CPSC320

Midterm Review

Robert Pringle

October 5, 2007

Language and grammar review.

- A language over an alphabet A is defined as a subset of A^*.
- A formal grammar is used to generate, for a generative grammar, or recognize, for an analytical grammar, strings contained in the language it represents.
- Grammars are usually defined by a 4-tuple containing a set of terminal symbols, a set of non-terminal symbols, a set of productions and the set containing the starting productions for the grammar.

Scanning, Parsing and Semantics

- Scanning is concerned mainly with the recognition of lexemes within a given source string.
- Parsing is concerned with the syntactic evaluation of these recognized lexemes to ensure they fit with the syntax of the language the parser represents.
- Semantics relate to the meaning of a particular source string. Though we can determine whether the source string confirms to the syntax of our host language we cannot usually fully determine the meaning of the source (or indeed in some cases whether it even makes any sense).

Scanning

- When scanning for lexemes in a source string we may come across the situation where it is possible to form multiple tokens with a liberal interpretation of the grammar the scanner represents.
- Scanners usually use the longest substring principle when attempting to form tokens. With this the token formed at the a given point in the source will be the longest token that could be formed at that point.
- Programming language tokens can be broken down into a number of categories, these include the following:
  - Reserved/Key Words
  - Constants or Literals
  - Special Symbols
  - Identifiers

Scanning Cont.

- In languages, such as C or Java, where whitespace characters hold no meaning other than as token delimeters we say the language is free-format while in languages where this is not the case, such as FORTRAN, we say the language is fixed-format.

Chomsky Hierarchy Grammar Restrictions.

Grammar Type	Production Forms	Restriction Description
Regular	$N \rightarrow NT^*$, $N \rightarrow T^*N$ or $N \rightarrow T^*$	Regular grammars use productions that only have one non-terminal on the right hand side of the production which is either the first or last symbol for right hand side of the production.
Context-free	$N \rightarrow (N^*T^*)^*$	Context-free grammars restrict the left-hand side of the productions to a single non-terminal symbol. Any combination of terminals and non-terminals is acceptable on the right hand side.
Context-sensitive	$X \rightarrow Y$ with $\|X\| \leq \|Y\|$ and $X, Y \subseteq (N^*T^*)^*$	Context-sensitive grammars require that rules be non-reducing from left to right (there must always be more terminals on the right-hand side than on the left-hand side).
Recursively Enumerable	$(N^*T^*)^* \rightarrow (N^*T^*)^*$	Recursively enumerable grammars have no restrictions on their production rules.

Regular Expressions

- A regular expression is an expression that can be used to generate or recognize strings in a particular language.
- A regular grammar can be equivalently represented by a regular expression.

Regular Expressions Operators

Operation/Entity	Usage/Symbol	Meaning
Alternation	$x\|y$	This regular expression indicates the next symbol generated/recognized will be either x or y.
Grouping	(x)	Similar to arithmetic expression this is used to group portions of the regular expressions together thus changing the precedence and scope of operators applied around them. Consider $x\|y^+$ and $(x\|y)^+$ that represent different languages.
Repetition (0 or more)	x^*	This operator is used to represent a repetition of the symbol/regular expression it is applied to zero or more times.
Repetition(1 or more)	x^+	This operator is used to represent a repetition of the symbol/regular expression it applied to one or more times.
Option	$x?$	This operator is used to represent a symbol/regular expression that is optional, it can be applied one or zero times.

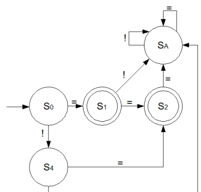
Finite Automata

- A finite automata is another way one can recognize or generate strings in a particular language.
- An automata is represented by a number of states and transitions and can be represented as a table or graphically.
- Transitions in a finite automata are usually triggered by a symbol in the alphabet of the language you are trying to recognize but for nondetermistic finite automata ϵ transitions are also possible.

Finite Automata Example

Consider the following different finite automata recognizing the language containing the ==, != and = strings with an alphabet { =, ! }. Below is a discrete finite automata represented in both tabular and graphical form for this language.

State		Input	
		=	!
→	S_0	S_1	S_4
*	S_1	S_2	S_A
*	S_2	S_A	S_A
	S_4	S_2	S_A
	S_A	S_A	S_A



Regular grammars.

- A regular grammar can be left-linear or right-linear.
- Left-linear grammars have non-terminals on the leftmost side of the right hand of the production rule.
  - $N \longrightarrow NT^*$ and $N \longrightarrow T^*$
- Right-linear grammars have non-terminals on the rightmost side of the right hand of the production rule.
  - $N \longrightarrow T^*N$ and $N \longrightarrow T^*$

Grammar Derivations

- A derivation shows the steps taken through the productions of a grammar in recognition or generation of a string.
- Left-most derivations expand productions from the left most non-terminal of the right hand side of a production.
- Right-most derivations expand productions from the right most non-terminal of the right hand side of a production.
- Derivations can also be represented by derivation trees whose root is the first non-terminal use and subsequent non-leaf subtrees the non-terminals used and the left nodes the terminals matched.

Ambiguous Grammars

- A grammar is considered ambiguous if there exists a string in the language the grammar represents that has more than one leftmost or rightmost derivation.

Syntax Diagram

- Method to represent grammar diagrammatically.
- Terminals in a syntax diagram are represented by ovals and non-terminals by squares.
- Each path from start to end represents a string generated or recognized by the grammar.
- Production alternatives, BNF $|$, repetition of one or more, EBNF $\{\ \}^+$, repetition of zero or more, EBNF $\{\ \}$ and options, EBNF $[\ ]$ can all be easily represented in a syntax diagram.

Syntax Diagram

For the following example consider a grammar with non-terminal A and terminal 0. Below we show the syntax diagram equivalence to various BNF and EBNF productions.

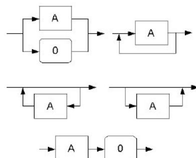


Figure: Syntax Diagram Examples.

Position	Represents	Equivalent BNF or EBNF production
Top left	Alternative	A | 0
Top Right	Repetition of one or more.	$\{ A \}^+$
Middle Left	Repetition of zero or more.	{ A }
Middle Right	Optional	[A]
Bottom	General Production	A 0

Grammar Practice

For each of the following languages consider whether the language can be represented by a regular grammar and if so represent it with a regular formal grammar, a regular expression and a DFA. If the grammar is not regular but is context-free represent the grammar with a BNF, an EBNF and a Syntax Diagram.

1. The language over the english alphabet consisting of palindromes.
2. The language consisting of C++ function prototypes.
3. The language consisting of all even integers.
4. The language over the english alphabet that contain an odd number of a's and even number of c's.
5. The language representing the course numbering system used for undergraduate courses at UNBC.

L and R-expressions

- L-expressions are expressions that evaluate to a memory location.
  - In an assignment statement in C the variable identifier you are using to assign to would be considered an l-expression as it contains the memory location to write the result back to.
- R-expressions are expressions that evaluate to a value.
  - In an assignment statement in C the expressions/value on the right hand would be considered an R-expression as it evaluates to a value that will stored in the variable on the left hand side.

Binding, Symbol Tables and Names

- Binding refers to associating attributes to a particular name.
- Binding can occur before the execution of a program in the static case or during run-time for the dynamic case.
- A symbol table is usually used to keep of a map of names to their attributes.
- It is possible to have the same name represented multiple times within a particular source but differentiated by its presence in different blocks/scopes.
- There are two cases for names we use within a particular program, constants whose whose values/attributes will not change throughout the execution and variables who values/attributes can change.

Free Variable Binding - Scoping

- Within a particular scope a free variable that is not declared within that scope usually refers to a variable declared in a parent scope.
- There are two ways to determine the binding of free variables for a particular scope, statically and dynamically.
- Static binding looks at the lexical/textual structure of the scope and looks at the parent scope the current scope is nested in.
- Dynamic binding looks at the caller, as well as its callers, for a particular procedure at runtime to determine a particular variable binding.
- The scope closest to the current scope, either in the lexical or call order sense, will be used for the binding of free variables.

Dynamic Memory Allocation and the Heap

- When we dynamically allocate memory in a language we usually utilize the heap for this purpose.
- With pointers/references it is possible to have alias, multiple names bound to the same memory location.
- It is also possible to have dangling pointers where the memory location a variable refers to has been reclaimed.
- Not de-allocating any dynamic memory throughout the execution of your program can lead to garbage, memory that is allocated but is not accessible from the program.
- Garbage collection is a method to automatically reclaim garbage memory without the programmer having to worry about de-allocating memory.

Denotational Semantics

- Method of showing the semantics/meaning for elements of a particular programming language.
- Demonstrates the mapping between the execution of a particular segment and its result.
  - For example you can show the state that executing a command generates or the state and value resulting from the evaluation of a particular expression.
- Semantic domains and functions need to be defined before you can show a semantic clause.

Semantic Domains and Functions

- Semantic domains demonstrate the domains over which various basic elements utilized by a particular programming language are defined.
- Examples of this include domains for primitive types, memory or generally the current state of the environment a particular program is running in.
- Semantic functions are used to define the mapping between a particular type of statement in a programming language to its results.
- Examples of this include commands that cause the state of memory to change and simply map from an old state to a new one or expressions, which with side-effects, can map the current state to a new state as well as generate a value.

Direct Denotational Semantic Example

Semantic domains:

$$\begin{aligned} \text{State} &= \text{Memory} \\ \text{Memory} &= \text{Identifier} \rightarrow \text{Value} \\ \text{Value} &= \text{Number} \times \text{Boolean} \end{aligned}$$

Semantic Functions:

$$\begin{aligned} E &: \text{Exp} \rightarrow \text{State} \rightarrow [[\text{Value} \times \text{State}] + \{\text{error}\}] \\ C &: \text{Com} \rightarrow \text{State} \rightarrow [\text{State} + \{\text{error}\}] \end{aligned}$$

For our example we will be utilizing the following semantic domains and functions for a language with operators having the same meaning as their C++ counterparts.

Direct Denotational Semantic Example

Let us consider the command *I-=2*. The semantic clause for this command is as follows:

$$\begin{aligned}\mathcal{C}[[I-=2]]s = \quad & (\mathcal{E}[[I]]s = (v, m)) \rightarrow \\ & \quad m[v-2/I], \\ & \quad \text{error}\end{aligned}$$

If add the stipulation that I must be a numeral then we get the following semantic clause indicating the value retrieved from I must be this particular type. This gives us the following semantic clause:

$$\begin{aligned}\mathcal{C}[[I-=2]]s = \quad & (\mathcal{E}[[I]]s = (v, m)) \rightarrow \\ & \quad \text{isNumeral(v)} \rightarrow \\ & \quad\quad m[v-2/I], \\ & \quad\quad \text{error}, \\ & \quad \text{error}\end{aligned}$$

Direction Denotational Semantic Example

Let us consider the command *if*($E_1 \mid\mid E_2$) C_1; where E_1 and E_2 are expressions and C_1 is a command. The semantic clause for this command which we shall name C_if is as follows:

$$
\begin{aligned}
\mathcal{C}[[C_if]]s = \quad & (\mathcal{E}[[E_1]]s = (v_1, s_1)) \rightarrow \\
& \quad v_1 \rightarrow \\
& \qquad \mathcal{C}[[C_1]]s_1, \\
& \qquad (\mathcal{E}[[E_2]]s_1 = (v_2, s_2)) \rightarrow \\
& \qquad\quad v_2 \rightarrow \\
& \qquad\qquad \mathcal{C}[[C_1]]s_2, \\
& \qquad\qquad s_2, \\
& \qquad\quad \text{error}, \\
& \quad \text{error}
\end{aligned}
$$

Direct Denotational Semantic Practice Cont.

1. `while(E1 || E2) if(E3) C;`
2. `if(E1) C1 else C2;`
3. `while(E1 && E2) C;`
4. `I1 = I2*2+3`