Chapter 2  Language Tools

In this chapter we get our first glimpse of actual programming language implementation. A good place to start with is the implementation of syntax analysis. Every language processor in our language processor classification performs some sort of syntax analysis (with the exception of the generator) . As we saw in the previous chapter, the structure of a programming language can be captured by rule sets called grammars. Therefore, we begin the chapter with a more detailed discussion of grammars including derivations and parse trees, and we will take a look at how grammars are used to specify the syntax of programming languages. We will use a grammar to specify our first programming language called Exp0.

Grammars are extremely useful for specifying the structure of programming languages, but what makes them even more powerful as a language specification tool is that we can easily convert grammars to programs that can recognize the structure of a programming language, i.e., we can convert grammars to programs that perform symbol grouping in an input stream according to the syntax rules of the language. Programs that recognize the structure of a programming language are called parsers. The algorithms that parsers use for grouping symbols in an input stream come in two flavors: (a) top-down or LL(k) parsing and (b) bottom-up or LR(k) parsing. We discuss examples of both of these approaches.

Programs that convert grammars into parsers are called parser generators. Here and for the remainder of the book we will be using a parser generator called ANTLR. ANTLR is a top-down parser generator that reads a grammar specification and produces a parser written in Java. We conclude the chapter with the construction of a couple of language processors. The first one is a reader for Exp0 that reads Exp0 programs and prints out information about the programs. The second language processor is an interpreter for the language Exp1 which is similar to Exp0 but with less restrictions.
2.1 Grammars

2.1.1 The Basics

Grammars are sets of rules that define the syntactic structure of a programming language as we have seen in the rules (1.1) through (1.4) in Chapter 1 (page 5). Rules in grammars specify how symbols, words, and phrases are combined to make up valid sentences, i.e., programs. When we talk about grammars in the context of specifying programming languages we usually mean context-free grammars as opposed to context-sensitive or regular grammars, for example. In context-free grammars the rules have a very specific form: there is only a single symbol on the left side of a rule and there are zero or more symbols on the right side of a rule. The fact that we allow a rule to have no symbols on the right means that a rule allows us to derive “nothing” or in more meaningful syntactical terms, that a rule with no symbols on the right side allows us to derive the empty string. We often write the empty string explicitly rather than writing a rule with no symbols on the right side for easier readability. For the remainder of the book we use the terms grammar and context-free grammar interchangeably.

Consider the following context-free grammar that specifies the syntactic structure of arithmetic expressions with the variables x, y, and z:

\[
\begin{align*}
\text{Sentence} & \rightarrow \text{Expression} \\
\text{Expression} & \rightarrow \text{Expression} \ast \text{Expression} \\
\text{Expression} & \rightarrow \text{Expression} - \text{Expression} \\
\text{Expression} & \rightarrow \text{Expression} \ast \text{Expression} \\
\text{Expression} & \rightarrow \text{Expression} / \text{Expression} \\
\text{Expression} & \rightarrow ( \text{Expression} ) \\
\text{Expression} & \rightarrow x \\
\text{Expression} & \rightarrow y \\
\text{Expression} & \rightarrow z
\end{align*}
\]

(2.1)

(2.2)

(2.3)

(2.4)

(2.5)

(2.6)

(2.7)

(2.8)

(2.9)

It is easy to see that this is a context-free grammar according to our discussion above; each rule has a single symbol on the left side and zero or more symbols on the right side. The grammar states that a sentence in this language is an expression and expressions can be add, subtract, multiply, or divide expressions as well as parenthesized expressions and variable names. Note that the parentheses are part of the language that we are defining, that is, the parentheses are part of the syntax of the language.
A grammar allows you to derive any valid sentence in the language that it defines by applying the rules to symbols appearing in the derivation until no further application is possible. In our case, we can derive the sentence $x + y$ from the symbol $Sentence$ in the above grammar as follows,

$$
\begin{align*}
Sentence & \rightarrow [\text{apply rule (1.1)}] \\
Expression & \rightarrow [\text{apply rule (1.2)}] \\
Expression + Expression & \rightarrow [\text{apply rule (1.7) to first Expression}] \\
x + Expression & \rightarrow [\text{apply rule (1.8)}] \\
x + y & \\
\end{align*}
$$

That means, the sentence $x + y$ is a valid sentence in this language.

The rules (1.2) through (1.9) in the grammar above all share the same left side. Therefore, given the symbol $Expression$ in a derivation we can apply any one of these rules to that symbol (each choice will derive a different program of course). In order to make that choice more explicit in a grammar, the rules are written with an explicit choice operator of the form '|'. The following is our grammar rewritten using this choice operator,

$$
\begin{align*}
Sentence & \rightarrow Expression \\
Expression & \rightarrow Expression + Expression \\
       & | Expression - Expression \\
       & | Expression \times Expression \\
       & | Expression / Expression \\
       & | ( Expression ) \\
       & | x \\
       & | y \\
       & | z
\end{align*}
$$

The choice operator states that an $Expression$ can either be an add expression, a subtract expression, a multiply expression, a divide expression, a parenthesized expression, or one of the variable names. Typically the choice operator makes grammars much less cluttered and therefore much easier to read.

We associate specific terminology with grammars. The rules are called productions and the symbol appearing on the left side of a production is called a non-terminal. Any symbol appearing in the grammar productions that is not a non-terminal, i.e., any symbol that does not appear on the left side of a production, is called a terminal. The terminals make up the syntax of a programming language. In our case, the terminal set includes the symbols $x$, $y$, and $z$ as well as the arithmetic symbols and the parentheses. None of these symbols appear on the left side of a production. Each grammar has one non-terminal that is
used to start all the derivations in that grammar, that non-terminal is called the start symbol. In our grammar the symbol Sentence is the start symbol. It is a convention that the start symbol is the non-terminal defined by the first rule in the grammar, that is, the start symbol is the symbol on the left side of the first rule of the grammar. We can summarize this,

A grammar consists of the following:

- A set of productions which are rules with a single symbol on the left and zero or more symbols on the right.
- The symbol on the left side of a production is called a non-terminal.
- Any symbol in the grammar that is not a non-terminal is called a terminal.
- We have a special symbol called the start-symbol.

2.1.2 Derivations

Given our grammar terminology, we can now be precise of what we mean by a derivation and valid sentence,

A derivation is a sequence of steps that begins with the start symbol and at each derivation step replaces a single non-terminal with the right side of a production that has that non-terminal on the left side. A valid sentence in the language of a grammar is a sequence of symbols arrived at through a derivation that contains only terminals.

Let us use the grammar (1.10) through (1.18) and show that the sentence

\[ x \ast (y + z) \]

is a valid sentence in the language of that grammar,

Sentence \[\text{[apply rule (1.10)]}\]
Expression \[\text{[apply rule (1.13)]}\]
Expression \ast Expression \[\text{[apply rule (1.16) to first Expression]}\]
x \ast Expression \[\text{[apply rule (1.15)]}\]
x \ast ( Expression ) \[\text{[apply rule (1.11)]}\]
x \ast ( Expression \ast Expression ) \[\text{[apply rule (1.17) to first Expression]}\]
x \ast ( y \ast Expression ) \[\text{[apply rule (1.18)]}\]
x \ast ( y \ast z )
Here we have consistently expanded the left-most non-terminal at each step of the derivation. This kind of derivation is called the left-most derivation. Consistently expanding the right-most non-terminal at each step of a derivation is called the right-most derivation,

\[
\begin{align*}
\text{Sentence} & \quad \text{[apply rule (1.10)]} \\
\text{Expression} & \quad \text{[apply rule (1.13)]} \\
\text{Expression} \ast \text{Expression} & \quad \text{[apply rule (1.15) to second Expression]} \\
\text{Expression} \ast ( \text{Expression} ) & \quad \text{[apply rule (1.11) to second Expression]} \\
\text{Expression} \ast ( \text{Expression} \ast \text{Expression} ) & \quad \text{[apply rule (1.18) to third Expression]} \\
\text{Expression} \ast ( \text{Expression} \ast \text{z} ) & \quad \text{[apply rule (1.17) to second Expression]} \\
x \ast ( y \ast \text{z} ) & \quad \text{[apply rule (1.16)]}
\end{align*}
\]

If a sentence is valid then it does not matter which derivation we chose to derive it from the start symbol, both derivation techniques will derive the same sentence. Later on we will see that we classify parsing algorithms according to the derivation that they construct for a program.

#### 2.1.3 Parse Trees

Derivations can be represented as tree structures. Consider the left-most derivation of the sentence,

\[
x \ast y \ast z
\]

using the grammar (1.10) through (1.18),

\[
\begin{align*}
\text{Sentence} & \quad \text{[apply rule (1.10)]} \\
\text{Expression} & \quad \text{[apply rule (1.11)]} \\
\text{Expression} \ast \text{Expression} & \quad \text{[apply rule (1.16)]} \\
x \ast \text{Expression} & \quad \text{[apply rule (1.13)]} \\
x \ast \text{Expression} \ast \text{Expression} & \quad \text{[apply rule (1.17)]} \\
x \ast y \ast \text{Expression} & \quad \text{[apply rule (1.18)]} \\
x \ast y \ast z & \quad \text{[apply rule (1.16)]}
\end{align*}
\]

We construct a parse tree for this derivation by first making \textit{Sentence} the root node of the tree and then for every non-terminal that we replace in the derivation we write the right side of the rule that we use to replace the non-terminal underneath it and connect all the symbols that we just placed underneath the non-terminal to the non-terminal symbol itself. Figure 1.1 shows the parse tree constructed in this way for the derivation above. The leaves of the parse tree spell out the sentence that we wanted to derive and we highlighted this by placing the derived sentence into a box at the bottom of the figure. We can now say the following,
A parse tree is a tree based representation of a derivation in a grammar.

An interesting and very desirable characteristic of parse trees is that they make the grouping of the symbols in an input stream structurally explicit. In Figure 1.1 we can see that $y \times z$ are grouped together as an expression and this expression together with the symbols $x$ and $+$ form another expression group. This hierarchical grouping of symbols, words, and phrases makes language processing much easier compared to processing textual representations of programs. We will see that the job of a parser is to construct such parse trees.

Another observation is that the same parse tree is obtained regardless whether we use a left-most or right-most derivation of our sentence. Try to prove to yourself that a right-most derivation of our sentence gives rise to the same parse tree (the first three steps in the right-most derivation should be the same as in the left-most derivation above).
2.1.4 An Example: Our Exp0 Language

Our first programming language that we consider here is a calculator language that allows you to store and print values. Let us call it Exp0; Exp for expression language. In order to keep it simple we only allow three variable names (x, y, and z) and our numbers are limited to single digits. The expressions in this language are in prefix format, e.g., the expression (+ x 1) means add the value 1 to the the value stored in variable x. If you know Lisp then this should look very familiar; in Lisp all expressions as well as statements are given in a prefix format. For the sake of simplicity we also only allow additive expressions, that is, addition and subtraction. To print a value to the terminal we write the symbol p followed by an expression. For example, the statement p (+ x 1) increments the value stored in x by one and then prints that value to the terminal. We use the symbol s followed by a variable name followed by an expression to store the value of the expression in the variable. Here, the statement s y (- 9 3) stores the value six in the variable y.

Here then is the grammar that defines the syntax of Exp0,

\[
\begin{align*}
Progs & \rightarrow \text{Stmt} \text{StmtList} \\
\text{StmtList} & \rightarrow \text{Stmt} \text{StmtList} \quad | \quad "" \\
\text{Stmt} & \rightarrow p \ Exp \ ; \\
& \quad | \quad s \ Var \ Exp \ ; \\
\text{Exp} & \rightarrow + \ Exp \ Exp \quad | \quad - \ Exp \ Exp \\
& \quad | \quad ( \ Exp \ ) \\
& \quad | \quad \text{Var} \\
& \quad | \quad \text{Num} \\
\text{Var} & \rightarrow x \\
& \quad | \quad y \\
& \quad | \quad z \\
\text{Num} & \rightarrow 0 \\
& \quad | \quad 1 \\
& \quad | \quad 2 \\
& \quad | \quad 3 \\
& \quad | \quad 4 \\
& \quad | \quad 5
\end{align*}
\]

Confidential and Copyrighted - © 2015 Franklin, Beedle and Associates - Draft September 14, 2015
With the exception of the first three rules the grammar looks pretty straightforward. For example, the rules (1.22) and (1.23) define the structure of print and store statements, respectively. Note the semicolon that terminates these statements. The rules (1.24) through (1.28) specify the structure of possibly parenthesized prefix expressions. Similar to the grammar for arithmetic expressions above we let variable names and numbers also be considered expressions here. Finally, the last two sets of rules, (1.29) through (1.31) and (1.32) through (1.41), define our variable names and our numbers.

Now, let us take a look at the first three rules, in particular, let us take a look at rules (1.20) and (1.21). These two rules specify the structure of statement lists. Rule (1.20) says that a statement list is a statement followed by a statement list. In other words, this is a recursive rule and if we apply this rule repeatedly we can derive a list of statements. Consider the following derivation,

\[
\begin{align*}
\text{StmtList} & \quad [\text{apply rule (1.20)}] \\
\text{Stmt StmtList} & \quad [\text{apply rule (1.20)}] \\
\text{Stmt Stmt StmtList} & \quad [\text{apply rule (1.20)}] \\
\text{Stmt Stmt Stmt StmtList} & \quad [\text{apply rule (1.20)}] \\
\vdots
\end{align*}
\]

In this context rule (1.21) comes in quite handy because it expresses the fact that a statement list can be empty. We can consider this rule the termination case of our recursion on statement lists,

\[
\begin{align*}
\text{StmtList} & \quad [\text{apply rule (1.20)}] \\
\text{Stmt StmtList} & \quad [\text{apply rule (1.20)}] \\
\text{Stmt Stmt StmtList} & \quad [\text{apply rule (1.20)}] \\
\text{Stmt Stmt Stmt StmtList} & \quad [\text{apply rule (1.20)}] \\
\text{Stmt Stmt Stmt} & \quad [\text{apply rule (1.21)}] \\
\text{ Stmt Stmt} & \quad ...
\end{align*}
\]

Here we derived a list of three statements and we can use the rules for the non-terminal \textit{Stmt} to derive an actual program. Notice that the rules for statement lists allows us to derive the empty list of statements as in this derivation,

\[
\begin{align*}
\text{StmtList} & \quad [\text{apply rule (1.21)}] \\
\text{ Stmt} & \quad ""
\end{align*}
\]

That means, a statement list is allowed to contain no statements at all. This brings us to the first rule (1.19) of our grammar. This rule states that a program
is a statement followed by a possibly empty statement list. In other words, a program must have at least one statement in it; a program is not allowed to be the empty list of statements.

Let us use the grammar (1.19) through (1.41) to derive the program

$$s \ x \ 1; \ p \ (+ \ x \ 1);$$

In this program we first store the value one in variable $x$ and then we print out the sum of the value stored in $x$ and the value one. Our left-most derivation starts with the start symbol $Prog$, which is the first non-terminal defined in the
grammar,

```
Prog          [apply rule (1.19)]
Stmt StmtList [apply rule (1.23)]
 s Var Exp ; StmtList [apply rule (1.29)]
 s x Exp ; StmtList  [apply rule (1.28)]
 s x Num ; StmtList  [apply rule (1.33)]
 s x 1 ; StmtList    [apply rule (1.20)]
 s x 1 ; Stmt StmtList [apply rule (1.23)]
 s x 1 ; p Exp ; StmtList [apply rule (1.26)]
 s x 1 ; p ( Exp ) ; StmtList [apply rule (1.24)]
 s x 1 ; p ( + Exp Exp ) ; StmtList [apply rule (1.27)]
 s x 1 ; p ( + Var Exp ) ; StmtList [apply rule (1.29)]
 s x 1 ; p ( + x Exp ) ; StmtList [apply rule (1.28)]
 s x 1 ; p ( + x Num ) ; StmtList [apply rule (1.33)]
 s x 1 ; p ( + x 1 ) ; StmtList [apply rule (1.21)]
 s x 1 ; p ( + x 1 ) ;
```

The interesting part of this derivation is that as soon as we have derived a statement fully we use the `StmtList` rule to either generate another `Stmt` non-terminal in order to derive another statement or we terminate the derivation by deriving the empty string from `StmtList`. The parse tree for this derivation appears in Figure 1.2. You should convince yourself that this is indeed the parse tree for the above derivation.

## 2.2 Parsers

In the previous sections we used our innate intelligence to come up with derivations that prove that some sentence is valid in the language of a grammar. Here we investigate if constructing a derivation for a sentence in a particular grammar can be performed algorithmically. The answer is of course a resounding yes and the programs that perform the construction of derivations are called parsers. We begin our discussion with the top-down parsing algorithm.

### 2.2.1 Parsing Algorithms

#### Top-Down Parsing

It is highly likely that when you constructed the derivation of a particular valid sentence you first scanned the structure of the sentence itself and then you scanned the right sides of the grammar rules. Once you had an overview of
the structure the right sides of the grammar rules you most likely picked a rule that matched some structure of the sentence and applied it to a non-terminal in your derivation. You kept doing this until you derived the terminals that made up the sentence.

Top-down parsing mimics this approach. However, in order to make this workable we need to extend grammars with an additional data structure called lookahead sets. Our Exp0 grammar extended with lookahead sets looks like this,

\[
\begin{align*}
    Prog & \to \{p, s\} Stmt StmtList \quad (2.42) \\
    StmtList & \to \{p, s\} Stmt StmtList \quad (2.43) \\
    & | \{""\} "" \quad (2.44) \\
    Stmt & \to \{p\} p \, Exp \, ; \quad (2.45) \\
    & | \{s\} s \, Var \, Exp \, ; \quad (2.46) \\
    Exp & \to \{+\} + \, Exp \, Exp \quad (2.47) \\
    & | \{-\} - \, Exp \, Exp \quad (2.48) \\
    & | \{\} (\, Exp \, ) \quad (2.49) \\
    & | \{x, y, z\} Var \quad (2.50) \\
    & | \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} Num \quad (2.51) \\
    Var & \to \{x\} x \quad (2.52) \\
    & | \{y\} y \quad (2.53) \\
    & | \{z\} z \quad (2.54) \\
    Num & \to \{0\} 0 \quad (2.55) \\
    & | \{1\} 1 \quad (2.56) \\
    & | \{2\} 2 \quad (2.57) \\
    & | \{3\} 3 \quad (2.58) \\
    & | \{4\} 4 \quad (2.59) \\
    & | \{5\} 5 \quad (2.60) \\
    & | \{6\} 6 \quad (2.61) \\
    & | \{7\} 7 \quad (2.62) \\
    & | \{8\} 8 \quad (2.63) \\
    & | \{9\} 9 \quad (2.64)
\end{align*}
\]

Here the lookahead sets appear between the curly braces for each rule.

From the point of view of top-down parsing the lookahead sets summarize the structure of the right side of each grammar rules and they are used in conjunction with a lookahead pointer to make choices between competing rules during the construction of a derivation. The lookahead pointer points to the
current symbol in the input stream. Consider the following input stream:

\[ p + 1 2 ; \text{\texttt{eof}} \]

The angle brackets around the \( p \) symbol represent the lookahead pointer. Now consider the left-most derivation of the program in the input stream using the extended grammar (1.42) through (1.64). The input stream is the structure on the left below and the derivation is the structure within the square brackets. As before, the symbol \( p \) is our current symbol and our derivation starts with the start symbol \( \text{Prog} \). We can observe that the lookahead symbol matches the lookahead set for the rule defining \( \text{Prog} \), rule (1.42), and therefore we can apply this rule,

\[
\begin{align*}
\text{<p> + 1 2 ; \text{\texttt{eof}}} & \quad \text{[Pro]} \\
\text{<p> + 1 2 ; \text{\texttt{eof}}} & \quad \text{[Stmt StmtList]}
\end{align*}
\]

Here we did not advance the lookahead pointer because the right side of the rule did not contain any terminal symbols. We are performing a left-most derivation, therefore, the next symbol we need to expand is \( \text{Stmt} \). Now, according to the grammar we have a choice between a print and a store statement, rules (1.45) and (1.46), respectively. The lookahead pointer points to a \( p \) symbol and we need to pick the \( \text{Stmt} \) rule whose lookahead set matches the lookahead symbol. In this case we must pick the rule (1.45) with the lookahead set \{\( p \)\}. We replace the non-terminal \( \text{Stmt} \) with the right side of rule (1.45) and then move the lookahead pointer to the next symbol because the rule begins with a terminal symbol that matches the lookahead pointer,

\[
\begin{align*}
\text{<p> + 1 2 ; \text{\texttt{eof}}} & \quad \text{[Stmt StmtList]} \\
p + <1> 2 ; \text{\texttt{eof}} & \quad \text{[p Exp ; StmtList]}
\end{align*}
\]

The next non-terminal we have to expand in this derivation is \( \text{Exp} \) and again we have a number of rules to choose from. Precisely, we have a choice of rules (1.47) through (1.51). However, only the lookahead set of rule (1.47) matches the lookahead pointer and therefore we will choose this rule to expand \( \text{Exp} \) and move the lookahead pointer to the next symbol,

\[
\begin{align*}
p + <1> 2 ; \text{\texttt{eof}} & \quad \text{[p Exp ; StmtList]} \\
p + <1> 2 ; \text{\texttt{eof}} & \quad \text{[p Num Exp ; StmtList]}
\end{align*}
\]

Our lookahead pointer now points to the symbol 1 which matches the lookahead set for \( \text{Exp} \) rule (1.51),

\[
\begin{align*}
p + <1> 2 ; \text{\texttt{eof}} & \quad \text{[p Num Exp ; StmtList]} \\
p + <1> 2 ; \text{\texttt{eof}} & \quad \text{[p Num Exp ; StmtList]}
\end{align*}
\]

We did not advance the lookahead pointer because the right side of the rule did not contain any terminal symbols. The next derivation step replaces the non-terminal \( \text{Num} \). Here again we have many choices but the lookahead pointer
tells us that we should use rule (1.56),

\[
\begin{align*}
p + <1> 2 ; \text{\texttt{eof}} & [p + \text{Num} \text{ Exp} ; \text{StmtList}] \\
p + 1 <2> ; \text{\texttt{eof}} & [p + 1 \text{ Exp} ; \text{StmtList}]
\end{align*}
\]

Expanding the remaining non-terminal \texttt{Exp} with rules (1.51) and (1.57), respectively, gives us the following derivation steps,

\[
\begin{align*}
p + 1 <2> ; \text{\texttt{eof}} & [p + 1 \text{ Exp} ; \text{StmtList}] \\
p + 1 <2> ; \text{\texttt{eof}} & [p + 1 \text{ Num} ; \text{StmtList}] \\
p + 1 2 <;> \text{\texttt{eof}} & [p + 1 2 ; \text{StmtList}]
\end{align*}
\]

Notice that in the derivation everything to the left of the semicolon consists now only of terminal symbols. Therefore, we can go ahead and move the lookahead pointer to the next symbol. Another way of looking at this is that the semicolon is matched as part of the original expansion of \texttt{Stmt}, this gives us,

\[
\begin{align*}
p + 1 2 <;> \text{\texttt{eof}} & [p + 1 2 ; \text{StmtList}] \\
p + 1 2 ; <\text{\texttt{eof}}> & [p + 1 2 ; \text{StmtList}]
\end{align*}
\]

The lookahead pointer is now pointing to the end of file marker. The last non-terminal we have to expand is \texttt{StmtList}. The rules for \texttt{StmtList}, (1.43) and (1.44), have the lookahead sets \{p, s\} and \{"\"\"\\}, respectively. Certainly the lookahead set of the first rule does not match the lookahead pointer, but we can use the second rule because an empty string lookahead set will match any symbol pointed to by the lookahead pointer,

\[
\begin{align*}
p + 1 2 ; <\text{\texttt{eof}}> & [p + 1 2 ; \text{StmtList}] \\
p + 1 2 ; <\text{\texttt{eof}}> & [p + 1 2 ;]
\end{align*}
\]

At this point the lookahead pointer points to the \text{\texttt{eof}} symbol, there are no non-terminals left in the derivation, and the derived sentence matches the program in the input stream: We say that we successfully parsed the input stream. Here is the whole derivation without any interruptions,

\[
\begin{align*}
<p> + 1 2 ; \text{\texttt{eof}} & [\text{Prog}] \\
<p> + 1 2 ; \text{\texttt{eof}} & [\text{Stmt} \text{ StmtList}] \\
p <\leftrightarrow> 1 2 ; \text{\texttt{eof}} & [p \text{ Exp} ; \text{StmtList}] \\
p + <1> 2 ; \text{\texttt{eof}} & [p + \text{ Exp} \text{ Exp} ; \text{StmtList}] \\
p + <1> 2 ; \text{\texttt{eof}} & [p + \text{ Num} \text{ Exp} ; \text{StmtList}] \\
p + 1 <2> ; \text{\texttt{eof}} & [p + 1 \text{ Exp} ; \text{StmtList}] \\
p + 1 <2> ; \text{\texttt{eof}} & [p + 1 \text{ Num} ; \text{StmtList}] \\
p + 1 2 <;> \text{\texttt{eof}} & [p + 1 2 ; \text{StmtList}] \\
p + 1 2 ; <\text{\texttt{eof}}> & [p + 1 2 ; \text{StmtList}] \\
p + 1 2 ; <\text{\texttt{eof}}> & [p + 1 2 ;]
\end{align*}
\]
The parse tree for this derivation is shown in Figure 1.3. This is called top-down parsing because we constructed the parse tree starting at the root of the parse tree (Prog) and then kept expanding it until we were left with a tree that only has terminals at its leaf nodes. Again, you should convince yourself that this parse tree represents the derivation above.

Another name for this approach to parsing is LL(1) parsing. The first L stands for the fact that we are reading the input stream from left to right. The second L indicates that we are performing a left-most derivation, and (1) means that we are using a single lookahead symbol.

Top-down parsing means building parse trees starting with the grammar start symbol. Another name for it is LL(1) parsing which means reading from the (L)eft constructing the (L)eft-most derivation using (1) lookahead symbol. Grammars that allow us to do LL(1) parsing are called LL(1) grammars.

The key to LL(1) parsing are the lookahead sets.

---

1This will become clearer when we study the actual parsing algorithm.
2.2. Parsers

Figure 2.4: Algorithm for computing the lookahead sets of a context-free grammar $G$.

Lookahead Sets

You probably noticed that the introduction of the lookahead pointer and the lookahead sets made constructing derivations very mechanical. There is no guess work in terms of which rule to apply when in order to construct a derivation. This is just what we need in order to have machines do this.

The key to this all was the idea of a lookahead set for each rule. Let us take a look at the construction of the lookahead sets. Computing lookahead sets is based on a recursive algorithm in two parts. The first part of the algorithm which is non-recursive is shown in Figure 1.4. The workings of this part are best explained with an example. Consider the following grammar,

$$
\begin{align*}
Exp & \rightarrow + Exp Exp \\
Exp & \rightarrow x \\
Exp & \rightarrow y 
\end{align*}
$$

Rather than using the choice operator we have written this grammar as an explicit list of rules. Let us pass this grammar as grammar $G$ to the function $ComputeLookaheadSets$ shown in Figure 1.4. On line 5 the algorithm starts by initializing a second grammar $G'$ as an empty list of rules. Then we get to the core of the algorithm. On line 6 we start a loop that iterates over the rules in grammar $G$. Let us take the first rule in our grammar: $Exp \rightarrow + Exp Exp$. It
certainly is of form $A \rightarrow RuleBody$ the pattern given in the loop statement with $A$ equal to $Exp$ and $RuleBody$ equal to $+ Exp Exp$. Moving into the body of the loop on line 7 we look at the first symbol $S$ of $RuleBody$. In our case $S$ is equal to the terminal $+$. Since the first symbol is a terminal symbol we can rewrite our grammar rule as

$$Exp \rightarrow \{+\} + Exp \ Exp$$

and append it to the grammar $G'$ according to lines 10 and 11. We now keep iterating over the remaining rules (1.66) and (1.67) in grammar $G$. Both of these rules have terminal symbols as the first symbols on their right sides and therefore the computations of our algorithm are similar to what we just went through for the first rule. Once we iterated through all three rules in $G$ our grammar $G'$ looks like this,

$$Exp \rightarrow \{+\} + Exp \ Exp$$

$$Exp \rightarrow \{x\} x$$

$$Exp \rightarrow \{y\} y$$

This is our original grammar extended with the lookahead sets and the lookahead sets themselves consist of the first terminal symbol of the right side of each rule.

Now let us investigate the recursive aspect of the algorithm. For this we rewrite our grammar as,

$$Exp \rightarrow + Exp \ Exp$$

$$Exp \rightarrow Var$$

$$Var \rightarrow x$$

$$Var \rightarrow y$$

Here we introduced a non-terminal for the variable symbols. As before, let us pass this grammar as grammar $G$ to the function $ComputeLookaheadSets$ shown in Figure 1.4. The first rule in this grammar is the same rule as in our original grammar, therefore, we already know that the algorithm will append the rule

$$Exp \rightarrow \{+\} + Exp \ Exp$$

to $G'$. Processing the second rule, (1.72),

$$Exp \rightarrow Var$$

turns out to be different. The first symbol on the right side of this rule is the non-terminal $Var$ and according to line 13 of our algorithm in Figure 1.4 we call the
2.2. Parsers

function LookaheadSet(N, G) returns L

// N is a non-terminal in G
// G is a context-free grammar
// L is a lookahead set

begin

let L be the empty set
for each rule N → RuleBody in G do

let Q be the first symbol in RuleBody
if Q is the empty string then

report an error and abort

else if Q is a terminal symbol then

let L := L ∪ {Q}

else if Q is a non-terminal symbol then

let L := L ∪ LookaheadSet(Q, G)

end if
end do

return L

end

Figure 2.5: Algorithm for computing the lookahead set L for a non-terminal N in some grammar G.

function LookaheadSet to which we pass the symbol Var and our grammar G. The function LookaheadSet is given in Figure 1.5.

The workings of the function LookaheadSet are very similar to the function ComputeLookaheadSet with one major difference, instead of iterating over all rules of the grammar G it will only iterate over rules whose left side is the non-terminal N which is passed into the function. In our case this means that it will only iterate over rules whose left side is the non-terminal Var, i.e., it will only iterate over the last two rules in the grammar G.

In Figure 1.5 the algorithm begins on line 6 by initializing L as an empty set where L is the lookahead set to be constructed. Then we iterate over rules that begin with Var. The first rule we have is rule (1.73),

\[ Var \rightarrow x \]

The right side of this rule is the terminal x and therefore we add it to the set L according to lines 11 and 12. Continuing we have rule (1.74),

\[ Var \rightarrow y \]

Again, the right side is a terminal and we add the symbol y to L. These are all the rules that have Var as the left side and therefore we are done with this
function and we return the set,

\[ L = \{x, y\} \]

from this function. Going back to line 13 in Figure 1.4, we now have a lookahead set for rule (1.72) and therefore we can add the rule,

\[ Exp \to \{x, y\} Var \]

to \(G'\) according to line 14. Iterating over the remaining two rules, (1.73) and (1.74) in the grammar is straightforward because the right sides consist of terminal symbols. The grammar \(G'\) computed by ComputeLookaheadSets is then,

\[
\begin{align*}
Exp & \to \{+\} + Exp \ Exp \\
Exp & \to \{x, y\} \ Var \\
\text{Term} & \to \ Var \\
\text{Term} & \to \ Bin \\
\text{Var} & \to \{x\} x \\
\text{Var} & \to \{y\} y \\
\text{Bin} & \to \ 0 \\
\text{Bin} & \to \ 1 \\
\end{align*}
\]

The lookahead set for the second rule is interesting in the sense that the lookahead set for the non-terminal \(Var\) consists of the possible variable names.

Let us take it one step further with the following grammar,

\[
\begin{align*}
Exp & \to \ + \ Exp \ Exp \\
Exp & \to \ \text{Term} \\
\text{Term} & \to \ Var \\
\text{Term} & \to \ Bin \\
\text{Var} & \to \{x\} x \\
\text{Var} & \to \{y\} y \\
\end{align*}
\]

Expressions in the language of this grammar consist of sums of the variables \(x\) and \(y\), and the digits \(0\) and \(1\). Tracing through the functions in Figure 1.4 and Figure 1.5 in order to compute the lookahead sets you should obtain the following grammar,

\[
\begin{align*}
Exp & \to \ {+} + Exp \ Exp \\
Exp & \to \ \{x, y, 0, 1\} \ Term \\
\text{Term} & \to \ \{x, y\} \ Var \\
\text{Term} & \to \ \{0, 1\} \ Bin \\
\text{Var} & \to \ \{x\} x \\
\end{align*}
\]
\[ \text{Var} \rightarrow \{y\} y \quad (2.92) \]
\[ \text{Bin} \rightarrow \{0\} 0 \quad (2.93) \]
\[ \text{Bin} \rightarrow \{1\} 1 \quad (2.94) \]

When you perform the trace you will find that the function \textit{LookaheadSet} will recurse in order to compute the lookahead set,

\[ L = \{x, y, 0, 1\} \]

for rule,

\[ \text{Exp} \rightarrow \text{Term} \]

Try to convince yourself that that is really the fact!

There are grammars for which it is impossible to construct lookahead sets. Consider the following grammar that is a slight modification of the grammar given in (1.65) through (1.67),

\[ \text{Exp} \rightarrow \text{Exp} + \text{Exp} \quad (2.95) \]
\[ \text{Exp} \rightarrow x \quad (2.96) \]
\[ \text{Exp} \rightarrow y \quad (2.97) \]

Our lookahead set algorithm in Figure 1.4 and Figure 1.5 will recurse indefinitely on the first rule of this grammar (convinces yourself!). The indefinite recursion is due to the fact that the first non-terminal on the right side of the rule is exactly the same non-terminal on the left side of the rule. This is called left-recursion and grammars that exhibit left-recursion in their rules cannot be used for LL(1) parsing. We say,

\textit{Left-recursive grammars are not LL(1).}

You have to be careful, left-recursion is not always immediately apparent. Consider the following grammar with mutually recursive rules,

\[ \text{Exp} \rightarrow \text{Term} \ast \text{Var} \quad (2.98) \]
\[ \text{Exp} \rightarrow \text{Var} \quad (2.99) \]
\[ \text{Term} \rightarrow \text{Exp} \ast \text{Var} \quad (2.100) \]
\[ \text{Term} \rightarrow \text{Var} \quad (2.101) \]
\[ \text{Var} \rightarrow x \quad (2.102) \]
\[ \text{Var} \rightarrow y \quad (2.103) \]

None of the rules in this grammar is immediately left-recursive. However, the mutual left-recursion of the first and third rule is problematic and again our
lookahead set algorithm will recurse indefinitely showing that this grammar is not LL(1). Later on we will see techniques that allow us to rewrite left-recursive grammars into grammars that are LL(1).

There is another class of grammars that is not considered to be LL(1); grammars where the prefixes of the right sides of rules overlap. Consider this grammar,

\[
\begin{align*}
\text{Exp} & \rightarrow + \text{Exp Exp} \\
\text{Exp} & \rightarrow \text{Term} \\
\text{Term} & \rightarrow \text{Var} \{ \text{Exp} \} \\
\text{Term} & \rightarrow \text{Var} \\
\text{Var} & \rightarrow x \\
\text{Var} & \rightarrow y
\end{align*}
\]

Notice that in this grammar the two rules defining the non-terminal Term both start with the non-terminal Var; we say that the two rules share a prefix. This is a problem because both rules will have the same lookahead set which prevents LL(1) parsing from being able to choose a rule unambiguously. We say,

\textit{Grammars that have rules defining the same non-terminal and that share a common prefix are not LL(1).}

Turns out there is an easy fix. We can rewrite the grammar above as follows,

\[
\begin{align*}
\text{Exp} & \rightarrow + \text{Exp Exp} \\
\text{Exp} & \rightarrow \text{Term} \\
\text{Term} & \rightarrow \text{Var Array} \\
\text{Array} & \rightarrow \{ \text{Exp} \} \\
\text{Array} & \rightarrow \"\" \\
\text{Var} & \rightarrow x \\
\text{Var} & \rightarrow y
\end{align*}
\]

This technique is called rule factoring and is often used to turn non-LL(1) grammars into LL(1) grammars.

There is one additional class of grammars that are not considered LL(1) according to our lookahead set computation. These are grammars that have rules with right sides that start with non-terminals deriving the empty string. Consider the following grammar,

\[
\begin{align*}
A & \rightarrow B a \\
& | \ c \\
B & \rightarrow b \\
& | \"\"
\end{align*}
\]
Without the test on line 9 and 10 our lookahead set algorithm in Figure 1.5 would compute the following lookahead sets for the grammar above,

\[
A \rightarrow \{b, "\"\} B a \quad \text{(2.121)}
\]
\[
| \{c\} c \quad \text{(2.122)}
\]
\[
B \rightarrow \{b\} b \quad \text{(2.123)}
\]
\[
| \{"\"\} "" \quad \text{(2.124)}
\]

Here the empty string as a lookahead symbol for the first rule does not make any sense because the right side of that rule is not the empty string and therefore we reject this grammar. We could of course extend out lookahead set algorithm to accommodate this empty string and compute the lookahead set for the first rule as \{a, b\} but the algorithm that does this in the general case is too complicated for our purposes considering that there is an easy rewrite of the grammar,

\[
A \rightarrow b a \quad \text{(2.125)}
\]
\[
| a \quad \text{(2.126)}
\]
\[
| c \quad \text{(2.127)}
\]

Grammars with rules whose right sides start with non-terminals that derive the empty string are said to contain nullable prefixes. Therefore, according to our definition of lookahead sets we say that,

\[\text{Grammars with nullable prefixes are not LL(1).}\]

The fact that we can have multiple grammars that define a language is interesting in itself. The fact that some of these grammars are LL(1) and some are not is even more interesting and leads to the following observation,

\[\text{A language is called LL(1) if it has at least one LL(1) grammar.}\]

### A Top-Down Parsing Algorithm

Now that we know how to construct grammars that have lookahead sets we can examine the top-down parsing algorithm itself. Figure 1.6 shows the algorithm. The algorithm takes an input stream and a context-free grammar extended with lookahead sets as inputs and returns \text{true} or \text{false} depending on whether the program in the input stream is a valid sentence in language of the grammar or not.\footnote{Once you have gotten to know this algorithm a little bit better you will see that it is easy to extend this algorithm to either return a derivation or a parse tree.} Internally the algorithm uses a stack to keep track of the derivations. Probably the best way to get a handle on how this algorithm works is by working
Figure 2.6: The top-down parsing algorithm given an input stream $I$ and a grammar $G'$. 

```plaintext
function TopDownParse($I, G'$) returns true or false
// $I$ is an input stream
// $G'$ is an LL(1) grammar extended with lookahead sets
begin
    let $S$ be an empty stack
    push \eos onto $S$
    push the start symbol of $G'$ onto $S$
    let $P$ be the lookahead pointer to the first symbol in $I$
while TopOfStack($S$) $\neq$ \eos do
    if TopOfStack($S$) is the non-terminal $A$ then
        find rule $A \rightarrow L$ RuleBody in $G'$ such that $P \in L$ or $L = \{\epsilon\}$
        if no such rule exists then
            return false
        else
            pop $S$
            push symbols of RuleBody in reverse order onto $S$
        end if
    else if TopOfStack($S$) is the terminal $a$ then
        if $a = P$ then
            pop $S$
            move $P$ to the next symbol
        else
            return false
        end if
    end if
end do
if $P = \eos$ then
    return true
else
    return false
end if
end
```
Through an example. Let’s use our Exp0 language and the same sentence that we used in the initial example of top-down parsing in order to see how this algorithm parses that sentence. We will use the Exp0 grammar extended with lookahead sets (1.42) through (1.64) on page 11 as our input grammar $G'$ and we will use the input stream

```
p + 1 2 ; \eos
```

as our input stream $I$. Figure 1.7 is a trace of the algorithm parsing the input stream $I$ using grammar $G'$.

We trace the state of the computation using the step numbers given in Step column of the table:

**Step 1:** Everything is initialized according to the lines 5 through 8 in the algorithm. The stack is initialized with the ‘end of stack’ symbol \eos, the

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack (S)</th>
<th>Input Stream (I)</th>
<th>Rule</th>
<th>Rule No.</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\eos Prog</td>
<td>&lt;p&gt; + 1 2 ; \eos</td>
<td>$\text{Prog} \to (p,s) \text{StmtList}$</td>
<td>(2.42)</td>
<td>10-17</td>
</tr>
<tr>
<td>2</td>
<td>\eos StmtList ; Stmt</td>
<td>&lt;p&gt; + 1 2 ; \eos</td>
<td>$\text{Stmt} \to {p} \text{p Exp ;}$</td>
<td>(2.45)</td>
<td>10-17</td>
</tr>
<tr>
<td>3</td>
<td>\eos StmtList ; Exp p</td>
<td>&lt;p&gt; + 1 2 ; \eos</td>
<td>$\text{StmtList} \to ({p}) \text{p Exp ;}$</td>
<td>(2.46)</td>
<td>10-17</td>
</tr>
<tr>
<td>4</td>
<td>\eos StmtList ; Exp</td>
<td>p &lt;+&gt; 1 2 ; \eos</td>
<td>$\text{Exp} \to {+} \text{Exp Exp}$</td>
<td>(2.47)</td>
<td>10-17</td>
</tr>
<tr>
<td>5</td>
<td>\eos StmtList ; Exp Exp +</td>
<td>p &lt;+&gt; 1 2 ; \eos</td>
<td>$\text{Exp} \to {p} \text{p Exp ;}$</td>
<td>(2.48)</td>
<td>10-17</td>
</tr>
<tr>
<td>6</td>
<td>\eos StmtList ; Exp Exp</td>
<td>p &lt;+&gt; 1 2 ; \eos</td>
<td>$\text{Exp} \to {+} \text{Exp Exp}$</td>
<td>(2.49)</td>
<td>10-17</td>
</tr>
<tr>
<td>7</td>
<td>\eos StmtList ; Exp Num</td>
<td>p &lt;+&gt; 1 2 ; \eos</td>
<td>$\text{Exp} \to {1} 1$</td>
<td>(2.50)</td>
<td>10-17</td>
</tr>
<tr>
<td>8</td>
<td>\eos StmtList ; Exp 1</td>
<td>p &lt;+&gt; 1 2 ; \eos</td>
<td>$\text{Exp} \to {0} 0$</td>
<td>(2.51)</td>
<td>10-17</td>
</tr>
<tr>
<td>9</td>
<td>\eos StmtList ; Exp 2</td>
<td>p &lt;+&gt; 1 2 ; \eos</td>
<td>$\text{Exp} \to {2} 2$</td>
<td>(2.52)</td>
<td>10-17</td>
</tr>
<tr>
<td>10</td>
<td>\eos StmtList ; Num</td>
<td>p &lt;+&gt; 1 2 ; \eos</td>
<td>$\text{Num} \to {0} 0$</td>
<td>(2.53)</td>
<td>10-17</td>
</tr>
<tr>
<td>11</td>
<td>\eos StmtList ; 2</td>
<td>p &lt;+&gt; 1 2 ; \eos</td>
<td>$\text{Num} \to {1} 1$</td>
<td>(2.54)</td>
<td>10-17</td>
</tr>
<tr>
<td>12</td>
<td>\eos StmtList ;</td>
<td>p &lt;+&gt; 1 2 ; \eos</td>
<td>$\text{Num} \to {2} 2$</td>
<td>(2.55)</td>
<td>10-17</td>
</tr>
<tr>
<td>13</td>
<td>\eos StmtList</td>
<td>p &lt;+&gt; 1 2 ; \eos</td>
<td>$\text{StmtList} \to ({}) \text{a }$</td>
<td>(2.44)</td>
<td>10-17</td>
</tr>
<tr>
<td>14</td>
<td>\eos</td>
<td>p &lt;+&gt; 1 2 ; \eos</td>
<td>(Return true)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.7: A trace of the top-down parsing algorithm given in Figure 1.6 using the extended grammar for Exp0 language (1.42) through (1.64) and the input stream $p + 1 2 ; \eos$. 

Confidential and Copyrighted - © 2015 Franklin, Beedle and Associates - Draft September 14, 2015
start symbol $Prog$ of the grammar is pushed, and the first symbol of the input stream is under the lookahead pointer. At this point we apply rule $Prog \rightarrow \{p, s\} Stmt StmtList$ according to lines 10 through 17. Note, this is the only rule that defines $Prog$ and we have $p \in \{p, s\}$. We pop $Prog$ and we push the symbols on the right side of the rule in reverse order.

**Step 2:** Here we see the effects of the rule application from Step 1 on the stack. The top of the stack is now the non-terminal symbol $Stmt$ and the lookahead pointer points to the symbol $p$. According to lines 10 through 17 we apply the rule $Stmt \rightarrow \{p\} p Exp ;$ which pops the symbol $Stmt$ and pushes the right side of the rule in reverse order. Note that the terminal $;$ is an integral part of this construction.

**Step 3:** The stack reflects the rule application from the previous step. Now the top of the stack has the terminal $p$ and the lookahead pointer points to the symbol $p$. This means, according to lines 18 through 24, that we can pop the stack and move the lookahead pointer to the next symbol.

**Step 4 – Step 12:** These steps perform operations similar to the operations we explored in Step 1 through Step 3: they either expand non-terminals on the stack or pop terminal symbols off the stack according to the appropriate rules.

**Step 13:** This step warrants a closer look. The top of the stack is the non-terminal $StmtList$ and the lookahead pointer points to the $\text{\textbackslash eof}$ symbol. Therefore, the only rule we are able to apply here is $StmtList \rightarrow \{"\"\} "$ according to line 11. The empty string lookahead set will match any lookahead symbol in the input stream including the end of stream symbol. We pop $StmtList$ and push the empty string. However, pushing an empty string onto the stack has no effect on the stack and we are left with just the $\text{\textbackslash eos}$ symbol on the stack.

**Step 14:** According to line 9 the loop of the algorithm now terminates transferring control to line 27. On line 27 we test whether the lookahead pointer points to the end of stream symbol which in our case it does, therefore, we return the value $true$.

A derivation is a sequence of rule applications, if we scan down the column called ‘Rule No.’ of the trace in Figure 1.7 we find that the derivation our algorithm constructed is the same as the derivation constructed in the example on
function BottomUpParse(I, G) returns true or false
// I is an input stream
// G is a context-free grammar with start symbol A
begin
  let S be an empty stack
  push \eos onto S
  let P be the lookahead pointer to the first symbol in I
  sort the rules in G as a list L in decreasing size of right sides
  loop
    search L for a rule Q \rightarrow RuleBody such that TopOfStack(S) = RuleBody
    if rule Q \rightarrow RuleBody exists then
      Reduce :
      pop RuleBody off S
      push Q onto S
    else if P \neq \eos then
      Shift :
      push P and move P to the next symbol
    else if TopOfStack(S) = A and P = \eos then
      Accept :
      return true
    else
      Reject :
      return false
    end if
  end loop
end

Figure 2.8: The bottom-up parsing algorithm given an input stream I and a grammar G.

page 13. Looking down the rule column of the trace it is not difficult to imagine that we can link those rules together in order to construct a parse tree.

Now, watch the movie: http://plipi.info/b/2/q2/figure.mov

This means we have achieved our goal, we now have a mechanical means to parse sentences and reject sentences that are not valid in some grammar.

Bottom-Up Parsing

In the previous sections we looked at top-down parsers which conceptually build parse trees top-down starting at the root. As you might guess, bottom-up parsers do exactly the opposite, rather than building a parse tree from the root a bottom-up parser constructs a parse tree starting with the leaves of the parse tree working its way up to the root node. A bottom-up parser also accepts an input stream
I and a grammar \( G \) as arguments and uses a stack \( S \) internally for processing. However, it is worthwhile pointing out that in this case the grammar does not have to be a LL(1) grammar and we do not have to compute the lookahead sets.

Traditionally bottom-up parsers are understood in terms of four different actions,

**Reduce**: Apply a grammar rule to the stack.

**Shift**: Push a new symbol from the input stream onto the stack

**Accept**: Indicate that the string in the input stream is a valid sentence in language of the grammar.

**Reject**: Indicate that the string in the input stream is not a valid sentence in the language of the grammar.

The algorithm in Figure 1.8 shows these actions as labels for different pieces of code accomplishing the appropriate tasks. If you look at the code carefully then you notice that the algorithm uses the rules of the grammar backwards compared to what we saw in the top-down parser. For example, on line 10 it tries to find a rule whose right side matches the symbols on the top of the stack.\(^3\) If it finds such a rule then it pops the symbols that make up the right side of the rule off the stack and pushes the non-terminal from the left side of the rule onto the stack – applying the rule to the stack backwards!

As in the previous section the best way to get a handle on how this algorithm works is by working through an example. Let us use the grammar (1.95) through (1.97) on page 19 as our input grammar \( G \) with start symbol \( Exp \). What is interesting here is that this grammar is not LL(1) and therefore cannot be used in conjunction with our top-down parsing algorithm. As we will see, the bottom-up parsing algorithm has no problem using this grammar. As our input stream \( I \) we will use the input stream, \( x + y \ \text{\textbackslash{e}o\text{f}} \)

Figure 1.9 is a trace of the bottom-up parsing algorithm in Figure 1.8. Here is the longer explanation of each step in the trace:

**Step 1**: The only thing that is on the stack in Step 1 is the \( \text{\textbackslash{e}o\text{f}} \) symbol, that means, no right side of any rule will be able to match that and therefore the algorithm will shift the symbol \( x \) from the input stream onto the stack.

---

\(^3\)Here we take a slightly broader view of the function \( \text{TopOfStack} \), rather than just returning the top most symbol on the stack the function returns as many symbols there are in the corresponding \( \text{RuleBody} \).
2.2. Parsers

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack ($)</th>
<th>Input Stream ($)</th>
<th>Rule</th>
<th>Rule No.</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\texttt{\textbackslash e o s} &lt;\texttt{e o s} + y \texttt{\textbackslash e o f}</td>
<td>&lt;\texttt{\textbackslash e o s} + y \texttt{\textbackslash e o f}</td>
<td>\texttt{\textbackslash e o s} + y \texttt{\textbackslash e o f}</td>
<td>(2.96)</td>
<td>Shift</td>
</tr>
<tr>
<td>2</td>
<td>\texttt{\textbackslash e o s} x</td>
<td>x + y \texttt{\textbackslash e o f}</td>
<td>\texttt{\textbackslash e o s} x</td>
<td>(2.96)</td>
<td>Reduce</td>
</tr>
<tr>
<td>3</td>
<td>\texttt{\textbackslash e o s} \texttt{\textbackslash e o s}</td>
<td>x + y \texttt{\textbackslash e o f}</td>
<td>\texttt{\textbackslash e o s} x + y \texttt{\textbackslash e o f}</td>
<td>(2.96)</td>
<td>Shift</td>
</tr>
<tr>
<td>4</td>
<td>\texttt{\textbackslash e o s} \texttt{\textbackslash e o s}</td>
<td>x + y \texttt{\textbackslash e o f}</td>
<td>\texttt{\textbackslash e o s} x + y \texttt{\textbackslash e o f}</td>
<td>(2.96)</td>
<td>Shift</td>
</tr>
<tr>
<td>5</td>
<td>\texttt{\textbackslash e o s} \texttt{\textbackslash e o s} + y</td>
<td>x + y \texttt{\textbackslash e o f}</td>
<td>\texttt{\textbackslash e o s} x + y \texttt{\textbackslash e o f}</td>
<td>(2.96)</td>
<td>Reduce</td>
</tr>
<tr>
<td>6</td>
<td>\texttt{\textbackslash e o s} \texttt{\textbackslash e o s} + y</td>
<td>x + y \texttt{\textbackslash e o f}</td>
<td>\texttt{\textbackslash e o s} x + y \texttt{\textbackslash e o f}</td>
<td>(2.96)</td>
<td>Reduce</td>
</tr>
<tr>
<td>7</td>
<td>\texttt{\textbackslash e o s} \texttt{\textbackslash e o s} + y</td>
<td>x + y \texttt{\textbackslash e o f}</td>
<td>\texttt{\textbackslash e o s} x + y \texttt{\textbackslash e o f}</td>
<td>(Return true)</td>
<td>Accept</td>
</tr>
</tbody>
</table>

Figure 2.9: A trace of the bottom parsing algorithm given in Figure 1.8 using the grammar (1.95) through (1.97) and the input stream $x + y \texttt{\textbackslash e o f}$. 
Step 2: The symbol \( x \) on the top of the stack is the right side of rule (1.96) and we reduce the stack by applying this rule in reverse to the top of the stack.

Step 3 - Step 4: There is nothing to match on the stack therefore the algorithm continues to shift new symbols onto the stack.

Step 5: The symbol \( y \) on the top of the stack is the right side of rule (1.97) and we reduce the stack by applying this rule in reverse to the top of the stack.

Step 6: The symbols \( Exp + Exp \) on the top of the stack form the right side of rule (1.95) and we reduce the stack by applying this rule in reverse to the top of the stack.

Step 7: The top of the stack is now the start symbol and the lookahead pointer points to the end of file marker, therefore, we are done parsing and we can return the value \( true \), i.e., accept the sentence in the input stream.

Now, watch the movie: http://plipi.info/b/2/q3/figure.mov

A scan of the Rule column of Figure 1.9 reveals a couple of interesting things:

1. The algorithm seems to be using the rules of the grammar in exactly the opposite order compared to the top-down parsing algorithm.

2. It is not difficult to see that this algorithm constructs the parse tree from the bottom up: it first constructs two subtrees using the rules \( Exp \to x \) and \( Exp \to y \) and then combines these two subtrees into a single tree using the rule \( Exp \to Exp + Exp \).

3. If we scan the Rule column from bottom to top we can see that the algorithm constructs a right-most derivation,

\[
\begin{align*}
Exp & \quad (1.95) \\
Exp + Exp & \quad (1.97) \\
Exp + y & \quad (1.96) \\
x + y & 
\end{align*}
\]

Another interesting observation is that this parsing algorithm did not use our lookahead pointer in order to make decisions as to which rules to apply to the stack. It simply uses the lookahead pointer as a way to remember where it is in the input stream. Because it does not use the lookahead pointer to make parsing
decisions we say that the algorithm uses (0) lookahead symbols. Also note that the algorithm reads the input stream from the left but constructs the right-most derivation. We can now say the following,

> **Bottom-up parsing means building parse trees starting from the leaves and working towards the root node. In our case, another name for this is LR(0) parsing because we are reading from the (L)eft constructing the (R)ight-most derivation using (0) lookahead symbols. Grammars that allow us to do LR(0) parsing are called LR(0) grammars.**

We could easily extend our LR(0) parser to be an LR(1) parser by computing what's called FOLLOW sets. The FOLLOW sets act like the lookahead sets in LL(1) parsing by providing additional information about the context of the rules to be applied by the parser. The main difference being that a FOLLOW set describes the set of terminal symbols that can appear to the right of a non-terminal symbol during a derivation as opposed to the terminals that appear as the first symbols on the right side of a rule. Instead of developing LR(1) parsers here we leave that as an exercise (see Exercise 10).

Our LL(1) and LR(0) parsing algorithms differ in the kind of grammars the algorithms can use for parsing input streams. We already saw that the grammar (1.95) through (1.97) is not LL(1) because we cannot construct the lookahead sets for this grammar. But we just proved that it is LR(0) because we were able to use it in our bottom-up parsing algorithm. You might be asking at this point: are there LL(1) grammars that are not LR(0)? The answer is yes and surprisingly the grammar for our Exp0 language (1.19) through (1.41) on page 7 is LL(1) as we have shown in the previous sections but not LR(0).

Consider the trace given in Figure 1.10 which uses the Exp0 grammar as the input grammar $G$ with the input stream $I$ as,

$$\text{p + 1 2 ; } \text{`eof}$$

The trace proceeds in a normal bottom-up parsing fashion from Step 1 through Step 11. In Step 12 no other rule with a single character on the right side can match the top of the stack except for rule (1.21) with the empty string symbol as its right side. We apply this rule here. The big problem comes in Step 13. Here we have a stack with the following configuration,

$$\text{\`eos Stmt StmtList}$$

and the right sides of both (1.19) and (1.20),

$$\begin{align*}
\text{Prog} & \rightarrow \text{Stmt StmtList} \\
\text{StmtList} & \rightarrow \text{Stmt StmtList}
\end{align*}$$
<table>
<thead>
<tr>
<th>Step</th>
<th>Stack (S)</th>
<th>Input Stream (I)</th>
<th>Rule</th>
<th>Rule No.</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\texttt{\textbackslash{eos}}</td>
<td>&lt;p&gt; + 1 2 ; \texttt{\textbackslash{eof}}</td>
<td></td>
<td></td>
<td>Shift</td>
</tr>
<tr>
<td>2</td>
<td>\texttt{\textbackslash{eos}} p</td>
<td>p \texttt{\leftrightarrow} 1 2 ; \texttt{\textbackslash{eof}}</td>
<td></td>
<td></td>
<td>Shift</td>
</tr>
<tr>
<td>3</td>
<td>\texttt{\textbackslash{eos}} p +</td>
<td>p + &lt;\texttt{i}1&gt; 2 ; \texttt{\textbackslash{eof}}</td>
<td></td>
<td></td>
<td>Shift</td>
</tr>
<tr>
<td>4</td>
<td>\texttt{\textbackslash{eos}} p + 1</td>
<td>p + 1 &lt;\texttt{\textbackslash{i}2}&gt; ; \texttt{\textbackslash{eof}}</td>
<td>Num \rightarrow 1</td>
<td>(2.33)</td>
<td>Reduce</td>
</tr>
<tr>
<td>5</td>
<td>\texttt{\textbackslash{eos}} p + Num</td>
<td>p + 1 &lt;\texttt{i}2&gt; ; \texttt{\textbackslash{eof}}</td>
<td>Exp \rightarrow Num</td>
<td>(2.28)</td>
<td>Reduce</td>
</tr>
<tr>
<td>6</td>
<td>\texttt{\textbackslash{eos}} p + Exp</td>
<td>p + 1 &lt;\texttt{i}2&gt; ; \texttt{\textbackslash{eof}}</td>
<td></td>
<td></td>
<td>Shift</td>
</tr>
<tr>
<td>7</td>
<td>\texttt{\textbackslash{eos}} p + Exp 2</td>
<td>p + 1 2 &lt;\texttt{i}2&gt; ; \texttt{\textbackslash{eof}}</td>
<td>Num \rightarrow 2</td>
<td>(2.34)</td>
<td>Reduce</td>
</tr>
<tr>
<td>8</td>
<td>\texttt{\textbackslash{eos}} p + Exp Num</td>
<td>p + 1 2 &lt;\texttt{i}2&gt; ; \texttt{\textbackslash{eof}}</td>
<td>Exp \rightarrow Num</td>
<td>(2.28)</td>
<td>Reduce</td>
</tr>
<tr>
<td>9</td>
<td>\texttt{\textbackslash{eos}} p + Exp Exp</td>
<td>p + 1 2 &lt;\texttt{i}2&gt; ; \texttt{\textbackslash{eof}}</td>
<td>Exp \rightarrow + Exp Exp</td>
<td>(2.24)</td>
<td>Reduce</td>
</tr>
<tr>
<td>10</td>
<td>\texttt{\textbackslash{eos}} p + Exp</td>
<td>p + 1 2 &lt;\texttt{i}2&gt; ; \texttt{\textbackslash{eof}}</td>
<td></td>
<td></td>
<td>Shift</td>
</tr>
<tr>
<td>11</td>
<td>\texttt{\textbackslash{eos}} p Exp ;</td>
<td>p + 1 2 ; &lt;\texttt{i}eof&gt;</td>
<td>Stmt \rightarrow p Exp ;</td>
<td>(2.22)</td>
<td>Reduce</td>
</tr>
<tr>
<td>12</td>
<td>\texttt{\textbackslash{eos}} Stmt</td>
<td>p + 1 2 ; &lt;\texttt{i}eof&gt;</td>
<td>StmtList \rightarrow (&quot;&quot; ) &quot;&quot;&quot;&quot;</td>
<td>(2.21)</td>
<td>Reduce</td>
</tr>
<tr>
<td>13</td>
<td>\texttt{\textbackslash{eos}} Stmt StmtList</td>
<td>p + 1 2 ; &lt;\texttt{i}eof&gt;</td>
<td>???</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

Figure 2.10: A trace of the bottom parsing algorithm given in Figure 1.8 using the original grammar for our Exp0 language (1.19) through (1.41) and the input stream \texttt{p + 1 2 ; \textbackslash{\text{eof}}}.
respectively, can match the stack. At this point the algorithm is stuck because it
has no way of knowing which rule to pick. We of course would pick rule (1.19)
in order to complete the parse, but the algorithm has no way of determining
that this is the correct choice at this point in time. In the terminology of LR(0)
parsers this is called a reduce-reduce conflict. Reduce-reduce conflicts show up
in grammars with rules that have exactly the same right sides. We say that,

Grammar with reduce-reduce conflicts are not LR(0).

This of course implies that our original Exp0 grammar is LL(1) but not LR(0).

It turns out that there is an easy fix by replacing the first three rules of our
Exp0 grammar with the rules,

\[
\begin{align*}
  Prog & \rightarrow Prog \text{ Stmt} \\
  Prog & \rightarrow \text{Stmt}
\end{align*}
\]

You should convince yourself that the rewritten grammar works with the bottom-
up parsing algorithm (see exercise 5).

Just as in the case of top-down parsing, grammars with rules whose right
sides share common prefixes are not considered LR(0) grammars. Consider this
grammar,

\[
\begin{align*}
  Stmt & \rightarrow \text{if } Exp \text{ then } Stmt \\
  & \quad | \text{if } Exp \text{ then } Stmt \text{ else } Stmt \\
  & \quad | \text{p} \\
  & \quad | \text{q} \\
  Exp & \rightarrow e
\end{align*}
\]

Figure 1.11 is a trace of the bottom-up parsing algorithm using this grammar
together with the input stream,

\[
\text{if e then p else q \text{ \text{\textbackslash eof}}}
\]

In the trace the steps 1 through 6 are straightforward with the exception that
here we grouped symbols together to make up keywords and we move the looka-
head pointer from keyword to keyword. The problem arises in step 7: we can
certainly apply rule (1.128) to reduce the top of the stack but then we are stuck
with a fragment of the if-then-else statement in the input stream for which there
are no rules to parse it any further. The other alternative is to shift the else key-
word onto the stack and continue parsing. This is called a shift-reduce conflict
and we say that,

\footnote{This is common practice called lexical analysis and we will take a look at this in more detail
in Section 1.3.}
<table>
<thead>
<tr>
<th>Step</th>
<th>Stack (S)</th>
<th>Input Stream (I)</th>
<th>Rule</th>
<th>Rule No.</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\eos</td>
<td>&lt;if&gt; e then p else q \eof</td>
<td></td>
<td></td>
<td>Shift</td>
</tr>
<tr>
<td>2</td>
<td>\eos if</td>
<td>if &lt;e&gt; then p else q \eof</td>
<td></td>
<td></td>
<td>Shift</td>
</tr>
<tr>
<td>3</td>
<td>\eos if e</td>
<td>if e &lt;then&gt; p else q \eof</td>
<td>Exp → e</td>
<td>(2.121)</td>
<td>Reduce</td>
</tr>
<tr>
<td>4</td>
<td>\eos if Exp</td>
<td>if e &lt;then&gt; p else q \eof</td>
<td></td>
<td></td>
<td>Shift</td>
</tr>
<tr>
<td>5</td>
<td>\eos if Exp then</td>
<td>if e then &lt;p&gt; else q \eof</td>
<td></td>
<td></td>
<td>Shift</td>
</tr>
<tr>
<td>6</td>
<td>\eos if Exp then p</td>
<td>if e then p &lt;else&gt; q \eof</td>
<td>Stmt → p</td>
<td>(2.119)</td>
<td>Reduce</td>
</tr>
<tr>
<td>7</td>
<td>\eos if Exp then Stmt</td>
<td>if e then p &lt;else&gt; q \eof</td>
<td>???</td>
<td></td>
<td>???</td>
</tr>
</tbody>
</table>

Figure 2.11: A trace of the bottom parsing algorithm given in Figure 1.8 using the grammar (1.128) through (1.132) and the input stream \texttt{if e then p else q \eof}.
A grammar that exhibits shift-reduce conflicts is not LR(0).

We could change the behavior of our algorithm in such a way that if it detects a shift-reduce conflict it always makes the decision to shift thus avoiding leaving snippets of syntax unused and unparsable in the input stream. This is in fact what most industrial strength bottom-up parsers do, they issue a warning about the shift-reduce conflict and resolve the conflict by always shifting. On the other hand, there is an easy way to rewrite this grammar so that it becomes LR(0) by factoring it (see exercise 6).

Similar to LL(1) languages we find that there are multiple grammars that define a language, some of the grammars are LR(0) some are not. We can assert the following,

*We call a language LR(0) if it has at least one LR(0) grammar.*

We should mention that technically LR(0) parsers can run into another kind of reduce-reduce conflict when the right side of a rule is a postfix of the right side of another rule. Consider the grammar,

\[
\begin{align*}
\text{List} & \rightarrow \text{List Item} \\
\text{List} & \rightarrow \text{Item} \\
\text{Item} & \rightarrow e
\end{align*}
\]

Here the right side of the second rule is a postfix to the right side of the first rule. When parsing the input stream e e \text{ \texttt{\textbackslash eof}} with this grammar a pure LR(0) parser has to make a choice when parsing the second e whether to apply the first rule or the second rule in order to reduce the stack to a non-terminal. If the parser were to choose the second rule the parse will terminate in an error. In our implementation of LR(0) parsers we eliminate this choice by forcing the parser to always apply the longest right side of a rule that matches the stack. That is the reason for sorting the rules of the grammar in decreasing size of their right sides. Therefore, in this case our LR(0) parser will always pick the first rule in order to reduce the stack.

### 2.2.2 Building Parsers by Hand

If we wanted to build a parser for a language it is clear that we could simply implement either the LL(1) parser in Figure 1.6 or the LR(0) parser in Figure 1.8 in our favorite programming language. However, both of these algorithms spend a significant amount of time searching the grammars for an appropriate rule to apply. The top-down algorithm searches for a rule to apply on line 11 and the
Let $G'$ be an LL(1) grammar with lookahead sets, then

- for each non-terminal $N$ in $G'$ start a new function say “function $N()$ returns void”.
- The body of each of these functions implements the right sides of the rules for non-terminal $N$ by
  - processing terminals, and
  - calling the functions of other non-terminals as appropriate.

Figure 2.12: Procedure for building a recursive descent parser.

bottom-up parser does so on line 10 in the respective algorithms. This kind of search is a source of inefficiency in the respective parsers.

It turns out that there is a clever way of turning grammars into parsers where no searching is done whatsoever. In this way of constructing a parser we turn non-terminals on the left side of rules into function definitions and non-terminals appearing within the right side of rules into functional calls. The resulting parser then has a function for each non-terminal in the grammar and each function knows how to parse the structures associated with its non-terminal. This kind of parser is called a recursive descent parser because it uses function recursion as a way to make progress during parsing. Figure 1.12 shows the procedure of constructing a recursive descent parser at a very high level. Most notably, we require the grammar to be an LL(1) grammar with lookahead sets.

Constructing a recursive descent parser is best illustrated with an example. Consider the LL(1) grammar with lookahead sets (1.75) through (1.78),

$$
\begin{align*}
Exp &\rightarrow \{+\} + Exp
n \quad Exp &\rightarrow \{x,y\} Var
\quad Var &\rightarrow \{x\} x
\quad Var &\rightarrow \{y\} y
\end{align*}
$$

This grammar defines two non-terminals, $Exp$ and $Var$, and therefore we expect that the recursive descent parser also has two functions. Let us construct the function for non-terminal $Exp$ in C/Java like pseudo code:

```c
function Exp() returns void
begin
    switch (getStreamSymbol())
    case ‘+’:
        nextStreamSymbol(); // match the ‘+’ symbol
```

Confidential and Copyrighted - © 2015 Franklin, Beedle and Associates - Draft September 14, 2015
Looking at this code carefully you will find that the function Exp() parses expressions according to the first two grammar rules above: the grammar rules that define the non-terminal Exp. The function first retrieves the symbol at the lookahead pointer in the input stream. Then, if this symbol matches the lookahead set of the first rule, i.e., if this lookahead symbol matches the + symbol then we execute the first rule. This means first moving the lookahead pointer to the next character skipping the + symbol and then calling Exp() recursively (twice – according to the right side of the first rule). On the other hand, if the symbol from the lookahead pointer matches either x or y then we continue parsing the input with the second rule of the grammar by calling the function Var(). If the symbol from the lookahead pointer matches none of the symbols in the lookahead sets then the input stream does not conform to the structure of an expression and we flag a syntax error. As you might have guessed at this point the function getStreamSymbol() returns the symbol currently pointed to by the lookahead pointer in the input stream. The function nextStreamSymbol() moves the lookahead pointer to the next symbol in the input stream. Finally, the function syntaxError() reports an error to the user and then aborts the parser.

As in the previous function this function uses the lookahead sets to select which rule to execute. In this case the right sides of the rules are very simple, they
simply match the variable names in the input stream. However, if the variable
names in the input stream are not correct then the function flags an error and
aborts.

Consider parsing the input stream,

```c
<+> x y \eof
```

with this recursive descent parser. To start the parser we call the function asso-
ciated with start symbol of our grammar, $Exp$. This function looks at the symbol
under the lookahead pointer, finds a + symbol, and uses the first alternative of
its switch statement. Here we move the lookahead pointer to the next symbol
in the input stream and then call ourselves recursively twice. The first time to
match the $x$ symbol and the second time to match the $y$ symbol. Notice that
there is no searching involved during parsing, everything is completely deter-
mined by the lookahead pointer and the lookahead sets in the case statements.

One way of looking at recursive descent parsers is that the function invocation
for a non-terminal activates the right sides of the associated rules all at the same
time. The precise right side to execute is then determined by comparing the
lookahead pointer and the lookahead set for each right side.

We know that this parser uses an LL(1) grammar but is it itself an LL(1)
parser? From the discussion above we already know that a recursive descent
parser reads the input from left to right; this gives us our first L. We also know
that it uses a single symbol lookahead pointer; this gives us our (1). Finally, the
right sides of the grammar rules are encoded in the recursive descent parsers
just the way they are written in the grammar, *i.e.* in the implementation of the
rule,

$$Exp \rightarrow \{+\} + Exp Exp$$

we first match the + symbol and then call the two functions for $Exp$,

```c
\begin{verbatim}
: ;
    case '+':
      nextStreamSymbol() // match the '+' symbol
      Exp() Exp()
      return
    :
\end{verbatim}
```

Since code in C/Java like languages (or any imperative language for that matter)
is executed top to bottom and left to right, we know that the function for the
first $Exp$ in the grammar rule is always executed before the second one. That
means the recursive descent parser constructs a left-most derivation. This gives us our second L. Therefore, a recursive descent parser is indeed a LL(1) parser.

Now, watch the movie: http://plipi.info/b/2/q4/figure.mov

It is straightforward to extend our recursive descent parsing functions in order to construct parse trees. Assume we have two kinds of tree generating functions, leafNode() and opNode(), both of which return the type Tree. With this we can rewrite our Exp() function as,

```plaintext
function Exp() returns Tree
begin
    switch (getStreamSymbol())
    case '+':
        nextStreamSymbol() // match the '+' symbol
        return new opNode('+', Exp(), Exp());
    case 'x':
    case 'y':
        return Var();
    default:
        syntaxError();
    end switch
end
```

and we can write our Var() function as,

```plaintext
function Var() returns Tree
begin
    switch (getStreamSymbol())
    case 'x':
        nextStreamSymbol() // match the 'x' symbol
        return new leafNode('x');
    case 'y':
        nextStreamSymbol() // match the 'y' symbol
        return new leafNode('y');
    default:
        syntaxError();
    end switch
end
```

Recursive descent parsers are top-down parsers, that is, they begin constructing a derivation at the start symbol. However, the parse tree itself is constructed bottom up. This is due to the fact that parse trees from the point of view of programming languages are inductive data structures that have to be constructed starting from the leaves to the root. Perhaps a quick look at the implementation of the first rule in our grammar will highlight this,
Figure 2.13: A parser generator viewed as a translator.

```java
  case '+':
    nextStreamSymbol() // match the '+' symbol
    return new opNode('+', Exp(), Exp());
```

Here we know that the parse tree node has to be a node describing the + symbol with two Exp's as subtrees. However, in C or Java we cannot construct this node until we have the actual structure for the subtrees. Thus, we parse top-down build the tree bottom-up.

### 2.2.3 Parser Generators

Very few language implementers would choose to implement a parser by hand. One reason being that common languages like Java or C typically involve hundreds of grammar rules and implementing those in a recursive descent parser by hand is a nontrivial undertaking. Another reason and perhaps the more important reason is that languages evolve during their lifecycle. As a language matures new features are added and old features are modified or depreciated. Changing a hand-coded parser to keep up with an evolving language might require major, time consuming reengineering of the parser every time the specification of the language changes.

Most language implementers would choose to use a parser generator in order to construct parsers. A parser generator reads a grammar specification and generates source code for a parser that can parse the valid sentences in the language of the specified grammar. In this way we can view a parser generator as a translator for a domain specific language (see Figure 1.13). Here the domain
specific language is the grammar specification and the target language is whatever language the parser code is generated in. In most cases this will be either Java or C/C++. The parser generator will go through all the typical stages of a translator. It first performs a syntax analysis (parsing!) of the grammar specification file and constructs an IR. Then it performs the semantic analysis of the grammar file. Here it checks whether all non-terminals have been defined by at least one rule, for instance. Finally, it uses the IR to generate the parser code. The advantage of using a parser generator is that as a language evolves all we need to do is to modify the grammar specification, the parser code is then generated from the modified grammar specification and does not need any additional engineering.

The particular parser generator we discuss here and will use throughout the rest of the book is called ANTLR (ANother Tool for Language Recognition).\footnote{Specifically we are using version 3 of ANTLR: www.antlr3.org}

By default ANTLR will generate recursive descent parser code in Java. More precisely, unless instructed otherwise ANTLR will generate an LL(*) parser in Java from a grammar specification. The (*) here stands for an arbitrarily long lookahead, that is, the generated parser uses as many lookahead symbols as it needs in order to select an appropriate rule for a particular derivation. On the other hand we can tell ANTLR to generate LL(k) parsers, that is, we can fix k to some value and tell ANTLR to generate a recursive descent parser that will use exactly k lookahead symbols during parsing. Setting k=1 will allow us to generate a recursive descent parser with a single lookahead symbol – an LL(1) parser, the kind of top-down parser we discussed above.

Figure 1.14 shows an ANTLR grammar specification for our Exp0 language. If we ignore the first couple of lines then this grammar specification looks very much like the grammar (1.19) through (1.41) on page 7 with some minor syntactic differences. Here the $\rightarrow$ from our original grammar is replaced by a colon and ANTLR insists that each rule group is terminated with a semicolon. Furthermore, individual terminal symbols of our language such as $p$, $s$, and the digits need to be enclosed in quotes. However, the big differences between our original grammar and the ANTLR grammar specification appear at the beginning of the file. The first line of the specification tells ANTLR the name of the language the grammar is to specify. Following that we have the options line which in this case tells ANTLR to produce an LL(1) parser. Perhaps the most surprising piece
Figure 2.14: ANTLR grammar specification for the Exp0 language.
of ANTLR grammar syntax is the rule on line 5. In our original grammar we used the following rules,

\[
\begin{align*}
\text{Prog} & \rightarrow \text{Stmt StmtList} \\
\text{StmtList} & \rightarrow \text{Stmt StmtList} \\
& \quad | \quad ""
\end{align*}
\]

in order to specify that a program in Exp0 consists of one or more statements. Now, lists of syntactic units appear quite often in programming languages and therefore ANTLR provides a shorthand notation for lists. For instance, the ANTLR rule,

```
prog : stmt+
```

states exactly the same thing as the three rules in our original grammar: that programs consist of one or more statements. The plus sign after the non-terminal stmt indicates that statements can appear more than once in a program, but a statement must appear at least once in a program. If we wanted to express a possibly empty lists of statements we could use ANTLR’s star notation for lists,

```
prog : stmt*
```

This new rule states that programs are comprised of zero or more statements. This kind of list notation makes grammars much more concise and therefore easier to read.

In order to generate a parser from the grammar in Figure 1.14 let us assume that you saved the grammar specification as file `exp0.g` in your current directory. Furthermore, let us assume that you have installed Java and ANTLR on your system\(^6\). In this case you can generate a parser with the following command,

```
$ java org.antlr.Tool exp0.g
```

Another parser generator that is widely used is YACC (Yet Another Compiler Compiler).\(^7\) YACC generates LR(1) parser code from a grammar specification file. More details on YACC can be found in Appendix B.

There is one more important concept we need to cover in terms of parser generators and that is the idea of rule actions. Actions are a way to embed code into a grammar specification that allows the generated parser to accomplish

\(^{6}\)For more details on ANTLR and its installation see Appendix A.

\(^{7}\)An open source version of YACC is available as Bison: www.gnu.org/software/bison.
tasks while parsing a program. In our case where we generate parsers in Java, actions are simply Java code attached to rules. Syntactically actions look like this,

```java
stmt: 'p' exp ';'
    { System.out.println("print statement"); }
| 's' var exp ';'
    { System.out.println("store statement"); }
```

Here, we use actions to print an appropriate message to the terminal depending on which rule was used in the derivation of the statement. Action code is inserted into the generated parser based on its position in the rule. In the case of ANTLR we are generating a top down parser and placing the action at the end of a rule means that we complete the full derivation for the non-terminal before we print out the message. If we had placed the action at the beginning of the rule,

```java
stmt: { System.out.println("print statement"); } 'p' exp ';'
| { System.out.println("store statement"); } 's' var exp ';'
```

then we would print out the message before the derivation of the non-terminal begins.

### 2.2.4 An Example: Our First Language Processor

Let us put this all together and build our first language processor. Our processor will read an Exp0 program, count the number of times the program references the value of a variable, and then prints the number of value references to the terminal. Looking at the grammar in Figure 1.14 we see that the only place where we can access the value of a variable in Exp0 is within an expression. Notice that the occurrence of a variable within an expression is very different from a variable occurrence within a store statement (at least when in occurs as the first argument to the store statement). When we have a variable reference within an expression then we are interested in retrieving the value that is stored in the variable. On the other hand, if we have a variable reference as the first argument to the store statement then we are interested in changing the value that is stored in the variable. In the traditional terminology of language processors we often refer to variable references within expressions as rvalues and variable references that allow us to change the value of the variable as lvalues. This terminology is based on the fact that variable references to the right of an assignment operator access the value of a variable, the rvalue, and variable reference to the left of an assignment operator modify the storage location of the variable, the lvalue.
Given these two distinct ways of referencing variables and given that in this case we are only interested in rvalues we have to build a language processor that understands the difference between lvalues and rvalues. That means our language processor has to parse the input and only count the variable occurrences within expressions and ignore the lvalues. A program that does simple pattern matching on the variable names will not work here. A program that parses its input and then produces some sort of summary output is called a reader according to our classification of language processors. That means our first language processor will be a reader for the Exp0 programming language.

A good place to start the discussion of our reader is the ANTLR grammar file extended with the appropriate actions shown in Figure 1.15. When comparing this grammar specification to the original ANTLR grammar specification in Figure 1.14 the most obvious difference besides the actions is the @members declaration. As part of generating a parser from a grammar specification file ANTLR generates a parser object and the @members sections allows us to add customized members to that object. Members can include data members as well as method members. Here the first member we are adding to the parser object is the refcount variable which will help us keep track of the rvalue variable references. We also add the member function summary which we will call when the parser is done in order to print out the total number of references we found. We gave a new name to the grammar specification of our reader: exp0count. This new name reflects the fact that this grammar not only specifies the syntax of Exp0 but also encodes behavior specific to our reader.

Scanning down the grammar file you will find two actions. One at the end of the rule for the non-terminal prog indicating that once we have parsed all the statements in a Exp0 program we will print out the summary. The other action appears in the rule,

```
exp : var { refcount++; }
```

Here we bump up the reference counter every time we derive the non-terminal var from the non-terminal exp. That is, we bump up the reference counter every time we discover a variable within an expression.

The last line of the grammar specification is a rule that allows the parser to derive white space characters such as the space, tab, and new line characters and the associated action instructs the parser to ignore these characters. That is, these characters will not appear in the derivation.

Assuming that you stored your grammar specification in the file exp0count.g then we can generate the parser with the command,

```java
java org.antlr.Tool exp0count.g
```
Figure 2.15: ANTLR grammar specification for the Exp0 variable rvalue counter.
This will create three new files in your directory,

```console
exp0count.tokens
exp0countLexer.java
exp0countParser.java
```

The first file contains information that tells ANTLR what kind of symbols may appear in Exp0 programs. The second file is a Java file that contains the class specification of an object that knows how to extract symbols from the input stream/input file. We will have more to say about this in the next section. Finally, the last file is the generated parser. If you look at the generated Java code for this parser you might be surprised that it doesn't quite look like the code for the recursive descent parser we discussed in Section 1.2.2. There are a couple of reasons for this: First, the parser needs to be able to handle the extended grammar constructs such as repeated syntactic units as in `stmt+`. Second, the improved ability to detect parsing errors with what is called a FOLLOW SET. Regardless of the internal structure of the generated parser from a user perspective it behaves just as the recursive descent parser we discussed earlier.

Now, in order to turn our reader into a valid Java program we need a top-level driver. The code for this top-level class is given in Figure 1.16. The first thing the code in this file does is to import the ANTLR runtime library which contains all the functionality that supports the generated parser. The file also contains the top-level class `exp0count`. The job of the top-level class `exp0count` is to act as a container for the static `main` function required by Java. This `main` function first checks whether a Exp0 input file name was actually specified at the command line. If not it will print out a usage message and quit. If the a file was specified then it will create an input stream which passed to the lexer (symbol extractor) and the lexer object is then passed to the parser. In order to start parsing we call the function generated for our start symbol, in this case we call `prog()`. Now, if you compile this file together with the generated parser we have a complete language processor for Exp0.

Assume that we have the following Exp0 program stored in `p1.exp0`,

```exp0
s x 1;
p x;
s y 2;
p y;
p (+ x y);
```

then we can run our language processor,

```console
$ java exp0count p1.exp0
Processing: p1.exp0
```

Confidential and Copyrighted - © 2015 Franklin, Beedle and Associates - Draft September 14, 2015
Figure 2.16: Driver program for the Exp0 reader.
2.3. Lexical Analysis

The language generated by the Exp0 ANTLR grammar in Figure 1.14 only allows single character words, that is, keywords, variable names, and numbers are restricted to a single character. However, real programming languages allow longer words as part of their syntax. This introduces the notion of lexical structure of a programming language and the lexical structure of programming languages is specified with rules similar to grammar rules. The lexical specification of a programming language defines how individual symbols are combined to form words. Furthermore, the lexical specification also defines how words are grouped into tokens which makes it easier to write grammars. The tokens defined in lexical rules act similarly to non-terminals in grammar rules in that they summarize the lexical structure of a group of related words.

The lexical structure of a programming language is specified with rules using regular expressions. Consider the rule that defines the lexical structure of an integer value,

\[
\text{INTVAL : '0' | '-'?('1'..'9')('0'..'9')*}
\]

The regular expression of this rule states that words of the token INTVAL have the following lexical structure,

1. it is either the character '0', or
2. it is an optional minus sign followed by a single character from the character class '1'..'9' followed by zero or more characters from the character class '0'..'9'.

The following are all valid words of this token: 0, -201, 3507. On the other hand, here are some words that do not belong to this token: -0, 010, +225.

Let us create another lexical rule that defines a token for the alpha-numeric structure of variables names typical for programming languages,

\[
\text{NAME : ('A'..'Z' | 'a'..'z')(A'..'Z' | a'..'z)(0'..'9')*}
\]

The source code for this language processor and all other language processor we will be talking about in this book is available at \text{http://plipi.info/source}.
This rule states that the lexical structure for words of the token NAME consists of a single character drawn from the character classes 'A'..'Z' or 'a'..'z' followed by zero or more characters drawn from the character classes 'A'..'Z', 'a'..'z', or '0'..'9'. In other words, variables names start with either an upper or lower case letter followed by zero or more upper or lower case letters and numbers.

From these lexical rules we can generate “parsers” for tokens called lexical analyzers or lexers for short. This means that syntax specifications for programming languages consist of two parts: A phrase structure specification using grammar rules and a lexical structure specification using rules with regular expressions. This is summarized in Figure 1.17. Parsing now proceeds as a two stage process. First the input stream is tokenized by the lexer and the tokens are sent to the parser as a token stream in order to recognize the phrase structure of the program. This also explains the structure of the driver program we encountered in Figure 1.16 where on line 22 we set up the lexer to recognize the lexical structure of the input program and then on line 26 we set up the parser to recognize the phrase structure of the input program.
2.3. Lexical Analysis

```plaintext
grammar exp1;
options{k=1;}

// grammar rules
prog : stmt+ ;
stmt : PRINT exp SEMI
     | STORE var exp SEMI
     ;
exp : PLUS exp exp
    | MINUS exp exp
    | POPEN exp PCLOSE
    | var
    | INTVAL
    ;
var : NAME ;

// lexical rules
PRINT : 'print'
STORE : 'store'
SEMI : ';
PLUS : '+
MINUS : '-
POPEN : '('
PCLOSE : ')
NAME : ('A'..'Z'|'a'..'z')('A'..'Z'|'a'..'z'|'0'..'9')* ;
INTVAL : '0' | '-'?( '1'..'9')('0'..'9')* ;
```

Figure 2.18: ANTLR grammar specification for the Exp1 language. This now includes the lexical specification.
2.3.1 An Example: The Exp1 Language

Of course in the Exp0 language all words consist of a single character so the lexer does not have much to do, so let’s extend Exp0 to include multi-character keywords and variable names as well as numbers that contain more than a single digit. The ANTLR specification of the resulting Exp1 programming language is given in Figure 1.18. Notice that this specification now neatly separates the phrase and lexical structure specifications. The grammar rules are now all expressed in terms of tokens and the tokens are defined as part of the the lexical specification. As is usual for ANTLR specifications, token names are given in capital letters and grammar non-terminals are written in lower case letters.

A closer look at the lexical specification reveals that we have two kinds of tokens: One kind of token consists only of a single word (e.g., consider the token PRINT, the only word belonging to this token is the word print). We call tokens that only consist of a single word structural tokens and they typically represent the keywords of the programming language in question. The other kind of token has a possibly infinite number of words (e.g., consider the token INTVAL). These tokens usually represent values.

2.3.2 Regular Expressions and Their Processors

In the previous section we informally introduced regular expressions as a way to specify the structure of words belonging to a token. It is time to formalize this. Here then is the full definition of regular expressions,

- Each letter 'A' through 'Z' and 'a' through 'z' is a regular expression (the quotes are necessary).
- Each number ‘0’ through ‘9’ is a regular expression (the quotes are necessary).
- Each printable character ‘(’, ‘)’, ‘-’, ‘+’, etc. is a regular expression (the quotes are necessary).
- If A and B are regular expressions then AB is also a regular expression and represents the concatenation of the two regular expressions.
- If A is a regular expression then (A) is also a regular expression. Parentheses allow us to group regular expressions. The use of quotes in regular expressions is very important because the regular expression (A) is different from the regular expression ‘( A )’. The former is the grouping of regular expression A and the latter is the concatenation of the three regular expressions ‘(’, A, and ‘)’.
- If A and B are regular expressions then A | B is also a regular expression and represents the choice between regular expression A and regular expression B.
2.3. Lexical Analysis

- If \( A \) is a regular expression then \( A? \) is also a regular expression and specifies the regular expression \( A \) as optional.

- If \( A \) is a regular expression then \( A^* \) is also a regular expression and specifies that the regular expression \( A \) can appear zero or more times.

- If \( A \) is a regular expression then \( A+ \) is also a regular expression and specifies that the regular expression \( A \) can appear one or more times.

In addition to this standard definition of regular expressions syntax specification tools provide extensions that make the definition of tokens easier. ANTLR, for example, provides the following extensions to regular expressions,

- The regular expression 'A..'Z' represents a single character between 'A' and 'Z'. Similarly for 'a..'z' and '0..'9'.

- The unquoted dot is a regular expression that represents any single printable character.

- The special characters 'n', 't', and 'r' are also regular expressions representing the newline character, the tab character, and the carriage return character, respectively.

- If \( A \) is a regular expression then \( \sim A \) is a regular expression representing not \( A \). This is useful in conjunction with character classes. For example the regular expression '(A..Z)\sim(A..Z)' specifies a word structure that starts with a capital letter followed by a single character that is not a capital letter.

- A regular expression which consists of the concatenation of individual characters can be represented as a string. For instance, the regular expression 'p' 'r' 'i' 'n' 't' can be compactly written as the regular expression 'print'.

Now that we have the full definition of regular expressions it is perhaps worthwhile to take a closer look at the regular expression for the token INTVAL in the specification of the language Exp1,

\[
'0' | '0'-'9'('1'..'9')('0'..'9')*
\]

First notice that this expression consists of two regular expressions, namely '0' and '0'-'9'('1'..'9')('0'..'9')*, joined together by the |-bar denoting a choice. The first regular expression is just the character '0' denoting the fact that one choice is just the digit 0. The second regular expression is the concatenation of three different regular expressions: '0'-'9', ('1'..'9'), and ('0'..'9')*. The first one denotes an optional minus sign, the second a single digit between 1 and 9, and the last one denotes zero or more digits between 0 and 9. The concatenation of the three regular expressions represents possibly negative numbers that start with a digit between 1 and 9 followed by zero or more digits between 0 and 9.
At least conceptually the processors for regular expressions are non-deterministic finite state machines (NFSM). NFSMs differ from their deterministic counterpart (DFSM) by the fact that they embody non-deterministic choices on particular input symbols and always “guess” the correct choice. It is a classic result from formal language theory that any NFSM can be converted to a DFSM and therefore our idea of a correct guess in the NFSM is justified.

Figure 2.19 shows the NFSM associated with the token `INTVAL` which is defined by the regular expression,

'0' | '-'?('1'..'9')('0'..'9')*

We have drawn the NFSM in such a way that the $\epsilon$-transitions, transitions that do not process an input symbol, denote the non-deterministic choices. You can see that we have an immediate choice on the start state (the left-most state with the dangling arrow pointing to it). This first choice is between the selecting the part of the NFSM to recognize the '0' character and the part that recognizes non-zero integers. Then we have another choice in the part of the NFSM that recognizes non-zero integers on whether to include the minus sign or not. Note that there is almost a one-to-one correspondence between the shape of the regular expression and the architecture of the NFSM. NFSMs process symbols from left to right in the input stream and when they reach an accepting state (states with the double border) without any further input to process they return the appropriate token name together with the recognized word.

Now, watch the movie: http://plipi.info/b/2/q6/figure.mov
Figure 2.20: Finite state machine for the token NAME.

Figure 2.20 shows the NFSM for the token NAME which we defined with the regular expression,

\[ ('A'..'Z' | 'a'..'z')('A'..'Z' | 'a'..'z' | '0'..'9')* \]

Here we used an \( \epsilon \)-transition to “paste” together two simpler NFSMs that each represent one of the two subexpressions,

\[ ('A'..'Z' | 'a'..'z') \]

and

\[ ('A'..'Z' | 'a'..'z' | '0'..'9')*, \]

respectively.

Typically, syntax specifications for programming languages contain many token definitions as we saw in our Exp1 grammar in Figure 1.18. In order to construct a sensical lexer, the parser generator will sort the token definitions from the most specific to the most general (at least conceptually). This is necessary because in our case of Exp1, for example, we want the word print to belong to the token PRINT and not to the token NAME. Also, at the conceptual level we can then envision that the parser generator generates lexer code that executes the NFSM for each token definition one after the other and the first one that matches the current input symbols returns the appropriate token name and the matched word to the parser. Of course, that is not how it is done in real parser generators because it would be too inefficient. Many efficient table driven algorithms have been designed to turn the regular expression based token definitions in to effective, deterministic lexers.

As an interesting aside on implementing lexers, any regular expression can be converted into a context-free grammar (another result from formal language theory) and if we can rewrite this grammar as an LL(1) grammar then we can use a recursive descent parser as the implementation strategy for the lexer (see exercise 26).
2.4 Syntax Directed Language Processing

For a certain class of languages we can do our processing as soon as we recognize a syntactic structure. This is called syntax directed language processing and is best illustrated with an example.

2.4.1 An Example: An Interpreter for Exp1

According to our classification of language processors an interpreter reads a program and executes the program directly (see Figure 1.6). In turns out that Exp1 is so simple that we do not need an IR to interpret Exp1 programs but we use a technique called syntax-directed interpretation where we execute the semantic rules of the language as soon as we recognize the corresponding syntactic structures.

In order to get a better idea of what interpretation is we turn to a language that you are very familiar with: algebra. Algebraic terms are groups of symbols that acquire mathematical meaning and interpretations using specific rules to manipulate these symbols. Consider the algebraic expression,

\[ x = 3 \]

We interpret this expression by first interpreting the symbol 3 as the mathematical value three, we then interpret the symbol \( x \) as a variable, and because the variable appears to the left of the symbol ‘=’ we assign the value three to the variable \( x \). Now consider the term,

\[ y = 2 + x \]

In order to interpret this term we first figure out what value the variable \( x \) has, we then interpret the symbol 2 as the mathematical value two, and finally we compute the value of the right term by interpreting the ‘+’ symbol as the addition operator computing the value five (if we assume that \( x \) has the value three). In order to complete the interpretation of this algebraic term we interpret the ‘=’ as an assignment of the value five to the variable \( y \).

One thing you probably noticed at this point is that the interpretation of algebraic terms is bottom-up, that is, it starts at the operands that are immediately computable, such as constant symbols or variables, and works its way up to the top-level operator which in this case is the assignment operator. This approach to interpretation is called syntax-directed interpretation because the interpretation process is guided by the syntactic structure of the terms.
Figure 2.21: Interpreting the parse tree for the program `store y + 2 x ;`.
In order to see what syntax-directed interpretation looks like for our Exp1 language let us start with the parse tree for the program,

store y + 2 x ;

Figure 1.21 shows the parse tree. It is clear from the structure of the tree that in order to compute a value to store into variable \( y \) we would have to interpret the tree starting at the right side leaves and then keep interpreting the operators and recomputing the values on the tree branches along the direction of the red arrows. One way to visualize syntax directed interpretation is that values percolate from the tree leaves up to the root. In our case, once interpretation reaches the root of the parse tree the value computed thus far is stored in the variable \( y \).

Just as in our exp0count program above where we tried to find all the variable references but not definitions in an Exp0 program we have to be careful with the interpretation of Exp1 programs and distinguish lvalues and rvalues. If a variable appears as an lvalue (that is it appears as the first argument to the STORE statement) then we assign a value to it and if a variable appear as an rvalue (that is it appears in the expression of the STORE statement) then we just look up the corresponding value for the variable. Value updates and lookups are usually accomplished with the help of a symbol table. Exp1 is simple enough that a simple hash table as a way to associate variable names with values suffices.

Now, watch the movie: http://plipi.info/b/2/q7/figure.mov

Now it turns out that we can achieve the same interpretation behavior in a parser without having to construct an explicit parse tree. Consider the non-terminal \( \text{exp} \) defined in the Exp1 grammar (Figure 1.18) as,

\[
\begin{align*}
\text{exp} & : \quad \text{PLUS exp exp} \\
& \quad | \quad \text{MINUS exp exp} \\
& \quad | \quad \text{POOPEN exp PCLOSE} \\
& \quad | \quad \text{var} \\
& \quad | \quad \text{INTVAL} \\
& ;
\end{align*}
\]

In a hand-built recursive descent parser we can parse this non-terminal with a corresponding function but in order to be able to pass around the appropriate values we declare an integer return value for the function. Our parsing function then looks something like this,

\[
\text{function exp() returns Integer}
\]
begin
switch inputToken()
case PLUS:
   return exp() + exp()
case MINUS:
   return exp() - exp()
case POPEN:
   Integer value = exp()
   matchToken(PCLOSE)
   return value
case VAR:
   Token var = inputToken() // match the rvalue–variable
   return lookupValue(var.getString())
case INTVAL:
   Token value = inputToken() // match the constant token
   return convertToInteger(value.getString())
default:
   syntaxError()
end switch
end

Having declared a return value for the parsing function exp implies that parsing an expression will actually compute an integer value. If we look back at the interpretation of the parse tree in Figure 1.21 then we see that this is exactly what is happening: any time we see the non-terminal exp in the tree we can observe that either an integer value is being computed or propagated.

Now, if we take a closer look at the parsing function itself we see that the exact same behavior that we observed on the parse tree is encoded here. For the tokens PLUS and MINUS we see that the function exp() calls itself recursively and then given the returned values performs the appropriate arithmetic operation in order to compute its own return value, that is, at this point we take the two values of propagated up from the subexpressions, add or subtract them as appropriate, and return the newly computed value. Something very similar happens with the token POPEN; here we simply return the value of the parenthesized expression. When we encounter rvalue variables we use the name of the variable in order to look up its associated value and then return that value. With constants we convert the string associated with the token into an integer value and return that. value It is easy to see that the function exp() represents a recursive function that will recurse to the recursion termination cases (VAR and INTVAL) and then backs out of the recursion while returning integer values. In this way we see computed values percolating from the bottom up to the top where they can then be used.

Let us take a look at the parsing function for statements. The grammar rule for statements in Exp1 is,

stmt : PRINT exp SEMI
and the corresponding parser function with the extensions to deal with expression values is,

```java
function stmt () returns void
begin
  switch inputToken ()
  case PRINT:
    Integer value = exp ()
    matchToken (SEMI)
    writeToOutput (value)
    return
  case STORE:
    Token var = inputToken () // match the lvalue-variable
    Integer value = exp ()
    matchToken (SEMI)
    updateVariable (var . getString (), value)
    return
  default:
    syntaxError ()
  end switch
end
```

The first thing to notice is that statements themselves do not return any values and therefore the return type of the corresponding parsing function is `void`. Looking at the function itself we see that in the case of a PRINT statement we compute the value of the expression while parsing it and then write that value to the output. In terms of the STORE statement we parse the lvalue-variable and then we parse the expression. The parsing function for the expression will return an integer value for the expression and it is this value that we use in order to update the value associated with the variable name.

Of course we don't want to build parsers by hand but we want to use a tool like ANTLR to generate our parsers. ANTLR provides the idea of an attribute that we can associate with a non-terminal and this allows us to encode the same parser behavior as in the hand-built parser. Figure 1.22 shows the ANTLR specification for our Exp1 interpreter. For space reasons we did not include the supporting Java code. If you do look at the Java code (see the QR code below) you will notice that the supporting Java code is split into two different parts: one part is the @header part for the declarations of libraries to include in your parser and the other part is the @members parts that allows you to add data and function members to the parser class as we have seen before.

Now, watch the movie: http://plipi.info/b/2/q8/exp1Interp.g
Figure 2.22: ANTLR specification for the Exp1 interpreter.
If we take a closer look at the specification file and ignore the actions for a minute we find a number of major differences between this specification and our original Exp1 specification in Figure 1.18. The first one is that structural tokens such as ‘print’ have been put directly into the grammar specification itself. Due to the tight integration of the parser generator and the lexer in ANTLR, ANTLR can generate lexical rules automatically for any structural tokens appearing in the grammar specification making ANTLR specifications nice and compact and easily readable. The second difference is that some non-terminals now have a return value. These are precisely the values that we are interested in during our interpretation of Exp1 programs and these are the same values that we saw as return values in our hand-built parser. And finally we have two additional lexical rules; one for comments and one for white space. A closer look at the actions associated with these lexical rules shows that there is a special directive for the parser to ignore both comments and white space: $channel=HIDDEN.

We begin by taking a look at the rules for the non-terminal exp,

```plaintext
exp returns [Integer value]
  : '⁺' e₁=exp e₂=exp { $value = $e₁.value + $e₂.value; }
  | '⁻' e₁=exp e₂=exp { $value = $e₁.value - $e₂.value; }
  | '(' e=exp ')' { $value = $e.value; }
  | var { $value = lookup($var.name); }
  | INTVAL { $value = new Integer($INTVAL.text); }

```

The first rule specifies the addition operation. Notice that the original rule has two exp non-terminals on the right. This introduces an ambiguity if we are trying to access the return values of each of these non-terminals. In order to get rid of this ambiguity we give names to each of the non-terminals. In our case we call the left exp non-terminal e₁ and the right exp non-terminal e₂. Recall that we declared return a value with name value for every non-terminal exp. In the rule actions we can access these return values with a special notation. We can access the return value of our first expression e₁ with the notation,

```
$e₁.value
```

Similarly for expression e₂. We also have a special notation to set the return value of the current exp non-terminal,

```
$value
```

With this the action for the first rule simply adds the two return values for expressions e₁ and e₂ and makes the resulting value the return value of the current expression non-terminal,

```
{ $value = $e₁.value + $e₂.value; }
```
You should convince yourself that the remaining rules encode exactly the same behavior as in the corresponding hand-build parsing function above.

The next important group of rules in our grammar are the rules for the non-terminal stmt,

\[
\text{stmt} : \text{print } \exp; \quad \{ \text{print($\exp$ \text{.value});} \} \\
| \text{store } \var \exp; \quad \{ \text{update($\var$ \text{.name}, $\exp$ \text{.value});} \}
\]

Since statements do not return any values there is no need to declare a return value for this non-terminal. The first rule specifies the PRINT statement and as expected the action associated with this rule takes the value computed by the non-terminal exp and prints it. The second rule specifies the STORE statement and again as expected the action updates the symbol table with the name-value pair which consists of the variable name and the value computed by the expression. You should compare this rule set with hand-built parsing function for statements above.

You should take notice that tokens have a built-in return value, namely the text of the string that was returned from the lexer as part of the token. For example, in the case of the token INTVAL we can access that text with the notation,

\[
\text{\$INTVAL\text{.text}}
\]

All we need to do in order to complete our interpreter is to write a driver program similar to the program appearing in Figure 1.16 but with the names adjusted for our new program accordingly. At this point you should download the code from the book website and experiment: http://plipi.info/source

### 2.4.2 An Example: A Pretty Printer for Exp1

Syntax directed language processing does not only apply to interpretation. We can also use syntax directed techniques to build simple translators. A pretty printer for our Exp1 language is a good example to study. As you might know, pretty printers are programs that read the source of a program written in some programming language and then generate code in the same language but formatted nicely so that the program is easy to read for humans. This is a great example of a simple translator shown in Figure 1.8 except in our case it is not necessary to construct an IR because we will use syntax directed translation. Our pretty printer accomplishes two things: One, it will put each statement on its own line. Two, the expressions will be rewritten into Lisp like syntax. In Lisp, each operation is embedded in a pair of parentheses. For example, to add two numbers in Lisp we write the following expression,

\[
(+ \ 2 \ 3)
\]
grammar exp1pp;
	options{k=1;}

@members {
  void emit(String s) {
    System.out.print(s);
  }
}

// grammar rules
prog : ( stmt ';' { emit(";\n"); } )+

stmt : 'print' { emit("print "); } exp
  | 'store' VAR { emit("store + $VAR.text + "); } exp
  ;
exp : '+' { emit("(+ "); } exp { emit(" "); } exp { emit("); }
  | '-' { emit("(- "); } exp { emit(" "); } exp { emit("); }
  | '(' exp ')' 
  | VAR { emit($VAR.text); }
  | INTVAL { emit($INTVAL.text); }
  ;

// lexical rules
VAR : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9')* ;
INTVAL : ('0'..'9')+ ;
COMMENT : '/\*'|'\* '/(\n|\r)* \r? \n? \n' { $channel=HIDDEN; }
WS : ( ' ' | '\t' | '\r' | '\n' ) { $channel=HIDDEN; }

Figure 2.23: ANTLR specification for the Exp1 pretty printer.
This means we also get rid of unnecessary parentheses. For example, the expression 
\((+ (2) (3))\) will be rewritten as above.

Figure 1.23 shows the ANTLR specification of our pretty printer. You will notice the usual prologue in the specification. Here we declare a parser member function emit that allows us to write strings to the terminal output. Skipping down to the lexical rules we see that nothing has changed from the specification of the syntax directed interpreter with the exception that now we have a token VAR instead of NAME.

Now, if you look at the grammar rule section of the specification and ignore the actions for a minute then you will notice that we have rewritten the grammar slightly. It still generates the same language and in this form makes it easier to generate code. Also notice that none of the non-terminals have return values. This is because we are dealing with a simple translator, a translator that does not perform any semantic analysis but simple does a mapping of the syntax.

The first rule of the grammar section is,

\[
\text{prog} : \ ( \ \text{stmt} \ ; \ )\ {\text{emit}} ( ; \n ) \{ \} \ + \;
\]

This is the rule that states that programs consist of one or more statements. We have changed this rule slightly compared to the same rule in the syntax directed interpreter by inserting the semicolon token and the action that emits code. In this form the rule states that every time we recognize a statement followed by a semicolon in the input stream we print out a semicolon followed by a newline character to the output. This classic syntax directed language processing: the actions are dictated by the syntactic structures recognized in the input stream.

The next group of rules specifies what statements look like,

\[
\text{stmt} : \\
\quad \text{'print'} \ {\text{emit}} ( \text{print } ) \{ \} \ \text{exp} \ |
\quad \text{'store'} \ \text{VAR} \ {\text{emit}} ( \text{store } + \text{VAR.text } + \text{* } ) \{ \} \ \text{exp} \ |
\]

In the first rule we emit the keyword print as soon as we recognized the token 'print' in the input stream. We then continue to process the input stream with the non-terminal exp. The second rule states that as soon as we recognized the tokens 'store' and VAR we emit the keyword store and the variable name then continue processing with the non-terminal exp.

The last group of rules in the grammar rule section specifies expressions,

\[
\text{exp} : \\
\quad \text{'+' } \{ \text{emit}(\text{+ } ) \} \ \text{exp} \ {\text{emit}} ( \text{* } )\{ \} \ \text{exp} \ {\text{emit}} ( \text{* } )\{ \} \ \text{exp} \ {\text{emit}} ( \text{* } )\{ \}
\]
\[
\quad \text{'-' } \{ \text{emit}(\text{- } )\} \ \text{exp} \ {\text{emit}} ( \text{* } )\{ \} \ \text{exp} \ {\text{emit}} ( \text{* } )\{ \} \ \text{exp} \ {\text{emit}} ( \text{* } )\{ \}
\]
\[
\quad \text{VAR} \ {\text{emit}} ( \text{VAR.text } )\{ \}
\]
\[
\quad \text{INTVAL} \ {\text{emit}} ( \text{INTVAL.text } )\{ \}
\]

Confidential and Copyrighted - © 2015 Franklin, Beedle and Associates - Draft September 14, 2015
The first rule specifies the addition operation. Here we emit output as soon as we recognize the token '+'. Recall that we want to rewrite the output in Lisp format. Therefore, instead of just emitting the plus sign we emit "(+ ", that is, an open parenthesis followed by the plus sign and a space character. We then continue processing with the first exp non-terminal. Once we have recognized the syntactic structure of the corresponding expression we emit a space character and then continue processing with the second exp non-terminal. Once we have recognized the syntactic structure of this second expression we emit the closing parenthesis. The second rule works identically except that we are dealing with subtraction. Given these two rules it is easy to see that the emitted code will have a Lisp like format in that addition and subtraction operations will always be surrounded by parentheses. The third rule is interesting. Here we recognize the syntactic structure of parenthesized expressions but we don’t emit any code for the parentheses. In essence we are deleting parentheses from the input program. These parentheses from the input program are superfluous because every non-trivial expression is already parenthesized in the output using the first two rules and therefore we do not emit them into the output. In the last two rules above we emit the strings of the recognized tokens VAR and INTVAL, respectively.

In order to get a deeper insight in how syntax directed translation works is perhaps best to envision the grammar of the pretty printer as a recursive descent parser. In that case the rule set for expressions from above could be viewed as the parsing function for expressions as follows:

```plaintext
function exp() returns void
begin
    switch inputToken()
    case PLUS:
        emit("(+ ")
        exp()
        emit(" ")
        exp()
        emit(")
        return
    case MINUS:
        emit("(- ")
        exp()
        emit(" ")
        exp()
        emit(")
        return
    case POPEN:
        exp()
        matchToken(PCLOSE)
        return
    case VAR:
        emit("VAR")
    end switch
end
```
2.5. Another Look at Grammars

2.5.1 Left-Recursion Elimination

Up to now we have sidestepped the issue of left-recursion because the programming languages we have considered so far were so simple and used pre-fix expression notation. However, in many instances left-recursion appears in grammar specifications when describing lists and other recursive syntactic structures. Consider the following grammar specification that defines lists of b’s that start with an a,

```
list : list 'b' | 'a'
```

Valid sentences in this grammar are: a, ab, abb, abbb, etc. (You should convince yourself that these are truly valid sentences by constructing the appropriate derivations).

One way to eliminate left-recursion in this grammar is to introduce a new non-terminal for the left-recursive symbol appearing in the body of the first rule.
and then using ANTLR’s list notation for lists with zero or more elements to
specify the list of b’s directly,

```plaintext
list : first ('b')* ;
first : 'a' ;
```

In this case we can write the grammar more compactly as,

```plaintext
list : 'a' ('b')* ;
```

It is easy to see that this grammar generates the same language as the original
left-recursive grammar.

Another example close to our hearts as programmers is the specification of
infix algebraic expressions. Perhaps the most intuitive way to specify the addi-
tion operator is as a left-recursive grammar,

```plaintext
exp : exp ' + ' exp |
| 'x' |
| 'y' |
| 'z' ;
```

This grammar specifies additive expressions in the variable x, y, and z. Valid sen-
tences in this grammar include: x, x+y, y+z, x+y+z, x+x+x+x. Now, in order
to eliminate left-recursion from this grammar we have to let go of our hierar-
chical notion of operator application and view an expression only in syntactical
terms. For example, instead of viewing the expression

```plaintext
x + x + x + x
```

we view as a list of syntactic entities,

```plaintext
x [+ x] [+ x] [+ x]
```

That is, the expression x + x + x + x can be viewed as x followed by three
instances of the syntactic entity + x. The use of the square brackets intends to
convey the fact that we are only looking at this from a syntactic point of view,
not from an algebraic point of view.

This insight allows us to rewrite our left-recursive expression grammar above
as,

```plaintext
exp : var (' + ' var)* ;
var : 'x' |
| 'y'
```
We eliminated the left-recursive symbol by introducing a new non-terminal for the variables and we replaced recursion with a list as a way to capture the structure of terms.

Adding subtraction to the grammar specification is straightforward with ANTLR's builtin notation,

\[
\text{exp} : \text{var} \ (\ (\ + \ \text{var}) \ | \ (\ - \ \text{var}) \ )^* \\
\text{var} : 'x' \\
| 'y' \\
| 'z'
\]

Here, an expression is an initial variable followed by zero or more syntactic terms of the form '+ var' or '- var. Beware that simply adding another rule to the grammar for subtraction does not work (see Exercise 17).

\subsection{2.5.2 Operator Associativity}

As soon as we are able to construct expressions with more than one operator of the same kind we run into the issue of operator associativity, that is, how is an expression evaluated that has more than one of the same kind of operator in it. From algebra we know that an expression of the form \(x + x + x + x\) is evaluated from left to right: \(((x + x) + x) + x\). In other words, the addition operator is left-associative.

Assume now that we would like to construct a syntax directed interpreter for infix, left associative addition expressions. More likely the grammar for such an interpreter would include the following snippet,

\[
\text{exp} : \text{var} \ \ (\ + \ \text{var})^* \\
\text{var} : 'x' \\
| 'y' \\
| 'z'
\]

From our work in the previous section it should be easy to see that this grammar admits expressions such as \(x + x + x + x\).

Since we are constructing a syntax directed interpreter we need to be able to return values from non-terminals and we need to set up the appropriate actions.
so that the values are set properly. One way to accomplish this and preserve the left-associativity of the addition operator is the following,

```plaintext
exp returns [Integer value]  
: v1=var { $value = $v1.value; }  
( '+' v2=var { $value += $v2.value; } )*  
;

var returns [Integer value]  
: 'x' { $value = lookup("x"); }  
| 'y' { $value = lookup("y"); }  
| 'z' { $value = lookup("z"); }  
;
```

The interesting part here is of course the first rule where we initialize the return value of the expression with the value of the first variable of the addition expression. The return value of the expression is then incremented by each subsequent variable value in the expression. In essence, we are processing the list of addition terms from left to right and therefore preserve the left-associativity of the addition operations.

This begs the immediate question of how to deal with right-associative operations. It turns out that right-associative operators are best described with right-recursive grammar rules in LL(1) grammars. Consider the C-style assignment operator. The C-style assignment operator is actually an expression and can appear anywhere where expressions are allowed. For example, in C the following code is legal,

```
a = 1 == 1
```

Here we assign the value one to the variable a. This assignment operation has a return value of one which then is compared to the value one. So, the overall value of this expression is the value one (C’s version of true).

Because assigning a value to a variable has a return value, namely the value that we assigned to the variable, this allows us to “daisy chain” assignments. Consider a program where the variables a, b, and c all need to be initialized to the value one. In C this can be accomplished in one statement,

```
a = b = c = 1;
```

But notice, in order for this statement to work the statement needs to be evaluated in this order: \(a = (b = (c = 1));\). The statement is evaluated from right to left implying that the assignment operator is right-associative. Here is an LL(1) grammar that would admit sentences like these,

```plaintext
assignexp  
: VAR '=' assignexp  
| INTVAL  
;
```
VAR : ( 'a'..'z') ;
INTVAL : ( '0'..'9')+ ;

Notice that the grammar is right-recursive in the assignexp non-terminal. You should convince yourself that parse tree constructed for the derivation of $a = b = c = 1$ is appropriate for the bottom-up interpretation of this sentence. We can also show this by constructing a syntax directed interpreter for this language,

```plaintext
assignexp returns [Integer value]
  : VAR '=' assignexp
      { update($VAR.text,$assignexp.value);
          $value = $assignexp.value;
      }
  | INTVAL
      { $value = new Integer($INTVAL.text);
      }
;  
VAR : ( 'a'..'z') ;
INTVAL : ( '0'..'9')+ ;
```

Here the right recursion computes a value for the non-terminal assignexp within the first rule. Once we have that value we update the variable with this new value and then continue to return this value as the return value for the current assignment expression. The recursion stops when we encounter an integer constant in the input stream.

### 2.5.3 Operator Precedence

In terms of grammars we have to look at one more case. The case where expressions consist of operators with different precedences. Consider expressions of the form $x + y \times z$. Algebra tells us that we need to evaluate this expression as $x + (y \times z)$. That is, operations with higher precedence are evaluated first. The straightforward approach of constructing a grammar that captures these kind of expressions DOES NOT work. Consider,

```plaintext
exp : var (('+ var) | ('* var))*
    ;
var : 'x'
    | 'y'
    | 'z'
    ;
```

The fact that this does not work is easy to see if you insert the actions for syntax directed interpretation,
exp returns [Integer value]
  : v1=var { $value = $v1.value; }
  | ('+' v2=var { $value += $v2.value; })
  | ('*' v3=var { $value *= $v3.value; })
  ;

var returns [Integer value]
  : 'x' { $value = lookup("x"); }
  | 'y' { $value = lookup("y"); }
  | 'z' { $value = lookup("z"); }
  ;

Here the expression \( x + y \ast z \) is processed from left to right and according to the actions the addition is performed before the multiplication which is not correct.

In order to get this right we have to rewrite the grammar in such a way that multiplication operations are evaluated before additions. The easiest way to accomplish this is to introduce a non-terminal for multiplication expressions and have this non-terminal appear within addition expressions. In this way, from a syntax directed point of view, if we want to evaluate addition expressions we first have to evaluate the multiplication expressions. Here is the rewritten grammar,

\[
\begin{align*}
\text{addexp} & : \text{multexp} (\text{'}+\text{'} \text{multexp})^* \\
\text{multexp} & : \text{var} (\text{'}\ast\text{'} \text{var})^* \\
\text{var} & : \text{'}x\text{' |
\text{'y'} | \text{'z'} \\
\end{align*}
\]

That this accomplishes exactly what we want becomes clear if we add the actions for syntax directed interpretation,

\[
\begin{align*}
\text{addexp returns [Integer value]}
  : e1=multexp { $value = $e1.value; }
  | ('+' e2=multexp { $value += $e2.value; })^* \\
\text{multexp returns [Integer value]}
  : e1=var { $value = $e1.value; }
  | ('\ast' e2=var { $value *= $e2.value; })^* \\
\end{align*}
\]
It is clear now that in order to evaluate an addition expression we first have to evaluate any multiplication expressions that exist in the overall expression. You should convince yourself that this works properly for the expression $x + y \times z$.

**Chapter Summary**

We started this chapter with a discussion of grammars and defining languages using grammars. Derivations within grammars allow us to show that a particular sentence is a valid sentence and belongs to the language of the grammar. Once we have a derivation it is straightforward to construct a parse tree and we can view parse trees as a visual representation of derivations.

The construction of derivations can be mechanized using a parsing algorithm. We classify parsing algorithm according to how they construct the associated derivation or parse tree: Top-down – starting at the grammar start symbol. Bottom-up – starting at the leaves of the parse tree. Top-down parsing algorithms are also called LL(k) algorithms in that they read the input from the (L)eft, construct the (L)eft-most derivation, and use (k) lookahead symbols. Bottom-up parsing algorithms are also called LR(k) algorithms because they read the input from the (L)eft, build the (R)ight-most derivation, and use (k) lookahead symbols. Here we studied the LL(1) and LR(0) parsing algorithms in more detail.

Practical parsing of programming languages is done as a two step process: First we perform the lexical analysis, that is, we analyze the structure of the language words. Then we perform the syntactic or phrase structure analysis of the program. The phrase or syntactical structure of a programming language is given using grammar rules while the lexical structure is specified using regular expressions.

When parsing languages with infix expressions we have to do some extra work if we want to use LL(1) parsers. The intuitive approach to describing infix expression require left-recursive rules which cannot be processed by LL(1) parsers. In ANTLR there is a straightforward way of turning left-recursive rules into rules describing a list structure. Furthermore, when dealing with infix expression we have to be careful with associativity and precedence information for
operators and let the grammar reflect this additional information.

**Bibliographic Notes**

A number of books introduce formal language theory, grammars, and derivations within grammars in addition to finite state machines. Perhaps the most accessible books are by Webber [17] and Sipser [14]. Most books on compiler construction discuss parsing theory in detail and introduce efficient algorithms for lexical and syntax analysis, e.g., [2, 3]. Donald Knuth’s seminal paper [8] introduced LR parsing in 1965 and made parsing practical and efficient. Terence Parr, the creator of ANTLR, continues to investigate LL parsing, see [12]. Grammars with respect to operator associativity and precedence are discussed in [18].

**Exercises**

1. Apply the lookahead set algorithm given in Figure 1.4 and Figure 1.5 to grammar (1.79) through (1.86) and compute the lookahead sets.

2. Apply the lookahead set algorithm given in Figure 1.4 and Figure 1.5 to grammar (1.19) through (1.41) and compute the lookahead sets.

3. Show that the lookahead set algorithm given in Figure 1.4 and Figure 1.5 recurses indefinitely given the grammar (1.95) through (1.97).

4. Compute the lookahead sets for
   (a) grammar (1.104) through (1.109)
   (b) grammar (1.110) through (1.116)

5. Rewrite the grammar (1.19) through (1.41) as a LR(0) grammar and then compute a trace of the bottom-up parsing algorithm in Figure 1.8 similar to Figure 1.9 that uses your new LR(0) grammar and the input stream p + 1 2 ; \texttt{eof}.

6. Rewrite the grammar (1.128) through (1.132) as a LR(0) grammar using factoring and then compute a trace of the algorithm in Figure 1.8 similar to Figure 1.9 that uses your LR(0) grammar and the input stream \texttt{if e then p else q \texttt{eof}} (Hint: we used factoring in top-down parsing in order to construct LL(1) grammars).
7. LL(0) parsers are parsers that read input from left to right, construct the left-most derivation but use no lookahead symbols. What does that mean for the kind of grammars such a parser could use for parsing?

8. Rewrite the lookahead set algorithm from Figure 1.4 and Figure 1.5 in such a way that it detects left-recursion, issues an error message, and terminates. The extension should be general enough so that it also detects left-recursion in mutual recursive rules.

9. We have shown that if-then-else statements as in grammar (1.128) through (1.132) create shift-reduce conflicts in LR(0) parsers. Show that right-recursive rules such as

\[
\begin{align*}
\text{List} & \rightarrow \text{Item} & (2.136) \\
& \mid \text{Item List} & (2.137) \\
\text{Item} & \rightarrow \text{e} & (2.138)
\end{align*}
\]

also create a shift-reduce conflict in our LR(0) parser. Compute a trace similar to Figure 1.10 for the input stream \text{e e e \textbackslash eof}. Propose a way to rewrite the grammar in order to eliminate the conflict.

10. Research the FIRST and FOLLOW set construction and then extend the LR(0) parsing algorithm in Figure 1.8 to a LR(1) parser by accommodating grammars extended with FOLLOW sets.

11. Design a lexical rule that defines a token for the binary representation of positive integer numbers.

12. Design a lexical rule defines a token for the floating point numbers (don't include the scientific notation).

13. Design an NFSM for the regular expression from exercise 11.

14. Design an NFSM for the regular expression from exercise 12.

15. Design an NFSM for each regular expression in the list of basic definitions of regular expressions on page 49. If a definition assumes preexisting regular expressions such as \( A \), then you can assume a previous NFSM \( M_A \) with a single input and a single output.

16. Rewrite the left-recursive grammar specification,
as a LL(1) grammar specification.

17. Show that the sentence \( x + y - z \) is not a valid sentence in the grammar,

\[
\text{exp} : \quad \text{var} \ (\text{'}+\text{'} \ \text{var})^* \\
| \quad \text{var} \ (\text{'}-\text{'} \ \text{var})^* \\
\]

\[
\text{var} : \quad 'x' \\
| \quad 'y' \\
| \quad 'z' \\
\]

18. (project) Implement a simple translator that reads an LL(1) grammar file, computes the lookahead sets, and outputs the extended grammar.

19. (project) Implement an LL(1) parser using the algorithm given in Figure 1.6 that reads an LL(1) grammar extended with lookahead sets and an input stream and builds a parse tree if the program in the input stream is a valid sentence in the language of the grammar.

20. (project) Implement an LR(0) parser using the algorithm given in Figure 1.8 that reads an LR(0) grammar and an input stream and builds a parse tree if the program in the input stream is a valid sentence in the language of the grammar.

21. (project) Write a program that reads a grammar file and flags the all rules that either create an LR(0) reduce-reduce conflict or a shift-reduce conflict.

22. (project) Reduce-reduce conflicts in bottom-up parsers can be resolved by guessing and backtracking: guess one alternative and try to parse the sentence; if this is not possible backtrack to the decision point and try another alternative. Extend the algorithm with backtracking on reduce-reduce conflicts and implement a parser using your extended algorithm that reads a grammar and an input stream and returns \textit{true} if the program in the input stream is a valid sentence in the language of the grammar otherwise \textit{false}. 

Confidential and Copyrighted - © 2015 Franklin, Beedle and Associates - Draft September 14, 2015
23. (project) Build a recursive descent parser for the Exp0 programming language in your favorite programming language. Show that it works by parsing some sentences that are valid in Exp0. Show that it rejects sentences with syntax errors.

24. (project) Modify your recursive descent parser from Exercise 23 to construct parse trees for input programs.

25. (project) Download the code for the Exp1 interpreter from the book website and extend the language with multiplication and integer division. Demonstrate that your interpreter works by running it on some telling examples.

26. (project) Take the token definitions in Figure 1.18 and implement a lexer based on a recursive descent parser.

27. (project) Rewrite the grammar in Figure 1.18 in such a way that it supports infix expressions and then construct a syntax directed interpreter for it.

28. (project) Rewrite the grammar in Figure 1.18 in such a way that it supports
   (a) the infix operations ‘*’ and ‘/’, multiplication and divide, respectively, as well as addition and subtraction.
   (b) properly encodes associativity and presence of all the operators.

and then construct a syntax directed interpreter for it.