extremely difficult because it is so hard to abstract from the level of primitive operations. In all but trivial examples, there will be a very large number of possible results and a programmer must verify that all of these are correct. Furthermore, he cannot rely on test runs to check a program's correctness exhaustively, nor can he expect to be able to duplicate erroneous executions for debugging.

For these reasons, concurrency that involves *unrestricted* interference is of very little practical interest except at the hardware level. In the following sections we shall discuss concurrent composition without interference, or with controlled forms of interference.

11.2 NON-INTERFERING PROCESSES

In the preceding section, constituents C_1 and C_2 in a concurrent command of the form

$$C_1 \| C_2$$

were allowed to be arbitrary commands, and this created serious difficulties if they interfered with one another. Let us therefore suppose that C_1 and C_2 are ***non-interfering***, that is to say, no storage updated by either is referred to or updated by the other. Then the effect of

$$C_1 \| C_2$$

is determinate. *All* interleavings of the two sequences of primitive operations (including sequential execution of C_1 and C_2 in either order) will have the same overall effect on the store because each constituent only modifies locations that are inaccessible to the other. The effect of the composite is therefore uniquely determined by the usual meanings of its immediate constituents, and it is not necessary to decompose these meanings into sequences of primitive actions. Non-interference is a rather severe constraint, but it greatly simplifies the semantics of concurrent composition.

As an illustration of non-interfering processes, the following is an excerpt from a program for processing a file using a procedure *update* concurrently with input and output operations:

```
⋮
while not eof(input) do
  begin
    output↑ := buffer;
    buffer := input↑;
    begin get(input)
       ||  update(buffer)
       ||  put(output)
    end
  end;
⋮
```

The three processes access disjoint parts of the store, so that their concurrent composition is equivalent to any of their sequential compositions, but might be executed more rapidly.

11.3 COOPERATING PROCESSES

It is possible to relax the requirement of strict non-interference if the language provides some facility that allows a programmer to specify that

arbitrary (i.e., *non*-primitive) operations on shared resources are to be executed indivisibly. Then, concurrent processes can ***cooperate*** to prevent undesirable interference.

For example, suppose that concurrent processes wish to cooperate in accumulating a *count* of some sort. At various times each process p_i will want to increment the shared counter as follows:

count := *count*+*ni*

However, without some means of specifying the *indivisibility* of these incrementing assignments, some of the increments can be lost during interleavings of more primitive operations, as in the following:

evaluation of '*count*+*ni*' in process p_i;
evaluation of '*count*+*nj*' in process p_j;
assignment to '*count*' in process p_i;
assignment to '*count*' in process p_j

Many facilities have been proposed for specifying indivisibility of operations. We will describe one that is used in several system programming languages, including CONCURRENT PASCAL, MODULA, and PASCAL PLUS. The basic principle is to specify the operations as procedures which operate on storage that the implementation makes available to at most *one* process at any time. For example, a shared counter might be described as follows:

```
monitor
  var count : integer;
  initial count := 0 end
within
  procedure increment(n : integer);
    begin count := count+n end;
  function total : integer;
    begin total := count end;
begin
     begin ··· increment(ni); ··· end
    ||begin ··· increment(nj); ··· end;
  write('TOTAL COUNT = ', total);
    ⋮
end
```

The composite definition form

monitor D_1 **within** D_2

is syntactically and semantically similar to the definition block discussed in Section 9.5.5. But, the implementation must *also* ensure that at any time at most *one* process is executing any of the procedures defined in D_2. If a process attempts to invoke such a procedure, it will be suspended until no other process has control of the monitor. This removes from the set of possible results of the concurrent processes any that arise from interleavings of the primitive operations of the procedures. In effect, the procedures defined in D_2 become *indivisible* operations.

Note that programmers should attempt to minimize the time spent executing indivisible procedures, because during this time other processes may be forced to wait. A dangerous situation known as ***deadlock*** must certainly be avoided. Consider, for example, the following program fragment involving two mutually defined monitors:

```
monitor
  var "resource1";
within
  procedure p1;
    begin ··· p2; ··· end;
  ⋮
monitor
  var "resource2";
within
  procedure p2;
    begin ··· p1; ··· end;
```

A procedure in each monitor invokes a procedure in the other monitor. The danger is that if concurrent processes invoke the two procedures, they may be suspended when they attempt to access the other resource. The two processes will then wait indefinitely and are said to be deadlocked. Some implementations of monitors attempt to prevent such situations by allowing a monitor procedure to access *only* its own resources. In general, however, programmers have the responsibility of avoiding deadlocks.

11.4 SYNCHRONIZED PROCESSES

In many applications, it is necessary for a programmer to control the *order* as well as the degree of interleaving of operations of concurrent processes. For example, suppose that it is desired to define a procedure *synchronize* with which *n* processes may synchronize their executions. A process that invokes *synchronize* is to be suspended until all *n* of the processes have done so.

Many approaches to process synchronization have been proposed. We will describe one which works well with monitors. It is based on a mechanism known as a ***condition*** on which a process can suspend itself and *wait* until the condition is *signalled* by some other process.

The following is a monitor that uses a condition to provide the synchronization facility described above:

```
monitor
  var count : 0..n;
      all : condition;
  initial count:= 0 end
within
  procedure synchronize;
    begin
      count:= count+1;
      if count < n
        then wait(all)
        else count:= 0;
      signal(all)
    end;
```

Execution of operation

wait(*c*)

on a condition *c* within a monitor procedure results in suspension of the executing process and release of its exclusive access to the resource. Operation

signal(*c*)

causes one process that may be waiting on condition *c* to *immediately* re-claim exclusive access to the resources of the monitor and continue processing from where it suspended itself. (If no processes are waiting on condition *c*, '*signal*(*c*)' has no effect.) In the example, *all* represents the condition that all of the processes have invoked *synchronize*.

In general, more than one process may be waiting on a condition. An implementation must then choose which one of these to re-activate when that condition is signalled. If an arbitrary choice is made, there is a danger that one of these processes may be delayed indefinitely in favor of other processes. This is known as ***starvation*** (or "indefinite overtaking"). A simple scheduling policy that avoids this is always to re-activate the longest-waiting process.

In some applications, it is convenient to have more flexibility over scheduling. Many of the languages that provide monitors and conditions extend the *wait* operation to accept an optional argument specifying the *priority* of the suspended process. The processes waiting on the condition may then be sorted by the implementation with respect to these priorities, and the one with the greatest priority when the condition is signalled will be selected for re-activation. This facility is particularly useful in operating-system applications.

Note that the correctness of programs that use conditions (or other synchronization mechanisms) depends on the implementation being ***fair*** in the way processing resources are shared among concurrent processes. For example, if $n=2$ and concurrent command

synchronize || *synchronize*

were implemented as if it were

synchronize ; *synchronize*

then the condition on which the first process is waiting would never be signalled, so that the execution would never terminate. However if neither process is indefinitely denied processing, then the execution terminates.

11.5 COMMUNICATING PROCESSES

Suppose that one process (the *producer*) wishes to ***communicate*** information to a concurrently executing process (the *consumer*). One way to avoid undesirable interference is to set up a "mailbox" in a monitor as follows:


```
monitor
  var mailbox : record case empty : Boolean of
                         true : ();
                         false : (contents : t)
                       end;
      filled, cleared : condition;
  initial mailbox.empty := true end
within
  procedure produce(x : t);
    begin
      mailbox.empty := false;
      mailbox.contents := x;
      signal(filled);
      if not mailbox.empty then wait(cleared)
    end;
  procedure consume(var y : t);
    begin
      if mailbox.empty then wait(filled);
      y := mailbox.contents;
      mailbox.empty := true;
      signal(cleared)
    end;
```

Condition *filled* is used to delay the consumer until there is a communication available. Similarly, condition *cleared* delays the producer until the communication it has just placed in the mailbox is consumed. (In practice, it would be necessary to buffer the communication, in order to smooth out temporary speed variations between the processes.)

Communication between concurrent processes (or "message-passing") is sufficiently important that it is convenient to have specialized high-level notation for it. The following illustrates one approach:

```
begin producer : begin  ··· send(consumer, x); ··· end
   ||    consumer : begin  ··· receive(producer, y); ··· end
end
```

Operation

send(I,E)

is a kind of "output" of the *r*-value of E to the process labelled I, much like '*write*(I,E)' in PASCAL. Similarly, operation

receive(I,L)

is an "input" into the process labelled I into the *l*-value of L, much like '*read*(I,L)'. Such communication may be *synchronized* by having a process wait when it is ready to send or receive until the process it names is also ready.

These communication primitives may also be used to implement our example of *cooperating* processes if the non-determinate control structures of Section 5.5 are generalized to allow a *receive* operation as a guard, as in the following:

```
begin
  counter : var count : integer;
                n : integer;
            begin
              count := 0;
              loop receive(p1,n) → count := count+n
                 | receive(p2,n) → count := count+n
              end;
              write('TOTAL COUNT =  ', count)
            end
  ‖ p1 : begin  ··· send(counter, n1); ···  end
  ‖ p2 : begin  ··· send(counter, n2); ···  end
end
```

A *receive* operation as a guard is satisfied if the process named is ready to send. If more than one such guard is satisfied, then an arbitrary (and not necessarily fair) choice is made of which communication to receive, and other sending processes must wait. In the example above, this ensures the indivisibility of the assignments to '*count*'. If no guard is satisfied but at least one of the processes named in a guard is still executing, the process waits. When none of the guards of an iteration is satisfied and all the processes named in the guards have terminated, the iteration terminates. Thus, in the example above, termination of *p*1 and *p*2 results in output of the final value of *count*, followed by termination of process *counter*.

This kind of synchronized internal input–output is a very flexible programming tool. However, it remains to be seen whether such communication primitives are efficiently implementable and effectively usable in practice.

EXERCISES

11.1 What are the possible results of executing the following?

```
var i : integer;
    b : Boolean;
begin
  i := 0; b := true;
  begin
    b := false
  ‖ while b do i := i+1
  end;
  write(i)
end
```

11.2 Comment on the following attempt to implement cooperative counting *without* using a monitor:

```
private
  var count : integer;
      inuse : Boolean;
  initial count := 0; inuse := false end
within
  procedure increment(n : integer);
    begin
      while inuse do (null);
      inuse := true;
      count := count+n;
      inuse := false
    end;
  ⋮
```

11.3 Conditions may be replaced in a language by commands of the form

await E

where expression E has type *Boolean*. If the value of E is *false*, then its effect is to suspend the executing process for an indeterminate (but finite) period of time. Then, the expression is tested again, and this repeats until the expression evaluates to *true*. When the process is suspended, it releases its control over the monitor, and must re-claim this to re-evaluate the expression.

For example, procedure *consume* in the "mailbox" example of Section 11.5 could be defined as follows:

```
procedure consume(var y : t);
  begin
    await not mailbox.empty;
    y := mailbox.contents;
    mailbox.empty := true
  end;
```

What are the advantages and disadvantages of this approach to synchronization? *Hint*: Is the following a correct definition of procedure *synchronize* (Section 11.4):

```
procedure synchronize;
  begin
    count := count + 1;
    await count = n;
    count := 0
  end;
```

11.4 Can an implementation allow a procedure which is defined in a monitor to invoke other procedures of the *same* monitor, or to invoke itself reentrantly?

11.5 Does the "mailbox" example of Section 11.5 work correctly if there is more than one producer or consumer process?

11.6 What problems arise in describing the semantics of concurrent composition of commands if it is possible to jump out of a concurrent process?

11.7 Show how to solve the synchronization problem of Section 11.4 using the *send* and *receive* communication primitives of Section 11.5.

PROJECTS

11.1 Devise syntactic constraints that would be sufficient to ensure that constituents C_1 and C_2 of a concurrent composition $C_1 \| C_2$ did not interfere.

11.2 Design language facilities that would allow programmers to make effective use of an array processor.

BIBLIOGRAPHIC NOTES

The material in this chapter is based primarily on papers by Hoare [11.3, 11.4, 11.5, 11.6]. Concurrent programming is discussed in books by Brinch Hansen [11.1, 11.2], Holt et al. [11.7] and Welsh and McKeag [11.10]. For projects 11.1 and 11.2, see Reynolds [11.9] and Perrott [11.8], respectively.

11.1 Brinch Hansen, P. *Operating System Principles*, Prentice-Hall, Englewood Cliffs, N.J. (1973).

11.2 Brinch Hansen, P. *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, N.J. (1977).

11.3 Hoare, C.A.R. "Towards a theory of parallel programming," in *Operating Systems Techniques* (eds., C. A. R. Hoare and R. N. Perrott), pp. 61–71, Academic Press, New York (1972).

11.4 Hoare, C. A. R. "Parallel programming: an axiomatic approach", *Computer Languages*, **1**, 151–60 (1975); also technical report CS-364, Computer Science Dept., Stanford University, Stanford, California (1975).

11.5 Hoare, C. A. R. "Monitors: an operating system structuring concept", *Comm. ACM*, **17** (10), 549–57 (1974).

11.6 Hoare, C. A. R. "Communicating sequential processes", *Comm. ACM*, **21** (8), 666–77 (1978).

11.7 Holt, R. C., G. S. Graham, E. D. Lazowska, and M. A. Scott, *Structured Concurrent Programming with Operating System Applications*, Addison-Wesley, Reading, Mass. (1978).

11.8 Perrott, R. H. "A language for array and vector processors", *ACM Trans. Prog. Lang. and Sys.*, **1** (2), 177–95 (1979).

11.9 Reynolds, J. C. "Syntactic control of interference", *Conf. Record of the 5th ACM Symp. Principles of Programming Languages*, pp. 39–46, ACM, New York (1978).

11.10 Welsh, J. and M. McKeag, *Structured System Programming*, Prentice-Hall International, London (1980).

12 TYPES

In principle, a mathematical function may be applied only to values that are in its domain of arguments. Applying an operation to a value that is *not* in its domain of arguments will be termed a ***domain incompatibility***. Dividing by zero, adding truth values, negating a character, and reading an empty file are typical examples of domain incompatibilities.

The first section of this chapter discusses several techniques for preventing domain incompatibilities. The remainder of the chapter will then focus on the most important of these: checking the *types* of program constituents before program execution.

12.1 PREVENTING DOMAIN INCOMPATIBILITIES

12.1.1 Domain Testing

Suppose the argument for an operation might *not* be in its domain of arguments. Should this be tested for by the implementation before carrying out the operation? The alternative would be to allow the operation to act on the representation of the value, even if this does not have a *semantically* well-defined result. There are few examples of the latter approach in good PASCAL processors, but consider

```
var i:integer;
begin
  write(i div 2);
    ⋮
end
```

Most PASCAL processors will treat the value initially contained in the new location as if it were an integer, though in principle it is "undefined".

There are some advantages to treating domain incompatibilities in this way. No time or storage space has to be devoted to domain testing, so that program execution is more efficient. This approach is also more flexible in that it allows programmers to access freely and "impersonate" the data representations used by an implementation. Thus, an operation may be applied to (representations of) values to which "abstractly" it is inapplicable, and a value that is *not* in the domain of arguments of an operation may be used to "impersonate" one that is (because their representations coincide). Such flexibility may allow an expert programmer to achieve the best possible execution efficiency.

But the *disadvantages* of this approach are very severe: if a "tolerated" domain incompatibility is in fact an inadvertent error (rather than a deliberate attempt to access the representation), the program's results will almost certainly be incorrect. Even if it is evident that errors have occurred, it will then be necessary to find their causes without any guidance as to where or when domain incompatibilities (if any) have occurred. Furthermore, interpreting the output will require knowledge of the data representations adopted by that particular processor, which programmers normally do not wish to be concerned with. Debugging in such circumstances is very time consuming and frustrating! Even when a program works correctly, it may be representation-dependent, and cannot reliably be used with other processors.

So, despite the efficiency and flexibility obtainable by omitting tests for domain incompatibilities, the risks and hidden costs of this approach are unacceptable in most circumstances. Except when execution efficiency is of critical importance, it is generally desirable for an implementation to test for domain incompatibility, and thereby help to prevent incorrect or representation-dependent output.

12.1.2 Coercion

What then is to be done by an implementation when an operation and its argument are incompatible? In general, two kinds of action are possible. One is to invoke an exception that aborts execution of the program and produces an error message. The other is to "convert" the argument or the operation so that they *are* compatible. Such an implicit conversion is termed a ***coercion***; the context is said to coerce the "improper" value into one that is proper, so that the computation can continue.

As an example of a coercion in PASCAL, consider assignment

$x{:=}i$

in the context of

```
var x:real;
    i:integer;
```

The integer contained in i is not in the domain required for location x. To allow the computation to continue, the representation of the number is converted to floating-point form and then treated as an element of the domain of real numbers. This is reasonable because the set of integer numbers is conceptually a subset of the reals and this interpretation is almost certainly what the programmer intends. In contrast, assignment

$i{:=}x$

is not allowed in PASCAL, and must be replaced by either

$i{:=}trunc(x)$

which truncates the fraction, or

$i{:=}round(x)$

which rounds to the nearest whole number.

The advantage of coercions is that they allow the programmer to omit explicit conversion, and some languages (such as PL/I and ALGOL 68) have very elaborate systems of coercions. However, if the domain incompatibility avoided by a coercion is in fact an *error*, the computation will be allowed to continue and possibly produce bizarre results. Programmers normally do not welcome error messages, but a message that helps in locating a bug is far more useful than meaningless output. For this reason, the introduction of coercions into programming languages should be done with considerable discretion.

When a language has both coercions *and* overloaded (Section 6.2.5) operators or procedure names, there are possibilities for ambiguity and programmer error. For example, in the context of

```
var x:real;
    i,j:integer;
```

assignment

$x{:=}i+j$

might be interpreted using either real or integer addition, depending on whether the integer-to-real conversion operation were applied to the arguments or the result, respectively. A good design principle is to ensure that in any such situation the results are equivalent.

12.1.3 Type Checking

The main objective of type checking is to determine *before* program execution whether a domain incompatibility can occur. If so, error messages or code for coercion or execution-time testing may be generated; if not, execution-time domain testing may safely be omitted.

For example, in the scope of PASCAL declaration

var *b*:*Boolean*;

expression '$-b$' has a type error because of incompatibility between the type of the operand and the type expected by the operator. If the program were executed, this would usually result in a domain incompatibility. (Of course, it is possible that the flow of control might avoid the error, or that the value actually is a number because of an insecurity.) On the other hand, expression '**not** *b*' is type-correct and is normally evaluated without testing whether *b* contains a truth value. (It may not if the implementation has insecurities.)

Replacing domain tests *during* execution (also known as "dynamic type checks") by type checks *before* execution has two main advantages. One is efficiency: a program component is typically executed many times but needs to be type-checked only once. Furthermore, type information may be used by implementations to improve efficiency in many other ways. The second advantage of type checks is that they catch minor programming errors before execution, and thereby simplify program testing and debugging. Type specifications also improve program readability by making explicit the data representations used by the programmer.

The main drawbacks of type checks are that (a) the syntax of a typed language must be more complex, and (b) restrictions must be imposed on the programmer's freedom of expression. Type checking should be used in languages to prevent domain incompatibilities whenever these disadvantages are outweighed by its benefits.

12.2 A CASE STUDY: TYPE CHECKING IN PASCAL

What are types? Different answers would be given for almost every language. In general, what can be said is that the type of a program component should be (a) reliable information about its semantic properties, yet (b) determinable syntactically, i.e., without requiring execution of the program. In this section, we shall discuss some aspects of the type system of PASCAL.

12.2.1 Indexing Types

Pre-defined types *Boolean*, *char*, and *integer*, programmer-defined enumerations, and subranges of these will be termed ***indexing*** types. The domains of values with which these types are associated should be evident.

(a) Type *Boolean* is associated with the domain of the two truth values, *true* and *false*.
(b) Type *char* is associated with an implementation-defined domain of characters that includes the 26 capital letters, the 10 decimal digits, and the "blank".
(c) Type *integer* is associated with the subset from $-maxint$ to *maxint* of the integers, where *maxint* is implementation-defined.
(d) An enumeration with n elements is associated with the domain of the first n natural numbers (or any domain isomorphic to it).
(e) If K_1 and K_2 are static expressions with values *min* and *max*, respectively, then the type expressed by

$$K_1..K_2$$

is associated with the subset of values from *min* to *max*.

In fact, it is convenient to regard *every* indexing type as a subrange. For example, predefined type identifier '*integer*' denotes a subrange type with bounds $-maxint$ and *maxint*.

Range limits are useful for both type checks before execution and tests during execution. Consider the following program fragment:

```
var  low: 0..10;
     med: 2..6;
     high: 8..12;
begin
  ⋮
  med:=high;
  low:=med;
  low:=high;
  ⋮
end
```

Although all of the locations may be expected to contain integers, assignment '*med*:=*high*' is erroneous because there is no overlap between the ranges of the source and target sub-expressions. Note that this kind of error

may be detected before execution (though in fact few PASCAL implementations do so). On the other hand, assignment '*low*:=*med*' may be executed without any execution-time domain testing because the range of the source is completely within the range of the target. Finally, assignment '*low*:=*high*' has some overlap, but in general it is impossible to be sure before execution whether the value of the source will always be in the range of the target; thus, execution-time testing would be desirable in this case. Even so, note that the value of *high* needs to be verified with respect to only the *upper* bound of the target's range, and not the lower bound.

In general, we will say that an indexing type t is ***index-compatible*** with another indexing type t' if t and t' are both subranges of the same basic type (either *Boolean*, *integer*, *char*, or an enumeration). If the range of values for t does not *overlap* the range of values for t', then a syntax error message may be generated. If the range of values possible is a *subset* of the range of values expected, then no execution-time testing is needed. It is evidently to a programmer's advantage to make use of subrange types and to specify bounds that are as "tight" as possible. Subrange type expressions also improve program readability and allow processors to minimize storage use.

We shall continue to use the letter t, possibly with primes or subscripts, to symbolize unspecified types, in the same way that we have been using T to stand for type expressions. Note that type expressions must be distinguished from the types they express, in part because static expressions and programmer-defined type identifiers may occur in type expressions.

12.2.2 Set Types

If T expresses an indexing type t, then type expression

set of T

Expresses the *set* type

set of t

Set of types are one of the composite kinds of type in PASCAL, in that they have type components, such as t. Type *set of t* is associated with the domain of all subsets of the domain with which t is associated. (In practice, t must be associated with a domain of relatively small cardinality, and in many implementations even the size of the *elements* must be small.)

Operation

E_1 **in** E_2

may then be type checked by requiring that the type of E_2 have the form *set of t* where the type of E_1 is index-compatible with t. The type of the whole expression is *Boolean*.

The set union operation

$E_1 + E_2$

may be type checked by requiring that the types of E_1 and E_2 be the form *set of* t_1 and *set of* t_2, respectively, where t_1 and t_2 are subranges of the same basic type. The tightest lower and upper bounds that may be estimated for the result type are the lesser of the lower bounds of t_1 and t_2, and the greater of the upper bounds of t_1 and t_2, respectively. The empty set literal must be treated as a special case. Type checking of the other set operations is similar.

12.2.3 Array Types

An ***array*** type has the form

array[t_1]*of* t_2

where t_1 must be an indexing type. Type *array*[t_1]*of* t_2 is associated with the domain of all functions *from* the domain with which t_1 is associated *to* the domain with which t_2 is associated.

Array subscripting operations of the form

L[E]

may be type checked as follows: (a) the type of L must be of the form *array*[t_1]*of* t_2, and (b) the type of E must be index-compatible with t_1. The type of the whole expression is t_2.

Note that in PASCAL the subscript bounds for a declared array are part of the *type* of that array name. This has two significant advantages.

(a) The index compatibility check described in Section 12.2.1 eliminates the need for many execution-time subscript bound checks, without any loss of security. Furthermore, for the tests that *are* required, the bounds are known before execution, and this will often improve the efficiency of such tests.

(b) The storage requirements for arrays are known *before* execution, which permits a simpler and sometimes more efficient implementation than is possible in languages (such as ALGOL 60 and PL/I) that allow the declared subscript bounds for an array to have been computed *during* execution. (These are known as *dynamic* array bounds, but should not be confused with the *flexible* array bounds in ALGOL 68, which may be changed after creation of the array.)

The major *disadvantage* of the approach in PASCAL is, of course, that it does not allow the size of an array to be data-dependent. Files or pointers may be used in PASCAL when the size of a storage structure cannot be estimated before execution.

The treatment of array types in PASCAL also affects the use of procedures with array parameters. Consider the following definitions:

```
type ta=array[1..m,1..n]of real;
     tb=array[1..n,1..m]of real;
  ⋮
procedure transpose(var a:ta; b:tb);
  var i:1..m; j:1..n;
  begin
    for i:=1 to m do
      for j:=1 to n do
        a[i,j]:=b[j,i]
  end;
```

Procedure *transpose* updates *a* to be the matrix transpose of *b*. However, this procedure may be used only on matrices of the *particular* dimensions *m* and *n* specified in the definitions for types *ta* and *tb*. If matrices of *various* sizes are to be transposed within a single program, then several essentially similar procedure definitions must be provided by the programmer. It is evident that this is very inconvenient and error-prone.

Some PASCAL implementations allow formal parameters to be specified by type expressions of the form

$$\textbf{array}[\cdots; I_1..I_2{:}I_3;\cdots]\textbf{of } I_4$$

In this, I_1 and I_2 are binding occurrences of identifiers, and I_3 and I_4 are applied occurrences of type identifiers. During the execution of the procedure body, I_1 and I_2 denote the lower and upper bounds of the subscript type of the corresponding actual parameter. For example, a more general matrix transposing procedure could then be defined as follows:

```
procedure transpose
  (var a:array[m1..n1:integer; m2..n2:integer]of real;
       b:array[m3..n3:integer; m4..n4:integer]of real);
  var i,j:integer;
  begin
    if (m1=m4) and (m2=m3) and
      (n1=n4) and (n2=n3)
      then for i:=m1 to n1 do
             for j:=m2 to n2 do
               a[i,j]:=b[j,i]
      else error
  end;
```


Note that this approach allows much less type checking than in the original form of the definition. For example, in the above, it is necessary to check *during* execution that the dimensions conform for transposition. Also, it would be quite complicated to verify before execution that subscripts are in range. In the original procedure, all of these checks could easily be made before execution, thereby improving the efficiency and security of the program. Note also that the lower and upper bound identifiers *m*1, *n*1, ··· are not usable as static expressions. This is why '*i*' and '*j*' had to be declared to be of type *integer*. In Section 12.3, an alternative approach which does not have these disadvantages will be described.

12.2.4 Record Types

If we temporarily ignore variant parts of **record** type expressions, a *record* type is a composite of the form

$$\textit{record } I_1{:}t_1;\ I_2{:}t_2;\cdots;\ I_n{:}t_n\ \textit{end}$$

where the I_i are the field names and the t_i are the corresponding types. Such a record type is associated with the *product* of the domains with which the field types are associated.

Expression

L.I

may then be type checked by requiring that the type of L be a record type having I as a field name. Then, the type of the whole expression is the corresponding field type. This is sufficient to ensure that no testing is necessary during execution.

However, consider now a **record** type expression with a "variant" part, as in

```
type sex=(male, female);
     person=
       record
         name: string;
         case s:sex of
           male:(bearded:Boolean);
           female:(pregnant:Boolean)
       end;
```

The variant part of a record type is associated with the *sum* of the domains with which the field lists of the variants are associated. Thus, in the context of

```
var p:person;
```

field *p.name* and the ***tag field***, *p.s*, are always accessible, but field *p.bearded* is meaningful only when *p.s=male*, and field *p.pregnant* only when *p.s=female*.

These requirements cannot be enforced using the kind of type check that was adequate for records without variant parts. But it is quite inefficient to test the value of the tag field at *every* use of a variant field. It is also quite complicated or even infeasible to implement such tests because of nested variant parts, the implicit bindings of field names set up by the **with** command, and the fact that the programmer can specify that the tag field is to be omitted. For these reasons, variant parts of records are insecure in most PASCAL processors.

A simple extension to the language would often reduce the need for testing during execution. Suppose, as suggested in Section 6.2.3, that the **with** command in PASCAL were to require a listing of which field names are to be bound, as in

```
with (name, bearded)=p do
  begin
    ···name···bearded···
  end
```

or

```
with (name, pregnant)=p do
  begin
    ···name···pregnant···
  end
```

It could then be checked before execution that all of the field names listed were meaningful for the same value of the tag field. A test would be necessary during execution to ensure that the tag field of the record currently had that value, but then no *further* such testing would be required in the body of the construct for the field names listed, provided that the record itself were not updated. Another possibility is to treat the field names listed in the same way as value parameters (i.e., bind them to new storage that is initialized by the value of the corresponding field); then the record itself could be accessed and updated without affecting the security of the field names listed.

This extended **with** construct would often be used in conjunction with the **case** command, as in


```
case p.s of
  male:   with (name, bearded)=p do
            begin···name···bearded···end;
  female: with (name, pregnant)=p do
            begin···name···pregnant···end
end
```

It would be convenient and more efficient to have a construct that combines these effects, as follows:

```
case p.s of
  male(name, bearded):
    begin···name···bearded···end;
  female(name, pregnant):
    begin···name···pregnant···end
end
```

Many languages have facilities similar to this. For example, the equivalent in ALGOL 68 of the above is

```
mode male=struct(bool bearded),
mode female=struct(bool pregnant),
mode person=struct(string name, union(male,female) s),
      ⋮
case s of p in
  (male m):
    begin···name of p···bearded of m···end,
  (female f):
    begin···name of p···pregnant of f···end
esac;
  ⋮
```

12.2.5 File Types

A ***file*** type has the form

file of t

and, as discussed in Section 4.6.2, is associated with values of the form $(f\uparrow, \tilde{f}, \tilde{f})$, where $f\uparrow$ and all the components of the finite sequences $\tilde{f}$ and $\tilde{f}$ are in the

domain with which the component type t is associated. Because the operations *read* and *write* are defined in terms of assignments to and from a file buffer, the component type of a file must be an ***assignable*** type. For pragmatic reasons, many PASCAL processors do not allow assignment of "large" values. The *non*-assignable types are typically files, and records or arrays with non-assignable components. For example, a file of files would not be permitted.

12.2.6 Pointer Types

A ***pointer*** type has the form

$$\uparrow t$$

and is associated with the domain of storage structures for the domain with which t is associated, plus the *nil* value. An expression of the form

$$E\uparrow$$

may then be type-checked by requiring that the type of E be a pointer type $\uparrow t$ (or a file type). The type of the whole expression is then t. However, this check cannot ensure that the value of E is not *nil*, which would be a domain incompatibility. Fortunately, this can be tested for during execution with little or no overhead in most implementations.

12.2.7 Type Equivalence

In a few contexts in PASCAL it is necessary to test whether the type of an expression is the *same* as the type of another expression or of a formal parameter. One such context is procedure invocation. The actual parameter corresponding to a **var** formal parameter must be an *l*-expression of the type specified in that formal parameter. This is because the formal parameter identifier will be bound to the *l*-value of that *l*-expression during executions of the procedure body.

For value parameters and assignments, only *assignable* types are allowed, but the compatibility requirements are relaxed to allow for *integer*-to-*real* coercion, index compatibility of subrange types, and so on. Unless one of these specific "loopholes" is applicable, the type of the source (or actual parameter) expression must be the same as the type of the target *l*-expression (or formal value parameter).

But when are two types "the same"? In some PASCAL processors, types are distinguished by their structure alone, and not by how they happen to have been expressed in the program. For example, in the context of

```
type t=record a:real; b:char end;
var x:array[Boolean]of t;
    y:array[Boolean]of record a:real; b:char end;
```

both '*x*' and '*y*' would have type

```
array[Boolean]of record a:real; b:char end
```

without regard to how it was expressed in the program. This approach is known as ***structural*** equivalence of types.

In most PASCAL processors, a different approach is used. Each *occurrence* in a program of a type expression (except a type identifier) is interpreted as expressing a type distinct from any other type expressed in that program, even those that are structurally similar. For example, '*x*' and '*y*' above would have different types, and the types defined by

```
type coordinate = record first, second:real end;
     complex = record first, second:real end;
```

would be distinct. Thus, in the scope of

```
var p:coordinate; z:complex;
```

both '*z*:=*p*' and '*p*:=*z*' would be in error.

This approach has been termed "name" equivalence of types, but this is rather misleading. Similar effects would be obtained even if the types were anonymous. For example, '*x*' and '*y*' have the *same* type in the scope of

```
var x,y:record first, second:real end;
```

and *distinct* types in the scope of

```
var x:record first, second:real end;
    y:record first, second:real end;
```

Furthermore, the assignment in

```
type t=···;
var x:t;
procedure q;
  type t=···;
  var y:t;
  begin
      ⋮
    x:=y;
      ⋮
  end;
⋮
```

is erroneous even though the types of 'x' and 'y' happen to have the same name. This approach would be better termed ***occurrence*** equivalence of types, because it regards types as being the same only if they are expressed by the same occurrence of a non-trivial type expression.

The motivation for adopting occurrence equivalence is to permit the programmer to make type distinctions that are more refined than the structural differences built into the language. However, this objective is only partially achieved by using occurrence equivalence. For example, if 'z' has type *complex* and '*p*' has type *coordinate* then '*z*:=*p*' is rejected, but its effect may still be achieved *selectively* by executing

```
z.first:=p.first;
z.second:=p.second
```

Because the *representation* of the "new" types is accessible, the intended distinction between them may be subverted by using these representation operations.

In Section 12.4 we shall discuss facilities that allow programmers to create *secure* "new" types.

12.2.8 Procedural Parameter Types

If the procedure name I in an invocation

I(···;E;···)

is a formal parameter identifier, type checking of actual parameters E, requires knowledge of the types of the corresponding formal parameters of that procedure. Some PASCAL processors require or allow a *specifier* for the formal parameter to supply this information. However, some processors do not require such type specification, and as a result, are either inefficient (because they do testing during execution) or insecure (because they do not do such testing).

12.3 STATIC AND POLYMORPHIC PROCEDURES

The principle of abstraction (Section 7.4) suggests that in a language that has static expressions and type expressions (such as PASCAL), it is possible to have facilities for defining procedures by abstraction from these syntactic categories. Furthermore, the principle of correspondence (Section 9.1) suggests that in a language with **const** and **type** *definitions*, it would also be possible to have corresponding **const** and **type** *parameters*. In this section we shall be exploring these possibilities.

12.3.1 Static Procedures

It may be recalled that a static expression is one that may be evaluated *before* execution of the program. The syntax of PASCAL permits only very simple forms of static expression, but some processors allow any expression that is composed out of literals, **const** or enumeration identifiers, operators, and pre-defined identifiers.

A ***static expression procedure*** would be one whose definition body and invocations are static expressions. A PASCAL-like syntax for defining such procedures could be obtained by extending the **const** form with a formal parameter list, as follows:

const I(···;P;···)=K;

where K is a static expression. For example,

const *sum*(**const** *n*)=*n**(*n*+1) **div** 2;

defines a static expression procedure which could be invoked as follows:

var *a*:**array**[1..*sum*(*m*)]**of** *real*;

where '*m*' must be a static expression.

A similar facility would allow programmer-defined ***type expression procedures***:

type I(···;P;···)=T;

Here are two examples:

```
type string(const n)=array[1..n]of char;
     pairof(type t)=record first,second:t end;
```

These might be invoked as follows:

```
type card=string(80);
     line=string(132);
     complex=pairof(real);
     rational=pairof(integer);
```

12.2.2 Polymorphic Procedures

Let us now consider the possibility of using **type** and **const** parameters in conventional (i.e., non-static) procedures. It may be recalled that procedures with array parameters (such as *transpose*) are unsatisfactory in PASCAL because they are specialized to *particular* array dimensions. Suppose that the syntax of PASCAL allowed type expressions (as well as type identifiers) in parameter specifications, as in

```
const m=···; n=···;
  ⋮
procedure transpose(var a:array[1..m,1..n]of real;
                        b:array[1..n, 1..m]of real);
  var i:1..m; j: 1..n;
  begin
    ⋮
  end;
```

This makes it evident that the reason the procedure is over-specialized is that identifiers '*m*' and '*n*' are bound in the context of the *definition* of the procedure, rather than in the contexts of its invocations. Rather than use dynamic binding, these identifiers should become **const** parameters of the procedure, as follows:

```
procedure transpose(const m,n;
                    var a:array[1..m, 1..n]of real;
                        b:array[1..n, 1..m]of real);
  var i:1..m; j:1..n;
  begin
    ⋮
  end;
```

Actual parameters corresponding to the formal **const** parameters must be static expressions. Then the usual type checking of the other parameters and the body would always be possible, using the values of these static expressions. If a program contained several such invocations with distinct arguments, the effect would be identical in every respect to that obtainable in PASCAL by defining several procedures, each particularized to a specific set of dimensions *m* and *n*. Thus with this approach type checking is much easier than with the alternative approach discussed in Section 12.2.3, and this would provide better execution efficiency and security. Procedures that may

be used with many types of parameters are termed ***polymorphic*** (or "generic").

It is in fact possible to simplify the notation for invoking such polymorphic procedures. Consider the following program fragment, which contains an invocation of *transpose*:

```
const k=10; l=12;
var x:array[1..k,1..l]of real;
    y:array[1..l,1..k]of real;
begin
  ⋮
  transpose(k,l,x,y);
  ⋮
end
```

It should be evident that a processor could use a straightforward "pattern match" to deduce the values of '*k*' and '*l*' from just the types of the *other* actual parameters and the types of the corresponding formal parameters. This suggests allowing actual static parameters to be omitted from invocations of polymorphic procedures, as in

```
transpose(x,y)
```

The definition of *transpose* might then be changed as follows to separate the static and dynamic parts of the parameter list:

```
procedure transpose[const m,n]
                   (var a:array[1..m,1..n]of real;
                        b:array[1..n,1..m]of real);
  var ···
  begin
    ⋮
  end;
```

Here is another example of a polymorphic procedure, this time using a **type** parameter as well as **const** parameters:

```
procedure maparray[const m,n; type t]
                  (var a:array[m..n]of t;
                   procedure p(var x:t));
  var i:m..n;
  begin
    for i:=m to n do p(a[i])
  end;
```

This procedure applies a procedure *p* to each component of an array *a*, where the subscript bounds and the base type of the array are static parameters. An example of an invocation of *maparray* might be

```
var tapes:array[1..nt]of text;
begin
  maparray(tapes, rewrite);
  ⋮
end
```

Note that it would usually be impractical to attempt to compile code for polymorphic procedures independently of their invocations. Thus, implementation of these facilities would involve some "administrative overhead" (comparable to that for a macro-assembler). This does not seem an excessive price to pay for the benefits they would provide. A more serious disadvantage is that even the type correctness of a procedure definition having **const** or **type** parameters cannot in general be checked independently of the invocations of that procedure. This issue will be addressed in Section 12.4.6.

12.4 NEW TYPES

12.4.1 Basic Concepts

Suppose that a programmer wishes to develop a program using a domain of values that is *not* a semantic primitive of the language being used, say "stacks" of characters equipped with operations for clearing, pushing, and popping a stack, selecting the top component, and testing for emptiness. In accordance with familiar programming principles of modularity and separation of levels of abstraction, it would be desirable to separate

(a) the *implementation* part of the program, which describes the *representation* for the domain of stacks and its operations in terms of some available domain of values, from
(b) the *logical* part of the program, where stacks are used (without reference to their concrete representation).

In the logical part of the program, the following would be regarded as domain incompatibilities:

(1) impersonation: applying a *stack* operation (such as *pop*) to an argument in some *other* domain (including the representation domain);
(2) unauthorized access: supplying a *stack* as the argument to an operation expecting an argument in some *other* domain (including the representation domain).

A language that claims to provide a facility for creating "new" types must be able to prevent these two abuses.

PASCAL and ALGOL 68 are often described as languages that allow programmers to define "new" types, but by this criterion, they do *not*. Consider PASCAL fragment

```
type stack=···{representation type}···;
function top(s:stack):char;
  begin···{representation operation}···end;
⋮
```

The elided type expression and procedure definition bodies describe the representation of stacks in terms of some existing domain of values. Thus they make up the implementation part of the program. However, if the PASCAL processor uses structural equivalence of types, then 'stack' is just another name for the representation type. This means that type checking would *not* prevent impersonation of a stack by a value in the representation domain *or* unauthorized access to a stack by a representation operation. If occurrence equivalence of types is used, direct impersonation is prevented. However, there is no restriction on access to the representation of a stack. As pointed out in Section 12.2.7, this also allows component-by-component construction of an impersonating value.

So, the type checks in PASCAL and ALGOL 68 only ensure the security of the domains "built into" these languages. In the following, we discuss facilities that may be used to get the same security for programmer-defined domains.

12.4.2 Definition Procedures

Some programming languages allow invocations of definition procedures (Section 9.6) to be used as type expressions. This makes it possible for a programmer to get the effect of creating a "new" type (in the sense discussed above) by keeping the representation "private" to the operations. For example, consider the following **class** definition:

```
class stack;
  private
    var a:array[1..n]of char;
        p:0..n;
  within
    procedure clear;
      begin p:=0 end;
    procedure push;
      begin p:=p+1 end;
    procedure pop;
      begin p:=p-1 end;
    selector top:char;
      a[p];
    function empty:Boolean;
      begin empty:=p=0 end
end;
```

If class invocations were allowed as type expressions, '*stack*' could then be used as follows:

```
var sa:array[1..m]of stack;
    k:1..m;
    ch:char;
begin
    ⋮
  sa[k].clear;
  sa[k].push;
  sa[k].top:=ch;
    ⋮
end
```

Type checks would ensure that the stack operations were applied *only* to stacks, and that stacks were operated upon *only* by these "authorized" operations.

So, it has been demonstrated that in some circumstances new types may be created using **class** definitions. However, this approach has a significant limitation: an operation defined in a **class** body can only access the representation of *one* instance of *that* class. For example, consider adding the following to the operations on stacks:


```
function isequalto(var s:stack):Boolean;
  var eq:Boolean;i:0..m;
  begin
    eq:=p=s.p; i:=p;
    while eq and (i>0) do
      if a[i]=s.a[i]
        then i:=i-1
        else eq:=false;
    isequalto:=eq
  end;
```

Then,

```
x.isequalto(y)
```

would test whether stack instances x and y were equal. However, the references in the definition of the procedure to the *private* identifiers of the *stack* parameter s are, in principle, forbidden. Some languages permit such access within the body of the class definition itself, but this is an *ad hoc* exception. More importantly, it does not allow for *mutually dependent* types, i.e., types whose operations must access the representation of instances of more than one new type. For example, *line* and *point* in a geometry package, or *node* and *edge* in a graph package might be mutually dependent types. The following section describes an approach which does not have this limitation.

12.4.3 newtype Definitions

The following (simultaneous) definitions illustrate a more flexible approach to creating new types:

```
newtype stack=record
                a:array[1..m]of char;
                p:0..m
              end;
procedure clear(var s:stack);
  begin s.p:=0 end;
procedure push(var s:stack);
  begin s.p:=s.p+1 end;
procedure pop(var s:stack);
  begin s.p:=s.p-1 end;
selector top(var s:stack):char;
  s.a[s.p];
function empty(var s:stack):Boolean;
  begin empty:=s.p=0 end;
```

The only unusual feature is the use of keyword '**newtype**' in place of the familiar '**type**'. The remaining definitions describe the operations for the new type in terms of operations on the *representation*, which is described on the right-hand side of the **newtype** definition. The right-hand sides of these definitions constitute the *implementation* part of the program. Type checks will ensure that the representation operations are compatible with the representation type.

In the scope of these definitions (i.e., the *logical* part of the program), type *stack* is distinct from all other types, including the representation type. Type checks will then ensure that stack operations are applied only to stacks (no impersonation), and that stacks are operated upon only by stack operations (no unauthorized access).

So, this approach also defines secure new types. Furthermore, it does not have the limitations of the other approach. For example, the implementation of an equality operation on stacks may be described with the other *stack* operations as follows:

```
function equal(var s,t:stack):Boolean;
  var eq:Boolean; i:0..m;
  begin
    eq:=s.p=t.p; i:=s.p;
    while eq and (i>0) do
      if s.a[i]=t.a[i]
        then i:=i-1
        else eq:=false;
    equal:=eq
  end;
```

Mutually dependent types may be specified as well, as in

```
newtype point=···;
        line=···;
procedure intersects(var p:point; var l1,l2:line);
  {computes the point p where two non-parallel lines l1 and
       l2 intersect}
  begin···end;
  ⋮
```

Facilities similar to **newtype** definitions are provided in several recent languages, including MODULA and ADA.

12.4.4 Inheritance

It may be recalled from Section 12.2.5 that in PASCAL certain types (files and structures with file components) are *non-assignable* and do not enjoy certain "privileges". The following are allowed only if *t* is an assignable type:

(a) assignments with expressions of type *t*,
(b) value parameters of type *t*,
(c) a type *file of t*.

Furthermore, array and record types are assignable only if all of their components are assignable. Similar distinctions are found in almost every practical programming language because of the inefficiency involved in copying "large" values.

Should *new* types be assignable? It would be desirable for some (e.g., a type *complex* for manipulating complex numbers); however, for others (e.g., *stack*) it would not, even if the representation were assignable. This suggests that the programmer should be able to specify whether or not a new type is to ***inherit*** the assignment operation from its representation.

We shall use definitions of the form

newdatatype I=T;

to bind I to a new *assignable* type. The updating operation for the new type is implemented by the updating operation on the representation, so that the type expressed by T must itself be assignable. The **newtype** form creates types that do *not* inherit the assignment operation and do not enjoy the privileges of assignable types.

For example, a new assignable type *complex* could be defined as follows:

```
newdatatype complex=record
                      realpart,imagpart:real
                    end;
procedure add(var z1:complex; z2:complex);
  begin
    z1.realpart:=z1.realpart+z2.realpart;
    z1.imagpart:=z1.imagpart+z2.imagpart
  end;
  ⋮
```

A similar approach to inheritance may be taken for *indexing* types (Section 12.2.1), a subset of the assignable types which enjoy additional privileges in PASCAL:

(a) operations *succ*, *pred*, and *ord*,
(b) equality and ordering relations,
(c) use as array subscripts and set elements,
(d) **for** iteration and **case** selection.

When it is convenient to allow inheritance of these privileges by a new type, this will be specified by a definition of the form

```
newindextype I=T;
```

where the type expressed by T must be an indexing type.

For example, in a program to compute "stable marriages" from priority rankings of possible spouses, the following definitions might be used:

```
newindextype
     man=1..n;
   woman=1..n;
    rank=1..n;
   ⋮
var
  m:man; w:woman; r:rank;
  wmr:array[man,rank]of woman;
  mwr:array[woman,rank]of man;
  rmw:array[man,woman]of rank;
  rwm:array[woman,man]of rank;
  wife:array[man]of woman;
  husb:array[woman]of man;
  single:set of woman;
begin
  ⋮
end
```

Type checks would then help to ensure that men, women, rankings, and integers were not confused.

12.4.5 New Type Constructors

It is often useful to define new type constructors, analogous to built-in type constructors like **array**[T]**of** T and **set of** T. This is possible by allowing **newtype** definitions to have static parameters, as in


```
newtype matrix(const m,n)=⋯;
selector access[const m,n]
                (var a:matrix(m,n); i:1..m; j:1..n):real;
  ⋯ ;
procedure transpose[const m,n]
                (var a:matrix(m,n); var b:matrix(n,m));
  var i:1..m; j:1..n;
  begin ⋯ end;
procedure matmult[const n1,n2,n3]
                (var a:matrix(n1,n2);
                 var b:matrix(n1,n3);
                 var c:matrix(n3,n2));
  var i1: 1..n1; i2: 1..n2; i3: 1..n3; x:real;
  begin ⋯ end;
  ⋮
```

In some cases, it is convenient to bind parameters for the whole set of simultaneous definitions. A parameterized **class** definition may be used for this, as in the following:

```
class stackof(type t; const m);
  newtype stack=record
                  a:array[1..m]of t;
                  p:0..m
                end;
  procedure clear(var s:stack);
    begin s.p:=0 end;
  procedure push(var s:stack);
    begin s.p:=s.p+1 end;
  procedure pop(var s:stack);
    begin s.p:=s.p-1 end;
  selector top(var s:stack):char;
    s.a[s.p];
  function empty(var s:stack):Boolean;
    begin empty:=s.p=0 end
end;
```

Parameters *t* and *m* are bound once and for all when the class is invoked. Here is an example of how it might be used:

```
x.stackof(char,32);
var q:x.stack; c:char;
begin
  x.clear(q);
  read(c);
  while c≠'$' do
    begin
      x.push(q);
      x.top(q):=c;
      read(c)
    end;
  :
end
```

12.4.6 newtype Parameters

We saw in the preceding sections that simultaneous definitions of the form

```
newtype I=T;
 :
procedure I_i(···;I_p:I;···); C_i;
 :
```

may be used to create new types. The right-hand sides of these definitions make up the *implementation* part of the program. Type expression T describes a representation for the new type, and procedure body C_i describes an implementation of an operation for the new type in terms of operations on the representation.

In this section we will consider the possibility of using **newtype** parameters that correspond (in the sense of the principle of correspondence) to **newtype** definitions. It would then be possible to have polymorphic procedures with formal parameter lists of the form

```
(newtype I;···; procedure I_i(···;I_p:I;···) ;···)
```

or, using the "pattern matching" discussed in Section 12.3,

```
[newtype I](···;procedure I_i(···;I_p :I;···) ;···)
```

Similarly, **newdatatype** and **newindextype** parameters are possible.

Here is an example:


```
procedure maparray[newtype t]
                  (var a:array[m..n]of t;
                   procedure p(var x:t));
  var i:m..n;
  begin
    for i:=m to n do p(a[i])
  end;
```

This is similar to the procedure defined in Section 12.3, but with parameter '*t*' specified by **newtype**, rather than **type**. The difference is that the body of the procedure definition is treated as the "logical" part of the program and is type checked independently of the representations supplied by invocations of the procedure. This means that the procedure definition can only assume properties of type *t* that are specified in its formal parameter list, i.e., an array *a* with components of type *t*, and a procedure *p* whose **var** parameter is of type *t*. This type checking ensures that the procedure will work correctly for all invocations whose actual parameters match these specifications.

A more general *maparray* procedure could be obtained by also abstracting with respect to the *subscript* type of the array and the subscript limits:

```
procedure maparray[newindextype ind; newtype t]
                  (var a:array[ind]of t;
                   procedure p(var x:t);
                   m,n:ind);
  var i:ind;
  begin
    for i:=m to n do p(a[i])
  end;
```

Similarly, here is the definition of a general (but inefficient) array sorting procedure:

```
procedure sort[newindextype ind; newdatatype d]
              (m,n:ind; var a:array[ind]of d;
               function lessthan(x,y:d):Boolean);
  var i,j,min:ind; x:d;
  begin
    for i:=m to pred(n) do
    begin
      min:=i;
      for j:=succ(i) to n do
        if lessthan(a[j],a[min]) then
          min:=j;
      x:=a[min]; a[min]:=a[i]; a[i]:=x
    end
  end;
```

These facilities make it possible to define procedures that are very general, yet may be type checked independently of their applications. Consequently, they are very suitable for inclusion in procedure libraries.

EXERCISES

12.1 Describe how the **case** and **for** constructs in PASCAL should be type-checked.

12.2 In Section 12.4.4, several of the examples involving non-assignable new types used a **var** parameter in a procedure that did *not* require that the parameter be an *l*-expression. It would be desirable in such cases to be able to use a parameter form that would allow verification that the formal parameter was not used as an *l*-expression, yet would be *implemented* like the **var** form. Formal parameters of the form

```
val I:T
```

could be used for this. For example, procedure *transpose* might then be defined as follows:

```
procedure transpose[const m,n]
                   (var a:matrix(m,n);
                    val b:matrix(n,m));
  ...;
```

What additional constraint is necessary if this is to work as expected? *Hint*: consider invocation

```
transpose(x,x)
```

12.3 What implementation difficulties may arise with *recursive* polymorphic procedure definitions? What constraints would avoid the difficulties?

12.4 Would it be possible or desirable for the equality operation ('=') to be inherited along with assignment when **newdatatype** is used?

12.5 A **semitype** form of definition has been proposed. It is to have the same effect as a **newtype** definition, except that representation operations *would* be allowed to have parameters of the defined type. For example, in the scope of

```
semitype complex=record realpart,imagpart:real end;
procedure add(var z1,z2: complex);
  begin···end;
  ⋮
var z:complex;
```


'*add*(*x*,*y*)' is allowed only when '*x*' and '*y*' are declared to have type *complex*, but it *is* possible to use '*z.realpart*' and '*z.imagpart*'.

(a) Show that impersonation may still be possible with this approach.
(b) Suggest a constraint that would prevent impersonation.
(c) Show how the effect of a **semitype** definition may be simulated using a **newtype** definition and a conversion operation.

*12.6 By using pointer types and recursive type definitions it is possible to define *infinitary* types in PASCAL. For example, the types defined by

```
type ref = ↑node;
     node= record
             info:integer;
             link:ref
           end;
```

are infinitary. Describe how structural equivalence of such types may be tested.

*12.7 The notion of *type* was originally developed in mathematics to avoid the following inconsistency, discovered by B. Russell. Suppose that it were possible to define a set R whose elements are all the sets that do not contain themselves. That is,

$$R=\{S|S\notin S\}$$

But then, $R\in R$ implies that $R\notin R$, and $R\notin R$ implies that $R\in R$. One way to prevent this contradiction is to use type restrictions to make the definition syntactically illegal.

Show that this example may be simulated in a version of PASCAL that does not require specification of parameters of procedural parameters, but is not expressible in versions of PASCAL that require such specification. *Hint*: Use characteristic functions to simulate sets.

*12.8 In some languages (including ALGOL 68), it is possible to define procedural types. An example in a PASCAL-like notation is

```
type p=function(p; integer):integer;
function f(g:p; n:integer):integer;
  begin
    if n=0 then f:=1 else f:=n*g(g,n-1)
  end;
```

(a) What would be the value of '*f*(*f*,*i*)' for $i\geq 0$?
(b) Would *f* be a different procedure if its definition were interpreted non-recursively?
(c) Simulate Russell's example of the preceding exercise using this notation. What would happen if an attempt were made to compute the analog of $R\in R$?

PROJECTS

12.1 Design a facility that would allow a programmer to define a ***subtype***, i.e., a type that is associated with some *subset* of an existing domain. The subset would be characterized by some predicate (i.e., truth-valued expression procedure) on the values of the base domain. Also, show how to simulate the effect of this using a **newtype** definition and conversion operations.

12.2 Design an extension to PASCAL that would allow declaration of arrays with dynamic (i.e., data-dependent) array bounds.

12.3 Design facilities that would allow programmer-defined coercions and operator overloading.

BIBLIOGRAPHIC NOTES

Type checking in PASCAL has been discussed by Habermann [12.1], Lecarme and Desjardins [12.3], Welsh et al. [12.13, 12.14], Wirth [12.15] and Tennent [12.12]. The material in Sections 12.3 and 12.4 is based on Hoare [12.2], Morris [12.5, 12.6], Reynolds [12.7, 12.8], Tennent [12.10, 12.11], and Milner [12.4]; however, some of the facilities described are controversial and have never (to my knowledge) been implemented. For project 12.3, see a paper by Reynolds [12.9], which is also the source of the consistency condition for coercions and overloaded operators described in Section 12.1.2.

12.1 Habermann, A. N. "Critical comments on the programming language PASCAL", *Acta Informatica*, **3**, 47–57 (1973).

12.2 Hoare, C. A. R. "Proof of correctness of data representations", *Acta Informatica*, **1**, 271–81 (1972).

12.3 Lecarme, O. and P. Desjardins. "More comments on the programming language PASCAL", *Acta Informatica*, **4**, 231–43 (1975).

12.4 Milner, R. "A theory of type polymorphism in programming", *J. Comp. Sys. Sci.*, **17**, 348–75 (1978).

12.5 Morris, J. H. *Lambda Calculus Models of Programming Languages*, Ph.D. thesis and report TR-57, Project MAC, M.I.T., Cambridge, Mass. (1968).

12.6 Morris, J. H. "Types are not sets", *Conference Record of the ACM Symposium on Principles of Programming Languages*, Boston, 120–4, ACM, New York (1973).

12.7 Reynolds, J. C. "Towards a theory of type structure", Colloque sur la programmation, *Lecture Notes in Computer Science*, **19**, 408–23, Springer, Berlin (1974).

12.8 Reynolds, J. C. "User-defined types and procedural data structures as complementary approaches to data abstraction", in *New Directions in Algorithmic Languages* 1975 (ed., S. Schuman), pp.157–68, IRIA, Rocquencourt, France (1975), also in *Programming Methodology* (ed., D. Gries), pp.309–17, Springer, New York (1978).

12.9 Reynolds, J. C. "Using category theory to design implicit conversions and generic operators", Proc. of the Aarhus Workshop on Semantics-Directed Compiler Generation, *Lecture Notes in Computer Science*, Springer, Berlin (1980).

12.10 Tennent, R. D. "Language design methods based on semantic principles", *Acta Informatica*, **8**, 97–112 (1977).

12.11 Tennent, R. D. "On a new approach to representation-independent data classes", *Acta Informatica*, **8**, 315–24 (1977).

12.12 Tennent, R. D. "Another look at type compatibility in PASCAL", *Software Practice and Experience*, **8**, 429–37 (1978).

12.13 Welsh, J. "Economic range checks in PASCAL", *Software Practice and Experience*, **8**(1), 85–97 (1978).

12.14 Welsh, J., W. J. Sneeringer and C. A. R. Hoare. "Ambiguities and insecurities in PASCAL", *Software Practice and Experience*, **7**, 685–96 (1977).

12.15 Wirth, N. "An assessment of PASCAL", *IEEE Trans. Software Engineering*, **1**, 192–8 (1975).

13 FORMAL SEMANTICS

In preceding chapters the semantics of programming language constructions were described *informally*. We saw in Section 2.6 that it is desirable to have *formal* descriptions of the syntax of languages. Similarly, a formal description of the *semantics* of a programming language is a precise specification of the meanings of programs, for use by programmers, language designers and implementers, and in theoretical investigations of language properties.

The basic idea of the approach we shall describe is to define *functions* that map syntactic structures into mathematical objects (such as numbers, truth values and functions) that model their meaning. The definitions of semantic functions will be *denotational*, that is to say, structured so that the meaning of any composite phrase is expressed in terms of the meanings of its immediate constituents. In the following sections this approach will be demonstrated for several simple languages.

13.1 BINARY NUMERALS

In Section 1.3, we discussed the syntax and semantics of binary numerals. This may be formalized as shown in Table 13.1. Semantic function $\mathcal{N}$ maps every binary numeral into the number that it denotes. The three sections of the table define (a) **Nml**, the syntactic domain of binary numerals, (b) **N**, the semantic domain of natural numbers, and (c) the semantic function, $\mathcal{N}$: **Nml** → **N**, using one "equation" for each of the possible forms of binary numeral. As a visual aid, syntactic operands are enclosed by special brackets "⟦" and "⟧".

13.2 A SIMPLE PROGRAMMING LANGUAGE

The same approach may be used to express formally the semantics of much more complex languages. We shall demonstrate this with a fairly simple programming language that does not have type expressions, side effects, data structures or, for the present, local bindings and jumps. The complete description (except for a concrete syntax) is given in Table 13.2. The rest of this section provides commentary on this formal description and should be read concurrently with it.

From the abstract syntax it may be seen that the forms of expression are: two literals, '0' and '1'; two unary operations ('−' and '**not**'); two binary operations ('+' and '='); global identifiers; parameter-less command procedure abstracts; and parentheses as brackets. The forms of command are: the null command; the assignment; the procedure invocation (**call**); control stuctures for sequential (';'), selective (**if**), and iterative (**while**) composition; and **begin**···**end** for bracketing. A program consists of a command with an

Abstract syntax

N∈**Nml** binary numerals

N ::= 0 | 1 | N0 | N1

Semantic domain

N={*zero*}+**N** natural numbers
={0, 1, 2, ···}

Semantic function

$\mathcal{N}$: **Nml** → **N**

$\mathcal{N}$⟦0⟧=0

$\mathcal{N}$⟦1⟧=1

$\mathcal{N}$⟦N0⟧=2×$\mathcal{N}$⟦N⟧

$\mathcal{N}$⟦N1⟧=2×$\mathcal{N}$⟦N⟧+1

Table 13.1 Semantics of Binary Numerals

input–output identifier. The input of the program is used as the initial value of the identifier, and the value of the identifier after executing the program becomes its output.

The following is an example of a program in this language:

```
program (x);
  begin
    y:=x;
    p:=(procedure x:=x+1);
    if x=y then x:=x+(−1) else x:=0;
    call p
  end.
```

For any numerical input, the output of the program is the same number; however, during execution of the program, the value of x is decreased and then increased. A truth value input produces an error message.

After the abstract syntax come the definitions of the semantic domains. For some of the domains, there is specified a distinctive symbol for the elements. For example,

b∈**B**=**T**+**Z** basic values

specifies that symbol '*b*', possibly with a subscript or a prime, always stands for some (perhaps unspecified) basic value, which may be either a truth value or an integer.

Stores were used in earlier chapters to describe updating assignments. For this language, identifiers may be associated directly with stored values (i.e., locations are not necessary), so that stores may be modelled by functions from identifiers to stored values:

s∈**S**=**Ide** → (**R**+{*unused*}) stores

For any identifier I, *s*⟦I⟧ is the value at I in store *s*. An uninitialized identifier is associated with the special value *unused*, so that in the initial store for a program execution, every identifier (except the one specified for input–output) is associated with *unused*.

Procedures may be modelled by functions from stores to stores (or *error*):

p∈**P**=**S** → **G** procedures

If p is a procedure and s is a store, then $p(s)$ is the result of invoking the procedure with s as the initial store. Note that the domain definitions are (mutually) recursive:

$$\mathbf{R}=\cdots\mathbf{P}\cdots$$
$$\mathbf{S}=\cdots\mathbf{R}\cdots$$
$$\mathbf{P}=\cdots\mathbf{S}\cdots$$

The theoretical consequences of this were discussed in Section 3.3.2.

Three semantic functions are needed, for expressions, commands and programs. An expression yields a value (or *error*) when it is evaluated relative to a store. This is modelled by having expressions denote functions from stores to expression results; i.e., $\mathscr{E}$: **Exp** $\rightarrow$ (**S** $\rightarrow$ **E**), so that if E is an expression, $\mathscr{E}[\![\mathrm{E}]\!]$ is a function from **S** to **E**, and $(\mathscr{E}[\![\mathrm{E}]\!])(s)$ is either *error* or the value of E relative to store s.

To reduce the number of parentheses, it is convenient to assume that function application associates to the left and to omit parentheses around non-syntactic single-symbol arguments to semantic functions. Then $\mathscr{E}[\![\mathrm{E}]\!]s$ is the value of E relative to store s. Also, it is assumed that the domain constructor "$\rightarrow$" associates to the right, so that the domains of arguments and results for $\mathscr{E}$ may be specified as follows:

$$\mathscr{E}:\mathbf{Exp}\rightarrow\mathbf{S}\rightarrow\mathbf{E}$$

and similarly for the other semantic functions.

Executing a command relative to a store yields a command result: either a new store or *error*. Consequently,

$$\mathscr{C}:\mathbf{Com}\rightarrow\mathbf{S}\rightarrow\mathbf{G}$$

where $\mathbf{G}=\mathbf{S}+\{error\}$, and $\mathscr{C}[\![\mathrm{C}]\!]s$ is the result of executing C relative to store s.

Finally, executing a program with some basic value as input yields an answer, either a basic value output or an error message, so that

$$\mathscr{M}:\mathbf{Pro}\rightarrow\mathbf{B}\rightarrow\mathbf{A}$$

where $\mathbf{A}=\mathbf{B}+\{error\}$, and $\mathscr{M}[\![\mathrm{M}]\!]b$ is the answer output by executing M with input b.

The equations for the semantic functions use familiar operations on truth values and integers (*not*, *and*, $-$, $+$, $=$) without explicit definitions. Furthermore, several notational abbreviations are used.

(a) For domain compatibility testing it is often necessary to test the "tag" component of elements of a domain sum, such as $\mathbf{B}=\mathbf{T}+\mathbf{Z}$. Define postfix predicates '$\cdot\,?\mathbf{T}$' and '$\cdot\,?\mathbf{Z}$' on **B** as follows:

$$b?\mathbf{T}=\begin{cases}\textit{true}, \text{ if } b \text{ has been injected into } \mathbf{B} \text{ from } \mathbf{T}\\ \textit{false}, \text{ otherwise,}\end{cases}$$

and similarly for $b?\mathbf{Z}$ and for other domain sums.

(b) A ternary (i.e., three-argument) selection operation '$\cdot\rightarrow\cdot\,,\cdot$' is defined as follows:

$$e\rightarrow x_1, x_2=\begin{cases}x_1, \text{ if } e=\textit{true}\\ x_2, \text{ if } e=\textit{false}\\ \textit{error}, \text{ if } (e?\mathbf{T})=\textit{false}.\end{cases}$$

so that e selects either x_1 or x_2 if it is a truth value, and *error* otherwise.

(c) Strictly speaking, injection and projection functions should be used to map values into and from domain sums:

injections

projections

However, it is usually clearer to omit these mappings from semantic descriptions when they can easily be inferred from context. For example, equation

$$\mathscr{E}[\![-\mathrm{E}]\!]s=e?\mathbf{Z}\rightarrow -e\,,\ \textit{error}$$
$$\text{where } e=\mathscr{E}[\![\mathrm{E}]\!]s$$

specifies that the value of a negation operation relative to store s is the negative of the value of the operand relative to s, provided that this is an integer; otherwise, the evaluation is in error. A sequence of projections of e into **R** and then **B** and then **Z** have been omitted, as well as injections of $-e$ back into **B**, and then **R**, and then **E**.

(d) An assignment command affects a store (i.e., a function from identifiers to stored values) at only one of its arguments. To describe this, we define a ternary operation '$\cdot[\cdot\mapsto\cdot]$' for "perturbing" a function as follows: $s[\mathrm{I}\mapsto r]$ is the function that is like s except that argument I is mapped into r; that is, for all I',

$$s[\mathrm{I} \mapsto r][\![\mathrm{I}']\!] = \begin{cases} r, & \text{if } \mathrm{I}=\mathrm{I}' \\ s[\![\mathrm{I}']\!], & \text{otherwise.} \end{cases}$$

Consequently, equation

$$\mathscr{C}[\![\mathrm{I}:=\mathrm{E}]\!]s = e?\mathbf{R} \to s[\mathrm{I} \mapsto e]\ ,\ error \qquad \text{where } e=\mathscr{E}[\![\mathrm{E}]\!]s$$

specifies that the result of executing I:=E relative to store s is the perturbed store $s[\mathrm{I} \mapsto e]$, where e is the value of E relative to s, provided that this is a storable value; otherwise, the execution is in error.

Note that auxiliary definitions of the form

$$\cdots \text{where } e = \cdots$$

can be used wherever convenient, but "non-mathematical" linguistic devices (updating assignments and sequencers) are *not* used in defining the semantic functions.

For the **while** construction a *recursive* auxiliary definition

$$\cdots \text{where rec, for all } s',\ p(s') = \cdots p \cdots$$

is used. The recurrence of p specifies the effect of the whole loop at that point, but with a different initial store. The semantics of the **while** command can be expressed without circularity by using a limit construction of the kind discussed in Sections 3.3.2, 5.3.4, and 9.2.2:

$$\mathscr{C}[\![\textbf{while E do C}]\!]s = \left[\lim_{i \to \infty} p_i\right](s) \qquad \text{where, for all } s',$$

$$p_{i+1}(s') = \mathscr{E}[\![\mathrm{E}]\!]s' \to (g?\mathbf{S} \to p_i(g)\ ,\ error)\ ,\ s'$$
$$\text{where } g = \mathscr{C}[\![\mathrm{C}]\!]s'.$$

where p_0 is the meaning of commands whose executions never terminate. Note that if C_0 is any never-terminating command (for example, '**while** 0=0 **do null**') and, for all i, C_{i+1} is command

if E **then begin** C; C_i **end else null**

then, for every i, p_i is the meaning of command C_i.

It should now be possible to read through the equations in Table 13.2 and verify that they express formally the semantics of the programming language. Note that the "propagation" approach described at the beginning of Chapter 10 is used to produce program result *error* if any evaluation or execution is in error. For example, equation

$$\mathscr{C}[\![\mathrm{C}_1;\mathrm{C}_2]\!]s = g?\mathbf{S} \to \mathscr{C}[\![\mathrm{C}_2]\!]g\ ,\ error \qquad \text{where } g=\mathscr{C}[\![\mathrm{C}_1]\!]s$$

specifies that the result of executing $\mathrm{C}_1;\mathrm{C}_2$ relative to store s is the result of executing C_2 relative to the result of executing C_1 relative to s, provided that this is not in error; otherwise, the execution of $\mathrm{C}_1;\mathrm{C}_2$ is also in error. A neater treatment of errors will be possible in Section 13.4 using continuations.

Abstract syntax

I∈Ide	identifiers
E∈Exp	expressions
C∈Com	commands
M∈Pro	programs

E ::=0 | 1 | −E | **not** E | E+E | E=E | I | **procedure** C | (E)

C ::=**null** | I:=E | **call** E | C ; C | **if** E **then** C **else** C
| **while** E **do** C | **begin** C **end**

M ::=**program** (I); C.

Semantic Domains

T={*true*,*false*}	truth values
Z={⋯,−2,−1,0,1,2,⋯}	integers
b∈**B**=**T**+**Z**	basic values
r∈**R**=**B**+**P**	storable values
s∈**S**=**Ide** → (**R**+{*unused*})	stores
p∈**P**=**S** → **G**	procedures
e∈**E**=**R**+{*error*}	expression results
g∈**G**=**S**+{*error*}	command results
A=**B**+{*error*}	answers (program results)

Table 13.2 Semantics of a Simple Programming Language

Semantic functions

$$\mathscr{E}: \mathbf{Exp} \to \mathbf{S} \to \mathbf{E}$$
$$\mathscr{C}: \mathbf{Com} \to \mathbf{S} \to \mathbf{G}$$
$$\mathscr{M}: \mathbf{Pro} \to \mathbf{B} \to \mathbf{A}$$

$\mathscr{E}[\![\mathbf{0}]\!]s = 0$

$\mathscr{E}[\![\mathbf{1}]\!]s = 1$

$\mathscr{E}[\![-\mathbf{E}]\!]s = e\,?\,\mathbf{Z} \to -e\ ,\ error$
where $e = \mathscr{E}[\![\mathbf{E}]\!]s$

$\mathscr{E}[\![\mathbf{not\ E}]\!]s = e\,?\,\mathbf{T} \to not(e)\ ,\ error$
where $e = \mathscr{E}[\![\mathbf{E}]\!]s$

$\mathscr{E}[\![\mathbf{E}_1 + \mathbf{E}_2]\!]s = e_1\,?\,\mathbf{Z}\ and\ e_2\,?\,\mathbf{Z} \to e_1 + e_2\ ,\ error$
where $e_i = \mathscr{E}[\![\mathbf{E}_i]\!]s$ for $i = 1,2$

$\mathscr{E}[\![\mathbf{E}_1 = \mathbf{E}_2]\!]s = e_1\,?\,\mathbf{B}\ and\ e_2\,?\,\mathbf{B} \to e_1 = e_2\ ,\ error$
where $e_i = \mathscr{E}[\![\mathbf{E}_i]\!]s$ for $i = 1,2$

$\mathscr{E}[\![\mathbf{I}]\!]s = s[\![\mathbf{I}]\!]\,?\,\mathbf{R} \to s[\![\mathbf{I}]\!]\ ,\ error$

$\mathscr{E}[\![\mathbf{procedure\ C}]\!]s = \mathscr{C}[\![\mathbf{C}]\!]$

$\mathscr{E}[\![(\mathbf{E})]\!]s = \mathscr{E}[\![\mathbf{E}]\!]s$

$\mathscr{C}[\![\mathbf{null}]\!]s = s$

$\mathscr{C}[\![\mathbf{I} := \mathbf{E}]\!]s = e\,?\,\mathbf{R} \to s\{\mathbf{I} \to e\}\ ,\ error$
where $e = \mathscr{E}[\![\mathbf{E}]\!]s$

$\mathscr{C}[\![\mathbf{call\ E}]\!]s = e\,?\,\mathbf{P} \to e(s)\ ,\ error$
where $e = \mathscr{E}[\![\mathbf{E}]\!]s$

Table 13.2 (Continued)

$\mathscr{C}[\![\mathbf{C}_1\ ;\ \mathbf{C}_2]\!]s = g\,?\,\mathbf{S} \to \mathscr{C}[\![\mathbf{C}_2]\!]g\ ,\ error$
where $g = \mathscr{C}[\![\mathbf{C}_1]\!]s$

$\mathscr{C}[\![\mathbf{if\ E\ then\ C}_1\ \mathbf{else\ C}_2]\!]s$
$= \mathscr{E}[\![\mathbf{E}]\!]s \to \mathscr{C}[\![\mathbf{C}_1]\!]s\ ,\ \mathscr{C}[\![\mathbf{C}_2]\!]s$

$\mathscr{C}[\![\mathbf{while\ E\ do\ C}]\!]s = p(s)$
where **rec**, for all s',
$p(s') = \mathscr{E}[\![\mathbf{E}]\!]s' \to (g\,?\,\mathbf{S} \to p(g)\ ,\ error)\ ,\ s'$
where $g = \mathscr{C}[\![\mathbf{C}]\!]s'$

$\mathscr{C}[\![\mathbf{begin\ C\ end}]\!]s = \mathscr{C}[\![\mathbf{C}]\!]s$

$\mathscr{M}[\![\mathbf{program\ (I);\ C.}]\!]b$
$= g\,?\,\mathbf{S}\ and\ g[\![\mathbf{I}]\!]\,?\,\mathbf{B} \to g[\![\mathbf{I}]\!]\ ,\ error$
where $g = \mathscr{C}[\![\mathbf{C}]\!](s\{\mathbf{I} \to b\})$
where, for all $\mathbf{I}'$, $s[\![\mathbf{I}']\!] = unused$

Table 13.2 (Continued)

The treatment of procedures is especially important. The procedure abstract is evaluated by simply applying $\mathscr{C}$ to the body, without supplying a store. This equation might have been written

$$\mathscr{E}[\![\mathbf{procedure\ C}]\!]s = p \qquad \text{where, for all } s',\ p(s') = \mathscr{C}[\![\mathbf{C}]\!]s'.$$

This makes it clear that the store s' for an *invocation* of the procedure need not be the store s of its *definition*. In general, it is possible to "right-cancel" arguments on both sides of a function definition (so long as there are no other occurrences of those arguments). Note that the equation for procedure invocation defines the meaning of that construct solely in terms of the value of its immediate constituent, and without referring to the abstract that defined the procedure.

13.3 ENVIRONMENTS

In this section the programming language of Section 13.2 will be extended to include *definitions*. Suppose that the abstract syntax is augmented to include two forms of definition and a command block, as follows:

D∈**Def** definitions

D ::= **new** I=E | **val** I=E

C ::=··· | **with** D **do** C

The **new** definition is an initialized declaration; the effect of the **val** definition is to bind the identifier to the *r*-value of the expression. The language is to use PASCAL-like (i.e., static) binding and scope conventions. For example, the output of the following program is 0, rather than 1:

```
program (x);
  with val a=0 do
    with val p=procedure x:=a do
      with val a=1 do
        call p.
```

The formal semantics of this language is given in Table 13.3. An abstract syntax and definitions of the domains of basic values are omitted.

In the definitions of the semantic domains, there are two major differences from Table 13.2.

(a) A domain **U** of *environments* has been introduced. An environment is a function that maps identifiers into the values they denote.

(b) Stores now map elements of a domain **L** of *locations* into the values they contain.

Note that the domain of *denotable* values (i.e., denotable by identifiers) differs from the domain of *storable* values (i.e., storable in a single location). Similar differences between value domains exist in almost every programming language.

The semantic functions for expressions, definitions, and commands are defined so that these constructs are interpreted relative to an environment as well as a store. For example,

$$\mathscr{C}[\![C]\!]u$$

is the store transformation denoted by command C in an environment u. Consequently,

$$\mathscr{C}[\![C]\!]u\ s$$

is the result of applying this store transformation to a store s, i.e., the resulting store (or *error*). Interpreting a definition yields a new environment, as well as a command result.

Semantic domains

$l \in \mathbf{L}$	locations
$r \in \mathbf{R}=\mathbf{B}+\mathbf{P}$	storable values
$s \in \mathbf{S}=\mathbf{L} \to (\mathbf{R}+\{unused\})$	stores
$p \in \mathbf{P}=\mathbf{S} \to \mathbf{G}$	procedures
$d \in \mathbf{D}=\mathbf{L}+\mathbf{R}+\{undefined\}$	denotable values
$u \in \mathbf{U}=\mathbf{Ide} \to \mathbf{D}$	environments
$e \in \mathbf{E}=\mathbf{R}+\{error\}$	expression results
$g \in \mathbf{G}=\mathbf{S}+\{error\}$	command results
$\mathbf{A}=\mathbf{B}+\{error\}$	answers

Semantic functions

$\mathscr{E}$: **Exp** → **U** → **S** → **E**
$\mathscr{D}$: **Def** → **U** → **S** → (**U**×**G**)
$\mathscr{C}$: **Com** → **U** → **S** → **G**
$\mathscr{M}$: **Pro** → **B** → **A**

$\mathscr{E}[\![0]\!]u\ s=0$

$\mathscr{E}[\![1]\!]u\ s=1$

$\mathscr{E}[\![\textbf{not E}]\!]u\ s=e\,?\,\mathbf{T} \to not(e)\ ,\ error$
where $e=\mathscr{E}[\![E]\!]u\ s$

$\mathscr{E}[\![E_1+E_2]\!]u\ s=e_1\,?\,\mathbf{Z}\ and\ e_2\,?\,\mathbf{Z} \to e_1+e_2\ ,\ error$
where $e_i=\mathscr{E}[\![E_i]\!]u\ s$ for $i=1,2$

Table 13.3 Semantics With Environments

$\mathscr{E}$[$E_1 = E_2$]$u\ s = e_1?\mathbf{B}$ *and* $e_2?\mathbf{B} \to e_1 = e_2$, *error*
where $e_i = \mathscr{E}[E_i]u\ s$ for $i = 1,2$

$\mathscr{E}$[I]$u\ s = d?\mathbf{L} \to s(d)$,
$d?\mathbf{R} \to d$, *error*
where $d = u$[I]

$\mathscr{E}$[**procedure** C]$u\ s = \mathscr{C}[C]u$

$\mathscr{E}$[(E)]$u\ s = \mathscr{E}[E]u\ s$

$\mathscr{D}$[**new** I = E]$u\ s$
$= e?\mathbf{R}$ *and* there is some $l \in \mathbf{L}$, such that $s(l) = unused$
$\to (u[\mathrm{I} \mapsto l], s[l \mapsto e])$, $(u, error)$
where $e = \mathscr{E}[E]u\ s$

$\mathscr{D}$[**val** I = E]$u\ s = e?\mathbf{R} \to (u[\mathrm{I} \mapsto e], s)$, $(u, error)$
where $e = \mathscr{E}[E]u\ s$

$\mathscr{C}$[**null**]$u\ s = s$

$\mathscr{C}$[I := E]$u\ s = d?\mathbf{L}$ *and* $e?\mathbf{R} \to s[d \mapsto e]$, *error*
where $d = u$[I]
and $e = \mathscr{E}[E]u\ s$

$\mathscr{C}$[**call** E]$u\ s = e?\mathbf{P} \to e(s)$, *error*
where $e = \mathscr{E}[E]u\ s$

$\mathscr{C}$[C_1 ; C_2]$u\ s = g?\mathbf{S} \to \mathscr{C}[C_2]u\ g$, *error*
where $g = \mathscr{C}[C_1]u\ s$

$\mathscr{C}$[**if** E **then** C_1 **else** C_2]$u\ s$
$= \mathscr{E}[E]u\ s \to \mathscr{C}[C_1]u\ s$, $\mathscr{C}[C_2]u\ s$

Table 13.3 (Continued)

The following points should be noted about the semantic equations.

(a) By specifying that *any* location unused in the current store may be allocated, it is possible to avoid consideration of the details of storage management. For example, the equation for **new** I = E specifies that if the execution is not in error, the environment and store that result differ only in that I is bound to some location l that was previously unused, and l now contains the value of E.

$\mathscr{C}$[**while** E **do** C]$u\ s = p(s)$
where **rec**, for all s',
$p(s') = \mathscr{E}[E]u\ s' \to (g?\mathbf{S} \to p(g)$, $error)$, s'
where $g = \mathscr{C}[C]u\ s'$

$\mathscr{C}$[**with** D **do** C]$u\ s = g?\mathbf{S} \to \mathscr{C}[C]u'\ g$, *error*
where $(u', g) = \mathscr{D}[D]u\ s$

$\mathscr{C}$[**begin** C **end**]$u\ s = \mathscr{C}[C]u\ s$

$\mathscr{M}$[**program** (I); C.]b
$= g?\mathbf{S}$ *and* $g(l)?\mathbf{B} \to g(l)$, *error*
where $g = \mathscr{C}[C](u[\mathrm{I} \mapsto l])(s[l \mapsto b])$
where, for all I′, u[I′] = *undefined*
and, for all l', $s(l') = unused$
and l is any location

Table 13.3 (Continued)

(b) For the block, a new environment is created for its body, but for other constructs, the immediate constituents inherit the given environment. In particular, this is true for the procedure abstract, so that free identifiers of a procedure are bound in the context of its definition, rather than its invocations.

(c) Evaluation of an identifier I using $\mathscr{E}$ includes a test of whether it denotes a storable value or a location. If l denotes a location, its current contents are returned; that is, $\mathscr{E}$ determines the *r*-value of its argument.

13.4 CONTINUATIONS

Continuations were used in Chapter 10 to describe the semantics of jumps. It may be recalled from Section 10.1 that the "normal" continuation for a computation is whatever should follow it, as a function of the expected "result" of that computation. We now introduce continuations into our *formal* descriptions.

In the preceding sections, semantic functions returned "local" intermediate results, such as expression values or stores. Let us now define the semantic functions so that they always return the "global" result of the whole program. Each semantic function is defined relative to a *continuation*

argument, in addition to an environment and a store. The continuation specifies what must be done with the intermediate result in order to produce the program answer (provided there is no error or jump).

For example, a command continuation specifies a computation that might follow execution of a command, as a function of the "normal" result of that execution. So a command continuation is a function whose argument is a store and which returns an answer (i.e. a program result):

$$c \in \mathbf{C} = \mathbf{S} \to \mathbf{A} \qquad \text{command continuations}$$

where **S** is the domain of stores and **A** is the domain of answers. Then the semantic function for commands would be

$$\mathscr{C}: \mathbf{Com} \to \mathbf{U} \to \mathbf{C} \to \mathbf{S} \to \mathbf{A}$$

where **U** is the domain of environments. (We shall see later why it is convenient to have the continuation argument precede the store argument.) So $\mathscr{C}[\![\mathrm{C}]\!]u\ c\ s$ is the answer computed by the program of which C is a component. If the execution of C results in a new store s', then the continuation argument c will be applied to s' to produce the program answer.

For example, the semantic equation for the null command is

$$\mathscr{C}[\![\mathbf{null}]\!]u\ c\ s = c(s)$$

This specifies that the answer produced by executing a null command relative to a continuation c is that obtained by applying c to the store argument s. Thus, program execution should continue without any change to the store. Note that the objects that are equated by this equation are answers (program results), and not "intermediate" results, like s.

An example of a semantic equation in which a continuation is defined as well as used is

$$\mathscr{C}[\![\mathrm{C}_1;\mathrm{C}_2]\!]u\ c\ s = \mathscr{C}[\![\mathrm{C}_1]\!]u\ c'\ s$$
$$\text{where, for all } s',\ c'(s') = \mathscr{C}[\![\mathrm{C}_2]\!]u\ c\ s'$$

This specifies that the sequential composition of commands C_1 and C_2 is to be executed as follows: first, C_1 is executed relative to a continuation c' that, if it is applied to some store s', will produce the answer obtained by executing C_2 relative to continuation c, i.e., the continuation of the whole construct. This is just a formal way of specifying the semantics expressed informally in Section 10.1.

Note that it is not necessary to test whether s' is *error* and then propagate the error, as was done in the preceding sections. When an error is discovered, the program answer *error* is selected, and the answer that would be obtained by following the "normal" continuation is ignored. An example of this approach to error handling in the semantics may be seen in the equation for the assignment command:

$$\mathscr{C}[\![\mathrm{I} := \mathrm{E}]\!]u\ c\ s = \cdots d\,?\,\mathbf{L} \to c(s[d \mapsto r])\ ,\ \mathit{error}$$
$$\text{where } d = u[\![\mathrm{I}]\!]$$

If the target identifier denotes a location, the answer is obtained by applying continuation c to an updated store; otherwise, the answer is *error*.

The same principles apply to other syntactic classes. For our language, the domain of expression continuations is

$$k \in \mathbf{K} = \mathbf{R} \to \mathbf{A}$$

Similarly, the domain of definition continuations is

$$q \in \mathbf{Q} = \mathbf{U} \to \mathbf{S} \to \mathbf{A}$$

(We will see later why this domain is more convenient than $(\mathbf{U} \times \mathbf{S}) \to \mathbf{A}$. It may be recalled from Chapter 3 that they are isomorphic.)

For example, equation

$$\mathscr{D}[\![\mathbf{val}\ \mathrm{I} = \mathrm{E}]\!]u\ q\ s = \mathscr{E}[\![\mathrm{E}]\!]u\ k\ s$$
$$\text{where, for all } r,\ k(r) = q(u[\mathrm{I} \mapsto r])(s)$$

specifies that a **val** definition is interpreted by evaluating the expression and then supplying the new environment and the store to the given definition continuation, q.

The language of Section 13.3 is re-defined using continuations in Table 13.4. It is possible to prove that the two definitions are effectively equivalent.

Some of the equations can be simplified by "right-cancelling" store arguments. For example, the equation for sequential composition of commands can be written

$$\mathscr{C}[\![\mathrm{C}_1\ ;\ \mathrm{C}_2]\!]u\ c = \mathscr{C}[\![\mathrm{C}_1]\!]u\ c'$$
$$\text{where } c' = \mathscr{C}[\![\mathrm{C}_2]\!]u\ c$$

Semantic domains

(B, R, S, and U are defined as before)

$A = B + \{error\}$	answers
$c \in C = S \to A$	command continuations
$k \in K = R \to A$	expression continuations
$q \in Q = U \to S \to A$	definition continuations
$p \in P = C \to S \to A$	procedures

Semantic functions

$$\mathscr{E}: \mathbf{Exp} \to \mathbf{U} \to \mathbf{K} \to \mathbf{S} \to \mathbf{A}$$
$$\mathscr{D}: \mathbf{Def} \to \mathbf{U} \to \mathbf{Q} \to \mathbf{S} \to \mathbf{A}$$
$$\mathscr{C}: \mathbf{Com} \to \mathbf{U} \to \mathbf{C} \to \mathbf{S} \to \mathbf{A}$$
$$\mathscr{M}: \mathbf{Pro} \to \mathbf{B} \to \mathbf{A}$$

$\mathscr{E}[\![0]\!]u\ k\ s = k(0)$

$\mathscr{E}[\![1]\!]u\ k\ s = k(1)$

$\mathscr{E}[\![\mathbf{not}\ \mathrm{E}]\!]u\ k\ s = \mathscr{E}[\![\mathrm{E}]\!]u\ k'\ s$
where, for all r, $k'(r) = r?\mathbf{T} \to k(not(r))\ ,\ error$

$\mathscr{E}[\![-\mathrm{E}]\!]u\ k\ s = \mathscr{E}[\![\mathrm{E}]\!]u\ k'\ s$
where, for all r, $k'(r) = r?\mathbf{Z} \to k(-r)\ ,\ error$

$\mathscr{E}[\![\mathrm{E}_1 + \mathrm{E}_2]\!]u\ k\ s$
$= \mathscr{E}[\![\mathrm{E}_1]\!]u\ k_1\ s$
where, for all r_1, $k_1(r_1) = r_1?\mathbf{Z} \to \mathscr{E}[\![\mathrm{E}_2]\!]u\ k_2\ s\ ,\ error$
where, for all r_2, $k_2(r_2) = r_2?\mathbf{Z} \to k(r_1 + r_2)\ ,\ error$

Table 13.4 Semantics With Continuations

in which the store arguments have been made implicit. Similarly, the equation for the block can be simplified to

$\mathscr{C}[\![\mathbf{with}\ \mathrm{D}\ \mathbf{do}\ \mathrm{C}]\!]u\ c = \mathscr{D}[\![\mathrm{D}]\!]u\ q$
where, for all u', $q(u') = \mathscr{C}[\![\mathrm{C}]\!]u'\ c$

$\mathscr{E}[\![\mathrm{E}_1 = \mathrm{E}_2]\!]u\ k\ s$
$= \mathscr{E}[\![\mathrm{E}_1]\!]u\ k_1\ s$
where, for all r_1, $k_1(r_1) = r_1?\mathbf{B} \to \mathscr{E}[\![\mathrm{E}_2]\!]u\ k_2\ s\ ,\ error$
where, for all r_2, $k_2(r_2) = r_2?\mathbf{B} \to k(r_1 = r_2)\ ,\ error$

$\mathscr{E}[\![\mathrm{I}]\!]u\ k\ s = d?\mathbf{L} \to k(s(d))\ ,$
$d?\mathbf{R} \to k(d)\ ,\ error$
where $d = u[\![\mathrm{I}]\!]$

$\mathscr{E}[\![\mathbf{procedure}\ \mathrm{C}]\!]u\ k\ s = k(\mathscr{C}[\![\mathrm{C}]\!]u)$

$\mathscr{E}[\![(\mathrm{E})]\!]u\ k\ s = \mathscr{E}[\![\mathrm{E}]\!]u\ k\ s$

$\mathscr{D}[\![\mathbf{new}\ \mathrm{I} = \mathrm{E}]\!]u\ q\ s = \mathscr{E}[\![\mathrm{E}]\!]u\ k\ s$
where, for all r,
$k(r) =$ there is some $l \in \mathbf{L}$ such that $s(l) = unused$
$\to q(u[\![\mathrm{I} \mapsto l]\!])(s[\![l \mapsto r]\!])\ ,\ error$

$\mathscr{D}[\![\mathbf{val}\ \mathrm{I} = \mathrm{E}]\!]u\ q\ s = \mathscr{E}[\![\mathrm{E}]\!]u\ k\ s$
where, for all r, $k(r) = q(u[\![\mathrm{I} \mapsto r]\!])(s)$

$\mathscr{C}[\![\mathbf{null}]\!]u\ c\ s = c(s)$

$\mathscr{C}[\![\mathrm{I} := \mathrm{E}]\!]u\ c\ s = \mathscr{E}[\![\mathrm{E}]\!]u\ k\ s$
where, for all r, $k(r) = d?\mathbf{L} \to c(s[\![d \mapsto r]\!])\ ,\ error$
where $d = u[\![\mathrm{I}]\!]$

$\mathscr{C}[\![\mathbf{call}\ \mathrm{E}]\!]u\ c\ s = \mathscr{E}[\![\mathrm{E}]\!]u\ k\ s$
where, for all r, $k(r) = r?\mathbf{P} \to r(c)(s)\ ,\ error$

$\mathscr{C}[\![\mathrm{C}_1\ ;\ \mathrm{C}_2]\!]u\ c\ s = \mathscr{C}[\![\mathrm{C}_1]\!]u\ c'\ s$
where, for all s', $c'(s') = \mathscr{C}[\![\mathrm{C}_2]\!]u\ c\ s'$

$\mathscr{C}[\![\mathbf{if}\ \mathrm{E}\ \mathbf{then}\ \mathrm{C}_1\ \mathbf{else}\ \mathrm{C}_2]\!]u\ c\ s$
$= \mathscr{E}[\![\mathrm{E}]\!]u\ k\ s$
where, for all r, $k(r) = r \to \mathscr{C}[\![\mathrm{C}_1]\!]u\ c\ s\ ,\ \mathscr{C}[\![\mathrm{C}_2]\!]u\ c\ s$

Table 13.4 (Continued)

This explains why it is convenient to have store arguments last and to "separate" the arguments of a definition continuation. For a language that allowed side effects, the domain of expression continuations would be $\mathbf{R} \to \mathbf{S} \to \mathbf{A}$.

$\mathscr{C}$[**while** E **do** C]$u\ c\ s = c'(s)$
where **rec**, for all s', $c'(s') = \mathscr{E}$[E]$u\ k\ s'$
where, for all r, $k(r) = r \rightarrow \mathscr{C}$[C]$u\ c'\ s'$, $c(s')$

$\mathscr{C}$[**with** D **do** C]$u\ c\ s = \mathscr{D}$[D]$u\ q\ s$
where $q(u')(s') = \mathscr{C}$[C]$u'\ c\ s'$

$\mathscr{C}$[**begin** C **end**]$u\ c\ s = \mathscr{C}$[C]$u\ c\ s$

$\mathscr{M}$[**program** (I); C.]$b = \mathscr{C}$[C]$(u[\mathrm{I} \mapsto l])\ c\ (s[l \mapsto b])$
where, for all I', u[I']$=$*undefined*
and, for all l', $s(l') =$ *unused*
and, for all s', $c(s') = s'(l)?\mathbf{B} \rightarrow s'(l)$, *error*
where l is any location

Table 13.4 (Continued)

With continuations in the semantic model, it is possible to add sequencers to the language. Suppose that the syntax is augmented by a labelled command and a sequencer, as follows:

S∈**Seq** sequencers

S ::=**goto** I

C ::= ⋯ | I: C | S

The scope of the label in a labelled command I: C will be restricted to this command itself.

For example, the following is a program in the extended language:

```
program (x);
  begin
    x :=0;
loop: if not (x=1) then
      begin
        x :=x+1;
        goto loop
      end
        else null
  end.
```


It will output 1 after looping once.

The semantics of these facilities may be described formally by augmenting domain **D** to include command continuations:

$d \in \mathbf{D} = \mathbf{L} + \mathbf{R} + \mathbf{C} + \{undefined\}$ denotable values

Then,

$\mathscr{C}$[I: C]$u\ c = c'$
where **rec** $c' = \mathscr{C}$[C]$(u[\mathrm{I} \mapsto c'])\ c$

which specifies execution of C relative to an environment in which the label identifier is bound to the continuation that begins with another execution of C. This continuation is defined recursively. Then,

$\mathscr{C}$[S]$u\ c = \mathscr{S}$[S]u

where $\mathscr{S}$: **Seq** → **U** → **S** → **A**, and the equation for the sequencer is

$\mathscr{S}$[**goto** I]$u\ s = d?\mathbf{C} \rightarrow d(s)$, *error*
where $d = u$[I]

so that the normal effect of a **goto** is to follow the continuation denoted by the destination identifier. Note that the semantic function for sequencers does not need a continuation argument.

13.5 CONTEXT-SENSITIVE SYNTAX

In Section 2.6.3, it was pointed out that BNF and similar notations for expressing concrete syntax do not make explicit the syntactic constraints imposed by static scope and type checking. The notation that we have been using to express semantics formally may also be used to specify context-sensitive constraints.

An example of this is given in Table 13.5 for the simple language that we have been using for illustration in this chapter. Because there are no type expressions in this language, the only constraints which can easily be tested for syntactically are that

(a) the target of an assignment must be either the input–output identifier of a program or bound by a **new** declaration,

$t \in \mathbf{Tp}$ types

$t ::= lv \mid rv \mid cv \mid undefined$

$x \in \mathbf{X} = \mathbf{Ide} \to \mathbf{Tp}$ contexts

$\mathbf{c}: \mathbf{Exp} \to \mathbf{X} \to \mathbf{T}$
$\mathbf{d}: \mathbf{Def} \to \mathbf{X} \to (\mathbf{X} + [error])$
$\mathbf{s}: \mathbf{Seq} \to \mathbf{X} \to \mathbf{T}$
$\mathbf{c}: \mathbf{Com} \to \mathbf{X} \to \mathbf{T}$
$\mathbf{m}: \mathbf{Pro} \to \mathbf{T}$

$\mathbf{c}[\![0]\!]x = true$

$\mathbf{c}[\![1]\!]x = true$

$\mathbf{c}[\![\mathbf{not}\ E]\!]x = \mathbf{c}[\![E]\!]x$

$\mathbf{c}[\![-E]\!]x = \mathbf{c}[\![E]\!]x$

$\mathbf{c}[\![E_1 + E_2]\!]x = (\mathbf{c}[\![E_1]\!]x)\ and\ (\mathbf{c}[\![E_2]\!]x)$

$\mathbf{c}[\![E_1 = E_2]\!]x = (\mathbf{c}[\![E_1]\!]x)\ and\ (\mathbf{c}[\![E_2]\!]x)$

$\mathbf{c}[\![I]\!]x = (x[\![I]\!] = lv)\ or\ (x[\![I]\!] = rv)$

$\mathbf{c}[\![\mathbf{procedure}\ C]\!]x = \mathbf{c}[\![C]\!]x$

$\mathbf{c}[\![(E)]\!]x = \mathbf{c}[\![E]\!]x$

Table 13.5 Context-sensitive Syntax

(b) the destination identifier of a **goto** must be a command label, and

(c) an applied occurrence of an identifier must be in the scope of a binding occurrence of that identifier.

The constraints are specified formally by defining functions that check phrase structures relative to a ***context*** (or "static environment", or "symbol table") which maps identifiers into their types:

$x \in \mathbf{X} = \mathbf{Ide} \to \mathbf{Tp}$ contexts

$\mathbf{d}[\![\mathbf{new}\ I = E]\!]x = \mathbf{c}[\![E]\!]x \to x[I \mapsto lv]\ ,\ error$

$\mathbf{d}[\![\mathbf{val}\ I = E]\!]x = \mathbf{c}[\![E]\!]x \to x[I \mapsto rv]\ ,\ error$

$\mathbf{s}[\![\mathbf{goto}\ I]\!]x = (x[\![I]\!] = cv)$

$\mathbf{c}[\![\mathbf{null}]\!]x = true$

$\mathbf{c}[\![I := E]\!]x = (x[\![I]\!] = lv)\ and\ (\mathbf{c}[\![E]\!]x)$

$\mathbf{c}[\![\mathbf{call}\ E]\!]x = \mathbf{c}[\![E]\!]x$

$\mathbf{c}[\![C_1\ ;\ C_2]\!]x = (\mathbf{c}[\![C_1]\!]x)\ and\ (\mathbf{c}[\![C_2]\!]x)$

$\mathbf{c}[\![\mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!]x$
$= (\mathbf{c}[\![E]\!]x)\ and\ (\mathbf{c}[\![C_1]\!]x)\ and\ (\mathbf{c}[\![C_2]\!]x)$

$\mathbf{c}[\![\mathbf{while}\ E\ \mathbf{do}\ C]\!]x = (\mathbf{c}[\![E]\!]x)\ and\ (\mathbf{c}[\![C]\!]x)$

$\mathbf{c}[\![\mathbf{with}\ D\ \mathbf{do}\ C]\!]x = (\mathbf{d}[\![D]\!]x \neq error)\ and\ (\mathbf{c}[\![C]\!](\mathbf{d}[\![D]\!]x))$

$\mathbf{c}[\![\mathbf{begin}\ C\ \mathbf{end}]\!]x = \mathbf{c}[\![C]\!]x$

$\mathbf{c}[\![I{:}\ C]\!]x = \mathbf{c}[\![C]\!](x[I \mapsto cv])$

$\mathbf{c}[\![S]\!]x = \mathbf{s}[\![S]\!]x$

$\mathbf{m}[\![\mathbf{program}\ (I);\ C.]\!] = \mathbf{c}[\![C]\!](x[I \mapsto lv])$
where, for all I', $x[\![I']\!] = undefined$

Table 13.5 (Continued)

For this language, the only types are (a) *lv*, which is associated with the domain **L** of locations, (b) *rv*, which is associated with the domain **R** of storable values, (c) *cv*, which is associated with the domain **C** of command continuations, and (d) *undefined*, which is the type of an unbound identifier. The types for a realistic programming language would be much more complex.

The semantic functions for expressions, sequencers, and commands simply check that, relative to some context, their argument satisfies the

appropriate constraints. The function for definitions returns either a new context, or propagates an error indication. The phrase structure M of a complete program is then syntactically well-formed just if $\mathcal{M}[\![M]\!]=true$.

13.6 SEMANTIC DOMAINS FOR PASCAL

This section describes the semantic domains for PASCAL. This will serve to illustrate how the approach to formal semantic description discussed in the preceding sections may be applied to a realistic programming language.

13.6.1 Basic Values

The basic (or "scalar") values in PASCAL are truth values, characters, integers, enumeration atoms, and real numbers:

$\mathbf{T}=\{false,true\}$	truth values
$\mathbf{H}=\{'A','B',\cdots,'Z','0',\cdots,'9',\cdots\}$	characters
$\mathbf{Z}=\{-maxint,\cdots,-2,-1,0,1,2,\cdots,maxint\}$	integers
At	enumeration atoms
Re	real numbers

The domain of enumeration atoms can be any domain isomorphic to **N**. We shall not attempt to define **Re**. The domain of ***indexing*** values is the sum of all the basic value domains except **Re**:

$$\mathbf{I}=\mathbf{T}+\mathbf{H}+\mathbf{Z}+\mathbf{At} \qquad \text{indexing values}$$

13.6.2 Stores

In this section we shall define the domains **R** of storable values, **S** of stores, **Rv** of *r*-values, and **Lv** of *l*-values.

The values storable in a single location in PASCAL include all of the basic values, plus sets, files, and pointers. A *set* value is conveniently modelled by a function from indexing values to truth values, so that the domain of set values may be defined as follows:

$$\mathbf{St}=\mathbf{I}\to\mathbf{T} \qquad \text{set values}$$

That is, $i\in\mathbf{I}$ is a member of $s\in\mathbf{St}$ just if $s(i)=true$. As discussed in Section 4.5.2, a *file* value may be modelled by two sequences of *r*-values. (The

"buffer" component of a file is selectively updateable and will be discussed later.) In the following, an additional component allows for differentiation between the "read" and "write" states of a file:

$$\mathbf{Fl}=\mathbf{Rv}^*\times\mathbf{Rv}^*\times\{read,write\} \qquad \text{file values}$$

A *pointer* is either an *l*-value or the special value *nil*:

$$\mathbf{Pt}=\mathbf{Lv}+\{nil\} \qquad \text{pointer values}$$

The domain of storable values is then

$$\mathbf{R}=\mathbf{I}+\mathbf{Re}+\mathbf{St}+\mathbf{Fl}+\mathbf{Pt}+\{undefined\} \qquad \text{storable values}$$

where *undefined* is the contents of an uninitialized location. A *store* is a function from locations to storable values (or to *unused* for unallocated locations). Consequently,

$$\mathbf{S}=\mathbf{L}\to(\mathbf{R}+\{unused\}) \qquad \text{stores}$$

where **L** is the domain of locations.

The domain of *r*-values may be defined recursively as the sum of the domains of storable values, records of *r*-values, and arrays of *r*-values:

$$\mathbf{Rv}=\mathbf{R}+(\mathbf{Ide}\to\mathbf{Rv})+(\mathbf{I}\to\mathbf{Rv}) \qquad r\text{-values}$$

Record and array-structured *r*-values may be modelled by functions from identifiers or indexing values to component *r*-values. The domain of *l-values* is similarly constructed, but with locations as the primitive components:

$$\mathbf{Lv}=\mathbf{L}+(\mathbf{Ide}\to\mathbf{Lv})+(\mathbf{I}\to\mathbf{Lv})+(\mathbf{Lv}\times\mathbf{L}) \qquad l\text{-values}$$

The $\mathbf{Lv}\times\mathbf{L}$ term models file *l*-values: the **Lv** component is the file buffer, and the **L** component contains an element of **Fl** (after initialization).

13.6.3 Environments

An *environment* in PASCAL maps identifiers into *l*-values or procedures. (Identifiers may also be bound to types and *r*-values by **type** and **const** definitions, but these bindings are most conveniently represented in the contexts of a context-sentitive syntax.)

Procedures may be modelled by mathematical functions whose results are program answers, and that accept as arguments (one at a time)

(a) a list of the values expressed by the actual parameters of the invocation,

(b) a continuation to be followed after executing the body, and

(c) a store.

Consequently, the procedure domains are

$$\mathbf{P}=\mathbf{E}^* \to \mathbf{C} \to \mathbf{S} \to \mathbf{A} \qquad \text{command procedures}$$
$$\mathbf{F}=\mathbf{E}^* \to \mathbf{K} \to \mathbf{S} \to \mathbf{A} \qquad \text{expression procedures}$$

where the following domains will be discussed later:

E	expressible values
C	command continuations
K	expression continuations
A	answers.

The domain of denotable values is then

$$\mathbf{D}=\mathbf{Lv}+\mathbf{P}+\mathbf{F}+(\mathbf{F}\times\mathbf{L}) \qquad \text{denotable values}$$

A function name denotes an element of $\mathbf{F}\times\mathbf{L}$ in the environment for the body of its definition; the location is for returning a value from the procedure.

The domain of *environments* is then

$$\mathbf{U}=(\mathbf{Ide} \to \mathbf{D})\times(\mathbf{N} \to \mathbf{C}) \qquad \text{environments}$$

The $\mathbf{N} \to \mathbf{C}$ component is for command labels. (It would be more consistent for this to be $\mathbf{Nml} \to \mathbf{C}$, but most PASCAL implementations do not distinguish between, for example, '**goto** 5' and '**goto** 05'. However, '**goto** (2+3)' and

```
const l=5;
    ⋮
goto l;
    ⋮
```

are not allowed.)

The domain of values *expressible* in PASCAL is then defined by

$$\mathbf{E}=\mathbf{Lv}+\mathbf{Rv}+\mathbf{P}+\mathbf{F} \qquad \text{expressible values}$$

13.6.4 Continuations

Continuations for commands, expressions, and definitions are needed in PASCAL. All accept a store and produce a program answer; expression and definition continuations also accept an expressible value or a new environment, respectively:

$\mathbf{A}$	answers
$\mathbf{C}=\mathbf{S} \to \mathbf{A}$	command continuations
$\mathbf{K}=\mathbf{E} \to \mathbf{S} \to \mathbf{A}$	expression continuations
$\mathbf{Q}=\mathbf{U} \to \mathbf{S} \to \mathbf{A}$	definition continuations.

The domain **A** of answers is implementation-dependent.

The entire collection of domain definitions is given in Appendix D. They provide a compact summary of the semantic structure of PASCAL, just as the abstract syntax summarizes its syntactic structure. It is a lengthy but relatively straightforward exercise to define the semantic functions that would complete the formal specification of the semantics of PASCAL.

13.7 DISCUSSION

In this section, we summarize and discuss properties of the method of formal semantic specification that was described and illustrated in the preceding sections.

Firstly, the semantics of a language is a *function* mapping (abstract) syntax to a domain of meanings. This has at least two important consequences:

(a) *Every* syntactic structure is mapped into a ***unique*** meaning, so that issues like incompleteness, inconsistency, and semantic ambiguity simply do not arise. (However, one question that should be addressed is whether the semantic model is ***fully abstract***, that is to say, minimal. For technical reasons that are beyond the scope of this book, domains may contain "unnecessary" elements. This is currently a research problem.)

(b) Semantic functions and meanings are *mathematical* objects, so that standard mathematical techniques may be used to prove results about their properties.

A second important property of the method is that semantic functions are defined so that the meaning of any composite syntactic structure is expressed in terms of the meanings of its immediate constituents. This allows semantic descriptions to be remarkably compact and modular (allowing for the complexity or irregularity of the language being described). Also, we have seen that this characteristic makes it possible to model language features such as procedures and labels "abstractly", that is to say, without introducing artificial distinctions in the meanings of constructs that are syntactically different but semantically indistinguishable in all contexts.

The two properties of semantic descriptions discussed above are relevant to all kinds of language, and indeed originated in the methods developed by mathematicians to express the semantics of logical calculi. A feature that is distinctive to the semantics of *programming* languages is the use of recursive definitions of domains and domain elements. These may be rigorously justified using the theory of approximations, limits, and continuity that was outlined in Section 3.3.2 and which takes into account the limitations of discrete computation.

An important consequence of this theoretical foundation is that *arbitrary* mathematical sets and functions are not necessarily permissible in semantic models for programming languages. A convenient way to ensure that unsuitable domains or functions are not used is to formalize a "safe" meta-language for expressing semantic descriptions, just as it is possible to formalize the syntax and semantics of a syntax description language like BNF.

It has been shown by D. Scott that a very small language called LAMBDA is, in principle, adequate for expressing the semantics of *any* programming language: *all* and *only* the computable functions or functionals (i.e. functions whose arguments or results are functions) are definable in LAMBDA. Consequently, any language describable in LAMBDA must be implementable in principle. Furthermore, LAMBDA is a "mathematical" language in the sense that it has no updating assignments or jumps, and identifier binding is treated in a conventional mathematical way. This property makes it convenient to manipulate semantic specifications in mathematical proofs. The notation that was used in preceding sections for semantic specifications may by regarded as a convenient extension (or "syntactic sugaring") of LAMBDA, and the specifications could be translated into "basic" LAMBDA without much difficulty.

In practice, the four domain constructions and the semantic concepts of stores, environments, and continuations that were discussed are adequate to model all of the features of PASCAL and, indeed, nearly all of the language features discussed in this book. The exceptions are non-determinacy (including concurrency) and **newtype** bindings, which seem to require additional constructions and are subjects of current research. In short, this small set of constructions and concepts provides a simple but general framework for rigorous analysis, comparison, and formal specification of the semantics of a large class of complex and diverse programming languages.

13.8 APPLICATIONS

The primary application for formal semantic descriptions is to allow a language designer to set out a complete and precise specification of a programming language. This can be used by programmers as the final authority on the language and by language implementers as a guide to providing correct implementations. Of course, good informal descriptions are still desirable, particularly if they are based upon the formal description. Informal descriptions are quite adequate for pedagogical purposes and for ordinary use by programmers, but if differences of opinion arise over the interpretation of an informal description, it is essential to have a formal and implementation-independent specification as the ultimate standard.

In the following sections, several other applications of formal semantics will be briefly surveyed.

13.8.1 Soundness of Program Logics

In recent years programming theorists have put considerable emphasis on developing methods for formally *specifying* programs and *verifying* that programs meet their specifications. It is hoped that use of such methods will result in more reliable programs and reduce the need for testing and debugging.

It is possible to apply a semantic function of the sort discussed in preceding sections of this chapter to an individual program and then reason about the resulting meaning of the program. A system known as LCF ("Logic for Computable Functions") has been implemented to verify and partially mechanize the development of such proofs. Another approach is to prove properties of programs in a particular language by using a logical system specifically designed for this purpose. The formal semantics of the language may be used to prove the soundness of the specialized logic.

The best-known method of formal specification for commands is by means of formulas of the form

$$\{A_1\}\ C\ \{A_2\}$$

where C is a command and A_1 and A_2 are ***assertions*** (i.e. truth-valued expressions) about values of identifiers. The intended interpretation of this formula is as follows: whenever the value of A_1 relative to a state is *true*, and executing C relative to that state terminates without error, then the value of A_2 relative to the resulting state is *true*.

For example, here is a specification of a command that assigns the factorial of n ($n!$) to f:

```
{n≥0} begin
        i:=0; f:=1;
        while i≠n do
          begin
            i:=i+1;
            f:=f•i
          end
      end {f=n!}
```

Note that in assertions it is possible to use well-defined notation that is not in the programming language.

It is possible to verify in a formal and systematic way that the command meets its specification by using a logical system of *axioms* and *inference rules*. For example, an axiom scheme for simple assignment commands is

$$\{A'\}\ I:=E\ \{A\}$$

where A' is A with all free occurrences of I substituted by E. (If necessary, identifiers must be changed to prevent clashes, as discussed in Section 8.2.) For example, formula

$$\{f•i=i!\}\ f:=f•i\ \{f=i!\}$$

is an instance of this axiom scheme.

An example of an inference rule is the following one for sequential composition of commands:

$$\frac{\{A_1\}\ C_1\ \{A_2\}\ ,\ \{A_2\}\ C_2\ \{A_3\}}{\{A_1\}\ C_1;C_2\ \{A_3\}}$$

If the formulas above the horizontal line have been proved, then one may deduce the formula below the line.

Here is an instance of the use of this rule:

$$\frac{\{(f•(i+1)=(i+1)!)\ \textbf{and}\ (i+1:0\,..\,n)\}\quad i:=i+1\quad \{(f•i=i!)\ \textbf{and}\ (i:0\,..\,n)\},\quad \{(f•i=i!)\ \textbf{and}\ (i:0\,..\,n)\}\quad f:=f•i\quad \{(f=i!)\ \textbf{and}\ (i:0\,..\,n)\}}{\{(f•(i+1)=(i+1)!)\ \textbf{and}\ (i+1:0\,..\,n)\}\quad i:=i+1;\ f:=f•i\quad \{(f=i!)\ \textbf{and}\ (i:0\,..\,n)\}}$$

The formulas above the line are instances of the axiom scheme for assignment, so that the inference rule allows us to deduce the formula below the line.

A very important inference rule is the following one for **while** loops:

$$\frac{\{\text{E }\textbf{and}\text{ A}\}\ \text{C}\ \{\text{A}\}}{\{\text{A}\}\ \textbf{while}\ \text{E}\ \textbf{do}\ \text{C}\ \{\text{A}\ \textbf{and not}\ \text{E}\}}$$

Assertion A is known as an ***invariant*** of the loop. For example, if E is '$i≠n$', C is

```
begin
  i:=i+1;
  f:=f•i
end
```

and A is '$(f=i!)$ **and** $(i:0\,..\,n)$', the proof of the formula above the line follows from our preceding example and the fact that $(i+1)!=(i+1)•i!$. Thus, the inference rule allows us to deduce the formula below the line. Note that (A **and not** E) implies $f=n!$, so that the rule for sequential composition and the fact that $0!=1$ may then be used to verify the specification of the whole command for computing the factorial of n.

But how do we know whether the axioms are valid and the inference rules sound? To be certain that our formal reasoning is semantically justified, this should be proved once and for all using the semantics of the programming language. To validate an axiom $\{A_1\}\ C\ \{A_2\}$ it must be proved that, for all stores s, if $\mathscr{A}[\![A_1]\!]s=true$ and $\mathscr{C}[\![C]\!]s$? S, then $\mathscr{A}[\![A_2]\!](\mathscr{C}[\![C]\!]s)=true$, where $\mathscr{A}$ is an interpretation function for assertions and $\mathscr{C}$ is the usual semantic function for commands. To verify the soundness of an inference rule, it must be shown that it preserves validity, that is to say, that validity of the formula below the line follows from validity of those above the line.

Unfortunately, the assignment axiom is valid only for rather simple programming languages without side-effects or aliasing. Furthermore, in languages with local scopes, formulas of the form

$$\{A_1\}\ C\ \{A_2\}$$

are almost always inadequate because they do not allow assumptions about free identifiers of the command to be made explicit in the specifications. This is particularly problematical when procedures are used. More elaborate logical systems are currently being developed that attempt to overcome these limitations by using semantic concepts such as environments.

13.8.2 Implementation

Another potentially important area of application for formal semantics is language implementation. Just as formal *syntax* and the theory underlying it have proved to be very useful for systematically writing or mechanically generating parsers for programming languages, it should be possible to use semantic descriptions to help produce interpreters or compilers. There exists a system known as SIS ("Semantics Implementation System") that produces implementations automatically by simply implementing the semantic meta-language, that is to say, by treating semantic specifications as *programs* for a simulated "LAMBDA machine". This approach is simple and general, but is far too inefficient to be practical.

Another direction of research focusses on validating implementation methods rather than producing implementations. It is possible to model implementation methods mathematically. These are known as ***operational*** models. It can then be proved that an operational model correctly implements the semantics of the language, that is to say, produces the same results in all circumstances.

We can illustrate operational models with one for the simple language of Section 13.2. Consider changing the semantic domains for that language as follows:

$$p \in \mathbf{P} = \mathbf{Com} \qquad \text{procedures}$$

A procedure will now be represented by the command part of the abstract that defined it. This models one approach to procedure implementation.

Then, the "semantic" functions for the operational model may be defined exactly as in Table 13.2, except for the following two equations:

$$\mathscr{C}[\![\textbf{procedure C}]\!]s = C$$

$$\mathscr{C}[\![\textbf{call E}]\!]s = e\,?\,\mathbf{P} \rightarrow \mathscr{C}(e)s,\ \mathit{error}$$
$$\text{where } e = \mathscr{E}[\![\mathbf{E}]\!]s$$

Although these appear quite similar to the equations of Table 13.2 they are fundamentally different. For example, suppose that C_1 and C_2 are syntactically distinct commands such that $\mathscr{C}[\![C_1]\!] = \mathscr{C}[\![C_2]\!]$. But then according to this operational model,

$$\mathscr{E}[\![\textbf{procedure } C_1]\!] \neq \mathscr{E}[\![\textbf{procedure } C_2]\!]$$

even though the procedures are semantically indistinguishable.

The crucial difference between the two descriptions is that in the operational model, the meaning of the procedure invocation is expressed in terms of the meaning of a command (called '*e*' in the equation) that is *not* one of its immediate constituents. Nonetheless, it can be proved that the operational model correctly implements the semantics of Table 13.2. More intricate implementation methods (such as the use of a stack for storage management and a display for environment management) have been validated.

Perhaps the most promising approach to semantics-directed language implementation is to develop general methods for systematically *transforming* semantic specifications into operational models that are directly implementable. The example discussed above is actually a simple case of a general transformation technique that allows a domain of *functions* (such as $S \rightarrow S$) to be represented by a domain of non-functional *data structures* (such as **Com**). Moreover, this and other transformations may be formalized and proved to be generally valid. This approach has been used to produce usable implementations of non-trivial programming languages in a fairly systematic way, but much work must be done before it can be used in practice.

13.8.3 Design

We began this book by observing that the problem of designing programming languages suited to both humans and computers is one of the most challenging of those faced by the computing scientist. Formal semantics can assist a language designer by allowing rigorous description of the semantics of his proposed language or language features. This makes evident any ambiguity, irregularity, or unnecessary complexity in the design, permits formal study of its properties, and facilitates comparison with other possible design approaches.

Of course, the role that formal semantics can play in language design must not be exaggerated. It is much like the role played by a programming language in program development. A language does not suggest what problems are worthwhile to solve, nor provide anything more than the framework for developing solutions. But a good language can protect a

programmer from his own mistakes and help him to structure a solution in such a way as to help him cope with its complexity. Similarly, formal semantics is a tool which can help language designers achieve their objectives.

If this book has convinced the reader that a programming language designer needs the expertise of a scientist, the precision of a mathematician, and the taste of an artist as well as the pragmatism of an engineer, then it has achieved one of its objectives.

EXERCISES

13.1 Describe formally the semantics of binary numerals with fractions.

13.2 Use the semantics of Table 13.2 to verify the claims made in the text about the program example in Section 13.2.

13.3 Suppose that a multiple assignment command

$$I_1,I_2 := E_1,E_2$$

for $I_1 \neq I_2$ were added to the language of Section 13.2. Describe formally its (usual) semantics.

13.4 Suppose that an iterative command

repeat C **until** E

were added to the language of Section 13.2. Describe formally its (usual) semantics.

13.5 Commands C_1 and C_2 are equivalent just if $\mathscr{C}[\![C_1]\!] = \mathscr{C}[\![C_2]\!]$. Which of the following pairs of commands are equivalent according to the semantics of Section 13.2?

(a) I := I and **null**
(b) **null** ; C and C
(c) **call procedure** C and C
(d) **while** E **do** C and **if** E **then begin** C; **while** E **do** C **end else null**

13.6 Use the semantics of Table 13.3 to verify the claims made about the example program in Section 13.3.

13.7 Suppose that a definition form

var $I_1 = I_2$

were added to the language of Section 13.3. I_2 must denote a location and the effect of the definition is to bind I_1 to this location. Describe formally the semantics of this facility.

13.8 Suppose that parameters were added to the language of Section 13.3 as follows:

P∈**Par** parameters

P ::= **val** I | **new** I

E ::= ··· | **procedure** (P); C | ···

C ::= ··· | **call** $E_1(E_2)$ | ···

The two parameter forms are to correspond semantically to the syntactically similar definition forms. Describe formally the semantics of these facilities, using the domain

$p \in \mathbf{P} = \mathbf{R} \to \mathbf{S} \to \mathbf{G}$ procedures

and an additional semantic function

$\mathscr{P} : \mathbf{Par} \to \mathbf{U} \to \mathbf{R} \to \mathbf{S} \to (\mathbf{U} \times \mathbf{G})$

What difficulty arises if the same approach is used for **var** parameters that are to correspond to **var** definitions as discussed in Exercise 13.7?

13.9 Modify the semantics of Table 13.3 to specify that free identifiers of procedures are to be bound *dynamically*.

13.10 Use the semantics of Table 13.4 and the equations for **goto** and labels to verify the claims made about the example program in Section 13.4.

13.11 Add sequencer **stop** to the language of Section 13.4 and define formally its (usual) semantics.

13.12 Replace sequencer **goto** I in the language of Section 13.4 by sequencer **leave** I, whose effect is to exit the command labelled by I. Describe formally the semantics of this facility, making whatever changes are necessary to the equations of other constructs.

13.13 Add expression form

begin C **result** E

to the language of Section 13.4 and define formally its (usual) semantics, making whatever changes are necessary to the equations of other constructs.

13.14 Which of the domain compatibility tests in the semantic specification of Table 13.4 would be redundant for programs that satisfy the constraints of the syntax in Table 13.5?

13.15 Add type expressions to the language of Section 13.4 and specify a context-sensitive syntax that would make it possible to reduce the number of domain compatibility tests necessary during program execution.

PROJECT

Devise formal semantic models for

a. selective *l*-expressions, such as L.1 and L[E];
b. the iterative control structure in ALGOL 68;
c. name parameters in ALGOL 60;
d. composite definition structures;
e. definition and invocation of classes;
f. coroutines;
g. backtracking.

BIBLIOGRAPHIC NOTES

More on the subject of formal semantics and its applications may be found in papers by Strachey[13.28], Scott and Strachey [13.26], Scott [13.24, 13.25], Strachey and Wadsworth [13.29], and Tennent [13.30], and in books by Gordon [13.7], Stoy [13.27], and Milne and Strachey [13.16]. The problem of full abstraction of semantic models of programming languages was first discussed in a paper by Milner [13.17]. Formal semantics of non-determinism and concurrency is discussed in papers by Milne and Milner [13.15], Schwarz [13.23], Francez et al. [13.5], Hennessey and Plotkin [13.9] Back [13.2], and Park [13.19]; semantics of **newtype** bindings is discussed in a paper by Reynolds [13.21] and a thesis by McCracken [13.14].

LCF has been described by Gordon et al. [13.8]. The logic of formulas of the form $\{A_1\}\ C\ \{A_2\}$ was first described by Hoare [13.10]; Apt [13.1] has surveyed this area. Validation of axioms and inference rules using mathematical semantics was first demonstrated by Ligler [13.13] and Donahue [13.4]. More powerful program logics are described in books by Reynolds [13.22] and de Bakker [13.3].

A description of SIS may be found in a paper by Mosses [13.18]. Verification of operational models has been treated by Milne and Strachey [13.16] and Gordon [13.6]. Transformations of language descriptions have been discussed by Reynolds [13.20]. Some current research on semantics-directed implementation may be found in a conference proceedings [13.11]. Tennent [13.30, 13.32] and Ligler [13.12] have discussed applications of denotational semantics in language design.

13.1 Apt, K. R. "Ten years of Hoare's logic, a survey", *Proc. 5th Scandinavian Logic Symposium* (eds., F. V. Jensen, B. H. Mayoh and K. K. Møller), pp. 1–44, Aalborg University Press (1979).

13.2 Back, R. J. "Semantics of unbounded non-determinism", Automata, Languages and Programming, *Lecture Notes in Computer Science*, **85**, Springer, Berlin (1980).

13.3 de Bakker, J. W. *Mathematical Theory of Program Correctness*, Prentice-Hall International, London (1980).

13.4 Donahue, J. E. "The mathematical semantics of axiomatically defined programming language constructs", *Proc. Int. Symposium on Proving and Improving Programs*, Arc-et-Senans, pp. 353–67, IRIA, Rocquencourt, France (1975).

13.5 Francez, N., C.A.R. Hoare, D. J. Lehmann, and W. P. deRoever, "Semantics of non-determinism, concurrency, and communication", *J. Comp. and Sys. Sci.*, **19**(3), 290–308 (1979).

13.6 Gordon, M. "Operational reasoning and denotational semantics", *Proc. Int. Symposium on Proving and Improving Programs*, Arc-et-Senans, pp. 83-98, IRIA, Rocquencourt, France (1975); also technical report CS-906, Computer Science Dept., Stanford University, Stanford, California (1975).

13.7 Gordon, M. *The Denotational Description of Programming Languages: An Introduction*, Springer, New York (1979).

13.8 Gordon, M., R. Milner, and C. Wadsworth. "Edinburgh LCF", *Lecture Notes in Computer Science*, **78**, Springer, Berlin (1979).

13.9 Hennessy, M.C.B., and G. D. Plotkin. "Full abstraction for a simple parallel programming language", Proc. Symposium on Mathematical Foundations of Computer Science, pp. 108–20, *Lecture Notes in Computer Science*, **74**, Springer, Berlin (1979).

13.10 Hoare, C.A.R. "An axiomatic basis for computer programming", *Comm. ACM*, **12**(10), 576–80, 583 (1969).

13.11 Jones, N. (ed.) Proc. of the Aarhus Workshop on Semantics-directed Compiler Generation, *Lecture Notes in Computer Science*, **94**, Springer, Berlin (1980).

13.12 Ligler, G. T. "A mathematical approach to language design, *Conf. Record of the 2nd ACM Symposium on Principles of Programming Languages*, pp. 41–53, ACM, New York (1975).

13.13 Ligler, G. T. "Surface properties of programming language constructs", *Proc. Int. Symposium on Proving and Improving Programs*, Arc-et-Senans, pp. 299–323, IRIA, Rocquencourt, France (1975).

13.14 McCracken, N. J. *An Investigation of a Programming Language with a Polymorphic Type Structure*, Ph.D. thesis, School of Computer and Information Science, Syracuse University (1979).

13.15 Milne, G., and R. Milner. "Concurrent processes and their syntax", *J. ACM*, **26**(2), 302–21 (1979).

13.16 Milne, R. E. and C. Strachey. *A Theory of Programming Language Semantics* (2 volumes), Chapman and Hall, London, and Wiley, New York (1976).

13.17 Milner, R. "Processes: a mathematical model of computing agents," *Logic Colloquium '73*, pp. 157–74, North-Holland, Amsterdam (1975).

13.18 Mosses, P. D. "Compiler generation using denotational semantics", Proc. Symposium on Mathematical Foundations of Computer Science, Gdansk, *Lecture Notes in Computer Science*, **45**, pp. 436–41, Springer, Berlin (1976).

13.19 Park, D. "On the semantics of fair parallelism", in *Abstract software specifications*, pp. 504–26, *Lecture Notes in Computer Science*, **86**, Springer, Berlin (1980).

13.20 Reynolds, J. C. "Definitional interpreters for higher order programming languages", *Proc. 25th ACM National Conf.*, pp. 717–40, ACM, New York (1972).

13.21 Reynolds, J. C. "Towards a theory of type structure", Proc. Colloque sur la Programmation, pp. 408–23, *Lecture Notes in Computer Science*, **19**, Springer, Berlin (1974).

13.22 Reynolds, J. C. *The Craft of Programming*, Prentice-Hall International, London (1981).

13.23 Schwarz, J. S. "Denotational semantics of parallelism", *Semantics of Concurrent Computation*, Proc. of the Int. Symposium, Evian, France, pp. 191–202, *Lecture Notes in Computer Science*, **70**, Springer, Berlin (1979).

13.24 Scott, D. S. "Mathematical concepts in programming language semantics", *Proc. 1972 Spring Joint Computer Conference*, pp. 225–34, AFIPS Press, Montvale, N.J. (1972).

13.25 Scott, D. S. "Data types as lattices", *SIAM J. on Computing*, **5**(3), 522–86 (1976).

13.26 Scott, D. S. and C. Strachey. "Towards a mathematical semantics for computer languages", *Proc. of the Symposium on Computers and Automata* (ed., J. Fox), pp. 19–46, Polytechnic Institute of Brooklyn Press, New York (1971); also technical monograph PRG–6, Programming Research Group, University of Oxford (1971).

13.27 Stoy, J. E. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass. (1977).

13.28 Strachey, C. "The varieties of programming language", *Proc. Int. Computing Symposium*, pp. 222–33, Cini Foundation, Venice (1972); also technical monograph PRG-10, Programming Research Group, University of Oxford (1973).

13.29 Strachey, C. and C. P. Wadsworth. *Continuations, a Mathematical Semantics for Handling Full Jumps*, technical monograph PRG-11, Programming Research Group, University of Oxford (1974).

13.30 Tennent, R. D. *Mathematical Semantics and Design of Programming Languages*, Ph.D thesis and technical report 59, Dept. of Computer Science, University of Toronto, Ontario (1973).

13.31 Tennent, R. D. "The denotational semantics of programming languages", *Comm. ACM*, **19**(8), 437–53 (1976).

13.32 Tennent, R. D. "Language design methods based on semantic principles", *Acta Informatica*, **8**, 97–112 (1977).

APPENDIX A
BIBLIOGRAPHY ON PROGRAMMING LANGUAGES

ADA

1 "Preliminary ADA Reference Manual", *SIGPLAN Notices*, **14** (6), part A (1979).

2 Ichbiah, J. D. et al. "Rationale for the design of the ADA programming language", *SIGPLAN Notices*, **14** (6), part B (1979).

ALGOL 60

1 Naur, P. (ed.): "Revised report on the algorithmic language ALGOL 60"; *Comm. ACM*, **6** (1), 1–20 (1963); also *Comp. J.*, **5**, 349–67 (1963), and *Numerische Mathematik*, **4**, 420–52 (1963).

2 De Morgan, R. M., I. D. Hill and B. A. Wichmann. "A supplement to the ALGOL 60 Revised Report", *Comp. J.*, **19**, 276–88 (1976); erratum: *Comp. J.*, **21**, 282 (1978).

3 Eckman, T. and C. E. Froberg. *Introduction to Algol Programming*, Studentlitteratur, Lund, Sweden (1965); distributed by Petrocelli, New York.

4 Knuth, D. E. "The remaining trouble spots in ALGOL 60", *Comm. ACM*, **10** (10), 611–18 (1967).

5 Wichmann, B. A. *Algol 60 Compilation and Assessment*, Academic Press, London (1973).

ALGOL W

1 Wirth, N. and C. A. R. Hoare. "A contribution to the development of ALGOL", *Comm. ACM*, **9** (6), 413–32 (1966).

2 Sites, R. L. *Algol W Reference Manual*, technical report CS-230, Computer Science Dept., Stanford University, Stanford, California (1972).

3 Kieburtz, R. B. *Structured Programming and Problem Solving with ALGOL W*, Prentice-Hall, Englewood Cliffs, N.J. (1975).

ALGOL 68

1 Lindsey, C. H. and S. G. van der Meulen. *An Informal Introduction to ALGOL 68* (revised edition), North Holland, Amsterdam (1977).

2 McGettrick, A. D. *ALGOL 68: A First and Second Course*, Cambridge University Press, Cambridge, England (1978).

3 Hoare, C. A. R. "Critique of ALGOL 68", *ALGOL Bulletin*, No. 29, pp. 27–9 (1968).

4 Sintzoff, M. "A brief review of ALGOL 68", *ALGOL Bulletin*, No. 37, pp. 54–6 (1974).

APL

1 Wiedmann, C. *Handbook of APL Programming*, Mason and Lipscomb, London, and Petrocelli, New York (1974).

2 Falkoff, A. D. and K. E. Iverson. "The design of APL", *IBM J. Research and Development*, **17** (4), 324–34 (1973).

3 Abrams, P. S. "What's wrong with APL?", *Proc. APL 75 Congress*, pp. 1–8, ACM, New York (1975).

BCPL

1 Richards, M. "BCPL: a tool for compiler writing and system programming", *Proc. AFIPS Spring Joint Computer Conf.*, vol. 34, pp. 557–66 (1969).

COBOL

1 *American National Standard COBOL*, ANS X3.23-1974, American National Standards Institute, New York (1974).

2 Jackson, M. A. "COBOL", in *Software Engineering* (ed., R. H. Perrott), pp. 47–67, Academic Press, London (1977).

CONCURRENT PASCAL

1 Brinch Hansen, P. *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, N.J. (1977).

FORTRAN

1 *American National Standard Programming Language FORTRAN*, ANS X3.9-1978, American National Standards Institute, New York (1978).

2 Wagener, J. L. *FORTRAN 77 Principles of Programming*, Wiley, New York (1980).

3 Brainerd, W. (ed.) "Fortran 77", *Comm. ACM*, **21** (10), 806–20 (1978).

LISP

1 McCarthy, J. "Recursive functions of symbolic expressions and their computation by machine, part 1", *Comm. ACM*, **3** (4), 184–95 (1960).

2 Siklossy, L. *Let's Talk LISP*, Prentice-Hall, Englewood Cliffs, N.J. (1976).

3 Allen, J. *Anatomy of LISP*, McGraw-Hill, New York (1978).

MODULA

1 Wirth, N. "Modula: a language for modular multiprogramming", *Software Practice and Experience*, **7** (1), 3–35 (1977).

2 Wirth, N. *MODULA-2*, Berichte des Instituts für Informatik No. 36, ETH, Zurich (1980).

PASCAL

1 Jensen, K., and N. Wirth. *PASCAL User Manual and Report*, Springer, New York and Berlin (2nd ed., 1974).

2 Welsh, J. and J. Elder. *Introduction to PASCAL*, Prentice-Hall International, London (1979).

3 Habermann, A. N. "Critical comments on the programming language PASCAL", *Acta Informatica*, **3**, 47–57 (1973).

4 Lecarme, O. and P. Desjardins. "More comments on the programming language PASCAL", *Acta Informatica*, **4**, 231–43 (1975).

5 Wirth, N. "An assessment of the programming language PASCAL", *IEEE Trans. on Software Engineering*, **1**, 192–8 (1975).

6 Welsh, J., W. J. Sneeringer and C. A. R. Hoare. "Ambiguities and insecurities in PASCAL", *Software Practice and Experience*, **7**, 685–96 (1977).

7 Addyman, A. M. "A draft proposal for PASCAL", *SIGPLAN Notices*, **15** (4), 1–66 (1980).

PASCAL PLUS

1 Welsh, J. "Pascal Plus, another language for modular multiprogramming", *Software Practice and Experience*, **9**, 947–57 (1979).

2 Welsh, J. and M. McKeag. *Structured System Programming*, Prentice-Hall International, London (1980).

PL/I

1 *American National Standard Programming Language PL/I*, ANS X3.53-1976, American National Standards Institute, New York (1976).

2 Nicholls, J. E. "Conflicting issues in programming language design", in *Software Engineering* (ed., R. H. Perrott), pp. 93–104, Academic Press, London (1977).

SIMULA

1 Dahl, O.-J., B. Myhraug and K. Nygaard. *SIMULA 67 Common Base Language*, S-22, Norwegian Computer Center, Oslo (1970).

2 Birtwistle, G. M., O.-J. Dahl, B. Myhraug and K. Nygaard. *SIMULA begin*; Auerbach, Philadelphia (1973), and Studentlitteratur, Lund, Sweden (1974).

3 Dahl, O.-J. and C. A. R. Hoare. "Hierarchical program structures", in *Structured Programming* (O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare), pp. 175–220, Academic Press, London (1972).

SNOBOL 4

1 Griswold, R. E., J. F. Poage and I. P. Polonsky. *The SNOBOL 4 Programming Language* (second edition), Prentice-Hall, Englewood Cliffs, N.J. (1971).

APPENDIX B
ABSTRACT SYNTAX FOR PASCAL

N	numerals
B	literals
O	operators
I	identifiers
L	*l*-expressions
E	expressions
K	static expressions
T	type expressions
Q	parameter specifiers
P	formal parameters
D	definitions
S	sequencers
C	commands
M	programs

L ::=I | L.I | L[E] | E↑

E ::= B | I | OE | EOE | I(···,E,···) | L.I | L[E] | E↑
| [···,E,···,E. . E,···] | (E)

K ::=B | I | OK

T ::=I | (···,I,···) | K. . K | ↑I | **set of** T
| **array**[T]**of** T | **file of** T | **record**···;I:T;···**end**
| **record**···;I:T;···**case** I:I **of**···;K:(···;I:T;···);···**end**

Q ::=I:I | **var** I:I
| **procedure** I(···;Q;···) | **function** I(···;Q;···):I

P ::=I:I | **var** I:I
| **procedure** I(···;Q;···) | **function** I(···;Q;···):I

D ::= **const** I=K; | **type** I=T; | **var** I:T;
| **procedure** I(···;P;···);C; | **function** I(···;P;···):I;C;

S ::= **goto** N

C ::= | L:=E | I | I(···,E,···) | C;C
| **if** E **then** C | **if** E **then** C **else** C
| **case** E **of**···;K:C;···**end**
| **while** E **do** C | **repeat** C **until** E
| **for** I:=E **to** E **do** C | **for** I:=E **downto** E **do** C
| N:C | S
| **with** L **do** C | ···D···**begin** C **end** | **begin** C **end**

M ::= **program** I(···,I,···);C.

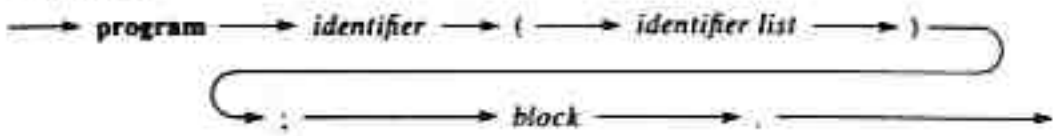
Notes

(a) Several "abbreviations" (such as multidimensional arrays) have been omitted.

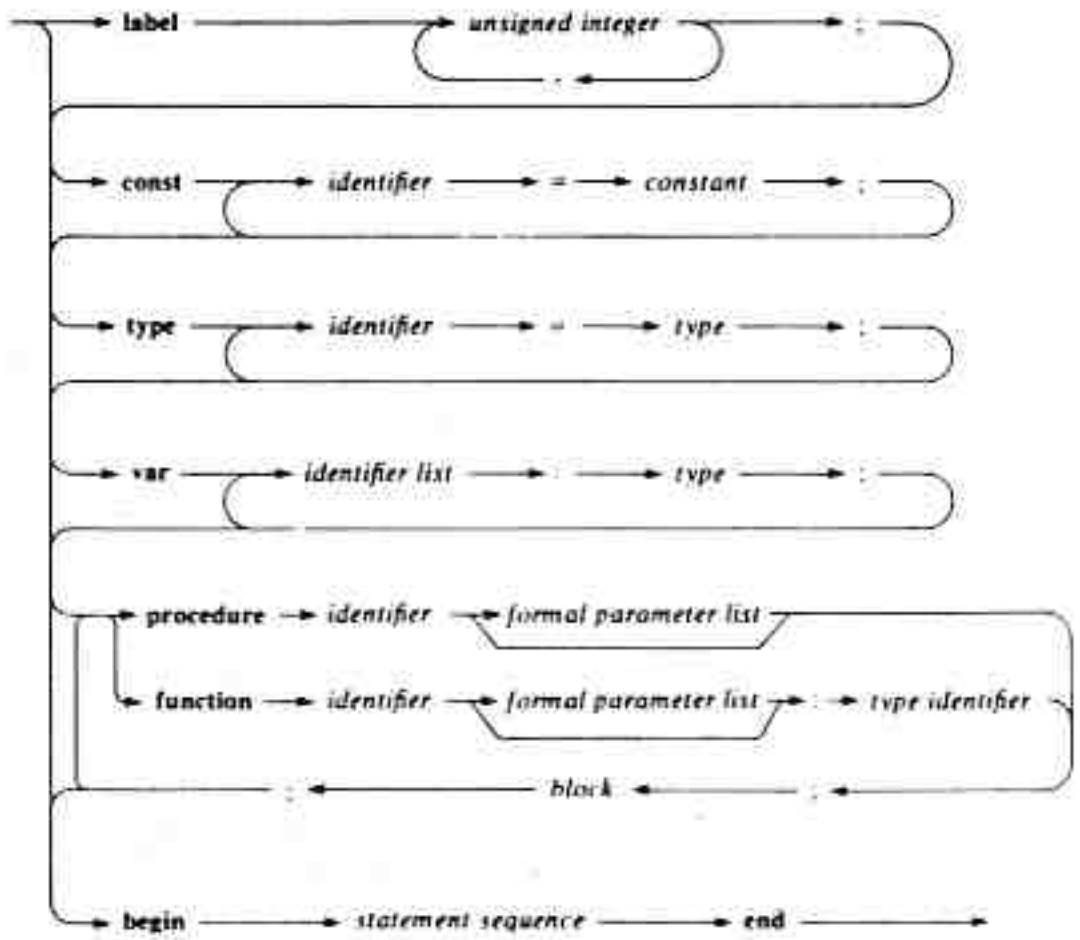
(b) **label** and **forward** declarations and **packed** types have been omitted.

APPENDIX C
SYNTAX DIAGRAMS FOR PASCAL

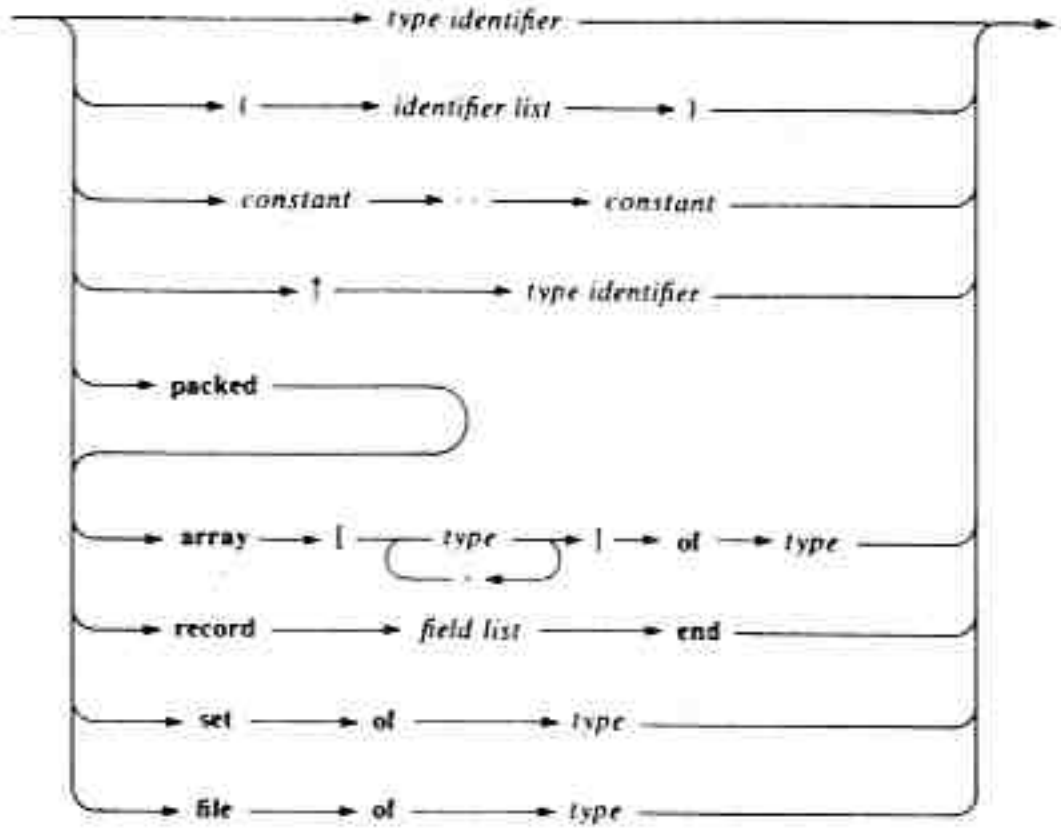
PROGRAM



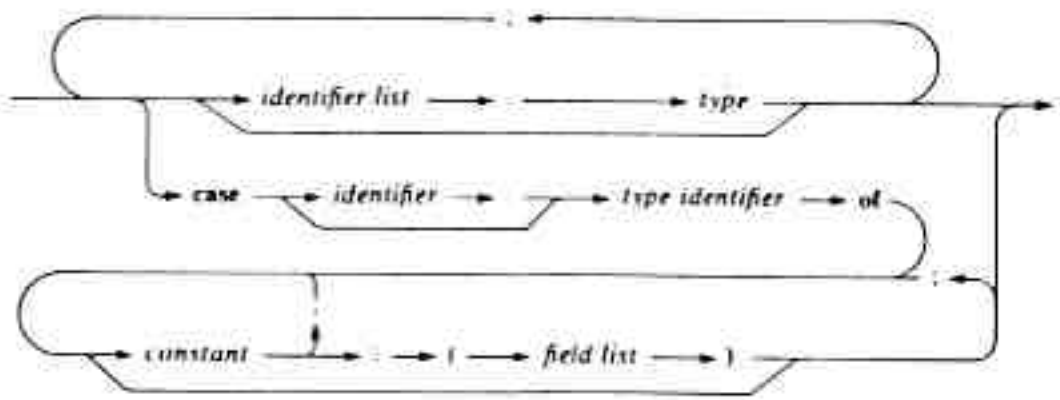
BLOCK



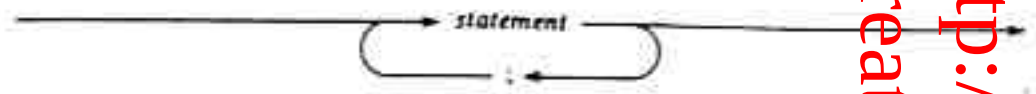
TYPE



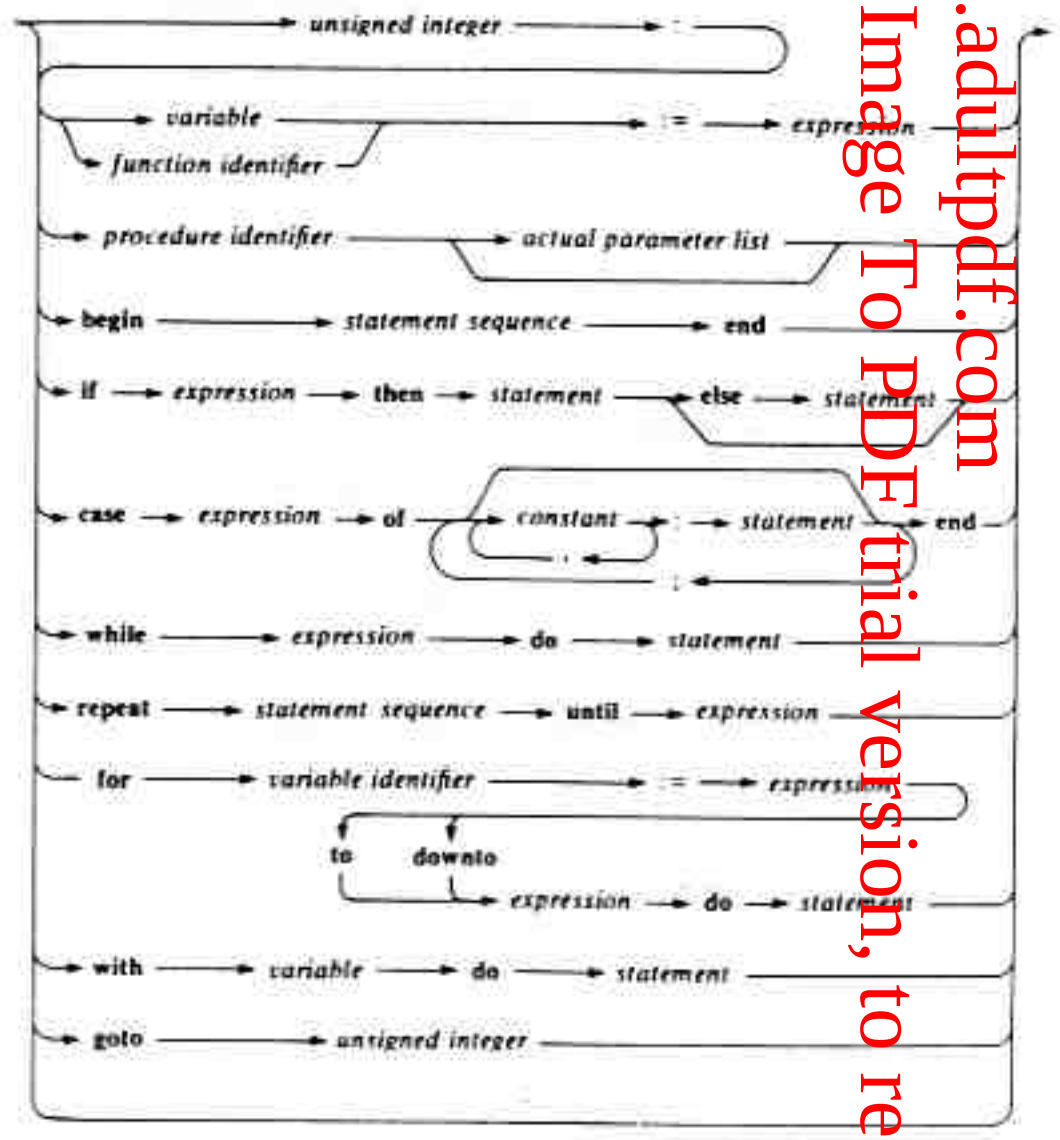
FIELD LIST



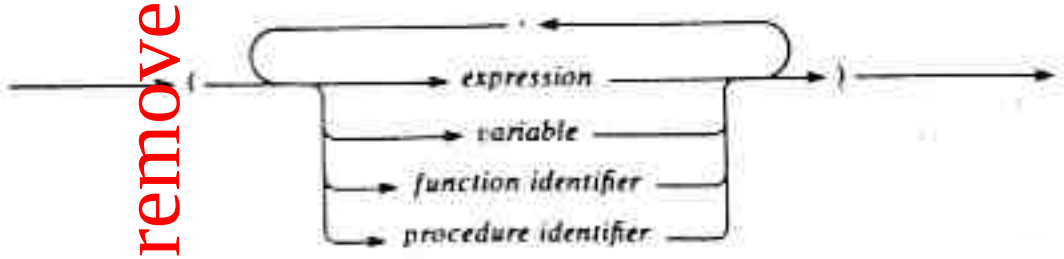
STATEMENT SEQUENCE



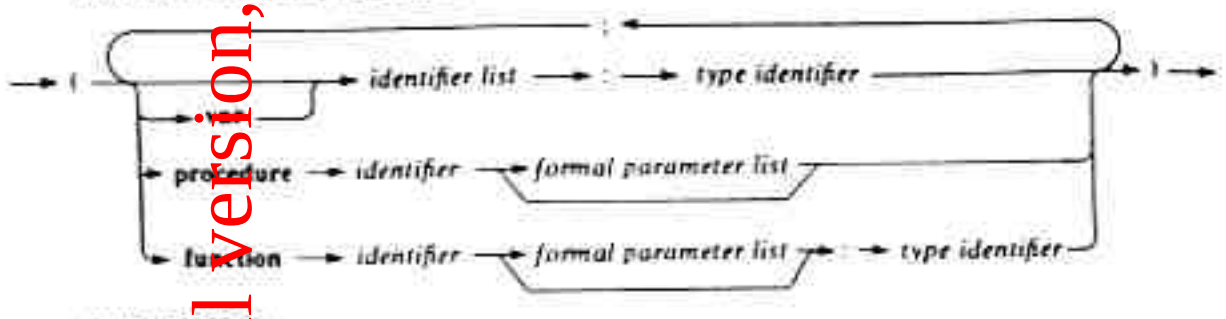
STATEMENT



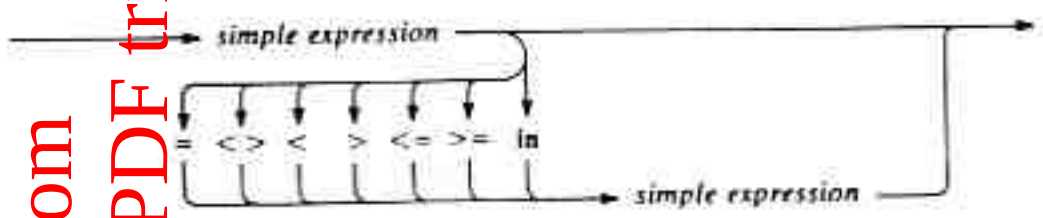
ACTUAL PARAMETER LIST



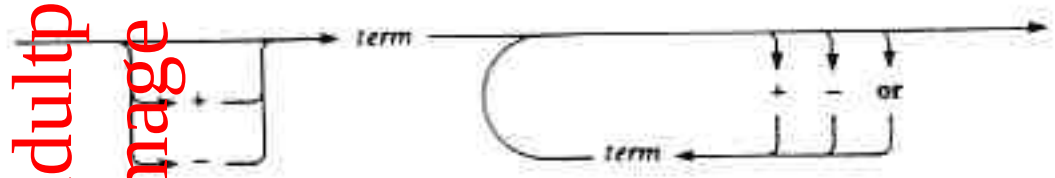
FORMAL PARAMETER LIST



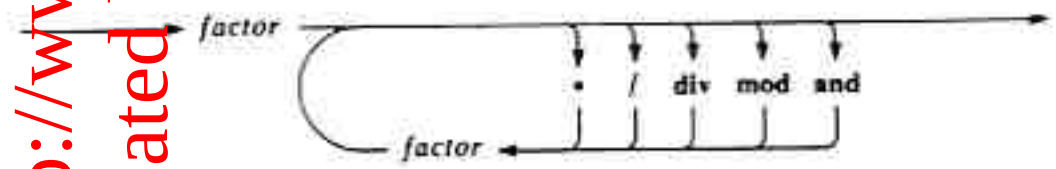
EXPRESSION



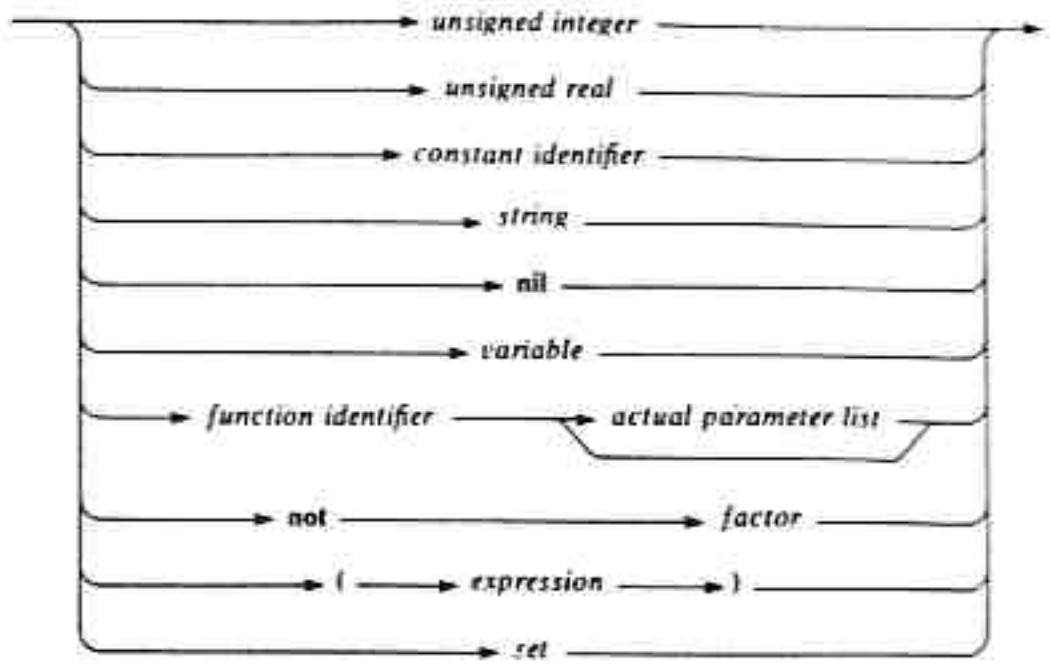
SIMPLE EXPRESSION



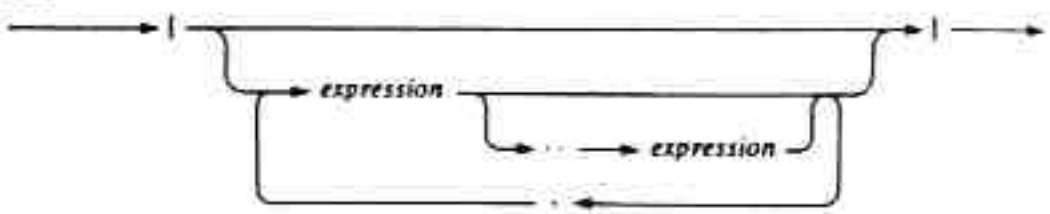
TERM



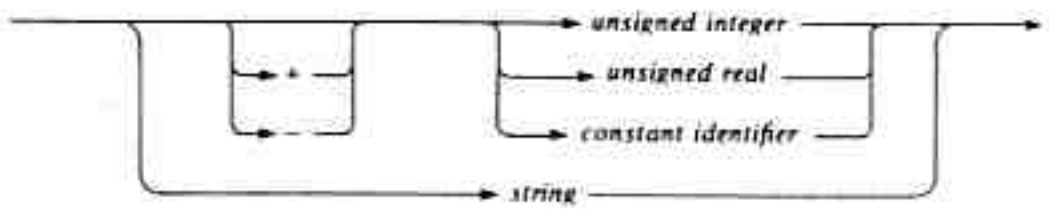
FACTOR



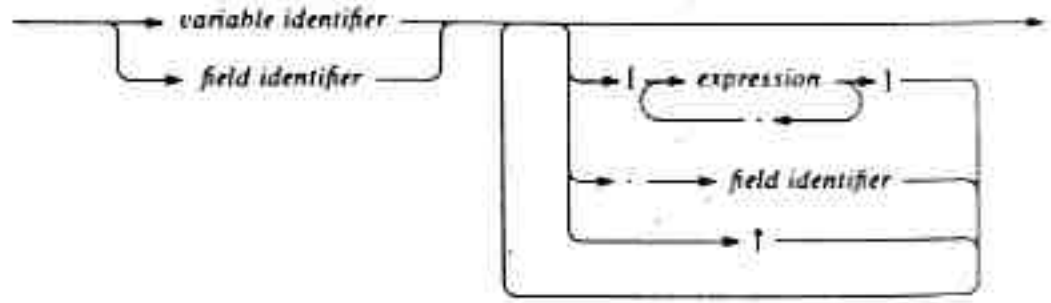
SET



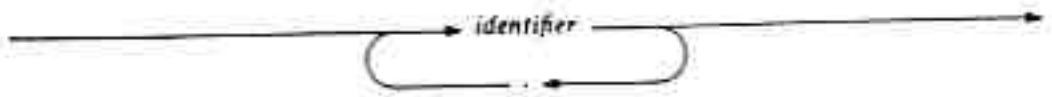
CONSTANT



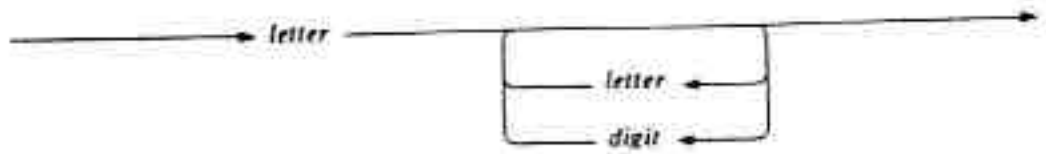
VARIABLE



IDENTIFIER LIST

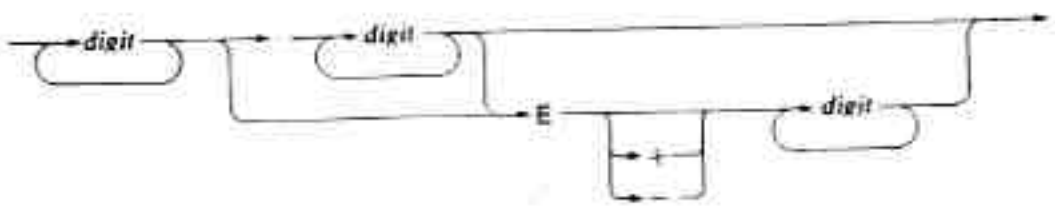


IDENTIFIER

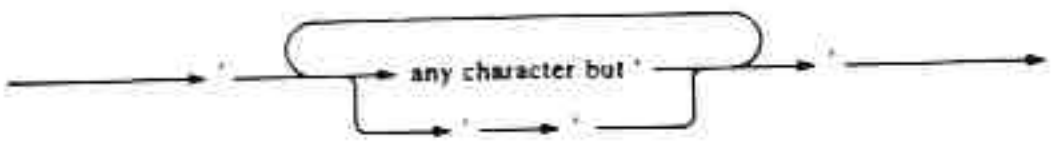


UNSIGNED INTEGER

UNSIGNED REAL



STRING



APPENDIX D
SEMANTIC DOMAINS FOR PASCAL

BASIC VALUES

T={*false*,*true*}	truth values
H={'A','B',⋯,'Z','0',⋯,'9',⋯}	characters
Z={−*maxint*,⋯,−2,−1,0,1,2,⋯,*maxint*}	integers
At	enumeration atoms
I=**T**+**H**+**Z**+**At**	indexing values
Re	real numbers

STORES

St=**I** → **T**	set values
Fl=**Rv***×**Rv***×{*read*, *write*}	file values
Pt=**Lv**+{*nil*}	pointer values
R=**I**+**Re**+**St**+**Fl**+**Pt**+{*undefined*}	storable values
L	locations
S=**L** → (**R**+{*unused*})	stores
Rv=**R**+(**Ide** → **Rv**)+(**I** → **Rv**)	*r*-values
Lv=**L**+(**Ide** → **Lv**)+(**I** → **Lv**)+(**Lv**×**L**)	*l*-values

ENVIRONMENTS

P=**E*** → **C** → **S** → **A**	command procedures
F=**E*** → **K** → **S** → **A**	expression procedures
D=**Lv**+**P**+**F**+(**F**×**L**)	denotable values
U=(**Ide** → **D**)×(**N** → **C**)	environments
E=**Lv**+**Rv**+**P**+**F**	expressible values

CONTINUATIONS

A	answers
C=**S** → **A**	command continuations
K=**E** → **S** → **A**	expression continuations
Q=**U** → **S** → **A**	definition continuations

INDEX

B

C

D

E

F

Michael J.C. Gordon

THE DENOTATIONAL DESCRIPTION OF PROGRAMMING LANGUAGES

An Introduction

Springer-Verlag
New York Heidelberg Berlin

Michael J. C. Gordon
Department of Computer Science
James Clerk Maxwell Building
University of Edinburgh
Mayfield Road
Edinburgh EH9 3JZ, Scotland
Great Britain

AMS Subject Classifications (1970): 68-A30

Library of Congress Cataloging in Publication Data

Gordon, Michael J C 1948–
The denotational description of programming languages.

Bibliography: p.
Includes indexes.
1. Programming languages (Electronic computers)
I. Title.
QA76.7.G67 001.6'424 79-15723

Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-387-90433-6 Springer-Verlag New York Heidelberg Berlin
ISBN 3-540-90433-6 Springer-Verlag Berlin Heidelberg New York

Preface

This book explains how to formally describe programming languages using the techniques of *denotational semantics*. The presentation is designed primarily for computer science students rather than for (say) mathematicians. *No* knowledge of the theory of computation is required, but it would help to have some acquaintance with high level programming languages. The selection of material is based on an undergraduate semantics course taught at Edinburgh University for the last few years. Enough descriptive techniques are covered to handle all of ALGOL 60, PASCAL and other similar languages.

Denotational semantics combines a powerful and lucid descriptive notation (due mainly to Strachey) with an elegant and rigorous theory (due to Scott). This book provides an introduction to the descriptive techniques without going into the background mathematics at all. In some ways this is very unsatisfactory; reliable reasoning about semantics (e.g. correctness proofs) cannot be done without knowing the underlying model and so learning semantic notation without its model theory could be argued to be pointless. My own feeling is that there is plenty to be gained from acquiring a purely intuitive understanding of semantic concepts together with manipulative competence in the notation. For these equip one with a powerful conceptual framework—a framework enabling one to visualize languages and constructs in an elegant and machine-independent way. Perhaps a good analogy is with calculus: for many practical purposes (e.g. engineering calculations) an intuitive understanding of how to differentiate and integrate is all that is needed. Once the utility of the ideas and techniques are appreciated it becomes much easier to motivate the underlying mathematical notions (like limits and continuity). Similarly an intuitive understanding of the descriptive techniques of denotational semantics is valuable, both as a tool for understanding programming, and as a motivation for the advanced theory.

Because the underlying mathematics is not described I have occasionally used notation which, whilst intuitively straightforward, is technically sloppy. For example I have used the symbol = with several conceptually similar but mathematically distinct, meanings. I felt it best not to introduce minor technical distinctions and restrictions if they could not be satisfactorily explained. Any reader familiar with Scott's theory should easily be able to detect and correct the few abuses of notation.

I have changed certain standard notations to be less standard but more mnemonic. Also since one or two standard symbols were not available on the typesetting machine used I have had to use substitutes. I hope the overall attractiveness of the typesetting compensates for the slightly unorthodox appearance of some of the notation.

Acknowledgements

None of the material in this book is original, that which has not been published is part of the folklore of the subject. In deciding how to present things I was strongly influenced by Milne and Strachey's advanced treatise [Milne & Strachey]. Jim Donahue, Robert Milne and Bob Tennent all made detailed criticisms of an early draft—I have tried to implement their suggestions as much as possible. Errors in the final draft were pointed out to me by members of CS4 and the postgraduate theory course (78/79); special thanks to: Gordon Brebner, Mark Jerrum, Alan Mycroft, Don Sanella and Michael Sanderson. Finally I'd like to thank the following people for numerous fruitful discussions, suggestions and explanations: Dana Angluin, Rod Burstall, Avra Cohn, Dave Macqueen, Robin Milner, Peter Mosses, Mogens Nielsen, Gordon Plotkin, Jerry Schwarz and Chris Wadsworth.

The writing of this book was supported by the Science Research Council.

Contents

1. Introduction

1.1. Syntax, semantics and pragmatics

The study of a natural or artificial language is traditionally split into three areas:

(i) Syntax: This deals with the form, shape, structure etc. of the various expressions in the language.
(ii) Semantics: This deals with the meaning of the language.
(iii) Pragmatics: This deals with the uses of the language.

For programming languages the words "syntax" and "semantics" both have fairly clear meanings; "pragmatics" on the other hand is rather more vague – some people use the term to cover anything to do with implementations (the idea being that languages are 'used' via implementations) others use the term for the study of the 'practical value' of programming concepts. We shall discuss our uses of "syntax" and "semantics" later in the book; the word "pragmatics" will be avoided.

1.2. The purposes of formal semantics

This book is about the *formal* semantics of programming languages. The word "formal" means that our study will be based on precise mathematical principles (although we shall avoid the actual mathematics). Many books on programming languages discuss meanings *intuitively* or *informally*. For example in the PASCAL report [Jensen and Wirth] the meaning of an assignment statement is described thus: "The assignment statement serves to replace the current value of a variable by a new value specified as an expression". For many purposes this is a perfectly adequate semantic description and one might wonder if there is any point in being more formal. To see that there is, we now briefly discuss three uses of formal techniques:

(i) Providing precise machine independent concepts.
(ii) Providing unambiguous specification techniques.
(iii) Providing a rigorous theory to support reliable reasoning.

We shall explain each of these in turn.

1.2.1. Providing precise and machine-independent concepts

When informally describing or comparing languages one frequently makes use of various concepts. For example recall the description of the PASCAL assignment statement quoted above. If one was trying to explain this to someone who knew nothing about programming one would have to explain:

(i) The concept of a "variable".
(ii) What the "current value" of a variable is.
(iii) How a value is "specified as an expression".

One way of explaining these would be to describe some actual implementation of PASCAL [say the one for the DEC 10 computer] and then to explain how variables, current values, expression evaluation etc. are represented. The trouble with this is that if someone else had been given explanations with respect to a different implementation [say the one for the CDC 6000 computer] then the two explanations might assign subtly different meanings to the same concepts. In any case the "essence" of these notions clearly is not dependent on any particular machine. What a formal semantics gives one is a *universal* set of concepts, defined in terms of abstract mathematical entities, which enable things like (i), (ii), (iii) above to be explained without reference to the arbitrary mechanisms of particular implementations.

Unfortunately it is rather hard to justify the analytic power of formal concepts before the concepts themselves have been described. As we proceed the reader must decide for himself whether we are really, as claimed, providing powerful tools for thought.

1.2.2. Providing unambiguous specification techniques

Both users and implementers of a programming language need a description that is comprehensible, unambiguous and complete. During the last twenty years or so notation such as Backus-Naur form (BNF for short), supported by the concepts of formal language theory, have provided adequate ways of specifying, and thinking about, the syntax of languages. Unfortunately the development of notation and concepts for dealing with semantics has been much slower. Almost all languages have been defined in English. Although these descriptions are frequently masterpieces of apparent clarity they have nevertheless usually suffered

from both inconsistency and incompleteness. For example many different interpretations were compatible with the defining reports of both ALGOL 60 [Knuth '67] and PASCAL [Welsh, Sneeringer and Hoare]. Experience has shown that it just is not possible to achieve the necessary precision using informal descriptions.

Another reason why formal semantic definitions are useful is that they can be made machine readable. This enables useful software tools to be produced—just as formal syntactic notations lead to tools such as parser generators. For example Peter Mosses has produced a system which generates test bed implementations of programming languages from formal specifications similar to those described in this book[Mosses].

1.2.3. Providing a rigorous theory to support reliable reasoning

Formal semantic techniques enable us to state and rigorously prove various properties of programs and programming languages. For example in order to prove a program correct one must show that its actual meaning coincides with its intended meaning, and to do this both meanings must be formalised.

When reasoning about programs one is often tempted to use various apparently plausible rules of inference. For example suppose at some point in a computation some sentence, S[E] say, involving the expression E is true; then after doing the assignment x:=E one would expect S[x] to hold (since x now has the value of E). Thus if "y is prime" is true before doing the assignment x:=y then "x is prime" is true after it (here E is y and S[E] is "E is prime"). Although at first sight this rule seems to be intuitively correct it does not work for most real languages. For example in PASCAL if x is of type real and y of type integer then if the sentence "y is an integer" is true before doing x:=y the sentence "x is an integer" is *not* true after it (since the value of y will have been coerced to a value of type real before being assigned to x). It can be shown [Ligler] that for ALGOL 60 the assignment rule discussed above can fail to hold in six ways.

A formal semantics provides tools for discovering and proving exactly when rules like the above one work: without a formal semantics one has to rely on intuition—experience shows this is not enough. For example the designers of EUCLID hoped that by making the language clean and simple the validity of various rules of inference would be intuitively obvious. Unfortunately when they came to actually write down the rules they found their correctness was not at all obvious and in the paper describing them [London *et al.*] they have to point out that they are still not sure the rules are all correct. If EUCLID had been given a formal semantics (of the type to be described in this book) then whether or not a given rule was correct would have been a matter of routine mathematical calculation.

In this book we shall concentrate on the topics discussed in 1.2.1. and 1.2.2.—readers interested in the underlying mathematics (and its use for rigorous reasoning) are advised to look at Joe Stoy's excellent text [Stoy].

1.3. Denotational semantics

The kind of formal semantics described in this book is called *denotational semantics*. The word "denotational" is used because in this approach language constructs are described by giving them *denotations*—these are abstract mathematical entities which model their meaning. We shall be going into great detail about what these denotations are and how they are related to the constructs that denote them—for the time being perhaps a hint of the idea can be conveyed by saying that the expressions (4+2), (12-6) and (2×3) all denote the same number, namely the number denoted by 6.

In the early days of the subject denotational semantics was called "mathematical semantics"—this term has now been abandoned since it incorrectly suggests that other kinds of semantics are non-mathematical. These other kinds of semantics—for example *operational semantics, axiomatic semantics* and *algebraic semantics*—are not alternative ways of doing what denotational semantics does; they each have their own set of goals. A certain amount of confusion has arisen from considering these kinds of semantics to be rival theories. This confusion leads one to undertake the useless activity of trying to decide which kind is best. To try and decide, for example, if denotational semantics is better than axiomatic semantics is like trying to decide if physical chemistry is better than organic chemistry.

The purposes of the various kinds of semantics are outlined below:

Denotational semantics
This is concerned with designing denotations for constructs. In this book

we shall only describe denotations which model abstract machine-independent meanings; however denotations can be (and have been) designed to model particular implementations (for examples see Vol. II of [Milne and Strachey]).

Operational semantics
This is concerned with the operations of machines which run programs—namely implementations. These operations can be described with machine oriented denotations or using non-denotationed techniques like automata theory.

Axiomatic semantics
This is concerned with axioms and rules of inference for reasoning about programs.

Algebraic semantics
This is a branch of denotational semantics making use of algebraic concepts.

1.4. Abstract entities and their description

Denotational semantics is concerned with denotations—abstract entities which model meanings. Now unfortunately such abstract entities are rather elusive—indeed it is impossible to discuss them without using some particular concrete notation. For example the only way to talk about numbers is to use some arbitrary set of names like **1**, **2**, **3** or **I**, **II**, **III** or **one**, **two**, **three** . . . etc. Since we can not *directly* manipulate abstract entities but only talk about them with languages, perhaps we should conclude that the languages are the only things that 'really exist'.

In this book we shall write as though abstract entities like numbers or sets really do exist and that they are different from the notations and languages used to talk about them. This approach is the standard one and (we believe) leads to clean and well structured thinking—for example it focuses one on the 'real problems' and prevents ones thought from getting bogged down in inessential details (like whether to use **1**, **2**, **3** or **I**, **II**, **III**—there is *one* set of numbers but many ways of naming them). However it is possible to use the notation of denotational semantics whilst

denying that this notation refers to abstract entities at all. If one adopts this attitude then a denotational description is thought of as specifying a translation from one language (the language being described) to another language (a 'meta language' consisting of the notations described in this book). The problem with this is that the metalanguage—the 'target language' of the translation—is ad hoc and not really properly defined. Now it is possible to formalize the metalanguage (see [Mosses]) but doing this, although very valuable for some purposes, is a separate task from our main goal—namely describing programming languages—just as formalizing the language of physics is a separate task from actually doing physics (the former activity is part of the philosophy of science).

Just as mathematicians use ad hoc pieces of notation to increase clarity so we use various informal abbreviations to make the descriptions of denotations clearer. If one thinks of semantics as being about abstract concepts then this is a natural and reasonable thing to do—the concepts are what really matter, the notation is just one of many possible ways of talking about them. If, on the other hand, one thinks of semantics as explaining meanings by translating them into a metalanguage then one will (rightfully) worry about the lack of precise details (for example exact syntax) of this metalanguage. For readers inclined toward the second view this book is likely to be frustrating and unsatisfactory.

2. A first example: the language TINY

In this chapter we shall describe the syntax and semantics of a little programming language called TINY. Our purpose is to provide a vehicle for illustrating various formal concepts in use. In subsequent chapters we shall describe these concepts systematically but here we just sketch out the main ideas and associated techniques.

2.1. Informal syntax of TINY

Tiny has two main kinds of constructs, *expressions* and *commands*, both of which can contain *identifiers* which are strings of letters or digits beginning with a letter (for example **x**, **y1**, **thisisaverylongidentifier**). If we let **I**, $\mathbf{I}_1$, $\mathbf{I}_2$, . . . stand for arbitrary identifiers, **E**, $\mathbf{E}_1$, $\mathbf{E}_2$, . . . stand for arbitrary expressions and **C**, $\mathbf{C}_1$, $\mathbf{C}_2$, . . . stand for arbitrary commands then the constructs of TINY can be listed as follows:

E::=0 | 1 | true | false | read | I | not E | $\mathbf{E}_1$ = $\mathbf{E}_2$ | $\mathbf{E}_1$ + $\mathbf{E}_2$
C::= I:= E | output E | if E then $\mathbf{C}_1$ else $\mathbf{C}_2$ | while E do C | $\mathbf{C}_1$;$\mathbf{C}_2$

This notation is a varient of BNF, the symbol "::=" should be read as "can be" and the symbol "|" as "or". Thus a command **C** can be **I:=E** or **output E** or **if E then $\mathbf{C}_1$ else $\mathbf{C}_2$** or **while E do C** or $\mathbf{C}_1$;$\mathbf{C}_2$. Notice that this syntactic description is ambiguous—for example it does not say whether **while E do $\mathbf{C}_1$;$\mathbf{C}_2$** is **(while E do $\mathbf{C}_1$);$\mathbf{C}_2$** or **while E do ($\mathbf{C}_1$;$\mathbf{C}_2$)**. We shall use brackets (as above) and indentation to avoid such ambiguities. In the next chapter we shall clarify further our approach to syntax.

2.2. Informal semantics of TINY

Each command of TINY, when executed, changes the *state*. This state has three components:

(i) The *memory*: this is a correspondence between identifiers and *values*. In the memory each identifier is either *bound* to some value or *unbound*.

(ii) The *input*: this is supplied by the user before programs are run; it consists of a (possibly empty) sequence of values which can be read using the expression **read** (explained later).

(iii) The *output*: this is an initially empty sequence of values which records the results of the command **output E** (explained later).

Each expression of TINY specifies a value; since expressions may contain identifiers (for example **x + y**) this value depends on the state. All values—the values of expressions, the values bound to identifiers or the values in the input or output—are either truth values (**true**, **false**) or numbers (**0,1,2,...**).

We shall now explain informally the meaning of each construct.

2.2.1. Informal semantics of expressions

The value of each expression is as follows:

(E1) **0** or **1**
The value of **0** is the number **0** and the value of **1** is the number **1**.

(E2) **true** or **false**
The value of **true** is the truth value **true** and the value of **false** is the truth value **false**.

(E3) **read**
The value of **read** is the next item on the input (an error occurs if the input is empty). **read** has the 'side effect' of removing the first item so after it has been evaluated the input is one item shorter.

(E4) **I**
The value of an expression **I** is the value bound to **I** in the memory (if **I** is unbound an error occurs).

(E5) **not E**
If the value of **E** is **true** then the value of **not E** is **false**; if the value of **E** is **false** then the value of **not E** is **true**. In all other cases an error occurs.

(E6) $\mathbf{E}_1 = \mathbf{E}_2$
The value of $\mathbf{E}_1 = \mathbf{E}_2$ is **true** if the value of $\mathbf{E}_1$ equals the value of $\mathbf{E}_2$, otherwise it is **false**.

(E7) $\mathbf{E}_1 + \mathbf{E}_2$
The value of $\mathbf{E}_1 + \mathbf{E}_2$ is the numerical sum of the values of $\mathbf{E}_1$ and $\mathbf{E}_2$ (if either of these values is not a number then an error occurs).

2.2.2. Informal semantics of commands

(C1) **I: = E**
I is bound to the value of **E** in the memory (overwriting whatever was previously bound to **I**).

(C2) **output E**
The value of **E** is put onto the output.

(C3) **if E then C_1 else C_2**
If the value of **E** is **true** then C_1 is done, if its value is **false** then C_2 is done (in any other case an error occurs).

(C4) **while E do C**
If the value of **E** is **true** then **C** is done and then **while E do C** is repeated starting with the state resulting from **C**'s execution. If the value of **E** is **false** then nothing is done. If the value of **E** is neither **true** nor **false** an error occurs.

(C5) $C_1;C_2$
C_1 and then C_2 are done in that order.

2.3. An example

The following TINY command outputs the sum of the numbers on the input. The end of the input is marked with **true**.

```
sum: = 0; x: = read;
while not (x = true) do sum: = sum + x; x: = read;
output sum
```

2.4. Formal semantics of TINY

We shall now informally formalize the above description of TINY. Our hope is to convey the general 'shape' of a denotational description. The reader should not attempt to grasp all the details (some of which are over-simplified) but just get the main ideas.

An essential part of our formalization will be the defining of various sets (for example sets of denotations). It turns out that some of the ways of defining sets we need only work properly if we use certain special sets called *domains*. We shall thus use "domain" instead of "set" – however intuitively domains can be thought of just like sets (indeed the class of

domains is just a subclass of the class of sets) and we shall employ normal set theoretic notation on them. For example {**x** | P[**x**]} is the set of all **x**'s satisfying P[**x**]; **x**ε**S** means **x** belongs to **S**; **f**:$\mathbf{S}_1 \to \mathbf{S}_2$ means **f** is a function from $\mathbf{S}_1$ to $\mathbf{S}_2$. In the next chapter (in fact in 3.2.) we shall explain why domains rather than sets must be used.

2.4.1. Syntax

To deal with the syntax we define the following *syntactic domains*:

Ide = {**I** | **I** is an identifier}
Exp = {**E** | **E** is an expression}
Com = {**C** | **C** is a command}

Domain names like these will always start with a capital letter. **Ide** is a standard domain, **Exp** and **Com** vary from language to language.

2.4.2. States, memories, inputs, outputs and values

We start by formalizing the concept of a state. To do this we define domains **State** of states, **Memory** of memories, **Input** of inputs, **Output** of outputs and **Value** of values. The definition of these domains consists of the following *domain equations* which we first state and then explain:

(i) **State = Memory × Input × Output**
(ii) **Memory = Ide → (Value + {unbound})**
(iii) **Input = Value***
(iv) **Output = Value***
(v) **Value = Num + Bool**

(i) Means that **State** is the domain of all triples (**m**,**i**,**o**) where **m**ε **Memory**, **i**ε**Input** and **o**ε**Output**. In general $\mathbf{D}_1 \times \mathbf{D}_2 \times \ldots \times \mathbf{D}_n$ is the domain $\{(\mathbf{d}_1,\mathbf{d}_2,\ldots,\mathbf{d}_n) \mid \mathbf{d}_1\varepsilon\mathbf{D}_1,\mathbf{d}_2\varepsilon\mathbf{D}_2,\ldots,\mathbf{d}_n\varepsilon\mathbf{D}_n\}$ of n-tuples.

(ii) Means that **Memory** is the domain of all functions from the domain **Ide** to the domain **[Value + {unbound}]**. The domain **[Value + {unbound}]** is the union of the domain **Value** and the one element domain **{unbound}**. In general $[\mathbf{D}_1 \to \mathbf{D}_2]$ is the domain $\{\mathbf{f} \mid \mathbf{f}:\mathbf{D}_1 \to \mathbf{D}_2\}$ of all functions from $\mathbf{D}_1$ to $\mathbf{D}_2$ and $[\mathbf{D}_1 + \mathbf{D}_2]$ is the 'disjoint union' of $\mathbf{D}_1$ and $\mathbf{D}_2$ – we clarify this later; for now think of + as ordinary union restricted to disjoint domains. If **m**ε**Memory** and **I**ε**Ide** then **m I**, the result of applying the function **m** to argument **I**, is either in **Value** or is **unbound**; in the former case the

value **m I** is the value bound to **I** in **m**, in the latter case **I** is unbound in **m**.

(iii) Means **Input** is the domain of all strings (including the empty string) of members of **Value**. In general $\mathbf{D}^*$ is the domain

$$\{(d_1,\ldots,d_n) \mid d_1 \varepsilon D, d_2 \varepsilon D, \ldots, d_n \varepsilon D\}$$

of strings or sequences over **D**. We shall denote the empty string by **( )** and sometimes write $d_1 . d_2 . \ldots . d_n$ instead of $(d_1,\ldots,d_n)$.

(iv) Means **Output** is the domain of all strings of values.

(v) Means that a value is either a number (i.e. a member of **Num**) or a truth value (i.e. a member of **Bool**). **Num** = {0,1,2,...} and **Bool** = {**true**, **false**}. Thus a state is a triple **(m,I,o)** where **m** is a function from identifiers to values (or **unbound**) and **I** and **o** are sequences of values.

2.4.3. Semantic functions

In this section we discuss the *semantic functions* for TINY. Semantic functions are functions which define the denotation of constructs. For TINY we need:

E:Exp→(denotations of expressions)
C:Com→(denotations of commands)

If **E** ε **Exp** and **C** ε **Com** then **E[E]** and **C[C]** are the results of applying the functions **E** and **C** to **E** and **C** respectively and are the denotation of the corresponding constructs defined by the semantics. We discuss these denotations later, but first note that:

(i) In general if **X** is a variable which ranges over some syntactic domain of constructs then we will use **X** for the corresponding semantic functions. Thus **C[I: = E]** is the denotation – or meaning – of **I: = E** etc.

(ii) The "emphatic brackets" **[** and **]** are used to surround syntactic objects when applying semantic functions to them. They are supposed to increase readability but have no other significance – **X[X]** is just the result of applying the function **X** to **X**.

2.4.3.1. Denotations of expressions

Since expressions produce values one might at first take their denotations to be members of **Value**. To model this idea one would give the semantic function **E** the type **Exp→Value** and then **E[E]** would be **E**'s value (e.g. **E[0] = 0**). This works for constant expressions but in general it fails to handle:

(i) The possibility of expressions causing errors (for example **1 + true**).

(ii) The dependence of some expression's values on the state (for example the value of **x + 1** depends on what **x** is bound to in the memory; the value of **read** depends on the input).

(iii) The possibility that the evaluation of an expression might change the state (for example **read** removes the first item from the input).

To handle (i) we must define **E:Exp→[Value + {error}]** so that:

$$\mathbf{E[E]} = \begin{cases} \mathbf{v} & \text{if } \mathbf{v} \text{ is } \mathbf{E}\text{'s value} \\ \mathbf{error} & \text{if } \mathbf{E} \text{ causes an error} \end{cases}$$

For example **E[1 + 1] = 2** but **E[1 + true] = error**.

To handle (ii) we must make the result of an evaluation a function of the state – i.e. define **E:Exp→[State→[Value + {error}]]** so that:

$$\mathbf{E[E]s} = \begin{cases} \mathbf{v} & \text{if } \mathbf{v} \text{ is } \mathbf{E}\text{'s value in } \mathbf{s} \\ \mathbf{error} & \text{if the evaluation causes an error} \end{cases}$$

For example:

$$\mathbf{E[1 + x]s} = \begin{cases} \mathbf{1 + (mx)} & \text{if } \mathbf{s = (m, I, o)} \text{ and } \mathbf{mx} \text{ is a number} \\ \mathbf{error} & \text{otherwise} \end{cases}$$

Thus the denotation **E[E]** of **E** is a function of type

Exp→[State→[Value + {error}]].

Finally to handle (iii) we must further complicate **E**'s type so that:

E:Exp→[State→[[Value × State] + {error}]]

and then

$$\mathbf{E[E]s} = \begin{cases} \mathbf{(v, s')} & \text{where } \mathbf{v} \text{ is } \mathbf{E}\text{'s value in } \mathbf{s} \text{ and } \mathbf{s'} \text{ is the state after the evaluation.} \\ \mathbf{error} & \text{if an error occurs.} \end{cases}$$

For example:

$$\mathbf{E[read]s} = \begin{cases} \mathbf{(hd\ l, (m, tl\ i, s))} & \text{if } \mathbf{s = (m, i, o)}, \mathbf{i} \text{ is non empty and has first member } \mathbf{hd\ i} \text{ and the rest is } \mathbf{tl\ i}. \\ \mathbf{error} & \text{if the input is empty.} \end{cases}$$

We formally define **E** by cases on the different kinds of expressions. For example the denotations of expressions of the form **I** are defined by the following *semantic clause*:

$$\mathbf{E[I]\,(m, i, o) = (m\,I = unbound) \rightarrow error, (m\,I, (m, i, o))}$$

Here "$\mathbf{b \rightarrow v_1, v_2}$" means "if **b** is true then $\mathbf{v_1}$ else $\mathbf{v_2}$"—we give a precise account of this notation in 3.4.5. Thus the above semantic clause says that if **m I** is **unbound** (i.e. **I** is unbound in **m**) then an error occurs otherwise the value of **I** is whatever it is bound to in the memory. The state resulting from the evaluation is **(m, i, o)**—i.e. the original state. An example of a semantic clause in which the evaluation changes the state is:

$$\mathbf{E[read]\,(m, i, o) = null\ i \rightarrow error, (hd\ i, (m, tl\ i, o))}$$

Here **null i** is true if **i** is empty, **hd i** is the first element of **i** and **tl i** the rest (see 3.3.3.3.). Thus if the input is empty an error results otherwise the value of **read** is the first item in the input and the resulting state has this item removed.

The rest of the semantic clauses for TINY are described in 2.4.4. below.

2.4.3.2. Denotations of commands

The effect of executing a command is to produce a new state or generate an error, thus:

$$\mathbf{C: Com \rightarrow [State \rightarrow [State + \{error\}]]}$$

a typical semantic clause is:

$$\mathbf{C[output\ E]\,s = (E[E]s = (v, (m, i, o))) \rightarrow (m, i, v.o), error}$$

Here **v.o** is the string resulting from sticking **v** onto the front of **o**. Thus this semantic clause says that **C[output E]** is a function which when applied to a state **s** first evaluates **E** and if **E** produces a value **v** and new

state **(m, i, o)** then the result is **(m, i, v. o)** otherwise the result is **error**.

2.4.4. Semantic clauses

In this section we describe and explain the semantic clauses for TINY.

2.4.4.1. Clauses for expressions

(E1) $\mathbf{E[0]s = (0, s)}$
$\mathbf{E[1]s = (1, s)}$

The value of a numeral is the corresponding number; the evaluation does not change the state.

(E2) $\mathbf{E[true]s = (true, s)}$
$\mathbf{E[false]s = (false, s)}$

The value of a boolean constant is the corresponding boolean value (truth value); the evaluation does not change the state.

(E3) $\mathbf{E[read]\,(m, i, o) = null\ i \rightarrow error, (hd\ i, (m, tl\ i, o))}$

This was explained above, **null i** is **true** if **i** is empty and **false** otherwise; **hd i** is the first element of **i** and **tl i** is the rest of **i**.

(E4) $\mathbf{E[I]\,(m, i, o) = (m\,I = unbound) \rightarrow error, (m\,I, (m, i, o))}$

This was explained in 2.4.3.1. above.

(E5) $\mathbf{E[not\ E]\,s = (E[E]s = (v, s')) \rightarrow (isBool\ v \rightarrow (not\ v, s'), error), error}$

isBool v is **true** if $\mathbf{v \varepsilon Bool}$ and is **false** otherwise (here $\mathbf{v \varepsilon Value = Num + Bool}$), **not: Bool → Bool** is the function defined by **not true = false** and **not false = true**. Thus the value of **not E** is **not** of the value of **E** (or **error** if **E**'s evaluation leads to a number or **error**) and the state is changed to the state **s'** resulting from **E**'s evaluation in **s**.

(E6) $\mathbf{E[E_1 = E_2]\,s = (E[E_1]s = (v_1, s_1)) \rightarrow ((E[E_2]s_1 = (v_2, s_2)) \rightarrow (v_1 = v_2, s_2), error), error}$

Here $\mathbf{v_1 = v_2}$ is **true** if $\mathbf{v_1}$ equals $\mathbf{v_2}$ and **false** otherwise. Thus the result of $\mathbf{E_1 = E_2}$ in **s** is obtained by first evaluating $\mathbf{E_1}$ in **s** to get $\mathbf{(v_1, s_1)}$ (or **error** —in which case **error** is the value of $\mathbf{E_1 = E_2}$), then evaluating $\mathbf{E_2}$ in $\mathbf{s_1}$ to

get (v_2, s_2) (or **error**—in which case **error** is the value of $E_1 = E_2$), finally $(v_1 = v_2, s_2)$ is returned as the result of $E_1 = E_2$.

(E7) $$\mathbf{E}[\![E_1 + E_2]\!]s = (\mathbf{E}[\![E_1]\!]s = (v_1, s_1))\rightarrow (\mathbf{E}[\![E_2]\!]s_1 = (v_2, s_2))\rightarrow (\mathbf{isNum}\ v_1\ \mathbf{and}\ \mathbf{isNum}\ v_2 \rightarrow (v_1 + v_2, s_2), \mathbf{error}), \mathbf{error}), \mathbf{error}$$

This semantic clause is similar to the preceding one. **isNum v** is **true** if **v** is a number (i.e. member of **Num**) and **false** otherwise. Thus (E7) says that to evaluate $E_1 + E_2$ one evaluates E_1, then evaluates E_2 (in the state resulting from E_1) then tests their values to make sure they are numbers and if so returns their sum and the state resulting from E_2. If either of these values is not a number or if E_1 or E_2 generates an error then $E_1 + E_2$ generates an error.

2.4.4.2. Clauses for commands

(C1) $$\mathbf{C}[\![I := E]\!]s = (\mathbf{E}[\![E]\!]s = (v, (m, i, o)))\rightarrow (m[v/I], i, o), \mathbf{error}$$

$$(m[v/I])\,I' = \begin{cases} v & \text{if } I = I' \\ m\,I' & \text{otherwise} \end{cases}$$

Thus $\mathbf{C}[\![I := E]\!]s$ is a state identical to the state resulting from the evaluation of **E** except that **E**'s value **v** is bound to **I** in the memory (if **E** produces **error** then so does **I := E**).

(C2) $$\mathbf{C}[\![\mathbf{output}\ E]\!]s = (\mathbf{E}[\![E]\!]s = (v, (m, i, o)))\rightarrow (m, i, v \,.\, o), \mathbf{error}$$

The semantic clause was explained in 2.4.3.2. above.

(C3) $$\mathbf{C}[\![\mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!]s = ((\mathbf{E}[\![E]\!]s = (v, s'))\rightarrow (\mathbf{isBool}\ v \rightarrow (v \rightarrow \mathbf{C}[\![C_1]\!]s', \mathbf{C}[\![C_2]\!]s'), \mathbf{error}), \mathbf{error})$$

isBool v is **true** if **v** is a truth value (i.e. if $v \in \mathbf{Bool} = \{\mathbf{true}, \mathbf{false}\}$) and is **false** otherwise. Thus if **E** produces result (v, s') when evaluated in **s** then C_1 or C_2 are executed (in the state s' resulting from **E**) depending on whether **E**'s value **v** is **true** or **false**.

(C4) $$\mathbf{C}[\![\mathbf{while}\ E\ \mathbf{do}\ C]\!]s = ((\mathbf{E}[\![E]\!]s = (v, s'))\rightarrow (\mathbf{isBool}\ v \rightarrow (v \rightarrow (\mathbf{C}[\![C]\!]s' = s'')\rightarrow \mathbf{C}[\![\mathbf{while}\ E\ \mathbf{do}\ C]\!]s'', \mathbf{error}), s'), \mathbf{error}), \mathbf{error})$$

If **E** produces an error then so does **while E do C**; if **E** produces value **v** and a new state s' then if **v** is **true** **C** is done to get s'' and then **while E do C** is done starting with s''. If **v** is **false** then the result of **while E do C** is s' the result of **E**. Finally if **v** is not a truth value or $\mathbf{C}[\![C]\!]s' = \mathbf{error}$ then **while E do C** causes an error. Notice that (C4) is *recursive* $\mathbf{C}[\![\mathbf{while}\ E\ \mathbf{do}\ s]\!]$ occurs on the right hand side.

(C5) $$\mathbf{C}[\![C_1; C_2]\!]s = (\mathbf{C}[\![C_1]\!]s = \mathbf{error})\rightarrow \mathbf{error}, \mathbf{C}[\![C_2]\!](\mathbf{C}[\![C_1]\!]s)$$

Thus if C_1 generates an error then so does $C_1; C_2$ otherwise C_2 is done in the state resulting from C_1.

This completes the semantic clauses, and also the formal semantics, of TINY. In some of the clauses, especially (E6), (E7), (C3) and (C4), the tests for the various error conditions makes it hard to follow what is going on—the semantics of non error executions fails to stand out. In the next chapter we describe various notations for clarifying and simplifying semantic clauses; the reader may like to peep ahead to chapter 4 to see how neat the clauses can be made.

An important thing to note about the semantic clauses above is that each construct has its denotation defined in terms of the denotation of its components; for example $\mathbf{C}[\![C_1; C_2]\!]$ is defined in terms of $\mathbf{C}[\![C_1]\!]$ and $\mathbf{C}[\![C_2]\!]$. The only exception to this is (C4), the clause for **while E do C**, where the denotation is defined 'in terms of itself'—we shall have more to say on such recursive definitions in 3.2.1.

2.4.5. Summary of the formal semantics of TINY

The formal semantic description of TINY just given had three main parts:

(i) Specification of the syntactic domains **Exp** and **Com**.
(ii) Specification of the semantic domains **State**, **Value** etc.
(iii) Specification of the semantic functions **E** and **C** which map syntactic entities to semantic ones.

In the next chapter we shall describe in detail the concepts and notations of these specifications. In subsequent chapters we shall refine and extend the concepts and notation, and apply the techniques to a wide variety of programming constructs (including most of ALGOL 60 and PASCAL).

3. General concepts and notation

This chapter is rather long and tedious—it is a 'user manual' for the notation and concepts we shall need. On first reading one should only look in detail at 3.1. and 3.2., the other sections should be quickly skimmed and then referred to later when their contents are used. Here is a quick overview of the main sections:

3.1. We explain the concept of 'abstract syntax'—the kind of syntactic description convenient for semantics.
3.2. We explain why we use "domains" rather than sets and discuss informally the role of the underlying mathematical theories.
3.3. We describe different kinds of domains and ways of building them.
3.4. We discuss the concept of a function and then describe numerous notations for manipulating them.

3.1. Abstract syntax

The programs of most languages are built out of various kinds of *constructs* such as identifiers, expressions, commands and declarations. For example let **C** be the TINY command **x:=read; sum:=sum+x**; then:

C is the command $\mathbf{C}_1;\mathbf{C}_2$
  where $\mathbf{C}_1$ is the command $\mathbf{I}_1:=\mathbf{E}_1$
    where $\mathbf{I}_1$ is the identifier **x**
    and $\mathbf{E}_1$ is the expression **read**
  and $\mathbf{C}_2$ is the command $\mathbf{I}_2:=\mathbf{E}_2$
    where $\mathbf{I}_2$ is the identifier **sum**
    and $\mathbf{E}_2$ is the expression $\mathbf{E}_{21}+\mathbf{E}_{22}$
      where $\mathbf{E}_{21}$ is the identifier **sum**
      and $\mathbf{E}_{22}$ is the identifier **x**

We call the various constructs which make up a given construct its *constituents*, for example $\mathbf{C}_1$, $\mathbf{C}_2$, $\mathbf{I}_1$, $\mathbf{I}_2$, $\mathbf{E}_1$, $\mathbf{E}_2$, $\mathbf{E}_{21}$, $\mathbf{E}_{22}$ are the constituents of **C**. The *immediate constituents* of a construct are its 'biggest' constituents, for example the immediate constituents of **C** are $\mathbf{C}_1$ and $\mathbf{C}_2$, the immediate constituents of $\mathbf{C}_1$ are $\mathbf{I}_1$ and $\mathbf{E}_1$, and the immediate constituents of $\mathbf{C}_2$ are $\mathbf{I}_2$ and $\mathbf{E}_2$.

As we noted after listing the semantic clauses of TINY the denotation assigned to a construct by a denotational semantics only depends on the kind of construct and the denotations of its immediate constituent. Thus for the purposes of semantics a syntactic description need only specify the various constructs and what their immediate constituents are. Other details of syntax—for example precedence—are irrelevent for semantics and can thus be ignored. A syntactic description which just lists the kinds of constructs and their immediate constituents is called an *abstract syntax*.

The notation we shall use to specify abstract syntax has two parts:

(i) A list of the various *syntactic categories* of the language (for TINY the syntactic categories are expressions and commands). For each category we provide a name for the domain of all constructs of that category (for TINY the names are "**Exp**" and "**Com**") and a *metavariable* to range over the domain (in TINY **E** ranges over **Exp** and **C** ranges over **Com**).

(ii) A list of *syntactic clauses*, each one specifying the various kinds of constructs in a category (for TINY the syntactic clauses were given in 2.1., namely **E** ::= **0** | ... and **C** ::= **I**:=**E** | ...).

The metavariables stand for constructs of the corresponding category (i.e. members of the corresponding syntactic domain). To distinguish different instances metavariables may be primed or subscripted (for example **E**, **E**', $\mathbf{E}_1$, $\mathbf{E}_2$ stand for expressions). The notation for each kind of construct given in the syntactic clauses must display the immediate constituents and distinguish it from other kinds of construct, but otherwise it is arbitrary. Usually one chooses a notation based on some actual concrete syntax for the language being defined.

We shall always use a fixed standard domain **Ide** of *identifiers* (with metavariable **I**). Although languages do in fact differ in what they allow as identifiers these differences are not normally *semantically* significant.

3.2. Sets and domains

In describing the semantics of TINY we mysteriously used the word "domain" instead of "set". As we hinted this was for technical mathematical reasons. Unfortunately to fully explain the difference between sets

and domains, and to convincingly justify our use of the latter, we would have to delve in detail into the underlying mathematics. This we wish to avoid and so I shall just very crudely sketch some of the issues involved. Readers who are not satisfied with this discussion should consult Joe Stoy's excellent book [Stoy] where all details are lucidly explained.

There are really two related, but separate, problems which lead to the use of domains rather than sets:

(i) Recursive definitions of functions.
(ii) Recursive definitions of sets.

We shall look at these in turn.

3.2.1. The problem of recursively defined functions

We often find it necessary to define functions recursively. For example the semantic clause for **while E do C**, (C4), defines **C[while E do C]** as some expression involving **C[while E do C]**. At first sight such 'definitions' are highly suspicious since they seem to assume the thing they are trying to define is already defined. The way out is to regard recursive definitions as *equations*—to regard (C4) as analogous to the quadratic equation $\mathbf{x} = \frac{1}{3}\mathbf{x}^2 + \frac{2}{3}$ (which 'defines' **x** to be **1** or **2**) and to seek methods of solving functional equations, like (C4), analogous to the methods we have for solving quadratic equations. It turns out that the kind of functional equations we want to solve in semantics only have solutions when the functions involved map between specially structured sets called domains. To see what goes wrong with ordinary sets consider the following two equations 'defining' **f**: **Num**→**Num**

(i) **f x = (f x) + 1**
(ii) **f x = f x**

If **Num** = {0,1,2,...} there is *no* **f** satisfying (i) and *every* **f** satisfies (ii), thus neither of these two equations in any sense defines **f**. On the other hand the equation:

(iii) **f x = (x = 0)→1, x × f(x-1)**

uniquely defines **f** to be the factorial function (i.e. for all **x**: **f x = x!**). If we used arbitrary sets then it would be just as easy to write bad definitions

like (i) and (ii) as good ones like (iii). What the theory of domains does is ensure *every* definition is good. It does this by:

(a) Requiring all domains to have a certain 'implicit structure' which can be shown to guarantee that all equations (including (i), (ii) and (iii)) have at least one solution.

(b) Providing a way, via the 'implicit structure', of choosing an 'intended solution' from among the various solutions guaranteed by (a).

For example the domain **Num**, in virtue of its 'implicit structure', contains besides **0**, **1**, **2**, ... an 'undefined' element **undefined** and then (i) and (ii) both end up defining **f** to be the 'undefined function': **f x = undefined** for all **x**$\in$**Num**. (iii) defines **f** to satisfy **f x = x!** if **x = 0**, **x = 1**, ... and **f undefined = undefined**. We shall not go into the details obscurely hinted at here; all the reader needs to know is that recursive definitions can be solved in a way that justifies the intuitions behind them. Some further discussion on recursive definitions occurs in 3.4.9.

3.2.2. The problem of recursively defined sets

It is often the case that intuitions about the 'data' manipulated by programs in a language leads us to want to define *recursively* the domains modelling the corresponding 'data-types'. As an example let us examine one way of adding parameterless procedures to TINY. We add to the syntax of TINY a new kind of expression **proc C** and a new kind of command **I**. The informal semantics of these is:

(E8) **proc C**
The value of **proc C** is a procedure which when invoked (by an identifier denoting it) executes **C**.

(C6) **I**
When the command **I** is executed the procedure denoted by **I** is invoked.

For example in a program:

```
P := proc (x := x + 1);
      ⋮
P;
      ⋮
P;
      ⋮
```

each time a **P** was executed **x** would be incremented by 1.

To handle the formal semantics we must extend the domain **Value** to include the denotations of procedures. Thus:

Value = Num + Bool + Proc

For denotations of procedures we simply take the state transformation done each time the procedure is invoked. Thus

Proc = State → [State + {error}]

Now the semantic clauses are simply:

(E8) **E[proc C] s = (C[C],s)**
(C6) **C[I](m,i,o) = (m I = unbound) → error,**
isProc(m I) → m I (m,i,o) , error

Here **m I (m,i,o)** is the procedure value **m I** denoted by **I** in **m** applied to the state **(m,i,o)**. To see the point of this example let us write out the domain equations for this extended TINY:

State = Memory × Input × Output
Memory = Ide → [Value + {unbound}]
Input = Value*
Output = Value*
Value = Num + Bool + Proc
Proc = State → [State + {error}]

These domain equations are recursive: **State** is defined in terms of **Input** which is defined in terms of **Value** which is defined in terms of **Proc** which is defined in terms of **State**. Thus just as the 'looping' of **while E do C** led us to define its denotation as a recursive *function*, so the embedding of procedures in states leads us to define **State** as a recursive

Domain. Now just as there are problems with recursive definitions of functions so there are problems—much harder problems in fact—with recursive definitions of domains. For example it can be shown mathematically that there are no *sets* satisfying the domain equations above! Fortunately if we work with domains, and interpret the domain building operators $\times$, $\rightarrow$, $+$ in a clever way, then solutions do exist. In other words set equations cannot in general be solved but domain equations can. The crucial two ideas which ensure all domain equations have solutions are firstly to define $[\mathbf{D}_1 \rightarrow \mathbf{D}_2]$ to be the domain, not of all functions, but just of those functions which preserve the 'implicit structure' on the domains $\mathbf{D}_1$ and $\mathbf{D}_2$ (this 'implicit structure' is present as a consequence of the exact definition of a domain), and secondly to interpret $=$ not as equality between domains but as 'isomorphism'. Alas we cannot explain these tantalising remarks further here.

3.2.3. The role of Dana Scott's theory

A major breakthrough in semantic theory came when Dana Scott showed [Scott] how to consistently interpret both kinds of recursive definitions—definitions of members of domains (see 3.2.1.) and definitions of domains themselves (see 3.2.2.)—in a single unified framework. In summary what he did was:

(i) Devise a class of 'structured' sets called domains and define the operators $\times$, $+$, $\rightarrow$, $*$ etc. on them.
(ii) Show how elements of domains could be defined recursively.
(iii) Show how domains themselves could be defined recursively.

Many of the intuitive ideas of denotational semantics have been around for ages—for example in 1966 Strachey published a paper "Towards a Formal Semantics" [Strachey] which contains several of the key concepts described in this book, and even before that John McCarthy had outlined what was essentially a denotational semantics of a fragment of ALGOL 60 [McCarthy].

The main achievement of Dana Scott was firstly to point out that the naive formalization of some of the early semantic models could not be consistently extended to non-trivial languages, and then to show how a consistent (and very beautiful) theory was in fact possible if domains (instead of arbitrary sets) were used to model denotations.

The problem with inconsistent formalizations (like Strachey's potentially was) is that they provide no basis for reliable reasoning. For example consider the proof of $\mathbf{1}=\mathbf{2}$ the proceeds by first assuming $\mathbf{0}=\mathbf{1}$ and then adding $\mathbf{1}$ to both sides—proofs of properties of programs from inconsistent formalization are just like this!

Of course the early inconsistent theories were not without value, for one thing they prompted the discovery of the modern consistent versions. It is usually easier to formalize ideas than to think of them and the early pioneers of semantics must take the credit for originating the general approach if not the fine details. There are plenty of other examples of inconsistent ideas being useful—for example the use of infinitesimals in calculus and the use of Dirac delta-functions in physics; both these were valuable for years (indeed centuries) before consistent models of them were devised.

Thus although the idea of denotational semantics is quite old it was nevertheless a major breakthrough when in 1969 Scott discovered his theory; for only then could we start to use semantic models as the basis for *trustworthy* proofs. Furthermore his theory has enormously sharpened our intuitions about the concepts involved and it is inconceivable that the descriptive techniques described in this book could have advanced as fast as they have without it. Also it seems that future developments in semantics (for example to handle concurrency [Milne and Milner]) will be even more dependent on the underlying theory.

3.2.4. The role of mathematics in the book

We shall not discuss the mathematics involved in Scott's theory at all; our approach to recursive equations is similar to an engineers approach to differential equations, namely we assume they have solutions but don't bother with the mathematical justification. In practice this is perfectly satisfactory for the kinds of things we discuss in this book; one only needs to dabble in the mathematics if:

(i) One wishes to perform rigorous proofs.
(ii) One wishes to devise descriptive techniques for radically different kinds of constructs (for example involving concurrency).

3.3. Defining domains

In this section we explain all the ways of defining domains we shall need. **D**, **D'**, $\mathbf{D_1}$, $\mathbf{D_2}$, ... etc. will stand for arbitrary domains and the names of all particular domains will start with a capital letter (for example: **Num**, **Value**, **State**).

3.3.1. Standard domains

The following domains are standard and will be used without further explanation:

numbers: **Num** = {0, 1, 2, ...}
truth values: **Bool** = {**true**, **false**}
identifiers: **Ide** = {I | I is a string of letters or digits beginning with a letter}

3.3.2. Finite domains

Finite domains will be defined explicitly by listing their elements, for example:

Spectrum = {**red**, **orange**, **yellow**, **green**, **blue**, **indigo**, **violet**}

3.3.3. Domain constructors

We shall build domains out of standard, or finite, domains using the various *domain constructors* described below. Since we are avoiding the underlying mathematics some of the descriptions will be slightly oversimplified; these oversimplifications only concern technical details and do not affect the main ideas.

3.3.3.1. Function space $[\mathbf{D_1}\to\mathbf{D_2}]$

$[\mathbf{D_1}\to\mathbf{D_2}]$ is the domain of functions from $\mathbf{D_1}$ to $\mathbf{D_2}$.

$$[\mathbf{D_1}\to\mathbf{D_2}] = \{\mathbf{f} \mid \mathbf{f}: \mathbf{D_1}\to\mathbf{D_2}\}$$

Comments

(i) $\mathbf{f}\varepsilon[\mathbf{D_1}\to\mathbf{D_2}]$ if and only if $\mathbf{f}: \mathbf{D_1}\to\mathbf{D_2}$ and we say "**f** has type $\mathbf{D_1}\to\mathbf{D_2}$". In general for any domain **D** (not necessarily a function space) If $\mathbf{d}\varepsilon\mathbf{D}$ we say "**d** has type **D**" and write "**d**: **D**".

(ii) If $\mathbf{f}\varepsilon[\mathbf{D_1}\to\mathbf{D_2}]$ then $\mathbf{D_1}$ is called the *source* of **f** and $\mathbf{D_2}$ the *target* of **f** (sometimes "domain" and "range" are used instead of "source" and "target" — but this conflicts with our other uses of the word "domain").

(iii) We may write $\mathbf{D_1}\to\mathbf{D_2}$ instead of $[\mathbf{D_1}\to\mathbf{D_2}]$; by convention $\to$ associates to the right so, for example,

$$\mathbf{D_1}\to\mathbf{D_2}\to\mathbf{D_3}\to\mathbf{D_4} \text{ means } [\mathbf{D_1}\to[\mathbf{D_2}\to[\mathbf{D_3}\to\mathbf{D_4}]]].$$

(iv) In Scott's theory $[\mathbf{D_1}\to\mathbf{D_2}]$ is defined to be the set of all functions which 'preserve the structure' of the domains; thus strictly speaking not every function **f** is in $[\mathbf{D_1}\to\mathbf{D_2}]$. In practice all the functions we shall use — indeed all the functions our notations allow us to define — do 'preserve structure' and no problems arise. Our assumption that all functions are in $[\mathbf{D_1}\to\mathbf{D_2}]$ is analogous to the engineers assumption that all functions are differentiable.

3.3.3.2. Product $[\mathbf{D_1}\times\mathbf{D_2}\times\ldots\times\mathbf{D_n}]$

$[\mathbf{D_1}\times\mathbf{D_2}\times\ldots\times\mathbf{D_n}]$ is the domain of all n-tuples $(\mathbf{d_1}, \mathbf{d_2}, \ldots, \mathbf{d_n})$ of elements $\mathbf{d_1}\varepsilon\mathbf{D_1}$, $\mathbf{d_2}\varepsilon\mathbf{D_2}$, ..., $\mathbf{d_n}\varepsilon\mathbf{D_n}$. If $\mathbf{d}\varepsilon[\mathbf{D_1}\times\mathbf{D_2}\times\ldots\times\mathbf{D_n}]$ then **el i d** is the ith coordinate of **d**. Thus

$$[\mathbf{D_1}\times\mathbf{D_2}\times\ldots\times\mathbf{D_n}] = \{(\mathbf{d_1}, \mathbf{d_2}, \ldots, \mathbf{d_n}) \mid \mathbf{d_1}\varepsilon\mathbf{D_1}, \mathbf{d_2}\varepsilon\mathbf{D_2}, \ldots, \mathbf{d_n}\varepsilon\mathbf{D_n}\}$$
$$\mathbf{el\ i}\ (\mathbf{d_1}, \mathbf{d_2}, \ldots, \mathbf{d_n}) = \mathbf{d_i}$$
$$\mathbf{d} = (\mathbf{el\ 1\ d}, \mathbf{el\ 2\ d}, \ldots, \mathbf{el\ n\ d})$$

3.3.3.3. Sequences $\mathbf{D}^*$

$\mathbf{D}^*$ is the domain of all *finite* sequences of elements of **D**. If $\mathbf{d}\varepsilon\mathbf{D}^*$ then either **d** is the empty sequence ( ) or $\mathbf{d} = (\mathbf{d_1}, \mathbf{d_2}, \ldots, \mathbf{d_n})$ where $\mathbf{n}>\mathbf{0}$ and each $\mathbf{d_i}$ is a member of **D**. As with the product **el i d** is the ith coordinate of **d**; **hd d** is the first member of **d** (thus **hd d** = **el 1 d**) and **tl d** is the sequence consisting of all but the first element of **d**; **null d** is **true** if **d** is the empty string and **false** otherwise. Thus:

$D^* = \{(d_1, d_2, \ldots, d_n) \mid 0<n, d_i \varepsilon D\} + \{( )\}$

$\mathbf{el}\ i\ (d_1, d_2, \ldots, d_n) = d_i$
$\mathbf{hd}\ (d_1, d_2, \ldots, d_n) = d_1$
$\mathbf{tl}\ (d_1, d_2, \ldots, d_n) = (d_2, \ldots, d_n)$

$$\mathbf{null}\ d = \begin{cases} \mathbf{true} & \text{if } d = ( ) \\ \mathbf{false} & \text{otherwise} \end{cases}$$

Comments (i) **el**, **hd**, **tl** and **null** can be thought of as functions with types:

$$\mathbf{el}:\mathbf{Num} \to D^* \to D$$
$$\mathbf{hd}:D^* \to D$$
$$\mathbf{tl}:D^* \to D^*$$
$$\mathbf{null}:D^* \to \mathbf{Bool}$$

el, **hd** and **tl** only make sense when applied to non-empty sequences.

(ii) An alternative notation for the sequence $(d_1, d_2, \ldots, d_n)$ is $d_1 . d_2 . \ldots . d_n$. Also if $d \varepsilon D$ and $d^* = (d_1, d_2, \ldots, d_n) \varepsilon D^*$ then $d . d^*$ is the sequence $(d, d_1, d_2, \ldots, d_n)$. When using this notation we may refer to sequences as *strings*.

3.3.3.4. Sum $[D_1 + D_2 + \ldots + D_n]$

Each member of $[D_1 + D_2 + \ldots + D_n]$ corresponds to *exactly one* member of some D_i. The difference between the sum $[D_1 + D_2 + \ldots + D_n]$ and the union of $D_1, \ldots, D_n$ is that if d is in the union and $d \varepsilon D_i$ and $d \varepsilon D_j$ (for some $i \neq j$) then it does not make sense to ask if d comes from D_i or D_j. Each member of $[D_1 + D_2 + \ldots + D_n]$ on the other hand is 'flagged' to indicate which domain it comes from. To make this clear we define

$$[D_1 + D_2 + \ldots + D_n] = \{(d_i, i) \mid d_i \varepsilon D_i, 0<i<n+1\}$$

so that if $d \varepsilon D_i$ and $d \varepsilon D_j$ then (d, i) represents d considered as a member of D_i and (d, j) represents d considered as a member of D_j.

If $D = [D_1 + D_2 + \ldots + D_n]$ and $d \varepsilon D$ then $\mathbf{isD}_i\ d$ is **true** if d corresponds to a member of D_i and **false** otherwise. If $\mathbf{isD}_i\ d = \mathbf{true}$ then $\mathbf{outD}_i\ d$ is the

member of D_i corresponding to d. If $d_i \varepsilon D_i$ then $\mathbf{inD}_i\ d_i$ is the member of D corresponding to d_i. In summary:

$[D_1 + D_2 + \ldots + D_n] = \{(d_i, i) \mid d_i \varepsilon D_i, 0<i<n+1\}$

$$\mathbf{isD}_i\ d = \begin{cases} \mathbf{true} & \text{if } d \text{ is of the form } (d_i, i). \\ \mathbf{false} & \text{otherwise.} \end{cases}$$

$$\mathbf{outD}_i\ d = \begin{cases} d_i & \text{if } d \text{ is of the form } (d_i, i) \\ \text{'undefined'} & \text{otherwise} \end{cases}$$

$\mathbf{inD}_i\ d_i = (d_i, i)$

Comments (i) $\mathbf{isD}_i$, $\mathbf{outD}_i$ and $\mathbf{inD}_i$ can be thought of as functions with types:

$$\mathbf{isD}_i:[D_1 + D_2 + \ldots + D_n] \to \mathbf{Bool}$$
$$\mathbf{outD}_i:[D_1 + D_2 + \ldots + D_n] \to D_i$$
$$\mathbf{inD}_i:D_i \to [D_1 + D_2 + \ldots + D_n]$$

(ii) If two summands of a sum have the same name then this notation breaks down. For example if $D = \mathbf{Num} + \mathbf{Num} + \mathbf{Bool}$ then $\mathbf{isNum}\ d$ is not well defined. We shall always have summands with distinct names—if one wants to avoid this restriction then a different notation must be used (for example **is1**, **is2**, **is3** etc as tests for the first, second and third summands).

(iii) To avoid having to clutter up expressions with $\mathbf{inD}_i$ and $\mathbf{outD}_i$ we shall usually allow context to determine whether an element is in D_i or in D where $D = [D_1 + D_2 + \ldots + D_n]$. We thus adopt the following conventions which are analogous to the 'coercions' of programming languages.

(a) If $d_i \varepsilon D_i$ occurs in a context requiring a member of $D = [D_1 + D_2 + \ldots + D_n]$ then we interpret the occurrence of d_i as $\mathbf{inD}_i\ d_i$. For example if $n \varepsilon \mathbf{Num}$, $\mathbf{Value} = \mathbf{Num} + \mathbf{Bool}$ and f: $\mathbf{Value} \to D$ (some D) then "**f n**" really means "**f (inNum n)**".

(b) If $d\varepsilon[D_1 + D_2 + \ldots + D_n]$ occurs in a context requiring a member of D_i, then we interpret the occurrence of **d** as $\mathbf{out}D_i\ \mathbf{d}$. For example if
f: D→[Num + Bool] and **g: Bool→D*** then "**g(f d)**" really means "**g(outBool (f d))**"

(iv) The domain D^* of finite sequences can be thought of as an infinite sum

$$D^* = [( )] + D + [D \times D] + [D \times D \times D] + \ldots$$

3.3.4. Domain equations

We discussed in 3.2.2. why we need to be able to define domains by recursive equations. In general such equations have the form

$$\begin{aligned} D_1 &= T_1[D_1, \ldots, D_n] \\ D_2 &= T_2[D_1, \ldots, D_n] \\ &\vdots \\ D_n &= T_n[D_1, \ldots, D_n] \end{aligned}$$

Where each $T_i[D_1, \ldots, D_n]$ is some expression built out of $D_1, \ldots, D_n$ and finite or standard domains using the domain constructors $\rightarrow, \times, *, +$ described above. The example we met in 3.3.2. was

State = **Memory** × **Input** × **Output**
Memory = **Ide→[Value + {unbound}]**
Input = **Value***
Output = **Value***
Value = **Num + Bool + Proc**
Proc = **State→[State + {error}]**

An example of a single domain equation is the domain D^∞ of infinite sequences of members of **D** defined by:

$$D^\infty = [D \times D^\infty]$$

If $s\varepsilon D^\infty$ then $s = (d_1, s_1)\varepsilon[D \times D^\infty]$ where $s_1 = (d_2, s_2)\varepsilon[D \times D^\infty]$ where $s_2 = (d_3, s_3)\varepsilon[D \times D^\infty]$ ... i.e. $s = (d_1, (d_2, (d_3, \ldots)))$.

3.4. Functions

The word "function" has a number of different meanings which, if not carefully distinguished, can lead to total confusion. It is especially important to distinguish mathematical functions from the so-called "functions" which occur as constructs in several programming languages (for example PASCAL, LISP and POP-2).

(i) A mathematical function **f** with *source* **A** and *target* **B** (which we write **f: A→B**) is a set of pairs $(a, b)\varepsilon A \times B$ such that if $(a, b_1)\varepsilon f$ and $(a, b_2)\varepsilon f$ then $b_1 = b_2$. If $(a, b)\varepsilon f$ we write $fa = b$ and thus $f = \{(a, fa) \mid a\varepsilon A\}$. In the notation **f: A→B** the expression "**A→B**" is called the *type* of **f** and denotes the set of all functions from **A** to **B**.

(ii) A 'function' in a programming language like PASCAL is a *construct* which describes a *rule* for transforming an argument—the actual parameter—to a result.

Now if **F** is a function construct (as in (ii)) then one can associate with it a mathematical function **f**: (actual parameters)→(results) defined by

fx = result of calling **F** on parameter **x**.

Indeed a simple denotational semantics might give **f** as the denotation of **F** (i.e. **F[F] = f**). However note that:

(a) Many *different* functions constructs can denote the same mathematical function (for example different algorithms for sorting a list).

(b) A function construct is a *finite* object whereas a mathematical function is typically an *infinite* set—for example the factorial function is the infinite set

$$\{(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), \ldots\}$$

(c) As we shall see later, except in simple cases, it will not be satisfactory to take the denotation of a function construct to be the mathematical function defined by its input/output behaviour. To handle side-effects, jumps out of the functions body etc. we will need more complicated denotations.

Unless the contrary is clear from context the word "function" in this book means "mathematical function".

We shall use two main techniques for defining functions:

(i) λ-notation (see 3.4.1.).
(ii) Definition by recursion (see 3.4.9.).

3.4.1. λ-notation

3.4.1.1. Basic Idea

Suppose E[x] is some expression involving **x** such that whenever **d**ε**D** is substituted for **x**—and we shall denote the result of such a substitution by E[**d**]—the resulting expression (namely E[**d**]) denotes a member of **D'**. For example if both **D** and **D'** are **Num** then E[**x**] could be **x+1** (so E[5]=5+1=6) or perhaps **x×x** (then E[5]=5×5=25). For such expressions the notation:

λx . E[x]

denotes the function **f: D→D'** such that:

for all **d**ε**D**: **fd** = E[**d**]

For example:

(i) **λx . x+1** denotes the successor function of type **Num→Num**.
(ii) **λx . x×x** denotes the squaring function of type **Num→Num**.
(iii) **λx . (x=0)→true, false** denotes the test-for-zero functions of type **Num→Bool**.

An expression of the form **λx . E[x]** is called a λ-*expression*, **x** is its *bound variable* and E[**x**] its body.

This is the central idea of λ-notation; the various elaborations described below are just to make the descriptions of complicated functions a bit more intelligible.

N.B. The body of a λ-expression always extends as far to the right as possible, thus **λx . x+1** is **λx . (x+1)** not **(λx . x)+1**.

3.4.1.2. Elaborations

3.4.1.2.1. Explicitly indicating source and/or target

Sometimes it is not clear what the source (**D** say) or target (**D'** say) of a function defined by a λ-expression are—for example consider **λx . x**—in such cases one can, if necessary, indicate the desired information by writing **:D→D'** after the λ-expression. For example **λx . x: Num→Num**, **λx . x: Bool→Bool**, **λx . x+1: Num→Num** etc.

Often the target but not the source is clear; in such cases one can write **:D** after the bound variable. For example **λx:Num . x**, **λx:Bool . x**, **λx:Num . x+1** etc.

3.4.1.2.2. More than one argument

If E[x_1, ..., x_n] is an expression having a value in **D'** when $x_1 \varepsilon D_1$, ..., $x_n \varepsilon D_n$ then the notation

λ(x_1, ..., x_n) . E[x_1, ..., x_n]

is used to describe the function **f: [D_1 × ... × D_n]→D'** such that:

for all $d_1 \varepsilon D_1$, ..., $d_n \varepsilon D_n$: **f(d_1, ..., d_n)** = E[d_1, ..., d_n]

Examples are:

(i) **λ(x_1,x_2) . x_1 + x_2** the addition function if type

[Num × Num]→Num

(ii) **λ(x_1,x_2) . x_1<x_2→error,x_1 − x_2**: the subtraction function of type

[Num × Num]→[Num + {error}]

This notation for more than one argument can be combined with explicit indication of source on target, for example

λ(x_1, x_2) . x_1 + x_2: [[Num × Num]→Num]
or **λ(x_1, x_2): [Num × Num] . x_1 + x_2**

Functions like this can either be thought of as having many arguments or, perhaps more elegantly, as having just one argument which is a tuple.

3.4.1.3. Applying λ-expressions to arguments

Just as we can form expressions like "**f1**" to denote the application of **f** to **1** so we can from expressions in which λ-expressions are applied to arguments, for example:

(λx . x+1) 2 = 2+1 = 3
(λ(x,y) . x+y) (2,3) = 2+3 = 5

When 'evaluating' **(λx . E[x])a** to E[**a**] one must only substitute **a** for those

occurrences of **x** in E[x] which are *not* bound by inner λ's. For example (λx . (λx . x))a evaluates to λx . x not λx . a.

3.4.1.4. Changing bound variables

The meaning of λ-expressions does not depend on the names of its bound variables—λx . x + 1, λn . n + 1, λm . m + 1 all denote the same function. In general bound variables can be renamed as long as the new name does not occur elsewhere in the λ-expression, if the new name does occur elsewhere then 'variable capture' may result and the meaning of the λ-expression change. An example of variable capture occurs if we rename **x** in λx . (λy . x) to **y** to get λy . (λy . y)—here the **x** is initially bound by the outer λ but after being renamed to **y** gets 'captured' by the inner λ and this capturing changes the meaning, thus:

$$((\lambda x . (\lambda y . x))1)\,2 = (\lambda y . 1)\,2 = 1$$
$$((\lambda y . (\lambda y . y))1)\,2 = (\lambda y . y)\,2 = 2$$

It is quite tricky (but not really difficult) to spell out exactly the general conditions under which variables get captured (see [Stoy]) but fortunately it is usually obvious in particular cases.

3.4.2. Higher order functions

The example in the previous section—λx . (λy . x)—is an example of a λ-expression which denotes a *higher order* function. Higher order functions are functions whose source or target contains functions. Thus λx . (λy . x) has a type of the form $D_1 \to [D_2 \to D_1]$ and so the result of applying λx . (λy . x) to an argument, d_1 say, is a member of $[D_2 \to D_1]$ (λy . d_1 to be exact) i.e. a function.

Another example of a higher order function is

twice: [[Num→Num]→[Num→Num]]

defined by **twice f** = λx . f(fx) (i.e. **twice** = λf . (λx . f(fx))). **twice f** is a function which does **f** twice, for example **twice** (λn . n + 1) is a function which applies λn . n + 1 twice—i.e. adds **1** twice—i.e. adds **2**. Thus **twice** (λn . n + 1) = λx . x + 2, this can be verified formally as follows:

$$\textbf{twice}\,(\lambda n . n+1) = \lambda x . (\lambda n . n+1)\,((\lambda n . n+1)\,x)$$
$$= \lambda x . (\lambda n . n+1)\,(x+1)$$
$$= \lambda x . (x+1)+1$$
$$= \lambda x . x+2$$

Higher order functions are very useful and we shall frequently use them; for example the semantic functions for TINY were higher order—**C: Com→[State→[State + {error}]]** so **C[E]** (the result of applying **C** to **E**) is a function. Higher order functions are perfectly ordinary and everything we say about functions in general applies to them also. The reason why we singled them out for discussion is because in many programming languages (for example ALGOL 60 and PASCAL) 'functions' (as opposed to mathematical functions) are subject to various constraints. For example in neither ALGOL 60 nor PASCAL can 'functions' return 'functions' as results (so twice could not be programmed). These constraints are to enable efficient implementations of the languages to be possible and do not reflect anything inherent in the concept of a function. Indeed other (less efficient) languages like LISP or POP-2 allow the unrestricted programming of higher order 'functions'. As discussed earlier (at the beginning of 3.4.) one must be careful not to confuse the mathematical concept of a function with the various 'functions' occurring in programming languages.

3.4.3. Important notational conventions on precedence and association

(i) $f\,x_1 x_2 \ldots x_n$ means $(\ldots((f\,x_1)x_2)\ldots)$. For example **f g x** means **(f g)x** *not* **f(g x)**. Thus application associates to the left.

(ii) $f{:}x_1{:}x_2{:}\ldots{:}x_n$ means $f(x_1(x_2(\ldots(x_{n-1}\ x_n)\ldots)))$. For example **f:g:x** means **f(g x)** *not* **(f g)x**. Thus ";" denotes application but unlike juxtaposition associates to the right.

(iii) $f_1\,x_{11} \ldots x_{1n};f_2\,x_{21} \ldots x_{2n};\ldots;f_n\,x_{n1} \ldots x_{nn}$ means $(f_1 x_{11} \ldots x_{1n});(f_2 x_{21} \ldots x_{2n});\ldots;(f_n x_{n1} \ldots x_{nn})$. For example **f g:x** means **(f g):x** i.e. **(f g) x** and **f;g x** means **f:(g x)** i.e. **f(g x)**. Thus ";" binds weaker than ordinary application (as denoted by juxtaposition).

(iv) $\lambda x_1 x_2 \ldots x_n . E[x_1, x_2, \ldots, x_n]$ means $\lambda x_1 . (\lambda x_2 . \ldots (\lambda x_n . E[x_1, x_2, \ldots, x_n]) \ldots)$. This notation can be extended to functions which take tuples as arguments, thus for example $\lambda x_1 (x_2, x_3) x_4 . E[x_1, x_2, x_3, x_4]$ means $\lambda x_1 . (\lambda (x_2, x_3) . (\lambda x_4 . E[x_1, x_2, x_3, x_4]))$.

(v) Conventions (i) and (iv) cooperate nicely together, for example

$$(\lambda x_1 x_2 \ldots x_n . E[x_1, x_2, \ldots, x_n]) d_1 d_2 \ldots d_n = E[d_1, d_2, \ldots, d_n]$$

(vi) $D_1 \to D_2 \to \ldots \to D_n$ means $[D_1 \to [D_2 \to [\ldots [D_{n-1} \to D_n] \ldots]]]$ (see 3.3.3.3.).

(vii) The body of a λ-expressions always extends as far to the right as possible, for example $\lambda x . f\ x\ y\ z$ means $\lambda x . (f\ x\ y\ z)$ not $(\lambda x . f)\ x\ y\ z$.

3.4.4. Currying

Consider the two functions **plus** and **plusc** defined below:

(i) **plus: [Num × Num] → Num** , **plus** $= \lambda (n, m) . n + m$
(ii) **plusc: Num → Num → Num** , **plusc** $= \lambda n m . n + m$

These functions are related in that **plus (n, m) = plusc n m**. It will often be convenient to use higher order functions like **plusc** which take their arguments 'one at a time' instead of **plus** which takes a pair (or more generally an n-tuple) as an argument. The advantage of **plusc** is that it can be applied to just one argument. For example **plusc 1** is a well formed expression denoting the function $\lambda m . 1 + m$, **plusc 2** denotes $\lambda m . 2 + m$ etc. – **plus 1**, **plus 2** do not make sense as **1**, **2** are not in the source **[Num × Num]** of **plus**. Another example is the semantic functions of TINY; because

$$\mathbf{C}: \mathbf{Com} \to \mathbf{State} \to [\mathbf{State} + \{\mathbf{error}\}]$$

We can use $\mathbf{C}[C]$ to denote the denotation of **C**; if $\mathbf{C}$ had type **[Com × State] → [State + {error}]** this expression would not make sense and we would have to use $\lambda s . \mathbf{C}[C, s]$.

Functions of more than one argument which take them 'one at a time' like **plusc** and $\mathbf{C}$ are called *curried* (after Mr. Curry). In general if **f**: $[D_1 \times D_2 \times \ldots \times D_n] \to D$ then there is an equivalent curried function, **curry f**, say of type $D_1 \to D_2 \to \ldots \to D_n \to D$ defined by

$$\mathbf{curry}\ f = \lambda x_1 x_2 \ldots x_n . f(x_1, x_2, \ldots, x_n)$$

curry itself is a higher order function defined by:

$$\mathbf{curry}: [[D_1 \times D_2 \times \ldots \times D_n] \to D] \to [D_1 \to D_2 \to \ldots \to D_n \to D]$$
$$\mathbf{curry} = \lambda f\ x_1 x_2 \ldots x_n . f(x_1, x_2, \ldots, x_n)$$

For example: **curry plus** $= (\lambda f\ x_1 x_2 . f(x_1, x_2))$ **plus**
$= (\lambda f . (\lambda x_1 x_2 . f(x_1, x_2)))$ **plus**
$= \lambda x_1 x_2 .$ **plus** (x_1, x_2)
$= \lambda x_1 x_2 . x_1 + x_2$
$=$ **plusc**

There is an inverse to **curry** called **uncurry** defined by:

$$\mathbf{uncurry}: [D_1 \to D_2 \to \ldots \to D_n \to D] \to [[D_1 \times D_2 \times \ldots \times D_n] \to D]$$
$$\mathbf{uncurry} = \lambda f (x_1, x_2, \ldots, x_n) . f\ x_1 x_2 \ldots x_n$$

The reader might like to show that **uncurry plusc = plus** and that for all **f**, **f'** of the appropriate types **curry (uncurry f) = f** and **uncurry (curry f') = f'**.

3.4.5. Conditionals

A very important standard function is the conditional **cond** defined by:

$$\mathbf{cond}: [D \times D] \to \mathbf{Bool} \to D \qquad (D \text{ arbitrary})$$

$$\mathbf{cond}\ (d_1, d_2)\ b = \begin{cases} d_1 & \text{if } b = \mathbf{true} \\ d_2 & \text{if } b = \mathbf{false} \end{cases}$$

We use the following notations:

(i) $b \to d_1, d_2$ means **cond** $(d_1, d_2)\ b$

(ii) $b_1 \to d_1, b_2 \to d_2, \ldots, b_n \to d_n, d_{n+1}$ means $b_1 \to d_1, (b_2 \to d_2, (\ldots (b_n \to d_n, d_{n+1}) \ldots))$. If $b_1, \ldots, b_n$ are mutually exclusive we may omit d_{n+1} and write $b_1 \to d_1, \ldots, b_n \to d_n$.

Strictly speaking there is a conditional function, **condD** say, for each domain **D**. Thus **CondNum: [Num × Num] → Bool → Num** is different from **condIde: [Ide × Ide] → Bool → Ide** etc. In practice when we write $b \to d_1, d_2$ it should be clear from context which domain is intended.

An important property of conditionals is: $(b \to f, g)\ x = b \to fx, gx$ here

f, g must have types of the form $D_1 \to D_2$ so the left hand conditional has type $[[D_1 \to D_2] \times [D_1 \to D_2]] \to \mathbf{Bool} \to [D_1 \to D_2]$ and the right hand conditional has type $[D_2 \times D_2] \to \mathbf{Bool} \to D_2$ using **cond** instead of → we have: **cond** $[D_1 \to D_2]$ **(f,g) b x = cond** D_2 **(f x, g x) b.**

3.4.6. Cases notation

We shall illustrate cases notation with a couple of examples and hope the reader gets the general idea from these.

Suppose $D = [\mathbf{Value} \times \mathbf{State}] + \{\mathbf{error}\}$ and $d \varepsilon D$ then the expression:

$(d = (v, s)) \to E_1[v, s],$
$(d = \mathbf{error}) \to E_2$

means if **d** corresponds to a member **(v,s)** of **[Value × State]** then $E_1[v,s]$ else if **d = error** then E_2

As another example suppose $D = D_1 + D_2 + D_3$
$D_1 = D_{11} \times D_{12}$
$D_2 = \{d_2\}$
$D_3 = D_{31} \times D_{32} \times D_{33}$

then the expression:

$(d = (d_{11}, d_{12})) \to E_1[d_{11}, d_{12}],$
$(d = d_2) \to E_2,$
$(d = (d_{31}, d_{32}, d_{33})) \to E_3[d_{31}, d_{32}, d_{33}], E_4$

means if **d** corresponds to $(d_{11}, d_{12}) \varepsilon D_1$ then $E_1[d_{11}, d_{12}]$, if **d** corresponds to $d_2 \varepsilon D_2$ then E_2, if **d** corresponds to $(d_{31}, d_{32}, d_{33}) \varepsilon D_3$ then $E_3[d_{31}, d_{32}, d_{33}]$ else E_4. This second cases statement is equivalent to

$\mathbf{isD_1}\, d \to (\lambda[d_{11}, d_{12}] . E_1[d_{11}, d_{12}])\ (\mathbf{outD_1}\ d),$
$\mathbf{isD_2}\, d \to E_2,$
$\mathbf{isD_3}\, d \to (\lambda[d_{31}, d_{32}, d_{33}] . E_3[d_{31}, d_{32}, d_{33}])\ (\mathbf{outD_3}\ d), E_4$

This shows that cases notation both tests values and binds their components to variables. Examples of the use of cases notation are the semantic clauses for TINY above.

3.4.7. Updating functions

If $f: D \to D'$, $d_1, \ldots, d_n \varepsilon D$ and $d_1', \ldots, d_n' \varepsilon D'$ then $f[d_1', \ldots, d_n'/d_1, \ldots, d_n]$ denotes the function identical to **f** except at $d_1, \ldots, d_n$ where it has values $d_1', \ldots, d_n'$ respectively. Thus:

$$\mathbf{f}[d_1', \ldots, d_n'/d_1, \ldots, d_n] = \lambda d .\ d = d_1 \to d_1',$$
$$d = d_2 \to d_2',$$
$$\vdots$$
$$d = d_n \to d_n', \mathbf{f}d$$

3.4.8. Generic functions

Functions like **cond** which, strictly speaking, are collections of functions, one for each domain of an appropriate type, are called *generic*. Other examples are **curry** and **uncurry**. The actual functions which make up the collections of functions are called *instances*, thus **condNum**, **condIde** are instances of **cond** etc. The expressions which describe the types of the instances of generic functions are called *generic types*. For example $[D \times D] \to \mathbf{Bool} \to D$ is the generic type of **cond**; an instance of this type with **D = Num** is $[\mathbf{Num} \times \mathbf{Num}] \to \mathbf{Bool} \to \mathbf{Num}$. The generic type of **curry** is $[[D_1 \times D_2 \times \ldots \times D_n] \to D] \to [D_1 \to D_2 \to \ldots \to D_n \to D]$, in **curry plus** (see 3.4.4) it is used at instance
$[[\mathbf{Num} \times \mathbf{Num}] \to \mathbf{Num}] \to [\mathbf{Num} \to \mathbf{Num} \to \mathbf{Num}]$.

3.4.9. Ways of defining functions (including recursion)

A typical definition of a function **f** has the form

$\mathbf{f}(x_1, \ldots, x_n) = E[x_1, \ldots, x_n]$
or $\mathbf{f}\, x_1 \ldots x_n = E[x_1, \ldots, x_n]$ if **f** is curried

For example:

plus (x, y) = x + y
plusc x y = x + y

An equivalent (and sometimes more convenient) way of writing such definitions is as

$$\mathbf{f} = \lambda(\mathbf{x}_1, \ldots, \mathbf{x}_n) \,.\, \mathbf{E}[\mathbf{x}_1, \ldots, \mathbf{x}_n]$$
$$\text{or } \mathbf{f} = \lambda \mathbf{x}_1 \ldots \mathbf{x}_n \,.\, \mathbf{E}[\mathbf{x}_1, \ldots, \mathbf{x}_n]$$

For example

$$\mathbf{plus} = \lambda(\mathbf{x}, \mathbf{y}) \,.\, \mathbf{x} + \mathbf{y}$$
$$\mathbf{plusc} = \lambda \mathbf{x}\mathbf{y} \,.\, \mathbf{x} + \mathbf{y}$$

We shall often need to define functions *recursively*, for example we might define **fact: Num→Num** thus:

$$\mathbf{fact}\ \mathbf{n} = (\mathbf{n} = \mathbf{0}) \rightarrow \mathbf{1},\ \mathbf{n} \times \mathbf{fact}\ (\mathbf{n}\text{-}\mathbf{1})$$
$$\text{or } \mathbf{fact} = \lambda \mathbf{n} \,.\, (\mathbf{n} = \mathbf{0}) \rightarrow \mathbf{1},\ \mathbf{n} \times \mathbf{fact}\ (\mathbf{n}\text{-}\mathbf{1})$$

We also sometimes need to define several functions by *mutual recursion*. If we use λ-expressions (as in the second definition of **fact** above) then the general form of mutually recursive definitions is:

$$\mathbf{f}_1 = \mathbf{E}_1[\mathbf{f}_1, \ldots, \mathbf{f}_n]$$
$$\mathbf{f}_2 = \mathbf{E}_2[\mathbf{f}_1, \ldots, \mathbf{f}_n]$$
$$\vdots$$
$$\mathbf{f}_n = \mathbf{E}_n[\mathbf{f}_1, \ldots, \mathbf{f}_n]$$

Where the $\mathbf{E}_i[\mathbf{f}_1, \ldots, \mathbf{f}_n]$ are λ-expressions. For example if $\mathbf{E}[\mathbf{f}]$ is $\lambda \mathbf{n} \,.\, (\mathbf{n} = \mathbf{0}) \rightarrow \mathbf{1},\ \mathbf{n} \times \mathbf{f}(\mathbf{n}\text{-}\mathbf{1})$ then the second definition of **fact** above is: $\mathbf{fact} = \mathbf{E}[\mathbf{fact}]$. We thus see **fact** is a 'fixed point' of the function $\lambda \mathbf{f} \,.\, \mathbf{E}[\mathbf{f}]$ – i.e. $(\lambda \mathbf{f} \,.\, \mathbf{E}[\mathbf{f}])\ \mathbf{fact} = \mathbf{fact}$. $\lambda \mathbf{f} \,.\, \mathbf{E}[\mathbf{f}]$ has type **[Num→Num]→[Num→Num]** and its 'fixed point' **fact** has type **[Num→Num]**. If we introduce a generic function **fix** of type **[D→D]→D** defined intuitively by:

$$\mathbf{fix}\ \mathbf{f} = \text{the fixed point of } \mathbf{f}$$

Then the recursive definition of **fact** can be written *non-recursively* as

$$\mathbf{fact} = \mathbf{fix}\ (\lambda \mathbf{f}\, \mathbf{n} \,.\, (\mathbf{n} = \mathbf{0}) \rightarrow \mathbf{1},\ \mathbf{n} \times \mathbf{f}\ (\mathbf{n}\text{-}\mathbf{1}))$$

here **fix** is used at the instance **D = [Num→Num]** of its generic type **[D→D]→D**.

The precise definition, and mathematical analysis, of the function **fix** constitute the "theory of fixed points", this was discussed in 3.2.1. We shall not use **fix** but will define things recursively instead.

3.4.10. Cancelling out variables

Consider the equation $\mathbf{fx} = \mathbf{gx}$ if this holds for all **x** then it follows that $\mathbf{f} = \mathbf{g}$. In general an equation like

$$\mathbf{f}\ \mathbf{x}_1 \ldots \mathbf{x}_i \ldots \mathbf{x}_n = (\mathbf{E}[\mathbf{x}_1, \ldots, \mathbf{x}_i])\ \mathbf{x}_{i+1} \ldots \mathbf{x}_n$$

can be simplified to

$$\mathbf{f}\ \mathbf{x}_1 \ldots \mathbf{x}_i = \mathbf{E}[\mathbf{x}_1, \ldots, \mathbf{x}_i]$$

as long as $\mathbf{x}_{i+1}, \ldots, \mathbf{x}_n$ do not occur in $\mathbf{E}[\mathbf{x}_1, \ldots, \mathbf{x}_i]$. A common example of such simplifications is with definitions like

$$\mathbf{f}\ \mathbf{x}\mathbf{y} = \mathbf{p}\mathbf{x} \rightarrow \mathbf{g}\mathbf{x}\mathbf{y},\ \mathbf{h}\mathbf{x}\mathbf{y}$$

using the fact that $\mathbf{px} \rightarrow \mathbf{gxy},\ \mathbf{hxy} = (\mathbf{px} \rightarrow \mathbf{gx},\ \mathbf{hx})\ \mathbf{y}$ we can reduce this equation to

$$\mathbf{f}\ \mathbf{x}\mathbf{y} = (\mathbf{px} \rightarrow \mathbf{gx},\ \mathbf{hx})\ \mathbf{y}$$

and then cancel **y** to get:

$$\mathbf{f}\ \mathbf{x} = \mathbf{p}\ \mathbf{x} \rightarrow \mathbf{g}\ \mathbf{x},\ \mathbf{h}\ \mathbf{x}$$

Now the definition:

$$\mathbf{f}\ \mathbf{x}_1 \ldots \mathbf{x}_n = (\mathbf{E}[\mathbf{x}_1, \ldots, \mathbf{x}_i])\mathbf{x}_{i+1} \ldots \mathbf{x}_n$$

is equivalent to

$$\mathbf{f} = \lambda \mathbf{x}_1 \ldots \mathbf{x}_n \,.\, (\mathbf{E}[\mathbf{x}_1, \ldots, \mathbf{x}_i])\ \mathbf{x}_{i+1} \ldots \mathbf{x}_n$$

and just as the first definition can be simplified to

$$\mathbf{f}\ \mathbf{x}_1 \ldots \mathbf{x}_i = \mathbf{E}[\mathbf{x}_1, \ldots, \mathbf{x}_i]$$

so the second one can be simplified to

$$\mathbf{f} = \lambda \mathbf{x}_1 \ldots \mathbf{x}_i \,.\, \mathbf{E}[\mathbf{x}_1, \ldots, \mathbf{x}_i]$$

In general $\lambda \mathbf{x}_1 \ldots \mathbf{x}_i \ldots \mathbf{x}_n \,.\, (\mathbf{E}[\mathbf{x}_1, \ldots, \mathbf{x}_i])\ \mathbf{x}_{i+1} \ldots \mathbf{x}_n$ can be simplified to $\lambda \mathbf{x}_1 \ldots \mathbf{x}_i \,.\, \mathbf{E}[\mathbf{x}_1, \ldots, \mathbf{x}_i]$ as long as $\mathbf{x}_{i+1}, \ldots, \mathbf{x}_n$ do not occur in $\mathbf{E}[\mathbf{x}_1, \ldots, \mathbf{x}_i]$. For example $\lambda \mathbf{xy} \,.\, \mathbf{px} \rightarrow \mathbf{gxy},\ \mathbf{hxy}$ can be simplified to $\lambda \mathbf{x} \,.\, \mathbf{px} \rightarrow \mathbf{gx},\ \mathbf{hx}$.

N.B. When using ";" (see 3.4.3.) one must be careful with cancelling. For example if **f x = g;h;x** then it *does not* follow that **f = g;h**. Similarly **λx . g;h;x** *does not* simplify to **g;h**.

3.4.11. **where** notation

Sometimes it is convenient to 'structure' expressions by writing

$$E[x_1, \ldots, x_n] \text{ where } x_1 = E_1 \text{ and } x_2 = E_2 \text{ and } \vdots \text{ and } x_n = E_n$$

instead of $E[E_1, \ldots, E_n]$. For example

$$x_1 \times x_2 \text{ where } x_1 = 2+3 \text{ and } x_2 = 6\text{-}7$$

instead of **(2 + 3) × (6-7)**. Use of **where** can sometimes shorten expressions and highlights essential aspects of their structure, for example

$$C[C_1; C_2]s = (s' = \text{error}) \rightarrow \text{error}, C[C_2]s' \text{ where } s' = C[C_1]s$$

One can also use **where** for 'subsidiary' functions, for example

$$(f1) + (f2) \text{ where } fx = x \times x$$

this means $((\lambda x . x \times x)1) + ((\lambda x . x \times x)2) = 5$

If the subsidiary function is recursive we draw attention to that fact by using "**whererec**" instead of "**where**". For example

$$\text{fact } 6 \text{ whererec fact } n = (n = 0) \rightarrow 1, n \times \text{fact } (n-1)$$

If several subsidiary functions are to be mutually recursive then we write

$$E[f_1, \ldots, f_n] \text{ whererec } f_1 = E_1[f_1, \ldots, f_n] \vdots \text{ and } f_n = E_n[f_1, \ldots, f_n]$$

(here we assume the subsidiary functions are being defined as λ-expressions but this is not necessary).

3.4.12. Composition and sequencing

3.4.12.1 Composition

If $f: D_1 \rightarrow D_2$ and $g: D_2 \rightarrow D_3$ then their *composition* $g \circ f: D_1 \rightarrow D_3$ is defined by $g \circ f = \lambda x . g(f x)$. Notice that in $g \circ f$, **f** is 'done first'.

3.4.12.2. Sequencing

We are going to define **f ★ g** where **f** and **g** have a variety of types, before exact details here is the essential idea: **f ★ g** is the function corresponding to first doing **f**, if **f** produces an error then **error** is the result of **f ★ g** otherwise the results of **f** are 'fed to' **g**. **f ★ g** is like **g∘f** except that

(i) Errors are propagated.
(ii) **g** may be curried.

Here now are the cases we shall need:

(a) Suppose $f: D_1 \rightarrow [D_2 + \{\text{error}\}]$ and $g: D_2 \rightarrow [D_3 + \{\text{error}\}]$ then $f \star g: D_1 \rightarrow [D_3 + \{\text{error}\}]$ is defined by:

$$f \star g = \lambda x . fx = \text{error} \rightarrow \text{error}, g(fx)$$

(b) Suppose $f: D_1 \rightarrow [[D_2 \times D_3] + \{\text{error}\}]$ and $g: D_2 \rightarrow D_3 \rightarrow [D_4 + \{\text{error}\}]$ then $f \star g: D_1 \rightarrow [D_4 + \{\text{error}\}]$ is defined by:

$$f \star g = \lambda x . (fx = \text{error}) \rightarrow \text{error}, (fx = (d_2, d_3)) \rightarrow g d_2 d_3$$

To see the use of ★ observe that the semantic clause:

$$C[C_1; C_2]s = (C[C_1]s = \text{error}) \rightarrow \text{error}, C[C_2](C[C_1]s)$$

can be simplified to

$$C[C_1; C_2] = C[C_1] \star C[C_2]$$

Here we have an example of case (a) with $D_1 = D_2 = D_3 = \text{State}$. As another example the semantic clause:

$$C[\text{output } E]s = (E[E]s = \text{error}) \rightarrow \text{error}, (E[E] = (v,(m,i,o))) \rightarrow (m, i, v . o)$$

can be simplified to

$$\mathbf{C[output\ E]} = \mathbf{E[E]} \star \lambda v(m,i,o).(m,i,v.o)$$

Here we have case (b) with $D_1 = D_3 = \mathbf{State}$, $D_2 = \mathbf{Value}$.

As a final example (again of case (b) with $D_1 = D_3 = \mathbf{State}$, $D_2 = \mathbf{Value}$) the semantic clause (E6) in 2.4.4.2. can be written:

$$\mathbf{E[E_1 = E_2]} = \mathbf{E[E_1]} \star \lambda v_1 . \mathbf{E[E_2]} \star \lambda v_2\, s.(v_1 = v_2, s)$$

The reader is urged to ensure he follows these examples by expanding the $\star$'s. We shall also use $\mathbf{f \star g}$ when **f** cannot produce **error**. Case (a) then corresponds to $\mathbf{f \star g = g \circ f}$, case (b) to $\mathbf{f \star g = g \circ (uncurry\ f)}$.

Finally note that:

(i) $\star$ is associative so expressions like $\mathbf{f \star g \star h}$ are unambiguous (since $\mathbf{f \star (g \star h) = (f \star g) \star h}$).

(ii) Expressions like $\mathbf{f \star g \star \lambda x . h \star k}$ mean $\mathbf{f \star g \star (\lambda x . (h \star k))}$—see 3.4.3. (vii).

4. Denotational description of TINY

The purpose of the battery of abbreviatory conventions described in the preceding chapter is to provide a set of notations which are both powerful and concise. We would like the expressions and definitions we write—semantic clauses for example—to be both brief and readable. Unfortunately there is some conflict between these two requirements—compact notations are often hard to understand, but, conversely, long descriptions often hide the really important facts in a confusing morass of minor detail. Exactly how to trade off compactness versus explicit detail is still a matter of controversy and the reader must decide for himself whether powerful abbreviations (like the sequencing operator $\star$ of 3.4.12.2. for example) are a help or hinderance to understanding.

In the rest of this chapter we describe TINY using the notations and concepts presented in chapter 3. We give the new version of the semantics without comment but indicating previous sections which explain various notations when they are first used. The new semantics is completely equivalent to the one given in chapter 2 and so the explanations there apply here also.

4.1. Abstract syntax (see 3.1.)

4.1.1. Syntactic domains

E ranges over the domain **Exp** of expressions
C ranges over the domain **Com** of commands

4.1.2. Syntactic clauses

$\mathbf{E} ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{read} \mid \mathbf{I} \mid \mathbf{not\ E} \mid \mathbf{E_1 = E_2} \mid \mathbf{E_1 + E_2}$
$\mathbf{C} ::= \mathbf{I := E} \mid \mathbf{output\ E} \mid \mathbf{if\ E\ then\ C_1\ else\ C_2} \mid \mathbf{while\ E\ do\ C} \mid \mathbf{C_1; C_2}$

4.2. Semantics

4.2.1. Semantic domains (see 3.3.)

State = Memory × Input × Output
Memory = Ide→[Value + {unbound}]
Input = Value*
Output = Value*
Value = Num + Bool

4.2.2. Auxiliary functions (see 3.4.)

These functions are useful for 'factoring out' common parts of the semantic clauses in 4.2.4.

4.2.2.1. result

Informal description: **result v** is the denotation of an expression which returns **v** as value and does not change the state.

Formal description: result:Value→State→[[Value × State] + {error}]
result = λv s . (v,s)
(see (iii) (a) in the comments at the end of 3.3.3.4.).

4.2.2.2. donothing

Informal description: **donothing** is the denotation of a command which has no effect.

Formal description: donothing:State→[State + {error}]
donothing = λs . s

4.2.2.3. checkNum

Informal description: **checkNum v** is the denotation of an expression which returns **v** and an unchanged state if v∈Num and **error** otherwise.

Formal description:
checkNum:Value→State→[[Value × State] + {error}]
checkNum = λv s . isNum v→(v,s), error
(see 3.3.3.4.).

4.2.2.4. checkBool

Informal description: **checkBool v** is the denotation of an expression which returns **v** and an unchanged state if v∈Bool and **error** otherwise.

Formal description:
checkBool:Value→State→[[Value × State] + {error}]
checkBool = λv s . isBool v→(v,s), error

4.2.3. Semantic functions

E:Exp→State→[[Value × State] + {error}]
C:Com→State→[State + {error}]

4.2.4. Semantic clauses

4.2.4.1. Clauses for expressions

(E1) E[0] = result 0, E[1] = result 1
(E2) E[true] = result true, E[false] = result false
(E3) E[read] =
λ(m,i,o) . null i→error, (hdi, (m, tli, o)) (see 3.3.3.3.)
(E4) E[I] =
λ(m,i,o) . m I = unbound→error, (m I, (m,i,o))
(E5) E[not E] =
E[E] ⋆ checkBool ⋆ λv . result (not v) (see 3.4.12.2.)
(E6) $E[E_1 = E_2]$ =
$E[E_1] \star \lambda v_1 . E[E_2] \star \lambda v_2 .$ result $(v_1 = v_2)$
(E7) $E[E_1 + E_2]$ =
$E[E_1] \star$ checkNum $\star \lambda v_1 . E[E_2] \star$ checkNum $\star \lambda v_2 .$
result $(v_1 + v_2)$

4.2.4.2. Clauses for commands

(C1) C[I:=E] = E[E] ⋆ λv (m, I, o) . (m[v/I], I, o)
(C2) C[output E] = E[E] ⋆ λv (m, I, o) . (m, I, v . o)
(C3) $C[\text{if E then } C_1 \text{ else } C_2]$ =
E[E] ⋆ checkBool ⋆ cond (C[C₁], C[C₂]) (see 3.4.5.)
(C4) C[while E do C] =
E[E] ⋆ checkBool ⋆ cond (C[C] ⋆ C[while E do C], donothing)
(C5) $C[C_1;C_2] = C[C_1] \star C[C_2]$

5. Standard semantics

The description of TINY given in the last chapter was designed to illustrate the main ideas and notations of denotational semantics. For real languages (such as PASCAL or ALGOL 60) it is necessary to use rather more sophisticated denotations, in particular:

(i) Instead of denotations transforming states 'directly' it is necessary for them to transform states indirectly via *continuations* (see 5.1. below). This enables jumps to be handled.

(ii) The binding of identifiers to values needs to be split into two parts: a mapping from identifiers to 'variables' and a mapping from 'variables' to values. This enables sharing or aliasing to be handled.

A semantics based on (i) and (ii) is called a *standard semantics*. If we describe languages using fixed standard techniques then comparisons between languages are made easier. The disadvantage is that for any particular language the 'fit' of the techniques may not be perfect. For example when we come to discuss the 'dynamic binding' of LISP the two-stage binding of identifiers to values of (ii) is not completely appropriate—however the distortion it involves is compensated for by being able to discuss very different languages (for example PASCAL and LISP) within a single framework.

In this chapter we describe the central ideas and techniques of standard semantics. In the next chapter we illustrate these by giving a standard semantics of a slight extension of TINY called SMALL. Then in the rest of the book we apply standard techniques to describe many different kinds of constructs (including most of those occurring in PASCAL and ALGOL 60).

5.1. Continuations

The development of *continuations* [Strachey and Wadsworth] was an important advance in the descriptive techniques of semantics. It led to:

(i) Simplification—both conceptual and notational—in the description of most constructs.

(ii) Smooth descriptions of various constructs which, previously, were

impossible to handle (for example jumps back into already exited procedures, co-routines and 'state saving').

5.1.1. Modelling the 'rest of the program'

Most constructs can be thought of as transforming some input into some result— the exact type of these depending on the construct. In TINY the inputs to both commands and expressions are states and the results are states (or **error**) is the case of commands and value/state pairs (or **error**) is the case of expressions.

In the kind of semantics described so far each construct *directly* denotes its input/result transformation, and the transformation of a complete program is got (roughly) by combining (with ★) the transformations of its components. This type of semantics is called a *direct semantics* and it suffers from the problem that there is no way a construct can avoid passing its result to the rest of the program following it. If a construct produces an abnormal result, say **error**, then the rest of the program has to cope with this. Thus semantic clauses get cluttered up with tests for abnormal values. Although these tests can be hidden in operators like ★ the checking involved is unnatural—intuitively when an error occurs the rest of the program is simply ignored and the computation just stops.

In a *continuation semantics* we make the denotations of constructs depend on the 'rest of the program'— or *continuation*—following them. The intuitive idea is that each construct decides for itself where to pass its result. Usually it will pass it to the continuation corresponding to the 'code' textually following it in the program—the *normal continuation*—but in some cases this will be ignored and the result passed to some other 'abnormal' continuation. For example:

(i) When an error occurs the normal continuation is ignored and control passes to a continuation corresponding to an error stop.

(ii) When a jump occurs the normal continuation is ignored and control passes to a continuation corresponding to the rest of the program following the label jumped to.

This is all very vague and probably will not be fully comprehensible until we have explained the formal details. Continuations can, at first, be a bit tricky and the reader should not worry if to start with everything seems

back to front and unnaturally higher order—one needs to work through a few concrete examples before thinking with continuations comes naturally.

We now explain exactly what continuations are and which aspects of the 'rest of the program' they model. Formally a continuation is a function from whatever the 'rest of the program' expects to be passed as an intermediate result—and this depends on what the 'rest of the program' follows— to the *final answer* of the program. For example in the TINY program **I:=E;C** the intermediate result passed to the 'rest of the program' following **I:=E**—i.e. passed to **C**—is a state and the final answer is a state or **error**; however the intermediate result passed to the 'rest of the program' following **E**—i.e. passed to **I:=;C**—is a value (**E**'s value) together with a state (the state after doing **E**). The continuations corresponding to the first case—i.e. corresponding to the 'rest of the program' following a command— are called *command continuations* and form a domain **Cont** defined by

Cont = State→[State + {error}]

- **Cont**: command continuations
- **State**: results of commands
- **[State + {error}]**: final answers

Continuations corresponding to the second case—i.e. corresponding to the 'rest of the program' following expressions—are called *expression continuations* and form a domain **Econt** defined by:

Econt = Value→State→[State + {error}]

- **Econt**: expression continuations
- **Value**: results of expressions (curried)
- **State→[State + {error}]**: final answers

We define **Econt** as above rather than uncurried as **[Value × State]→[State + {error}]** so that **Econt = Value→Cont**—this turns out to be very convenient when we come to write semantic clauses.

We shall use **c**, **c'**, $\mathbf{c_1}$, $\mathbf{c_2}$ etc for typical members of **Cont**; **k**, **k'**, $\mathbf{k_1}$, $\mathbf{k_2}$ etc. will be typical members of **Econt**. The continuations described above are appropriate for TINY but slightly too simple for general use. In 5.5. we describe domains **Cc** and **Ec** of *standard* command and expression continuations; however for the time being we shall continue to work with **Cont** and **Econt** since they embody the essential ideas without unnecessary extra detail.

5.1.2. Direct and continuation semantics

Recall that in our direct semantics of TINY (see 4) we used semantic functions with the following types:

E:Exp→State→[[Value × State] + {error}]
C:Com→State→[State + {error}]

In a *continuation semantics* the denotations of constructs are functions of continuations as well as of states. If we (temporarily) use **E'** and **C'** for the continuation semantic functions, then their types are:

E':Exp→Econt→State→[State + {error}]
(i.e. E':Exp→Econt→Cont)
C':Com→Cont→State→[State + {error}]
(i.e. C':Com→Cont→Cont)

and are defined so that:

$$\mathbf{E'[E]\,k\,s} = \begin{cases} \mathbf{k\,v\,s'} & \text{where } \mathbf{E} \text{ has value } \mathbf{v} \text{ and transforms } \mathbf{s} \text{ to } \mathbf{s'}. \\ \mathbf{error} & \text{otherwise.} \end{cases}$$

$$\mathbf{C'[C]\,c\,s} = \begin{cases} \mathbf{c\,s'} & \text{if } \mathbf{C} \text{ transforms } \mathbf{s} \text{ to } \mathbf{s'} \\ \mathbf{error} & \text{otherwise} \end{cases}$$

Thus in general the continuation denotation of a construct is a function of a continuation and state which yields the final answer of the program—this final answer being obtained by passing some intermediate results (e.g. **v**, **s'** etc.) to some continuation (i.e. to the 'rest of the program'). Normally this continuation will correspond to the program text following the construct, but sometimes other continuations are appropriate. For

example in the case of *jumps*—commands of the form **goto l** say—the semantic clause will roughly be:

$\mathbf{C'[goto\ l]\,c\,s = c'\,s}$ where $\mathbf{c'}$ is some continuation specified by **l** (e.g. bound to **l** in **s**)

We shall discuss this kind of thing in detail later, however for the time being we concentrate on explaining what a continuation semantics just for TINY looks like. We start with identifiers.

The continuation semantic clause for identifiers is:

$$\mathbf{E'[I]\,k\,(m,i,o) = (m\,I = unbound) \rightarrow error,\ k(m\,I)(m,i,o)}$$

Thus the final answer yielded by **I** when the 'rest of the program' is **k** and the state is **(m,i,o)** is **error** if **I** is unbound in **m**, and otherwise is the final answer produced by **k** when it is passed **m I** and **(m,i,o)** as intermediate results. Notice how continuation semantics allows us to explicitly specify final answers—e.g. **error**—as well as intermediate results (e.g. **m I**). In the case that **m I** is **unbound** we ignore **k** and immediately stop with **error**. In a direct semantics one can only specify the intermediate results and so **error** has to be an intermediate result with the consequence that messy operators like ★ are then needed to handle it. To see this more clearly here is the semantic clause for $\mathbf{C_1;C_2}$:

$$\mathbf{C'[C_1;C_2]\,c\,s = C'[C_1]\,(C'[C_2]\,c)\,s}$$

This says we evaluate $\mathbf{C_1}$ with a continuation $\mathbf{C'[C_2]\,c}$ (i.e. a 'rest of the program' corresponding to $\mathbf{C_2}$ followed by **c**). Now if $\mathbf{C_1}$ generates an error then it can simply ignore its continuation $\mathbf{C'[C_2]\,c}$ and 'stop' with final answer **error**. In a direct semantics denotations cannot choose to *not* send an intermediate result to the 'rest of the program'. One way of thinking of direct semantics is as a special kind of continuation semantics in which one is always forced to send some intermediate result to the normal (i.e. textual) continuation. This follows from the fact that one can prove (for TINY) that:

for all $\mathbf{E}\varepsilon\mathbf{Exp}$, $\mathbf{k}\varepsilon\mathbf{Econt}$: $\mathbf{E'[E]\,k = E[E]}\star\mathbf{k}$
for all $\mathbf{C}\varepsilon\mathbf{Com}$, $\mathbf{c}\varepsilon\mathbf{Cont}$: $\mathbf{C'[C]\,c = C[C]}\star\mathbf{c}$

For TINY this actually defines $\mathbf{E'}$ and $\mathbf{C'}$, but normally one has to write out semantic clauses for continuation semantic functions directly since there

may be no direct semantics available. For example it is rather hard (though not impossible) to define $\mathbf{C[goto\ l]}$ and to extend ★ so that: $\mathbf{C'[goto\ l]\,c = C[goto\ l]}\star\mathbf{c}$. The meaning of ★ (at least as defined in 3.4.12.2.) implies that if $\mathbf{C[goto\ l]\,s}$ is not **error** then $\mathbf{C[goto\ l]}\star\mathbf{c = c\,(C[goto\ l])}$ so there is no way to avoid **c** getting some intermediate result passed to it. In other words $\mathbf{C'[goto\ l]\,c = C[goto\ l]}\star\mathbf{c}$ is incompatible with $\mathbf{C'[goto\ l]\,c}$ ignoring **c**. By complicating ★ it is possible to get round this and to obtain a direct semantics of **goto l** (i.e. to concoct a definition of $\mathbf{C[goto\ l]}$ that works) but the details are messy and only of technical interest (see [Bjorner & Jones]). For languages in which the flow of control may fail to follow the textual structure of programs the most natural description is a continuation semantics. From now on all our semantics will use continuations and we shall drop the primes on continuation semantic functions.

5.1.3. Continuation semantics of TINY

In this section we illustrate the ideas just described by giving a continuation semantics of TINY.

5.1.3.1. Semantic domains and functions

Domains:

State	=	**Memory × Input × Output**
Memory	=	**Ide→[Value + {unbound}]**
Input	=	**Value***
Output	=	**Value***
Value	=	**Num + Bool**
Cont	=	**State→[State + {error}]**
Econt	=	**Value→Cont**

Functions:

E:Exp→Econt→Cont
C:Com→Cont→Cont

(we no longer use $\mathbf{E'}$ and $\mathbf{C'}$)

5.1.3.2. Semantic clauses

Expressions:

(E1) $\mathbf{E[0]\,k\,s = k\,0\,s,\ E[1]\,k\,s = k\,1\,s}$

Each numeral passes the appropriate number, together with an unchanged state, to the continuation. The state is unchanged since evaluating numerals has no side effects. Notice that by cancelling (see 3.4.10.) (E1) can be simplified to:

$$\mathbf{E[0]\,k = k\,0\,,\ E[1]\,k = k\,1}$$

(E2) $\mathbf{E[true]\,k = k\ true\,,\ E[false]\,k = k\ false}$

(E3) $\mathbf{E[read]\,k\,(m,i,o) = null\ i \to error\,.\ k\,(hd\ i)\,(m,tl\ i,o)}$

If the input **i** is empty then the program stops with final answer **error**, otherwise **hd i** and the side-effected state **(m,tl i,o)** are passed as intermediate results to **k**.

(E4) $\mathbf{E[I]\,k\,(m,i,o) = (m\ I = unbound) \to error\,.\ k\,(m\ I)\,(m,i,o)}$

This was explained above.

(E5) $\mathbf{E[not\ E]\,k\,s = E[E]\,(\lambda v\,s'\,.\,isBool\ v \to k\,(not\ v)\,s'\,,\ error)\,s}$

E is evaluated and the resulting value **v** and state **s'** are passed to the continuation $\mathbf{(\lambda v\,s'\,.\,isBool\ v \to k\,(not\ v)\,s',\ error)}$. This continuation sends **not v** and **s'** to **k** if $\mathbf{v \in Bool}$, otherwise **k** is ignored and the result is **error**. If we define $\mathbf{err = \lambda s\,.\,error}$ then the techniques of 3.4.10 allow us to simplify (E5) to:

$$\mathbf{E[not\ E]\,k = E[E]\lambda v\,.\,isBool\ v \to k\,(not\ v),\ err}$$

(E6) $\mathbf{E[E_1 = E_2]\,k = E[E_1]\,\lambda v_1\,.\,E[E_2]\,\lambda v_2\,.\,k(v_1 = v_2)}$

We evaluate $\mathbf{E_1}$ to get $\mathbf{v_1}$, which is then passed to $\mathbf{\lambda v_1\,.\,E[E_2]\lambda v_2\,.\,k(v_1 = v_2)}$, this evaluates $\mathbf{E_2}$ to get $\mathbf{v_2}$ which is then passed to the continuation $\mathbf{\lambda v_2\,.\,k(v_1 = v_2)}$ which passes $\mathbf{v_1 = v_2}$ to **k**. If $\mathbf{E_1}$ caused an error then its continuation would be ignored, for example

$$\begin{aligned}\mathbf{E[read = E]\,k\,(m,(),o)} &= \mathbf{E[read](\ldots)(m,(),o)}\\ &= \mathbf{error}\ \text{(since}\ \mathbf{null\ () = true}\ \text{– see (E3))}\end{aligned}$$

(E7) $\mathbf{E[E_1 + E_2]\,k =}$
$\mathbf{E[E_1]\lambda v_1\,.\,E[E_2]\lambda v_2\,.\,isNum\ v_1\ and\ isNum\ v_2 \to k(v_1 + v_2),\ err}$

$\mathbf{err = \lambda s\,.\,error}$; thus $\mathbf{E_1}$ is evaluated to get $\mathbf{v_1}$ then $\mathbf{E_2}$ is evaluated to get $\mathbf{v_2}$, then if $\mathbf{v_1}$ and $\mathbf{v_2}$ are both numbers $\mathbf{v_1 + v_2}$ is passed to **k**, otherwise **k** is ignored and **error** results.

Commands:

(C1) $\mathbf{C[I:=E]\,c = E[E]\lambda v(m,i,o)\,.\,c(m[v/I],i,o)}$

E is evaluated and its result passed to the continuation $\mathbf{\lambda v(m,i,o)\,.\,c(m[v/I],i,o)}$ which passes $\mathbf{(m[v/I],i,o)}$ to **c**.

(C2) $\mathbf{C[output\ E]\,c = E[E]\lambda v(m,i,o)\,.\,c(m,i,v\,.\,o)}$

The value of **E** is put on to the front of the output component of the store and the result passed to **c**. Further discussion of output is given in the next section.

(C3) $\mathbf{C[if\ E\ then\ C_1\ else\ C_2]\,c =}$
$\mathbf{E[E]\lambda v\,.\,isBool\ v \to (v \to C[C_1]\,c,\ C[C_2]\,c),\ err}$

(C4) $\mathbf{C[while\ E\ do\ C]\,c =}$
$\mathbf{E[E]\lambda v\,.\,isBool\ v \to (v \to C[C];\ C[while\ E\ do\ C]\,c,\ c),\ err}$

As described in 3.4.3. (iii) $\mathbf{C[C];\ C[while\ E\ do\ C]\,c}$ means $\mathbf{C[C](C[while\ E\ do\ C]\,c)}$. Thus if **E**'s value is **true** **C** is done and the resulting state passed back to $\mathbf{C[while\ E\ do\ C]\,c}$ ((C4) is recursive). If **v** is **false** then the state resulting from **E** is sent straight to **c**.

Thus $\mathbf{C[while\ false\ do\ C]\,c}$
$\begin{aligned}&= \mathbf{E[false]\lambda v\,.\,isBool\ v \to (v \to \ldots,\ c),\ err}\\ &= \mathbf{(\lambda v\,.\,isBool\ v \to (v \to \ldots,\ c),\ err)\ false}\\ &= \mathbf{(false \to \ldots,\ c)}\\ &= \mathbf{c}\end{aligned}$

so **while false do C** has no effect.

(C5) $\mathbf{C[C_1;C_2] = C[C_1] \circ C[C_2]}$

here $\circ$ is function composition (see 3.4.12.1.) so (C5) is equivalent to:

$$\mathbf{C[C_1;C_2]\,c\,s = C[C_1]\,(C[C_2]\,c)\,s}$$

which we discussed above.

5.1.4. Final answers and output

In this section we point out, and correct, three inaccuracies which have crept into our semantics. These are:

5.1.4.1. Final answers are not states

In defining continuations we took the domain of final answers to be **[State + [error]]**. This is unnatural—in practice the result of running a program is not thought of as the whole state but just the output (together with an indication of whether the program halted normally or via an error).

5.1.4.2. Output is not part of the state

By defining **State = Memory × Input × Output** we made the output file just as accessible to programs as the memory and input. Usually information which has been output cannot be retrieved—the command **output E** should put **E**'s value directly on to the final answer rather than pass it to the rest of the program embedded in the state.

5.1.4.3. Output can be infinite

The definition **Output = Value*** assumes that the output of programs can be represented by a *finite* string of values. This is incorrect for non-terminating programs like: **x:=0; while true do (output x; x:=x+1)** whose output is infinite (namely **0.1.2.3**...).

All the inaccuracies of the last three sections can be corrected if we remove output from the state and change the type of continuations. The required domain equations are:

$$\begin{aligned}
\textbf{State} &= \textbf{Memory} \times \textbf{Input} \\
\textbf{Memory} &= \textbf{Ide} \to [\textbf{Value} + \{\textbf{unbound}\}] \\
\textbf{Input} &= \textbf{Value}^* \\
\textbf{Value} &= \textbf{Num} + \textbf{Bool} \\
\textbf{Cont} &= \textbf{State} \to \textbf{Ans} \\
\textbf{Econt} &= \textbf{Value} \to \textbf{Cont} \\
\textbf{Ans} &= [\textbf{error}, \textbf{stop}] + [\textbf{Value} \times \textbf{Ans}]
\end{aligned}$$

The domain **Ans** of *final answers* is defined recursively (see 3.2.2. and 3.3.2.). A final answer is either **error**, **stop** or a value paired with another answer—'unwinding' this we see an answer is either a *finite* string of values ending with **error** or **stop** or an infinite string of values. Because output is no longer part of the state we must change the semantic clause for **output E** to:

$$\mathbf{C[output\ E]\,c = E[E]\lambda v s\,.\,(v, c\,s)}$$

Thus **E** is evaluated to get a value **v** and state **s** and then the final answer is **v** followed by whatever the 'rest of the program' **c** yields when given **s**. This equation models output as like a 'system call' to an output device—if the program 'crashes' *after* outputting then the values output will already be in the final answer (e.g. 'printed on the line printer'). For example if **c = λs . error** a continuation representing a 'rest of the program' that generates an error for all states, then:

$$\begin{aligned}
\mathbf{C[output\ 0]\,c} &= \mathbf{E[0]\lambda v s\,.\,(v, c\,s)} \\
&= \mathbf{E[0]\lambda v s\,.\,(v, error)} \\
&= \mathbf{(\lambda v s\,.\,(v, error))\,0} \\
&= \mathbf{\lambda s\,.\,(0, error)}
\end{aligned}$$

With the old semantics (with output in the state) outputting is like writing an 'in core' file and so if a 'crash' occurs all previous output will be lost. For example recall (C2) in 5.1.3.2.—with that definition of **C[output E]** we get:

$$\begin{aligned}
\mathbf{C[output\ 0]\,c} &= \mathbf{E[0]\lambda v(m,i,o)\,.\,c(m,i,v\,.\,o)} \\
&= \mathbf{E[0]\lambda v(m,i,o)\,.\,error} \\
&= \mathbf{E[0]\lambda v s\,.\,error} \\
&= \mathbf{(\lambda v s\,.\,error)\,0} \\
&= \mathbf{\lambda s\,.\,error}
\end{aligned}$$

Using the new continuations with **Ans** we can also express the fact that when a program has finished it should stop. If **C** is a command representing some program to be run then the answer it produces with initial state **s** is just **C[C](λs . stop)s**—the 'rest of the program' represented by the continuation **λs . stop** just outputs **stop** and stops. If **C** were to run on forever **λs . stop** would never be reached.

In general the domain **Ans** of final answers is language dependent—for some languages (e.g. machine code) it might include the whole state (as we initially had things) or even the sequence of states gone through

during the computation. Determining **Ans** in part of the work that has to be done when we write the semantics of a language.

5.2. Locations, stores and environments

In some languages identifiers are not bound directly to values but instead are bound to things called *variables* and it is these that possess values. one might think that these variables were formally unecessary and that (as in TINY) one could just bind each identifier to some value (namely the value possessed by the corresponding variable). This works fine unless what is called *sharing* (or *aliasing*) is possible.

5.2.1. Sharing

Sometimes distinct identifiers can come to denote the same variable so that assigning to one of them will have the side effect of changing the value of the other. Such sharing of a single variable by several identifiers (or aliases) may occur as the result of an explicit command or declaration —for example one might have a command **let** $\mathbf{I}_1 = = \mathbf{I}_2$ whose evaluation causes $\mathbf{I}_1$ and $\mathbf{I}_2$ to denote the same variable. Sharing can also come about indirectly—for example in PASCAL if one declares a procedure of the form **procedure P(var x:real; var y:real)**<statement part> and then executes a call **P(z,z)** then inside the body of the procedure (i.e. inside <statement part>) both **x** and **y** will share the variable denoted by **z**.

If sharing is possible then we have no choice but to introduce some model of variables into our semantics. For suppose not: then if $\mathbf{I}_1$ and $\mathbf{I}_2$ share, when we execute the command $\mathbf{I}_1 := \mathbf{1}$ we must ensure in our model that $\mathbf{I}_2$ gets value **1** also—the semantic clause for assignments would thus have to be:

$$\mathbf{C[I:=E]\,c = E[E]\lambda v(m,i,o)\,.\,c(m[v,v,\ldots,v/I,I_1,\ldots,I_n],i,o)}$$

where $\mathbf{I}_1,\ldots,\mathbf{I}_n$ are all the identifiers which share with **I**

However to make this precise we have to define what it means for two identifiers to share and the simplist way of doing this is to say two identifiers share when they denote the same variable.

5.2.2. Variables and locations

The word "variable" has a number of misleading connotations so in connection with formal semantics the less loaded term "location" is often used instead. To model locations (i.e. variables) we introduce a new primitive domain, **Loc**, typical members of which are **ι**, **ι'**, $\boldsymbol{\iota}_1$, $\boldsymbol{\iota}_2$ etc. The only structure we shall assume on **Loc** is that locations can be tested for equality with **=**. When modelling implementations it may be convenient to make **Loc** more concrete, for example to identify it with **Num**—for abstract semantics this is unnecessary.

5.2.3. Stores

To model the association of locations with values we introduce the concept of a *store*. The domain **Store** of stores is defined by:

$$\mathbf{Store = Loc \to [Sv + \{unused\}]}$$

Where **Sv** is a language dependent domain of *storable values*—the values that can be held in locations. Exactly what the storable values of a language are is an important question to ask and forms a dimension along which languages can be classified [Strachey 74]. If $\mathbf{s \varepsilon Store}$ and $\boldsymbol{\iota \varepsilon Loc}$ then if $\mathbf{s\,\iota = v \varepsilon Sv}$ we say **ι** has (or holds) value **v** and if $\mathbf{s\,\iota = unused}$ we say **ι** is unused in **s**. Certain constructs (for example ALGOL 60 blocks), cause new locations to come into use and to model these we assume a function **new:Store→[Loc + {error}]** which has the property that if **isLoc(new s) = true** then **s(new s) = unused**—i.e. **new s** is an unused location in **s** (**new s = error** means that there are no locations available in **s**). There are serious dangers in only *partially* specifying function like we have just done for **new**. For example suppose we had assumed **new:Store→Loc** and that, for all **s**, **s(new s) = unused**. Then for any $\mathbf{v \varepsilon Sv}$ if $\mathbf{s \varepsilon Store}$ is defined by $\mathbf{s = \lambda\iota\,.\,v}$ then $\mathbf{v = s(new\,s) = unused!}$ The lesson to learn from this is that one must be *very* careful about *assuming* there exist functions with given properties. In fact the assumptions we made are safe. To see that there do exist **new**'s satisfying them we arbitrarily choose $\boldsymbol{\iota}_1, \boldsymbol{\iota}_2, \ldots, \boldsymbol{\iota}_n \varepsilon \mathbf{Loc}$ and then *define* **new** by:

$$\begin{aligned}\mathbf{new} = \lambda\mathbf{s}\,.\,&\mathbf{s}\,\iota_1 = \mathbf{unused} \to \iota_1,\\ &\mathbf{s}\,\iota_2 = \mathbf{unused} \to \iota_2,\\ &\qquad\vdots\\ &\mathbf{s}\,\iota_n = \mathbf{unused} \to \iota_n, \mathbf{error}\end{aligned}$$

This models finite stores with locations $l_1, l_2, \ldots, l_n$ and where **new s** is the first unused l_i (or **error** if all the l_i's are in use). Clearly for *this particular* **new** if **isLoc(new s) = true** then **s(new s) = unused**. When Modelling implementations it is necessary to specify details of storage allocation and so particular **new**'s must be defined; however for our purposes this is unnecessary. Finally note that we shall use **s**, $\mathbf{s}_1$, $\mathbf{s}_2$ etc. to range over **store** and $\mathbf{v}_1,\ldots,\mathbf{v}_n/l_1,\ldots,l_n$ to denote the 'little' store defined by:

$$\mathbf{v}_1,\ldots,\mathbf{v}_n/l_1,\ldots,l_n = \lambda l\,.\,l = l_1 \rightarrow \mathbf{v}_1,\ l = l_2 \rightarrow \mathbf{v}_2,\ \vdots\ l = l_n \rightarrow \mathbf{v}_n,\ \mathbf{unused}$$

Thus in $\mathbf{v}_1,\ldots,\mathbf{v}_n/l_1,\ldots,l_n$ the only locations in use are $l_1,\ldots,l_n$ and these hold $\mathbf{v}_1,\ldots,\mathbf{v}_n$ respectively.

5.2.4. Environments

To model the binding of identifiers we use the concept of an *environment*. The domain **Env** of environments is defined by:

$$\mathbf{Env} = \mathbf{Ide} \rightarrow [\mathbf{Dv} + \{\mathbf{unbound}\}]$$

Where **Dv** is the language dependent domain of *denotable values*. If the only thing identifiers can be bound to is locations then **Dv = Loc**; however in most languages identifiers can denote other things besides locations — for example constants (e.g. numbers), arrays, records, procedures etc. Exactly what the denotable values are is an important question to ask about a language and, like the storable values, forms a dimension for classification. We shall use **r**, **r**′, $\mathbf{r}_1$, $\mathbf{r}_2$ etc to range over **Env** and also the following two notations:

(i) If $\mathbf{d}_1,\ldots,\mathbf{d}_n \varepsilon \mathbf{Dv}$ and $l_1,\ldots,l_n \varepsilon \mathbf{Ide}$ then $\mathbf{d}_1,\ldots,\mathbf{d}_n/l_1,\ldots,l_n$ is the 'little' environment defined by

$$\mathbf{d}_1,\ldots,\mathbf{d}_n/l_1,\ldots,l_n = \lambda l\,.\,l = l_1 \rightarrow \mathbf{d}_1,\ \vdots\ l = l_n \rightarrow \mathbf{d}_n,\ \mathbf{unbound}$$

(ii) If $\mathbf{r}_1, \mathbf{r}_2 \varepsilon \mathbf{Env}$ then $\mathbf{r}_1[\mathbf{r}_2] \varepsilon \mathbf{Env}$ is defined by:

$$\mathbf{r}_1[\mathbf{r}_2] = \lambda l\,.\,\mathbf{r}_2 l = \mathbf{unbound} \rightarrow \mathbf{r}_1 l,\ \mathbf{r}_2 l$$

If we put (i) and (ii) together we get

$$\mathbf{r}[\mathbf{d}_1,\ldots,\mathbf{d}_n/l_1,\ldots,l_n] = \lambda l\,.\,l = l_1 \rightarrow \mathbf{d}_1,\ \vdots\ l = l_n \rightarrow \mathbf{d}_n,\ \mathbf{r}l$$

which is consistent with the notation described in 3.4.7.

5.3. Standard domains of values

In TINY we had just one domain of values, namely **Value**. In general one needs to distinguish several value domains, amongst the most important are:

(i) *Storable* values **Sv** — the values which can be stored in locations in the store, we use **v**, **v**′, $\mathbf{v}_1$, $\mathbf{v}_2$ etc. to range over **Sv**.
(ii) *Denotable* values **Dv** — the values which can be bound to (or "denoted by") identifiers in the environment. We use **d**, **d**′, $\mathbf{d}_1$, $\mathbf{d}_2$ etc. to range over **Dv**.
(iii) *Expressible* values **Ev** — the values expressions can produce as results. We use **e**, **e**′, $\mathbf{e}_1$, $\mathbf{e}_2$ etc. to range over **Ev**.

Although these domains usually intersect they are conceptually distinct and are not normally the same. For example in PASCAL constants such as numbers and truth values are denotable but in ALGOL 60 they are not. In PASCAL locations are storable but in ALGOL 60 they are not. Other domains of values besides **Sv**, **Dv** and **Ev** may also be needed for particular languages. For example one may need to explicitly distinguish and define the 'outputable values' or the values that can be passed to procedures. Another general purpose value domain which we discuss later (in 5.8.) is **Rv** the domain of *R-values* — this is the domain of 'dereferenced' expression values (called "R-values" because they are extracted on the right of assignments).

5.4. Blocks, declarations and scope

In languages like ALGOL 60 and PASCAL commands change the

contents of locations but not the binding of identifiers to locations. For example in the ALGOL 60 program:

```
begin integer x;
      ⋮
      x:=1;
      ⋮
      x:=2;
      ⋮
end
```

throughout the body of the block (the text between **begin** and **end**) **x** denotes a *fixed* location, ι say, in the environment but the contents of ι changes with each assignment to **x**. The only way bindings in the environment can be changed is with *declarations*. For example in:

```
begin integer x;
      integer y;
      ⋮
      begin  integer x;
             ⋮
      end
      ⋮
end
```

The **x** in the inner and outer blocks denote different locations. Since **y** is not declared in the inner block it denotes the same location in this as it does in the outer block. Each declaration 'holds' throughout a certain part of the program. In ALGOL 60 declarations hold throughout the body of the block in which they are introduced unless they are *overwritten* by a declaration in an inner block. In PASCAL declarations hold throughout the procedure body at whose head they occur unless they are overwritten by a with-statement (see 9.) within this procedure body. In the example above (which is based on ALGOL 60) the outer declaration of **x** holds throughout the parts of the outer block not included in the inner one. The declaration of **y** holds throughout the whole outer block. If **x** (like **y**) were not declared in the inner block then its outer declaration would hold there also.

The parts of a program where a declaration holds are called its *scope*.

The example above illustrates *holes* in scope: the inner block is a hole in the scope of the outer declaration of **x** but not of **y**. The scope of an identifier with respect to a declaration is the scope of the declaration. If this declaration is unique (or defined by the context) then we can unambiguously talk about the scope of the identifier. Thus "the scope of y" makes sense in the above example but not "the scope of x"; the latter phrase is ambiguous between the scopes of the inner and out declarations of **x**.

In standard semantics:

(i) Commands change the store but not the environment.
(ii) Declarations change the environment (and possibly also the store).

Thus environments only change at 'scope boundaries'. Declarations may change the store as well as the environment since—for example in the case of variable declarations—they may 'allocate' new storage (i.e. put new locations into use). In the language SMALL described below in 6. there are variable declarations of the form **var I = E** which have the effect that:

(i) A new location, ι say, is obtained.
(ii) **E**'s value is stored in ι.
(iii) ι is bound to **I** in the environment.

(ii) changes the store and (iii) the environment.

In general the declarations of a language form a syntactic category modelled by a syntactic domain **Dec**. We shall use **D** to range over **Dec** and **D** for the associated semantic function (see 5.6.). In SMALL **Dec** will be defined by:

$$\mathbf{D} ::= \mathbf{const}\ \mathbf{I} = \mathbf{E} \mid \mathbf{var}\ \mathbf{I} = \mathbf{E} \mid \mathbf{proc}\ \mathbf{I}(\mathbf{I}_1);\mathbf{C} \mid \mathbf{fun}\ \mathbf{I}(\mathbf{I}_1);\mathbf{E} \mid \mathbf{D}_1;\mathbf{D}_2$$

var I = E is as just described, **const I = E** binds **E**'s value directly to **I** in the environment (and so does not change the store), **proc I(I₁);C** declares a *procedure* named **I** with *formal parameter* **I₁** and *body* **C** (see 5.9.1. for details), **fun I(I₁);E** declares a *function* named **I** with formal parameter **I₁** and body **E** (see 5.9.2.) and finally **D₁;D₂** is a *compound declaration* whose effects are those of **D₁** followed by those of **D₂**.

In the semantics (see 5.5.3. and 5.6.) each declaration generates a 'little' environment containing the bindings it specifies. For example

const I = E generates e/I where e is E's value, whereas **var** I = E generates ι/I where ι is a new location updated with E's value. This little environment is then passed, together with a possibly changed store, to the rest of the program following the declaration.

5.5. Standard domains of continuations

In standard semantics three main kinds of continuation are used.

5.5.1. Command continuations

Since commands pass a store to the rest of the program following them we define **Cc** – the domain of *command continuations* – by:

$$\mathbf{Cc = Store \rightarrow Ans}$$

Where **Ans** is the domain of *final answers*. The exact structure of **Ans** is language dependent; all we shall assume is that it contains an error element **error**. We shall use **c**, **c'**, $\mathbf{c_1}$, $\mathbf{c_2}$ etc. to range over **Cc**.

5.5.2. Expression continuations

Since expressions pass their values, together with a possibly changed store, to the rest of the program following them we define **Ec** – the domain of *expression continuations* – by:

$$\mathbf{Ec = Ev \rightarrow Store \rightarrow Ans}$$

or more neatly:

$$\mathbf{Ec = Ev \rightarrow Cc}$$

We use **k**, **k'**, $\mathbf{k_1}$, $\mathbf{k_2}$ etc to range over **Ec**.

5.5.3. Declaration continuations

Since declarations pass an environment, together with a possibly changed store, to the rest of the program following them we define **Dc** – the domain of *declaration continuations* – by:

$$\mathbf{Dc = Env \rightarrow Store \rightarrow Ans}$$

or more neatly:

$$\mathbf{Dc = Env \rightarrow Cc}$$

We use **u**, **u'**, $\mathbf{u_1}$, $\mathbf{u_2}$ etc. to range over **Dc**.

5.6. Standard semantic functions

In a standard semantics we use semantic functions with the following types:

$$\mathbf{E: Exp \rightarrow Env \rightarrow Ec \rightarrow Store \rightarrow Ans}$$
$$\mathbf{C: Com \rightarrow Env \rightarrow Cc \rightarrow Store \rightarrow Ans}$$
$$\mathbf{D: Dec \rightarrow Env \rightarrow Dc \rightarrow Store \rightarrow Ans}$$

When no errors, jumps etc. occur the intuitive meanings of these functions are:

E[E] r k s = k e s' where **e** is E's value in environment **r** and store **s** and **s'** is the store after E's evaluation.

C[C] r c s = c s' where **s'** is the store resulting from executing **C** in environment **r** and store **s**.

D[D] r u s = u r' s' where **r'** is the environment consisting of the 'bindings' specified in **D** (when evaluated with respect to **r** and **s**) and **s'** the store resulting from **D**'s evaluation.

Thus each semantic function passes the appropriate intermediate results to its continuation. Examples of typical semantic clauses for SMALL are

(E1) **E[0] r k s = k 0 s**

0's value **0** is passed, together with an unchanged store, to **k**.

(C7) **C[C₁;C₂] r c s = C[C₁] r (λs'.C[C₂] r c s') s**

$\mathbf{C_1}$ is executed in environment **r** and store **s** to get a store **s'** which is passed to the continuation **λs'.C[C₂] r c s'** which executes $\mathbf{C_2}$ in the same environment but in store **s'** and then finally sends the resulting store on to **c**. Notice that since commands don't change the environment **r** is passed to both $\mathbf{C_1}$ and $\mathbf{C_2}$. Note also that the semantic clause can be simplified to

$$\mathbf{C[C_1;C_2]\,r\,c = C[C_1]\,r\,(C[C_2]\,r\,c)} \qquad \text{(see 3.4.10.)}$$

or $$\mathbf{C[C_1;C_2]\,r\,c = C[C_1]\,r;C[C_2]\,r\,c} \qquad \text{(see 3.4.3. (iii))}$$

or even $$\mathbf{C[C_1;C_2]\,r = C[C_1]\,r \circ C[C_2]\,r} \qquad \text{(see 3.4.12.1.).}$$

(D1) $$\mathbf{D[const\ I = E]\,r\,u\,s = E[E]\,r\,(\lambda e s'.\,u(e/I)s')\,s}$$

Here **E**'s value **e** is bound to **I** to form the little environment **e/I** which is passed on to **u** together with the store **s'** resulting from **E**'s evaluation. The clause can be simplified to:

$$\mathbf{D[const\ I = E]\,r\,u = E[E]\,r\,\lambda e.\,u(e/I)}$$

(N.B. This clause although right in spirit has a minor error—the value of **E** needs to be 'dereferenced' before being bound to **I**. See 5.8 for a discussion and 6.2.3.4. for the correct equation).

The rest of the semantic clauses for SMALL are explained in the next chapter (specifically 6.2.3.).

5.7. Continuation transforming functions

In this section we describe some functions for transforming continuations. These functions *intuitively* transform values and stores to new values and stores but are defined over continuations to make them convenient for use in continuation semantics. The general idea is as follows: suppose **f:Ev→Store→[[Ev × Store] + [error]]** then instead of using **f** we shall use **f':Ec→Ec** defined by:

$$\mathbf{f'\,k = \lambda e s.\,(f\,e\,s = (e',s'))\rightarrow k\,e's', error}$$
$$\mathbf{= f \star k} \quad \text{(using } \star \text{ as defined in 3.4.12.2.)}$$

Thus **f'** takes a continuation **k** and produces another continuation which 'first does **f** and then passes the results to **k**'. The relationship between **f** and **f'** can be factored out into a function **mkconfun** defined so that **f' = mkconfun f**. The appropriate definition is simply:

$$\mathbf{mkconfun = \lambda f k.\,f \star k}$$

The use of these continuation transformers is as follows: frequently we want to evaluate an expression, **E** say, and then transform the resulting values by a sequence of functions, $\mathbf{f_1,\ldots,f_n}$ say, and then pass the result of these transformations to the 'rest of the program', **k** say. Now if

$\mathbf{f_1',\ldots,f_n'}$**:Ec→Ec** are the continuation transformations corresponding to $\mathbf{f_1,\ldots,f_n}$ (i.e. $\mathbf{f_i' = mkconfun\ f_i}$) then the desired effect is modelled by:

$$\mathbf{E[E]\,r\,(f_1'(f_2'\ldots(f_n'\,k)\ldots))}$$

or using ";" (see 3.4.3.):

$$\mathbf{E[E]\,r;f_1';\ldots;f_n';k}$$

To see that this works we simply note that

$$\mathbf{f_1'(f_2'\ldots(f_n'\,k)\ldots) = f_1 \star f_2 \star \ldots \star f_n \star k}$$

Summing up an expression like **E[E] r;f₁';...;fₙ';k** should be 'read' as "evaluate **E**, do $\mathbf{f_1',\ldots,f_n'}$ and then pass the results to **k**".

In what follows we define the continuation transformations (e.g. $\mathbf{f_1'}$) directly since the non continuation ones (e.g. $\mathbf{f_1}$) will not be of use to us. If we were going to use direct semantics then it would be the non continuation transformations that we would need. For example compare **checkNum** and **checkBool** which we used in the direct semantics of TINY (see 4.2.2.) with **Num?** and **Bool?** which are what we shall use with continuations.

5.7.1. cont:Ec→Ec

Informal description

cont k e s checks that **e** is a location and if so looks up its contents in **s** and passes the result, together with **s**, to **k**. If **e** is not a location or if it is but is unused in **s** then **cont k e s = error**.

Formal description

cont = λk e s. isLoc e→(s e = unused→error, k(s e)s); error

5.7.2. update:Loc→Cc→Ec

Informal description

update ı c e s stores **e** at location **ı** in **s** and passes the resulting store to **c**. If **e** is not storable value then **update ı c e s = error**.

Formal description

update = λι c e s . isSv e→c(s[e/ι]), error

5.7.3. ref:Ec→Ec

Informal description

ref k e s gets an unused location from **s**, updates it with **e** and then passes it, and the updated store, to **σ**. If **s** has no unused locations available then **ref k e s = error**.

Formal description

ref = λk e s . new s = error→error, update (new s)(k(new s))e s

5.7.4. deref:Ec→Ec

Informal description

deref k e s tests whether **e** is a location and if so passes its contents in **s**, together with **s**, to **k**. If **e** is not a location then **e** and **s** are passed to **k**.

Formal description

deref = λk e s . isLoc e→cont k e s, k e s

5.7.5. err:Cc

Informal description

err is the error continuation.

Formal description

err = λs . error

5.7.6. Domain checks: D?:Ec→Ec

Informal description

For each summand **D** of **Ev** we define a function **D?** which checks whether an element is in **D** and produces an error if not. **D? k e s** passes **e** and **s** to **k** if **e** is in **D** and is **error** otherwise.

Formal description

D? = λk e . isD e→k e, err

Examples

If Ev = Num + Bool + ... then
Num? = λk e . isNum e→k e, err
Bool? = λk e . isBool e→k e, err

5.8. Assignment and L and R values

The meaning of an assignment **I := E** in TINY is the function which takes a state and then updates the memory component at **I** with **E**'s value. In standard semantics—where instead of states we have environments and stores—**E** is evaluated and its value stored in the location, ι say, denoted by **I** in the environment (if **I** does not denote a location then an error results). The assignment only changes the contents of ι in the store; the binding of **I** to ι in the environment is unaffected.

Suppose I_1 and I_2 denote locations ι_1 and ι_2 in the environment then what is the effect of $I_1 := I_2$? There are two obvious possibilities:

(i) Location ι_2 is stored in location ι_1.
(ii) The *contents* of location ι_2 is stored in location ι_1.

In both ALGOL 60 and PASCAL (ii) describes what happens (in PASCAL the effect of (i) can be obtained using pointers—see 9.1.).

Thus in standard semantics we shall assume that expressions occurring on the *right* of assignments have their values *dereferenced*—i.e. have their value looked up in the store if they are locations.

On the *left* of assignments, however, we need a location, not its contents, since it is a location we update. The dereferencing that is necessary on the right hand side can be done with the function **deref** defined in 5.7.4. If **k** is a continuation expecting a dereferenced value (e.g. the 'rest of the program' following an expression on the right of an assignment) then **deref k** is a continuation which when sent an undereferenced value will first dereference it and then send the result to **k**. The standard semantics of **I := E** is thus:

```
C[I := E] r c s =
  E[I] r k, s
  where k, = λe, s, . isLoc e,→E[E] r k, s, . error
    where k, = λe, s, . deref k, e, s,
      where k, = λe, s, . update e, c e, s,
```

We evaluate **I** and pass the results to k_1. k_1 takes the results, e_1 and s_1, of **I**'s evaluation, checks e_1 is a location and then evaluates **E** and passes the results to k_2. k_2 takes the results, e_2 and s_2, of **E**, dereferences e_2 and passes the results to k_3. k_3 takes the results, e_3 and s_3, of dereferencing e_2, updates the location e_1 with contents e_3 and then passes the resulting store to **c**. This semantic clause can be simplified. First notice that:

$$k_3 = \text{update}\, e, c$$

hence $k_2 = \text{deref}\, k_3$
$= \text{deref;update}\, e_1;c$

hence $k_1 = \lambda e_1 .\ \text{isLoc}\, e_1 \rightarrow E[E]\, r;\text{deref;update}\, e_1;c, \text{err}$
$= \lambda e_1 .\ \text{Loc?}(\lambda \iota .\ E[E]\, r;\text{deref;update}\, \iota;c)e_1$
$= \text{Loc?}\, \lambda \iota .\ E[E]\, r;\text{deref;update}\, \iota;c$

and thus:

$$C[I := E]\, r\, c = E[I]\, r;\text{Loc?}\, \lambda \iota .\ E[E]\, r;\text{deref;update}\, \iota;c$$

5.8.1. L and R values

To distinguish the different values extracted from expressions during assignments the following terminology is often used:

(i) The value of an expression needed on the *left* of an assignment is the expression's *L-value*. This value is a location and is obtained by the semantic function **E** without any dereferencing.
(ii) The value of an expression needed on the right of an assignment is the expression's *R-value*. This value is (Normally) obtained by dereferencing the value obtained with **E**.

There are often other contexts besides the right hand sides of assignments in which we need to dereference expression values. For example in

output E, if **E**'s value is a location (e.g. **E** = **I**), then we should output its contents. Rather than continually write **E[E] r;deref** it is traditional to define a new semantic function **R** for obtaining R-values.

R:Exp→Env→Ec→Cc

and is defined by: **R[E] r k = E[E] r (deref k)**

A function **L:Exp→Env→Ec→Cc** defined by **L[E] r k = E[E] r (Loc? k)** and which obtains L-values is also sometimes used. Using **E** and **R** the semantic clause for assignments becomes simply:

$$C[I := E]\, r\, c = L[I]\, r\, \lambda \iota .\ R[E]\, r;\text{update}\, \iota\, c$$

In some languages only a subset of the expressible values are allowed as the contents of locations. For example in PASCAL procedures are expressible but cannot be assigned to variables. The storable subset of **Ev** is the domain **Rv** of R-values, and it is convenient to make the semantic function **R** check that it only produces R-values. Thus, in this case, we define:

$$R[E]\, r\, k = E[E]\, r;\text{deref;Rv?;}k$$

5.9. Procedures and functions

In this section we outline the standard semantics of declarations and calls of procedures and functions. Note that by "functions" we mean the programming concept of a function—see the discussion at the beginning of 3.4.

5.9.1. Procedures

A procedure, declared by **proc I(I,):C** say, has *name* **I**, *formal parameter* **I**, and *body* **C**. Within the scope of its declaration it can be *called*, for example by a command of the form **I(E)**. The effect of this is to execute the body **C** in an environment identical to the environment when the procedure was declared except that the formal parameter **I**, denotes **E**'s value. The evaluation of procedure bodies in an environment derived from the *declaration time* environment is called *static binding*. Other kinds of binding using other environments are possible (for example *dynamic binding*—found in LISP and POP-2—uses the *call time* environment). The

semantics and merits of different kinds of binding are discussed in detail in 8.2.; for the time being we stick to static binding since it is the one used in ALGOL 60 and PASCAL. The expression **E** in a call **I(E)** is called the *actual parameter*—thus when a procedure is called the actual parameter value is bound to the formal parameter. The semantics of a procedure declaration is to bind some 'procedure value' to the procedure name in the environment, thus:

$$\mathbf{D[proc\ I(I_1);C]\,r\,u = u(p/I)}$$

where **p** is a procedure value modelling a procedure with formal parameter **I**, and body **C**.

Procedure values are members of the domain **Proc** defined by:

$$\mathbf{Proc = Cc \to Ev \to Store \to Ans}$$

We use **p**, **p′**, $\mathbf{p_1}$, $\mathbf{p_2}$ etc. to range over **Proc**. The definition of **Proc** can be simplified to:

$$\mathbf{Proc = Cc \to Ec}$$

The intuitive idea is that if $\mathbf{p} \in \mathbf{Proc}$ then:

$\mathbf{p\,c\,e\,s = c\,s'}$ where **s′** is the store resulting from executing **p**'s body with actual parameter **e** and store **s**.

The continuation **c** passed to procedure values is analogous to the 'return address' in an implementation—it enables the procedure to return to the rest of the program following its call when execution of its body has finished. The complete semantic clause for procedure declarations can now be given:

$$\mathbf{D[proc\ I(I_1);C]\,r\,u = u(p/I)}$$
$$\mathbf{where\ p = \lambda c\,e\,.\,C[C]\,r[e/I_1]\,c}$$

Notice that the environment $\mathbf{r[e/I_1]}$ in which the body **C** is evaluated is derived from the declaration time environment **r**—i.e. we have static binding. Notice also that for this equation to make sense **p**'s must be denotable values—i.e. **Dv** = ... + **Proc** + ...

The semantics of a procedure call is given by:

$$\mathbf{C[I(E)]\,r\,c\,s =}$$
$$\mathbf{E[I]\,r(\lambda e_1\,s_1\,.\,IsProc\,e_1 \to E[E]\,r(\lambda e_2\,s_2\,.\,e_1\,c\,e_2)s_1,\ error)s}$$

We evaluate **I** to get $\mathbf{e_1}$ and $\mathbf{s_1}$ and check that $\mathbf{e_1}$ is a procedure value. If it is not then an error occurs. If it is we evaluate **E** in $\mathbf{s_1}$ to get $\mathbf{e_2}$ and $\mathbf{s_2}$ and then apply $\mathbf{e_1}$ to 'return address' **c** and actual parameter value $\mathbf{e_2}$. This equation can be written more compactly as:

$$\mathbf{C[I(E)]\,r\,c = E[I]\,r;Proc?\,\lambda p\,.\,E[E]\,r;p\,c}$$

which can be 'read' as: "evaluate **I**, check the result is in **Proc**, if so call the result **p** and evaluate **E** and pass its value to **p c**".

5.9.2. Functions

In many languages one can declare 'functions' which, when called, return a value as a result of the call. For example if **fact** is such a function then **fact(n)** is an expression and things like **output fact(n)** make sense. Note that these 'functions' must be carefully distinguished from the mathematical functions they denote. In both ALGOL 60 and PASCAL the result of a function is the last value assigned to the function name in its body. For example in PASCAL a function to square an integer **n** would be declared thus:

$$\mathbf{function\ square\ (n:integer):integer;square := n \times n}$$

Since function calls are expressions the continuation passed to a function value must be an expression continuation. Thus we define the domain **Fun** of function values by:

$$\mathbf{Fun = Ec \to Ec}$$

We use **f**, **f′**, $\mathbf{f_1}$, $\mathbf{f_2}$ etc. to range over **Fun**. The semantics of function calls is:

$$\mathbf{E[I(E)]\,r\,k = E[I]\,r;Fun?\,\lambda f\,.\,E[E]\,r;f\,k}$$

which can be understood by analogy with the clause for procedure calls which it closely resembles. Note that for this to make sense **f**'s must be denotable.

In SMALL (see next chapter) function declarations are of the form **fun I(I₁);E**. Thus the body of a function is an *expression* which, when

evaluated, yields the value to be returned. This way of yielding values is much cleaner than assigning to the function's name; we discuss some of the problems of that in 8.2.1. The semantic clause for function declarations is similar to the one for procedures:

$$\mathbf{D[fun}\,I(I_,);E]\,r\,u = u(f/I)$$
$$\text{where } f = \lambda k\,e\,.\,E[E]\,r[e/I_,]\,k$$

5.9.3. Summary

Domains: $\mathbf{Proc} = \mathbf{Cc} \to \mathbf{Ec}$ – procedure values **p**
$\mathbf{Fun} = \mathbf{Ec} \to \mathbf{Ec}$ – function values **f**

Clauses for declarations:

$$\mathbf{D[proc}\,I(I_,);C]\,r\,u = u((\lambda c\,e\,.\,C[C]\,r[e/I_,]\,c)/I)$$
$$\mathbf{D[fun}\,I(I_,);E]\,r\,u = u((\lambda k\,e\,.\,E[E]\,r[e/I_,]\,k)/I)$$

Clauses for calls:

$$\mathbf{C[}I(E)\mathbf{]}\,r\,c = E[I]\,r;\mathbf{Proc?}\,\lambda p\,.\,E[E]\,r;p\,c$$
$$\mathbf{E[}I(E)\mathbf{]}\,r\,k = E[I]\,r;\mathbf{Fun?}\,\lambda f\,.\,E[E]\,r;f\,k$$

5.10. Non standard semantics and concurrency

Standard semantics is based on the idea that the meaning of a construct is its effect on the final answer of programs in which it occurs. This implies, for example, that **x := 2** and **x := 1;x := x+1** have the same meaning. For most purposes this is exactly what we want, but not always. Consider a language containing a construct **cobegin** $\mathbf{C_1, C_2}$ **coend** whose meaning is that $\mathbf{C_1}$ and $\mathbf{C_2}$ are executed *concurrently* until they both terminate. In such a language **x := 2** and **x := 1;x := x+1** must have different denotations because, for example, **cobegin x := 2, x := 2 coend** can have different effects than **cobegin x := 2, x := 1;x := x+1 coend**. In the second case the **x := 2** might be done after the **x := 1** but before the **x := x+1** resulting in **x** denoting **3**. The only result of
cobegin x := 2, x := 2 coend is for **x** to denote **2**. If we had
C[x := 2] = C[x := 1;x := x+1] then since the denotation of a construct must depend only on the denotations of its immediate constituents; no matter how we defined **C[cobegin** $\mathbf{C_1, C_2}$ **coend]** we would have:

$$\begin{aligned}\mathbf{C[cobegin}\ x := 2, x := 1;x := x+1\ \mathbf{coend]} &= \ldots C[x := 2]\ldots C[x := 1;x := x+1]\ldots\\ &= \ldots C[x := 2]\ldots C[x := 2]\ldots\\ &= \mathbf{C[cobegin}\ x := 2, x := 2\ \mathbf{coend]}\end{aligned}$$

To enable a semantics of **cobegin** $\mathbf{C_1, C_2}$ **coend** to be given each command must denote some sort of 'process' in which the 'interleavable operations' are modelled (see [Milner] for example); in such cases standard semantics is not apporpriate. Non-standard semantics are also useful for modelling implementations, see [Milne and Strachey].

6. A second example: the Language SMALL

The language SMALL described in this chapter has two roles:

(i) To illustrate the main ideas of standard semantics.
(ii) To provide a 'kernel language' which we shall extend as we discuss the semantics of various constructs in later chapters.

6.1. Syntax of SMALL

6.1.1. Syntactic domains

The *primitive* syntactic domains of SMALL are:

Ide	The domain of identifiers **I**
Bas	The domain of basic constants **B**
Opr	The domain of binary operators **O**

We do not further specify these primitive domains. **Bas** could, for example, be {**0, 1, 2,** ...} and **Opr** could be {+, -, ×, /, <, >, ...}.

The *compound* syntactic domains of SMALL are:

Pro	The domain of programs **P**
Exp	The domain of expressions **E**
Com	The domain of commands **C**
Dec	The domain of declarations **D**

These domains are defined by the following syntactic clauses:

6.1.2. Syntactic clauses

P ::= program C

E ::= B | true | false | read | I | $E_1(E_2)$
| if E then E_1 else E_2 | E_1 O E_2

C ::= E_1 := E_2 | output E | $E_1(E_2)$ | if E then C_1 else C_2
| while E do C | begin D;C end | C_1;C_2

D ::= const I = E | var I = E | proc I(I,);C | fun I(I,);E | D_1;D_2

We have procedure (and function) calls of the form $E_1(E_2)$ rather than just I(E) to permit calls like **(if E_1 then I_1 else I_2)(E_2)** in which either the procedure (or function) denoted by I_1 or the one denoted by I_2 is applied to E_2 depending on the value of E_1. Similarly we permit assignments of the form **(if E_1 then I_1 else I_2) := E_2** or even **I(E_1) := E_2** (in which the location assigned to is the result of the function call I(E_1)).

6.2. Semantics of SMALL

6.2.1. Semantic domains

The primitive semantic domains of SMALL are:

Num	The domain of numbers **n**.
Bool	The domain of bodeans **b**.
Loc	The domain of locations **ι**.
Bv	The domain of basic values **e**.

Basic values are the denotations of basic constants. If **Bas** = {**0, 1, 2,** ...} then **Bv** = {**0, 1, 2,** ...} – however for simplicity we shall not specify any particular choice of basic values or constants.

The compound semantic domains are defined by the following domain equations:

Dv	**= Loc + Rv + Proc + Fun**	– denotable values **d**
Sv	**= File + Rv**	– storable values **v**
Ev	**= Dv**	– expressible values **e**
Rv	**= Bool + Bv**	– R-values **e**
File	**= Rv***	– files **i**
Env	**= Ide→[Dv + {unbound}]**	– environments **r**
Store	**= Loc→[Sv + {unused}]**	– stores **s**
Cc	**= Store→Ans**	– command continuations **c**
Ec	**= Ev→Cc**	– expression continuations **k**
Dc	**= Env→Cc**	– declaration continuations **u**
Proc	**= Cc→Ec**	– procedure values **p**
Fun	**= Ec→Ec**	– function values **f**
Ans	**= {error, stop} + [Rv × Ans]**	– final answers **a**

We assume **Loc** contains a location **Input** which holds the input file. We

have not provided SMALL with any facilities for creating new files, this is discussed in 9.5. For the time being the only file around is the input.

6.2.2. Semantic functions

We assume as given:

$$\mathbf{B}:\mathbf{Bas}\to\mathbf{Bv}$$
$$\mathbf{O}:\mathbf{Opr}\to[\mathbf{Rv}\times\mathbf{Rv}]\to\mathbf{Ec}\to\mathbf{Cc}$$

For example if **Bas** = {**0**, **1**, **2**, ...}, **Opr** = {+, ×, -, /, ...} and **Bv** = **Num** then we might have **B[0]** = **0**, **O[+](1, 2)k** = **k 3**, **O[/](1, 0)k** = **err** etc.

The meaning of the non-basic constructs are given by:

$$\mathbf{P}:\mathbf{Pro}\to\mathbf{File}\to\mathbf{Ans}$$
$$\mathbf{R}:\mathbf{Exp}\to\mathbf{Env}\to\mathbf{Ec}\to\mathbf{Cc}$$
$$\mathbf{E}:\mathbf{Exp}\to\mathbf{Env}\to\mathbf{Ec}\to\mathbf{Cc}$$
$$\mathbf{C}:\mathbf{Com}\to\mathbf{Env}\to\mathbf{Cc}\to\mathbf{Cc}$$
$$\mathbf{D}:\mathbf{Dec}\to\mathbf{Env}\to\mathbf{Dc}\to\mathbf{Cc}$$

These are defined by the following semantic clauses.

6.2.3. Semantic clauses

6.2.3.1. Programs

(P) **P[program C] i = C[C]( ) (λs . stop)(i/input)**

The final answer produced when **program C** is run with input **i** is obtained by evaluating the command **C** with an empty environment (**( ) = λl . unbound**), a store in which the only used location is **input** which has contents **i** (**(i/input) = λι . ι = input→i, unused**) and a continuation which when sent a store stops with final answer **stop**.

6.2.3.2. Expressions

(R) **R[E] r k = E[E] r;deref;Rv?;k**

E is evaluated, its result dereferenced and then checked to make sure it is an R-value and then passed to the rest of the program.

(E1) **E[B] r k = k(B[B])**

B[B] is passed to the rest of the program.

(E2) **E[true] r k = k true , E[false] r k = k false**

The appropriate boolean value is passed to the rest of the program.

(E3) **E[read] r k s =**
null(s input)→error, k(hd(s input))(s[tl(s input)/input])

If the input file (i.e. **s input**) is empty then an error occurs, otherwise the first item on the input (**hd(s input)**) is sent to the rest of the program **k** together with a store in which the item first read has been removed from the input file (**s[tl(s input)/input]**).

(E4) **E[I] r k = (r I = unbound)→err, k(r I)**

If **I** is unbound an error occurs, otherwise the value denoted by **I** in the environment is sent to the rest of the program.

(E5) **E[E₁(E₂)] r k = E[E₁] r;Fun?λf . E[E₂] r;f;k**

$\mathbf{E}_1$ is evaluated and its value checked to ensure it is a function **f**, $\mathbf{E}_2$ is then evaluated and its value passed to **f** and finally the result of **f** is passed to the rest of the program **k**.

(E6) **E[if E then E₁ else E₂] r k =**
R[E] r;Bool?;cond(E[E₁] r k, E[E₂] r k)

E is evaluated for its R-value which is then checked to ensure it is a boolean, then $\mathbf{E}_1$ or $\mathbf{E}_2$ are evaluated depending on whether **E**'s value was **true** or **false**.

(E7) **E[E₁ O E₂] r k =**
R[E₁] r λe₁ . R[E₂] r λe₂ . O[O](e₁, e₂)k

$\mathbf{E}_1$ is evaluated for its R-value $\mathbf{e}_1$, then $\mathbf{E}_2$ is evaluated for its R-value $\mathbf{e}_2$ and finally the result of doing **O** to $\mathbf{e}_1$ and $\mathbf{e}_2$ are sent to the rest of the program **k**.

6.2.3.3. Commands

(C1) **C[E₁ := E₂] r c =**
E[E₁] r;Loc? λι . R[E₂] r;update ι;c

E_1 is evaluated for its L-value ι, then E_2 is evaluated for its R-value which is then stored in ι and the resulting store passed to the rest of the program **c**.

(C2) $\mathbf{C[output\ E]\,r\,c} =$
$\mathbf{R[E]\,r\lambda e\,s\,.\,(e,\,c\,s)}$

E is evaluated for its R-value **e** and new store **s**, then **e** is put onto the front of the answer produced when the rest of the program **c** is passed **s**.

(C3) $\mathbf{C[E_1(E_2)]\,r\,c} =$
$\mathbf{E[E_1]\,r;Proc?\lambda p\,.\,E[E_2]\,r;p;c}$

E_1 is evaluated and its value checked to ensure it is a procedure **p**, E_2 is then evaluated and its value passed to **p** and finally the store resulting from **p** is passed to the rest of the program **c**.

(C4) $\mathbf{C[if\ E\ then\ C_1\ else\ C_2]\,r\,c} =$
$\mathbf{R[E]\,r;Bool?;cond(C[C_1]\,r\,c,C[C_2]\,r\,c)}$

E is evaluated for its R-value which is then checked to ensure it is a boolean, then C_1 or C_2 are evaluated depending on whether E's value was **true** or **false**.

(C5) $\mathbf{C[while\ E\ do\ C]\,r\,c} =$
$\mathbf{R[E]\,r;Bool?;cond(C[C]\,r;C[while\ E\ do\ C]\,r\,c\,,\,c)}$

E is evaluated for its R-value which is then checked to ensure it is a boolean, then if **E**'s value is **true** **C** is evaluated and the resulting store passed to the beginning of **while E do C** again. If **E**'s value is **false** then the rest of the program is immediately done.

(C6) $\mathbf{C[begin\ D;C\ end]\,r\,c = D[D]\,r\lambda r'\,.\,C[C]\,r[r']\,c}$

D is evaluated to get a little environment $\mathbf{r'}$, then **C** is evaluated in **r** updated with $\mathbf{r'}$ and then the rest of the program **c** is done.

(C7) $\mathbf{C[C_1;C_2]\,r\,c = C[C_1]\,r;C[C_2]\,r;c}$

C_1 is done, then C_2 is done and then the rest of the program **c** is done.

6.2.3. Declarations

(D1) $\mathbf{D[const\ I = E]\,r\,u = R[E]\,r\lambda e\,.\,u(e/I)}$

E is evaluated for its R-value **e** and then the little environment **e/I** is passed to the rest of the program **u**.

(D2) $\mathbf{D[var\ I = E]\,r\,u = R[E]\,r;ref\lambda\iota\,.\,u(\iota/I)}$

E is evaluated for its R-value which is then stored in a new location ι and the little environment ι/I passed to the rest of the program **u**.

(D3) $\mathbf{D[proc\ I(I_1);C]\,r\,u = u((\lambda c\,e\,.\,C[C]\,r[e/I_1]\,c)/I)}$

A little environment in which the procedure value $\lambda c\,e\,.\,C[C]\,r[e/I_1]\,c$ is bound to **I** is passed to the rest of the program **u**. When this procedure value is called the body **C** is evaluated in an environment got from the declaration time environment **r** by binding the actual parameter **e** to the formal parameter I_1.

(D4) $\mathbf{D[fun\ I(I_1);E]\,r\,u = u((\lambda k\,e\,.\,E[E]\,r[e/I_1]\,k)/I)}$

A little environment in which the function value $\lambda k\,e\,.\,E[E]\,r[e/I_1]\,k$ is bound to **I** is passed to the rest of the program **u**.

(D5) $\mathbf{D[D_1;D_2]\,r\,u = D[D_1]\,r\lambda r_1\,.\,D[D_2]\,r[r_1]\lambda r_2\,.\,u[r_1[r_2]]}$

D_1 is evaluated in **r** to get a little environment r_1, then D_2 is evaluated in $r[r_1]$ (i.e. **r** updated with the bindings in r_1) to get another little environment r_2 and then finally $r_1[r_2]$ (the bindings from D_1 updated with those of D_2) are passed to the rest of the program **u**. Notice that D_2 is evaluated in an environment which contains the effects of D_1 and that if D_1 and D_2 contain declarations of the same identifier then the result of $D_1;D_2$ is the result of D_2. For example **const x = 1;const x = x+1** would bind **x** to **2**.

6.3. A worked example

To illustrate how the semantics of SMALL works we show how to use it to 'evaluate' **program begin var x = read;output x end** with respect to an initial input file **i** such that **null I = false** and **hd i = 1** (for example **I = 1.2.3...**). We show as expected that

$\mathbf{P[program\ begin\ var\ x = read;output\ x\ end]\,I = (1,\,stop)}$
i.e. $\mathbf{C[begin\ var\ x = read;output\ x\ end]()(\lambda s\,.\,stop)(i/input) =}$
$\mathbf{(1,\,stop)}$

We do the calculation in several stages, first let $\mathbf{r} = ()$, $\mathbf{s}_1 = (\mathbf{i}/\mathbf{input})$, $\mathbf{s}_2 = \mathbf{s}_1[\mathbf{tl}\,\mathbf{i}/\mathbf{input}] = (\mathbf{tl}\,\mathbf{i}/\mathbf{input})$, assume $\mathbf{t}_2 = \mathbf{new}\,\mathbf{s}_2$ and finally let $\mathbf{s}_3 = \mathbf{s}_2[1/\mathbf{t}_2]$. Then:

(i) $\mathbf{E[read]\,r\,k\,s}_1$
$= \mathbf{null}(\mathbf{s}_1\,\mathbf{input})\rightarrow\mathbf{error},\,\mathbf{k}(\mathbf{hd}(\mathbf{s}_1\,\mathbf{input}))(\mathbf{s}_1[\mathbf{tl}(\mathbf{s}_1\,\mathbf{input})/\mathbf{input}])$ (E3)
$= \mathbf{null}\,\mathbf{i}\rightarrow\mathbf{error},\,\mathbf{k}\,1\,(\mathbf{s}_1[\mathbf{tl}\,\mathbf{i}/\mathbf{input}])$ ($\mathbf{s}_1\,\mathbf{input} = \mathbf{i}$)
$= \mathbf{k}\,1\,\mathbf{s}_2$ ($\mathbf{null}\,\mathbf{i} = \mathbf{false}$)

(ii) $\mathbf{D[var\,x = read]\,r\,u\,s}_1$
$= (\mathbf{R[read]\,r;ref}\,\lambda\mathbf{t}\,.\,\mathbf{u}(\mathbf{t}/\mathbf{x}))\,\mathbf{s}_1$ (D2)
$= (\mathbf{E[read]\,r;deref;Rv?;ref}\,\lambda\mathbf{t}\,.\,\mathbf{u}(\mathbf{t}/\mathbf{x}))\,\mathbf{s}_1$ (R)
$= \mathbf{E[read]\,r}(\mathbf{deref}(\mathbf{Rv?}(\mathbf{ref}\,\lambda\mathbf{t}\,.\,\mathbf{u}(\mathbf{t}/\mathbf{x}))))\,\mathbf{s}_1$ (definition of ;)
$= \mathbf{deref}(\mathbf{Rv?}(\mathbf{ref}\,\lambda\mathbf{t}\,.\,\mathbf{u}(\mathbf{t}/\mathbf{x})))\,1\,\mathbf{s}_2$ (by (i) above)
$= \mathbf{isLoc}\,1\rightarrow\ldots,\mathbf{Rv?}(\mathbf{ref}\,\lambda\mathbf{t}\,.\,\mathbf{u}(\mathbf{t}/\mathbf{x}))\,1\,\mathbf{s}_2$ (definition of **deref**)
$= \mathbf{Rv?}(\mathbf{ref}\,\lambda\mathbf{t}\,.\,\mathbf{u}(\mathbf{t}/\mathbf{x}))\,1\,\mathbf{s}_2$ ($\mathbf{isLoc}\,1 = \mathbf{false}$)
$= \mathbf{isRv}\,1\rightarrow(\mathbf{ref}\,\lambda\mathbf{t}\,.\,\mathbf{u}(\mathbf{t}/\mathbf{x}))\,1\,\mathbf{s}_2,\ldots$ (definition of **Rv?**)
$= \mathbf{ref}\,(\lambda\mathbf{t}\,.\,\mathbf{u}(\mathbf{t}/\mathbf{x}))\,1\,\mathbf{s}_2$ ($\mathbf{isRv}\,1 = \mathbf{true}$)
$= \mathbf{new}\,\mathbf{s}_2 = \mathbf{error}\rightarrow\mathbf{error},$
$\quad\mathbf{update}\,(\mathbf{new}\,\mathbf{s}_2)((\lambda\mathbf{t}\,.\,\mathbf{u}(\mathbf{t}/\mathbf{x}))(\mathbf{new}\,\mathbf{s}_2))\,1\,\mathbf{s}_2$ (definition of **ref**)
$= \mathbf{update}\,\mathbf{t}_2\,(\mathbf{u}(\mathbf{t}_2/\mathbf{x}))\,1\,\mathbf{s}_2$ ($\mathbf{new}\,\mathbf{s}_2 = \mathbf{t}_2$)
$= \mathbf{isSv}\,1\rightarrow\mathbf{u}(\mathbf{t}_2/\mathbf{x})(\mathbf{s}_2[1/\mathbf{t}_2]),\,\mathbf{error}$ (definition of **update**)
$= \mathbf{u}(\mathbf{t}_2/\mathbf{x})(\mathbf{s}_2[1/\mathbf{t}_2])$ ($\mathbf{isSv}\,1 = \mathbf{true}$)
$= \mathbf{u}(\mathbf{t}_2/\mathbf{x})\mathbf{s}_3$ ($\mathbf{s}_3 = \mathbf{s}_2[1/\mathbf{t}_2]$)

(iii) $\mathbf{C[begin\,var\,x = read;output\,x\,end]\,r\,c\,s}_1$
$= \mathbf{D[var\,x = read]\,r}(\lambda\mathbf{r}'\,.\,\mathbf{C[output\,x]\,r}[\mathbf{r}']\,\mathbf{c})\,\mathbf{s}_1$ (C6)
$= (\lambda\mathbf{r}'\,.\,\mathbf{C[output\,x]\,r}[\mathbf{r}']\,\mathbf{c})\,(\mathbf{t}_2/\mathbf{x})\,\mathbf{s}_3$ (by (ii) above)
$= \mathbf{C[output\,x]\,r}[\mathbf{t}_2/\mathbf{x}]\,\mathbf{c}\,\mathbf{s}_3$
$= \mathbf{R[x]\,r}[\mathbf{t}_2/\mathbf{x}]\,(\lambda\mathbf{e}\,\mathbf{s}\,.\,(\mathbf{e},\mathbf{c}\,\mathbf{s}))\,\mathbf{s}_3$ (C2)
$= \mathbf{E[x]\,r}[\mathbf{t}_2/\mathbf{x}]\,(\mathbf{deref}(\mathbf{Rv?}\,\lambda\mathbf{e}\,\mathbf{s}\,.\,(\mathbf{e},\mathbf{c}\,\mathbf{s})))\,\mathbf{s}_3$ (R)
$= \mathbf{deref}(\mathbf{Rv?}\,\lambda\mathbf{e}\,\mathbf{s}\,.\,(\mathbf{e},\mathbf{c}\,\mathbf{s}))\,\mathbf{t}_2\,\mathbf{s}_3$ ((E4), $\mathbf{r}[\mathbf{t}_2/\mathbf{x}]\mathbf{x} = \mathbf{t}_2$)
$= \mathbf{isLoc}\,\mathbf{t}_2\rightarrow\mathbf{cont}(\mathbf{Rv?}\,\lambda\mathbf{e}\,\mathbf{s}\,.\,(\mathbf{e},\mathbf{c}\,\mathbf{s}))\,\mathbf{t}_2\,\mathbf{s}_3,\ldots$ (definition of **deref**)
$= \mathbf{cont}(\mathbf{Rv?}\,\lambda\mathbf{e}\,\mathbf{s}\,.\,(\mathbf{e},\mathbf{c}\,\mathbf{s}))\,\mathbf{t}_2\,\mathbf{s}_3$
$= (\mathbf{Rv?}\,\lambda\mathbf{e}\,\mathbf{s}\,.\,(\mathbf{e},\mathbf{c}\,\mathbf{s}))\,1\,\mathbf{s}_3$ (definition of **cont**, $\mathbf{isLoc}\,\mathbf{t}_2 = \mathbf{true}$ and $\mathbf{s}_3\,\mathbf{t}_2 = 1$)
$= (\lambda\mathbf{e}\,\mathbf{s}\,.\,(\mathbf{e},\mathbf{c}\,\mathbf{s}))\,1\,\mathbf{s}_3$ (definition of **Rv?**, $\mathbf{isRv}\,1 = \mathbf{true}$)
$= (1,\mathbf{c}\,\mathbf{s}_3)$

(iv) $\mathbf{P[program\,begin\,var\,x = read;output\,x\,end]\,i}$
$= \mathbf{C[begin\,var\,x = read;output\,x\,end]}()(\lambda\mathbf{s}\,.\,\mathbf{stop})(\mathbf{i}/\mathbf{input})$ (P)
$= (1,(\lambda\mathbf{s}\,.\,\mathbf{stop})\mathbf{s}_3)$ (by (iii) above)
$= (1,\mathbf{stop})$

Calculations like this, though very tedious, are purely mechanical. The Semantics Implementation System (SIS) of Peter Mosses [Mosses] is a computer system which, by automating such calculations, runs programs directly from a denotational semantics. Although the resulting 'implementation' is *very* inefficient it is nevertheless useful for 'debugging semantics' and as an aid to language designers. An implementation package (on DEC tapes) is currently available for PDP 10's from Peter Mosses at Aarhus University, Denmark.

Prentice Hall International
Series in Computer Science

C. A. R. Hoare, Series Editor

BACKHOUSE, R.C., *Program Construction and Verification*
BACKHOUSE, R.C., *Syntax of Programming Languages: Theory and practice*
DE BAKKER, J.W., *Mathematical Theory of Program Correctness*
BIRD, R., AND WADLER, P., *Introduction to Functional Programming*
BJÖRNER, D., AND JONES, C.B., *Formal Specification and Software Development*
BORNAT, R., *Programming from First Principles*
BUSTARD, D., ELDER, J., AND WELSH, J., *Concurrent Program Structures*
CLARK, K.L., AND MCCABE, F.G., *micro-Prolog: Programming in logic*
CROOKES, D., *Introduction to Programming in Prolog*
DROMEY, R.G., *How to Solve it by Computer*
DUNCAN, F., *Microprocessor Programming and Software Development*
ELDER, J., *Construction of Data Processing Software*
GOLDSCHLAGER, L., AND LISTER, A., *Computer Science: A modern introduction (2nd edn)*
GORDON, M.J.C., *Programming Language Theory and its Implementation*
HAYES, I. (ED.), *Specification Case Studies*
HEHNER, E.C.R., *The Logic of Programming*
HENDERSON, P., *Functional Programming. Application and implementation*
HOARE, C.A.R., *Communicating Sequential Processes*
HOARE, C.A.R., AND SHEPHERDSON, J.C. (EDS), *Mathematical Logic and Programming Languages*
HUGHES, J.G., *Database Technology: A software engineering approach*
INMOS LTD, *occam Programming Manual*
INMOS LTD, *occam 2 Reference Manual*
JACKSON, M.A., *System Development*
JOHNSTON, H., *Learning to Program*
JONES, C.B., *Systematic Software Development using VDM*
JONES, G., *Programming in occam*
JOSEPH, M., PRASAD, V.R., AND NATARAJAN, N., *A Multiprocessor Operating System*
LEW, A., *Computer Science: A mathematical introduction*
MACCALLUM, I., *Pascal for the Apple*
MACCALLUM, I., *UCSD Pascal for the IBM PC*
MEYER, B., *Object-oriented Software Construction*
PEYTON JONES, S.L., *The Implementation of Functional Programming Languages*
POMBERGER, G., *Software Engineering and Modula-2*
REYNOLDS, J.C., *The Craft of Programming*
RYDEHEARD, D.E., AND BURSTALL, R.M., *Computational Category Theory*
SLOMAN, M., AND KRAMER, J., *Distributed Systems and Computer Networks*
TENNENT, R.D., *Principles of Programming Languages*
WATT, D.A., WICHMANN, B.A., AND FINDLAY, W., *ADA: Language and methodology*
WELSH, J., AND ELDER, J., *Introduction to Modula-2*
WELSH, J., AND ELDER, J., *Introduction to Pascal (3rd edn)*
WELSH, J., ELDER, J., AND BUSTARD, D., *Sequential Program Structures*
WELSH, J., AND HAY, A., *A Model Implementation of Standard Pascal*
WELSH, J., AND MCKEAG, M., *Structured System Programming*
WIKSTROM, Å., *Functional Programming using Standard ML*

PROGRAMMING LANGUAGE THEORY AND ITS IMPLEMENTATION

Applicative and imperative paradigms

Michael J.C. Gordon
Computer Laboratory,
University of Cambridge
and
SRI International

PRENTICE HALL
NEW YORK LONDON TORONTO SYDNEY TOKYO

First published 1988 by
Prentice Hall International (UK) Ltd.
66 Wood Lane End, Hemel Hempstead,
Hertfordshire, HP2 4RG
A division of
Simon & Schuster International Group

Printed and bound in Great Britain by
A. Wheaton & Co. Ltd, Exeter.

Library of Congress Cataloging in Publication Data
Gordon, Michael J.C., 1948–
Programming language theory and its implementation.
(Prentice-Hall international series in computer science)
Bibliography: p.
Includes index.
1. Programming languages (Electronic computers)
I. Title. II. Series.
QA76.7.G673 1988 005.13 88-9791
ISBN 0-13-730417-X

British Library Cataloguing in Publication Data
Gordon, Michael J.C., 1948–
Programming language theory and its implementation. – (Prentice Hall International series in computer science)
1. Computer systems. Programming languages. Theories
I. Title
005.1'01
ISBN 0-13-730417-X
ISBN 0-13-730409-9 Pbk

1 2 3 4 5 92 91 90 89 88

ISBN 0-13-730417-X
ISBN 0-13-730409-9 PBK

To Avra

Contents

Preface

Formal methods are becoming an increasingly important part of the design of computer systems. This book provides elementary introductions to two of the mathematical theories upon which these methods are based:

(i) Floyd-Hoare logic, a formal system for proving the correctness of imperative programs.

(ii) The λ-calculus and combinators, the mathematical theory underlying functional programming.

The book is organised so that (i) and (ii) can be studied independently. The two theories are illustrated with working programs written in LISP, for which an introduction is provided. It is hoped that the programs will both clarify the theoretical material and show how it can be applied in practice. They also provide a starting point for student programming projects in some interesting and rapidly expanding areas of non-numerical computation.

Floyd-Hoare logic is a theory of reasoning about programs that are written in conventional *imperative* programming languages like Fortran, Algol, Pascal, Ada, Modula-2 etc. Imperative programming consists in writing commands that modify the state of a machine. Floyd-Hoare logic can be used to establish the correctness of programs already written, or (better) as the foundation for rigorous software development methodologies in which programs are constructed in parallel with their verifications (an example of such a methodology is VDM [37]). Furthermore, thinking about the logical properties of programs provides a useful perspective, even if one is not going to verify them in detail. Floyd-Hoare logic has influenced the design of several programming languages, notably Euclid [46]. It is also the basis for *axiomatic semantics*, in which the meaning of a programming language is specified by requiring that all programs written in it satisfy the rules and axioms of a formal logic. Since Hoare's first paper [32] was published, there have been many reformulations of his ideas, e.g. by Dijkstra [16]. We do not describe these developments here, partly because Hoare's original formulation is still the simplest and (in my opinion) the best to learn first, and partly because it is the basis for commercial program verifiers (e.g. Gypsy [20]). The principles underlying such verifiers are described in Chapter 3

and the implementation of an example system is presented in Chapter 11. Good introductions to the recent developments in verification theory are the books by Gries [26] and Backhouse [3].

The λ-calculus is a theory of *higher-order* functions, i.e. functions that take functions as arguments or return functions as results. It has inspired the design of *functional* programming languages including LISP [53], ML [55], Miranda [70] and Ponder [17]. These languages provide notations for defining functions that are based directly on the λ-calculus; they differ in the 'mathematical purity' of the other features provided. Closely related to the λ-calculus is the theory of combinators. This provides an elegant 'machine code' into which functional languages can be compiled and which is simple to interpret by firmware or hardware. It is straightforward to prove the correctness of the algorithm for compiling functional programs to combinators (see Section 8.2); proving the correctness of compiling algorithms for imperative languages is usually extremely difficult [13,54,62].

Although functional programs execute more slowly than imperative ones on current computers, it is possible that this situation will be reversed in the future. Functional languages are well suited to exploit the multiprocessor architectures that are beginning to emerge from research laboratories.

Both the λ-calculus and the theory of combinators were originally developed as foundations for mathematics before digital computers were invented. They languished as obscure branches of mathematical logic until rediscovered by computer scientists. It is remarkable that a theory developed by logicians has inspired the design of both the hardware and software for a new generation of computers. There is an important lesson here for people who advocate reducing support for 'pure' research: the pure research of today defines the applied research of tomorrow.

If one is forced to use an imperative language (as most programmers are), then Floyd-Hoare logic provides a tool for establishing program correctness. However, many people feel that imperative programs are intrinsically difficult to reason about and that functional programming is a better basis for formal correctness analysis [6]. One reason for this is that functions are well-understood mathematical objects and thus do not call for a special logic; ordinary mathematics suffices. However, this view is not universally held; an eloquent case for the mathematical simplicity of imperative programming can be found in recent work by Dijkstra [16], Hehner [27] and Hoare [34]. Furthermore, the functions arising in functional programming are often unlike traditional mathematical functions, and special logics, e.g. LCF [25,49], have had to be devised for reasoning about them.

My own view is that both imperative and functional programming have their place, but that it is likely that functional languages will gradually replace imperative ones for general purpose use. This is already beginning

to happen; for example, ML replaced Pascal in 1987 as the first languag that computer science students are taught at Cambridge University. Ther is growing evidence that

(i) programmers can solve problems more quickly if they use a functiona language, and

(ii) the resulting solutions are more likely to be correct.

Because of (ii), and the relative ease of verifying the correctness of com pilers for functional languages, functional programming is likely to hav importance in safety-critical applications.

Parts I and II of this book provide an introduction to the theory under lying both imperative and functional programming. Part III contains som working programs which illustrate the material described in the first tw parts. Floyd-Hoare logic is illustrated by an implementation of a complet program verifier. This consists of a verification condition generator an a simple theorem prover. The λ-calculus and combinators are illustrate by a toolkit for experimenting with the interpretation and compilation c functional programs. The example systems in Part III are implemented i LISP and are short enough that the reader can easily type them in and ru them. A tutorial on the subset of LISP used is given in Chapter 9.

This book has evolved from the lecture notes for two undergraduat courses at Cambridge University which I have taught for the last few years It should be possible to cover all the material in about thirty hours. Ther are no mathematical prerequisites besides school algebra. Familiarity wit simple imperative programming (e.g. a first course in Pascal) would b useful, but is not essential.

I am indebted to the various students and colleagues at Cambridge wh have provided me with feedback on the courses mentioned above, and t Graham Birtwistle of the University of Calgary and Elsa Gunter of the Un versity of Pennsylvania, who pointed out errors in the notes and gave muc advice and help on revising them into book form. In addition, Graha Birtwistle provided detailed and penetrating comments on the final dra of the manuscript, as did Jan van Eijck of SRI International, Mike Fourma of Brunel University and Avra Cohn of Cambridge University. These fou people discovered various errors and made many helpful suggestions. Ma tin Hyland of Cambridge University explained to me a number of subt points concerning the relationship between the λ-calculus and the theor of combinators.

The preparation of camera-ready copy was completed with the suppo of a Royal Society/SERC Industrial Fellowship at the Cambridge Com puter Science Research Center of SRI International. The typesetting wa

done using LaTeX [41]. I am grateful to the versatile Elsa Gunter who, whilst simultaneously working as a researcher in the Cambridge Computer Laboratory and writing up her Ph.D. on group theory, was also employed by me to typeset this book. I would also like to thank Prentice Hall's copy editors for helping to counteract excesses of informality in my writing style.

Finally, this book would not exist without the encouragement and patience of Helen Martin, Prentice Hall's Acquisitions Editor, and Tony Hoare, the Series Editor.

M.J.C.G.
SRI International
Cambridge, England
April 8, 1988

Part I

Proving Programs Correct

Chapter 1

Program Specification

A simple programming language containing assignments, conditionals, blocks, WHILE-commands and FOR-commands is introduced. This language is then used to illustrate Hoare's notation for specifying the partial correctness of programs. Hoare's notation uses predicate calculus to express conditions on the values of program variables. A fragment of predicate calculus is introduced and illustrated with examples.

1.1 Introduction

In order to prove mathematically the correctness of a program one must first specify what it means for it to be correct. In this chapter a notation for specifying the desired behaviour of *imperative* programs is described. This notation is due to C.A.R. Hoare.

Executing an imperative program has the effect of changing the *state*, i.e. the values of program variables[1]. To use such a program, one first establishes an initial state by setting the values of some variables to values of interest. One then executes the program. This transforms the initial state into a final one. One then inspects (using print commands etc.) the values of variables in the final state to get the desired results. For example, to compute the result of dividing `y` into `x` one might load `x` and `y` into program variables `X` and `Y`, respectively. One might then execute a suitable program (see Example 7 in Section 1.4) to transform the initial state into a final state in which the variables `QUOT` and `REM` hold the quotient and remainder, respectively.

The programming language used in this book is described in the next section.

[1] For languages more complex than those described in this book, the state may consist of other things besides the values of variables [23].

1.2 A little programming language

Programs are built out of *commands* like assignments, conditionals etc. The terms 'program' and 'command' are really synonymous; the former will only be used for commands representing complete algorithms. Here the term 'statement' is used for conditions on program variables that occur in correctness specifications (see Section 1.3). There is a potential for confusion here because some writers use this word for commands (as in 'for-statement' [33]).

We now describe the *syntax* (i.e. form) and *semantics* (i.e. meaning) of the various commands in our little programming language. The following conventions are used:

1. The symbols V, V_1, ... , V_n stand for arbitrary variables. Examples of particular variables are `X`, `REM`, `QUOT` etc.
2. The symbols E, E_1, ... , E_n stand for arbitrary expressions (or terms). These are things like `X` + 1, $\sqrt{2}$ etc. which denote values (usually numbers).
3. The symbols S, S_1, ... , S_n stand for arbitrary statements. These are conditions like `X` < `Y`, `X`^{2} = 1 etc. which are either true or false.
4. The symbols C, C_1, ... , C_n stand for arbitrary commands of our programming language; these are described in the rest of this section.

Terms and statements are described in more detail in Section 1.5.

1.2.1 Assignments

Syntax: V := E

Semantics: The state is changed by assigning the value of the term E to the variable V.

Example: `X:=X+1`

This adds one to the value of the variable `X`.

1.2.2 Sequences

Syntax: C_1; ··· ;C_n

Semantics: The commands C_1, ··· , C_n are executed in that order.

Example: `R:=X; X:=Y; Y:=R`

The values of `X` and `Y` are swapped using `R` as a temporary variable. This command has the *side effect* of changing the value of the variable `R` to the old value of the variable `X`.

1.2.3 Blocks

Syntax: `BEGIN VAR` V_1; ··· `VAR` V_n; C `END`

Semantics: The command C is executed, and then the values of V_1, ··· , V_n are restored to the values they had before the block was entered. The initial values of V_1, ··· , V_n inside the block are unspecified.

Example: `BEGIN VAR R; R:=X; X:=Y; Y:=R END`

The values of `X` and `Y` are swapped using `R` as a temporary variable. This command does *not* have a side effect on the variable `R`.

1.2.4 One-armed conditionals

Syntax: `IF` S `THEN` C

Semantics: If the statement S is true in the current state, then C is executed. If S is false, then nothing is done.

Example: `IF ¬(X=0) THEN R:= Y DIV X`

If the value `X` is not zero, then `R` is assigned the result of dividing the value of `Y` by the value of `X`.

1.2.5 Two-armed conditionals

Syntax: `IF` S `THEN` C_1 `ELSE` C_2

Semantics: If the statement S is true in the current state, then C_1 is executed. If S is false, then C_2 is executed.

Example: `IF X<Y THEN MAX:=Y ELSE MAX:=X`

The value of the variable `MAX` it set to the maximum of the values of `X` and `Y`.

1.2.6 WHILE-commands

Syntax: `WHILE` S `DO` C

Semantics: If the statement S is true in the current state, then C is executed and the `WHILE`-command is then repeated. If S is false, then nothing is done. Thus C is repeatedly executed until the value of S becomes true. If S never becomes true, then the execution of the command never terminates.

Example: `WHILE` $\neg$`(X=0) DO X:= X-2`

If the value of `X` is non-zero, then its value is decreased by 2 and then process is repeated. This `WHILE`-command will terminate (with `X` having value `0`) if the value of `X` is an even non-negative number. In all other states it will not terminate.

1.2.7 FOR-commands

Syntax: `FOR` V`:=`E_1 `UNTIL` E_2 `DO` C

Semantics: If the values of terms E_1 and E_2 are positive numbers e_1 and e_2 respectively, and if $e_1 \leq e_2$, then C is executed $(e_2-e_1)+1$ times with the variable V taking on the sequence of values e_1, e_1+1, ..., e_2 in succession. For any other values, the `FOR`-command has no effect. A more precise description of this semantics is given in Section 2.1.11.

Example: `FOR N:=1 UNTIL M DO X:=X+N`

If the value of the variable `M` is m and $m \geq 1$, then the command `X:=X+N` is repeatedly executed with `N` taking the sequence of values `1`, ..., m. If $m < 1$ then the `FOR`-command does nothing.

1.2.8 Summary of syntax

The syntax of our little language can be summarized with the following specification in BNF notation[2]

[2] BNF stands for Backus-Naur form; it is a well-known notation for specifying syntax.


```
<command>
   ::= <variable>:=<term>
     | <command>; ... ;<command>
     | BEGIN VAR <variable>; ... VAR <variable>; <command> E
     | IF <statement> THEN <command>
     | IF <statement> THEN <command> ELSE <command>
     | WHILE <statement> DO <command>
     | FOR <variable>:=<term> UNTIL <term> DO <command>
```

Note that:

- *Variables*, *terms* and *statements* are as described in Section 1.5.
- Only declarations of the form '`VAR` <*variable*>' are needed. The types of variables need not be declared (unlike in Pascal).
- Sequences C_1; ... C_n are valid commands; they are equivalent to `BEGIN` C_1; ... C_n `END` (i.e. blocks without any local variables).
- The BNF syntax is ambiguous: it does not specify, for example, whether `IF` S_1 `THEN IF` S_2 `THEN` C_1 `ELSE` C_2 means

  `IF` S_1 `THEN (IF` S_2 `THEN` C_1 `ELSE` C_2`)`

  or

  `IF` S_1 `THEN (IF` S_2 `THEN` C_1`) ELSE` C_2

  We will clarify, whenever necessary, using brackets.

1.3 Hoare's notation

In a seminal paper [32] C.A.R. Hoare introduced the following notation for specifying what a program does[3]:

$$\{P\}\ C\ \{Q\}$$

where:

- C is a program from the programming language whose programs are being specified (the language in Section 1.2 in our case).

[3] Actually, Hoare's original notation was $P\ \{C\}\ Q$ not $\{P\}\ C\ \{Q\}$, but the latter form is now more widely used.

- P and Q are conditions on the program variables used in C.

Conditions on program variables will be written using standard mathematical notations together with *logical operators* like $\wedge$ ('and'), $\vee$ ('or'), $\neg$ ('not') and $\Rightarrow$ ('implies'). These are described further in Section 1.5.

We say $\{P\}\ C\ \{Q\}$ is true, if whenever C is executed in a state satisfying P and if the execution of C terminates, then the state in which C's execution terminates satisfies Q.

Example: $\{\mathtt{X} = 1\}$ `X:=X+1` $\{\mathtt{X} = 2\}$. Here P is the condition that the value of `X` is 1, Q is the condition that the value of `X` is 2 and C is the assignment command `X:=X+1` (i.e. '`X` becomes `X+1`'). $\{\mathtt{X} = 1\}$ `X:=X+1` $\{\mathtt{X} = 2\}$ is clearly true. □

An expression $\{P\}\ C\ \{Q\}$ is called a *partial correctness specification*; P is called its *precondition* and Q its *postcondition*.

These specifications are 'partial' because for $\{P\}\ C\ \{Q\}$ to be true it is *not* necessary for the execution of C to terminate when started in a state satisfying P. It is only required that *if* the execution terminates, *then* Q holds.

A stronger kind of specification is a *total correctness specification*. There is no standard notation for such specifications. We shall use $[P]\ C\ [Q]$.

A total correctness specification $[P]\ C\ [Q]$ is true if and only if the following conditions apply:

(i) Whenever C is executed in a state satisfying P, then the execution of C terminates.

(ii) After termination Q holds.

The relationship between partial and total correctness can be informally expressed by the equation:

$$\text{Total correctness} = \text{Termination} + \text{Partial correctness}.$$

Total correctness is what we are ultimately interested in, but it is usually easier to prove it by establishing partial correctness and termination separately.

Termination is often straightforward to establish, but there are some well-known examples where it is not. For example[4], no one knows whether the program below terminates for all values if `X`:

```
WHILE X>1 DO
  IF ODD(X) THEN X := (3×X)+1 ELSE X := X DIV 2
```

[4] This example is taken from Exercise 2 on page 17 of Reynolds's book [63].

(The expression `X DIV 2` evaluates to the result of rounding down `X/2` to a whole number.)

Exercise 1
Write a specification which is true if and only if the program above terminates. □

In Part I of this book Floyd-Hoare logic is described; this only deals with partial correctness. Theories of total correctness can be found in the texts by Dijkstra [16] and Gries [26].

1.4 Some examples

The examples below illustrate various aspects of partial correctness specification.

In Examples 5, 6 and 7 below, `T` (for 'true') is the condition that is always true. In Examples 3, 4 and 7, $\wedge$ is the logical operator 'and', ie. if P_1 and P_2 are conditions, then $P_1 \wedge P_2$ is the condition that is true whenever both P_1 and P_2 hold.

1. $\{\mathtt{X} = 1\}$ `Y:=X` $\{\mathtt{Y} = 1\}$

This says that if the command `Y:=X` is executed in a state satisfying the condition $\mathtt{X} = 1$ (i.e. a state in which the value of `X` is 1), then, if the execution terminates (which it does), then the condition $\mathtt{Y} = 1$ will hold. Clearly this specification is true.

2. $\{\mathtt{X} = 1\}$ `Y:=X` $\{\mathtt{Y} = 2\}$

This says that if the execution of `Y:=X` terminates when started in a state satisfying $\mathtt{X} = 1$, then $\mathtt{Y} = 2$ will hold. This is clearly false.

3. $\{\mathtt{X}=\mathtt{x} \wedge \mathtt{Y}=\mathtt{y}\}$ `BEGIN R:=X; X:=Y; Y:=R END` $\{\mathtt{X}=\mathtt{y} \wedge \mathtt{Y}=\mathtt{x}\}$

This says that if the execution of `BEGIN R:=X; X:=Y; Y:=R END` terminates (which it does), then the values of `X` and `Y` are exchanged. The variables `x` and `y`, which don't occur in the command and are used to name the initial values of program variables `X` and `Y`, are called *auxiliary* variables (or *ghost* variables).

4. $\{\mathtt{X}=\mathtt{x} \wedge \mathtt{Y}=\mathtt{y}\}$ `BEGIN X:=Y; Y:=X END` $\{\mathtt{X}=\mathtt{y} \wedge \mathtt{Y}=\mathtt{x}\}$

This says that `BEGIN X:=Y; Y:=X END` exchanges the values of `X` and `Y`. This is not true.

5. $\{\mathtt{T}\}\ C\ \{Q\}$

This says that whenever C halts, Q holds.

6. $\{P\}\ C\ \{\mathtt{T}\}$

This specification is true for every condition P and every command C (because T is always true).

7. {T}

```
BEGIN
  R:=X;
  Q:=0;
  WHILE Y≤R DO
    BEGIN R:=R-Y; Q:=Q+1 END
END
```

$\}\ C$

$\{\mathtt{R} < \mathtt{Y} \wedge \mathtt{X} = \mathtt{R} + (\mathtt{Y} \times \mathtt{Q})\}$

This is $\{\mathtt{T}\}\ C\ \{\mathtt{R} < \mathtt{Y} \wedge \mathtt{X} = \mathtt{R} + (\mathtt{Y} \times \mathtt{Q})\}$ where C is the command indicated by the braces above. The specification is true if whenever the execution of C halts, then Q is quotient and R is the remainder resulting from dividing Y into X. It is true (even if X is initially negative!)

In this example a program variable Q is used. This should not be confused with the Q used in 5 above. The program variable Q (notice the font) ranges over numbers, whereas the postcondition Q (notice the font) ranges over statements. In general, we use `typewriter font` for particular program variables and *italic font* for variables ranging over statements. Although this subtle use of fonts might appear confusing at first, once you get the hang of things the difference between the two kinds of 'Q' will be clear (indeed you should be able to disambiguate things from context without even having to look at the font).

Exercise 2
Let C be as in Example 7 above. Find a condition P such that:

$$[P]\ C\ [\mathtt{R} < \mathtt{Y} \wedge \mathtt{X} = \mathtt{R} + (\mathtt{Y} \times \mathtt{Q})]$$

is true. □

Exercise 3
When is $[\mathtt{T}]\ C\ [\mathtt{T}]$ true? □

Exercise 4
Write a partial correctness specification which is true if and only if the command C has the effect of multiplying the values of X and Y and storing the result in X. □

Exercise 5
Write a specification which is true if the execution of C always halts when execution is started in a state satisfying P. □

1.5 Terms and statements

The notation used here for expressing pre- and postconditions is base a language called *first-order logic* invented by logicians around the tu this century. For simplicity, only a fragment of this language will be u

Things like:

$$\mathtt{T}, \quad \mathtt{F}, \quad \mathtt{X} = 1, \quad \mathtt{R} < \mathtt{Y}, \quad \mathtt{X} = \mathtt{R}+(\mathtt{Y}\times\mathtt{Q})$$

are examples of *atomic statements*. Statements are either true or f The statement T is always true and the statement F is always false. statement $\mathtt{X} = 1$ is true if the value of X is equal to 1. The stater $\mathtt{R} < \mathtt{Y}$ is true if the value of R is less than the value of Y. The staten $\mathtt{X} = \mathtt{R}+(\mathtt{Y}\times\mathtt{Q})$ is true if the value of X is equal to the sum of the value with the product of Y and Q.

Statements are built out of *terms* like:

$$\mathtt{X}, \quad 1, \quad \mathtt{R}, \quad \mathtt{Y}, \quad \mathtt{R}+(\mathtt{Y}\times\mathtt{Q}), \quad \mathtt{Y}\times\mathtt{Q}$$

Terms denote *values* such as numbers and strings, unlike statements w are either true or false. Some terms, like 1 and 4 + 5, denote a fixed va whilst other terms contain *variables* like X, Y, Z etc. whose value can v We will use conventional mathematical notation for terms, as illustrate the examples below:

$$\mathtt{X}, \quad \mathtt{Y}, \quad \mathtt{Z},$$

$$1, \quad 2, \quad 325,$$

$$-\mathtt{X}, \quad -(\mathtt{X}+1), \quad (\mathtt{X}\times\mathtt{Y})+\mathtt{Z},$$

$$\sqrt{(1+\mathtt{X}^2)}, \quad \mathtt{X}!, \quad \sin(\mathtt{X}), \quad \mathrm{rem}(\mathtt{X},\mathtt{Y})$$

T and F are atomic statements that are always true and false respectiv Other atomic statements are built from terms using *predicates*. Here some more examples:

$$\mathrm{ODD}(\mathtt{X}), \quad \mathrm{PRIME}(3), \quad \mathtt{X} = 1, \quad (\mathtt{X}+1)^2 \geq \mathtt{X}^2$$

ODD and PRIME are examples of predicates and $=$ and $\geq$ are example *infixed* predicates. The expressions X, 1, 3, X+1, $(\mathtt{X}+1)^2$, $\mathtt{X}^2$ are exam of terms.

Compound statements are built up from atomic statements using following logical operators:

$\neg$	(not)
$\wedge$	(and)
$\vee$	(or)
$\Rightarrow$	(implies)
$\Leftrightarrow$	(if and only if)

The single arrow $\rightarrow$ is commonly used for implication instead of $\Rightarrow$. We use $\Rightarrow$ to avoid possible confusion with the the use of $\rightarrow$ for λ-conversion in Part II.

Suppose P and Q are statements, then:

- $\neg P$ is true if P is false, and false if P is true.
- $P \wedge Q$ is true whenever both P and Q are true.
- $P \vee Q$ is true if either P or Q (or both) are true.
- $P \Rightarrow Q$ is true if whenever P is true, then Q is true also. By convention we regard $P \Rightarrow Q$ as being true if P is false. In fact, it is common to regard $P \Rightarrow Q$ as equivalent to $\neg P \vee Q$; however, some philosophers called intuitionists disagree with this treatment of implication.
- $P \Leftrightarrow Q$ is true if P and Q are either both true or both false. In fact $P \Leftrightarrow Q$ is equivalent to $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

Examples of statements built using the connectives are:

`ODD(X)` $\vee$ `EVEN(X)`	`X` is odd or even.
$\neg$(`PRIME(X)` $\Rightarrow$ `ODD(X)`)	It is not the case that if `X` is prime, then `X` is odd.
`X` $\leq$ `Y` $\Rightarrow$ `X` $\leq$ `Y`^{2}	If `X` is less than or equal to `Y`, then `X` is less than or equal to `Y`^{2}.

To reduce the need for brackets it is assumed that $\neg$ is more binding than $\wedge$ and $\vee$, which in turn are more binding than $\Rightarrow$ and $\Leftrightarrow$. For example:

$\neg P \wedge Q$	is equivalent to	$(\neg P) \wedge Q$
$P \wedge Q \Rightarrow R$	is equivalent to	$(P \wedge Q) \Rightarrow R$
$P \wedge Q \Leftrightarrow \neg R \vee S$	is equivalent to	$(P \wedge Q) \Leftrightarrow ((\neg R) \vee S)$

Chapter 2

Floyd-Hoare Logic

The idea of formal proof is discussed. Floyd-Hoare logic is then introduced as a method for reasoning formally about programs.

In the last chapter three kinds of expressions that could be true or false were introduced:

(i) Partial correctness specifications $\{P\}\ C\ \{Q\}$.

(ii) Total correctness specifications $[P]\ C\ [Q]$.

(iii) Statements of mathematics (e.g. $(\mathtt{X}+1)^2 = \mathtt{X}^2 + 2 \times \mathtt{X} + 1$).

It is assumed that the reader knows how to prove simple mathematical statements like the one in (iii) above. Here, for example, is a proof of this fact.

1.	$(\mathtt{X}+1)^2$	$= (\mathtt{X}+1) \times (\mathtt{X}+1)$	Definition of $()^2$.
2.	$(\mathtt{X}+1) \times (\mathtt{X}+1)$	$= (\mathtt{X}+1) \times \mathtt{X} + (\mathtt{X}+1) \times 1$	Left distributive law of $\times$ over $+$.
3.	$(\mathtt{X}+1)^2$	$= (\mathtt{X}+1) \times \mathtt{X} + (\mathtt{X}+1) \times 1$	Substituting line 2 into line 1.
4.	$(\mathtt{X}+1) \times 1$	$= \mathtt{X}+1$	Identity law for 1.
5.	$(\mathtt{X}+1) \times \mathtt{X}$	$= \mathtt{X} \times \mathtt{X} + 1 \times \mathtt{X}$	Right distributive law of $\times$ over $+$.
6.	$(\mathtt{X}+1)^2$	$= \mathtt{X} \times \mathtt{X} + 1 \times \mathtt{X} + \mathtt{X} + 1$	Substituting lines 4 and 5 into line 3.
7.	$1 \times \mathtt{X}$	$= \mathtt{X}$	Identity law for 1.
8.	$(\mathtt{X}+1)^2$	$= \mathtt{X} \times \mathtt{X} + \mathtt{X} + \mathtt{X} + 1$	Substituting line 7 into line 6.
9.	$\mathtt{X} \times \mathtt{X}$	$= \mathtt{X}^2$	Definition of $()^2$.
10.	$\mathtt{X} + \mathtt{X}$	$= 2 \times \mathtt{X}$	2=1+1, distributive law.
11.	$(\mathtt{X}+1)^2$	$= \mathtt{X}^2 + 2 \times \mathtt{X} + 1$	Substituting lines 9 and 10 into line 8.

This proof consists of a sequence of lines, each of which is an instance of an *axiom* (like the definition of $()^2$) or follows from previous lines by a

rule of inference (like the substitution of equals for equals). The statement occurring on the last line of a proof is the statement *proved* by it (thus $(X+1)^2 = X^2 + 2 \times X + 1$ is proved by the proof above).

To construct formal proofs of partial correctness specifications axioms and rules of inference are needed. This is what Floyd-Hoare logic provides. The formulation of the deductive system is due to Hoare [32], but some of the underlying ideas originated with Floyd [18].

A proof in Floyd-Hoare logic is a sequence of lines, each of which is either an *axiom* of the logic or follows from earlier lines by a *rule of inference* of the logic.

The reason for constructing formal proofs is to try to ensure that only sound methods of deduction are used. With sound axioms and rules of inference, one can be confident that the conclusions are true. On the other hand, if any axioms or rules of inference are unsound then it may be possible to deduce false conclusions; for example[1]

1.	$\sqrt{-1 \times -1}$	$= \sqrt{-1 \times -1}$	Reflexivity of $=$.
2.	$\sqrt{-1 \times -1}$	$= (\sqrt{-1}) \times (\sqrt{-1})$	Distributive law of $\sqrt{}$ over $\times$.
3.	$\sqrt{-1 \times -1}$	$= (\sqrt{-1})^2$	Definition of $()^2$.
4.	$\sqrt{-1 \times -1}$	$= -1$	definition of $\sqrt{}$.
5.	$\sqrt{1}$	$= -1$	As $-1 \times -1 = 1$.
6.	1	$= -1$	As $\sqrt{1} = 1$.

A formal proof makes explicit what axioms and rules of inference are used to arrive at a conclusion. It is quite easy to come up with plausible rules for reasoning about programs that are actually unsound (some examples for FOR-commands can be found in Section 2.1.11). Proofs of correctness of computer programs are often very intricate and formal methods are needed to ensure that they are valid. It is thus important to make fully explicit the reasoning principles being used, so that their soundness can be analysed.

Exercise 6

Find the flaw in the 'proof' of $1 = -1$ above. □

For some applications, correctness is especially important. Examples include life-critical systems such as nuclear reactor controllers, car breaking systems, fly-by-wire aircraft and software controlled medical equipment. At the time of writing, there is a legal action in progress resulting from the death of several people due to radiation overdoses by a cancer treatment machine that had a software bug [38]. Formal proof of correctness provides a way of establishing the absence of bugs when exhaustive testing is impossible (as it almost always is).

[1] This example was shown to me by Sylva Cohn.

The Floyd-Hoare deductive system for reasoning about programs will be explained and illustrated, but the mathematical analysis of the soundness and completeness of the system is beyond the scope of this book (however, there is a brief discussion of what is involved in Section 2.2).

2.1 Axioms and rules of Floyd-Hoare logic

As discussed at the beginning of this chapter, a *formal proof* of a statement is a sequence of lines ending with the statement and such that each line is either an instance of an axiom or follows from previous lines by a rule of inference. If S is a statement (of either ordinary mathematics or Floyd-Hoare logic) then we write $\vdash S$ to mean that S has a proof. The statements that have proofs are called *theorems*. As discussed earlier, in this book only the axioms and rules of inference for Floyd-Hoare logic are described; we will thus simply assert $\vdash S$ if S is a theorem of mathematics without giving any formal justification. Of course, to achieve complete rigour such assertions must be proved, but for details of this the reader will have to consult a book (such as [10,47,49]) on formal logic.

The axioms of Floyd-Hoare logic are specified below by *schemas* which can be *instantiated* to get particular partial correctness specifications. The inference rules of Floyd-Hoare logic will be specified with a notation of the form:

$$\frac{\vdash S_1, \ldots, \vdash S_n}{\vdash S}$$

This means the *conclusion* $\vdash S$ may be deduced from the *hypotheses* $\vdash S_1, \ldots, \vdash S_n$. The hypotheses can either all be theorems of Floyd-Hoare logic (as in the sequencing rule below), or a mixture of theorems of Floyd-Hoare logic and theorems of mathematics (as in the rule of preconditioning strengthening described in Section 2.1.2).

2.1.1 The assignment axiom

The assignment axiom represents the fact that the value of a variable V *after* executing an assignment command $V := E$ equals the value of the expression E in the state *before* executing it. To formalize this, observe that if a statement P is to be true *after* the assignment, then the statement obtained by substituting E for V in P must be true *before* executing it.

In order to say this formally, define $P[E/V]$ to mean the result of replacing all occurrences of V in P by E. Read $P[E/V]$ as 'P with E for

V'. For example,

$$(\mathtt{X+1} > \mathtt{X})[\mathtt{Y+Z}/\mathtt{X}] \;=\; ((\mathtt{Y+Z})+1 > \mathtt{Y+Z})$$

The way to remember this notation is to remember the 'cancellation law'

$$V[E/V] \;=\; E$$

which is analogous to the cancellation property of fractions

$$v \times (e/v) \;=\; e$$

The assignment axiom

$$\vdash \; \{P[E/V]\}\ V{:=}E\ \{P\}$$

Where V is any variable, E is any expression, P is any statement and the notation $P[E/V]$ denotes the result of substituting the term E for all occurrences of the variable V in the statement P.

Instances of the assignment axiom are:

1. $\vdash \{\mathtt{Y} = 2\}\ \mathtt{X} := 2\ \{\mathtt{Y} = \mathtt{X}\}$
2. $\vdash \{\mathtt{X} + 1 = \mathtt{n} + 1\}\ \mathtt{X} := \mathtt{X} + 1\ \{\mathtt{X} = \mathtt{n} + 1\}$
3. $\vdash \{E = E\}\ \mathtt{X} := E\ \{\mathtt{X} = E\}$

Many people feel the assignment axiom is 'backwards' from what they would expect. Two common erroneous intuitions are that it should be as follows:

(i) $\vdash \{P\}\ V{:=}E\ \{P[V/E]\}$.

Where the notation $P[V/E]$ denotes the result of substituting V for E in P.

This has the clearly false consequence that $\vdash$ {x=0} x:=1 {x=0}, since the (x=0)[x/1] is equal to (x=0) as 1 doesn't occur in (x=0).

(ii) $\vdash \{P\}\ V{:=}E\ \{P[E/V]\}$.

This has the clearly false consequence $\vdash$ {x=0} x:=1 {1=0} which follows by taking P to be x=0, V to be x and E to be 1.

The fact that it is easy to have wrong intuitions about the assignment axiom shows that it is important to have rigorous means of establishing the validity of axioms and rules. We will not go into this topic here, aside from remarking that it is possible to give a *formal semantics* of our little programming language and then to *prove* that the axioms and rules of inference of Floyd-Hoare logic are sound. Of course, this process will only increase our confidence in the axioms and rules to the extent that we believe the correctness of the formal semantics. The simple assignment axiom above is not valid for 'real' programming languages. For example, work by G. Ligler [44] shows that it can fail to hold in six different ways for the language Algol 60.

One way that our little programming language differs from real languages is that the evaluation of expressions on the right of assignment commands cannot 'side effect' the state. The validity of the assignment axiom depends on this property. To see this, suppose that our language were extended so that it contained the 'block expression'

```
BEGIN Y:=1; 2 END
```

This expression, E say, has value 2, but its evaluation also 'side effects' the variable Y by storing 1 in it. If the assignment axiom applied to expressions like E, then it could be used to deduce:

$$\vdash \{\mathtt{Y=0}\}\ \mathtt{X:=BEGIN\ Y:=1;\ 2\ END}\ \{\mathtt{Y=0}\}$$

(since (Y=0)[E/X] = (Y=0) as X does not occur in (Y=0)). This is clearly false, as after the assignment Y will have the value 1.

2.1.2 Precondition strengthening

The next rule of Floyd-Hoare logic enables the preconditions of (i) and (ii) on page 16 to be simplified. Recall that

$$\frac{\vdash S_1, \ \ldots\ , \vdash S_n}{\vdash S}$$

means that $\vdash S$ can be deduced from $\vdash S_1, \ldots, \vdash S_n$.

Using this notation, the rule of precondition strengthening is

Precondition strengthening

$$\frac{\vdash P \Rightarrow P', \qquad \vdash \{P'\}\ C\ \{Q\}}{\vdash \{P\}\ C\ \{Q\}}$$

Examples

1. From the arithmetic fact $\vdash$ `X+1=n+1` $\Rightarrow$ `X=n`, and 2 on page 16 it follows by precondition strengthening that

$$\vdash \{\mathtt{X} = \mathtt{n}\}\ \mathtt{X} := \mathtt{X} + 1\ \{\mathtt{X} = \mathtt{n} + 1\}.$$

The variable `n` is an example of an *auxiliary* (or *ghost*) variable. As described earlier (see page 9), auxiliary variables are variables occurring in a partial correctness specification $\{P\}\ C\ \{Q\}$ which do not occur in the command C. Such variables are used to relate values in the state before and after C is executed. For example, the specification above says that if the value of `X` is `n`, then after executing the assignment `X:=X+1` its value will be `n+1`.

2. From the logical truth $\vdash$ `T` $\Rightarrow$ $(E{=}E)$, and 3 on page 16 one can deduce[2]:

$$\vdash \{\mathtt{T}\}\ \mathtt{X} := E\ \{\mathtt{X} = E\}$$

□

2.1.3 Postcondition weakening

Just as the previous rule allows the precondition of a partial correctness specification to be strengthened, the following one allows us to weaken the postcondition.

Postcondition weakening

$$\frac{\vdash \{P\}\ C\ \{Q'\}, \qquad \vdash Q' \Rightarrow Q}{\vdash \{P\}\ C\ \{Q\}}$$

Example: Here is a little formal proof.

1.	$\vdash$ `{R=X∧0=0} Q:=0 {R=X∧Q=0}`	By the assignment axiom.
2.	$\vdash$ `R=X` $\Rightarrow$ `R=X∧0=0`	By pure logic.
3.	$\vdash$ `{R=X} Q:=0 {R=X∧Q=0}`	By precondition strengthening.
4.	$\vdash$ `R=X+Y×Q` $\Rightarrow$ `R=X+(Y×Q)`	By laws of arithmetic.
5.	$\vdash$ `{R=X+Y×Q} Q:=0 {R=X+(Y×Q)}`	By postcondition weakening.

□

[2] If it is not obvious that $\vdash$ `T` $\Rightarrow$ $(E{=}E)$ is a logical truth, then you should read an elementary introduction to formal logic, e.g. [10,19,47,49].

The rules precondition strengthening and postcondition weakening are sometimes called the *rules of consequence*.

2.1.4 Specification conjunction and disjunction

The following two rules provide a method of combining different specifications about the same command.

Specification conjunction

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \qquad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}$$

Specification disjunction

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \qquad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}$$

These rules are useful for splitting a proof into independent bits. For example, they enable $\vdash$ `{P} C {Q1∧Q2}` to be proved by proving separately that both $\vdash$ `{P} C {Q1}` and $\vdash$ `{P} C {Q2}`.

The rest of the rules allow the deduction of properties of compound commands from properties of their components.

2.1.5 The sequencing rule

The next rule enables a partial correctness specification for a sequence $C_1;C_2$ to be derived from specifications for C_1 and C_2.

The sequencing rule

$$\frac{\vdash \{P\}\ C_1\ \{Q\}, \qquad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1;C_2\ \{R\}}$$

Example: By the assignment axiom:

(i) $\vdash$ {X=x∧Y=y} R:=X {R=x∧Y=y}

(ii) $\vdash$ {R=x∧Y=y} X:=Y {R=x∧X=y}

(iii) $\vdash$ {R=x∧X=y} Y:=R {Y=x∧X=y}

Hence by (i), (ii) and the sequencing rule

(iv) $\vdash$ {X=x∧Y=y} R:=X; X:=Y {R=x∧X=y}

Hence by (iv) and (iii) and the sequencing rule

(v) $\vdash$ {X=x∧Y=y} R:=X; X:=Y; Y:=R {Y=x∧X=y}

□

2.1.6 The derived sequencing rule

The following rule is derivable from the sequencing and consequence rules.

The derived sequencing rule

$$\frac{\begin{array}{ll}\vdash \{P_1\}\ C_1\ \{Q_1\} & \vdash P \Rightarrow P_1 \\ \vdash \{P_2\}\ C_2\ \{Q_2\} & \vdash Q_1 \Rightarrow P_2 \\ \vdots & \vdots \\ \vdash \{P_n\}\ C_n\ \{Q_n\} & \vdash Q_n \Rightarrow Q_{n-1}\end{array}}{\vdash \{P\}\ C_1;\ \ldots\ ;\ C_n\ \{Q\}}$$

The derived sequencing rule enables (v) in the previous example to be deduced directly from (i), (ii) and (iii) in one step.

2.1.7 The block rule

The block rule is like the sequencing rule, but it also takes care of local variables.

The block rule

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{P\}\ \texttt{BEGIN VAR}\ V_1;\ \ldots;\ \texttt{VAR}\ V_n;\ C\ \texttt{END}\ \{Q\}}$$

where none of the variables $V_1, \ldots, V_n$ occur in P or Q.

The syntactic condition that none of the variables $V_1, \ldots, V_n$ occur in P or Q is an example of a *side condition*. It is a syntactic condition that must hold whenever the rule is used. Without this condition the rule is invalid; this is illustrated in the example below.

Note that the block rule is regarded as including the case when there are no local variables (the '$n = 0$' case).

Example: From $\vdash$ {X=x ∧ Y=y} R:=X; X:=Y; Y:=R {Y=x ∧ X=y} (see page 20) it follows by the block rule that

$\vdash$ {X=x ∧ Y=y} BEGIN VAR R; R:=X; X:=Y; Y:=R END {Y=x ∧ X=y}

since R does not occur in X=x ∧ Y=y or X=y ∧ Y=x. Notice that from

$\vdash$ {X=x ∧ Y=y} R:=X; X:=Y {R=x ∧ X=y}

one *cannot* deduce

$\vdash$ {X=x ∧ Y=y} BEGIN VAR R; R:=X; X:=Y END {R=x ∧ X=y}

since R occurs in {R=x ∧ X=y}. This is as required, because assignments to local variables of blocks should not be felt outside the block body. Notice, however, that it is possible to deduce:

$\vdash$ {X=x ∧ Y=y} BEGIN R:=X; X:=Y END {R=x ∧ X=y}.

This is correct because R is no longer a local variable. □

The following exercise addresses the question of whether one can show that changes to local variables inside a block are invisible outside it.

Exercise 7

Consider the specification

{X=x} BEGIN VAR X; X:=1 END {X=x}

Can this be deduced from the rules given so far?

(i) If so, give a proof of it.

(ii) If not, explain why not and suggest additional rules and/or axioms to enable it to be deduced.

□

2.1.8 The derived block rule

From the derived sequencing rule and the block rule the following rule for blocks can be derived.

The derived block rule

$$\frac{\begin{array}{ll} & \vdash P \Rightarrow P_1 \\ \vdash \{P_1\}\ C_1\ \{Q_1\} & \vdash Q_1 \Rightarrow P_2 \\ \vdash \{P_2\}\ C_2\ \{Q_2\} & \vdash Q_2 \Rightarrow P_3 \\ \vdots & \vdots \\ \vdash \{P_n\}\ C_n\ \{Q_n\} & \vdash Q_n \Rightarrow Q \end{array}}{\vdash \{P\}\ \texttt{BEGIN VAR}\ V_1;\ \ldots\ \texttt{VAR}\ V_n; C_1;\ \ldots\ ;\ C_n\ \{Q\}}$$

where none of the variables $V_1, \ldots, V_n$ occur in P or Q.

Using this rule, it can be deduced *in one step* from (i), (ii) and (iii) on page 20 that:

$$\vdash \{\texttt{X=x} \wedge \texttt{Y=y}\}\ \texttt{BEGIN VAR R; R:=X; X:=Y; Y:=R END}\ \{\texttt{Y=x} \wedge \texttt{X=y}\}$$

Exercise 8
Show $\vdash \{\texttt{X=x} \wedge \texttt{Y=y}\}\ \texttt{X:=X+Y; Y:=X-Y; X:=X-Y}\ \{\texttt{Y=x} \wedge \texttt{X=y}\}$ □

Exercise 9
Show $\vdash \{\texttt{X=R+(Y×Q)}\}\ \texttt{BEGIN R:=R-Y; Q:=Q+1 END}\ \{\texttt{X=R+(Y×Q)}\}$ □

2.1.9 The conditional rules

There are two kinds of conditional commands: one-armed conditionals and two-armed conditionals. There are thus two rules for conditionals.

The conditional rules

$$\frac{\vdash \{P \wedge S\}\ C\ \{Q\}, \qquad \vdash P \wedge \neg S \Rightarrow Q}{\vdash \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C\ \{Q\}}$$

$$\frac{\vdash \{P \wedge S\}\ C_1\ \{Q\}, \qquad \vdash \{P \wedge \neg S\}\ C_2\ \{Q\}}{\vdash \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}}$$

Example: Suppose we are given that

(i) $\vdash \texttt{X≥Y} \Rightarrow \texttt{max(X,Y)=X}$
(ii) $\vdash \texttt{Y≥X} \Rightarrow \texttt{max(X,Y)=Y}$

Then by the conditional rules (and others) it follows that

$$\vdash \{\texttt{T}\}\ \texttt{IF X≥Y THEN MAX:=X ELSE MAX:=Y}\ \{\texttt{MAX=max(X,Y)}\}$$

□

Exercise 10
Give a detailed formal proof that the specification in the previous example follows from hypotheses (i) and (ii). □

Exercise 11
Devise an axiom and/or rule of inference for a command SKIP that has no effect. Show that if IF S THEN C is regarded as an abbreviation for IF S THEN C ELSE SKIP, then the rule for one-armed conditionals is derivable from the rule for two-armed conditionals and your axiom/rule for SWAP. □

Exercise 12
Suppose we add to our little programming language commands of the form:

$$\texttt{CASE}\ E\ \texttt{OF BEGIN}\ C_1;\ \ldots\ ;\ C_n\ \texttt{END}$$

These are evaluated as follows:

(i) First E is evaluated to get a value x.
(ii) If x is not a number between 1 and n, then the CASE-command has no effect.

(iii) If $x = i$ where $1 \leq i \leq n$, then command C_i is executed.

Why is the following rule for CASE-commands wrong?

$$\frac{\vdash \{P \wedge E = 1\}\ C_1\ \{Q\},\ \ldots\ ,\vdash \{P \wedge E = n\}\ C_n\ \{Q\}}{\vdash \{P\}\ \texttt{CASE}\ E\ \texttt{OF BEGIN}\ C_1;\ \ldots\ ;\ C_n\ \texttt{END}\ \{Q\}}$$

Hint: Consider the case when P is '$X = 0$', E is 'X', C_1 is 'Y:=0' and Q is '$Y = 0$'.

□

Exercise 13
Devise a proof rule for the CASE-commands in the previous exercise and use it to show:

```
⊢ {1≤X∧X≤3}
  CASE X OF
    BEGIN
      Y:=X-1;
      Y:=X-2;
      Y:=X-3
    END
  {Y=0}
```

□

Exercise 14
Show that if $\vdash$ {P∧S} C_1 {Q} and $\vdash$ {P∧¬S} C_2 {Q}, then it is possible to deduce:

$$\vdash \{P\}\ \texttt{IF S THEN}\ C_1\ \texttt{ELSE IF}\ \neg\texttt{S THEN}\ C_2\ \{Q\}.$$

□

2.1.10 The WHILE-rule

If $\vdash \{P \wedge S\}\ C\ \{P\}$, we say: P is an *invariant* of C whenever S holds. The WHILE-rule says that if P is an invariant of the body of a WHILE-command whenever the test condition holds, then P is an invariant of the whole WHILE-command. In other words, if executing C once preserves the truth of P, then executing C any number of times also preserves the truth of P.

The WHILE-rule also expresses the fact that after a WHILE-command has terminated, the test must be false (otherwise, it wouldn't have terminated).

The WHILE-rule

$$\frac{\vdash \{P \wedge S\}\ C\ \{P\}}{\vdash \{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{P \wedge \neg S\}}$$

Example: By Exercise 9 on page 22

```
⊢ {X=R+(Y×Q)} BEGIN R:=R-Y; Q:=Q+1 END {X=R+(Y×Q)}
```

Hence by precondition strengthening

```
⊢ {X=R+(Y×Q)∧Y≤R} BEGIN R:=R-Y; Q:=Q+1 END {X=R+(Y×Q)}
```

Hence by the WHILE-rule (with P = 'X=R+(Y×Q)')

(i)
```
⊢ {X=R+(Y×Q)}
  WHILE Y≤R DO
    BEGIN R:=R-Y; Q:=Q+1 END
  {X=R+(Y×Q)∧¬(Y≤R)}
```

It is easy to deduce that

(ii) {T} R:=X; Q:=0 {X=R+(Y×Q)}

Hence by (i) and (ii), the sequencing rule and postcondition weakening

```
⊢ {T}
  R:=X;
  Q:=0;
  WHILE Y≤R DO
    BEGIN R:=R-Y; Q:=Q+1 END
  {R<Y∧X=R+(Y×Q)}
```

□

With the exception of the WHILE-rule, all the axioms and rules described so far are sound for total correctness as well as partial correctness. This is because the only commands in our little language that might not terminate are WHILE-commands. Consider now the following proof:

1. $\vdash$ {T} X:=0 {T} (assignment axiom)
2. $\vdash$ {T∧T} X:=0 {T} (precondition strengthening)
3. $\vdash$ {T} WHILE T DO X:=0 {T∧¬T} (2 and the WHILE-rule)

If the WHILE-rule were true for total correctness, then the proof above would show that:

$$\vdash \ [\mathtt{T}]\ \mathtt{WHILE\ T\ DO\ X:=0}\ [\mathtt{T} \wedge \neg \mathtt{T}]$$

but this is clearly false since WHILE T DO X:=0 does not terminate, and even if it did then T ∧ ¬T could not hold in the resulting state.

Extending Floyd-Hoare logic to deal with termination is quite tricky. One approach can be found in Dijkstra [16].

2.1.11 The FOR-rule

It is quite hard to capture accurately the intended semantics of FOR-commands in Floyd-Hoare logic. Axioms and rules are given here that *appear* to be sound, but they are not necessarily complete (see Section 2.2). An early reference on the logic of FOR-commands is Hoare's 1972 paper [33]; a comprehensive treatment can be found in Reynolds [63].

The intention here in presenting the FOR-rule is to show that Floyd-Hoare logic can get very tricky. All the other axioms and rules were quite straightforward and may have given a false sense of simplicity: it is very difficult to give adequate rules for anything other than very simple programming constructs. This is an important incentive for using simple languages.

One problem with FOR-commands is that there are many subtly different versions of them. Thus before describing the FOR-rule, the intended semantics of FOR-commands must be described carefully. In this book, the semantics of

$$\mathtt{FOR}\ V{:=}E_1\ \mathtt{UNTIL}\ E_2\ \mathtt{DO}\ C$$

is as follows:

(i) The expressions E_1 and E_2 are evaluated once to get values e_1 and e_2 respectively.

(ii) If either e_1 or e_2 is not a number, or if $e_1 > e_2$, then nothing is done.

(iii) If $e_1 \leq e_2$ the FOR-command is equivalent to:

```
BEGIN VAR V;
  V:=e1; C; V:=e1+1; C ; ... ; V:=e2; C
END
```

i.e. C is executed $(e_2{-}e_1){+}1$ times with V taking on the sequence of values $e_1, e_1{+}1, \ldots, e_2$ in succession. Note that this description is not rigorous: 'e_1' and 'e_2' have been used both as numbers and as expressions of our little language; the semantics of FOR-commands should be clear despite this.

FOR-rules in different languages can differ in subtle ways from the one here. For example, the expressions E_1 and E_2 could be evaluated at each iteration and the controlled variable V could be treated as global rather than local. Note that with the semantics presented here, FOR-commands cannot go into infinite loops (unless, of course, they contain non-terminating WHILE-commands).

To see how the FOR-rule works, suppose that

$$\vdash \ \{P\}\ C\ \{P[V{+}1/V]\}$$

Suppose also that C does not contain any assignments to the variable V. If this is the case, then it is intuitively clear (and can be rigorously proved) that

$$\vdash \ \{(V = v)\}\ C\ \{(V = v)\}$$

hence by specification conjunction

$$\vdash \ \{P \wedge (V = v)\}\ C\ \{P[V{+}1/V] \ \wedge (V = v)\}$$

Now consider a sequence $V{:=}v;\ C$. By Example 2 on page 18,

$$\vdash \ \{P[v/V]\}\ V{:=}v\ \{P \wedge (V = v)\}$$

Hence by the sequencing rule

$$\vdash \ \{P[V/v]\}\ V{:=}v;\ C\ \{P[V{+}1/V] \ \wedge (V = v)\}$$

Now it is a truth of logic alone that

$$\vdash \ P[V{+}1/V] \wedge (V = v) \ \Rightarrow \ P[v{+}1/V]$$

hence by postcondition weakening

$$\vdash \ \{P[v/V]\}\ V{:=}v;\ C\ \{P[v{+}1/V]\}$$

Taking v to be $e_1, e_1{+}1, \ldots, e_2$ and using the derived sequencing rule we can thus deduce

$$\{P[e_1/V]\}\ V{:=}e_1;\ C;\ V{:=}e_1{+}1;\ \ldots\ ;\ V{:=}e_2;\ C\ \{P[e_2/V]\}$$

This suggests that a FOR-rule could be:

$$\frac{\vdash \{P\}\ C\ \{P[V{+}1/V]\}}{\vdash \{P[E_1/V]\}\ \mathtt{FOR}\ V{:=}E_1\ \mathtt{UNTIL}\ E_2\ \mathtt{DO}\ C\ \{P[E_2{+}1/V]\}}$$

Unfortunately, this rule is unsound. To see this, first note that:

1. $\vdash$ {Y+1=Y+1} X:=Y+1 {X=Y+1} (assignment axiom)
2. $\vdash$ {T} X:=Y+1 {X= Y+1} (1 and precondition strengthening)
3. $\vdash$ X=Y $\Rightarrow$ T (logic: 'anything implies true')
4. $\vdash$ {X=Y} X:=Y+1 {X=Y+1} (2 and precondition strengthening)

Thus if P is 'X=Y' then:

$$\vdash \{P\}\ \texttt{X:=Y+1}\ \{P\texttt{[Y+1/Y]}\}$$

and so by the FOR-rule above, if we take V to be Y, E_1 to be 3 and E_2 to be 1, then

$$\vdash \{\ \underbrace{\texttt{X=3}}_{P\texttt{[3/Y]}}\ \}\ \texttt{FOR Y:=3 UNTIL 1 DO X:=Y+1}\ \{\ \underbrace{\texttt{X=2}}_{P\texttt{[1+1/Y]}}\ \}$$

This is clearly false: it was specified that if the value of E_1 were greater than the value of E_2 then the FOR-command should have no effect, but in this example it changes the value of X from 3 to 2.

To solve this problem, the FOR-rule can be modified to

$$\frac{\vdash \{P\}\ C\ \{P\texttt{[}V\texttt{+1/}V\texttt{]}\}}{\vdash \{P\texttt{[}E_1\texttt{/}V\texttt{]}\ \wedge\ E_1 \leq E_2\}\ \texttt{FOR}\ V\texttt{:=}E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P\texttt{[}E_2\texttt{+1/}V\texttt{]}\}}$$

If this rule is used on the example above all that can be deduced is

$$\vdash \{\texttt{X=3} \wedge \underbrace{\texttt{3}\leq\texttt{1}}_{\text{never true!}}\ \}\ \texttt{FOR Y:=3 UNTIL 1 DO X:=Y+1}\ \{\texttt{X=2}\}$$

This conclusion is harmless since it only asserts that X will be changed if the FOR-command is executed in an impossible starting state.

Unfortunately, there is still a bug in our FOR-rule. Suppose we take P to be 'Y=1', then it is straightforward to show that:

$$\vdash \{\underbrace{\texttt{Y=1}}_{P}\}\ \texttt{Y:=Y-1}\ \{\ \underbrace{\texttt{Y+1=1}}_{P\texttt{[Y+1/Y]}}\ \}$$

so by our latest FOR-rule

$$\vdash \{\ \underbrace{\texttt{1=1}}_{P\texttt{[1/Y]}}\ \wedge\ \texttt{1}\leq\texttt{1}\}\ \texttt{FOR Y:=1 UNTIL 1 DO Y:=Y-1}\ \{\ \underbrace{\texttt{2=1}}_{P\texttt{[1+1/Y]}}\ \}$$

Whatever the command does, it doesn't lead to a state in which 2=1. The problem is that the body of the FOR-command modifies the controlled variable. It is not surprising that this causes problems, since it was explicitly

assumed that the body didn't modify the controlled variable when we motivated the FOR-rule. It turns out that problems also arise if any variables in the expressions E_1 and E_2 (which specify the upper and lower bounds) are modified. For example, taking P to be Z=Y, then it is straightforward to show

$$\vdash \{\underbrace{\texttt{Z=Y}}_{P}\}\ \texttt{Z:=Z+1}\ \{\ \underbrace{\texttt{Z=Y+1}}_{P\texttt{[Y+1/Y]}}\ \}$$

hence the rule allows us the following to be derived:

$$\vdash \{\ \underbrace{\texttt{Z=1}}_{P\texttt{[1/Y]}}\ \wedge\ \texttt{1}\leq\texttt{Z}\}\ \texttt{FOR Y:=1 UNTIL Z DO Z:=Z+1}\ \{\ \underbrace{\texttt{Z=Z+1}}_{P\texttt{[Z+1/Y]}}\ \}$$

This is clearly wrong as one can never have Z=Z+1 (subtracting Z from both sides would give 0=1). One might think that this is not a problem because the FOR-command would never terminate. In some languages this might be the case, but the semantics of our language were carefully defined in such a way that FOR-commands always terminate (see the beginning of this section).

To rule out the problems that arise when the controlled variable or variables in the bounds expressions, are changed by the body, we simply impose a side condition on the rule that stipulates that the rule cannot be used in these situations. The final rule is thus:

The FOR-rule

$$\frac{\vdash \{P \wedge (E_1 \leq V) \wedge (V \leq E_2)\}\ C\ \{P\texttt{[}V\texttt{+1/}V\texttt{]}\}}{\vdash \{P\texttt{[}E_1\texttt{/}V\texttt{]} \wedge (E_1 \leq E_2)\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P\texttt{[}E_2\texttt{+1/}V\texttt{]}\}}$$

where neither V, nor any variable occurring in E_1 or E_2, is assigned to in the command C.

This rule does not enable anything to be deduced about FOR-commands whose body assigns to variables in the bounds expressions. This precludes such assignments being used if commands are to be reasoned about. The strategy of only defining rules of inference for non-tricky uses of constructs helps ensure that programs are written in a perspicuous manner. It is possible to devise a rule that does cope with assignments to variables in bounds expressions, but it is not clear whether it is a good idea to have such a rule.

The FOR-axiom

To cover the case when $E_2 < E_1$, we need the FOR-axiom below.

The FOR-axiom

$$\vdash \{P \wedge (E_2 < E_1)\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P\}$$

This says that when E_2 is less than E_1 the FOR-command has no effect.

Example: By the assignment axiom and precondition strengthening

$$\{\texttt{X = ((N-1)×N) DIV 2}\}\ \texttt{X:=X+N}\ \{\texttt{X=(N×(N+1)) DIV 2}\}$$

Strengthening the precondition of this again yields

$$\vdash \{(\texttt{X=((N-1×N) DIV 2)} \wedge (1 \leq \texttt{N}) \wedge (\texttt{N} \leq \texttt{M})\}\ \texttt{X:=X+N}\ \{\texttt{X=(N×(N+1)) DIV 2}\}$$

Hence by the FOR-rule

$$\vdash \{\texttt{(X=((1-1)×1) DIV 2)} \wedge (1 \leq \texttt{M})\}\ \texttt{FOR N:=1 UNTIL M DO X:=X+N}\ \{\texttt{X=(M×(M+1)) DIV 2}\}$$

Hence

$$\vdash \{\texttt{(X=0)} \wedge (1 \leq \texttt{M})\}\ \texttt{FOR N:=1 UNTIL M DO X:=X+N}\ \{\texttt{X=(M×(M+1)) DIV 2}\}$$

□

Note that if

(i) $\vdash \{P\}\ C\ \{P[V{+}1/V]\}$, or

(ii) $\vdash \{P \wedge (E_1 \leq V)\}\ C\ \{P[V{+}1/V]\}$, or

(iii) $\vdash \{P \wedge (V \leq E_2)\}\ C\ \{P[V{+}1/V]\}$

then by precondition strengthening one can infer

$$\vdash \{P \wedge (E_1 \leq V) \wedge (V \leq E_2)\}\ C\ \{P[V{+}1/V]\}$$

Exercise 15
Show that

$$\vdash \{\texttt{N}\geq 1\}\ \texttt{BEGIN X:=0; FOR N:=1 UNTIL N DO X:=X+N END}\ \{\texttt{X=(N×(N+1)) DIV 2}\}$$

□

2.1.12 Arrays

Floyd-Hoare logic can be extended to cope with arrays so that, for example, the correctness of inplace sorting programs can be verified. However, it is not as straightforward as one might expect to do this. The main problem is that the assignment axiom does not apply to array assignments of the form $A(E_1){:=}E_2$ (where A is an array variable and E_1 and E_2 are expressions).

One might think that the axiom in Section 2.1.1 could be generalized to

$$\vdash \{P[E_2/A(E_1)]\}\ \ A(E_1) := E_2\ \ \{P\}$$

where '$P[E_2/A(E_1)]$' denotes the result of substituting E_2 for all occurrences of $A(E_1)$ throughout P. Alas, this does not work. Consider the following case:

$$P \equiv \texttt{'A(Y)=0'}, \qquad E_1 \equiv \texttt{'X'}, \qquad E_2 \equiv \texttt{'1'}$$

Since A(X) does not occur in P, it follows that $P[1/A(X)] = P$, and hence the generalized axiom yields

$$\vdash \{\texttt{A(Y)=0}\}\ \ \texttt{A(X):=1}\ \ \{\texttt{A(Y)=0}\}$$

This specification is clearly false if X=Y. To avoid this, the array assignment axiom must take into account the possibility that changes to A(X) may also change A(Y), A(Z), ... (since X might equal Y, Z, ...).

We will not go into details of the Floyd-Hoare logic of arrays here, but a thorough treatment can be found in more advanced books (e.g. [63,1,26]).

2.2 Soundness and completeness

It is clear from the discussion of the FOR-rule in Section 2.1.11 that it is not always straightforward to devise correct rules of inference. As discussed at the beginning of Chapter 2, it is very important that the axioms and rules be sound. There are two approaches to ensure this:

(i) Define the language by the axioms and rules of the logic.

(ii) Prove that the logic fits the language.

Approach (i) is called *axiomatic semantics*. The idea is to *define* the semantics of the language by requiring that it make the axioms and rules of inference true. It is then up to implementers to ensure that the logic matches the language. One snag with this approach is that most existing languages have already been defined in some other way (usually by informal and ambiguous natural language statements). An example of a language defined axiomatically is Euclid [46]. The other snag with axiomatic semantics is that it is known to be impossible to devise complete Floyd-Hoare logics for certain constructs (this is discussed further below). It could be argued that this is not a snag at all but an advantage, because it forces programming languages to be made logically tractable. I have some sympathy for this latter view; it is clearly not the position taken by the designers of Ada.

Approach (ii) requires that the axioms and rules of the logic be proved valid. To do this, a mathematical model of states is constructed and then a function, **Meaning** say, is defined which takes an arbitrary command C to a function **Meaning** (C) from states to states. Thus **Meaning** (C) (s) denotes the state resulting from executing command C in state s. The specification $\{P\}C\{Q\}$ is then defined to be true if whenever P is true in a state s and **Meaning** (C) $(s) = s'$ then Q is true in state s'. It is then possible to attempt to prove rigorously that all the axioms are true and that the rules of inference lead from true premisses to true conclusions. Actually carrying out this proof is likely to be quite tedious, especially if the programming language is at all complicated, and there are various technical details which require care (e.g. defining **Meaning** to correctly model non-termination). The precise formulation of such soundness proofs is beyond the scope of this book, but details can be found in the text by Loeckx and Sieber [45].

Even if we are sure that our logic is sound, how can we be sure that every true specification can be proved? It might be the case that for some particular P, Q and C the specification {P}C{Q} was true, but the rules of our logic were too weak to prove it (see Exercise 7 on page 21 for an example). A logic is said to be *complete* if every true statement in it is provable.

There are various subtle technical problems in formulating precisely what it means for a Floyd-Hoare logic to be complete. For example, it is necessary to distinguish incompleteness arising due to incompleteness in the assertion language (e.g. arithmetic) from incompleteness due to inadequate axioms and rules for programming language constructs. The completeness of a Floyd-Hoare logic must thus be defined independently of that of its assertion language. Good introductions to this area can be found in Loeckx and Sieber [45] and Clarke's paper [11]. Clarke's paper also contains a discussion of his important results showing the impossibility of giving complete inference systems for certain combinations of programming language constructs. For example, he proves that it is impossible to give a sound and complete system for any language combining procedures as parameters of procedure calls, recursion, static scopes, global variables and internal procedures as parameters of procedure calls. These features are found in Algol 60, which thus cannot have a sound and complete Floyd-Hoare logic.

2.3 Some exercises

The exercises in this section have been taken from various sources, including Alagić and Arbib's book [1] and Cambridge University Tripos examinations.

Exercise 16
The exponentiation function *exp* satisfies:

$$exp(m, 0) = 1$$
$$exp(m, n+1) = m \times exp(m, n)$$

Devise a command C that uses repeated multiplication to achieve the following partial correctness specification:

$$\{X = x \wedge Y = y \wedge Y \geq 0\}\ C\ \{Z = exp(x, y) \wedge X = x \wedge Y = y\}$$

Prove that your command C meets this specification. □

Exercise 17
Show that

```
⊢ {M≥0}
  BEGIN
    X:=0;
    FOR N:=1 UNTIL M DO X:=X+N
  END
  {X=(M×(M+1)) DIV 2}
```

□

Exercise 18
Deduce:

```
⊢ {S = (x×y)-(X×Y)}
WHILE ¬ODD(X) DO
  BEGIN Y:=2×Y; X:=X DIV 2 END
{S = (x×y)-(X×Y) ∧ ODD(X)}
```

□

Exercise 19
Deduce:

```
⊢ {S = (x×y)-(X×Y)}
WHILE ¬(X=0) DO
  BEGIN
    WHILE ¬ODD(X) DO
      BEGIN Y:=2×Y; X:=X DIV 2 END;
      S:=S+Y;
      X:=X-1
  END
{S = x×y}
```

□

Exercise 20
Deduce:

```
⊢ {X=x ∧ Y=y}
BEGIN
  S:=0;
  WHILE ¬(X=0) DO
    BEGIN
      WHILE ¬ODD(X) DO
        BEGIN Y:=2×Y; X:=X DIV 2 END;
        S:=S+Y;
        X:=X-1
    END
END
{S = x×y}
```

□

Exercise 21
Prove the following invariant property.

```
⊢ {S = (x-X)×y}
  BEGIN
    VAR R;
    R:=0;
    WHILE ¬(R=Y) DO
      BEGIN S:=S+1; R:=R+1 END;
      X:=X-1
  END
  {S = (x-X)×y}
```

Hint: Show that `S = (x-X)×y + R` is an invariant for `S:=S+1; R:=R+1`. □

Exercise 22
Deduce:

```
⊢ {X=x ∧ Y=y}
  BEGIN
    S:=0;
    WHILE ¬(X=0) DO
      BEGIN
        VAR R;
        R:=0;
        WHILE ¬(R=Y) DO
          BEGIN S:=S+1; R:=R+1 END;
        X:=X-1
      END
  END
  {S = x×y}
```

□

Exercise 23
Using $(P \times X^N = x^n) \wedge \neg(X=0) \wedge (N>0)$ as an invariant, deduce:

```
⊢ {X=x ∧ N=n}
  BEGIN
    P:=1;
    IF ¬(X=0)
      THEN
        WHILE ¬(N=0) DO
          BEGIN
            IF ODD(N) THEN P:=P×X;
            N:=N DIV 2;
            X:=X×X
          END
      ELSE P:=0
  END
  {P = x^n}
```

□

Exercise 24
Prove that the command

```
BEGIN
  Z:=0;
  WHILE ¬(X=0) DO
    BEGIN
      IF ODD(X) THEN Z:=Z+Y;
      Y:=Y×2;
      X:=X DIV 2
    END
END
```

computes the product of the initial values of X and Y and leaves the result in Z. □

Exercise 25
Prove that the command

```
BEGIN
  Z:=1;
  WHILE N>0 DO
    BEGIN
      IF ODD(N) THEN Z:=Z×X;
      N:=N DIV 2;
      X:=X×X
    END
END
```


assigns x^n to Z, where x and n are the initial values of X and N respectively and we assume $n \geq 0$. □

Exercise 26
Devise a proof rule for a command

REPEAT *command* UNTIL *statement*

The meaning of REPEAT C UNTIL S is that C is executed and then S is tested; if the result is true, then nothing more is done, otherwise the whole REPEAT command is repeated. Thus REPEAT C UNTIL S is equivalent to C; WHILE S DO C. □

Exercise 27
Use your REPEAT rule to deduce:

```
⊢ {S = C+R ∧ R<Y}
  REPEAT
    S:=S+1; R:=R+1
  UNTIL R=Y
  {S = C+Y}
```

□

Exercise 28
Use your REPEAT rule to deduce:

```
⊢ {X=x ∧ Y=y}
  BEGIN
    S:=0;
    REPEAT
      R:=0;
      REPEAT
        S:=S+1; R:=R+1
      UNTIL R=Y;
      X:=X-1
    UNTIL X=0
  END
  {S = x×y}
```

□

Exercise 29
Assume gcd(X,Y) satisfies:

```
⊢ (X>Y) ⇒ gcd(X,Y)=gcd(X-Y,Y)
⊢ gcd(X,Y)=gcd(Y,X)
⊢ gcd(X,X)=X
```

Prove:

```
{(A>0) ∧ (B>0) ∧ (gcd(A,B)=gcd(X,Y))}
WHILE A>B DO A:=A-B;
WHILE B>A DO B:=B-A
{(0<B) ∧ (B≤A) ∧ (gcd(A,B)=gcd(X,Y))}
```

Hence, or otherwise, use your rule for REPEAT commands to prove:

```
{A=a ∧ B=b}
REPEAT
  WHILE A>B DO A:=A-B;
  WHILE B>A DO B:=B-A
UNTIL A=B
{A=B ∧ A=gcd(a,b)}
```

□

Exercise 30

Prove:

```
{N≥1}
BEGIN
  PROD=0;
  FOR X:=1 UNTIL N DO PROD := PROD+M
END
{PROD = M×N}
```

□

Exercise 31

Prove:

```
{X>0 ∧ Y>0}
BEGIN
  S:=0;
  FOR I:=1 UNTIL X DO
    FOR J:=I UNTIL Y DO
      S:=S+1
END
{S = X×Y}
```

□

Chapter 3

Mechanizing Program Verification

The architecture of a simple program verifier is described. Its operation is justified with respect to the rules of Floyd-Hoare logic.

After doing only a few exercises, the following two things will be painfully clear:

(i) Proofs are typically long and boring (even if the program being verified is quite simple).

(ii) There are lots of fiddly little details to get right, many of which are trivial (e.g. proving $\vdash$ (R=X ∧ Q=0) ⇒ (X = R + Y×Q)).

Many attempts have been made (and are still being made) to automate proof of correctness by designing systems to do the boring and tricky bits of generating formal proofs in Floyd-Hoare logic. Unfortunately logicians have shown that it is impossible in principle to design a decision procedure to decide automatically the truth or falsehood of an arbitrary mathematical statement [58]. However, this does not mean that one cannot have procedures that will prove many useful theorems. The non-existence of a general decision procedure merely shows that one cannot hope to prove *everything* automatically. In practice, it is quite possible to build a system that will mechanize many of the boring and routine aspects of verification. This chapter describes one commonly taken approach to doing this.

Although it is impossible to decide automatically the truth or falsity of arbitrary statements, it *is* possible to check whether an arbitrary formal proof is valid. This consists in checking that the results occurring on each line of the proof are indeed either axioms or consequences of previous lines. Since proofs of correctness of programs are typically very long and boring,

they often contain mistakes when generated manually. It is thus useful to check proofs mechanically, even if they can only be generated with human assistance.

3.1 Overview

In the previous chapter it was shown how to prove $\{P\}C\{Q\}$ by proving properties of the components of C and then putting these together (with the appropriate proof rule) to get the desired property of C itself. For example, to prove $\vdash \{P\}C_1;C_2\{Q\}$ first prove $\vdash \{P\}C_1\{R\}$ and $\vdash \{R\}C_2\{Q\}$ (for suitable R), and then deduce $\vdash \{P\}C_1;C_2\{Q\}$ by the sequencing rule.

This process is called *forward proof* because one moves forward from axioms via rules to conclusions. In practice, it is more natural to work backwards: starting from the goal of showing $\{P\}C\{Q\}$ one generates subgoals, subsubgoals etc. until the problem is solved. For example, suppose one wants to show:

$$\{X=x \wedge Y=y\}\ R:=X;\ X:=Y;\ Y:=R\ \{Y=x \wedge X=y\}$$

then by the assignment axiom and sequencing rule it is sufficient to show the subgoal

$$\{X=x \wedge Y=y\}\ R:=X;\ X:=Y\ \{R=x \wedge X=y\}$$

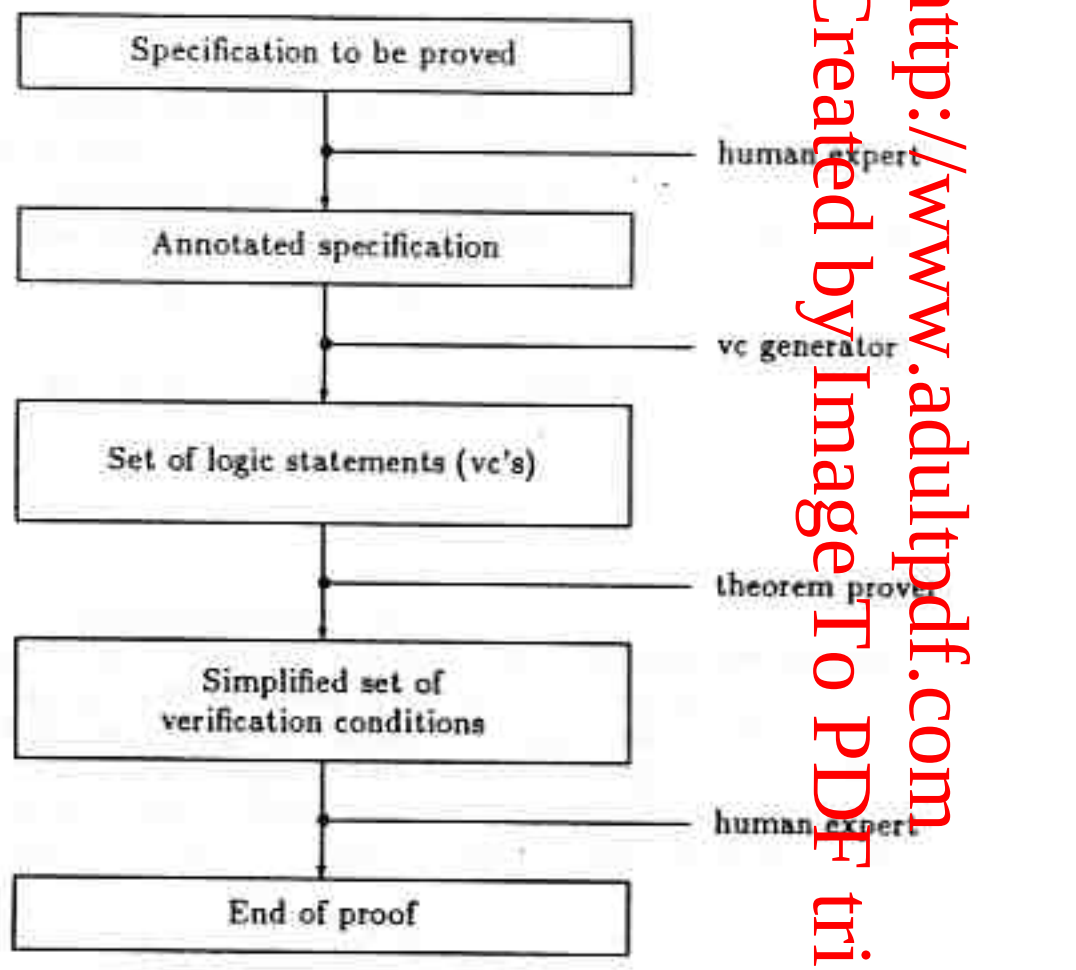
because $\vdash \{R=x \wedge X=y\}\ Y:=R\ \{Y=x \wedge X=y\}$). By a similar argument this ubgoal can be reduced to

$$\{X=x \wedge Y=y\}\ R:=X\ \{R=x \wedge Y=y\}$$

hich clearly follows from the assignment axiom.

This chapter describes how such a *goal oriented* method of proof can formalized; in Chapter 11 a complete LISP program verifier is given to ustrate how it can be mechanized.

The verification system described here can be viewed as a proof checker at also provides some help with generating proofs. The following diagram es an overview of the system.



The system takes as input a partial correctness specification annotated with mathematical statements describing relationships between variables. From the annotated specification the system generates a set of purely mathematical statements, called *verification conditions* (or vc's). In Section 3.5 it is shown that if these verification conditions are provable then the original specification can be deduced from the axioms and rules of Floyd-Hoare logic.

The verification conditions are passed to a *theorem prover* program which attempts to prove them automatically; if it fails, advice is sought from the user. We will concentrate on those aspects pertaining to Floyd-Hoare logic and say very little about theorem proving here. Chapter 10 contains the description of a very simple theorem prover based on rewriting and implemented in LISP. It is powerful enough to do most of the examples discussed in this chapter automatically.

The aim of much current research is to build systems which reduce the role of the slow and expensive human expert to a minimum. This can be

achieved by:

- reducing the number and complexity of the annotations required, and
- increasing the power of the theorem prover.

The next section explains how verification conditions work. In Section 3.5 their use is justified in terms of the axioms and rules of Floyd-Hoare logic. Besides being the basis for mechanical verification systems, verification conditions are a useful way of doing proofs by hand.

3.2 Verification conditions

The following sections describe how a goal oriented proof style can be formalized. To prove a goal $\{P\}C\{Q\}$, three things must be done. These will be explained in detail later, but here is a quick overview:

(i) The program C is *annotated* by inserting into it statements (often called *assertions*) expressing conditions that are meant to hold at various intermediate points. This step is tricky and needs intelligence and a good understanding of how the program works. Automating it is a problem of artificial intelligence.

(ii) A set of logic statements called *verification conditions* (vc's for short) is then generated from the annotated specification. This process is purely mechanical and easily done by a program.

(iii) The verification conditions are proved. Automating this is also a problem of artificial intelligence.

It will be shown that if one can prove all the verification conditions generated from $\{P\}C\{Q\}$ (where C is suitably annotated), then $\vdash \{P\}C\{Q\}$.

Since verification conditions are just mathematical statements, one can think of step 2 above as the 'compilation', or translation, of a verification problem into a conventional mathematical problem.

The following example will give a preliminary feel for the use of verification conditions.

Suppose the goal is to prove (see the example on page 25)


```
{T}
 BEGIN
   R:=X;
   Q:=0;
   WHILE Y≤R DO
     BEGIN R:=R-Y; Q:=Q+1 END
 END
{X = R+Y×Q ∧ R<Y}
```

This first step (1 above) is to insert annotations. A suitable annotated specification is:

```
{T}
 BEGIN
   R:=X;
   Q:=0; {R=X ∧ Q=0} ⟵P1
   WHILE Y≤R DO {X = R+Y×Q} ⟵P2
     BEGIN R:=R-Y; Q:=Q+1 END
 END
{X = R+Y×Q ∧ R<Y}
```

The annotations P_1 and P_2 state conditions which are intended to hold *whenever* control reaches them. Control only reaches the point at which P_1 is placed once, but it reaches P_2 each time the WHILE body is executed and whenever this happens P_2 (i.e. X=R+Y×Q) holds, even though the values of R and Q vary. P_2 is an *invariant* of the WHILE-command.

The second step (2 above), which has yet to be explained, will generate the following four verification conditions:

(i) $T \Rightarrow (X=X \wedge 0=0)$

(ii) $(R=X \wedge Q=0) \Rightarrow (X = R+(Y\times Q))$

(iii) $(X = R+(Y\times Q)) \wedge Y\leq R) \Rightarrow (X = (R-Y)+(Y\times(Q+1)))$

(iv) $(X = R+(Y\times Q)) \wedge \neg(Y\leq R) \Rightarrow (X = R+(Y\times Q) \wedge R<Y)$

Notice that these are statements of arithmetic; the constructs of our programming language have been 'compiled away'.

The third step (3 above) consists in proving these four verification conditions. They are all easy and are proved automatically by the theorem prover described in Chapter 11 (see page 212). The steps are now explained in detail.

3.3 Annotation

An annotated command is a command with statements (called *assertions*) embedded within it. A command is said to be properly annotated if statements have been inserted at the following places:

(i) Before each command C_i (where $i > 1$) in a sequence $C_1;C_2;\ \ldots\ ;C_n$ which is *not* an assignment command,

(ii) After the word DO in WHILE and FOR commands.

Intuitively, the inserted assertions should express the conditions one expects to hold *whenever* control reaches the point at which the assertion occurs.

A properly annotated specification is a specification $\{P\}C\{Q\}$ where C is a properly annotated command.

Example: To be properly annotated, assertions should be at points ① and ② of the specification below:

```
{X=n}
 BEGIN
   Y:=1;  ←—①
   WHILE X≠0 DO ←—②
     BEGIN Y:=Y×X; X:=X-1 END
 END
{X=0 ∧ Y=n!}
```

Suitable statements would be:

at ①: {Y = 1 ∧ X = n}
at ②: {Y×X! = n!}

□

The verification conditions generated from an annotated specification $\{P\}C\{Q\}$ are described by considering the various possibilities for C in turn. This process is justified in Section 3.5 by showing that $\vdash \{P\}C\{Q\}$ if all the verification conditions can be proved.

3.4 Verification condition generation

In this section a procedure is described for generating verification conditions for an annotated partial correctness specification $\{P\}C\{Q\}$. This procedure is *recursive* on C.

Assignment commands

The single verification condition generated by

$$\{P\}\ V:=E\ \{Q\}$$

is

$$P \Rightarrow Q[E/V]$$

Example: The verification condition for

{X=0} X:=X+1 {X=1}

is

X=0 ⇒ (X+1)=1

(which is clearly true). □

One-armed conditional

The verification conditions generated by

$$\{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C\ \{Q\}$$

are

(i) $(P \wedge \neg S) \Rightarrow Q$

(ii) the verifications generated by

$$\{P \wedge S\}\ C\ \{Q\}$$

Example: The verification conditions for

{T} IF X<0 THEN X:=-X {X≥0}

are T ∧ ¬(X<0) ⇒ X≥0 together with the verification conditions f {T ∧ (X<0)} X:=-X {X≥0}, i.e. T ∧ (X<0) ⇒ -X≥0. The two vc's a thus:

(i) T ∧ ¬(X>0) ⇒ X≥0

(ii) `I ∧ (X<0) ⇒ -X≥0`

These are equivalent to `X≥0 ⇒ X≥0` and `X<0 ⇒ -X≥0`, respectively, which are both clearly true. □

Two-armed conditional

The verification conditions generated from

$$\{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}$$

are

(i) the verification conditions generated by

$$\{P \wedge S\}\ C_1\ \{Q\}$$

(ii) the verifications generated by

$$\{P \wedge \neg S\}\ C_2\ \{Q\}$$

Exercise 32

What are the verification conditions for the following specification?

`{T} IF X≥Y THEN MAX:=X ELSE MAX:=Y {MAX=max(X,Y)}`

Do they follow from the assumptions about `max(X,Y)` given in the example on page 23? □

If $C_1; \ldots; C_n$ is properly annotated, then (see page 44) it must be of one of the two forms:

1. $C_1; \ldots; C_{n-1}; \{R\}C_n$, or
2. $C_1; \ldots; C_{n-1}; V := E$.

where, in both cases, $C_1; \ldots; C_{n-1}$ is a properly annotated command.

Sequences

1. The verification conditions generated by

$$\{P\}\ C_1; \ldots; C_{n-1};\ \{R\}\ C_n\ \{Q\}$$

(where C_n is not an assignment) are:

(a) the verification conditions generated by

$$\{P\}\ C_1; \ldots; C_{n-1}\ \{R\}$$

(b) the verifications generated by

$$\{R\}\ C_n\ \{Q\}$$

2. The verification conditions generated by

$$\{P\}\ C_1; \ldots; C_{n-1}; I := E\ \{Q\}$$

are the verification conditions generated by

$$\{P\}\ C_1; \ldots; C_{n-1}\ \{Q[E/V]\}$$

Example: The verification conditions generated from

`{X=x ∧ Y=y} R:=X; X:=Y; Y:=R {X=y ∧ Y=x}`

are those generated by

`{X=x ∧ Y=y} R:=X; X:=Y {(X=y ∧ Y=x)[R/Y]}`

which, after doing the substitution, simplifies to

`{X=x ∧ Y=y} R:=X; X:=Y {X=y ∧ R=x}`

The verification conditions generated by this are those generated by

`{X=x ∧ Y=y} R:=X {(X=y ∧ R=x)[Y/X]}`

which, after doing the substitution, simplifies to

`{X=x ∧ Y=y} R:=X {Y=y ∧ R=x}`.

The only verification condition generated by this is

$$\texttt{X=x} \wedge \texttt{Y=y} \Rightarrow (\texttt{Y=y} \wedge \texttt{R=x})[\texttt{X/R}]$$

which, after doing the substitution, simplifies to

$$\texttt{X=x} \wedge \texttt{Y=y} \Rightarrow \texttt{Y=y} \wedge \texttt{X=x}$$

which is obviously true. □

The procedure for generating verification conditions from blocks involves checking the syntactic condition that the local variables of the block do not occur in the precondition or postcondition. The need for this is clear from the side condition in the block rule (see page 20); this will be explained in more detail when the procedure for generating verification conditions is justified in Section 3.5.

Blocks

The verification conditions generated by

$$\{P\}\ \texttt{BEGIN VAR}\ V_1; \ldots ;\texttt{VAR}\ V_n; C\ \texttt{END}\ \{Q\}$$

are

(i) the verification conditions generated by $\{P\}C\{Q\}$, and

(ii) the syntactic condition that none of $V_1, \ldots, V_n$ occur in either P or Q.

Example: The verification conditions for

$$\{\texttt{X=x} \wedge \texttt{Y=y}\}\ \texttt{BEGIN VAR R; R:=X; X:=Y; Y:=R END}\ \{\texttt{X=y} \wedge \texttt{Y=x}\}$$

are those generated by $\{\texttt{X=x} \wedge \texttt{Y=y}\}\ \texttt{R:=X; X:=Y; Y:=R}\ \{\texttt{X=y} \wedge \texttt{Y=x}\}$ (since R does not occur in $\{\texttt{X=x} \wedge \texttt{Y=y}\}$ or $\{\texttt{X=y} \wedge \texttt{Y=x}\}$). See the previous example for the verification conditions generated by this. □

Exercise 33
What are the verification conditions for the following specification?

$$\{\texttt{X = R+(Y×Q)}\}\ \texttt{BEGIN R:=R-Y; Q:=Q+1 END}\ \{\texttt{X = R+(Y×Q)}\}$$

□

Exercise 34
What are the verification conditions for the following specification?

$$\{\texttt{X=x}\}\ \texttt{BEGIN VAR X; X:=1 END}\ \{\texttt{X=x}\}$$

Relate your answer to this exercise to your answer to Exercise 1 on page 21. □

A correctly annotated specification of a WHILE-command has the form

$$\{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C\ \{Q\}$$

Following the usage on page 24, the annotation R is called an invariant.

WHILE-commands

The verification conditions generated from

$$\{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C\ \{Q\}$$

are

(i) $P \Rightarrow R$

(ii) $R \wedge \neg S \Rightarrow Q$

(iii) the verification conditions generated by $\{R \wedge S\}\ C\ \{R\}$.

Example: The verification conditions for

```
{R=X ∧ Q=0}
 WHILE Y≤R DO {X=R+Y×Q}
   BEGIN R:=R-Y; Q=Q+1 END
{X = R+(Y×Q) ∧ R<Y}
```

are:

(i) $\texttt{R=X} \wedge \texttt{Q=0} \Rightarrow (\texttt{X = R+(Y×Q)})$

(ii) $\texttt{X = R+Y×Q} \wedge \neg(\texttt{Y≤R}) \Rightarrow (\texttt{X = R+(Y×Q)} \wedge \texttt{R<Y})$

together with the verification condition for

```
{X = R+(Y×Q) ∧ (Y≤R)}
 BEGIN R:=R-Y; Q:=Q+1 END
{X=R+(Y×Q)}
```

which (see Exercise 33) consists of the single condition

(iii) `X = R+(Y×Q) ∧ (Y≤R) ⇒ X = (R-Y)+(Y×(Q+1))`

The WHILE-command specification is thus true if (i), (ii) and (iii) hold, i.e.

```
⊢ {R=X ∧ Q=0}
  WHILE Y≤R DO
    BEGIN R:=R-Y; Q:=Q+1 END
  {X = R+(Y×Q) ∧ R<Y}
```

if

```
⊢ R=X ∧ Q=0 ⇒ (X = R+(Y×Q))
```

and

```
⊢ X = R+(Y×Q) ∧ ¬(Y≤R) ⇒ (X = R+(Y×Q) ∧ R<Y)
```

and

```
X = R+(Y×Q) ∧ (Y≤R) ⇒ X = (R-Y)+(Y×(Q+1))
```

□

Exercise 15
What are the verification conditions generated by the annotated program for computing n! (the factorial of n) given in the example on page 44? □

A correctly annotated specification of a FOR-command has the form

$\{P\}$ FOR V:=E_1 UNTIL E_2 DO $\{R\}$ C $\{Q\}$

FOR-commands

The verification conditions generated from

$\{P\}$ FOR V:=E_1 UNTIL E_2 DO $\{R\}$ C $\{Q\}$

are

(i) $P \Rightarrow R[E_1/V]$
(ii) $R[E_2+1/V] \Rightarrow Q$
(iii) $P \wedge E_2 < E_1 \Rightarrow Q$
(iv) the verification conditions generated by

$$\{R \wedge E_1 \leq V \wedge V \leq E_2\}\ C\ \{R[V+1/V]\}$$

(v) the syntactic condition that neither V, nor any variable occurring in E_1 or E_2, is assigned to inside C.

Example: The verification conditions generated by

```
{X=0 ∧ 1≤M}
 FOR N:=1 UNTIL M DO {X=((N-1)×N) DIV 2} X:=X+N
{X = (M×(M+1)) DIV 2}
```

are

(i) `X=0 ∧ 1≤M ⇒ X=((1-1)×1) DIV 2`
(ii) `X = (((M+1)-1)×(M+1)) DIV 2 ⇒ X = (M×(M+1)) DIV 2`
(iii) `X=0 ∧ 1≤M ∧ M<1 ⇒ X = (M×(M+1)) DIV 2`
(iv) The verification condition generated by

```
{X = ((N-1)×N) DIV 2 ∧ 1≤N ∧ N≤M}
 X:=X+N
{X = (((N+1)-1)×(N+1)) DIV 2}
```

which, after some simplification, is

```
X = ((N-1) × N) DIV 2 ∧ 1 ≤ N
                ⇒
N ≤ M ⇒ X+N = (N × (N+1)) DIV 2
```

which is true since

$$\frac{(\mathtt{N}-1)\times\mathtt{N}}{2}+\mathtt{N}=\frac{2\mathtt{N}+(\mathtt{N}-1)\times\mathtt{N}}{2}$$

$$=\frac{2\mathtt{N}+\mathtt{N}^2-\mathtt{N}}{2}$$

$$=\frac{\mathtt{N}+\mathtt{N}^2}{2}$$

$$=\frac{\mathtt{N}\times(\mathtt{N}+1)}{2}$$

(Exercise: justify this calculation in the light of the fact that

$$(x+y)\ \mathtt{DIV}\ z \neq (x\ \mathtt{DIV}\ z)+(y\ \mathtt{DIV}\ z)$$

as is easily seen by taking x, y and z to be 3, 5 and 8, respectively.)

(v) Neither `N` or `M` is assigned to in `X:=X+N`

□

□

3.5 Justification of verification conditions

It will be shown in this section that an annotated specification $\{P\}C\{Q\}$ is provable in Floyd-Hoare logic (i.e. $\vdash \{P\}C\{Q\}$) if the verification conditions generated by it are provable. This shows that the verification conditions are *sufficient*, but not that they are necessary. In fact, the verification conditions are the weakest sufficient conditions, but we will neither make this more precise nor go into details here. An in-depth study of preconditions can be found in Dijkstra's book [16].

It is easy to show (see the exercise below) that the verification conditions are not necessary, i.e. that the verification conditions for $\{P\}C\{Q\}$ not being provable doesn't imply that $\vdash \{P\}C\{Q\}$ cannot be deduced.

Exercise 36
Show that

(i) The verification conditions from the annotated specification

`{T} WHILE F DO {F} X:=0 {T}`

are not provable.

(ii) $\vdash$ `{T} WHILE F DO X:=0 {T}`

□

The argument that the verification conditions are sufficient will be by *induction* on the structure of C. Such inductive arguments have two parts. First, it is shown that the result holds for assignment commands. Second, it is shown that when C is not an assignment command then if the result holds for the constituent commands of C (this is called the *induction hypothesis*), then it holds also for C. The first of these parts is called the *basis* of the induction and the second is called the step. From the basis and the step it follows that the result holds for all commands.

Assignments

The only verification condition for $\{P\}V:=E\{Q\}$ is $P \Rightarrow Q[E/V]$. If this is provable, then as $\vdash \{Q[E/V]\}V:=E\{Q\}$ (by the assignment axiom on page 16) it follows by precondition strengthening (page 17) that $\vdash \{P\}V:=E\{Q\}$.

One-armed conditionals

If the verification conditions for $\{P\}$ `IF` S `THEN` C $\{Q\}$ are provable, then $\vdash P \wedge \neg S \Rightarrow Q$ and all the verification conditions for $\{P \wedge S\}\ C\ \{Q\}$ are provable. Hence by the induction hypothesis $\vdash \{P \wedge S\}\ C\ \{Q\}$ and hence by the one-armed conditional rule (page 22) it follows that $\vdash \{P\}$ `IF` S `THEN` C $\{Q\}$.

Two-armed conditionals

If the verification conditions for $\{P\}$ `IF` S `THEN` C_1 `ELSE` C_2 $\{Q\}$ are provable, then the verification conditions for both $\{P \wedge S\}\ C_1\ \{Q\}$ and $\{P \wedge \neg S\}\ C_2\ \{Q\}$ are provable. By the induction hypothesis we can assume that $\vdash \{P \wedge S\}\ C_1\ \{Q\}$ and $\vdash \{P \wedge \neg S\}\ C_2\ \{Q\}$. Hence by the two-armed conditional rule (page 22) $\vdash \{P\}$ `IF` S `THEN` C_1 `ELSE` C_2 $\{Q\}$.

Sequences

There are two cases to consider:

(i) If the verification conditions for $\{P\}\ C_1;\ldots;C_{n-1};\{R\}C_n\ \{Q\}$ are provable, then the verification conditions for $\{P\}\ C_1;\ldots;C_{n-1}\ \{R\}$ and $\{R\}\ C_n\ \{Q\}$ must both be provable and hence by induction we have $\vdash \{P\}\ C_1;\ldots;C_{n-1}\ \{R\}$ and $\vdash \{R\}\ C_n\ \{Q\}$. Hence by the sequencing rule (page 19) $\vdash \{P\}\ C_1;\ldots;C_{n-1};C_n\ \{Q\}$.

(ii) If the verification conditions for $\{P\}\ C_1;\ldots;C_{n-1};V := E\ \{Q\}$ are provable, then it must be the case that the verification conditions for $\{P\}\ C_1;\ldots;C_{n-1}\ \{Q[E/V]\}$ are also provable and hence by induction we have $\vdash\ \{P\}\ C_1;\ldots;C_{n-1}\ \{Q[E/V]\}$. It then follows by the assignment axiom that $\vdash\ \{Q[E/I]\}\ V := E\ \{Q\}$, hence by the sequencing rule $\vdash\ \{P\}\ C_1;\ldots;C_{n-1};V := E\{Q\}$.

Blocks

If the verification conditions for $\{P\}$`BEGIN VAR` $V_1;\ldots;$`VAR` $V_n;C$ `END` $\{Q\}$ are provable, then the verification conditions for $\{P\}\ C\ \{Q\}$ are provable and $V_1,\ldots,V_n$ do not occur in P or Q. By induction $\vdash\ \{P\}\ C\ \{Q\}$ hence by the block rule (page 20) $\vdash\ \{P\}$ `BEGIN VAR` $V_1;\ldots;$`VAR` $V_n;C$ `END` $\{Q\}$.

WHILE-commands

If the verification conditions for $\{P\}$ `WHILE` S `DO` $\{R\}\ C\ \{Q\}$ are provable, then $\vdash\ P \Rightarrow R$, $\vdash\ (R \wedge \neg S) \Rightarrow Q$ and the verification conditions for $\{R \wedge S\}\ C\ \{R\}$ are provable. By induction $\vdash\ \{R \wedge S\}\ C\ \{R\}$, hence by the `WHILE`-rule (page 24) $\vdash\ \{R\}$ `WHILE` S `DO` $C\ \{R \wedge \neg S\}$, hence by the consequence rules (see page 19) $\vdash\ \{P\}$ `WHILE` S `DO` $C\ \{Q\}$.

FOR-commands

Finally if the verification conditions for

$$\{P\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ \{R\}\ C\ \{Q\}$$

are provable, then

(i) $\vdash\ P \Rightarrow R[E_1/V]$

(ii) $\vdash\ R[E_1/V] \Rightarrow Q$

(iii) $\vdash\ P \wedge E_2 < E_1 \Rightarrow Q$

(iv) The verification conditions for

$$\{R \wedge E_1 \le V \wedge V \le E_2\}\ C\ \{R[V+1/V]\}$$

are provable.

(v) Neither V, nor any variable in E_1 or E_2, is assigned to in C.

By induction $\vdash\ \{R \wedge E_1 \le V \wedge V \le E_2\}\ C\ \{R[V+1/V]\}$, hence by the `FOR`-rule

$$\vdash\ \{R[E_1/V] \wedge E_1 \le E_2\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{R[E_2+1/V]\}$$

hence by (i), (ii) and the consequence rules

(vi) $\vdash\ \{P \wedge E_1 \le E_2\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{Q\}$.

Now by the `FOR`-axiom (page 30)

$$\vdash\ \{(P \wedge E_2 < E_1) \wedge E\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{P \wedge E_2 < E_1\},$$

hence by the consequence rules and (iii)

$$\vdash\ \{P \wedge E_2 < E_1\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{Q\}.$$

Combining this last specification with (vi) using specification disjunction (page 19) yields

$$\vdash\ \{P \wedge E_2 < E_1) \vee (P \wedge E_1 \le E_2)\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{Q \vee Q\}$$

Now $\vdash\ Q \vee Q \Rightarrow Q$ and

$$\vdash\ (P \wedge E_2 < E_1) \vee (P \wedge E_1 \le E_2) \Rightarrow P \wedge (E_2 < E_1 \vee E_1 \le E_2)$$

but $\vdash\ E_2 < E_1 \vee E_1 \le E_2$, hence

$$\vdash\ (P \wedge E_2 < E_1) \vee (P \wedge E_1 \le E_2)$$

and so one can conclude:

$$\vdash\ \{P\}\ \texttt{FOR}\ V := E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ C\ \{Q\}$$

Thus the verification conditions for the `FOR`-command are sufficient.

Exercise 37
Annotate the specifications in Exercises 17 to 25 (they start on page 33) and then generate the corresponding verification conditions. □

Exercise 38
Devise verification conditions for commands of the form

`REPEAT` C `UNTIL` S

(See Exercise 26, page 37.) □

Exercise 39
Do Exercises 27–31 using verification conditions. □

Exercise 40
Show that if no variable occurring in P is assigned to in C, then $\vdash\ \{P\}\ C\{P\}$. *Hint:* Use induction on the structure of C, see page 52. □

Chapter 4

Introduction to the λ-calculus

The λ-calculus notation for specifying functions is introduced. Various technical definitions are explained and motivated, including the rules of α-, β- and η-conversion.

The λ-calculus (or lambda-calculus) is a theory of functions that was originally developed by the logician Alonzo Church as a foundation for mathematics. This work was done in the 1930s, several years before digital computers were invented. A little earlier (in the 1920s) Moses Schonfinkel developed another theory of functions based on what are now called 'combinators'. In the 1930s, Haskell Curry rediscovered and extended Schönfinkel's theory and showed that it was equivalent to the λ-calculus. About this time Kleene showed that the λ-calculus was a universal computing system; it was one of the first such systems to be rigorously analysed. In the 1950s John McCarthy was inspired by the λ-calculus to invent the programming language LISP. In the early 1960s Peter Landin showed how the meaning of imperative programming languages could be specified by translating them into the λ-calculus. He also invented an influential prototype programming language called ISWIM [42]. This introduced the main notations of functional programming and influenced the design of both functional and imperative languages. Building on this work, Christopher Strachey laid the foundations for the important area of denotational semantics [23,67]. Technical questions concerning Strachey's work inspired the mathematical logician Dana Scott to invent the theory of domains, which is now one of the most important parts of theoretical computer science. During the 1970s Peter Henderson and Jim Morris took up Landin's work and wrote a number of influential papers arguing that functional programming had important advantages for software engineering [29,28]. At about the same time David Turner proposed that Schönfinkel and Curry's combinators could be used as the machine code of computers for executing functional programming languages. Such computers could exploit mathematical properties of the

λ-calculus for the parallel evaluation of programs. During the 1980s several research groups took up Henderson's and Turner's ideas and started working on making functional programming practical by designing special architectures to support it, some of them with many processors.

We thus see that an obscure branch of mathematical logic underlies important developments in programming language theory, such as:

(i) The study of fundamental questions of computation.

(ii) The design of programming languages.

(iii) The semantics of programming languages.

(iv) The architecture of computers.

4.1 Syntax and semantics of the λ-calculus

The λ-calculus is a notation for defining functions. The expressions of the notation are called *λ-expressions* and each such expression denotes a function. It will be seen later how functions can be used to represent a wide variety of data and data-structures including numbers, pairs, lists etc. For example, it will be demonstrated how an arbitrary pair of numbers (x, y) can be represented as a λ-expression. As a notational convention, mnemonic names are assigned in bold or underlined to particular λ-expressions; for example $\underline{1}$ is the λ-expression (defined in Section 5.3) which is used to represent the number one.

There are just three kinds of λ-expressions:

(i) **Variables:** x, y, z etc. The functions denoted by variables are determined by what the variables are bound to in the *environment*. Binding is done by abstractions (see 3 below). We use V, V_1, V_2 etc. for arbitrary variables.

(ii) **Function applications or combinations:** if E_1 and E_2 are λ-expressions, then so is $(E_1\ E_2)$; it denotes the result of applying the function denoted by E_1 to the function denoted by E_2. E_1 is called the *rator* (from 'operator') and E_2 is called the *rand* (from 'operand'). For example, if $(\underline{m}, \underline{n})$ denotes a function representing the pair of numbers m and n (see Section 5.2) and **sum** denotes the addition function [1] λ-calculus (see Section 5.5), then the application $(\mathbf{sum}\ \underline{m}, \underline{n})$ denotes $\underline{m+n}$.

[1] Note that **sum** is a λ-expression, whereas + is a mathematical symbol in the 'meta-language' (i.e. English) that we are using for talking about the λ-calculus.

(iii) **Abstractions:** if V is a variable and E is a λ-expression, then $\lambda V.\ E$ is an abstraction with *bound variable* V and *body* E. Such an abstraction denotes the function that takes an argument a and returns as result the function denoted by E in an environment in which the bound variable V denotes a. More specifically, the abstraction $\lambda V.\ E$ denotes a function which takes an argument E' and transforms it into the thing denoted by $E[E'/V]$ (the result of substituting E' for V in E, see Section 4.8). For example, $\lambda x.\ \mathbf{sum}(x, \underline{1})$ denotes the add-one function.

Using BNF, the syntax of λ-expressions is just:

$$\begin{aligned} \langle \lambda\text{-expression} \rangle \ ::= \ & \langle \text{variable} \rangle \\ | \ & (\langle \lambda\text{-expression} \rangle \ \langle \lambda\text{-expression} \rangle) \\ | \ & (\lambda \ \langle \text{variable} \rangle \ . \ \langle \lambda\text{-expression} \rangle) \end{aligned}$$

If V ranges over the syntax class ⟨ *variable* ⟩ and $E, E_1, E_2, \ldots$ etc. range over the syntax class ⟨ *λ-expression* ⟩, then the BNF simplifies to:

$$E ::= V \mid \underbrace{(E_1\ E_2)}_{\text{applications (combinations)}} \mid \underbrace{\lambda V.\ E}_{\text{abstractions}}$$

(variables)

The description of the meaning of λ-expressions just given above is vague and intuitive. It took about 40 years for logicians (Dana Scott, in fact [66]) to make it rigorous in a useful way. We shall not be going into details of this.

Example: $(\lambda x.\ x)$ denotes the 'identity function': $((\lambda x.\ x)\ E) = E$. □

Example: $(\lambda x.\ (\lambda f.\ (f\ x)))$ denotes the function which when applied to E yields $(\lambda f.\ (f\ x))[E/x]$, i.e. $(\lambda f.\ (f\ E))$. This is the function which when applied to E' yields $(f\ E)[E'/f]$ i.e. $(E'\ E)$. Thus

$$((\lambda x.\ (\lambda f.\ (f\ x)))\ E) = (\lambda f.\ (f\ E))$$

and

$$((\lambda f.\ (f\ E))\ E') = (E'\ E)$$

□

Exercise 41
Describe the function denoted by $(\lambda x.\ (\lambda y.\ y))$. □

Example: Section 5.3 describes how numbers can be represented by λ-expressions. Assume that this has been done and that $\underline{0}$, $\underline{1}$, $\underline{2}$, ... are λ-expressions which represent 0, 1, 2, ..., respectively. Assume also that add is a λ-expression denoting a function satisfying:

$$((\text{add } \underline{m})\ \underline{n}) = \underline{m+n}.$$

Then $(\lambda x.\ ((\text{add } \underline{1})\ x))$ is a λ-expression denoting the function that transforms $\underline{n}$ to $\underline{1+n}$, and $(\lambda x.\ (\lambda y.\ ((\text{add } x)y)))$ is a λ-expression denoting the function that transforms $\underline{m}$ to the function which when applied to $\underline{n}$ yields $\underline{m+n}$, namely $\lambda y.\ ((\text{add } \underline{m})y))$. □

The relationship between the function sum in (ii) at the beginning of this section (page 60) and the function add in the previous example is explained in Section 5.5.

4.2 Notational conventions

The following conventions help minimize the number of brackets one has to write.

1. Function application associates to the left, i.e. $E_1\ E_2\ \cdots\ E_n$ means $((\cdots (E_1\ E_2) \cdots)\ E_n)$. For example:

$E_1\ E_2$	means	$(E_1\ E_2)$
$E_1\ E_2\ E_3$	means	$((E_1\ E_2)E_3)$
$E_1\ E_2\ E_3\ E_4$	means	$(((E_1\ E_2)E_3)E_4)$

2. $\lambda V.\ E_1\ E_2\ \ldots\ E_n$ means $(\lambda V.\ (E_1\ E_2\ \ldots\ E_n))$. Thus the scope of '$\lambda V$' extends as far to the right as possible.

3. $\lambda V_1 \cdots V_n.\ E$ means $(\lambda V_1.\ (\cdots\ (\lambda V_n.\ E)\ \cdots))$. For example:

$\lambda x\ y.\ E$	means	$(\lambda x.\ (\lambda y.\ E))$
$\lambda x\ y\ z.\ E$	means	$(\lambda x.\ (\lambda y.\ (\lambda z.\ E)))$
$\lambda x\ y\ z\ w.\ E$	means	$(\lambda x.\ (\lambda y.\ (\lambda z.\ (\lambda w.\ E))))$

Example: $\lambda x\ y.$ add $y\ x$ means $(\lambda x.\ (\lambda y.\ ((\text{add } y)\ x)))$. □

4.3 Free and bound variables

An occurrence of a variable V in a λ-expression is *free* if it is not within the scope of a 'λV', otherwise it is *bound*. For example

$$(\lambda x.\ y\ x)(\lambda y.\ x\ y)$$

(free, free, bound, bound)

4.4 Conversion rules

In Chapter 5 it is explained how λ-expressions can be used to represent data objects like numbers, strings etc. For example, an arithmetic expression like $(2+3) \times 5$ can be represented as a λ-expression and its 'value' 25 can also be represented as a λ-expression. The process of 'simplifying' $(2+3) \times 5$ to 25 will be represented by a process called *conversion* (or *reduction*). The rules of λ-conversion described below are very general, yet when they are applied to λ-expressions representing arithmetic expressions they simulate arithmetical evaluation.

There are three kinds of λ-conversion called α-conversion, β-conversion and η-conversion (the original motivation for these names is not clear). In stating the conversion rules the notation $E[E'/V]$ is used to mean the result of substituting E' for each *free* occurrence of V in E. The substitution is called *valid* if and only if no free variable in E' becomes bound in $E[E'/V]$. Substitution is described in more detail in Section 4.8.

The rules of λ-conversion

- **α-conversion.**

  Any abstraction of the form $\lambda V.\ E$ can be converted to $\lambda V'.\ E[V'/V]$ provided the substitution of V' for V in E is valid.

- **β-conversion.**

  Any application of the form $(\lambda V.\ E_1)\ E_2$ can be converted to $E_1[E_2/V]$, provided the substitution of E_2 for V in E_1 is valid.

- **η-conversion.**

  Any abstraction of the form $\lambda V.\ (E\ V)$ in which V has no free occurrence in E can be reduced to E.

The following notation will be used:

- $E_1 \underset{\alpha}{\longrightarrow} E_2$ means E_1 α-converts to E_2.
- $E_1 \underset{\beta}{\longrightarrow} E_2$ means E_1 β-converts to E_2.
- $E_1 \underset{\eta}{\longrightarrow} E_2$ means E_1 η-converts to E_2.

In Section 4.4.4 below this notation is extended.

The most important kind of conversion is β-conversion; it is the one that can be used to simulate arbitrary evaluation mechanisms. α-conversion is to do with the technical manipulation of bound variables and η-conversion expresses the fact that two functions that always give the same results on the same arguments are equal (see Section 4.7). The next three subsections give further explanation and examples of the three kinds of conversion (note that 'conversion' and 'reduction' are used below as synonyms).

4.4.1 α-conversion

A λ-expression (necessarily an abstraction) to which α-reduction can be applied is called an *α-redex*. The term 'redex' abbreviates 'reducible expression'. The rule of α-conversion just says that bound variables can be renamed provided no 'name-clashes' occur.

Examples

$$\lambda x.\ x \underset{\alpha}{\longrightarrow} \lambda y.\ y$$

$$\lambda x.\ f\ x \underset{\alpha}{\longrightarrow} \lambda y.\ f\ y$$

It is *not* the case that

$$\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y \underset{\alpha}{\longrightarrow} \lambda y.\ \lambda y.\ \mathbf{add}\ y\ y$$

because the substitution $(\lambda y.\ \mathbf{add}\ x\ y)[y/x]$ is not valid since the y that replaces x becomes bound. □

4.4.2 β-conversion

A λ-expression (necessarily an application) to which β-reduction can be applied is called a *β-redex*. The rule of β-conversion is like the evaluation of a function call in a programming language: the body E_1 of the function $\lambda V.\ E_1$ is evaluated in an environment in which the 'formal parameter' V is bound to the 'actual parameter' E_2.

Examples

$$(\lambda x.\ f\ x)\ E \underset{\beta}{\longrightarrow} f\ E$$

$$(\lambda x.\ (\lambda y.\ \mathbf{add}\ x\ y))\ \underline{3} \underset{\beta}{\longrightarrow} \lambda y.\ \mathbf{add}\ \underline{3}\ y$$

$$(\lambda y.\ \mathbf{add}\ \underline{3}\ y)\ \underline{4} \underset{\beta}{\longrightarrow} \mathbf{add}\ \underline{3}\ \underline{4}$$

It is *not* the case that

$$(\lambda x.\ (\lambda y.\ \mathbf{add}\ x\ y))\ (\mathbf{square}\ y) \underset{\beta}{\longrightarrow} \lambda y.\ \mathbf{add}\ (\mathbf{square}\ y)\ y$$

because the substition $(\lambda y.\ \mathbf{add}\ x\ y)[(\mathbf{square}\ y)/x]$ is not valid, since y is free in $(\mathbf{square}\ y)$ but becomes bound after substitution for x in $(\lambda y.\ \mathbf{add}\ x\ y)$. □

It takes some practice to parse λ-expressions according to the conventions of Section 4.2 so as to identify the β-redexes. For example, consider the application:

$$(\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y)\ \underline{3}\ \underline{4}.$$

Putting in brackets according to the conventions expands this to:

$$(((\lambda x.\ (\lambda y.\ ((\mathbf{add}\ x)\ y)))\ \underline{3})\ \underline{4})$$

which has the form:

$$((\lambda x.\ E)\ \underline{3})\ \underline{4}$$

where

$$E = (\lambda y.\ \mathbf{add}\ x\ y)$$

$(\lambda x.\ E)\ \underline{3}$ is a β-redex and could be reduced to $E[\underline{3}/x]$.

4.4.3 η-conversion

A λ-expression (necessarily an abstraction) to which η-reduction can be applied is called an *η-redex*. The rule of η-conversion expresses the property that two functions are equal if they give the same results when applied to the same arguments. This property is called *extensionality* and is discussed further in Section 4.7. For example, η-conversion ensures that $\lambda x.\ (\mathbf{sin}\ x)$ and **sin** denote the same function. More generally, $\lambda V.\ (E\ V)$ denotes the function which when applied to an argument E' returns $(E\ V)[E'/V]$. If V does not occur free in E then $(E\ V)[E'/V] = (E\ E')$. Thus $\lambda V.\ E\ V$ and E both yield the same result, namely $E\ E'$, when applied to the same arguments and hence they denote the same function.

Examples

$$\lambda x.\ \mathbf{add}\ x \underset{\eta}{\longrightarrow} \mathbf{add}$$

$$\lambda y.\ \mathbf{add}\ x\ y \underset{\eta}{\longrightarrow} \mathbf{add}\ x$$

It is *not* the case that

$$\lambda x.\ \mathbf{add}\ x\ x \underset{\eta}{\longrightarrow} \mathbf{add}\ x$$

because x is free in $\mathbf{add}\ x$. □

4.4.4 Generalized conversions

The definitions of $\underset{\alpha}{\longrightarrow}$, $\underset{\beta}{\longrightarrow}$ and $\underset{\eta}{\longrightarrow}$ can be generalized as follows:

- $E_1 \underset{\alpha}{\longrightarrow} E_2$ if E_2 can be got from E_1 by α-converting any subterm.

- $E_1 \underset{\beta}{\longrightarrow} E_2$ if E_2 can be got from E_1 by β-converting any subterm.
- $E_1 \underset{\eta}{\longrightarrow} E_2$ if E_2 can be got from E_1 by η-converting any subterm.

Examples

$$((\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y)\ \underline{3})\ \underline{4} \underset{\beta}{\longrightarrow} (\lambda y.\ \mathbf{add}\ \underline{3}\ y)\ \underline{4}$$

$$(\lambda y.\ \mathbf{add}\ \underline{3}\ y)\ \underline{4} \underset{\beta}{\longrightarrow} \mathbf{add}\ \underline{3}\ \underline{4}$$

□

The first of these is a β-conversion in the generalized sense because $(\lambda y.\ \mathbf{add}\ \underline{3}\ y)\underline{4}$ is obtained from $((\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y)\underline{3})\underline{4}$ (which is not itself a β-redex) by reducing the subexpression $(\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y)\underline{3}$. We will sometimes write a sequence of conversions like the two above as:

$$((\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y)\ \underline{3})\ \underline{4} \underset{\beta}{\longrightarrow} (\lambda y.\ \mathbf{add}\ \underline{3}\ y)\ \underline{4} \underset{\beta}{\longrightarrow} \mathbf{add}\ \underline{3}\ \underline{4}$$

Exercise 42
Which of the three β-reductions below are generalized conversions (i.e. reductions of subexpressions) and which are conversions in the sense defined on page 63? □

(i) $(\lambda x.\ x)\ \underline{1} \underset{\beta}{\longrightarrow} \underline{1}$

(ii) $(\lambda y.\ y)\ ((\lambda x.\ x)\ \underline{1}) \underset{\beta}{\longrightarrow} (\lambda y.\ y)\underline{1} \underset{\beta}{\longrightarrow} \underline{1}$

(iii) $(\lambda y.\ y)\ ((\lambda x.\ x)\ \underline{1}) \underset{\beta}{\longrightarrow} (\lambda x.\ x)\ \underline{1} \underset{\beta}{\longrightarrow} \underline{1}$

In reductions (ii) and (iii) in the exercise above one starts with the same λ-expression, but reduce redexes in different orders.

An important property of β-reductions is that no matter in which order one does them, one always ends up with equivalent results. If there are several disjoint redexes in an expression, one can reduce them in parallel. Note, however, that some reduction sequences may never terminate. This is discussed further in connection with the normalization theorem of Chapter 7. It is a current hot research topic in 'fifth generation computing' to design processors which exploit parallel evaluation to speed up the execution of functional programs.

4.5 Equality of λ-expressions

The three conversion rules preserve the meaning of λ-expressions, i.e. if E_1 can be converted to E_2 then E_1 and E_2 denote the same function. This property of conversion should be intuitively clear. It is possible to give a mathematical definition of the function denoted by a λ-expression and then to prove that this function is unchanged by α-, β- or η-conversion. Doing this is surprisingly difficult [67] and is beyond the scope of this book.

We will simply *define* two λ-expressions to be equal if they can be transformed into each other by a sequence of (forwards or backwards) λ-conversions. It is important to be clear about the difference between *equality* and *identity*. Two λ-expressions are identical if they consist of *exactly* the same sequence of characters; they are equal if one can be converted to the other. For example, $\lambda x.\ x$ is equal to $\lambda y.\ y$, but not identical to it. The following notation is used:

- $E_1 \equiv E_2$ means E_1 and E_2 are identical.
- $E_1 = E_2$ means E_1 and E_2 are equal.

Equality (=) is defined in terms of identity (≡) and conversion ($\xrightarrow[\alpha]{}$, $\xrightarrow[\beta]{}$ and $\xrightarrow[\eta]{}$) as follows.

Equality of λ-expressions

If E and E' are λ-expressions then $E = E'$ if $E \equiv E'$ or there exist expressions $E_1, E_2, \ldots, E_n$ such that:

1. $E \equiv E_1$
2. $E' \equiv E_n$
3. For each i either
   (a) $E_i \xrightarrow[\alpha]{} E_{i+1}$ or $E_i \xrightarrow[\beta]{} E_{i+1}$ or $E_i \xrightarrow[\eta]{} E_{i+1}$ or
   (b) $E_{i+1} \xrightarrow[\alpha]{} E_i$ or $E_{i+1} \xrightarrow[\beta]{} E_i$ or $E_{i+1} \xrightarrow[\eta]{} E_i$.

Examples

$$(\lambda x.\ x)\ \underline{1} = \underline{1}$$

$$(\lambda x.\ x)\ ((\lambda y.\ y)\ \underline{1}) = \underline{1}$$

$$(\lambda x.\ \lambda y.\ \mathbf{add}\ x\ y)\ \underline{3}\ \underline{4} = \mathbf{add}\ \underline{3}\ \underline{4}$$

□

From the definition of = it follows that:

(i) For any E it is the case that $E = E$ (equality is *reflexive*).

(ii) If $E = E'$, then $E' = E$ (equality is *symmetric*).

(iii) If $E = E'$ and $E' = E''$, then $E = E''$ (equality is *transitive*).

If a relation is reflexive, symmetric and transitive then it is called an *equivalence relation*. Thus = is an equivalence relation.

Another important property of = is that if $E_1 = E_2$ and if E_1' and E_2' are two λ-expressions that only differ in that where one contains E_1 the other contains E_2, then $E_1' = E_2'$. This property is called *Leibnitz's law*. It holds because the same sequence of reduction for getting from E_1 to E_2 can be used for getting from E_1' to E_2'. For example, if $E_1 = E_2$, then by Leibnitz's law $\lambda V.\ E_1 = \lambda V.\ E_2$.

It is essential for the substitutions in the α- and β-reductions to be valid. The validity requirement disallows, for example, $\lambda x.\ (\lambda y.\ x)$ being α-reduced to $\lambda y.\ (\lambda y.\ y)$ (since y becomes bound after substitution for x in $\lambda y.\ x$). If this invalid substitution were permitted, then it would follow by the definition of = that:

$$\lambda x.\ \lambda y.\ x = \lambda y.\ \lambda y.\ y$$

But then since:

$$(\lambda x.\ (\lambda y.\ x))\ \underline{1}\ \underline{2} \xrightarrow[\beta]{} (\lambda y.\ \underline{1})\ \underline{2} \xrightarrow[\beta]{} \underline{1}$$

and

$$(\lambda y.\ (\lambda y.\ y))\ \underline{1}\ \underline{2} \xrightarrow[\beta]{} (\lambda y.\ y)\ \underline{2} \xrightarrow[\beta]{} \underline{2}$$

one would be forced to conclude that $\underline{1} = \underline{2}$. More generally by replacing $\underline{1}$ and $\underline{2}$ by any two expressions, it could be shown that any two expressions are equal!

Exercise 43
Find an example which shows that if substitutions in β-reductions are allowed to be invalid, then it follows that any two λ-expressions are equal. □

Example: If $V_1, V_2, \ldots, V_n$ are all distinct and none of them occur free in any of $E_1, E_2, \ldots, E_n$, then

$$\begin{aligned}
&(\lambda V_1\ V_2 \cdots V_n.\ E)\ E_1\ E_2 \cdots E_n \\
&\quad = \quad ((\lambda V_1.\ (\lambda V_2 \cdots V_n.\ E))E_1)\ E_2 \cdots E_n \\
&\quad \xrightarrow[\beta]{} \quad ((\lambda V_2 \cdots V_n.\ E)[E_1/V_1])\ E_2 \cdots E_n \\
&\quad = \quad (\lambda V_2 \ldots V_n.\ E[E_1/V_1])E_2 \cdots E_n \\
&\qquad\qquad \vdots \\
&\quad = \quad E[E_1/V_1][E_2/V_2] \cdots [E_n/V_n]
\end{aligned}$$

□

Exercise 44
In the last example, where was the assumption used that $V_1, V_2, \ldots, V_n$ are all distinct and that none of them occur free in any of $E_1, E_2, \ldots, E_n$? □

Exercise 45
Find an example to show that if $V_1 = V_2$, then even if V_2 is not free in E_1, it is not necessarily the case that:

$$(\lambda V_1 V_2.E)\ E_1\ E_2 = E[E_1/V_1][E_2/V_2]$$

□

Exercise 46
Find an example to show that if $V_1 \neq V_2$, but V_2 occurs free in E_1, then it is not necessarily the case that:

$$(\lambda V_1 V_2.\ E)\ E_1\ E_2 = E[E_1/V_1][E_2/V_2]$$

□

4.6 The $\longrightarrow$ relation

In the previous section $E_1 = E_2$ was defined to mean that E_2 could be obtained from E_1 by a sequence of forwards *or backwards* conversions. A special case of this is when E_2 is got from E_1 using only forwards conversions. This is written $E_1 \longrightarrow E_2$.

Definition of $\longrightarrow$

If E and E' are λ-expressions, then $E \longrightarrow E'$ if $E \equiv E'$ or there exist expressions $E_1, E_2, \ldots, E_n$ such that:

1. $E \equiv E_1$
2. $E' \equiv E_n$
3. For each i either $E_i \xrightarrow[\alpha]{} E_{i+1}$ or $E_i \xrightarrow[\beta]{} E_{i+1}$ or $E_i \xrightarrow[\eta]{} E_{i+1}$.

Notice that the definition of $\longrightarrow$ is just like the definition of $=$ on page 68 except that part (b) of 3 is missing.

Exercise 47
Find E, E' such that $E = E'$ but it is not the case that $E \longrightarrow E'$. □

Exercise 48
[very hard!] Show that if $E_1 = E_2$, then there exists E such that $E_1 \longrightarrow E$ and $E_2 \longrightarrow E$. (This property is called the Church-Rosser theorem. Some of its consequences are discussed in Chapter 7.) □

4.7 Extensionality

Suppose V does not occur free in E_1 or E_2 and

$$E_1\ V = E_2\ V$$

Then by Leibnitz's law (see page 69)

$$\lambda V.\ E_1\ V = \lambda V.\ E_2\ V$$

so by η-reduction applied to both sides

$$E_1 = E_2$$

It is often convenient to prove that two λ-expressions are equal using this property, i.e. to prove $E_1 = E_2$ by proving $E_1\ V = E_2\ V$ for some V not occuring free in E_1 or E_2. We will refer to such proofs as being by *extensionality*.

Exercise 49
Show that

$$(\lambda f\ g\ x.\ f\ x\ (g\ x))\ (\lambda x\ y.\ x)\ (\lambda x\ y.\ x)\ =\ \lambda x.\ x$$

□

4.8 Substitution

At the beginning of Section 4.4 $E[E'/V]$ was defined to mean the result of substituting E' for each *free* occurrence of V in E. The substitution was said to be valid if no free variable in E' became bound in $E[E'/I]$. In the definitions of α- and β-conversion, it was stipulated that the substitutions involved must be valid. Thus, for example, it was only the case that

$$(\lambda V.\ E_1)\ E_2 \xrightarrow[\beta]{} E_1[E_2/V]$$

as long as the substitution $E_1[E_2/V]$ was valid.

It is very convenient to extend the meaning of $E[E'/V]$ so that we don't have to worry about validity. This is achieved by the definition below which has the property that for *all* expressions E, E_1 and E_2 and *all* variables V and V'

$$(\lambda V.\ E_1)\ E_2 \longrightarrow E_1[E_2/V] \quad \text{and} \quad \lambda V.\ E \longrightarrow \lambda V'.\ E[V'/V]$$

To ensure this property holds, $E[E'/V]$ is defined recursively on the structure of E as follows:

E	$E[E'/V]$
V	E'
V' (where $V \neq V'$)	V'
$E_1\ E_2$	$E_1[E'/V]\ E_2[E'/V]$
$\lambda V.\ E_1$	$\lambda V.\ E_1$
$\lambda V'.\ E_1$ (where $V \neq V'$ and V' is not free in E')	$\lambda V'.\ E_1[E'/V]$
$\lambda V'.\ E_1$ (where $V \neq V'$ and V' is free in E')	$\lambda V''.\ E_1[V''/V'][E'/V]$ where V'' is a variable not free in E' or E_1

This particular definition of $E[E'/V]$ is based on (but not identical to) the one in Appendix C of [4]. A LISP implementation of it is given in Chapter 12 on page 228.

To illustrate how this works consider $(\lambda y.\ y\ x)[y/x]$. Since y is free in $y\ x$ we must use the last case of the table above. Since x does not occur in $y\ x$ or y,

$$(\lambda y.\ y\ x)[y/x] \equiv \lambda z.\ (y\ x)[z/y][y/x] \equiv \lambda z.\ (z\ x)[y/x] \equiv \lambda z.\ z\ y$$

In the last line of the table above, the particular choice of V'' is not specified. Any variable not occurring in E' or E_1 will do. In Chapter 12 an implementation of substitution in LISP is given.

A good discussion of substitution can be found in the book by Hindley and Seldin [31] where various technical properties are stated and proved. The following exercise is taken from that book.

Exercise 50
Use the table above to work out

(i) $(\lambda y.\ x\ (\lambda x.\ x))[(\lambda y.\ y\ x)/x]$.

(ii) $(y\ (\lambda z.\ x\ z))[(\lambda y.\ z\ y)/x]$.

□

It is straightforward, but rather tedious, to prove from the definition of $E[E'/V]$ just given that indeed

$$(\lambda V.\ E_1)\ E_2 \longrightarrow E_1[E_2/V] \quad \text{and} \quad \lambda V.\ E \longrightarrow \lambda V'.\ E[V'/V]$$

for all expressions E, E_1 and E_2 and all variables V and V'.

In Chapter 8 it will be shown how the theory of combinators can be used to decompose the complexities of substitution into simpler operations. Instead of combinators it is possible to use the so-called *nameless terms* of De Bruijn [8]. De Bruijn's idea is that variables can be thought of as 'pointers' to the λs that bind them. Instead of 'labelling' λs with names (i.e. bound variables) and then pointing to them via these names, one can point to the appropriate λ by giving the number of levels 'upwards' needed to reach it. For example, $\lambda x.\ \lambda y.\ x\ y$ would be represented by $\lambda\lambda 2\ 1$. As a more complicated example, consider the expression below in which we indicate the number of levels separating a variable from the λ that binds it.

$$\lambda x.\ \lambda y.\ \overset{2}{x}\ \overset{1}{y}\ (\lambda y.\ \overset{1}{x}\ \overset{1}{y}\ y)$$

In De Bruijn's notation this is $\lambda\lambda 2\ 1\ \lambda 3\ 1\ 1$.

A free variable in an expression is represented by a number bigger than the depth of λs above it; different free variables being assigned different numbers. For example,

$$\lambda x.\ (\lambda y.\ y\ x\ z)\ x\ y\ w$$

would be represented by

$$\lambda(\lambda 1\ 2\ 3)\ 1\ 2\ 4$$

Since there are only two λs above the occurrence of 3, this number must denote a free variable; similarly there is only one λ above the second occurrence of 2 and the occurrence of 4, so these too must be free variables. Note that 2 could not be used to represent w since this had already been used to represent the free y; we thus chose the first available number bigger than 2 (3 was already in use representing z).

Care must be taken to assign big enough numbers to free variables. For example, the first occurrence of z in $\lambda x.\ z\ (\lambda y.\ z)$ could be represented by 2, but the second occurrence requires 3; since they are the same variable we must use 3.

Example: With De Bruijn's scheme $\lambda x.\ x\ (\lambda y.\ x\ y\ y)$ would be represented by $\lambda 1(\lambda 2\ 1\ 1)$. □

Exercise 51
What λ-expression is represented by $\lambda 2(\lambda 2)$? □

Exercise 52
Describe an algorithm for computing the De Bruijn representation of the expression $E[E'/V]$ from the representations of E and E'. □

Bibliography

[1] Alagić, S. and Arbib, M.A., *The Design of Well-structured and Correct Programs*, Springer-Verlag, 1978.

[2] Augustsson, L., 'A compiler for lazy ML', in *Proceedings of the ACM Symposium on LISP and Functional Programming*, Austin, pp. 218-227, 1984.

[3] Backhouse, R.C., *Program Construction and Verification*, Prentice Hall, 1986.

[4] Barendregt, H.P., *The Lambda Calculus* (revised edition), Studies in Logic 103, North-Holland, Amsterdam, 1984.

[5] Barron, D.W. and Strachey, C. 'Programming', in Fox, L. (ed.), *Advances in Programming and Non-numerical Computation* (Chapter 3), Pergamon Press, 1966.

[6] Bird, R. and Wadler, P., *An Introduction to Functional Programming*, Prentice Hall, 1988.

[7] Boyer, R.S. and Moore, J S., *A Computational Logic*, Academic Press, 1979.

[8] De Bruijn, N.G., 'Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation', *Indag. Math.*, **34**, pp. 381-392, 1972.

[9] Burge, W., *Recursive Programming Techniques*, Addison-Wesley, 1975.

[10] Chang, C. and Lee, R.C., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[11] Clarke, E.M. Jr., 'The characterization problem for Hoare logics', in Hoare, C.A.R. and Shepherdson, J.C. (eds), *Mathematical Logic and Programming Languages*, Prentice Hall, 1985.

[12] Clarke, T.J.W., et al., 'SKIM – the S, K, I Reduction Machine', in *Proceedings of the 1980 ACM LISP Conference*, pp. 128-135, 1980.

[13] Cohn, A.J., 'High level proof in LCF', in Joyner, W.H., Jr. (ed.), *Fourth Workshop on Automated Deduction*, pp. 73-80, 1979.

[14] Curry, H.B. and Feys, R., *Combinatory Logic, Vol. I*, North Holland, Amsterdam, 1958.

[15] Curry, H.B., Hindley, J.R. and Seldin, J.P. *Combinatory Logic, Vol. II*, Studies in Logic 65, North Holland, Amsterdam, 1972.

[16] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, 1976.

[17] Fairbairn, J. and Wray, S.C., 'Code generation techniques for functional languages', in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, Cambridge, Mass., pp. 94-104, 1986.

[18] Floyd, R.W., 'Assigning meanings to programs', in Schwartz, J.T. (ed.), *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19* (American Mathematical Society), Providence, pp. 19-32, 1967.

[19] Genesereth, M.R and Nilsson, N.J., *Logical Foundations of Artificial Intelligence*, Morgan Kaufman Publishers, Los Altos, 1987.

[20] Good, D.I., 'Mechanical proofs about computer programs', in Hoare, C.A.R. and Shepherdson, J.C. (eds), *Mathematical Logic and Programming Languages*, Prentice Hall, 1985.

[21] Gordon, M.J.C., 'On the power of list iteration', *The Computer Journal*, **22**, No. 4, 1979.

[22] Gordon, M.J.C., 'Operational reasoning and denotational semantics', in *Proceedings of the International Symposium on Proving and Improving Programs, Arc-et-Senans*, pp. 83-98, IRIA, Rocquencourt, France, 1975.

[23] Gordon, M.J.C.,*The Denotational Description of Programming Languages*, Springer-Verlag, 1979.

[24] Gordon, M.J.C., 'Representing a logic in the LCF metalanguage', in Néel, D. (ed.), *Tools and Notions for Program Construction*, Cambridge University Press, 1982.

[25] Gordon, M.J.C., Milner, A.J.R.G. and Wadsworth, C.P., *Edinburgh LCF: a mechanized logic of computation*, Springer Lecture Notes in Computer Science, Springer-Verlag, 1979.

[26] Gries, D., *The Science of Programming*, Springer-Verlag, 1981.

[27] Hehner, E.C.R., *The Logic of Programming*, Prentice Hall, 1984.

[28] Henderson, P., *Functional Programming, Application and Implementation*, Prentice Hall, 1980.

[29] Henderson, P. and Morris, J.M., 'A lazy evaluator' in *Proceedings of The Third Symposium on the Principles of Programming Languages, Atlanta, Georgia*, pp. 95-103, 1976.

[30] Hindley, J.R., 'Combinatory reductions and lambda-reductions compared', *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, **23**, pp. 169-180, 1977.

[31] Hindley, J.R. and Seldin, J.P., *Introduction to Combinators and λ-Calculus*, London Mathematical Society Student Texts, 1, Cambridge University Press, 1986.

[32] Hoare, C.A.R., 'An axiomatic basis for computer programming', *Communications of the ACM*, **12**, pp. 576-583, October 1969.

[33] Hoare, C.A.R., 'A Note on the FOR Statement', BIT, **12**, pp. 334-341, 1972.

[34] Hoare, C.A.R., 'Programs are predicates', in Hoare, C.A.R. and Shepherdson, J.C. (eds), *Mathematical Logic and Programming Languages*, Prentice Hall, 1985.

[35] Hoare, C.A.R. and Shepherdson, J.C. (eds), *Mathematical Logic and Programming Languages*, Prentice Hall, 1985.

[36] Hughes, R.J.M., 'Super combinators: a new implementation method for applicative languages', in *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh*, 1982.

[37] Jones, C.B., *Systematic Software Development Using VDM*, Prentice Hall, 1986.

[38] Joyce, E., 'Software bugs: a matter of life and liability', *Datamation*, **33**, No. 10, May 15, 1987.

[39] Kleene, S.C., 'λ-definability and recursiveness', *Duke Math. J.*, pp. 340-353, 1936.

[40] Krishnamurthy, E.V. and Vickers, B.P., 'Compact numeral representation with combinators', *The Journal of Symbolic Logic*, **52**, No. 2, pp. 519-525, June 1987.

[41] Lamport, L., LaTeX: *A Document Preparation System*, Addison-Wesley, 1986.

[42] Landin, P.J., 'The next 700 programming languages', *Comm. Assoc. Comput. Mach.*, 9, pp. 157-164, 1966.

[43] Lévy, J.-J., 'Optimal reductions in the lambda calculus', in Hindley, J.R. and Seldin, J.P. (eds), *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, Academic Press, New York and London, 1980.

[44] Ligler, G.T., 'A mathematical approach to language design', in *Proceedings of the Second ACM Symposium on Principles of Programming Languages*, pp. 41-53, 1985.

[45] Loeckx, J. and Sieber, K., *The Foundations of Program Verification*, John Wiley & Sons Ltd. and B.G. Teubner, Stuttgart, 1984.

[46] London, R.L., et al. 'Proof rules for the programming language Euclid', *Acta Informatica*, **10**, No. 1, 1978.

[47] Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[48] Manna, Z. and Waldinger, R., 'Problematic features of programming languages: a situational-logic approach', *Acta Informatica*, **16**, pp. 371-426, 1981.

[49] Manna, Z. and Waldinger, R., *The Logical Basis for Computer Programming*, Addison-Wesley, 1985.

[50] Mason, I.A., *The Semantics of Destructive LISP*, CSLI Lecture Notes 5, CSLI Publications, Ventura Hall, Stanford University, 1986.

[51] Mauny, M. and Suárez, A., 'Implementing functional languages in the categorical abstract machine', in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pp. 266-278, Cambridge, Mass., 1986.

[52] McCarthy, J., 'History of LISP', in *Proceedings of the ACM SIGPLAN History of Programming Languages Conference*, published in *SIGPLAN Notices*, 13, Number 8, August 1978.

[53] McCarthy, J. et al., *LISP 1.5 Manual*, The M.I.T. Press, 1965.

[54] Milne, R.E. and Strachey, C., *A Theory of Programming Language Semantics* (2 volumes; second one covers compiler correctness), Chapman and Hall, London (and Wiley, New York), 1975.

[55] Milner, A.J.R.G., 'A proposal for Standard ML', in *Proceedings of the ACM Symposium on LISP and Functional Programming, Austin*, 1984.

[56] Minsky, M., *Computation: Finite and Infinite Machines*, Prentice Hall, 1967.

[57] Morris, J.H., *Lambda Calculus Models of Programming Languages*, Ph.D. Dissertation, M.I.T., 1968.

[58] Nagel, E. and Newman, J.R., *Gödel's Proof*, Routledge & Kegan Paul, London, 1959.

[59] Paulson, L.C., 'A higher-order implementation of rewriting', *Science of Computer Programming*, **3**, pp 143-170, 1985.

[60] Paulson, L.C., *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge University Press, 1987.

[61] Peyton Jones, S.L., *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

[62] Polak, W., *Compiler Specification and Verification*, Springer-Verlag Lecture Notes in Computer Science, No. 124, 1981.

[63] Reynolds, J.C., *The Craft of Programming*, Prentice Hall, London, 1981.

[64] Schoenfield, J.R., *Mathematical Logic*, Addison-Wesley, Reading, Mass., 1967.

[65] Schönfinkel, M., 'Über die Bausteine der mathematischen Logik', *Math. Annalen* **92**, pp. 305-316, 1924. Translation printed as 'On the building blocks of mathematical logic', in van Heijenoort, J. (ed.), *From Frege to Gödel*, Harvard University Press, 1967.

[66] Scott, D.S., 'Models for various type free calculi', in Suppes, P. et al. (eds), *Logic, Methodology and Philosophy of ScienceIV*, Studies in Logic 74, North-Holland, Amsterdam, 1973.

[67] Stoy, J.E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, M.I.T. Press, 1977.

[68] Turner, D.A., 'A new implementation technique for applicative languages', *Software Practice and Experience*, 9, pp. 31-49, 1979.

[69] Turner, D.A., 'Another algorithm for bracket abstraction', *The Journal of Symbolic Logic*, 44, No. 2, pp. 267-270, June 1979.

[70] Turner, D.A., 'Functional programs as executable specifications', in Hoare, C.A.R. and Shepherdson, J.C. (eds), *Mathematical Logic and Programming Languages*, Prentice Hall, 1985.

[71] Wadsworth, C.P., 'The relation between computational and denotational properties for Scott's D_∞-models of the lambda-calculus', *S.I.A.M. Journal of Computing*, 5, pp. 488-521, 1976.

[72] Wadsworth, C.P., 'Some unusual λ-calculus numeral systems', in Hindley, J.R. and Seldin, J.P. (eds), *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, Academic Press, New York and London, 1980.

[73] Wilensky, R., *LISPcraft*, W. W. Norton & Company, Inc., 1984.

[74] Wos, L. et al., *Automated Reasoning: Introduction and Applications*, Englewood Cliffs, NJ: Prentice Hall, 1984.

Index