Emacs For Aldor: An External Learning System

Jeremy (Ya-Yu) Hsieh

B.Sc., University Of Northern British Columbia, 2003

Thesis Submitted In Partial Fulfillment Of

The Requirements For The Degree Of

Master of Science

in

Mathematical, Computer, And Physical Sciences

(Computer Science)

The University Of Northern British Columbia

May 2006

© Jeremy (Ya-Yu) Hsieh, 2006

Abstract

In this thesis, I observed and established communications between Emacs, the external lexical analyzers, and the parser. There are two versions of the lexical analyzers: internal and external. The internal lexical analyzer was implemented in the programming language of Emacs-LISP. The core of the external lexical analyzer was implemented in `Flex`. For the parser, `Bison` and `Flex` were used. This thesis describes (1) details of each component; (2) comparisons of the difference between internal and external information processing; (3) experimental advantages and disadvantages of each version; (4) evaluations of the efficiencies based on running time, memory usage, features presentations, and other resources usage; (5) what needs to be done to achieve such communications; and (6) Aldor syntaxes and grammar.

Table of Contents

Abstract **ii**

Table of Contents **iii**

List of Tables **v**

List of Figures **vi**

1 Introduction **1**

1.1 What is ALDOR? 2

1.1.1 Lexical Features of ALDOR 4

1.2 What is EMACS? 9

1.3 Literature Review 10

1.4 Purpose of my Work 11

1.5 Emacs Concepts 12

1.6 Version Details 17

2 The Internal Lexical Analyzer **18**

2.1 Overview of Building the Internal Lexical Analyzer 19

2.2 Method 20

2.3 Efficiency 26

2.4 Correctness 27

2.5 Problems Encountered and their Solutions 27

2.6 Conclusion and Summary 29

3 The External Lexical Analyzer **31**

3.1 Overview of Building the External Lexical Analyzer 31

3.2 Method 34

3.3 Communication or Pipes 40

3.4 Efficiency 41

3.5 Correctness 42

3.6 Problems Encountered and their Solutions 44

3.7 Conclusion and Summary 52

4 The Aldor Parser **53**
4.1 Overview of Building the Parser . . . 54
4.2 Method . . . 55
4.3 Communication . . . 64
4.4 Efficiency . . . 66
4.5 Correctness . . . 68
4.6 Problems Encountered and their Solutions . . . 68
4.7 Future Work . . . 73
4.8 Conclusion and Summary . . . 74

5 Aldor Mode **76**
5.1 Progress and Methods . . . 77
5.2 Syntax-based Colouring . . . 78
5.3 Updating Algorithms for the Aldor Mode . . . 79
5.4 Features . . . 81
5.5 Efficiency . . . 82
5.6 Problems Encountered and their Solutions . . . 83
5.7 Conclusion and Summary . . . 83

6 Conclusions and Summary **84**
6.1 Summary of my Research Timeline . . . 85
6.2 What can People Learn from my Work? . . . 86
6.3 Comparison of Lexical Analyzers . . . 86
6.4 Further Improvement and Future Work . . . 87

Bibliography **88**

A A Lexical Structure for Aldor **92**
A.1 Lexical Structure . . . 92
A.1.1 Characters . . . 92
A.1.2 The Escape Character . . . 93
A.1.3 Tokens . . . 93
A.2 Differences Between the Implemented Lexical Scanner and the Aldor Lexical Structure . . . 96

B Context Free Grammars for Aldor **98**

C Abstract Syntax for Aldor **112**

D Codes for my Work **117**

E UML Diagram of External Lexical Analyzer **118**

F UML Diagram of Parser **120**

List of Tables

1.1 Token Categories for ALDOR Tokens 5
1.2 EMACS LISP and FLEX Regular Expressions for ALDOR Tokens. . . 6
1.3 EMACS-LISP and FLEX Regular Expressions for ALDOR Floating Point Tokens. 7
1.4 The Version Details of the Programs Used 16

5.1 The Colour Association Used by the ALDOR Mode 78

List of Figures

2.1 A Regular Expression to Match ALDOR Reserved Words 22
2.2 A Regular Expression to Match ALDOR String Tokens 23
2.3 A Regular Expression to Match ALDOR Intermediate Tokens 24
2.4 A Regular Expression to Match ALDOR Floating Point Literals Tokens 25
2.5 A Regular Expression to Match ALDOR Pre-document Tokens 25
2.6 ALDOR Token Precedence 28

3.1 Definitions for Some ALDOR Tokenizer Components 37
3.2 `Actions` to Take When ALDOR Finds a Token 38
3.3 Timings of the Internal and External Lexical Analyzers 43
3.4 The `aldor-process-done-sentinel` Function 47
3.5 The `aldor-restore-point-filter` Function 49
3.6 The `aldor-get-1-value` Function 51

4.1 Function that Prints the Start and End Point 58
4.2 Example of Token Handling 59
4.3 Example of `Bison` Codes for the Update "On The Fly" Algorithm . . 61
4.4 Example of Tree Structure in `Bison` 63
4.5 Timing Data for Tree Structure Parser 67
4.6 Rules to Resolve Ambiguous Grammars 69
4.7 A Tree Structure with a `NULL` Problem 70
4.8 Solution for the Tree Structure with `NULL` Problem 71
4.9 A Tree Structure with a Sibling Problem 72
4.10 Solution for the Tree Structure with Sibling Problem 72

E.1 UML Diagram for the External Lexical Analyzer 119

F.1 UML Diagram for the Parser 121

Chapter 1

Introduction

There are many different programming languages. As the computer world expands, programmers require more powerful and up-to-date programming languages to produce effective executable programs. For this reason, some programming languages have been modified and upgraded to keep pace with the current computer programming world (for example, C to C^{++}, and its transformation into C#). As a result, some new programming languages are created and some of the old and out-of-date programming languages become less popular. Since there are so many different programming languages, good programming text editors will make an important difference in the coding environments for programmers. Good text editors will not only help programmers program faster, and more efficiently, but also improve the readability and quality of program code.

I have designed a protocol which helps programmers to expand the language modes in the EMACS text editor. Currently, the language modes which are supported by EMACS are written in EMACS-LISP and interpreted by the EMACS text editor itself. I also developed a method which enable the EMACS text editor to interact with external sources. In this thesis the sources are an external lexical analyzer and an external parser.

Lexical analysis is the name given to the processing of an input sequence of characters (such as the source code of a computer program) to produce, as output, a sequence of symbols called “lexical tokens”, or just “tokens”. A lexical analyzer makes it possible for EMACS to do syntax-based colouring. Additionally, it provides information which is required by a parser.

The parser starts the process of analyzing an input sequence (read from a file or a keyboard, for example) in order to determine its grammatical structure with respect to a given formal grammar. It transforms input text into a data structure, usually a tree, which is suitable for later processing and which captures the implied hierarchy of the input. Generally, parsers operate in two stages, first identifying the meaningful tokens in the input, and then building a parse tree from those tokens.

The programming language with which I chose to demonstrate the algorithms is called ALDOR. The result of my demonstration is an ALDOR mode for EMACS.

In this chapter I first describe the programming language ALDOR, and then describe what text editors are. My reason for choosing EMACS as my research editor instead of some other text editor are discussed. Following that is a survey of previous work. Finally, I finish up this chapter with the explanation of basic components which I need in my work.

1.1 What is Aldor?

ALDOR was derived from another program language, called `AXIOM`. This language is very young and new; therefore, there is not that much related information which one can find from the Internet or in libraries. My main resources for the programming language ALDOR are from ALDOR's official site (`http://www.aldor.org/`), my supervisor and discussions with some ALDOR development people through e-mail. Additionally, the following references also gave me a better understanding of

the programming language ALDOR: "ALDOR User Guide" [27] and "libaldor User Guide and Reference Manual" [3]. Some of the presentation papers and technical reports [2, 26, 24, 22, 23, 7, 25] are useful for programmers to understand the programming language ALDOR and implement some simple programs. If a programmer wishes to learn the details of ALDOR, [29] discussed categories in ALDOR and [18] discussed the type system.

The programming language ALDOR is a kind of mathematical programming language which was designed to do tasks similar to MAPLE but not like those of FORTRAN. It is an imperative programming language with first-class types and which supports functional programming. ALDOR also has following characteristics: it has a two-level object model with inheritance (c.f. HASKELL); it allows overloading of symbol names; types and functions are (constant) values; and it has generators, post facto extensions, and other non-standard language features. It has foreign language interfaces for: LISP, C, C^{++}, and FORTRAN 77. Moreover, it provides an automatic garbage collection feature; not to mention the ability to compile to other languages like FOAM, LISP, or C.

The primary considerations for the programming language ALDOR are generality, composability, efficiency, and interoperability. ALDOR looks like both functional and modern programming languages. The structure of the language looks like C^{++}, JAVA, C, BASIC, among other modern programming languages — with a semicolon at the end of statements, matched parentheses, and control loops. On the other hand, the features it provides and the structure of the language itself are similar to some of functional programming languages, such as SML, FORTRAN, and MAPLE. These features make ALDOR a very good testing language to make use of. Finally, ALDOR can inter-operate with many other languages, like C, C^{++}, JAVA, and others.

The programming language ALDOR has an LALR(1) grammar. An LALR (Look-

Ahead Left to right and produce a Rightmost derivation) parser is a specialized form of LR parser that can deal with more context-free grammars than simple LR parsers but fewer than LR(1) parsers can [28]. It is a very popular type of parser because it gives a good trade-off between the number of grammars it can deal with and the size of the parsing tables it requires.

There are some principles of the programming language ALDOR discussed in [15]. One of these is: "All objects are first-class". For this reason, programmers can use variables whenever and wherever they wish. It is also legal to write functions inside other functions (nested functions). Another principle is: "Types, both domains and categories, are treated in the same way as any other objects"[15]. In other words, in ALDOR, types get treated just like any other variables or constants. Hence, ALDOR programmers can place type strings on the left hand side of assignment operators and assign some values to them — this is a completely legal operation in ALDOR. This principle increases the difficulty involved in achieving syntax-based colouring for the editor mode. If type strings can be variables, then whether they are syntactically coloured as a type or as a variable ought to depend on a parse and not just on a lexical class. As a result, it is essential to have an ALDOR parser to provide additional information.

1.1.1 Lexical Features of Aldor

There are some ALDOR tokens which I explain here for clarity in later chapters. In this section, Table 1.1 describes the token types and Table 1.2 defines ALDOR tokens and their regular expression forms.

The "_" character has been treated specially in ALDOR programming language. In other programming languages, the "_" character is treated as part of identifiers; nevertheless, ALDOR treats the character as an escape character. An escape character followed by one or more white space characters causes the white space to be

Token Categories

Category	*Example*	*Description*
Reserved Keywords	`add`	The reserved words for ALDOR.
Definable Keywords	`case`	Keywords that the user can define.
Class	`MachineInteger`	Identifiers that are likely to identify categories.
Import functions	`include`	Identifiers that are like to identify the names of import functions.
Pre-Document	`+++ I am Pre-Doc`	Start with three plus signs. Used to document the code, appear before the code they describe. Can be extracted by automatic documentation tools.
Post-Document	`++ I am Post-Doc`	Start with two plus signs. Used to document the code. Appear after the code it described. Can be extracted by automatic documentation tools.
Comment	`-- I am Comment`	Start with two minus signs. Used to comment the code. Ignored by automatic documentation tools.
String	`"a string"`	String literals. Start and end with double quotes. Represent character data.
Identifier	`id_count`	"Variable" tokens. Must not be reserved words or operators.
Float	`1.234e56`	Floating point literals.
Integer	`3`	Integer literals.
Definable Operator	`#`	Operators that can be re-defined by programmers.
Reserved Operator	`==`	Reserved operators that can not be re-defined.
Future Operator	`[|`	Operators that may be included in ALDOR in the future.

Table 1.1: Token Categories for ALDOR Tokens

Classification of ALDOR tokens

<table>
<tr><th colspan="2">Class</th></tr>
<tr><th>LISP rx S-expression</th><th>FLEX syntax</th></tr>
<tr><td colspan="2">Reserved Keywords, Definable Keywords, Class, Import functions
Hard coded for both</td></tr>
<tr><td colspan="2">Pre-Document</td></tr>
<tr><td>(and (group (repeat 3 "+"))
(1+ not-newline))</td><td>"+++".*</td></tr>
<tr><td colspan="2">Post-Document</td></tr>
<tr><td>(and (group (repeat 2 "+"))
(1+ not-newline))</td><td>"++".*</td></tr>
<tr><td colspan="2">Comment</td></tr>
<tr><td>(and (group (repeat 2 "-"))
(1+ not-newline))</td><td>"--".*</td></tr>
<tr><td colspan="2">String</td></tr>
<tr><td>(and "\""
(0+ (or (and "_" (1+ space))
(and "_" (not space))
(not (any "_\n\""))))
"\"")</td><td>\"[^\"]*\"</td></tr>
<tr><td colspan="2">Identifier</td></tr>
<tr><td>(and
word-start
(or
(and "0" word-end)
(and "1" word-end)
(and (any "a-zA-Z")
(* (or (any
"a-zA-Z0-9!?|"))))))</td><td>"0"
|"1"
| {alpha} ({alpha}
| {digit}| [!?|])*</td></tr>
<tr><td colspan="2">Float
See Table 1.3</td></tr>
<tr><td colspan="2">Definable Operator, Reserved Operator, Future Operator
Hard coded for both</td></tr>
</table>

Table 1.2: EMACS LISP and FLEX Regular Expressions for ALDOR Tokens.

Aldor FLOAT tokens

<table>
<tr><th colspan="2">Float</th></tr>
<tr><td>(or
(and (* digit) "."
(+ ,esc-digit)
(zero-or-one ,expon))
(and digit
(* ,esc-digit) "."
(* ,esc-digit)
(zero-or-one ,expon))
(and digit
(* ,esc-digit) ,expon)
(and ,radix
(or
(and
(* ,esc-long-digit)
"."
(+ ,esc-long-digit)
(zero-or-more ,expon))
(and
(+ ,esc-long-digit)
"."
(* ,esc-long-digit)
(zero-or-more ,expon))
(and (+ ,esc-long-digit)
,expon))))</td><td>{digit}*"."
{esc_digit}+
{expon}? |

{digit}+"."
{esc_digit}*
{expon}? |

{digit}+
{expon} |

{radix}
{esc_long_digit}*
"."
{esc_long_digit}+
{expon}? |

{radix}
{esc_long_digit}+
"."
{esc_long_digit}*
{expon}? |

{radix}
{esc_long_digit}+
{expon}</td></tr>
</table>

Table 1.3: EMACS-LISP and FLEX Regular Expressions for ALDOR Floating Point Tokens.

ignored. [27, *pp*.241]. It also has the following effects on other tokens. It converts keywords into non-reserved identifiers; it allows visual grouping in integer and floating-point literals; it also allows arbitrary characters to be included in strings and identifiers.

Because ALDOR is a programming language for computer algebra it has some unusual tokenization rules for numbers. To begin with, "0" and "1" are identifiers rather than integer literals. This allows domains to define 0 and 1 without being forced to define all integer literals. In ALDOR it is also possible for domains to give new meanings to integer and floating-point literals. These literals may contain a radix (between 2 and 36), which allows a programmer to use different number bases without declaring them specially or doing conversions. However, these powerful features increase the difficulty of recognition of ALDOR tokens.

In ALDOR, it is valid for an identifier to contains the symbols "`!`", "`?`", or "`|`"; however, none of these symbols is allowed to be the first character of an identifier. The final set of tokens which I want to introduce is pre-document, post-document, comment, and string tokens. ALDOR pre-document tokens are defined as any symbol sequence following three plus (`+++`) signs. ALDOR post-document tokens are defined as any symbol sequence following by two plus (`++`) signs. ALDOR comments are defined as symbol sequences which start with two minus (`--`) signs. Similar to the way `Javadoc` handles comments in Java programs, ALDOR pre-document and post-document tokens can be extracted by some ALDOR tools to create documentation. Finally, a string token is defined as any character sequence which starts and ends with a double quote (`"`) character, where the sequence between the start and end quotes contains no un-escaped double quote characters or new-lines. More details about ALDOR tokens are given in Appendix A.

1.2 What is Emacs?

"Emacs is the extensible, customizable, self-documenting real-time display editor"

— Emacs User Guide ([12])

It is a "real-time" editor because display is updated very frequently, usually after each character or pair of characters typed by a user. (Most text editors do this.)

"Self-documenting" means that at any time, users can type a special command, to find out what their options are. There are complete built-in information files for the editor and the programming language, as well as keyword accessible help for commands, functions, and options. Furthermore, the documentation is customizable in that users can add to, delete, or modify the documentation at any point of time.

"Customizable" means that users can change the definitions of Emacs commands. Therefore it can satisfy different kinds of users' personal habits and preferences. For this reason, users can turn Emacs into a personal text editor which suits them best. This is one good feature which most text editors do not provide.

"Extensible" means that users or programmers can go beyond simple customization and write entirely new commands or programs in Emacs. Emacs is an open source text editor, which allows programmers to modify the core, create extensions for it, and fix bugs.

Emacs was chosen for this project because it makes more sense to extend an existing editor, rather than to write a new one. Emacs is not only a text editor which provides many powerful features and is open source, but it also has a large community which uses it. For these reasons, Emacs was the text editor I chose to work on. More details about text editor Emacs can be found in: [4, 20, 21].

1.3 Literature Review

Before I started to work on this thesis, I first did some research on text editors, lexical analyzers, parsers, communication algorithms and the materials which relate to this thesis. The main idea of my work is for a computer program, particularly a text editor, to communicate with an external source; for this reason, the main focus is on the communication between computers, the programs interaction, and the techniques of information processing.

Although I did not find as many references as I expected to help me on this thesis, there are still some remarkable references which I would like to introduce. By reading these references, I was able to settle on the research direction. Furthermore, I gained a great deal of knowledge which relates to this thesis. "Programming on an already full brain" [11] introduced an editor tool called `Emacs Menus` which helps programmers to develop a program. This paper explains the tools which `Emacs Menus` provides, and explains their concepts in details.

"Practical applications of a syntax directed program manipulation environment" [6] is another reference which I found very helpful. This paper brings in information about a syntax directed editor and abstract representation of data. After reading this paper, I learned about syntax directed editors and some possible features which can be implemented with extra information generated by a parser.

The paper "UNIX Emacs: a retrospective (lessons for flexible system design)" [1] helped me to gain a better understanding of EMACS and allowed me to comprehend the concept of the EMACS core. After reading this paper, I was clear on what can be added to the text editor EMACS.

Finally, after I finished this thesis, I found a very new article called "An EMACS mode for ALDOR" [13] by Ralf Hemmecke. This paper introduced an ALDOR mode for EMACS. Hemmecke's ALDOR mode provides a token identification feature which

is similar to both of my internal and external lexical analyzers. However, my Aldor mode provides a lot more information to users from my lexical analyzers, the text properties, and a parse tree. In Hemmecke's Aldor mode, the parser is not present, and no parse tree is generated. The indentation section in his paper caught my attention. It used the same indentation logic as I implemented here; which is to find the open and close brackets, and then calculate the proper indentation levels. Nevertheless, the Aldor parser in this thesis generates a parse tree. With all information provided by my parser, users will be able to implement a more powerful and useful indentation function.

In conclusion, even though I did not find a lot of information related to text editor learning; I still gained a great amount of knowledge on other components which are required for me to complete a text editor mode. Although there are some other programming modes for Emacs, none of them were implemented externally and running in parallel with Emacs. Therefore, I concluded that the research field is relatively new. On the other hand, there are already a lot or studies done on the components which are necessary to make the whole experiment work.

1.4 Purpose of my Work

In the current computer world there are many different kinds of text editors for users to write or modify their text files. I have been a programmer for a long time but I rarely find any text editor that really knows programming languages themselves. For example, all programming text editors allow users to modify program files; most of them do indentation and syntax-based colouring for programming languages; some programming text editors allows users to build or debug programs; and a few programming text editors provide tools for a user to add functions. For example, some programming text editors allow a C^{++}programmer to click a menu to execute

a simple command that adds a `while`-loop template with proper indentation. Some match parentheses and provide template filling, which allows users to fill in all the required information (such as termination conditions and the actions this loop presents).

However, according to my knowledge, almost none of the "free" programming text editors teach programmers how to program, or assist programmers to program a project. I believe that most programmers have encountered the situation where they know what they want to do, but were unable to recall the functions they wish to use; or what its syntax is; or whether it even exists. Therefore, a programming text editor that helps programmers to program would be very useful. It is not very complicated to provide these features. One can achieve the learning and assistance features by adding a data base which includes all the required information, and then apply some database retrieval algorithms to complete the search operations, and finally return results to programmers.

However, no matter how complete a data base is; some information may be absent from it. Moreover, for a programming text editor to support a new programming language, it will require a "plug in" with a complete data base. It is not a very efficient way to supply languages support features. For these reasons, it would be better if a computer is able to interact with external sources (i.e. those external to the text editor itself) efficiently.

1.5 Emacs Concepts

In this section I explain some basic components of EMACS and some of the functions I used in this thesis. The programming languages LISP and EMACS-LISP are different. The skeleton of the text editor EMACS is mainly programmed in the programming language EMACS-LISP; therefore, the programming language which

I learned to implement EMACS components is EMACS-LISP. There are a few terms related to EMACS-langLISP that I want to explain first to help readers to understand this thesis.

In EMACS, each file opens in a *buffer* ([10, *pp.* 501–516], [20, Chapter 15]). However, not all buffers are associating with files. A buffer may contains something other than a file. For instance, a shell program can be displayed in a buffer. Additionally, the text displayed in a buffer is not the actual text file on the hard drive; instead, it is a duplicate version. EMACS places buffers in *windows*, and each window opens in a *frame*. EMACS gives users the ability to switch between buffers inside a window; furthermore, a buffer may be displayed in more than one window [10, *pp.* 517–550].

The component which contains one or more windows is called a *frame*. Users can open many windows in a frame. A frame in the EMACS text editor is the same as a window in many other programs. Therefore, it is very important to remember the definitions and the difference between windows and frames in the EMACS text editor.

A buffer associated with a file contains a copy of the text on the hard drive. Thus, any modification in a buffer will not affect the original file until the buffer is saved. The methodology of a programming text editor is to open a program file in a text buffer, and allow a programmer to modify text in the buffer. If a user decides to close the buffer without saving, the original file is untouched. If the buffer is saved, the original file will be replaced and updated to match the buffer [10, *pp.* 551–578]).

EMACS also provides built-in automatic recovery and backup systems. The automatic recovery system generates files which start and end with "`#`"s (pound signs). Moreover, users can customize the back up strategy of the automatic recovery system. Normally, EMACS saves the buffer to a temporary file with the same name as the file plus "`#`" signs at the begin and end (for example, `temp.txt` will be saved as "`#temp.txt#`"). The automatic recovery system can be triggered in many different

ways. For example, after a constant number of characters in a buffer have been modified (by default, 300 keystrokes), or a constant amount of time, EMACS will write the buffer to an automatic recovery file. On the other hand, the backup system generates a file which has the same name as the working file plus a tilde at the end of the file name. For example, "`temp.txt`" will be saved as "`temp.txt~`". Once the backup system saves the file, the file which was created by the automatic recovery system is deleted. Therefore, even when EMACS does not crash, the backup system still saves backup files just in case users want to roll back to previous versions. For these reasons, EMACS is a very powerful and safe environment to create a program, edit reports, and do text modifications [10, *pp.*489–500].

The programming modes in the EMACS text editor give programmers support for particular programming languages. For example, `PASCAL-mode` is one of the programming modes supplied by the EMACS text editor. Programmers who use EMACS to edit their PASCAL program would receive some help from EMACS. Each programming mode provides different support to programmers. Some of the programming modes provide an indentation feature, which re-formats and indents the contents of a program according to scope levels. Such powerful tools require a parser. Some of the programming modes provide compiling features which allow programmers to compile their code directly from the text editor. On the other hand, some of the programming modes only provide a syntax-based colouring feature which helps programmers to identify the types of tokens. Users can write their own mode in EMACS. For more details about programming modes, see the EMACS-LISP Manual [10, *pp.* 405–439].

Each character position in a buffer or a string can have a *text property* list which may contain more than one text property. Each of properties is assigned to character at that particular position in buffer. For this reason, the 'E' character at the beginning of the last sentence may not have the same text properties as the

'E' which position at the beginning of the paragraph. Each property has its own name and value, and it is acceptable for many characters to contain the same text property. Copying text between strings and buffers preserves the properties along with the characters. Similarly, moving characters also moves the text properties. Some examples of text properties are the colour or the font of text. The only way to remove properties from a character is to call a remove property function from EMACS [10, *pp.*640–657] .

Overlays [10, *pp.*766–772] are other ways that properties can be attached to a buffer. An overlay specifies properties that apply to part of a buffer. Each overlay applies to a specified range of a buffer, and contains a property list (a list whose elements are alternating property names and values). An example of an overlay is the text highlighting used for copy and paste in a text editor. Users can select and highlight a region of text, and do some operations upon the text such as copy, replace, delete, or cut. However, applying an overlay does not modify any text properties. In this thesis, the parser appends overlays onto ALDOR code in an EMACS buffer. The parser overlays clearly indicate the beginning and ending of blocks, functions, variables, iteration functions, and other useful information.

In the terminology of operating systems, a *process* is a space in which a program can execute. EMACS runs in a process, and EMACS-LISP programs can invoke other programs in processes of their own. These are called sub-processes or child processes of a EMACS process, which is their parent process. A sub-process of EMACS may be synchronous or asynchronous, depending on how it was created. With a synchronous sub-process, a LISP program waits for the sub-process to terminate before continuing execution. However, an asynchronous sub-process can run in parallel with an EMACS-LISP program. This kind of sub-process is represented within EMACS-LISP by an object which is also called a process [10, *pp.*733–754]. EMACS-LISP programs can use this object to communicate with a sub-process or

Versions

Name	*Version*
ALDOR	Version 1.0.2 and all previous versions
EMACS	Version 21.3.1 (i386-msvc-nt5.1.2600) of 2003-3-27 on buffy
`Flex`	Version 2.5.4
`Bison`	Version 1.875b
Makefile and gcc	Version 3.3.3 (i686-pc-cygwin)
OS (Operating System)	Microsoft Windows XP (Home Edition) Version 2002 – Service Pack 2

Table 1.4: The Version Details of the Programs Used

to control it. I used an asynchronous process to construct the external version of the ALDOR lexical analyzer. I used a synchronous process for my ALDOR parser because a parser needs to look ahead and an asynchronous process may cause problems.

A *process sentinel* [10, *pp.*750–751] is a function that gets executed whenever the associated process changes status for any reason, including signals (whether sent by EMACS or caused by the process's own actions) that terminate, stop, or continue the process. A process sentinel also gets executed if the process exits. A sentinel runs only while EMACS is waiting for terminal input, for time to elapse, or for process output. The advantage of this design is to avoid synchronization errors that could result from running them at random places in the middle of other LISP programs. I used a process sentinel in my external version of the ALDOR lexical analyzer to determine when it had finished.

A process *filter function* [10, *pp.*733–754] is a function that receives standard output from the associated process. If a process has a filter, then all output from that process are passed to the filter. The process buffer is used directly for output from a process only when there is no filter. A filter function can only be called when EMACS is waiting for something, because process output arrives only at such times. Filters and sentinels are very similar in some cases. I used filter functions to restore

the reading point in a buffer to its original place (see Section 3.6).

1.6 Version Details

This ends the discussion of EMACS concepts. In next few chapters, we look at internal ALDOR lexical analyzers, external ALDOR lexical analyzers, an external ALDOR parser, and ALDOR mode for EMACS. Table 1.4 are the versions and builds of the programming language, programs, and applications which I used to establish my work.

Chapter 2

The Internal Lexical Analyzer

Every programming text editor requires a lexical analyzer. The function of lexical analysis is to produce tokens. Tokens which are generated by a lexical analyzer may not all have the same properties. In my research, my lexical analyzer returns token types according to the source text.

A definition of a token is a basic, grammatically indivisible unit of a language such as a keyword, operator, numbers or identifier. For example, the type of the token “if” is a reserved word in ALDOR. Although some programming languages have only a few different kinds of tokens, there are many complicated programming languages that contain more than thirty kinds of tokens. In the following sections, there are some examples and explanation about the ALDOR tokens.

Lexical analyzers, parsers, and text editors are inseparable. To achieve proper syntax-based colouring, a text editor needs the token types which are classified by a lexical analyzer. On the other hand, a parser requires the token types from a lexical analyzer to construct syntax trees for programs. Therefore, a lexical analyzer is one of the main components enabling a programming text editor to work.

There are some programming languages which already are widely used in the computer programming world; for examples: C^{++}, JAVA, BASIC, and PASCAL. For

these programming languages, one can easily find many lexical analyzers to perform token classification. On the other hand, there are some programming languages such as ALDOR that do not have as many lexical analyzers available. Furthermore, there is not yet a text editor designed to support syntax-based colouring and token identification for the ALDOR programming language. Since ALDOR is not yet widely used, resource and tools which support the language are very limited.

2.1 Overview of Building the Internal Lexical Analyzer

The first step to produce a lexical analyzer for ALDOR is to understand properties of a lexical analyzer. There are many lexical analysis algorithms available, which makes it possible to do comparisons between them. In my opinion, the most important properties of a lexical analyzer are correctness and efficiency. A lexical analyzer should be able to classify tokens in a buffer within a reasonable time. Moreover, high speed classification should not affect the correctness of the results. Additionally, I demand my lexical analyzer to be understandable by anyone who wishes to implement a lexical analyzer based on mine. A lexical analyzer should provide maximum support to its users (programmers and text editors) with minimum learning requirements. EMACS already supports many different programming languages and presents many programming language modes. Hence, one can study the code which has been implemented within the EMACS text editor to try to understand how to implement an internal lexical analyzer.

EMACS-LISP provides many built-in functions which help programmers write efficient programs for EMACS. Since it provides so many built-in functions, it is up to programmers to decide how to use the tools they have available. Sometimes, it

is very difficult to decide when and how to use them.

There are many different kinds of lexical analyzer available, and most of them take different approaches and use different algorithms. Each approach has its own advantages and disadvantages. Thus, understanding the algorithms which implement lexical analyzers is a very important task. Finally, after discussion with my supervisor, I decided on the approach of identifying the ALDOR tokens by regular expression.

After I completed the lexical analyzers, tests on the time and space usage were performed upon them to ensure the correctness and efficiency (see [19] about programming languages and their memory usage).

2.2 Method

My internal lexical analyzer is implemented in a file called `aldor-internal.el`. The lexical analyzer provides functions to traverse a source buffer and add text properties to it based on tokenization information. My first version of an internal lexical analyzer reads from the source buffer and then writes the tokenized result into an intermediate buffer. Finally, it processes the information in the intermediate buffer and puts results back into the source buffer including colour highlighting and text properties.

Nevertheless, I was not fully satisfied with the results from the first approach due to its memory space usage. For this reason, I eventually came up with an algorithm which works better and updates the source file "on the fly". This approach improves my internal lexical analyzer speed by nearly a factor of two. This version of the lexical analyzer fetches a token, finds the type, then updates colour and text properties of the token directly. For this reason, there is no intermediate buffer involved. An intermediate buffer not only wastes resources (memory and space),

but also requires a lot of time to process.

Since this algorithm is faster, I used it in my lexical analyzer. The combination of all improvements led to my final version of a lexical analyzer recognition algorithm. I first collected all the ALDOR tokens and then grouped them into three sub-types which I will discuss later in this section. Finally, I developed optimized regular expressions for each token and wrapped the tokens with EMACS built-in functions to improve the speed of token recognition.

There are two scanning functions in my program. One function called `aldor-scan`, requests users to provide names of files that need to be tokenized. The other function, called `aldor-scan-buffer`, scans the currently selected EMACS buffer, runs a lexical analyzer, and applies the syntax-based colouring and text properties to the text buffer. Other components which make my lexical analyzer work are functions called `aldor-find-token` and `aldor-colour-syntax`. The `aldor-find-token` function calls the defined regular expressions functions and tries to identify a type for a token. Once a token type has been found, `aldor-find-token` calls `aldor-colour-syntax` and passes on the following information: the beginning and end position of the token; the type of the token; and the source buffer reference. The `aldor-colour-syntax` function uses the information it has to insert desired colours and apply appropriate text properties to the token string. One thing to notice, a lexical analyzer changes how the information in the buffer is displayed, not the contents of the file. As a result, the source file will not be modified unless the user decides to save the buffer and overwrite the actual file. However, results of a lexical analyzer are not be saved. In other words, EMACS will not save any of the text properties when it saves a buffer.

I use regular expressions to perform token recognition and handling. I also use an explicitly loaded function called `rx` from the EMACS package. This function will translate an S-expression (a balanced-parentheses expression) to a regular expression

```
(setq aldor-reserved-word
        (rx (and
             word-start
             (or
             "add"        "always"    "and"        "assert"    "break"
             [... see Appendix A for a complete list ...]
             "return"     "rule"      "select"     "then"      "throw"
             "to"         "try"       "where"      "while"     "with"
             "yield")
             word-end)))
```

Figure 2.1: A Regular Expression to Match Aldor Reserved Words

string. I separated the tokens into many different categories. Then I wrote one regular expression for each token.

The Aldor reserved words are the simplest category to implement because reserved words are defined as fixed strings. The Emacs-LISP code in Figure 2.1 defines the regular expression for the Aldor reserved word token strings. Function `setq` assigns the value in the second argument to the first argument. In my case, the first argument is an Aldor reserved word type; and the second argument is the value which is returned by the `rx` function. The rule for reserved words is quite simple as it requires three elements to return a reserved word: a word-start token; a keyword declared within an "`or`" block; and a word-end token. Both `word-start` and `word-end` tokens are predefined by Emacs (for details about these functions, please refer to the Emacs-LISP manual [10, *pp.* 687–710]).

There are many other types of tokens which are declared in a similar way to reserved word tokens. The following token types fall in this category: `Reserved` keywords, `Definable` keywords, `Class` keywords, `Import Function` keywords, `Future` operator, `Reserved` operators, and `Definable` operators.

The second kind of tokens are identifiers, integer literals, floating-point literals and any tokens which cannot be defined by a fixed list of strings. A regular expres-

```
(setq aldor-string
          (rx (and "\""
              (0+ (or (and "_" (1+ space))
                     (and "_" (not space))
                     (not (any "_\n\""))))
              "\"" )))
```

Figure 2.2: A Regular Expression to Match ALDOR String Tokens

sion is the perfect technique to handle these tokens. Among all token types without a fixed form, the `String` token is the simplest type to handle. Figure 2.2 contains EMACS-LISP codes which defines the regular expression for such a token.

An ALDOR string is defined as a sequence of symbols which starts and ends with double quotes. Between these two double quotes any sequences of character, word, sentence, and space are allowed but double quotes and new line characters are unacceptable. Moreover, the "_" underscore symbol can be displayed in an ALDOR string as long as the following rules apply: 1) an "_" is followed by one or more whitespace characters; 2) an "_" is followed by exactly one non-white space character (possibly a double-quote), or 3) an "_" is followed by any character other than an underscore, a double quote, or a newline character. Hence, it is legal for a string token to contain one or more underscore "_" symbols.

Among all token types without a fixed form, the floating-point literal token is the most complicated to define. Since ALDOR supports floating-point literals in many forms, the regular expression for such a type is not simple. Figure 2.3 includes a collection of intermediate regular expression which are required not only for floating-point literal tokens, but also other numeric literal tokens.

ALDOR floating-point type tokens can only be represented by very complicated regular expression in EMACS-LISP code. After a few attempts and discussion with my supervisor, I finally designed the EMACS-LISP code which define ALDOR `aldor-float` type tokens (see Figure 2.4).

```
;; Some basic and intermediate regular expressions that are
;; used in other regular expressions more than once.

esc-digit '(and (zero-or-one "_") digit)
long-digit '(any "0-9A-Za-z")   ; CAN be replace by "letter"....
esc-long-digit `(and (zero-or-one "_") ,long-digit)
radix `(and digit (0+ ,esc-digit) "r")
expon `(and (any "eE")
            (zero-or-one (any "+-"))
            (1+ ,esc-digit))
```

Figure 2.3: A Regular Expression to Match ALDOR Intermediate Tokens

I would also like to briefly explain an EMACS-LISP symbol ` (back quote) and `rx-to-string`. The function `rx-to-string` parses and produces code for regular expression which are written in S-expression regular expression form. On the other hand, the back quote function translates everything in its matched parentheses to a list without evaluating the contents. For this reason, the expression which is defined by the back quote function can be substituted into other regular expressions by adding a comma in front of it. For more details about EMACS-LISP functions and symbols, please refer to the Regular Expression sections in the EMACS-LISP Manual [10, *pp*.687–710]. Additionally, for details about EMACS-LISP declarations for integer literals and other token types without fixed forms, please refer to the documentation and comments in the `aldor-internal.el` file.

The third kind of declaration category contains ALDOR `Pre-Document`, `Post-Document`, and `Comment` token types. All three kinds of token types start with some repeated special symbols and end with a new line character. The `Pre-Document` token type is defined by any string after three plus (`+++`) symbols (see Figure 2.5).

The new operators which I now introduce are `repeat` and `group`. A `repeat` operator takes two arguments, the number of repetitions of the symbol and the repeating symbol. A `group` operator groups the repeating symbols together. In

```
;  The following regular expression defined the floating-point
;  type token:
;
;         {digit}*"."{esc_digit}+{expon}?|
;         {digit}+"."{esc_digit}*{expon}?|
;         {digit}+{expon}|
;         {radix}{esc_long_digit}*"."{esc_long_digit}+{expon}?|
;         {radix}{esc_long_digit}+"."{esc_long_digit}*{expon}?|
;         {radix}{esc_long_digit}+{expon}
;
;  The following is the EMACS-LISP code for "float"

(setq aldor-float
      (rx-to-string
            `(or
                (and (* digit) "."
                     (+ ,esc-digit) (zero-or-one ,expon))
                (and digit (* ,esc-digit) "." (* ,esc-digit)
                     (zero-or-one ,expon))
                (and digit (* ,esc-digit) ,expon)
                (and ,radix
                     (or
                       (and (* ,esc-long-digit)
                             "."
                             (+ ,esc-long-digit)
                             (zero-or-more ,expon))
                       (and (+ ,esc-long-digit)
                             "."
                             (* ,esc-long-digit)
                             (zero-or-more ,expon))
                       (and (+ ,esc-long-digit) ,expon))))))
```

Figure 2.4: A Regular Expression to Match ALDOR Floating Point Literals Tokens

```
(setq aldor-pre-document
             (rx (and (group (repeat 3 "+"))
                      (1+ not-newline))))
```

Figure 2.5: A Regular Expression to Match ALDOR Pre-document Tokens

the case of the ALDOR `Pre-Document` token, three plus signs (`+++`) are grouped together. `Post-Document` and `Comment` tokens are implemented in a similar way. The difference is that `Post-Document` tokens starts with two plus signs instead of three; while `Comment` tokens starts with two minus signs (`--`). The above is summarized in Table 1.2.

2.3 Efficiency

This section discusses the efficiency of the update "on the fly" approach. As computer speed improves, modern computers are able to process the entire source buffer before a programmer implements a simple function in a buffer. Hence, the timing efficiency becomes a less serious threat for overall efficiency. For a small ALDOR program buffer, the space and time efficiencies are almost inconspicuous. Once I am dealing with a buffer which contains a huge ALDOR program, then space usage and time efficiency become very important.

For the updating "on the fly" approach, extra memory usage is close to zero. It is very safe to declare that the space usage per line will always stay constant; and the size of a buffer will not affect the space efficiency at all. The next issue is the time usage. In this version of the internal ALDOR lexical analyzer, each token is updated "on the fly". One of the greatest advantages of this approach is buffers get updated almost instantly. There is no time wasted in the transformation from an original buffer to a reorganized buffer. An internal ALDOR lexical analyzer which uses this approach updates two thousands lines of ALDOR codes within two seconds on my machine (Windows XP base machine, an AMD 2400+ CPU, 512 MB of RAM, and no other applications running besides EMACS).

In conclusion, the outcome for the time and space efficiency of my internal ALDOR lexical analyzer is acceptable. In Chapter 3, I include a data chart which

contains all the timing for this approach.

2.4 Correctness

A lexical analyzer should at least correctly identify tokens in a buffer. I ran my internal lexical analyzer against many different ALDOR files. All ALDOR tokens turned out to be classified correctly. In other words, all tokens were identified as expected.

In summary, I have completed a lexical analyzer which provides good performance with acceptable resource usage. My requirements for the program were that it must not only serve the purpose of what it was designed for, but also that it uses all resources correctly. The final version of my internal lexical analyzer met all the requirements which I had.

2.5 Problems Encountered and their Solutions

In this section, I discuss problems which I encountered during the development of my internal lexical analyzer. There are two major problems which I would like to explain. The first problem is the priority problem and the second one is the token type identification problem.

During the development of the internal lexical analyzers for ALDOR, I noticed the priority problem. The priority problem is due to bad design of token priority. For example, I would not wish to see any reserved word classified as an identifier. Since my program identifies tokens by regular expression, sometimes there is a tie situation. In this case, my program uses the first rule which was declared. In the example, if the identifier token type was declared before the reserved words token type; the token would be classified as an identifier. For this reason, an identifier

Type	Priority
Reserved	B
Definable Keyword	B
Class	B
Import Function	B
Pre-Document	A
Post-Document	A
Comment	A
String	A
Identifier	E
Float	C
Integer	D
Definable Operator	B
Reserved Operator	B
Future Operator	B

Figure 2.6: Aldor Token Precedence

token type should have the lowest priority among all token types.

Although my lexical analyzer tries to match the longest possible sequence in a buffer, I still need to write multiple separate regular expressions to recognize the various kinds of tokens, and prioritize those regular expressions to ensure that they are tried in an order that guarantees matching the longest sequence. For this reason, I developed a priority chart for the Aldor programming language (See Figure 2.6).

The most critical problem which I encountered is the "0" and "1" symbol problem (token identification problem). According to Aldor syntax, 0 and 1 should be treated as identifiers, but their type may change depending on the surrounding context and on the effect of explicit or implicit `import` statements. For this reason, both of them can possibly be an identifier, an integer or a floating-point number, a function name, a procedure, or a string (and any other type which is valid in Aldor). As a result, classification of the type of these two symbols becomes very complicated. To determine the exact token type for 0 and 1 would required a parser. In conclusion, a lexical analyzer has absolutely no way to find the type of 0 and 1;

finding their types is as difficult as parsing the program. For this reason, in both internal and external lexical analyzers, I have treated these two tokens as identifiers.

There were many problems and bugs which I encountered while I developed my internal version of the lexical analyzer for ALDOR. Most of them were resolved by doing more research, re-designing the program, and asking other people. This section discussed some of the common problems which everyone may encounter during design time. Furthermore, this section may help other programmers who wish to write lexical analyzers make decisions on when to use what approach. This section should help others not to make the same mistakes which I made while I was developing the program.

2.6 Conclusion and Summary

In conclusion, the internal version of the lexical analyzer which I designed for the programming language ALDOR worked very well. It is working faster than my original expectation and it is also very resource efficient. Although there are some minor details I would like to change, the outcomes should satisfy most people overall. Definitely, this lexical analyzer has set the foundation for my future research and `aldor-mode` components.

Additionally, implementing a good lexical analyzer is a very important step before implementing parsers. For a programmer who wishes to build parsers for any of the programming languages, a good lexical analyzer will surely help them in their later work. Another important thing which I learned from this exercise is to think generically. I tried to make my lexical analyzer very flexible, understandable, and easy to modify. As a result, anyone who studies this thesis should be able to implement an internal lexical analyzer with syntax-based colouring and token identification features for any programming language. Like lexical analyzers for

other programming language modes, my internal lexical analyzer is implemented in Emacs-LISP and uses Emacs regular expressions. Although it is not much different from pre-existing lexical analyzers, it may be easier for others to read.

This ends the discussion of my internal lexical analyzer. In the following chapter we look at an external lexical analyzer that communicates with Emacs via pipes. In Chapter 6 we compare the lexical analyzer presented in this chapter with the one presented in the next chapter.

Chapter 3

The External Lexical Analyzer

In the last chapter, I described internal lexical analyzers for the programming language ALDOR. In this chapter, I discuss external lexical analyzers. I believe that independent programs are more generic and more flexible. I compare both internal and external lexical analyzers through comparisons of: time, memory, efficiency, resource usage, and usefulness.

3.1 Overview of Building the External Lexical Analyzer

As mentioned, I worked on both external and internal lexical analyzers simultaneously. For this reason, I learned a lot from both versions during development. One of the advantages to this strategy is that if I could apply the same logic to both lexical analyzers, then I would be able to compare the differences in results. From all of these results, I eventually found the best method for the internal and external version of the program.

The primary challenge with an external lexical analyzer is program-to-program communication. This research also compares the internal and external lexical ana-

lyzers for efficiency and ease of implementation. For this reason, establishing correct and efficient communication is one of the most important components for this portion of the research.

For two people to talk to each other, they must first agree to use the same language to talk. Otherwise, they will not be able to understand each other. Even if they do not share the same language, they must have some common form of communication in order to be able to exchange words, thoughts, or ideas with each other. For example, Japanese and Chinese share some of the same sets of characters. Hence, it is possible for people from these two nations to communicate with each other. Some other ways of communicating are sign language, drawings, or simply finding a translator.

Similarly, for two or more programs to communicate with each other, there must be some way for them to talk or communicate their needs. In the case of my external lexical analyzer, there are pipes (intermediate media) which are created by EMACS in order to make communication possible. Once an intermediate medium has been established, the remaining issue is for those programs to speak the same language. Just like with humans, the only way for computers or programs to interact with each other, is to exchange information in the same language.

The language that my external lexical analyzer uses to communicate with EMACS consists of lists of LISP expressions that contain three arguments. The arguments of the expressions are a beginning point, an end point, and the type of an identified token. To fetch a list of three-tuples from an intermediate buffer, I implemented the functions `aldor-read-from-list` and `aldor-get-1-value`. The latter function returns one three-tuple LISP expression to the `aldor-read-from-list` function. The information is then used to complete all remaining tasks of the lexical analyzer. Since EMACS is implemented in EMACS-LISP, I believe it would more be efficient to process information in a list of partial EMACS-LISP code.

Unfortunately, sharing the same language is not enough to establish communication. At a later stage of my research, I realized the timing is also very important. For example, suppose there are two people who want to meet in a park. For one reason or another, one person is late. As a result, the other person may assume that his friend will not be coming, forgot or such, and decides that he should go home. Hence, the meeting of these two people is cancelled. Similarly, I also ran into the same problem with my analyzer: an EMACS-LISP program process terminated and assumed scanning was finished while an external lexical analyzer was still scanning the source file. Since the program is running outside of EMACS, it needs time to scan the source buffer and to generate the information which can be read by the EMACS program, and only then could it give all the information to the EMACS program. The `process` function of EMACS sends a request to execute an external lexical analyzer and waits for results to be returned. How long should EMACS wait for results? How long will it take for a lexical analyzer to return the results? What is the best and most efficient way to handle this communication and waiting problem? All of the above questions presented challenges in completing this external lexical analyzer. Thus, the research I did for this version of the lexical analyzer is heavily concentrated on these problems.

One thing I realized was that all methods which I applied in an external version also applied to the internal version, whereas the methods which I applied to the internal version often could not be applied to the external version. The reason for this is that there are more limitations on an external version of an analyzer. The challenge of establishing a tokenizing method for an external lexical analyzer is not the implementation; it is primarily the problems of communication and timing.

3.2 Method

The key for building internal lexical analyzers for EMACS is to use resources efficiently. Internal versions of programming language lexical analyzers are heavily focused on the combination of function efficiency and optimized resource usage. Since all information is provided inside EMACS itself; I was able to focus on the optimization of the lexical analyzers' performance. Development of an external analyzer for EMACS is a completely different experience. An external analyzer for a particular programming language fetches the contents of an EMACS buffer and processes the information it gets. When an external analyzer finishes its tokenizing process, it sends results back to EMACS in a format which can be understood and processed by EMACS. Once EMACS has received this information, it is able to complete the scanning job, ultimately returning or displaying the appropriate result in its buffer. Obviously, the first challenge which I came across is transferring information.

When I did my research, I found a few lexical analyzer programs such as `Python` and `Flex`. For these programs, programmers provide tokenizing rules, and these programs will generate results as set of tokens. Among all of these programs, I chose a tokenizer language called `Flex` (see [16], [17] for more details). For complete information on `Flex`, please refer to the official `Flex` website, or consult the `man` pages provided by UNIX. Here are a few reasons for using `Flex` to implement my lexical analyzer. Firstly, it is very well documented. For this reason, the logic and techniques used by `Flex` to do the tokenizing process were understandable. There are other programs similar to `Flex` which do a comparable job; but, do not have documentation as good as that of `Flex`. Secondly, I tested a variety of these programs, and `Flex` was one of the few programs which was able to get the job done in a reasonable amount of time with accurate results. Since I tried to make the entire program work efficiently — its components being the external lexical analyzer for

Aldor and Emacs — the running time of my external analyzer is very important. Thirdly, the implementation required for `Flex` is fairly simple. It is very close to `C` in programming style, a language I am proficient in. This made it very easy to learn and then write a good rule tokenizing program. For these reasons, I decided to use `Flex` as my external end of the pipe.

One of drawbacks which I soon become conscious of is the lack of flexibility of a lexical analyzer generated by `Flex`. In general, a generic program such as `Flex` is less powerful than a program created to meet a specific need, such as for a specific language. Clearly, when I write a program, I can customize results that I want, along with all the information that I need. Moreover, I am able to modify and shape a program to suit my needs. Due to the limitation of `Flex`, I spent lots of time on researching the syntax, logic, and implementation of `Flex`. Instead of letting `Flex` generates results in the format which I wish for, I had to make Emacs be able to understand the information which was created by `Flex`. At that point in time, my main challenge was to attain communication between Emacs and `Flex`. Somehow, I had to develop a middle stage such that both ends of the pipe would be able to talk to, and understand each other without any mistranslations or misunderstandings.

The implementation of the Aldor lexical analyzer using `Flex` did not take too much time. Instead, it was the process of getting tokenizing rules that proved to be more difficult. Although I had already completed rules for the internal version of the lexical analyzer, the trail which led to the completion of an external lexical analyzer was still cloudy. `Flex` has its own rules for regular expressions and, moreover, the implementations for token recognition rules are very different between Emacs, an internal version; and `Flex`, the external version. Basically, Emacs and `Flex` have different representations of regular expressions. All token rules needed some minor adjustment due to this difference. As a result, I spent many hours testing the correctness of `Flex` defined tokens, and correcting the errors.

The first set of rules which I put into operation were some basic intermediate regular expression (Figure 3.1). Additionally, `Flex` allows me to define the actions to take when a token is found. Figure 3.2 contains some examples of how to define actions for `Flex` when a token has been found or classified.

By this example, one can easily observe that actions have been defined as a token identification followed by a function to handle the token. These actions are written in the format used by `C`. Earlier I covered all the different token cases and actions which are taken when some token has been classified. At the beginning of my example, when a "`ws`" (white space) symbol has been identified, my program then increases variable `init_p` (defined by me) by one. If a "`nl`" (new line) token has been classified, the program should takes no action with regards to the `init_p` variable. If a reserved word or constant symbol string token is found, the program then returns results to the program which called this `Flex` program. Fortunately, `Flex` gives programmers flexibility to not only return results or complete a single action, programmers can actually run a series of actions inside an action block. For this reason, I can call pre-defined functions of `C`, such as `strncpy` or user defined functions, like `print_lisp`. From this, a programmer is able to generate results in a preferred format by simply writing format functions. The last piece of `Flex` is the driver program which is once again implemented in C.

After a few different attempts and improvements I implemented my final version with a new function, `print_lisp`, which prints the results to a file with a format that EMACS-LISP is able to understand. In the final version, I cleaned up most of the extra statements and optimized the token classification rules for ALDOR. Therefore, the running time for my external lexical analyzer is very reasonable. The information which is provided by the `print_lisp` function is: the starting position, ending position, and type of a token. Notice that the function `print_lisp` returns types of tokens instead of the actual token strings. The EMACS text editor does not

```
esc  _

alpha                   [A-Za-z%]
one_digit               [2-9]
digit                   [0-9]
esc_digit               {digit}|{esc}
long_digit              {digit}|[A-Z]
esc_long_digit          {long_digit}|{esc}
radix                   {digit}{esc_digit}*"r"

int         {one_digit}|{digit}{esc_digit}+|{radix}{esc_long_digit}+
expon       [eE][+-]?{esc_digit}+
float       {digit}*"."{esc_digit}+{expon}?|
            {digit}+"."{esc_digit}*{expon}?|
            {digit}+{expon}|
            {radix}{esc_long_digit}*"."{esc_long_digit}+{expon}?|
            {radix}{esc_long_digit}+"."{esc_long_digit}*{expon}?|
            {radix}{esc_long_digit}+{expon}
dot  "."

future_word             "delay"|"fix"|"is"|"isnt"|"let"|"rule"
future_op               [\[{(]"|"|"|"[\]})]|[‘&]|"||"
definable_op_word       "by"|"case"|"mod"|"quo"|"rem"
definable_op            [\-\\#*+ or <=>@~\^]|"<<"|">>"|"<="|">="|"<-"|
                        ">"|"\\ or "|" or \\"|"**"|".."|"~="|"^="

id  "0"|"1"|{alpha}({alpha}|{digit}|[!?|])*
quote  ["‘"]
string_char  [^_‘‘"\n]|esc[^\n]
ws  \t|" "
nl  \n["\015"]

pre_doc   "+++".*
post_doc  "++".*
comment   "--".*
string  \"[^\"]*\"
```

Figure 3.1: Definitions for Some ALDOR Tokenizer Components

```
{ws}                 ++init_p;
{nl}

add         { print_lisp( init_p, yyleng, "aldor_reserved", "ADD" );}
always      { print_lisp( init_p, yyleng, "aldor_reserved", "ALWAYS" );}
and         { print_lisp( init_p, yyleng, "aldor_reserved", "AND" );}
assert      { print_lisp( init_p, yyleng, "aldor_reserved", "ASSERT" );}
break       { print_lisp( init_p, yyleng, "aldor_reserved", "BREAK" );}
but         { print_lisp( init_p, yyleng, "aldor_reserved", "BUT" );}

(lines omitted)

stdout|newline|space|include|rep|per|Per \{
  strncpy(tokenString,yytext,20);
  print_lisp( init_p, yyleng, "aldor_import_function", yytext);
 \}

   /* pre_doc */
{pre_doc} \{
  strncpy(tokenString,yytext,20);
  print_lisp( init_p, yyleng, "aldor_pre_doc", yytext);
\}

(lines omitted)

"}"        { print_lisp( init_p, yyleng, "aldor_res_op", "RBRACE" );}
"=="       { print_lisp( init_p, yyleng, "aldor_res_op", "VERY_EQUAL" );}
"|"        { print_lisp( init_p, yyleng, "aldor_res_op", "BAR" );}

  /* Future ops */
"[|"       { print_lisp( init_p, yyleng, "aldor_future_op", "LLBRACK" );}
"{|"       { print_lisp( init_p, yyleng, "aldor_future_op", "LLBRACE" );}
"(|"       { print_lisp( init_p, yyleng, "aldor_future_op", "LLPAREN" );}
"|]"       { print_lisp( init_p, yyleng, "aldor_future_op", "RRBRACK" );}
```

Figure 3.2: Actions to Take When Aldor Finds a Token

need to know the actual token; it is more helpful to know the type of the token.

Once I finished my study on functionality provided by `Flex`, I implemented a function called `aldor-scan` in EMACS, which takes one argument, the name (or path) of a target file, sets all the required variables, then starts the communication with my external lexical analyzer and generates the result. I then used an intermediate buffer, which allowed the two programs which I created to communicate with each other. In other words, an intermediate buffer acts as the bridge connecting both sides of the programs. This intermediate buffer called "`middle.txt`" (Note: the name of this buffer is customizable, and can be any name programmers wish it to be). Once I secure the connection, I also need a variable which will store and call the procedure. In my case, the procedure is the external lexical analyzer for ALDOR. Finally, I have to let my `aldor-scan` function know where to output results. For a lexical analyzer, putting results back into the source buffer is the most direct and efficient way to notify users what changes the lexical analyzer made. For this reason, the result buffer is same as the source buffer.

One of the mistakes I made in an earlier stage was missing a file existence check. Sometimes, users may enter invalid paths or file names; as a result, the whole program will crash due to being unable to locate the source file and update the information back to the buffer. Fortunately, EMACS has automatic file creation abilities. If a file does not exist, EMACS will create a new empty buffer named as the user requested. To fix the file non-existence error, I added error checking. Since EMACS already has a built-in function to handle error checking, it was not too hard for me to implement. All it took was some thinking to consider all the possible problems.

Once I initialize all the required components, I then call my external lexical analyzer from the `aldor-scan` function. I also have another function called `aldor-read-from-list` to read results from the intermediate buffer. Coming to a comple-

tion, the `aldor-read-from-list` procedure calls the `aldor-colour-syntax` procedure to add additional properties to each token. These properties are associated with buffer text itself. Therefore, if text is deleted, then all the properties are also deleted. Additionally, if text gets moved to another location, properties will also move to the other location. One of the biggest advantages of associating the properties with text is that it gives us additional information. In my case, users are not only able to identify the types of tokens by colour highlighting, but they are also able to see the types of tokens displayed in the "`echo`" area of the EMACS text editor as they scroll their cursor over tokens. Furthermore, programmers can always add in more properties to token text to help them implement more advanced features.

After deciding on the algorithm and implementing my external lexical analyzer, I then did a fair amount of tune up and problem solving on my external version of the ALDOR lexical analyzer.

3.3 Communication or Pipes

As mentioned in an earlier chapter, the key for program communication is to unify language and synchronize the timing. If programs are able to agree on one protocol and use the same format then mutual understanding between programs is really not that problematic. In my research, I used a lists of three-argument EMACS-LISP expressions as the intermediate language between an external lexical analyzer and the EMACS text editor. My external lexical analyzer works as follows: the EMACS text editor first sends its request to an external program and expects to learn something from it. Secondly, the external program generates results in an intermediate language and feeds those results to an intermediate buffer which was initially created by the EMACS text editor. Finally, the EMACS text editor takes the results from the intermediate buffer and learns from the results. From this algorithm

we can see that the most efficient implementation will have the external program generate results in EMACS-LISP code (or some code which can be evaluated by EMACS) and have EMACS compile code from the intermediate buffer. To summarize, the intermediate language is code in EMACS-LISP. Therefore, EMACS can simply evaluate results one-by-one from an intermediate buffer. Please refer to Appendix E for the UML diagram of the lexical analyzer.

On the other hand, synchronizing the timing is also very important. In the computer world, as long as communication is involved in a given application, then timing is an important issue. For example: in any network which requires secure communication, an uncoordinated communication, or timing problem may result in an error, duplicated data, or transmission problems. In parallel computing and distributed systems, timing is also a very important component.

In this thesis, I have a waiting time of 0.1 seconds for my EMACS scanning program. Therefore, results are updated within a reasonable time. The program basically looks into an intermediate buffer for token information. If the intermediate buffer is empty, then it waits for one tenth of a second (0.1s), and attempts to read from the intermediate buffer again. Therefore, eventually, it will be able to identify entire tokens in a source file and associate all desired properties to tokens.

3.4 Efficiency

I performed my tests on a Windows XP based machine, with an AMD 2400+ CPU and 512 MB of RAM. I used an internal EMACS elapsed time function to time the processing time of my programs. During the timing process, since I wanted to get the best and most accurate result. I did not run any other applications other than EMACS. I performed the same test three times, and averaged the results. However, the timing for all three times is very close. For this reason, I did not perform any

more tests of timing. I only performed my tests on one machine. In my concern, the main focus was to do the timing comparison between the internal and external lexical analyzers. As a result, it is pointless to perform tests with a faster or slower machine. Overall, I will get the same comparison result on their timing ratio.

I timed the processing time for both internal and external lexical analyzers. Figure 3.3 is the data and time table for internal lexical analyzer versus external lexical analyzer. From the figure, it is easy to observe that an internal lexical analyzer runs faster than an external lexical analyzer.

3.5 Correctness

There might be some problem which will affect the correctness of the lexical analyzer. For instance, the longest sequence matching rule should always be applied to the lexical analyzer. If I do not follows the longest sequence matching rule, a token scanned later may overwrite a previously classified token. The basic idea is correct; an algorithm proposed is an over-write algorithm to avoid some problematic situations. For instance, in a `Pre-Document` token, there should not be any other token type in it. Since "`for`" is one of the reserved words for ALDOR, it would be wrong if such text string gets identified as a reserved word inside a `Pre-Document` token. On the other hand, the overwrite algorithm also raises a problem. For example, suppose we have following string:

```
+++ I love ++, and -- likes to program..
```

Assume the token scanning order is `Pre-Document`, `Post-Document`, and then `Comment`. As a result, I get a `Comment` token insides a `Post-Document` token; and a troublesome `Post-Document` token inside a `Pre-Document` token. Therefore, the scanning process is not correct.

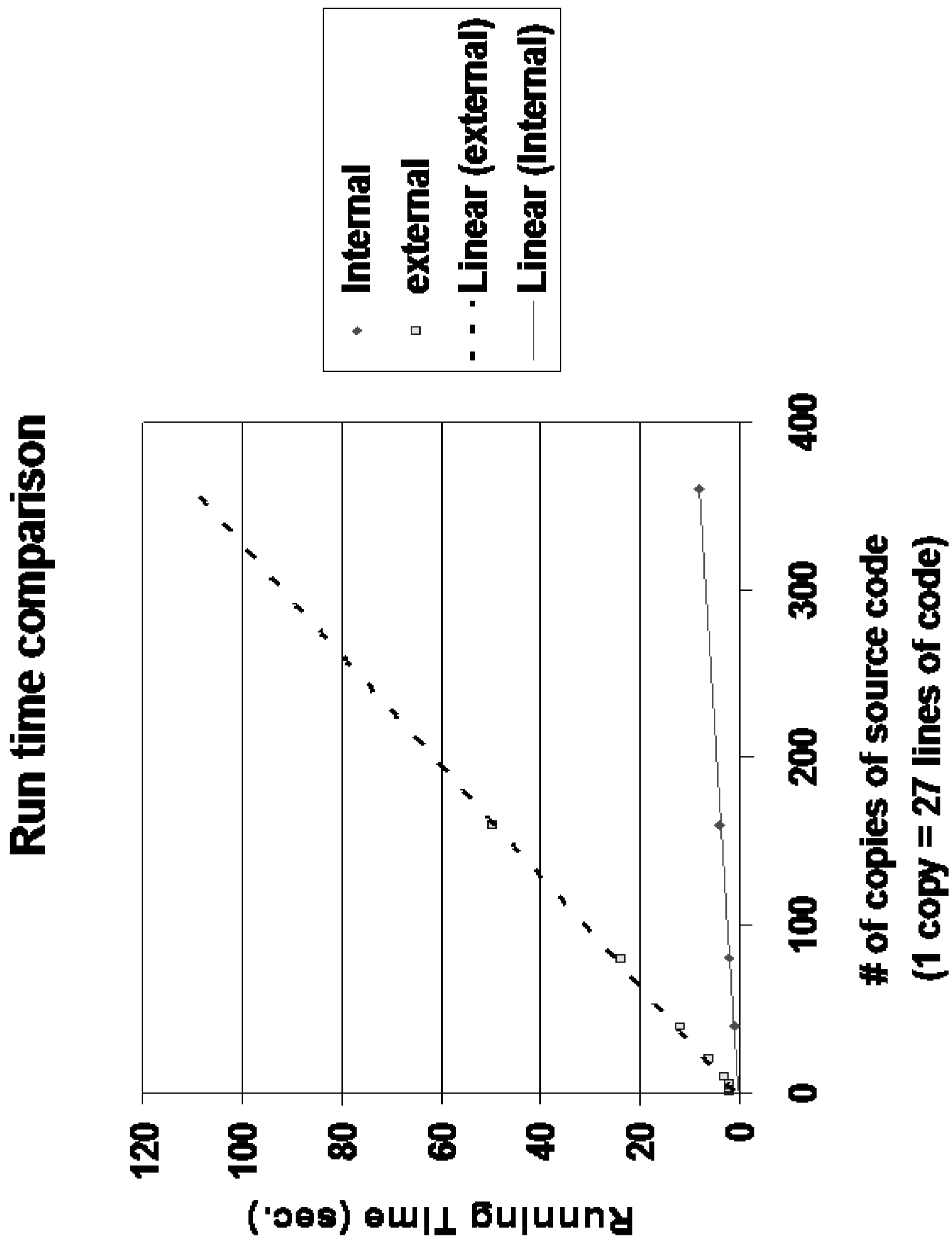

Figure 3.3: Timings of the Internal and External Lexical Analyzers

My scanning algorithm implemented in `Flex` seems to be correct for the cases it is designed to handle. This version of the lexical analyzer tries to match up a token by the longest match rule. Hence, there will be no longest sequence match problem for my lexical analyzer. For example, it handles the above example string with the longest match token type it can classify; therefore, such a string will be classify as

a `Pre-Document` token — which is what I expect.

In conclusion, correctness is more important than time efficiency. Generally, any lexical analyzers should sacrifice its efficiency for a correct result. Overall, it is more important for a program to do its job right and serve its purpose.

3.6 Problems Encountered and their Solutions

Similar to most of my experiments, the very first version of my external lexical analyzer failed dreadfully. It was a total disaster as the outcome not only crashed my computer, but also modified my entire program file containing all of the lexical analyzer functions. I quickly noticed that I had not set my path correctly. As a result, instead of modifying a source buffer, the program ends up modified itself. Luckily, I always keep a back-up copy of all of my work. Thus, I did not use too much time to recover my lost work.

Eventually, I realized the problems I encountered with the buffer were not the only problems. I also encountered difficulties with my intermediate buffer. An intermediate buffer should be cleared every time before its use. Otherwise, an intermediate buffer will have a lot of extra useless information. Worst of all, some of the extra information may lead to a wrong result or even crash the program. Mismatched information may also lead EMACS to update a source buffer with information that is meant for another buffer. Furthermore, if the other buffer is longer than the source buffer, then EMACS will try to put those erroneous properties beyond the end of the source buffer which then causes termination of the program. EMACS has to do all of its work according to information which is provided by an external lexical analyzer. Therefore, if the token information is not correct, then EMACS will not be able to do its job accurately.

The solution which I used was to delete the intermediate buffer as soon as the

entire source buffer has been scanned and all properties have been associated. In my program, there is no file actually being created; as per the discussion in the introduction, a buffer is different from a file. I used a buffer which is generated by EMACS to store all information which is then returned by an external lexical analyzer. At the end of my program, I removed the buffer. Hence, the cleanliness of the intermediate buffer has been assured.

The problems which I encountered above were very simple compared to the timing and communication problems which I met later. The major challenge when it comes to the timing and communication problems is to nail down actual problems. Any mistake could result in program termination, a blank result, or a faulty result.

Of all the work I did, I spent the majority of my time debugging problems, learning of potential problems, and trying to find a solution to fix those problems. Initially, when I ran my program and nothing happened, there might be many possible causes. Moreover, there were many places which may have problems. For this reason, it was very hard to find out where the problem is and what caused it in the first place. I used a debugger in an attempt to find problems in a systematic process; since there are no other reasonable ways to nail down the problem. Although such a process takes a long time to make any progress, my hard work eventually paid off. I finally found the first bug — a communication bug. Basically, my external lexical analyzer did not send results to an intermediate buffer. As a result, the intermediate buffer was empty, and EMACS was unable to do any work on the source file. The solution which I found is to use another function called `start-process`. I also created variables to hold all of the required information. Please refer to Appendix E for the UML diagram for the external lexical analyzer.

Once I figured out that problem, and changed my approach, the intermediate buffer writing problem was solved. I ensured that all of the results got written to an intermediate buffer in the format which I requested, a format which could

be understood by EMACS. However, even though results had been written to an intermediate buffer, my program still did not work. I found out that there was a series of problems in my program — all of which led to failures of my program. I spent a few months discovering and correcting these problems. The most critical problem which I met was not being able to know what went wrong. As mentioned before, the same error may be due to many different reasons in many different places. During the debugging period, I modified a portion of correct code countless times. I encountered several different problems at the same time, and I realized that in order to make the whole program work, I must fix every single problem in the program all at once. As a result, everything which I previously implemented for my program seemed to be wrong. I had changed many instances of correct logic to wrong logic, and my program yielded even worse results.

I organized problems into two major categories: timing problems and non-timing problems. As mentioned above, timing is very important when programs attempt to communicate with each other. In the computer world, every micro-second may lead to a completely different result. Therefore, I spent a lot of time arranging the order of function calls as well as working on timing — trying to connect results with time very neatly. During communications, the major problem is a waiting problem. A waiting problem is also known as an EMACS timing problem. Such a problem is caused by the uncertainty of total waiting time required by the external lexical analyzer to read from the source buffer and write results to an intermediate buffer. As mentioned in the last paragraph of Section 3.3, I solved the problem by adding in the waiting time of one tenth of a second (0.1s). Therefore, I can always assume the programs at both ends of the pipe are running and the results are always processed by EMACS and generated by the external lexical analyzer.

As soon as I fixed the waiting problem a new problem arose. I ran into another waiting problem — a waiting for the result problem. Since EMACS has a one tenth of

```
(defun aldor-process-done-sentinel (process string)
  (when (memq (process-status process) '(exit signal closed nil))
    (setq aldor-scanner-done t)))
```

Figure 3.4: The `aldor-process-done-sentinel` Function

a second (0.1s) waiting time to fix the read or write problem caused by an external lexical analyzer; it is always waiting for new results. Even though my external lexical analyzer had already finished and terminated, EMACS still waited for more results from my external lexical analyzer. This problem also took me a long time to figure out. Once the problem was identified, it was very easy to provide a solution. The solution I came up with is to check the status of my external lexical analyzer procedure (see Figure 3.4).

The sentinel function which I wrote is called `aldor-process-done-sentinel`. The sentinel tracks the status of the process (external lexical analyzer). This sentinel is triggered if the state of the process changes. If the status changes to any of following states — `exit`, `signal`, `close`, or `nil`; then it sets the variable the `aldor-scanner-done` to `t` (true). The `aldor-scanner-done` variable has boolean type. The purpose of this variable is to determine whether my external lexical analyzer procedure has completed, exited or is in any of the terminating states.

Another critical problem which I encountered is the `erase-buffer` issue. This problem is caused by an intermediate buffer having been deleted or removed before my external lexical analyzer starts to scan the source buffer. For this reason, once my external lexical analyzer finishes scanning the source buffer, it cannot find an intermediate buffer to write results to. I overcame the `erase-buffer` problem by re-implementing my entire program, changing the order of function calls, and introducing the `aldor-process-done-sentinel` function which I discussed above.

One of the side effects of a timing problem is the delay problem. A delay problem is caused by bad timing in EMACS. The function which reads from an intermediate

buffer is called `aldor-read-from-list`. The problem was that my program terminated before the `aldor-read-from-list` function was called. In other words, the EMACS procedure finished my program before an external lexical analyzer wrote its result to an intermediate buffer. Fortunately, the `aldor-process-done-sentinel` function also fixed this problem. My `aldor-process-done-sentinel` function ensures that the EMACS program and `aldor-read-from-list` function do not terminate before the procedure (external lexical analyzer) is complete.

On the other hand, there are also many other non-timing problems which resulted in my program not working properly. One of the most basic and important observations is that some commands in EMACS have side effects. Therefore, it is very important for EMACS-LISP programmers to know side effects for each action they take. Any EMACS-LISP programmer who does not understand the side effects, would likely have bad result. If EMACS-LISP programmers comprehend the side effects of functions, then the side effects will certainly become very powerful features. For example, inserting a string into a buffer is not simply just inserting a string into a buffer; it also has the side effect of moving `point` which stores the current position or location of the text cursor. Furthermore, an insert function also has a side effect on `point-max`. This function returns the last text position or location of a buffer which is also known as the end of the buffer position. As another example, the function called `save-excursion` is one of the functions which has the side effect of restoring states of a buffer.

The most critical problem which I encountered of non-timing problems is a "point" problem. A "point" problem occurs when `point`, the current text cursor location, has not been set properly. There are many different variations of problems which arise from the point problem. For example, one of the problems I encountered has "point" passed `point-max` (the end of a buffer); as a result, the program either runs an infinite loop, as it is not able to reach the end of the buffer; or returns a

```
(defun aldor-restore-point-filter (proc string)
  (with-current-buffer (process-buffer proc)
    (save-excursion
      (goto-char (process-mark proc))
      (insert string)
      (set-marker (process-mark proc) (point)))))
```

Figure 3.5: The `aldor-restore-point-filter` Function

program exception error.

For another example, during the waiting time (100 ms) for EMACS, it is the only chance and opportunity to read the information from an intermediate buffer (similar to the Reader and Writer problem). However, once my external lexical analyzer resumes its work, it will then continue to write information into an intermediate buffer until either the entire source buffer has been scanned, or 100 milli-seconds waiting period expires again. Regretfully, as my external lexical analyzer writes results into an intermediate buffer, it has the side affect of moving `point`, the current cursor location to another position, causing EMACS to not know where it should resume its information reading. Additionally EMACS cannot start from a new `point`; otherwise, it will result in a section of code without desired properties.

The solution which I developed is to add in a filter which would restore `point` to its original place — reset it to exactly where it was located before new information was inserted. The function which makes this filter possible is called `save-excursion`. A `save-excursion` function remembers the original state before the codes inside its scope has been executed. As a side effect of insertion, the `state`, `point`, and `max-point` are modified. Figure 3.5 is the actual implementation for the filter.

The logic for this function is very simple. Firstly, it sets the current working buffer to the intermediate buffer. As mentioned, setting the working buffer correctly is very important; otherwise, it may end up modifying my programming

buffer or source file instead of an intermediate buffer. The second step includes using a `save-excursion` function to store the original state before any changes have been made. Thirdly, the program moves the current working text cursor to wherever `process-mark` of the process is currently located. Usually the value of `process-mark` is located at the end of buffer. This function then inserts result strings into an intermediate buffer. As mentioned, the side effect of insert function moves `point`. As a final step, the function sets `process-marker` to wherever `point` is. After exiting the filter, `save-excursion` function restores the point to whatever position it was in before entering the filter. Therefore, `point` will remain correctly positioned.

Some of the functions which I used to solve problems include `aldor-colour-syntax` function and `aldor-get-1-value` function. The `aldor-colour-syntax` function solved the problem for properties which had not been inserted properly. Additionally, I made many different attempts to improve the efficiency for this function. In the end, I decided to use an EMACS built-in function, `put-text-property`, to implement this function.

There are two versions of the `aldor-get-1-value` function. One of my approaches passes a marker as the argument; the other approach, which I used for my final version, passes a buffer (intermediate buffer) as the argument. A successful `aldor-get-1-value` was implemented after several failures. The logic for this function can be broken down into many smaller pieces (see Figure 3.6).

Initially, the `aldor-get-1-value` function sets variable `value` to an intermediate buffer or nil according to the state of the intermediate buffer. Next, this function initializes the `done` variable to `nil`. After which, this function loops through the intermediate buffer until my external lexical analyzer has completed and no more values can be read from the intermediate buffer. At this point, this function sets the variable `done` to the value of either `aldor-scanner-done` which was set within

```
(defun aldor-get-1-value (buffer)
  (let ((value
        (condition-case nil
            (read buffer)
          (error nil)))
        (done nil))
    (while (and (not value) (not done))
      (setq done (or aldor-scanner-done
                     (null (accept-process-output nil 1))
                     (sit-for 0.1)
                     ))
      (setq value
            (condition-case nil
                (read buffer)
              (error nil))))
    value))
```

Figure 3.6: The `aldor-get-1-value` Function

the `aldor-process-done-sentinel` function or the value returned by the built-in function `sit-for` (returning true means that at least a portion of the input has been read); or to the negation of the function `accept-process-output`. Finally, the function sets the variable `value` to either the result from the read buffer, or nil if there were any errors.

In conclusion, the sentinel and filter functions are the keys to solving most of my timing as well as non-timing problems; without them, this lexical analyzer would not work. A sentinel notifies a program of a change of status of the process. Therefore, a program is able to know when a scanning process is complete, and whether or not it is an appropriate time to read from an intermediate buffer. On the other hand, a filter function helps to set the point properly for programs, which solves the most critical non-timing problem, the "point" problem. Fixing problems actually took less time than figuring out the location and cause of the problems. Overall, the most challenging part of my whole research is a combination of timing and communication

problems.

3.7 Conclusion and Summary

In conclusion, this part of the program has been the longest and most challenging portion of my entire graduate term. Nonetheless, the purpose of my research is mainly based on computer communication and Emacs learning from outside sources. Since I encountered so many different tribulations, I gained great experience in the area of debugging communication problems and building techniques to solve these bugs.

Moreover, I learned how to handle the timing problem and was actually able to think through most of the possible solutions. The outcome of my work is extremely gratifying. The external lexical analyzer for Aldor is working smoothly. However, the most important success of this work is not the completion of an external lexical analyzer; instead, it is how to connect the communication components between programs. Please refer to Appendix E for the UML diagram of my external lexical analyzer.

This chapter discussed external lexical analyzers, and their concepts. The next chapter describes development of an external parser. My external parser uses the Flex language as well. However, the implementations and involved components are completely different.

Chapter 4

The Aldor Parser

For a programming text editors, a parser is one of the most important components. If a lexical analyzer is the heart of a programming text editor; then a parser is the arms and legs. In other words, without a lexical analyzer, it is impossible to implement a programming text editor with any tools. It is very easy to construct a text editor; however, it is not as easy to create a programming text editor. The major difference between these two types of text editors is that a programming text editor actually has to know the programming language in order to be able to help a programmer. On the other hand, a plain text editor does not require any information of its contents. Therefore, a plain text editor will not be as much help to programmers.

Nevertheless, there are some limitations to a lexical analyzer. A lexical analyzer has the ability to classify tokens. Furthermore, some of lexical analyzers have the capacity to present an indentation feature for programming languages. However, it will never be able to provide some useful features which a parser can provide. For instance, a lexical analyzer cannot help programmers to identify syntax errors; nor can it help programmers to find the start and end locations of a function, type of a variable, loops, or scope bounds.

For these reasons, it is possible to complete programming text editors without implementing a parser (just as people can survive without their legs and arms); however, with additional information supplied by a parser, a programming text editor will become more powerful and have the ability to help programmers to code their programs (with arms and legs, people can do more).

4.1 Overview of Building the Parser

As discussed in the introductory chapter, ALDOR has a reasonable number of grammar rules. This was one reason I decided to use ALDOR as my target programming language. One of main challenges I face is an ambiguous grammar problem. There will be more discussion of this topic in a later section of this chapter. Moreover, it is very difficult to find more information related to ALDOR since this programming language is relatively young, and not as popular as other widely used programming languages.

One of the main objectives which I tried to reach is separating a processing text editor from its reference resource. Any ordinary text editor will have a chance to work for particular programming languages with the minimum modifications. For this reason, I decided to implement my parser externally.

Similar to my external lexical analyzer for ALDOR, the communication between a parser and EMACS is one of the major challenges. The communication which involves a parser is more complicated than an external lexical analyzer. For an external parser to work, it requires three-way communication between the EMACS text editor, an external ALDOR lexical analyzer, and an external ALDOR parser.

4.2 Method

There are many references on the Internet that provide information on programming language parsers. The implementations are very different between a parser and a lexical analyzer (see [8], [14]). Parsers requires much more information about programming languages than lexical analyzers need. One of the reasons for this is that for every parser, there must be a lexical analyzer to provide token information. As a result, it is very important to understand communications between a parser and its lexical analyzer.

I did my external lexical analyzer for ALDOR in the programming language called `Flex`. Hence, I mainly concentrated research on parser programming languages which matches `Flex` tokenizer programming language. Since `Flex` is a very popular tokenizer generation programming language, it is quite easy to find a parser language which will be able to communicate with `Flex` and accept the results which are generated by `Flex`. Some parser generating languages which support `Flex` are `YACC` and `Bison` (see [9], [5]). `Bison` is an upgraded version of `YACC`; they have almost identical syntax and almost identical functionality. However, `Bison` is newer and provides more features than `YACC`.

The primary goal which I tried to establish is finding the communication methods and algorithms between `Bison` and `Flex`. Regretfully, as I gained more knowledge about the algorithms which establish the communication between `Bison` and `Flex`; it became very clear to me that I must re-implement my entire external lexical analyzer. In other words, I have to re-implement my external lexical analyzer and change the algorithms which I used.

As I implemented the parsers, I suddenly realized that `Bison` does not support the grammar which is provided by ALDOR official site. See Section 4.6 for details.

Eventually, I was able to make `Bison` to accept ALDOR grammar and started

the implementation of an Aldor parser. When I implemented my external lexical analyzer in `Flex`, I had not prepared it to work for a parser. Therefore, I re-implemented the `Flex` part of the program to create a lexical analyzer that generates the information which is required by a `Bison` parser.

After many different approaches and comparisons, I combined some algorithms and implemented my Aldor parsers. There are two major versions of parsers which I constructed. A tree structure algorithm parser generates a parse tree that contains all parsing information. The second parser used an update "on the fly" algorithm. This parser generates results much faster. However, the information which can be retrieved from an update "on the fly" algorithm parser is very limited. For instance, such parser does not generate any knowledge about a programming language structure. Thus, it is very hard for one to use it to implement an indentation feature. An update "on the fly" parser processes information directly to a buffer without knowing any details of parsing structure. It just follows the grammar and returns parsing scopes once a scope can be identified.

Finally, after some discussion with my supervisor, and survey with other programmers on their preferences; I decided to implement a tree structure parser. Although it runs slower than an update "on the fly" parser; it is able to provide more information and help for programmers.

Nevertheless, I would like to discuss the concept of the update "on the fly" algorithm as it will be easier to discuss the difference between these two very dissimilar parsers and their algorithms. The update "on the fly" algorithm is an one-pass parsing process. Bison generates a tree bottom up so it is not as easy to create an one-pass parse tree. This version of parser returns results as it move from one grammar rule to another grammar rule. As long as the parser can parse its source file and succeed with some grammar, then it will trigger its functions to return results. On the other hand, a parser which is implemented in this algorithm is simpler

compared to other parsing algorithms (not to mention the more complicated tree structure algorithm).

For an update "on the fly" parser to work, there are some required variables. Similar to an external ALDOR lexical analyzer, a position variable is needed to store token positions. However, the concepts and algorithm of the position variable is a little bit different for this parser. From the view of bottom level, each token which is generated by the lexical analyzer is a terminating node for this parser. On the other hand, there are many non-terminal nodes. The basic concept for this algorithm is for a parent scope to fetch the starting point of its first child and the end location of its last child. There are a few possible cases.

Firstly, if it found a token, then it is in a terminating rule. For this reason, it does not have any children; thus, the starting and ending point of its children are the same as its own starting and ending point. Secondly, if it is a non-terminal rule, and it has only one child, then the start and end point for this particular non-terminal rule will also be exactly the same as its child.

Finally, in a rule which has more than one child, then the starting point for this scope will be the start point of its first child; and the end position will be the end position of its last child. Additionally, the start and end point are inherited from the bottom level.

The key parts of the update "on the fly" parser was implemented in two files called `aldor_uof_parser.lex` and `aldor_uof_parser.y`. The token communication mechanism between these two parts is internal. Since `Flex` and `Bison` made a very good parser package, `Flex` is able to generate the information which Bison needs. After some modifications and code implementation, I am able to make `aldor_uof_parser.lex` application generates tokens and pass require information to `aldor_uof_parser.y` applications. Following that, `aldor_uof_parser.y` generates some EMACS-LISP code and output results to an intermediate buffer. Finally,

```
int init_p = 1;

/* Function prints the start point, end point and the type of a token */
void set_lisp( int& start, int leng )
{
  yylloc.first_column = start;
  yylloc.first_line = line_num;
  yylloc.last_column = start + leng;
  yylloc.last_line = line_num;

  start += leng;
}
```

Figure 4.1: Function that Prints the Start and End Point

I had EMACS fetch the EMACS-LISP code from an intermediate buffer, then execute and process all information in the intermediate buffer. Please refer to Appendix F for the UML diagram of the parser.

The language used in an intermediate buffer is EMACS-LISP. In other words, `Bison` returns a complete EMACS program to an intermediate buffer. For this reason, EMACS only needs to wait for the completion signal from an external parser, and then execute the program which is in an intermediate buffer.

Figure 4.1 shows the implementation of a function which set the start and end position for an update "on the fly" parser.

Similar to an external lexical analyzer, the `init_p` variable acts as a position pointer which used to store the position of the current token. The `yylloc-` variables are `Flex` built-in variables which help `Bison` to get the start and end position along with the first and last line numbers. As mentioned in Section 4.2, `Flex` and `Bison` are a working pair and thus their variables are designed to fit each other perfectly. Figure 4.2 contains some applications of `set_lisp` to various cases.

My `set_lisp` function stores information which `Bison` needs to defines a terminal rule. At the end of each `Flex` definition, it returns a token type. Therefore,

```
/* examples for some of the tokens */
"mod"       { set_lisp( init_p, yyleng);
              return MOD;}
"quo"       { set_lisp( init_p, yyleng);
              return QUO;}
"rem"       { set_lisp( init_p, yyleng);
              return REM;}

/* example for some of the non-constant tokens */
/* import functions                            */

stdout|newline|space|include|rep|per|Per {
  strncpy(tokenString,yytext,20);
  set_lisp( init_p, yyleng);
  return ID;
 }

/* example for some of the operators         */
/* Other ops - temporary colour as future ops*/
"::"    { set_lisp( init_p, yyleng);
          return DCOLON;}
"."     { set_lisp( init_p, yyleng);
          return DOT;}
":*"    { set_lisp( init_p, yyleng);
          return COLON_STAR;}
"+-"    { set_lisp( init_p, yyleng);
          return PLUSMINUS;}
```

Figure 4.2: Example of Token Handling

Bison is able to terminate in a terminal rule and finish parsing. Figure 4.3 has some examples of Bison codes which interact with the information returned by Flex.

A printTree function prints out a message which can be executed by EMACS. @$.first_column and @$.last_column are Bison variables which find the start and end position of its child or children.

The implementation for a tree structure parser is completely different. There are many different files involved in order to construct the algorithm. The required files are as follows: global.h, util.h, util.c, aldor_parser.lex, and aldor_parser.y. Please refer to Appendix D for instructions about how to find the complete code for these files.

In order to build a parse tree, I first categorized the grammar and developed an abstract syntax grammar for ALDOR. I then defined tree nodes so source code can be broken down into many nodes, with each of these nodes having the ability to connect to other nodes. It was a very long task to develop and design such a nodes list. I managed to categorize all nodes into seven major types. Moreover, for each of the major node types, I divided them into several smaller sub-node types. The main challenge was to define exactly the numbers of required node types. If there are too many different type of nodes, a parse tree will become too complicated and the parsing process will be very slow. On the other hand, if there are not enough node kinds, parts of a parse tree will not be connected and the result is an incomplete parse tree. This leads to a node shortage problem.

For the ALDOR grammar, the maximum number of children required in a parse tree is three. For example, for the "if" statement the first child defines conditions for the statement; the second child defines actions to be taken if the evaluation of conditions turns out to be true; and the third child is an "else" statement, which defines actions to be taken if the evaluation of first argument turn out to be false. Regardless of the number of children, a tree node also needs a sibling link which

```
/* For the code on the parser side - Bison code */
/* Here are some examples                       */

cases:
            binding_Collection
            {
              print_overlay( @$.first_column, @$.last_column, "cases");
            }

casesOpt:
            binding_Collection
            {
              print_overlay( @$.first_column, @$.last_column, "casesOpt");
            }
          | /* empty */
            {
              print_overlay( @$.first_column, @$.last_column, "casesOpt");
            }

id:
            ID
            {
              print_overlay( @$.first_column, @$.last_column, "id");
            }
          | POUND
            {
            print_overlay( @$.first_column, @$.last_column, "id");
            }
          | TILDE
            {
            print_overlay( @$.first_column, @$.last_column, "id");
        }
```

Figure 4.3: Example of Bison Codes for the Update “On The Fly” Algorithm

links to the next node; and requires another variable to store such a link.

There are differences between a sibling node and a child node. A sibling node lies on the same level as its sibling. On the other hand, a child node is one level beneath its parent node. A child node's scope is a subset of its parent's scope; while a sibling node's scope is independent from other sibling node's scopes. Keeping this distinction makes indentation possible.

As I developed an abstract syntax for ALDOR, I inserted a kind of node called "`TempNode`". "`TempNode`" will not display in a parse tree, nor will it apply an overlay to a source file. In other words, certain nodes generate scopes which I did not find useful to users. For this reason, I replaced these kinds of nodes as "`TempNode`". However, my implementation for tree nodes is very flexible and easy to modify. Programmers can replace one or more "`TempNodes`" by some new type of nodes which they declare. However, they must then insert and modify some files. As a result, new node scopes will be displayed in a parse tree if found, and overlay scopes will be applied to the source code.

Similar to the update "on the fly" algorithm, a tree structure algorithm also needs the `set_lisp` function to store the start and end locations for all non-terminal and terminal rules (tokens). On the other hand, in `Bison`, every rule must create its own node according to its type in order to construct a parse tree. Fortunately, I already spent a lot of time studying ALDOR and had resolved the ambiguity of ALDORS' grammar. Hence, I did not have a tough time classifying node types for the rules. As soon as a node is created, variables of the node will store all the information. Finally, I attach a sibling or children to the node if they exist.

There are some `Bison` variables which I used to build parse trees. The variable `$$` represents nodes. Variables `$1` and `$3` represent the first and third node of their parent node. The "`newTempNode(TK)`" function and other different kinds of node generator functions are defined in the files "`util.h`" and "`util.c`". Figure 4.4 shows

```
infixedExprs:
            infixedExpr
            {
              $$ = $1;
            }
          | infixedExpr COMMA infixedExprs
            {
              $$ = newTempNode(TK);
              $$ ->start_position = @$.first_column;
              $$ ->end_position = @$.last_column;
              $$ ->token_type = "infixedExprs";
              $$ ->child[0] = $1;
              $$ ->child[1] = $3;
            }
```

Figure 4.4: Example of Tree Structure in `Bison`

one example of my tree structure parser algorithms. After many re-implementations and corrections, I was finally able to get my parser to parse an ALDOR program properly.

The parser program which I designed not only has the ability to parse ALDOR programs, but also presents a syntax error checking feature. For this reason, my parser is able to notify programmers of syntax errors with their location and line number. I did once think of implementing error recovery features; but it does not fit into my research topic. For this reason, error recovery features are left as future improvements.

Lastly, there are two versions of output for parsers which use the tree structure algorithm. One of these parser prints results as a syntax tree while the other parser generates EMACS-LISP code, so that it can communicate with the EMACS text editor.

For the parser, EMACS does not use filters or sentinel functions. It simply waits for the signal which is generated by the external parser, then evaluates the intermediate buffer (`aldor_parse_middle.el`) which contains results from my external

parser. By evaluating the intermediate buffer (`aldor_parse_middle.el`), EMACS inserts overlay properties on the source buffer according to information which is generated by the parser. The concept of an overlay is different from that of text properties. Overlay properties belong to a region of a buffer while text properties belong to a text string. For this reason, if one moves a text token, EMACS also moves the text properties associated with the text token; however, the same action will not affect an overlay. Additionally, an overlay region expands as soon as the programmers insert more text into the region.

In conclusion, both versions of parser provides parsing information. However, each parser has its own advantages and disadvantages. An "update on the fly" parser returns the result faster than a tree structure parser. On the other hand, a tree structure parser provides more information to the user than an "update on the fly" parser. Therefore, although both parsers serve the same purpose, parse the source program; but they can be apply differently according to users' needs.

4.3 Communication

The method which I used to establish communication between an external parser and EMACS is different from the method which I used for my external lexical analyzer and EMACS. The major difference between these two methods is how they handle the intermediate buffer.

The external ALDOR lexical analyzer produced semi-EMACS-LISP code. As a result, EMACS must read results one by one from the intermediate buffer and process each of these results individually. The external parser, however, generated pure-EMACS-LISP code. For this reason, EMACS can evaluate the entire intermediate buffer all at once.

Moreover, communication with my parser program is three-way communication.

As mentioned in Section 4.2, a `Bison` parser program acts as an intermediate program to bridge communication between a lexical analyzer and EMACS. One of the main focuses of this parser exercise is to learn communication methods which involve more than two programs.

The communications between a lexical analyzer and a parser program is as follows. A `Flex` lexical analyzer identifies tokens and returns them one by one. The `Bison` parser, requests tokens one by one from the `Flex` lexical analyzer. This process continues until every token in a source buffer has been scanned. Communication between `Bison` and `Flex` is a series of function calls. `Bison` starts a loop and terminates only if all `text` in the source buffer has been processed, the parser has parsed the initial goal, and a tree has been successfully built. During the procedure of `Bison` creating a tree, it calls the `Flex` program repeatedly until termination. The `Flex` lexical analyzer returns one token at a time to the `Bison` parser and stays at the beginning of the next token, ready for the next request from `Bison`. However, if the lexical analyzer reaches the end of the buffer, it continually returns an end-of-file token. Finally, when `Bison` finished, it generates the final results in an intermediate buffer if the source buffer can be parsed correctly; or it returns syntax error messages to notify users that the parsing process failed.

The outcome of this exercise is acceptable and the greatest shortcoming was a lack of speed. As I used an external lexical analyzer and an external parser, it is not unreasonable for my program to have a long processing or parsing time. Even though I tried some other methods, the main predicament is still related to communication. As the size of the programs grow, communication between these programs will increase as well. For this reason, it will take longer for a driver program to receive results from these external programs.

In conclusion, there is a chain reaction in the communication network. As each program is delayed for a short period of time, the whole package ends up becoming

inefficient. For this reason, it is very important to design a good algorithm for programs to communicate with each other without waiting for each other. Nevertheless, some waiting is inevitable.

4.4 Efficiency

The overall efficiency for my external parser is fair. The external parser program has acceptable memory resource management results; however, the time resource management is not as good.

Firstly, space efficiency had been done well for the update "on the fly" parser. For this version of the parser, the only extra memory storage required is an intermediate buffer. On the other hand, a tree structure parser requires a lot of extra space to store its nodes, node types, and all the information which is associated with the nodes. For this reason, the tree structure parser has the worst space efficiency over the entire collection of parsers which I designed.

Secondly, the time efficiency for update "on the fly" parser is good. Since this algorithm updates an intermediate buffer almost instantly; an update "on the fly" parser has the best time management among the parsers which I developed. However, this parser program still has to wait for its lexical analyzer program to finish classifying tokens. The tree structure parser is even slower because it has to construct a tree and assign value to its variables. For instance, it take about one minutes to parse 1400 lines of ALDOR code. Hence, I do not claim that the time efficiency is good for any of the external parsers. Figure 4.5 shows the timing data for my tree structure parser.

Finally, the usefulness of result are very clear. An update "on the fly" parser only provides basic overlay information to EMACS, making the usefulness for an update "on the fly" parser poor. On the other hand, the usefulness for a tree structure

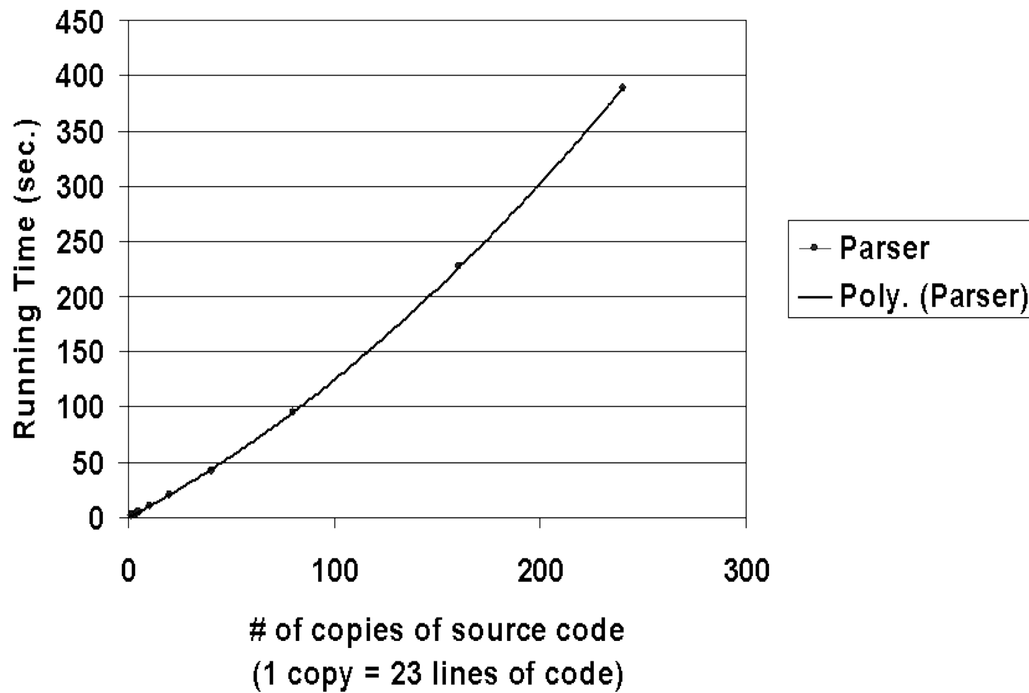

Figure 4.5: Timing Data for Tree Structure Parser

parser is acceptable. According to my research, a tree structure parser provides the maximum amount of information compared to any parser algorithm. In other words, there is no more information which a parser could supply other than returning a complete tree structure with information on each node.

In conclusion, time efficiency is the most critical drawback for a tree structure parser. On the other hand, a tree structure parser has the ability to provide maximum information to its requester. Therefore, it is very hard for one to judge which of these algorithms is the better approach. An update "on the fly" parser will satisfy the users who look for speed and some basic parsing information. Conversely, most of the programmers may care about functionality, as long as the running time is reasonable. As a result, I implemented the parsing process as an on-call function, instead of an auto-load function. Consequently, programmers can parse any ALDOR

program through ALDOR mode in EMACS at any time they wish.

4.5 Correctness

The parsers correctly identify the start and end position of each rule block. Thus, EMACS is able to apply overlay regions correctly to the source buffer. Additionally, a tree structure parser has the ability to print out an entire parse tree if there is no syntax error in the program. Otherwise, it will print the location and the line number of the syntax error.

Moreover, I reduced the rules and developed an abstract syntax grammar for the ALDOR programming language. As a result, I successfully cut the node count from three thousand four hundred to ninety nine (for `parser_simple.as`) without affecting the tree structure. The abstract syntax grammar is a grammar which is developed from the concrete syntax. It has fewer rules but is still able to parse a program properly. I also successfully reduced overlay insertion commands in an intermediate buffer by removing any unneeded nodes in `Bison`.

In conclusion, there is no possibility for a `Bison` generated parser to go wrong when all of information produced by the `Flex` lexical analyzer is correct. Some possible errors may be caused by not correctly setting the token position or by not properly setting programming language grammar rules. Since I already developed a tree structure parser, other programmers can just use my programs as a template and modify the grammar rules.

4.6 Problems Encountered and their Solutions

The ALDOR team has their own parsing process. They pre-process the parameterized grammar by another program to generates non-parameterized rules before

1. Create some non-terminal rules for each level of precedence.
2. Isolate the corresponding parts of the grammar.
3. Force the parser to recognize higher precedence sub expressions first.
4. Use some ad-hoc rules in the parser (match up `else` with the closest `if`).
5. Change the abstract grammar as required. Use temporary nodes and the like.

Figure 4.6: Rules to Resolve Ambiguous Grammars

passing the processed grammar to a parsing tool such as `YACC` or `Bison`. For this reason, `Bison` thought the parameterized grammar from ALDOR official site is ambiguous. For example, the `BindL(R,L)` in original ALDOR grammar requires some reworking in order for `Bison` to accept the rule. There is a lot of information which teaches programmers to resolve a set of ambiguous grammar. Figure 4.6 contains some of key points which I used to work on the ALDOR grammar rules.

After I fixed the grammar, there were still many reduce-reduce and reduce-shift problems which existed. A reduce-reduce conflict occurs if there are two or more rules that apply to the same sequence of input. The most famous example of a shift-reduce conflict is the `if-then-else` statement. Consider an `if-then-else` statement that has two `if` and one `else` statements. The problem is to define which `if`-statement the `else`-statement should bind to. Both reduce-reduce and shift-reduce conflicts are ambiguity errors. As mentioned in Section 4.2, `Bison` has the ability to recognize the ambiguities of a grammar. Hence, with help from `Bison`, I eventually developed a set of unambiguous ALDOR grammar rules.

As I worked on the grammar, I also noticed that some of the rules could never be reached. One of the reasons is that some tokens required by these rule cannot be generated. If a token is not in the lexical analyzer rules, then the token will not be generated. Thus, rules which contain these tokens will never be reached. Since these rules are unreachable by any grammar, they will not affect the correctness of my

```
curlyContentB_Labelled:
            preDocument labelled postDocument
            {
              $$ = newBlockNode(Curly1K);
              $$->sibling = $1;
              $1->sibling = $2;
              $2->sibling = $3;
            }
```

Figure 4.7: A Tree Structure with a `NULL` Problem

parser. As a result, I simply ignored those rules. The original grammar for Aldor is found at Aldor's official website and a new Aldor grammar which I developed is found in Appendix B. Additionally, the abstract syntax which I developed is found in Appendix C.

Among all the problems I encountered while developed this parser, the sibling problem was the most troublesome issue. The sibling problem actually took me a very long time to figure out, and it has a few variations. Firstly, there are chances for a node in a programming tree to be `NULL`. If a node has not been created and has thus a value of `NULL`, then it will be a big problem when the parser tries to assign a sibling to it. Figure 4.7 is an example of this situation.

The first time I encountered this problem was with the `Post-Document`, `Pre-Document`, and `Comment` rules. It is very common for this situation to occur. Since not every programmers writes pre-documents, the `Pre-Document` token may not exist. In this case the `Pre-Document` token node will be a null node. Therefore, when a parser tries to assign a sibling to it, the parser will crash. For this reason, I created some generic non-terminal rules to make everything work. I also checked every rule which has the possibility for this problem to occur. Figure 4.8 shows actual code of my solutions for this problem.

The second variation of a sibling problem is the over-written sibling problem. I did not notice this problem until I printed my syntax trees. I spent a long time

```
curlyContentB_Labelled:
            preDocument labelled postDocument
            {
              if( $1 == NULL && $2 == NULL )
                $$ = $3;

              else if ($1 == NULL && $3 == NULL)
                $$ = $2;

              else if ($2 == NULL && $3 == NULL)
                $$ = $1;

              $$ = newBlockNode(Curly1K);
              $$->child[0] = $1;
              $$->child[1] = $2;
              $$->child[2] = $3;
            }
```

Figure 4.8: Solution for the Tree Structure with `NULL` Problem

identifying this problem and creating a solution for it. This problem also occurs quite frequently. This problem happens when a program assigns a child node as the node itself. Unfortunately, if the child node has a sibling, it will over-write the sibling of its parent node. Consider Figure 4.9 as an example. In this example, `ParentNode` has the body of `FirstNode` and is a sibling of `SecondNode`. Additionally, `FirstNode` (body of `Parent`) can be derived as the `ChildNode` with the sibling of `ChildSibling`. Since `Bison` is a bottom up parser which builds the parse tree from terminal nodes (tokens) to parent nodes. `SecondNode` will be assigned as the sibling of `FirstNode` before `FirstNode` has been evaluated. When `FirstNode` is later evaluated, it is assigned to `FirstNode` (`$$ = $1`); which causes `SecondNode` being over-written by the `ChildSibling`.

I solved this problem by add another level of non-terminal nodes. Consider the example in Figure 4.10. `FirstNode` and `SecondNode` are both children of

```
Parent:
     FirstNode SEMI SecondNode
     {
         $$ = $1;
         $$->sibling = $3;
     }

FirstNode
     ChildNode ChildSibling
     {
         $$ = $1;
         $$->sibling = $2;
     }
```

Figure 4.9: A Tree Structure with a Sibling Problem

```
Parent:
     FirstNode SEMI SecondNode
     {
         $$ = newTempNode(TK);
         $$->child[0] = $1;
         $$->child[1] = $3;
     }

FirstNode
     ChildNode ChildSibling
     {
         $$ = $1;
         $$->sibling = $2;
     }
```

Figure 4.10: Solution for the Tree Structure with Sibling Problem

`ParentNode`. Therefore, the order of these two nodes will not get mixed up. Additionally, since every node has its own branch, an over-write problem does not happen. Similar to the `FirstNode` situation; one can always add in a level of nodes to prevent an over-written sibling problem.

In addition, there is a node shortage problem. If there are not enough types of nodes to connect a parse tree, then the tree will not parse properly. As mentioned in Section 4.2, I tried to develop an abstract syntax grammar. A node shortage problem was caused by deleting the connection nodes from the parser. There are references on-line that talk about how to develop abstract grammars from concrete syntax grammars. It is worth the time and effort to create the abstract syntax grammar to improve parser efficiency.

Lastly, there are communications problems between `Bison` and `Flex`; `Bison` and the utility files; and `Bison` and EMACS. However, the major problems are improper buffer settings and buffer waiting problems (see Section 3.6 on page 44).

In conclusion, the major problems for my parser are ALDOR grammar rules and the implementation logic of `Bison`. Additionally, communication between `Bison` and other programs can be confusing at times.

4.7 Future Work

One of the major improvements which to add to the parser is an automatic re-parse system. I consider such as a future improvement, since it does not directly relate to my current research topic. The major concepts for this re-parse system are as follows: (1) when to re-parse and (2) what to re-parse. A parser should re-parse its source buffer once the buffer had been modified. However, I do not wish to re-parse the source buffer very often, since the parsing process is very time costly. Therefore, I propose that a parser only re-parse its source buffer if a source buffer

is idle for more than a constant amount of time. On the other hand, what should be re-parsed? Re-parsing a region of a source buffer would definitely be faster and more efficient than re-parsing the entire source buffer. Nevertheless, if I wish to re-parse part of source buffer, I have to find an appropriate parsing scope first. In some cases, modification of the buffer may lead to a completely different parse tree. Therefore, how far (on the scope levels) should a parser trace back? Determining this will be major challenge for a re-parsing process.

In conclusion, the re-parsing process is not simple, and it requires more research. For this reason, I left the automatically re-parsing process as a future improvement. At this point of time, users must re-parse their source buffers manually.

4.8 Conclusion and Summary

In conclusion, a tree structure parser is slower but more powerful than an update "on the fly" parser. The issue is the substance of what users want and what they need. For some people, a tree structure parser may be more than what they need. On the other hand, for some people, an update "on the fly" parser may not provide enough information.

The main learning and research foci of external parsers are the algorithms for parsers and communication between multiple programs. One of the greatest advantages for me is that I had already written an external lexical analyzer. As a result, when I performed research related to communication on multiple programs, the research results I found made more sense to me. Moreover, my experience helped me in the debugging process and in finding find problems. It also allowed me to figure out possible communication mistakes faster. Additionally, if I had not completed the external lexical analyzers, I would not have been able to finish these parser. I was able to learn the methods of multi-program communication, and to perform

several different kinds of parsing techniques.

Examination of a few different kinds of parser algorithms made me think very carefully on the balance of a program. I learned that to implement some features, other features may have to be sacrificed. Overall, there is really no guideline or standard for a good program. Different people have different demands for a given program. It all depends on what a user needs.

In this chapter, an external ALDOR parser has been discussed. In the next chapter, an ALDOR mode for EMACS will be introduced. An ALDOR mode puts lexical analyzers and parser into a package, and provides other tools.

Chapter 5

Aldor Mode

There are already many different programming language modes which exist in EMACS. However, none of these programming modes are implemented outside of EMACS. Half of the functions I used to create my ALDOR mode for EMACS are implemented and executed outside EMACS. For this reason, the ALDOR mode which I present contains work which I believe has not been done previously.

The `aldor-mode` is the final parts of this thesis. An ALDOR programming mode for EMACS is a big package which collects together all functions which are related to ALDOR. The `aldor-mode` for EMACS wraps up the work I have completed into a single package. Moreover, there is no previously existing text editor designed to support ALDOR. Hence, for ALDOR programmers, EMACS may eventually become one of the most popular text editors that they use.

Nevertheless, there is still a lot of research left to add programming modes to EMACS. For instance, there are still many tools which can be implemented to support programmers. The whole idea of these "helping" features and tools in a text editor is to assist a programmer to code faster and more efficiently.

5.1 Progress and Methods

Although this component comes last within the thesis; research on language modes for EMACS occurred throughout my research. Among the entire collection of modes which EMACS provides, `cc-mode` is the most complicated programming mode in EMACS. For this reason, if one can understand the concept of `cc-mode` in EMACS, then they will be able to institute a very powerful ALDOR mode for EMACS.

Eventually, I abandoned the thought of implementing an `aldor-mode` similar to `cc-mode`. EMACS `cc-mode` is too complicated for an entry level EMACS-LISP programmer to understand. There are a few reasons which may explain why `cc-mode` is so complicated. Since `cc-mode` deals with "on the fly" updates to syntax analysis, the implementation for such features is very complex. Moreover, the language analysis occurs inside the package, which make it very hard to separate the analysis from the editing commands. However, the main concentration of this part of my research was on communication with external programs.

After both versions of the ALDOR lexical analyzers were completed, the research was mainly focused on the easiest and shortest programming mode — the `Pascal` programming mode. Unfortunately, it is not easy to write a new programming mode for EMACS just by simply studying programming language modes. One must also use the EMACS-LISP manual which presents a lot of information related to programming language modes and methods for adding a new programming language mode to the EMACS text editor.

Although there is a lot of information on-line which discusses how to add a mode to EMACS, most of these links redirected me to the GNU EMACS manual site or simply displayed partial contents of the EMACS-LISP manual.

My ALDOR mode provides some basic features, such as syntax-based colouring and parsing. On the other hand, the `cc-mode` in EMACS is more powerful and able to give more support to programmers. For example, the `cc-mode` updates and re-

Colours

Aldor *Types*	Emacs-LISP – *Colour*
Reserved	'font-lock-keyword-face
Definable Keyword	'font-lock-keyword-face
Class	'font-lock-type-face
Import Function	'font-lock-function-name-face
Pre-Document	'font-lock-doc-face
Post-Document	'font-lock-doc-face
Comment	'font-lock-comment-face
String	'font-lock-string-face
Identifier	'font-lock-variable-name-face
Float	'font-lock-constant-face
Integer	'font-lock-constant-face
Definable Operator	'font-lock-builtin-face
Reserved Operator	'font-lock-builtin-face
Future Operator	'font-lock-builtin-face

Table 5.1: The Colour Association Used by the Aldor Mode

colours a token almost instantly. However, my Aldor mode is expandable, more generic, and able to accept external resources. Hence it may be modified to create another programming language mode. Meanwhile, the `cc-mode` for Emacs is almost impossible to modify to suit another programming language.

5.2 Syntax-based Colouring

Among all programming text editors, syntax-based colouring is the most basic of requirements. Programming text editors should have the ability to identify different types of tokens and colour them with appropriate colours. Research on programming text editors shows that it is always a very good practice to follow the habits of the majority. In the same way, I intended to keep the syntax-based colour highlighting consistent within Emacs. Table 5.1 lists the colours which I associate with Aldor token types.

One of the greatest advantages of this approach is to ensure the consistency of

the font colours. The `font-lock-face` variables can be changed in the customize pages in EMACS. If programmers decide to change the colours which represents functions on their machines, they will end up changing the whole colour scheme. In other words, the colours will be kept consistent for different programming languages.

The concepts used to colour the token strings are very simple. All it takes is for a lexical analyzer to classify the token types. An internal lexical analyzer calls functions to put all the desired text-properties on token strings. An external lexical analyzer generates an intermediate buffer which provides all the information for EMACS to do its job; then EMACS calls the syntax-based colouring functions to add selected text-properties to text strings.

In conclusion, the colour syntax process for a programming language only requires a lexical analyzer for that particular programming language. As soon as the token can be identified, and token strings can be located, then everything will be completed quickly. Nevertheless, that is only an approximation of the truth. For instance, in C^{++}, there is a variable `c++-font-lock-extra-types` that the programmer can set so that classes like `string`, `list`, and `size_t` appear in `font-lock-type-face` rather than `font-lock-variablename-face`. The need for this variable results from the fact that `cc-mode` does not know how to parse header files. Similarly, in `aldor-mode`, it would be nice to put identifiers after "`:`" in `font-lock-type-face`, but this requires parsing.

5.3 Updating Algorithms for the Aldor Mode

An ordinary programming text editor should also provide real time updating features. For instance, as a programmer enters code, the editor should be able to update results immediately. Thus, a programmer will be able to tell if any syntax errors have occurred, and be able to see the modifications they have made.

Furthermore, a more powerful programming text editor provides features for updating the source buffer according to any of the insertions, deletions, or modifications applied by the programmers. When should text be updated? There are many different update algorithms. One algorithm suggests the newly modified text should be updated within an appropriate amount of time while performing a minimal number of updates. In addition, this algorithm suggests updating and refreshing the screen at constant time intervals. However, this will not work well with a huge buffer. Since the buffer contains so much information, such an algorithm is very inefficient and inappropriate for handling the buffer update. Therefore, I did some minor modification, to make a very powerful updating algorithm. The algorithm still refreshes the screen at constant time intervals. However, it now only updates the text field which has been modified or has not yet been coloured.

My approach was to update the region of text which is located two lines above and two lines below the current position. This event is triggered when the key "space" or "newline" is pressed. The algorithm assumes that the key press of "space" or "newline" is a signal of reaching the end of the token characters. There is a critical problem associated with this algorithm. If a programmer types a set of characters, and then moves the cursor and clicks the mouse at another location and starts typing again, the algorithm will not work properly since none of the end token characters has been entered.

The solution which I suggest to solve this problem is to remember the minimum and maximum location of the programmer's cursor. In this case, the programmer can click anywhere they wish to, and can modify whatever information they please. As soon as they press any of the end token characters, all texts which do not have properties will be updated all at once.

There are other text updating algorithms. For example, some text editors try to classify the entire text string which is currently being modified by a programmer.

For this reason, the text token is always up to date. As soon as there is any change in the text, it will trigger a lexical analyzer to classify the text token.

5.4 Features

There are some features which I provide for ALDOR mode in EMACS. The most basic feature which I provide is syntax-based colour highlighting. Syntax-based colouring is present once a programmer opens a file whose name ends with "`.as`" (the ALDOR format). In other words, as soon as the programmer opens an ALDOR file the whole buffer is coloured. Additionally, when programmers hover their mouse on top of a token string, the echo area displays the token type for the token. Therefore, the programmer is able to identify the type of any tokens in the buffer.

Furthermore, I implemented a "print parse tree" feature for ALDOR mode in EMACS. This feature asks a programmer to select an ALDOR file. Once a programmer chooses a file, the EMACS text editor pops open a new frame and displays the tree in that new frame. This feature allows programmers to understand how an ALDOR program has been parsed and to see the parse tree form of the program. The shortcut key for the ALDOR tree printing feature is set as (`M-RET`).

Moreover, as mentioned in Chapter 4, there is a "parsing" feature for programmers who wish to parse their ALDOR program and get extra information. At this stage, the parser only inserts overlays to ALDOR program buffers. However, it has the ability to be modified to have more powerful features. The shortcut key for the parsing feature is (`M-p`).

The last feature that I provide for ALDOR programmers is a comment region feature. For instance, users can highlight any region in a buffer, to comment it out. This feature is associate with the shortcut key (`M-c`). In this way, the users will be able to comment out any section of code they wish to leave out. However, if a user

plans to re-implement the comment function, `newcomment.el` code in EMACS will be a good reference to study. It provides more general functions to do the same.

In summary, my ALDOR mode is similar to other existing EMACS language modes in that it provides the syntax-based colouring of tokens and allows the programmer to comment out a region. However, to print a parse tree of a program and to echo the token type selected by the cursor are new features.

5.5 Efficiency

The ALDOR mode for EMACS executes very efficiently. As my lexical analyzer was implemented carefully; the time and space efficiency is very good. Therefore, the overall efficiency for both my internal and external versions in the `aldor-mode` is acceptable.

The parser does not have good time efficiency. However, my parser is still able to return results within a reasonable amount of time. As for space efficiency, a tree node parser takes more space than an update "on the fly" parser. However, the intermediate buffers for both parsers are as big as a tree, since they have to store all nodes. For this reason, the space efficiency for my parsers are not very good.

The ALDOR print tree feature is similar to a parsing feature. In order for a parser to draw a complete ALDOR program tree, it requires full parsing of the source buffer. Hence, the time efficiency is similar to the ALDOR parser. However, results of tree printing do help many programmers.

Finally, the "`aldor-comment-region`" feature is very efficient. It runs relatively fast and it does not take up any memory space. Moreover, this feature can help ALDOR programmers in many different ways.

5.6 Problems Encountered and their Solutions

The major challenge was to learn methods which allow users to add a new programming language mode into EMACS. The EMACS syntax table for a mode is difficult to understand. However, after I did research on EMACS language modes and I received help from my supervisor, I was able to resolve the problems I encountered.

The insertion of hooks was very confusing. However, after studying the manual and doing some experiments on hooks, it seemed to be much simpler and easier to understand. In conclusion, to put all of the components together and create an ALDOR mode for EMACS text editor is not really a very complicated job. The EMACS-LISP manual is very well documented. For that reason, almost all of the problems can be solved by reading through the manual then performing some experiments.

5.7 Conclusion and Summary

The `aldor-mode` was one of the easier exercises in my research. The main focus of this portion was to merge all of the components together and to add the `aldor-mode` into EMACS.

In conclusion, `aldor-mode` for EMACS connects the work I completed. Programmers are able to see each of the components. Furthermore, it gives people a chance to think through the purpose of each component in the ALDOR mode. My research process was very long and I waited a long time to put all of the pieces together. My work will help others to gain knowledge about programming language modes. Eventually, they too will be able to add a new language mode to EMACS.

The ALDOR mode packages together the ALDOR lexical analyzers, parser, and other tools. It integrates the programming that I did. In the next chapter I summarize the work done in the entire thesis.

Chapter 6

Conclusions and Summary

I implemented an internal lexical analyzer, an external lexical analyzer, two versions of a parser, and bound them together with an ALDOR mode. In this chapter, I briefly review the whole thesis and state my conclusions.

An internal ALDOR lexical analyzer was implemented in EMACS-LISP. This version of lexical analyzer runs relative quickly and is able to update a buffer almost instantly. However, the codes for an internal ALDOR lexical analyzer is not re-usable for other programming languages.

An external ALDOR lexical analyzer was implemented in FLEX and C. The main challenge of implementing this version of the lexical analyzer was communication. The EMACS text editor and my external lexical analyzer communicate through pipes and an intermediate buffer. Although, an external lexical analyzer does not return results as fast as an internal lexical analyzer, the code can be modified and re-used for other programming languages.

The lexical analyzers put text properties into the buffer, and perform a syntax-based colouring feature. Moreover, I implemented an external lexical analyzer to provide all information required by my external ALDOR parser.

The ALDOR parsers were implemented in BISON and FLEX. My external AL-

DOR parsers create overlays and can be used to implement many powerful features. Additionally, a tree node parser generates parse trees for ALDOR programs with node information and binding. However, my parsers will only support full buffer parsing. Currently the parsers do not support partial parsing.

6.1 Summary of my Research Timeline

Internal lexical analyzers were the starting point of my research process. During the development time for my internal ALDOR lexical analyzers, I learned a lot about EMACS. Since I was working on both versions of the lexical analyzers, I was able to compare the two versions side by side. However, most of the scanning algorithms for external lexical analyzers could not be applied to internal lexical analyzers.

By the time I completed part of my internal lexical analyzer, I had gained enough knowledge to implement the external lexical analyzer for EMACS. Nevertheless, the main challenge was the communication problems between programs. At this stage, I took a lot of time to understand and learn the concepts for communication between programs.

After I completed both versions of the lexical analyzer, I started to work on my parser. The research I did gave me several different choices of algorithms to parse a program and create a parsing tree. Fortunately, I had done the external lexical analyzer already. Thus, I did not have a very hard time debugging communication problems between `Bison` and EMACS. Nevertheless, I encountered communication problems between `Flex` and `Bison`. Additionally, the ALDOR grammar rules which I had in the beginning were ambiguous to `Bison`. For this reason, I had to try to resolve the grammar which `Bison` determined as ambiguous before being able to continue in my work. I managed to get it done before I finished my ALDOR mode for EMACS. For this reason, I am able to put the parsing features and print tree

features into the `aldor-mode`.

Lastly, I wrapped up my entire work with an ALDOR mode for EMACS. The development process for `aldor-mode` was not too rough. As a result, I had enough time to add in some extra features such as the `aldor-comment-region`, Parse this buffer and `aldor-print-tree` functions.

6.2 What can People Learn from my Work?

As I have finished this research on EMACS, people can now learn from the examples which I provide, and can learn how to add a new language mode to EMACS. At least, they will not need to dig into the details about EMACS and EMACS-LISP. My goal is to develop a system not only for ALDOR, but also for other languages. For instance, other programmers should be able to add a new programming language mode once they know the grammar and tokenizing rules for a programming language. They can use my work as a template to develop a new language mode in EMACS. People can also learn about how to separate syntax analysis from editing. I have also shown that it is possible for EMACS to communicate with external resources and process the returned information.

6.3 Comparison of Lexical Analyzers

It is very interesting to compare the internal and the external versions of the lexical analyzers. The results of my experiments match with what I had originally thought. Before I worked on this thesis, I suspected that an external approach would work slower but better than the internal version. As the results show, my internal lexical analyzer works faster than the external lexical analyzer. The internal lexical analyzer has speed and space advantages. However, the external version can work concurrently via pipes, rather than in a batch mode via files. Additionally, the ex-

ternal lexical analyzer is expandable and re-usable. The major factor which slows down the external version is program waiting time. As with what I learned from my parser, sometimes a program has to sacrifice its speed in exchange for more utility.

Nonetheless, an external lexical analyzer is more useful than an internal version. Whereas, the internal version only works inside EMACS, an external version has the ability to output in any format and provide input to programs other than EMACS. For this reason, an external lexical analyzer has more flexibility than the internal version. Therefore, the experimental external lexical analyzer turned out to be very acceptable. Basically, it has very reasonable time and memory efficiency. The timing of the lexical analyzers has been measured and plotted, and is displayed in Chapter 3 of this thesis. In conclusion, internal lexical analyzers process faster than external lexical analyzers but an external lexical analyzer is more useful than an internal version.

6.4 Further Improvement and Future Work

At the current stage, I established basic methods for interaction between EMACS and external programs. However, the `aldor-mode` in EMACS can still improve a lot. For example, the parser provides a lot of information which can be used to improve both the performance and features of the `aldor-mode`.

The update scheme for both lexical analyzer and parser can be improved as well. For instance, have Emacs update the modified buffer after a constant amount of time. Moreover, have the parser reparse a section of buffer instead of the whole buffer. One could also implement an automatic reparse feature for the mode and an automatic syntax recovery or suggesting system.

As for future work, I am planning to develop and experiment with some new algorithms that will enable the computer to better interact with the outside world.

Bibliography

[1] Nathaniel S. Borenstein and James Gosling. Unix Emacs: a retrospective (lessons for flexible system design). In *UIST '88: Proceedings of the 1st annual ACM SIGGRAPH symposium on User Interface Software*, pages 95–101, New York, NY, USA, 1988. ACM Press.

[2] Peter Broadbery and Manuel Bronstein. A first course on Aldor with libaldor. available from `www.aldor.org`, March 2002.

[3] Manuel Bronstein. *libaldor User Guide and Reference Manual.* available from `www.aldor.org`, March 2002.

[4] Debra Cameron, Bill Rosenblatt, and Eric Raymond. *Learning GNU Emacs.* O'Reilly, 1996.

[5] Charles Donnelly and Richard Stallman. *Bison.* Free Software Foundation, December 2004.

[6] V Donzeau-Gouge, B. Lang, and B. Mélèse. Practical applications of a syntax directed program manipulation environment. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 346–354, Piscataway, NJ, USA, 1984. IEEE Press.

[7] Martin N. Dunstan. Aldor compiler internals II: Code generation and optimisation. available from `www.aldor.org`, September 2000.

[8] Bryan Ford. Packrat parsing:: simple, powerful, lazy, linear time, functional pearl. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 36–47, New York, NY, USA, 2002. ACM Press.

[9] Free Software Foundation. *Bison 1.875 Manual*. Free Software Foundation, November 2002.

[10] Free Software Foundation. *GNU Emacs Lisp Reference Manual, For Emacs Version 21, Revision 2.8*. Free Software Foundation, 2002.

[11] Christopher Fry. Programming on an already full brain. *Commun. ACM*, 40(4):55–64, 1997.

[12] Greg Harvey. What is Emacs? available from `www.gnu.org/software/emacs/emacs.html`, 2003.

[13] Ralf Hemmecke. An Emacs mode for Aldor. available from `www.hemmecke.de/aldor/aldor-mode.pdf`, November 2005.

[14] Michael D. Hutton. Noncanonical extensions of lr parsing methods. available from `www.eecg.toronto.edu/~mdhutton/papers/noncan.pdf`, 1990.

[15] A D Kennedy. Semantics of categories in Aldor. available from `www.ph.ed.ac.uk/~bj/paraldor/WWW/docs/discussion/define.pdf` , 2001.

[16] John Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly, 1992.

[17] Vern Paxson. *Flex Manual, version 2.5.* Free Software Foundation, 1995.

[18] Erik Poll and Simon Thompson. The Type System of Aldor. Technical Report 11-99, Computing Laboratory, University of Kent at Canterbury, July 1999.

[19] Roger C. Schank. Language and memory. Technical report, Yale University, 1990.

[20] Richard M. Stallman. *GNU Emacs Manual.* Free Software Foundation, 2002.

[21] Keith Waclean. A tutorial introduction to GNU Emacs. available from `http://www.lib.uchicago.edu/keith/tcl-course/emacs-tutorial.html`, 1997.

[22] Stephen M. Watt. Aldor III. available from `www.aldor.org`, September 2000.

[23] Stephen M. Watt. Aldor: Implementation I. available from `www.aldor.org`, September 2000.

[24] Stephen M. Watt. Aldor: Interfaces. available from `www.aldor.org`, September 2000.

[25] Stephen M. Watt. Aldor: The language and recent directions. available from `www.aldor.org`, June 2000.

[26] Stephen M. Watt. Aldor: An introduction to the language. available from `www.aldor.org`, September 2002.

[27] Stephen M. Watt. *Aldor User Guide.* The Numerical Algorithms Group Limited, 2002.

[28] LALR parser. http://en.wikipedia.org/wiki/LALR_parser. in Wikipedia, the free encyclopedia.

[29] Saul Youssef. Prospects for category theory in Aldor. available from `http://atlas.bu.edu/~youssef/papers/math/aldor/aldor.pdf`, 2001.

Appendix A

A Lexical Structure for Aldor

A.1 Lexical Structure

The following information is referenced from the ALDOR User Guide [27].

A.1.1 Characters

The standard ALDOR character set contains the following 97 characters:

- the blank, tab and newline characters
- the Roman letters: `a-z A-Z`
- the digits: `0-9`
- and the following special characters:

`(`	left parenthesis	`)`	right parenthesis
`[`	left bracket	`]`	right bracket
`{`	left brace	`}`	right brace
`<`	less than	`>`	greater than
`,`	comma	`.`	period

;	semicolon	:	colon
?	question mark	!	exclamation mark
=	equals	_	underscore
+	plus	-	minus (hyphen)
&	ampersand	*	asterisk
/	slash	\	back-slash
'	apostrophe (quote)	`	grave (back-quote)
"	double quote	|	vertical bar
^	circumflex	~	tilde
@	commercial at	#	sharp
$	dollar	%	percent

Other characters may appear in source programs, but only in comments and string-style literals. Blank, tab and newline are called white space characters. All the special characters except quote, grave and ampersand are required for use in tokens. Grave and ampersand are reserved for future use.

A.1.2 The Escape Character

Underscore is used as an escape character, which alters the meaning of the following text. The nature of the change depends on the context in which the underscore appears. An escaped underscore is not an escape character. An escape character followed by one or more white space characters causes the white space to be ignored. The remainder of this section assumes that escaped white space has been removed from the source.

A.1.3 Tokens

The sequence of source characters is partitioned into *tokens*. The longest possible match is always used.

The tokens are classified as follows:

- the following language-defined keywords:

```
add       and     always   assert  break
but       catch   default  define  delay
do        else    except   export  extend
fix       for     fluid    free    from
generate  goto    has      if      import
in        inline  is       isnt    iterate
let       local   macro    never   not
of        or      pretend  ref     repeat
return    rule    select   then    throw
to        try     where    while   with
yield

.   ,   ;    :   ::  :*   $  @
|   =>  +->  :=  ==  ==>  '
[   ]   {    }   (   )
```

  The characters in a keyword cannot be escaped. That is, if a character is escaped, the token is not treated as a keyword.

- the following are not defined by the language but are reserved words for future use:

```
delay  fix  is  isnt  let  rule
(|  |)  [|  |]  {|  |}  `  &  ||
```

- the following set of definable *operators*:

```
by  case  mod  quo  rem

#   +   -   +-  ~   ^
*   **  ..  =   ~=  ^=
/   \   /\  \/  <   >
<=  >=  <<  >>  <-  ->
```

  The characters in an operator cannot be escaped.

- *identifiers*:

```
0
1
[%a-zA-Z][%?!a-zA-Z0-9]*
```

Any non-white space standard character may be included in an identifier by escaping it. Thus “`a`”, “`_*`”, “`a_*`” and “`_if`” are all identifiers. The escape character is not part of the identifier so “`ab`” “`_a_b`” represent the same identifier. Identifiers are the only tokens for which the leading character may be escaped.

- *string-style literals*:

  ```
  ‘"’[^"]*‘"’
  ```

  An underscore or double quote may be included in a string-style literal by escaping it.

- *integer-style literals*:

  ```
  [2-9]
  [0-9][0-9]+
  [0-9]+‘r’[0-9A-Z]+
  ```

  Escape characters are ignored in integer-style literals and so may be used to group digits.

- *Floating point-style literals*:

  ```
  [0-9]*‘.’[0-9]+{[eE]{[+-]}[0-9]+}
  [0-9]+‘.’[0-9]*{[eE]{[+-]}[0-9]+}
  [0-9]+[eE]{[+-]}[0-9]+
  [0-9]+‘r’[0-9A-Z]*‘.’[0-9A-Z]+{e{[+-]}[0-9]+}
  [0-9]+‘r’[0-9A-Z]+‘.’[0-9A-Z]*{e{[+-]}[0-9]+}
  [0-9]+‘r’[0-9A-Z]+‘e’{[+-]}[0-9]+
  ```

  Escape characters are ignored in floating point-style literals and so may be used to group digits.

  Certain lexical contexts restrict the form of floats allowed. This distinguishes cases such as `sin 1.2` *vs* `m.1.2`. A floating point literal may not

  1. begin with “`.`”, unless the preceding token is a keyword other than “`)`”, “`|)`”, “`]`” or “`}`”;
  2. contain “`.`”, if the preceding token is “`.`”;
  3. end with “`.`”, if the following character is “`.`”.

- *comments*:

  The two characters “`--`” and all characters up to the end of the line. Underscores are not treated as escape characters in comments.

- *documentation*:

  The two characters "++" and all characters up to the end of the line. Underscores are not treated as escape characters in documentation.

- *leading white space*:

  a sequence of blanks or tabs at the beginning of a line.

- *embedded white space*:

  a sequence of blanks or tabs not at the beginning of a line.

- *newline*:

  a new line character.

- *layout markers*:

  `SETTAB  BACKSET  BACKTAB`

  These do not appear in a source program but may be used to represent a linearized from of the token sequence.

Comments and embedded white space are always ignored, except as used to separate tokens. For example, "`abc`" is taken as one token but "`a b c`" is taken as three.

A.2 Differences Between the Implemented Lexical Scanner and the Aldor Lexical Structure

There are some differences between my version of the tokens and the tokens from the ALDOR official web site. One of the major differences is that my external lexical analyzer does not handle an "`_`" (underscore) character. In the ALDOR official tokens list, "`_`" acts as an escape symbol. In other words, the "`_`" character can be ignored. For example, an ALDOR program reads the following as exactly the same token string: "`apple`", "`_apple`", "`a_pp_l_e`", etc.. However, in my program, an "`_`" symbol does not get implemented, so, it is not handled properly. The major reason I did not implemented it is due to the difficulty of the word counter. This symbol will mess up my work count variable, and cause the buffer not colour properly. For the reason, I had not implement the "_" identification in a string.

Finally, the last difference is the layout markers. Although the type of token does exist in ALDOR official tokens, it does not exist in my program. These tokens are created by complicated rules that track levels of indentation that I chose not to implemented. However, they may be included in the future.

Appendix B

Context Free Grammars for Aldor

```
/* RESERVE WORDS  */
%token ADD ALWAYS AND ASSERT BREAK BUT CATCH DEFAULT DEFINE DELAY DO
       ELSE EXCEPT EXPORT EXTEND FIX FLUID FOR FREE FROM GENERATE GOTO
       HAS IF IMPORT IN INLINE IS ISNT ITERATE LET LOCAL MACRO NEVER
       NOT OF OR PRETEND REF REPEAT RETURN RULE SELECT THEN THROW TO TRY
       WHERE WHILE WITH YIELD BY CASE FINALLY MOD QUO REM

/* SYMBOLS - DEFINEABLE */
%token SETMINUS POUND TIMES PLUS MINUS DIVIDE LESSTHAN EQUALS GREATERTHAN
       ATSIGN TILDE CARET VERY_LESS VERY_GREATER LESSEQ GREATEREQ GETS
       TO_SYMBOL JOIN MEET STARSTAR DOTDOT UNEQUALS CARETEQUALS

/* SYMBOLS - RESERVED OPS */
%token DOLLAR SQUOTE LPAREN RPAREN FUN_ARROW COMMA COLON IS_ASSIGNED
       SEMI MACRO_IS EXIT LBRACK RBRACK LBRACE RBRACE VERY_EQUAL BAR

/* SYMBOLS - FUTURE OPS */
%token LLBRACK LLBRACE LLPAREN RRBRACK RRBRACE RRPAREN GRAVE
       AMPERSAND OROR

/* OTHERS */
%token INT ID FLOAT PREDOC POSTDOC STRING
       COLON_STAR DCOLON DOT PLUSMINUS BACKSET BACKTAB SETTAB

%%
start:    curlyContents_Labelled
```

```
expression:   labelled
            |  expression SEMI labelled
            |  expression SEMI

labelled  : comma
          | declaration
          | ATSIGN atom labelled
          | ATSIGN atom

declaration :
            MACRO sig
          | EXTEND sig
          | LOCAL sig
          | FLUID sig
          | DEFAULT sig
          | DEFINE sig
          | FIX sig
          | INLINE sigOp fromPartOpt
          | IMPORT sigOp fromPartOpt
          | EXPORT sigOp
          | EXPORT sigOp toPart
          | EXPORT sigOp fromPart

toPart:
            TO infixed

fromPartOpt:
            fromPart
          | /* empty */

fromPart:
            FROM infixedCL

infixedCL:
            infixed
          | infixedCL COMMA infixed

sigOp:
            sig
          | /* empty */

sig:
```

```
            declBinding
          | block

declPart:
            COLON type
          | COLON_STAR type

comma:
            commaItem
          | commaItem COMMA comma

commaItem:
            binding_AS
          | binding_AS WHERE commaItem


declBinding:
            bindingR_IED_AS

infixedExprsDecl:
            infixedExprs
          | infixedExprs declPart

infixedExprs:
            infixedExpr
          | infixedExpr COMMA infixedExprs

dgc_Binding_Symbol:
            IS_ASSIGNED
          | VERY_EQUAL
          | MACRO_IS
          | FUN_ARROW

bindingR_IED_AS:
            infixedExprsDecl
          | infixedExprsDecl dgc_Binding_Symbol binding_AS

binding_AS:
            bindingL_Infixed_AS

binding_BS:
            bindingL_Infixed_BS
```

```
binding_Collection:
            bindingL_Infixed_Collection

bindingL_Infixed_AS:
            anyStatement
          | infixed dgc_Binding_Symbol binding_AS

bindingL_Infixed_Collection:
            collection
          | infixed dgc_Binding_Symbol binding_Collection

bindingL_Infixed_BS:
            balStatement
          | infixed dgc_Binding_Symbol binding_BS

anyStatement:
            IF commaItem THEN binding_AS
          | flow_as

balStatement:
            flow_bs

flow_as:
            collection
          | IF commaItem
            THEN binding_BS
            ELSE binding_AS
          | collection EXIT binding_AS
          | iterators REPEAT binding_AS
          | TRY binding_AS CATCH casesOpt finalPart_AS
          | TRY binding_AS CATCH casesOpt alwaysPart_AS
          | SELECT binding_AS IN cases
          | DO binding_AS
          | DELAY binding_AS
          | GENERATE genBound binding_AS
          | ASSERT binding_AS
          | ITERATE
          | ITERATE name
          | BREAK
          | BREAK name
          | RETURN
          | RETURN collection
          | YIELD binding_AS
```

```
          | EXCEPT binding_AS
          | GOTO id
          | NEVER

flow_bs   : collection
          | IF commaItem
            THEN binding_BS
            ELSE binding_BS
          | collection EXIT binding_BS
          | iterators REPEAT binding_BS
          | TRY binding_AS CATCH casesOpt finalPart_BS
          | TRY binding_AS CATCH casesOpt alwaysPart_BS
          | SELECT binding_AS IN cases
          | DO binding_BS
          | DELAY binding_BS
          | GENERATE genBound binding_BS
          | ASSERT binding_BS
          | ITERATE
          | ITERATE name
          | BREAK
          | BREAK name
          | RETURN
          | RETURN collection
          | YIELD binding_BS
          | EXCEPT binding_BS
          | GOTO id
          | NEVER

genBound:
            TO commaItem OF
          | /* empty */

cases:
            binding_Collection

casesOpt:
            binding_Collection
          | /* empty */

alwaysPart_AS:
            ALWAYS binding_AS

alwaysPart_BS:
```

```
            ALWAYS binding_BS


finalPart_BS:
            FINALLY binding_BS
          | /* empty */

finalPart_AS:
            FINALLY binding_AS
          | /* empty */


collection:
            infixed
          | infixed iterators

iterators:
            iterators1

iterators1:
            iterator
          | iterators1 iterator

iterator:
            FOR forLHS IN infixed
          | FOR forLHS IN infixed suchThatPart
          | WHILE infixed

forLHS:
            infixed
          | FREE infixed
          | LOCAL infixed
          | FLUID infixed

suchThatPart:
            BAR infixed

infixed:
            infixedExpr
          | infixedExpr declPart
          | block

infixedExpr:
```

```
            e11_OP
          | e3

e3:
            e4
          | e3 AND e4
          | e3 OR e4
          | e3 latticeOp e4

e4:
            e5
          | e4 HAS e5
          | e4 relationOp e5
          | relationOp e5

e5:
            e6
          | e5 seqOp
          | e5 seqOp e6

e6:
            e7
          | e6 plusOp e7
          | plusOp e7

e7:
            e8
          | e7 quotientOp e8

e8:
            e9
          | e8 timesOp e9

e9:
            e11_e12
          | e11_e12 powerOp e9

e11_OP:
            op
          | e11_OP DCOLON e12
          | e11_OP ATSIGN e12
          | e11_OP PRETEND e12
```

```
e11_e12:
            e12
          | e11_e12 DCOLON e12
          | e11_e12 ATSIGN e12
          | e11_e12 PRETEND e12

type:
            e11_e12

e12:
            e13
          | e13 arrowOp e12

e13:
            e14
          | e14 DOLLAR qualTail

qualTail:
            leftJuxtaposed
          | leftJuxtaposed DOLLAR qualTail

opQualTail:
            molecule
          | molecule DOLLAR opQualTail

e14:
            e15
          | e14 EXCEPT e15
          | WITH declMolecule
          | e14 WITH declMolecule
          | ADD declMolecule
          | e14 ADD declMolecule

e15:
            application

op:
            arrowOp
          | latticeOp
          | relationOp
          | seqOp
          | plusOp
          | quotientOp
```

```
          | timesOp
          | powerOp

nakedOp:
            arrowTok
          | latticeTok
          | relationTok
          | seqTok
          | plusTok
          | quotientTok
          | timesTok
          | powerTok

arrowOp:
            arrowTok
          | arrowTok DOLLAR opQualTail

latticeOp:
            latticeTok
          | latticeTok DOLLAR opQualTail

relationOp:
            relationTok
          | relationTok DOLLAR opQualTail

seqOp:
            seqTok
          | seqTok DOLLAR opQualTail

plusOp:
            plusTok
          | plusTok DOLLAR opQualTail

quotientOp:
            quotientTok
          | quotientTok DOLLAR opQualTail

timesOp:
            timesTok
          | timesTok DOLLAR opQualTail

powerOp:
            powerTok
```

```
          | powerTok DOLLAR opQualTail

arrowTok:
            GETS
          | TO_SYMBOL

latticeTok:
            JOIN
          | MEET

relationTok:
            ISNT
          | IS
          | CASE
          | VERY_LESS
          | VERY_GREATER
          | LESSTHAN
          | LESSEQ
          | EQUALS
          | GREATEREQ
          | GREATERTHAN
          | UNEQUALS
          | CARETEQUALS

seqTok:
            BY
          | DOTDOT

plusTok:
            PLUS
          | MINUS
          | PLUSMINUS

quotientTok:
            MOD
          | QUO
          | REM

timesTok:
            TIMES
          | DIVIDE
          | SETMINUS
```

```
powerTok:
            CARET
          | STARSTAR

application:
            rightJuxtaposed

rightJuxtaposed:
            jRight_Mol

leftJuxtaposed:
            jLeft_Mol

jRight_Mol:
            jLeft_Mol
          | jLeft_Mol jRight_Atom
          | NOT jRight_Atom

jRight_Atom:
            jLeft_Atom
          | jLeft_Atom jRight_Atom
          | NOT jRight_Atom

jLeft_Mol :
            molecule
          | NOT blockEnclosure
          | jLeft_Mol blockEnclosure
          | jLeft_Mol DOT blockMolecule

jLeft_Atom:
            atom
          | NOT blockEnclosure
          | jLeft_Atom blockEnclosure
          | jLeft_Atom DOT blockMolecule

molecule  :
            atom
          | enclosure

enclosure :
            parened
          | bracketed
          | quotedIds
```

```
declMolecule:
            application
          | block
          | /* empty */

blockMolecule:
            atom
          | enclosure
          | block

blockEnclosure:
            enclosure
          | block

block:
            piled_Expression
          | curly_Labelled

parened:
            LPAREN RPAREN
          | LPAREN expression RPAREN

bracketed:
            LBRACK RBRACK
          | LBRACK expression RBRACK

quotedIds:
            SQUOTE SQUOTE
          | SQUOTE names SQUOTE

names:
            name
          | name COMMA names

atom:
            id
          | literal

id:
            ID
          | POUND
          | TILDE
```

```
name:
            ID
          | nakedOp

literal:
              INT
          |   FLOAT
          |   STRING

doc_Expression:
            preDocument expression postDocument


preDocument:
            preDocumentList

preDocumentList:
            PREDOC preDocumentList
          | /* empty */

postDocument:
            postDocumentList

postDocumentList:
            POSTDOC postDocumentList
          | /* empty */

piled_Expression:
            SETTAB pileContents_Expression BACKTAB

pileContents_Expression:
            doc_Expression
          | pileContents_Expression BACKSET doc_Expression

curly_Labelled:
            LBRACE
            curlyContents_Labelled
            RBRACE

curlyContents_Labelled:
            curlyContentsList_Labelled
```

```
curlyContentsList_Labelled:
            curlyContent1_Labelled
          | curlyContent1_Labelled curlyContentB_Labelled %prec SEMI

curlyContent1_Labelled:
            curlyContent1_Labelled curlyContentA_Labelled
          | /* empty */

curlyContentA_Labelled:
            curlyContentB_Labelled SEMI postDocument

curlyContentB_Labelled:
            preDocument labelled postDocument
```

Appendix C

Abstract Syntax for Aldor

```
decl_type:
            MACRO  | FLUID | EXTEND | DEFAULT | LOCAL
          | INLINE | FREE  | IMPORT | EXPORT

binding_symbol:
            IS_ASSIGNED | VERY_EQUAL | MACRO_IS | FUN_ARROW

naked_operator:
            GETS        | TO_SYMBOL | CARETEQUALS | JOIN      | CARET
          | MEET        | BY        | ISNT        | DOTDOT    | IS
          | PLUS        | CASE      | MINUS       | VERY_LESS
          | PLUSMINUS   | VERY_GREATER            | MOD       | LESSTHAN
          | QUO         | LESSEQ    | REM         | EQUALS    | TIMES
          | GREATERTHAN | DIVIDE    | GREATEREQ   | SETMINUS
          | UNEQUALS    | STARSTAR

literal :
            INTEGER_LITERAL | FLOATING_POINT_LITERAL | STRING_LITERAL

name :
            IDENTIFIER | naked_operator

names :
            name | names COMMA name

atom :
```

```
              literal | IDENTIFIER

molecule :
              atom
            | LPAREN RPAREN
            | LPAREN expression RPAREN
            | LBRACK RBRACK
            | LBRACK expression RBRACK
            | SQUOTE SQUOTE
            | SQUOTE names SQUOTE
            | block

block:
              LBRACE expression RBRACE

pre_doc_list :
              <empty> | PREDOC pre_doc_list

post_doc_list :
              <empty> | POSTDOC post_doc_list

expression :
              expression COMMA expression
            | expression SEMI expression
            | expression SEMI
            | pre_doc_list expression post_doc_list
            | atom ATSIGN expression
            | atom ATSIGN
            | declaration
            | comma_item

comma_item :
              binding
            | binding WHERE comma_item

binding :
              statement
            | infixed binding_symbol statement

sig_block :
              signature
            | block
```

```
declaration :
              decl_type sig_block
            | decl_type sig_block FROM infixed
            | decl_type sig_block TO infixed
            | decl_type sig_block FROM infixed TO infixed

signature :
              infixed
            | infixed binding_symbol binding

collection :
              infixed
            | infixed iterator_list

case :
              collection
            | infixed binding_symbol collection

statement :
              IF comma_item THEN binding
            | IF comma_item THEN binding ELSE binding
            | collection
            | collection EXIT binding
            | iterator_list REPEAT binding
            | TRY binding CATCH case
            | TRY binding FINALLY binding
            | TRY binding CATCH case FINALLY binding
            | SELECT binding OF case
            | ASSERT binding
            | ITERATE
            | ITERATE IDENTIFIER
            | BREAK
            | BREAK IDENTIFIER
            | RETURN
            | RETURN collection
            | YIELD binding
            | EXCEPT binding
            | GOTO IDENTIFIER
            | GENERATE binding
            | GENERATE TO comma_item OF binding
            | NEVER

iterator_list :
```

```
             iterator
           | iterator_list COMMA iterator

forLHS_KW :
             <empty> | FREE | FLUID | LOCAL

iterator :
             FOR forLHS_KW infixed IN infixed
           | FOR forLHS_KW infixed IN infixed "|" infixed
           | WHILE infixed

infixed :
             infixed_expr
           | infixed COMMA infixed_expr
           | infixed_expr COLON infixed_expr
           | infixed_expr COLON_STAR infixed_expr
           | block

infixed_expr :
             ADD infixed_expr
           | NOT infixed_expr
           | WITH infixed_expr
           | operator infixed_expr
           | infixed_expr operator
           | infixed_expr infixed_expr
           | infixed_expr ADD infixed_expr
           | infixed_expr AND infixed_expr
           | infixed_expr ARROW infixed_expr
           | infixed_expr AT infixed_expr
           | infixed_expr DCOLON infixed_expr
           | infixed_expr DOLLAR infixed_expr
           | infixed_expr DOT infixed_expr
           | infixed_expr EXCEPT infixed_expr
           | infixed_expr HAS infixed_expr
           | infixed_expr OR infixed_expr
           | infixed_expr PRETEND infixed_expr
           | infixed_expr WITH infixed_expr
           | infixed_expr operator infixed_expr
           | molecule
           | operator

operator :
             naked_operator
```

```
            | naked_operator DOLLAR opQualTail

opQualTail :
              molecule
            | molecule DOLLAR opQualTail
```

Appendix D

Codes for my Work

Instead of insert the source codes in this section, it will make more sense to put them on my web site. Hence, others will be able to download, modify, and execute the codes. Additionally, it will require around one hundred pages to print the codes. All of my codes will be upload to my UNBC web site at the following address:

`http://web.unbc.ca/~hsiehy`

Appendix E

UML Diagram of External Lexical Analyzer

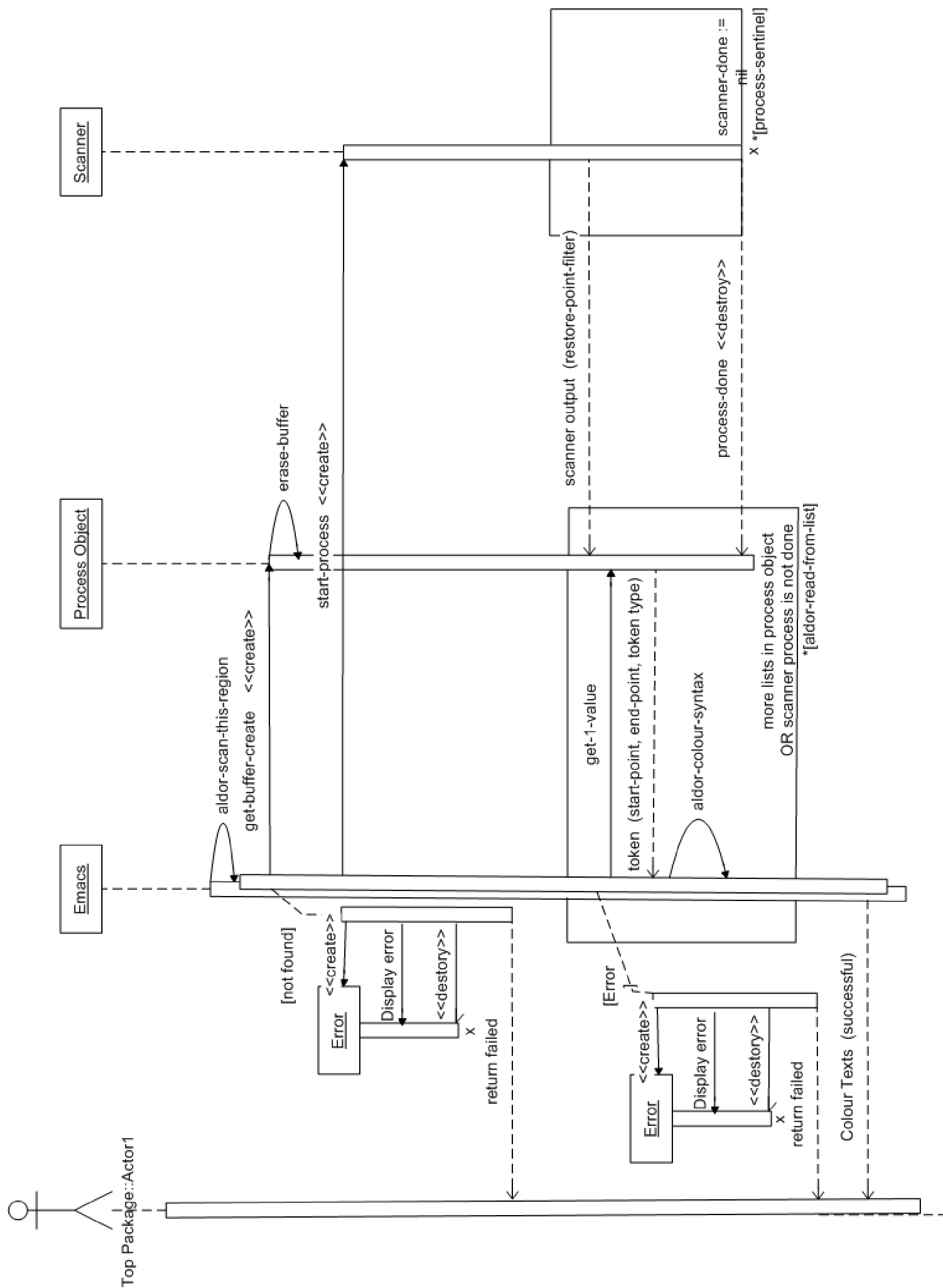

Figure E.1: UML Diagram for the External Lexical Analyzer

Appendix F

UML Diagram of Parser

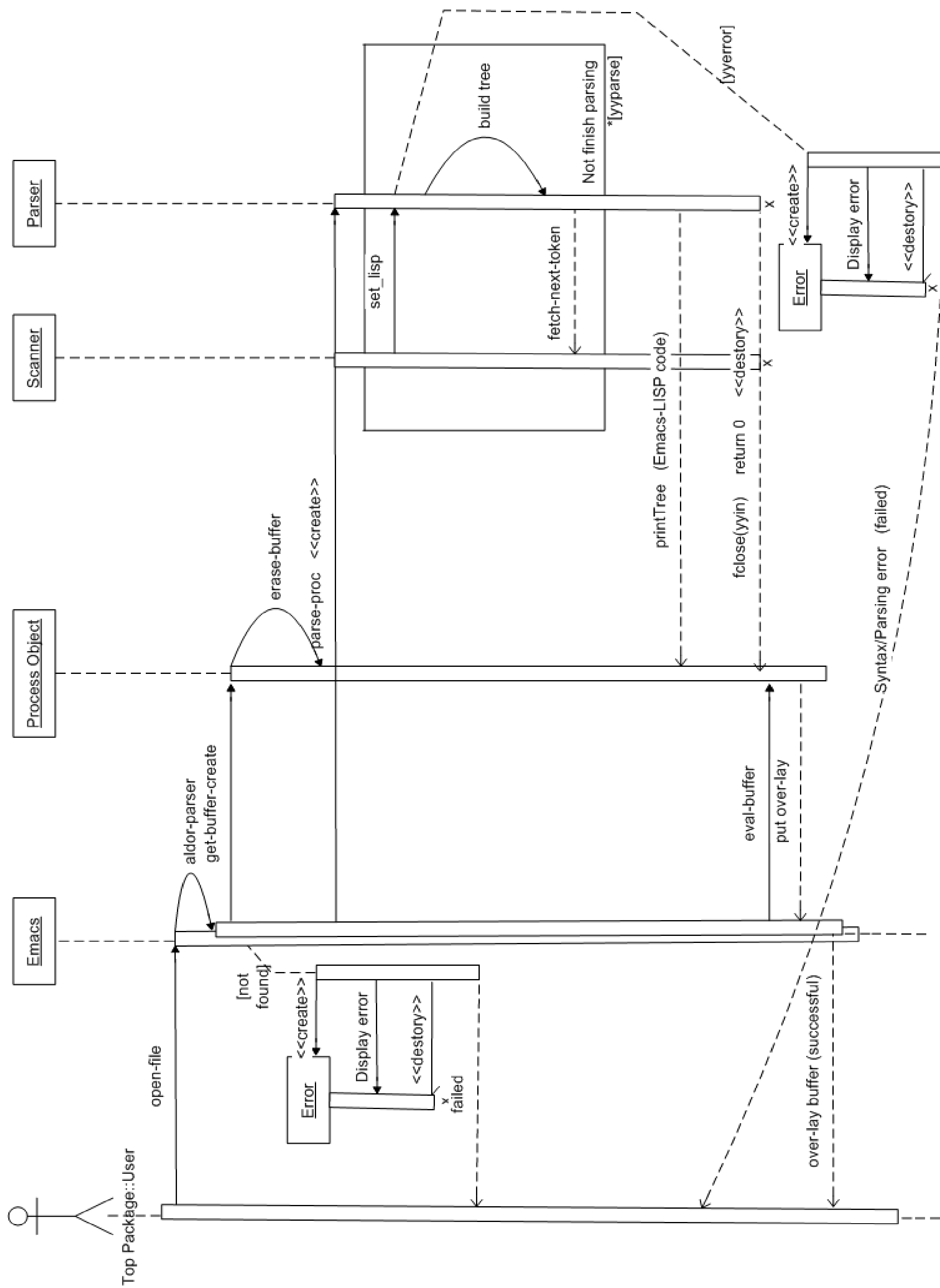

Figure F.1: UML Diagram for the Parser