Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 2, Parts 1 and 2: Formal Grammars

© 2007 Jones and Bartlett P

Programs are Strings

Consider the following code fragment:

```
void main(void)
  {
    char *message[] = {"Hello ", "World"};
    int i;

    for(i = 0; i < 2; ++i)
      printf("%s", message[i]);
    printf("\n");
}
```

The code might as well appear as follows:

void main(void) { char *message[] = {”Hello ”, ”World”}; int i; for(i = 0; i < 2; ++i) printf(”%s”, message[i]); printf(”\n”); }

© 2007 Jones and Bartlett P

Another Example

Consider the Lisp function definition:

```
(defun hello
   (print
     (cons 'Hello (list 'World))))
```

Again, as the compiler sees it:

(defun hello (print (cons 'Hello (list 'World))))

© 2007 Jones and Bartlett P

The Point(s)

- The source code (i.e., your program) is merely a sequence of symbols.
- This sequence (string) is characterized via formal grammars.
- The programming language which allows production of these strings exists to facilitate communication between the programmer and the compiler.

© 2007 Jones and Bartlett P

Formal Grammars

- Grammars generate languages
- The syntax of a programming language may be described through specification of a grammar.
- There are many different types of grammars (later).

© 2007 Jones and Bartlett P

Language Syntax Specification and Productions

- The syntax of a programming language may basically be written using a set of reserved words (primitives) and rules (productions) for combining these reserved words to form programs.
- For example, a part of the syntax of Pascal may be shown in BNF:

$$function - identifier ::= identifier$$

$$identifier ::= letter\{letter - or - digit\}$$

$$letter ::= a|b|c|\ldots|z|\ldots|A|\ldots|Z$$

$$digit ::= 0|1|2|3|4|\ldots|9$$

$$letter - or \ - \ digit ::= letter|digit$$

::= means 'is defined as'

| means 'or'

© 2007 Jones and Bartlett P

{} indicates items which may be repeated zero or more times

© 2007 Jones and Bartlett P

What About a Programming Language Compiler?

- Part of a compiler is a *grammatical recognizer* or *parser*.
- A compiler or interpreter for a (higher-level) language attempts to generate lower level machine instructions by determining the desired structure of the input high level language program through *parsing* the input or source code.
- The compiler/interpreter begins with processing of the program as an input string.
- This processing is based upon checking if the string meets a formal specification of the programming language syntax.

© 2007 Jones and Bartlett P

An Alphabet and Forming Strings

- An alphabet (V) is a finite, nonempty set of *symbols*, e.g.:

$$V = \{a, b, c, \ldots z\}$$

- The concatenation of a and b, denoted $a \circ b$, produces a sequence of two symbols simply denoted hereafter as ab.
- A *string* over V is either a single symbol from V or a sequence of symbols formed by concatenation of zero or more symbols from V. Therefore, from V above, 'a' 'ab' 'az' and '$azab$' are strings or *sentences* over V.
- The length of (or number of symbols in) string s is denoted $|s|$.

© 2007 Jones and Bartlett P

- A string has a natural ordering of elements from left to right.
- Often it is convenient to denote a string like $x = aaa \ldots a$, where a symbol (or sequence of symbols) is repeated n times as $x = a^n$. For example:

$$aabbbcccc = a^2b^3c^4$$

© 2007 Jones and Bartlett P

The Closure Set

Define V^+ as

$$V^+ = V \cup V^2 \cup V^3 \cup \ldots$$

V^+ is the set of all nonempty sentences producible using V. Adding the empty string to V^+ produces V^*, i.e.,

$$V^* = \{\epsilon\} \cup V^+$$

V^* is denoted the *closure (set)* of V and $V^+ = V^* - \{\epsilon\}$ is often called the *positive closure* of V.

© 2007 Jones and Bartlett P

Languages Using Strings

Grammars are used with V to give some meaning to a subset of strings, $L \subseteq V^*$.

L is called a *language*. Furthermore:

- Languages are generated by grammars.
- Another viewpoint is that a grammar restricts the production of strings from V.
- Grammars are used to recognize (parse) elements of a language.

© 2007 Jones and Bartlett P

Grammar: Formal Definition

A *grammar*

$$G = (V_T, V_N, P, S)$$

consists of the following four entities:

1. A set of terminal or primitive symbols (primitives), denoted V_T (or, alternately, Σ). These are the elemental 'building blocks' of the grammar (and corresponding language).

2. A set of *non-terminal symbols, or variables*, which are used as intermediate quantities in the generation of an outcome consisting solely of terminal symbols. This set is denoted as V_N (or, alternately, N).

   Note that V_T and V_N are disjoint sets, i.e., $V_T \cap V_N = \emptyset$.

3. A set of *productions, or production rules or rewriting rules*

© 2007 Jones and Bartlett P

which govern how strings may be formed. It is this set of productions, coupled with the terminal symbols, which principally gives the grammar its 'structure.' The set of productions is denoted P.

4. A starting (or root) symbol, denoted S. $S \in V_N$.

© 2007 Jones and Bartlett P

Grammar Application Modes

A grammar may be used in one of two modes:

1. *Generative*: The grammar is used to create a string of terminal symbols using P; a *sentence* in the language of the grammar is thus generated.
2. *Analytic*: Given a sentence (possibly in the language of the grammar), together with specification of G, one seeks to determine:
   (a) If the sentence was generated by G; and, if so,
   (b) The structure (usually characterized as the sequence of productions used) of the sentence. This is where we begin to consider semantics.

© 2007 Jones and Bartlett P

Languages, Possible Strings and $L(G)$

- Any subset $L \subseteq V_T^*$ is a *language*.
- If $|L|$ is finite, the language is called finite, otherwise it is infinite.
- The *language generated by grammar* G, denoted $L(G)$, is the set of all strings which satisfy
  1. Each string consists solely of terminal symbols from V_T of G; and
  2. Each string was produced from S using P of G.

© 2007 Jones and Bartlett P

Grammar Types/Definitions

1. Symbols beginning with a capital letter (e.g., S_1 or S) are elements of V_N.
2. Symbols beginning with a lowercase letter (e.g., a or b) are elements of V_T.
3. n denotes the length of string s, i.e.,

$$n = |s|$$

4. Greek letters (e.g., α, β) represent (possibly empty) strings, typically comprised of terminals and/or nonterminals.

© 2007 Jones and Bartlett P

T_1: Context-Sensitive (CS)

A T_1 or context sensitive grammar restricts productions to:

$$\alpha\alpha_i\beta \rightarrow \alpha\beta_i\beta$$

meaning β_i replaces α_i *in the context of* α *and* β, where α, $\beta \in (V_N \cup V_T)^*, \alpha_i \in V_N$ and $\beta_i \in (V_N \cup V_T)^* - \{\epsilon\}$. Note that α or β (or both) may equal ϵ.

Furthermore, $|\alpha\alpha_i\beta| \leq |\alpha\beta_i\beta|$ Replacements involving the empty string are treated as a special case.

© 2007 Jones and Bartlett P

T_2:Context Free (CFG) (corrected)

In a T_2 or context-free grammar[a], the production restrictions are:

$$\alpha_1 = S_1 \in V_N$$

(α_1 *must be a single nonterminal*)

$$S_1 \rightarrow \beta_2$$

where $\beta_2 \in (V_N \cup V_T)^*$.

Note a CFG production allows S_1 to be replaced by string β_2 *independently or irrespective of the context in which* S_1 *appears.*

[a]There appear to be multiple definitions of this.

© 2007 Jones and Bartlett P

Representation and Complexity

In progressing from a T_0 to a T_4 grammar, notice that as production restrictions increase, representational power decreases.

At the same time, increasing production restrictions leads to simpler parsers.

Parsing complexity in a T_2 (CFG) grammar is a linear function of the number of rewrite rules in a derivation.

Denoting $L(T_i)$ as the class of language generated by grammar T_i, the above restrictions indicate:

$$L(T_3) \subset L(T_2) \subset L(T_1) \subset L(T_0)$$

© 2007 Jones and Bartlett P

Parsing and Complexity

Context free grammars and the class of context-free languages they generate are paramount in the study of programming languages, since they are reasonably adequate for describing the syntax of many programming languages. It is also worth noting:

- Context-free grammars are important because they are the most descriptively versatile grammars for which effective (and efficient) parsers are available.
- *The production restrictions increase in going from context-sensitive (T_1) to context free (T_2) cases.*

© 2007 Jones and Bartlett P

A Sample G_s: Type and $L(G_s)$

$$S \rightarrow AB$$

$$S \rightarrow C$$

$$A \rightarrow C$$

$$A \rightarrow a$$

$$B \rightarrow b$$

$$B \rightarrow c$$

$$C \rightarrow d$$

In BNF:

```
<s> ::= <a><b> | <c>
<a> ::= <c> | a
<b> ::= b | c
<c> ::= d
```

S is the starting symbol and V_T and V_N may be deduced from the productions.

© 2007 Jones and Bartlett P

1. What type(s) of grammar do we have?
2. What is $L(G_s)$ corresponding to G_s?

© 2007 Jones and Bartlett P