Chapter 3

Syntax Analysis

The second phase of a compiler is called ***syntax analysis***. The input to this phase consists of a stream of tokens put out by the lexical analysis phase. They are then checked for proper syntax, i.e. the compiler checks to make sure the statements and expressions are correctly formed. Some examples of syntax errors in Java are:

```
x = (2+3) * 9);              // mismatched parentheses

if x>y x = 2;                // missing parentheses

while (x==3) do f1();        // invalid keyword do
```

When the compiler encounters such an error, it should put out an informative message for the user. At this point, it is not necessary for the compiler to generate an object program. A compiler is not expected to guess the intended purpose of a program with syntax errors. A good compiler, however, will continue scanning the input for additional syntax errors.

The output of the syntax analysis phase (if there are no syntax errors) could be a stream of atoms or syntax trees. An ***atom*** is a primitive operation which is found in most computer architectures, or which can be implemented using only a few machine language instructions. Each atom also includes operands, which are ultimately converted to memory addresses on the target machine. A ***syntax tree*** is a data structure in which the interior nodes represent operations, and the leaves represent operands, as discussed in Section 1.2.2. We will see that the parser can be used not only to check for proper syntax, but to produce output as well. This process is called ***syntax directed translation***.

Just as we used formal methods to specify and construct the lexical scanner, we will do the same with syntax analysis. In this case however, the formal methods are far more sophisticated. Most of the early work in the theory of compiler design focused on syntax analysis. We will introduce the concept of a formal grammar not only as a means of specifying the programming language, but also as a means of implementing the syntax analysis phase of the compiler.

3.0 Grammars, Languages, and Pushdown Machines

Before we discuss the syntax analysis phase of a compiler, there are some concepts of formal language theory which the student must understand. These concepts play a vital role in the design of the compiler. They are also important for the understanding of programming language design and programming in general.

3.0.1 Grammars

Recall our definition of ***language*** from Chapter 2 as a set of strings. We have already seen two ways of formally specifying a language – regular expressions and finite state machines. We will now define a third way of specifying languages, i.e. by using a grammar. A ***grammar*** is a list of rules which can be used to produce or generate all the strings of a language, and which does not generate any strings which are not in the language. More formally a grammar consists of:

1. A finite set of characters, called the ***input alphabet***, the ***input symbols***, or ***terminal symbols***.

2. A finite set of symbols, distinct from the terminal symbols, called ***nonterminal symbols***, exactly one of which is designated the ***starting nonterminal***.

3. A finite list of ***rewriting rules***, also called ***productions***, which define how strings in the language may be generated. Each of these rewriting rules is of the form $\alpha \rightarrow \beta$, where α and β are arbitrary strings of terminals and nonterminals, and α is not null.

The grammar specifies a language in the following way: beginning with the starting nonterminal, any of the rewriting rules are applied repeatedly to produce a ***sentential form***, which may contain a mix of terminals and nonterminals. If at any point, the sentential form contains no nonterminal symbols, then it is in the language of this grammar. If G is a grammar, then we designate the language specified by this grammar as L(G).

A ***derivation*** is a sequence of rewriting rules, applied to the starting nonterminal, ending with a string of terminals. A derivation thus serves to demonstrate that a particular string is a member of the language. Assuming that the starting nonterminal is `S`, we will write derivations in the following form:

$$S \Rightarrow \alpha \Rightarrow \beta \Rightarrow \gamma \Rightarrow \ldots \Rightarrow x$$

where α, β, γ are strings of terminals and/or nonterminals, and `x` is a string of terminals.

In the following examples, we observe the convention that all lower case letters and numbers are terminal symbols, and all upper case letters (or words which begin with an upper case letter) are nonterminal symbols. The starting nonterminal is always `S` unless otherwise specified. Each of the grammars shown in this chapter will be numbered (G1, G2, G3, ...) for reference purposes. The first example is grammar G1, which consists of four rules, the terminal symbols {`0,1`}, and the starting nonterminal, `S`.

G1:

```
1.  S → 0S0
2.  S → 1S1
3.  S → 0
4.  S → 1
```

An example of a derivation using this grammar is:

```
S ⇒ 0S0 ⇒ 00S00 ⇒ 001S100 ⇒ 0010100
```

Thus, `0010100` is in L(G1), i.e. it is one of the strings in the language of grammar G1. The student should find other derivations using G1 and verify that G1 specifies the language of palindromes of odd length over the alphabet {`0,1`}. A ***palindrome*** is a string which reads the same from left to right as it does from right to left.

L(G1) = {`0, 1, 000, 010, 101, 111, 00000, ...` }

In our next example, the terminal symbols are {`a,b`} (ε represents the null string and is not a terminal symbol).

G2:

```
1.  S → ASB
2.  S → ε
3.  A → a
4.  B → b
```

```
S ⇒ ASB ⇒ AASBB ⇒ AaSBB ⇒ AaBB ⇒ AaBb ⇒ Aabb ⇒ aabb
```

Thus, `aabb` is in L(G2). G2 specifies the set of all strings of `a`'s and `b`'s which contain the same number of `a`'s as `b`'s and in which all the `a`'s precede all the `b`'s. Note that the null string is permitted in a rewriting rule.

L(G2) = { `ε, ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb, ...`}
= {$a^n b^n$} such that $n \geq 0$

This language is the set of all strings of `a`'s and `b`'s which consist of zero or more `a`'s followed by exactly the same number of `b`'s.

Two grammars, g1 and g2, are said to be ***equivalent*** if L(g1) = L(g2) – i.e., they specify the same language. In this example (grammar G2) there can be several different derivations for a particular string – i.e., the rewriting rules could have been applied in a different sequence to arrive at the same result.

Sample Problem 3.0 (a)

Show three different derivations using the grammar shown below:

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

Solution

```
S  ⇒  a S A  ⇒  a B A A  ⇒  a B a b A  ⇒  a B a b a b
      ⇒  a b A a b a b  ⇒  a b a b a b a b
S ⇒ a S A ⇒ a S a b  ⇒  a B A a b  ⇒  a b A A a b
      ⇒  a b a b A a b  ⇒  a b a b a b a b
S  ⇒ B A  ⇒  b A A  ⇒  b a b A  ⇒  b a b a b
```

Note that in the solution to this problem we have shown that it is possible to have more than one derivation for the same string: `abababab`.

3.0.2 Classes of Grammars

In 1959 Noam Chomsky, a linguist, suggested a way of classifying grammars according to complexity. The convention used below, and in the remaining chapters, is that the term "string" includes the null string and that, in referring to grammars, the following symbols will have particular meanings:

`A,B,C,...`	A single nonterminal
`a,b,c,...`	A single terminal
`...,X,Y,Z`	A single terminal or nonterminal
`...,x,y,z`	A string of terminals
$\alpha, \beta, \gamma, ...$	A string of terminals and nonterminals

Here is Chomsky's classification of grammars:

0. Unrestricted – An ***unrestricted grammar*** is one in which there are no restrictions on the rewriting rules. Each rule may consist of an arbitrary string of terminals and nonterminals on both sides of the arrow (though ε is permitted on the right side of the arrow only). An example of an unrestricted rule would be:

$$SaB \rightarrow cS$$

1. Context-Sensitive – A ***context-sensitive grammar*** is one in which each rule must be of the form:

$$\alpha A \gamma \rightarrow \alpha \beta \gamma$$

where α,β and γ are any string of terminals and nonterminals (including ε), and `A` represents a single nonterminal. In this type of grammar, it is the nonterminal on the left side of the rule (`A`) which is being rewritten, but only if it appears in a particular ***context***, α on its left and γ on its right. An example of a context-sensitive rule is shown below:

```
SaB → caB
```

which is another way of saying that an `S` may be rewritten as a `c`, but only if the `S` is followed by `aB` (i.e. when `S` appears in that context). In the above example, the left context is null.

2. Context-Free – A ***context-free grammar*** is one in which each rule must be of the form:

$$A \rightarrow \alpha$$

where `A` represents a single nonterminal and α is any string of terminals and nonterminals. Most programming languages are defined by grammars of this type; consequently, we will focus on context-free grammars. Note that both grammars G1 and G2, above, are context-free. An example of a context-free rule is shown below:

```
A  →  aABb
```

3. Right Linear – A ***right linear grammar*** is one in which each rule is of the form:

```
A → aB
```
or
```
A → a
```

where `A` and `B` represent nonterminals, and `a` represents a terminal. Right linear grammars can be used to define lexical items such as identifiers, constants, and keywords.

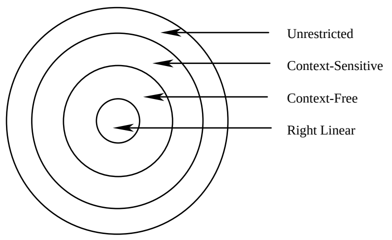


Figure 3.1 Classes of Grammars

Note that every context-sensitive grammar is also in the unrestricted class. Every context-free grammar is also in the context-sensitive and unrestricted classes. Every right linear grammar is also in the context-free, context-sensitive, and unrestricted classes. This is represented by the diagram of Figure 3.1, above, which depicts the classes of grammars as circles. All points in a circle belong to the class of that circle.

A ***context-sensitive language*** is one for which there exists a context-sensitive grammar. A ***context-free language*** is one for which there exists a context-free grammar. A ***right linear language*** is one for which there exists a right linear grammar. These classes of languages form the same hierarchy as the corresponding classes of grammars.

We conclude this section with an example of a context-sensitive grammar which is not context-free.

G3:

```
1.  S → aSBC
2.  S → ε
3.  aB → ab
4.  bB → bb
5.  C → c
6.  CB → CX
7.  CX → BX
8.  BX → BC
```

```
S ⇒ aSBC ⇒ aaSBCBC ⇒ aaBCBC ⇒ aaBCXC ⇒ aaBBXC ⇒ aaBBCC
      ⇒ aabBCC ⇒ aabbCC ⇒ aabbCc ⇒ aabbcc
```

The student should perform other derivations to understand that

L(G3) = {ε, abc, aabbcc, aaabbbccc, ...}
= $\{a^n b^n c^n\}$ where $n \geq 0$

i.e., the language of grammar G3 is the set of all strings consisting of `a`'s followed by exactly the same number of `b`'s followed by exactly the same number of `c`'s. This is an example of a context-sensitive language which is not also context-free; i.e., there is no context-free grammar for this language. An intuitive understanding of why this is true is beyond the scope of this text.

Sample Problem 3.0 (b):

Classify each of the following grammar rules according to Chomsky's classification of grammars (in each case give the largest – i.e., most restricted – classification type that applies):

		Solution:
1.	`aSb → aAcBb`	Type 1, Context-Sensitive
2.	`B → aA`	Type 3, Right Linear
3.	`a → ABC`	Type 0, Unrestricted
4.	`S → aBc`	Type 2, Context-Free
5.	`Ab → b`	Type 1, Context-Sensitive
6.	`AB → BA`	Type 0, Unrestricted

3.0.3 Context-Free Grammars

Since programming languages are typically specified with context-free grammars, we are particularly interested in this class of grammars. Although there are some aspects of programming languages that cannot be specified with a context-free grammar, it is generally felt that using more complex grammars would only serve to confuse rather than clarify. In addition, context-sensitive grammars could not be used in a practical way to construct the compiler.

Context-free grammars can be represented in a form called Backus-Naur Form (BNF) in which nonterminals are enclosed in angle brackets <>, and the arrow is replaced by a `::=,` as shown in the following example:

```
<S> ::= a <S> b
```

which is the BNF version of the grammar rule:

```
S → a S b
```

This form also permits multiple definitions of one nonterminal on one line, using the alternation vertical bar (|).

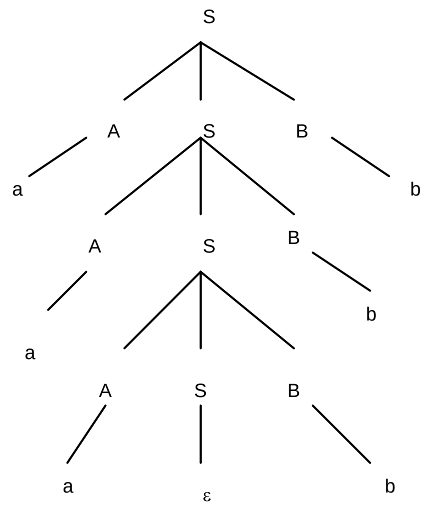


Figure 3.2 A Derivation Tree for `aaabbb` Using Grammar G2

```
<S> ::= a <S> b | ε
```

which is the BNF version of two grammar rules:

```
S → a S b
S → ε
```

BNF and context-free grammars are equivalent forms, and we choose to use context-free grammars only for the sake of appearance.

We now present some definitions which apply only to context-free grammars. A ***derivation tree*** is a tree in which each interior node corresponds to a nonterminal in a sentential form and each leaf node corresponds to a terminal symbol in the derived string. An example of a derivation tree for the string `aaabbb`, using grammar G2, is shown in Figure 3.2.

A context-free grammar is said to be ***ambiguous*** if there is more than one derivation tree for a particular string. In natural languages, ambiguous phrases are those which may have more than one interpretation. Thus, the derivation tree does more than show that a particular string is in the language of the grammar – it shows the structure of the string, which may affect the meaning or semantics of the string. For example, consider the following grammar for simple arithmetic expressions:

G4:

```
1.    Expr → Expr + Expr
2.    Expr → Expr * Expr
3.    Expr → ( Expr )
```

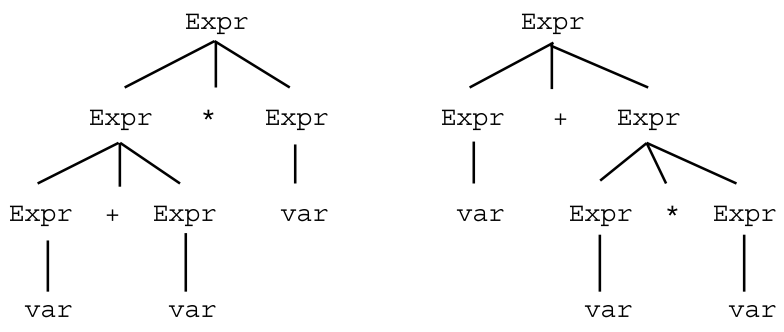


Figure 3.3 Two Different Derivation Trees for the String var + var ∗ var

```
4.    Expr → var
5.    Expr → const
```

Figure 3.3 shows two different derivation trees for the string `var+var*var`, consequently this grammar is ambiguous. It should be clear that the second derivation tree in Figure 3.3 represents a preferable interpretation because it correctly shows the structure of the expression as defined in most programming languages (since multiplication takes precedence over addition). In other words, all subtrees in the derivation tree correspond to subexpressions in the derived expression. A nonambiguous grammar for expressions will be given in Section 3.1.

A ***left-most derivation*** is one in which the left-most nonterminal is always the one to which a rule is applied. An example of a left-most derivation for grammar G2 above is:

$$S \Rightarrow ASB \Rightarrow aSB \Rightarrow aASBB \Rightarrow aaSBB \Rightarrow aaBB \Rightarrow aabB \Rightarrow aabb$$

We have a similar definition for ***right-most derivation***. A left-most (or right-most) derivation is a ***normal form*** for derivations; i.e., if two different derivations can be written in the same normal form, they are equivalent in that they correspond to the same derivation tree. Consequently, there is a one-to-one correspondence between derivation trees and left-most (or right-most) derivations for a grammar.

3.0.4 Pushdown Machines

Like the finite state machine, the ***pushdown machine*** is another example of an abstract or theoretic machine. Pushdown machines can be used for syntax analysis, just as finite state machines are used for lexical analysis. A pushdown machine consists of:

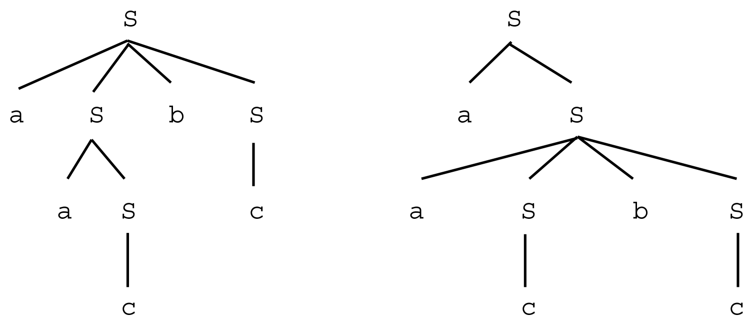
1. A finite set of states, one of which is designated the ***starting state***.

Sample Problem 3.0 (c)

Determine whether the following grammar is ambiguous. If so, show two different derivation trees for the same string of terminals, and show a left-most derivation corresponding to each tree.

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

Solution:



$S \Rightarrow a\ S\ b\ S \Rightarrow a\ a\ S\ b\ S \Rightarrow a\ a\ c\ b\ S \Rightarrow a\ a\ c\ b\ c$
$S \Rightarrow a\ S \Rightarrow a\ a\ S\ b\ S \Rightarrow a\ a\ c\ b\ S \Rightarrow a\ a\ c\ b\ c$

We note that the two derviation trees correspond to two different left-most derivations, and the grammar is ambiguous.

2. A finite set of input symbols, the ***input alphabet***.

3. An infinite stack and a finite set of stack symbols which may be pushed on top or removed from the top of the stack in a last-in first-out manner. The stack symbols need not be distinct from the input symbols. The stack must be initialized to contain at least one stack symbol before the first input symbol is read.

4. A state transition function which takes as arguments the current state, the current input symbol, and the symbol currently on top of the stack; its result is the new state of the machine.

5. On each state transition the machine may advance to the next input symbol or retain the input pointer (i.e., not advance to the next input symbol).

6. On each state transition the machine may perform one of the stack operations, push(X) or pop, where X is one of the stack symbols.

7. A state transition may include an exit from the machine labeled either Accept or Reject. This determines whether or not the input string is in the specified language.

Note that without the infinite stack, the pushdown machine is nothing more than a finite state machine as defined in Chapter 2. Also, the pushdown machine halts by taking an exit from the machine, whereas the finite state machine halts when all input symbols have been read.

An example of a pushdown machine is shown below, in Figure 3.4, in which the rows are labeled by stack symbols and the columns are labeled by input symbols. The ↩ character is used as an ***endmarker***, indicating the end of the input string, and the ▽ symbol is a stack symbol which we are using to mark the bottom of the stack so that we can test for the empty stack condition. The states of the machine are `S1` (in our examples `S1` will always be the starting state) and `S2`, and there is a separate transition table for each state. Each cell of those tables shows a stack operation (push() or pop), an input pointer function (advance or retain), and the next state. “Accept” and “Reject” are exits from the machine. The language of strings accepted by this machine is a^nb^n where $n \geq 0$ – i.e., the same language specified by grammar G2, above. To see this, the student should trace the operation of the machine for a particular input string. A trace showing the sequence of stack configurations and states of the machine for the input string `aabb` is shown in Figure 3.5. Note that while in state `S1` the machine is pushing `X`'s on the stack as each `a` is read, and while in state `S2` the machine is popping an `X` off the stack as each `b` is read.

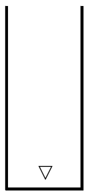
An example of a pushdown machine which accepts any string of correctly balanced parentheses is shown in Figure 3.6. In this machine, the input symbols are left and right parentheses, and the stack symbols are `X` and ▽. Note that this language could not be accepted by a finite state machine because there could be an unlimited number of left parentheses before the first right parenthesis. The student should compare the language accepted by this machine with the language of grammar G2.

The pushdown machines, as we have described them, are purely deterministic machines. A ***deterministic*** machine is one in which all operations are uniquely and completely specified regardless of the input (computers are deterministic), whereas a ***nondeterministic*** machine may be able to choose from zero or more operations in an unpredictable way. With nondeterministic pushdown machines it is possible to specify a larger class of languages. In this text we will not be concerned with nondeterministic machines.

We define a ***pushdown translator*** to be a machine which has an ***output function*** in addition to all the features of the pushdown machine described above. We may include this output function in any of the cells of the state transition table to indicate that the machine produces a particular output (e.g. `Out(x)`) before changing to the new state.

We now introduce an extension to pushdown machines which will make them easier to work with, but will not make them any more powerful. This extension is the ***Replace operation*** designated `Rep(X,Y,Z,...)`, where `X`, `Y`, and `Z` are any stack symbols. The replace function replaces the top stack symbol with all the symbols in its argument list. The Replace function is equivalent to a pop operation followed by a push

S1	a	b	↩
X	Push (X) Advance S1	Pop Advance S2	Reject
▽	Push (X) Advance	Reject	Accept



S2	a	b	↩
X	Reject S1	Pop Advance S2	Reject
▽	Reject	Reject	Accept

Figure 3.4 A Pushdown Machine to Accept the Language of Grammar G2

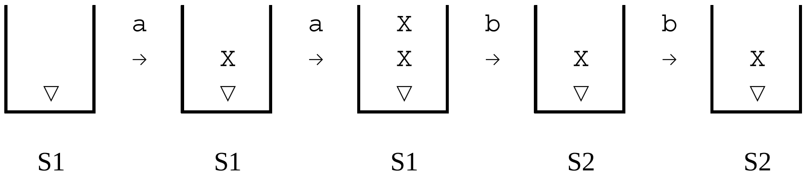


Figure 3.5 Sequence of Stacks as Pushdown Machine of Figure 3.4 Accepts the Input String aabb

S1	(	)	↩
X	Push (X) Advance S1	Pop Advance S1	Reject
▽	Push (X) Advance S1	Reject	Accept

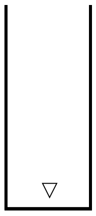


Figure 3.6 Pushdown Machine to Accept Any String of Well-Balanced Parentheses

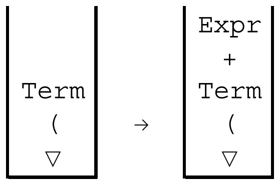


Figure 3.7 Effect on Stack of `Rep (Term, +, Expr)`

operation for each symbol in the argument list of the replace function. For example, the function `Rep (Term,+,Expr)` would pop the top stack symbol and push the symbols `Term, +`, and `Expr` in that order, as shown on the stack in Figure 3.7. (In this case, the stack symbols are separated by commas). Note that the symbols to be pushed on the stack are pushed in the order listed, left to right, in the Replace function. An ***extended pushdown machine*** is one which can use a Replace operation in addition to push and pop.

An extended pushdown machine is not capable of specifying any languages that cannot be specified with an ordinary pushdown machine – it is simply included here as a convenience to simplify some problem solutions. An ***extended pushdown translator*** is a pushdown translator which has a replace operation as defined above.

An example of an extended pushdown translator, which translates simple infix expressions involving addition and multiplication to postfix is shown in Figure 3.8, in which the input symbol `a` represents any variable or constant. An ***infix expression*** is one in which the operation is placed between the two operands, and a ***postfix expression*** is one in which the two operands precede the operation:

Infix	Postfix
`2 +  3`	`2 3 +`
`2 + 3 * 5`	`2 3 5 * +`
`2 * 3 + 5`	`2 3 * 5 +`
`(2 + 3) * 5`	`2 3 + 5 *`

Note that parentheses are never used in postfix notation. In Figure 3.8 the default state transition is to stay in the same state, and the default input pointer operation is `advance`. States `S2` and `S3` show only a few input symbols and stack symbols in their transition tables, because those are the only configurations which are possible in those states. The stack symbol `E` represents an expression, and the stack symbol `L` represents a left parenthesis. Similarly, the stack symbols `Ep` and `Lp` represent an expression and a left parenthesis on top of a plus symbol, respectively.

3.0.5 Correspondence Between Machines and Classes of Languages

We now examine the class of languages which can be specified by a particular machine. A language can be accepted by a finite state machine if, and only if, it can be specified with a right linear grammar (and if, and only if, it can be specified with a regular expression). This means that if we are given a right linear grammar, we can construct a finite state machine which accepts exactly the language of that grammar. It also means that if we are given a finite state machine, we can write a right linear grammar which specifies the same language accepted by the finite state machine.

S1	a	+	*	(	)	↩
E	Reject	push(+)	push(*)	Reject	pop retain S3	pop retain
E_p	Reject	pop out(+)	push(*)	Reject	pop retain S2	pop retain S2

L	push(E)
	out(a)
		Reject	Reject	push(L)	Reject	Reject

L_p	push(E)
	out(a)
		Reject	Reject	push(L)	Reject	Reject

L_s	push(E)
	out(a)
		Reject	Reject	push(L)	Reject	Reject

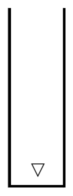
+	push(E_p)
	out(a)
		Reject	Reject	push(L_p)	Reject	Reject

*	pop out(a*)	Reject	Reject	push(L_s)	Reject	Reject

▽	push(E)
	out(a)
		Reject	Reject	push(L)	Reject	Accept

S2	)	↩
+	pop out(+) retain,S3	pop out(+) retain,S1
*	pop out(*) S1	Reject

S3	)
L	Rep(E) S1
L_p	Rep(E) S1
E	pop retain
L_s	pop retain S2
▽	Reject



Initial Stack

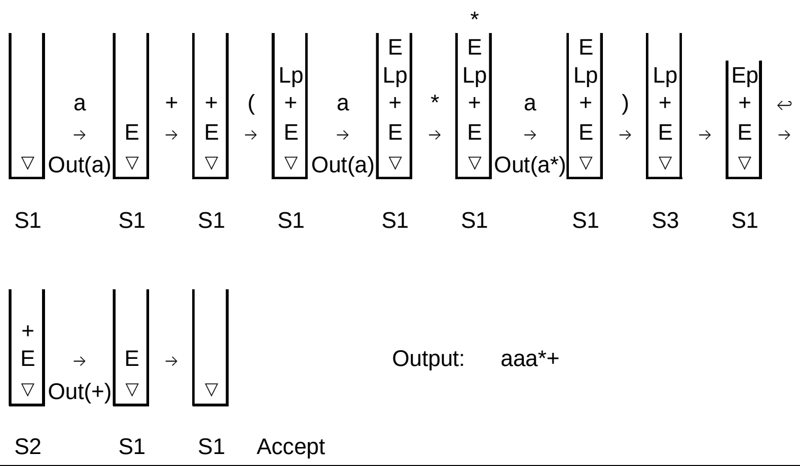
Figure 3.8 Pushdown Translator for Infix to Postfix Expressions

There are algorithms which can be used to produce any of these three forms (finite state machines, right linear grammars, and regular expressions), given one of the other two (see, for example, Hopcroft and Ullman [1979]). However, here we rely on the student's ingenuity to solve these problems.

Sample Problem 3.0 (d)

Show the sequence of stacks and states which the pushdown machine of Figure 3.8 would go through if the input were: `a+(a*a)`

Solution:



Output: aaa*+

Sample Problem 3.0 (e)

Give a right linear grammar for each of the languages specified in Sample Problem 2.0(a).

Solution:

(1) Strings over {0,1} containing an odd number of 0's.

1. S → 0
2. S → 1S
3. S → 0A
4. A → 1
5. A → 1A
6. A → 0S

(2) Strings over {0,1} which contain three consecutive 1's.

1. S → 1S
2. S → 0S
3. S → 1A
4. A → 1B
5. B → 1C
6. B → 1
7. C → 1C
8. C → 0C
9. C → 1
10. C → 0

(3) Strings over $\{0,1\}$ which contain exactly three 0's.

1.	`S → 1S`
2.	`S → 0A`
3.	`A → 1A`
4.	`A → 0B`
5.	`B → 1B`
6.	`B → 0C`
7.	`B → 0`
8.	`C → 1C`
9.	`C → 1`

(4) Strings over $\{0,1\}$ which contain an odd number of zeros and an even number of 1's.

1.	`S → 0A`
2.	`S → 1B`
3.	`S → 0`
4.	`A → 0S`
5.	`A → 1C`
6.	`B → 0C`
7.	`B → 1S`
8.	`C → 0B`
9.	`C → 1A`
10.	`C → 1`

We have a similar correspondence between machines and context-free languages. Any language which can be accepted by a deterministic pushdown machine can be specified by a context-free grammar. However, there are context-free languages which cannot be accepted by a deterministic pushdown machine. First consider the language, P_c, of palindromes over the alphabet $\{0,1\}$ with centermarker, c. $P_c = wcw^r$, where w is any string of 0's and 1's, and w^r is w reversed. A grammar for P_c is shown below:

```
S → 0S0
S → 1S1
S → c
```

Some examples of strings in this language are: `c, 0c0, 110c011, 111c111.`

The student should verify that there is a deterministic pushdown machine which will accept P_c. However, the language, P, of palindromes over the alphabet $\{0,1\}$ without centermarker cannot be accepted by a deterministic pushdown machine. Such a machine would push symbols on the stack as they are input, but would never know when to start popping those symbols off the stack; i.e., without a centermarker it never knows for sure when it is processing the mirror image of the initial portion of the input string. For this language a ***nondeterministic*** pushdown machine, which is one that can pursue several different courses of action, would be needed. Nondeterministic machines are beyond the scope of this text. A grammar for P is shown below:

```
S → 0S0
S → 1S1
S → 0
S → 1
S → ε
```

The subclass of context-free languages which can be accepted by a deterministic push-down machine are called ***deterministic context-free languages***.

Exercises 3.0

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

(a)
```
S → a S
S → b A
A → b S
A → c
```

(b)
```
S → a B c
B → A B
A → B A
A → a
B → ε
```

(c)
```
S      → a S B c
a S A  → a S b b
B c    → A c
S b    → b
A      → a
```

(d)
```
S      → a b
a      → a A b B
A b B  → ε
```

2. Classify the grammars of Problem 1 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 Problem 5.

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

(c)  1.  S → a S b S      (d)  1.  S → a S b c
     2.  S → a S               2.  S → A B
     3.  S → c                 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 the binary representation of the integer `i`, and $(N_i)^r$ is N_i written right to left (reversed). Example for `i=19: 10011c00101`

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.

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↩`	(b)	`(a+a)*a↩`
(c)	`(a)↩`	(d)	`((a))↩`

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 centermarker c. This is the language, P_c, referred to in Section 3.0.5.

12. Show a *grammar* for the language of valid regular expressions over the alphabet {0,1}. Hint: Think about grammars for arithmetic expressions.

3.1 Ambiguities in Programming Languages

Ambiguities in grammars for programming languages should be avoided. One way to resolve an ambiguity is to rewrite the grammar of the language so as to be unambiguous. For example, the grammar G4 in Section 3.0.3 is a grammar for simple arithmetic expressions involving only addition and multiplication. As we observed, it is an ambiguous grammar because there exists an input string for which we can find more than one derivation tree. This ambiguity can be eliminated by writing an equivalent grammar which is not ambiguous:

G5:

```
1.    Expr → Expr + Term
2.    Expr → Term
3.    Term → Term * Factor
4.    Term → Factor
5.    Factor → ( Expr )
6.    Factor → var
7.    Factor → const
```

A derivation tree for the input string `var + var * var` is shown, below, in Figure 3.9. The student should verify that there is no other derivation tree for this input string, and that the grammar is not ambiguous. Also note that in any derivation tree using this grammar, subtrees correspond to subexpressions, according to the usual precedence rules. The derivation tree in Figure 3.9 indicates that the multiplication takes precedence over the addition. The left associativity rule would also be observed in a derivation tree for `var + var + var`.

Another example of ambiguity in programming languages is the conditional statement as defined by grammar G6:

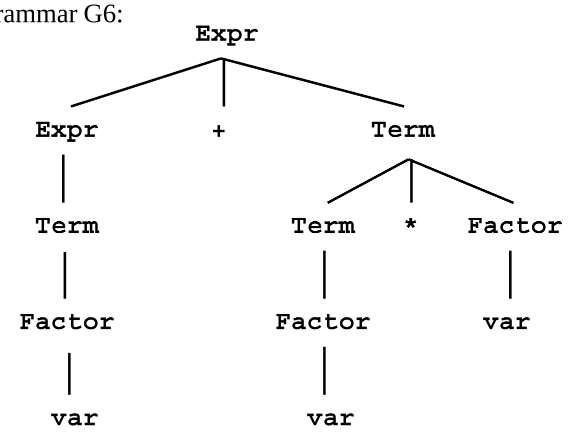


Figure 3.9 A Derivation Tree for `var + var * var` Using Grammar G5

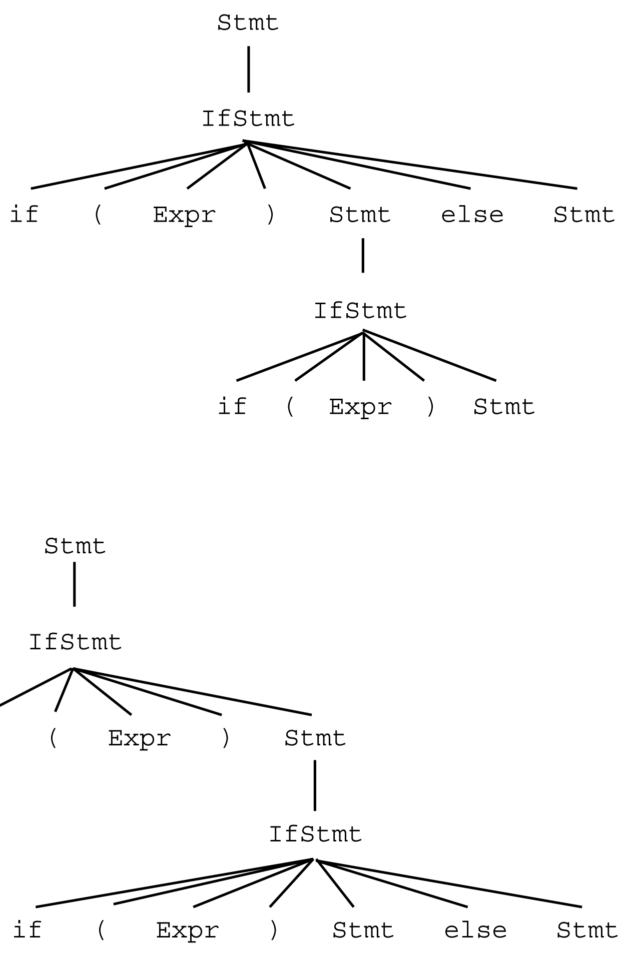


Figure 3.10 Two Different Derivation Trees for: `if ( Expr ) if ( Expr ) Stmt else Stmt`

G6:

```
1.    Stmt → IfStmt
2.    IfStmt → if ( Expr ) Stmt
3.    IfStmt → if ( Expr ) Stmt else Stmt
```

Think of grammar G6 as part of a larger grammar in which the nonterminal `Stmt` is completely defined. For the present example we will show derivation trees in which some of the leaves are left as nonterminals. Two different derivation trees for the input string `if (Expr) if (Expr) Stmt else Stmt` are shown, above, in Figure 3.10. In this grammar, an `Expr` is interpreted as False (0) or True (non-zero), and a `Stmt` is

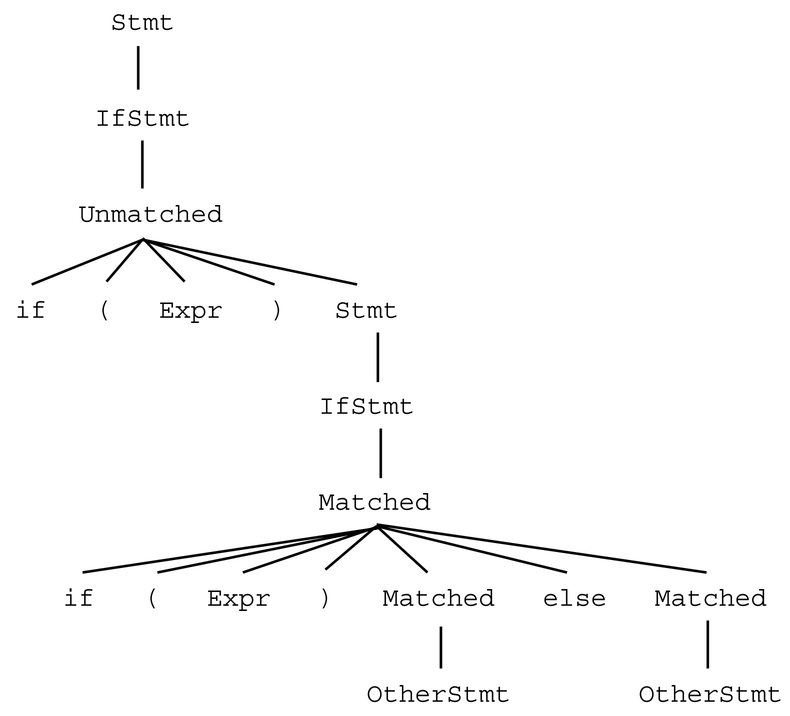


Figure 3.11 A Derivation Tree for `if ( Expr ) if ( Expr ) OtherStmt else OtherStmt` using Grammar G7

any statement, including `if` statements. This ambiguity is normally resolved by informing the programmer that *elses* always are associated with the closest previous unmatched *ifs*. Thus, the second derivation tree in Figure 3.10 corresponds to the correct interpretation. The grammar G6 can be rewritten with an equivalent grammar which is not ambiguous:

G7:

```
1.     Stmt → IfStmt
2.     IfStmt → Matched
3.     IfStmt → Unmatched
4.     Matched → if ( Expr ) Matched else Matched
5.     Matched → OtherStmt
6.     Unmatched → if ( Expr ) Stmt
7.     Unmatched → if ( Expr ) Matched else Unmatched
```

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, expression, for`, ...). A derivation tree for the string `if ( Expr ) if ( Expr ) OtherStmt else OtherStmt` is shown in Figure 3.11.

Exercises 3.1

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 Problem 2 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 → a A b          S → x B y
A → b A a          B → y B x
A → a              B → x
```

Which grammar in Section 3.1 is *isomorphic* to the grammar of Problem 4 in Section 3.0?

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

(a) `if ( Expr ) OtherStmt`

(b) `if ( Expr ) OtherStmt else if ( Expr ) OtherStmt`

(c) `if ( Expr ) if ( Expr ) OtherStmt else Stmt else OtherStmt`

(d) `if ( Expr ) if ( Expr ) if ( Expr ) Stmt else OtherStmt`

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, below, 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.

A ***parsing algorithm*** is one which solves the parsing problem for a particular class of grammars. A good parsing algorithm will be applicable to a large class of grammars and will accommodate the kinds of rewriting rules normally found in grammars for programming languages. For context-free grammars, there are two kinds of parsing algorithms – ***bottom up*** and ***top down***. These terms refer to the sequence in which the derivation tree of a correct input string is built. A parsing algorithm is needed in the syntax analysis phase of a compiler.

There are parsing algorithms which can be applied to any context-free grammar, employing a complete search strategy to find a parse of the input string. These algorithms are generally considered unacceptable since they are too slow; they cannot run in “polynomial time” (see Aho and Ullman [1972], for example).

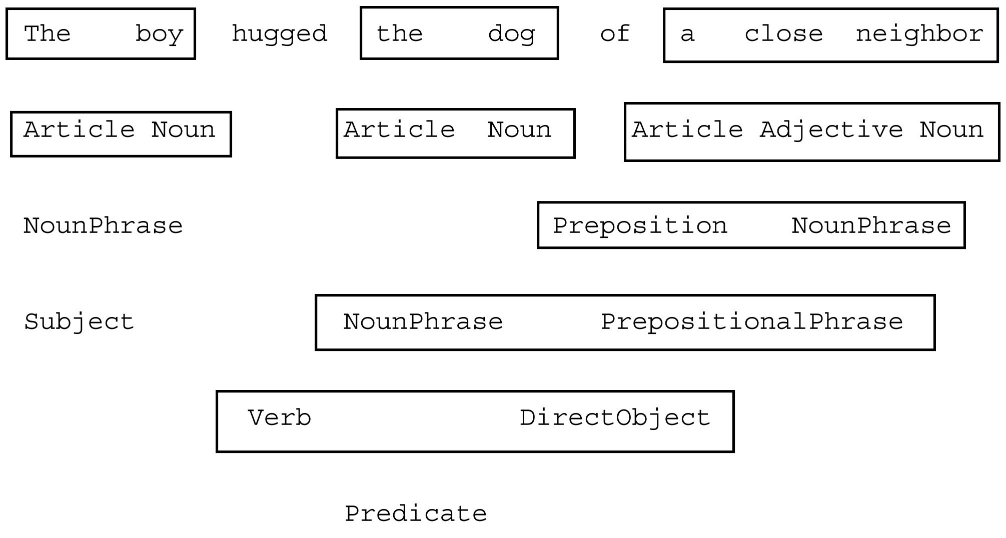


Figure 3.12 Diagram of an English sentence

3.3 Chapter Summary

Chapter 3, on ***syntax analysis***, serves as an introduction to the chapters on *parsing* (chapters 4 and 5). In order to understand what is meant by parsing and how to use parsing algorithms, we first introduce some theoretic and linguistic concepts and definitions.

We define *grammar* as a finite list of *rewriting rules* involving *terminal* and *nonterminal* symbols, and we classify grammars in a hierarchy according to complexity. As we impose more restrictions on the rewriting rules of a grammar, we arrive at grammars for less complex languages. The four classifications of grammars (and languages) are (0) *unrestricted*, (1) *context-sensitive*, (2) *context-free*, and (3) *right linear*. The context-free grammars will be most useful in the syntax analysis phase of the compiler, since they are used to specify programming languages.

We define *derivations* and *derivation trees* for context-free grammars, which show the structure of a derived string. We also define *ambiguous grammars* as those which permit two different derivation trees for the same input string.

Pushdown machines are defined as machines having an *infinite stack* and are shown to be the class of machines which corresponds to a subclass of context-free languages. We also define *pushdown translators* as pushdown machines with an output function, as this capability will be needed in compilers.

We take a careful look at ambiguities in programming languages, and see ways in which these ambiguities can be resolved. In particular, we look at grammars for simple arithmetic expressions and `if-else` statements.

Finally, we define the *parsing problem*: given a grammar and a string of input symbols, determine whether the string belongs to the language of the grammar, and, if so, determine its structure. We show that this problem corresponds exactly to the problem of diagramming an English sentence. The two major classifications of parsing algorithms are top-down, and bottom-up, corresponding to the sequence in which a derivation tree is built or traversed.