3.0.6 Exercises

1. Show three different *derivations* using each of the following grammars, with starting nonterminal S.

   (a)

   1. S $\rightarrow$ a S
   2. S $\rightarrow$ b A
   3. A $\rightarrow$ b S
   4. A $\rightarrow$ c

   (b)

   1. S $\rightarrow$ a B c
   2. B $\rightarrow$ A B
   3. A $\rightarrow$ B A
   4. A $\rightarrow$ a
   5. B $\rightarrow \epsilon$

   (c)

   1. S $\rightarrow$ a S B c
   2. a S A $\rightarrow$ a S b b
   3. B c $\rightarrow$ A c
   4. S b $\rightarrow$ b
   5. A $\rightarrow$ a

   (d)

   1. S $\rightarrow$ a b
   2. a $\rightarrow$ a A b B
   3. A b B $\rightarrow \epsilon$

2. Classify the grammars of the previous problem according to Chomsky's definitions (give the most restricted classification applicable).

3. Show an example of a *grammar rule* which is:

   (a) `Right Linear`
   (b) `Context-Free, but not Right Linear`
   (c) `Context-Sensitive, but not Context-Free`
   (d) `Unrestricted, but not Context-Sensitive`

4. For each of the given input strings show a derivation tree using the following grammar.

1. S → S a A
2. S → A
3. A → A b B
4. A → B
5. B → c S d
6. B → e
7. B → f

(a) eae (b) ebe (c) eaebe (d) ceaedbe (e) cebedaceaed

5. Show a left-most derivation for each of the following strings, using grammar G4 of section 3.0.3.

   (a) var + const (b) var + var * var (c) (var) (d) (var + var) * var

6. Show derivation trees which correspond to each of your solutions to the previous problem.

7. Some of the following grammars may be ambiguous; for each ambiguous grammar, show two different derivation trees for the same input string:

   (a)

   1. S → a S b
   2. S → A A
   3. A → c
   4. A → S

   (b)

   1. S → A a A
   2. S → A b A
   3. A → c
   4. A → z

   (c)

   1. S → a S b S
   2. S → a S
   3. S → c

   (d)

   1. S → a S b c
   2. S → A B
   3. A → a
   4. B → b

8. Show a pushdown machine that will accept each of the following languages:
(a) $\{a^n b^m\} m > n > 0$
(b) $a * (a + b)c*$
(c) $\{a^n b^n c^m d^m\} m, n \geq 0$
(d) $\{a^n b^m c^m d^n\} m, n > 0$
(e) $\{N_i c (N_{i+1})^r\}$
- where N_i is a binary representation of the integer i, and $(N_i)^r$ is N_i written right to left (reversed). Examples:

i	A string which should be accepted
19	10011c00101
19	10011c001010
15	1111c00001
15	1111c0000100

Hint: Use the first state to push N_i onto the stack until the c is read. Then use another state to pop the stack as long as the input is the complement of the stack symbol, until the top stack symbol and the input symbol are equal. Then use a third state to ensure that the remaining input symbols match the symbols on the stack. A fourth state can be used to allow for leading (actaully, trailing) zeros after the c.

9. Show the output and the sequence of stacks for the machine of Figure 3.8 for each of the following input strings:

(a) $a + a * a \hookleftarrow$
(b) $(a + a) * a \hookleftarrow$
(c) $(a) \hookleftarrow$
(d) $((a)) \hookleftarrow$

10. Show a grammar and an extended pushdown machine for the language of prefix expressions involving addition and multiplication. Use the terminal symbol a to represent a variable or constant. Example: *+aa*aa

11. Show a pushdown machine to accept palindromes over {0,1} with center-marker c. This is the language, Pc, referred to in section 3.0.5.

12. Show a grammar for the language of valid regular expressions (as defined in section 2.0) over the alphabet {0,1}. You may assume that concatenation is always represented by a raised dot. An example of a string in this language would be:
$(0 + 1 \cdot 1) * \cdot 0$
An example of a string not in this language would be:
$((0 + +1)$
Hint: Think about grammars for arithmetic expressions.

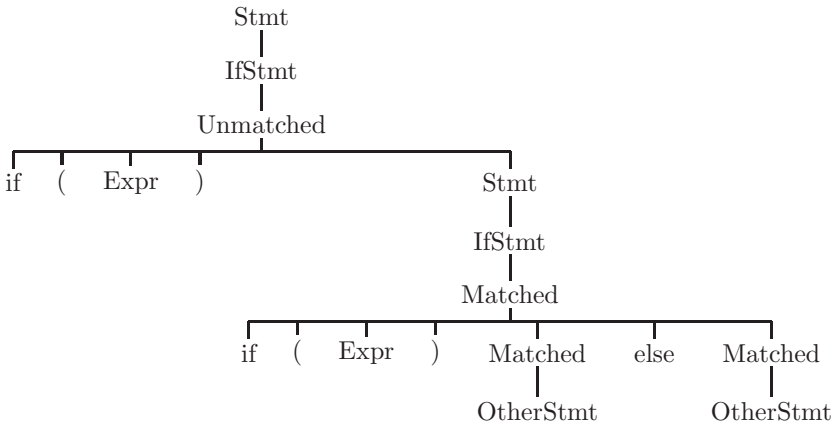

Figure 3.11: A derivation tree for the string `if ( Expr ) if ( Expr ) Stmt else Stmt` using grammar G7

This grammar differentiates between the two different kinds of *if* statements, those with a matching *else* (Matched) and those without a matching *else* (Unmatched). The nonterminal OtherStmt would be defined with rules for statements other than if statements (while, assignment, for, ...). A derivation tree for the string `if ( BoolExpr ) if ( BoolExpr ) OtherStmt else OtherStmt` is shown in Figure 3.11.

3.1.1 Exercises

1. Show derivation trees for each of the following input strings using grammar G5.

   (a) $var * var$

   (b) $(var * var) + var$

   (c) (var)

   (d) $var * var * var$

2. Extend grammar G5 to include subtraction and division so that subtrees of any derivation tree correspond to subexpressions.

3. Rewrite your grammar from the previous problem to include an exponentiation operator, `^`, such that `x^y` is x^y. Again, make sure that subtrees in a derivation tree correspond to subexpressions. Be careful, as exponentiation is usually defined to take precedence over multiplication and associate to the right:

   ```
   2*3^2 = 18  and  2^2^3 = 256
   ```

4. Two grammars are said to be isomorphic if there is a one-to-one correspondence between the two grammars for every symbol of every rule. For example, the following two grammars are seen to be isomorphic, simply by making the following substitutions: substitute B for A, x for a, and y for b.

   $S \rightarrow aAb$ $\quad$ $S \rightarrow xBy$
   $A \rightarrow bAa$ $\quad$ $B \rightarrow yBx$
   $A \rightarrow a$ $\quad$ $B \rightarrow x$

   Which *grammar* in section 3.0 is isomorphic to the grammar of Exercise 4 in section 3.1?

5. How many different derivation trees are there for each of the following `if` statements using grammar G6?

   (a) `if ( BoolExpr ) Stmt`

   (b) `if ( BoolExpr ) Stmt else if ( BoolExpr ) Stmt`

   (c) `if ( BoolExpr ) if ( BoolExpr ) Stmt else Stmt else Stmt`

   (d) `if ( BoolExpr ) if ( BoolExpr ) if ( BoolExpr ) Stmt else Stmt`

6. In the original C language it is possible to use assignment operators: `var =+ expr` means `var = var + expr` and `var =- expr` means `var = var - expr`. In later versions of C, C++, and Java the operator is placed before the equal sign:

   `var += expr` and `var -= expr`.

   Why was this change made?

3.2 The Parsing Problem

The student may recall, from high school days, the problem of diagramming English sentences. You would put words together into groups and assign syntactic types to them, such as noun phrase, predicate, and prepositional phrase. An example of a diagrammed English sentence is shown in Figure 3.12. The process of diagramming an English sentence corresponds to the problem a compiler must solve in the syntax analysis phase of compilation.

The syntax analysis phase of a compiler must be able to solve the parsing problem for the programming language being compiled: Given a grammar, G, and a string of input symbols, decide whether the string is in L(G); also, determine the structure of the input string. The solution to the parsing problem will be 'yes' or 'no', and, if 'yes', some description of the input string's structure, such as a derivation tree.