Syntax

2

Once you've learned to program in one language, learning a similar programming language isn't all that hard. But, understanding just how to write in the new language takes looking at examples or reading documentation to learn its details. In other words, you need to know the mechanics of putting a program together in the new language. Are the semicolons in the right places? Do you use *begin...end* or do you use curly braces (i.e. { and })? Learning how a program is put together is called learning the syntax of the language. Syntax refers to the words and symbols of a language and how to write the symbols down in some meaningful order.

Semantics is the word that is used when deriving meaning from what is written. The semantics of a program refers to what the program will do when it is executed. Informally it is much easier to say what a program does than to describe the syntactic structure of the program. However, syntax is a lot easier to formally describe than semantics. In either case, if you are learning a new language, you need to learn something about both the syntax and semantics of the language.

2.1 Terminology

Once again, the *syntax* of a programming language determines the well-formed or grammatically correct programs of the language. *Semantics* describes how or whether such programs will execute.

- *Syntax* is how programs look
- *Semantics* is how programs work

Many questions we might like to ask about a program either relate to the syntax of the language or to its semantics. It is not always clear which questions pertain to

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_2

syntax and which pertain to semantics. Some questions may concern semantic issues that can be determined statically, meaning before the program is run. Other semantic issues may be dynamic issues, meaning they can only be determined at run-time. The difference between static semantic issues and syntactic issues is sometimes a difficult distinction to make.

The code

```
a=b+c;
```

is correct syntax in many languages. But is it a correct C++ statement?

1. Do *b* and *c* have values?
2. Have *b* and *c* been declared as a type that allows the + operation? Or, do the values of *b* and *c* support the + operation?
3. Is *a* assignment compatible with the result of the expression $b + c$?
4. Does the assignment statement have the proper form?

There are lots of questions that need to be answered about this assignment statement. Some questions could be answered sooner than others. When a C++ program is compiled it is translated from C++ to machine language as described in the previous chapter. Questions 2 and 3 are issues that can be answered when the C++ program is compiled. However, the answer to the first question might not be known until the C++ program executes in some cases. The answers to questions 2 and 3 can be answered at *compile-time* and are called *static* semantic issues. The answer to question 1 is a *dynamic* issue and is probably not determinable until run-time. In some circumstances, the answer to question 1 might also be a static semantic issue. Question 4 is definitely a syntactic issue.

Unlike the dynamic semantic issues, the correct syntax of a program is statically determinable. Said another way, determining a syntactically valid program can be accomplished without running the program. The syntax of a programming language is specified by a grammar. But before discussing grammars, the parts of a grammar must be defined. A *terminal* or *token* is a symbol in the language.

- C++, Java, and Python terminals: *while*, *for*, *(*, *;*, *5*, *b*
- Type names like *int* and *string*

Keywords, types, operators, numbers, identifiers, etc. are all tokens or terminals in a language.

A *syntactic category* or *nonterminal* is a set of phrases, or strings of tokens, that will be defined in terms of symbols in the language (terminal and nonterminal symbols).

- C++, Java, or Python nonterminals: <statement>, <expression>, <if-statement>, etc.
- Syntactic categories define parts of a program like statements, expressions, declarations, and so on.

A *metalanguage* is a higher-level language used to specify, discuss, describe, or analyze another language. English is used as a metalanguage for describing programming languages, but because of the ambiguities in English, more formal metalanguages have been developed. The next section describes a formal metalanguage for describing programming language syntax.

2.2 Backus Naur Form (BNF)

Backus Naur Format (i.e. BNF) is a formal metalanguage for describing language syntax. The word *formal* is used to indicate that BNF is unambiguous. Unlike English, the BNF language is not open to our own interpretations. There is only one way to read a BNF description.

BNF was used by John Backus to describe the syntax of Algol in 1963. In 1960, John Backus and Peter Naur, a computer magazine writer, had just attended a conference on Algol. As they returned from the trip it became apparent that they had very different views of what Algol would look like. As a result of this discussion, John Backus worked on a method for describing the grammar of a language. Peter Naur slightly modified it. The notation is called BNF, or Backus Naur Form or sometimes Backus Normal Form. BNF consists of a set of rules that have this form:

<syntactic category> ::= a string of terminals and nonterminals

The symbol ::= can be read as *is composed of* and means the syntactic category is the set of all items that correspond to the right hand side of the rule.

Multiple rules defining the same syntactic category may be abbreviated using the | character which can be read as "or" and means set union. That is the entire language. It's not a very big metalanguage, but it is powerful.

2.2.1 BNF Examples

Here are a couple BNF examples from Java.

```
<primitive-type> ::= boolean
<primitive-type> ::= char
```

BNF syntax is often abbreviated when there are multiple similar rules like these primitive type rules. Whether abbreviated or not, the meaning is the same.

```
<primitive-type> ::= boolean | char | byte | short | int | long | float | ...
<argument-list> ::= <expression> | <argument-list> , <expression>
<selection-statement> ::=
  if ( <expression> ) <statement> |
  if ( <expression> ) <statement> else <statement> |
  switch ( <expression> ) <block>
<method-declaration> ::=
  <modifiers> <type-specifier> <method declarator> <throws-clause> <method-body> |
  <modifiers> <type-specifier> <method-declarator> <method-body> |
```

```
<type-specifier> <method-declarator> <throws-clause> <method-body> |
<type-specifier> <method-declarator> <method-body>
```

This description can be described in English: *The set of method declarations is the union of the sets of method declarations that explicitly throw an exception with those that don't explicitly throw an exception with or without modifiers attached to their definitions*. The BNF is much easier to understand and is not ambiguous like this English description.

2.2.2 Extended BNF (EBNF)

Since a BNF description of the syntax of a programming language relies heavily on recursion to provide lists of items, many definitions use these extensions:

1. **item?** or **[item]** means the item is optional.
2. **item*** or **{item}** means zero or more occurrences of an item are allowable.
3. **item+** means one or more occurrences of an item are allowable.
4. Parentheses may be used for grouping

2.3 Context-Free Grammars

A BNF is a way of describing the grammar of a language. Most interesting grammars are context-free, meaning that the contents of any syntactic category in a sentence are not dependent on the context in which it is used. A context-free grammar is defined as a four tuple:

$$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$$

where

- $\mathcal{N}$ is a set of symbols called nonterminals or syntactic categories.
- $\mathcal{T}$ is a set of symbols called terminals or tokens.
- $\mathcal{P}$ is a set of productions of the form $n \rightarrow \alpha$ where $n \in \mathcal{N}$ and $\alpha \in \{\mathcal{N} \cup \mathcal{T}\}^*$.
- $\mathcal{S} \in \mathcal{N}$ is a special nonterminal called the start symbol of the grammar.

Informally, a context-free grammar is a set of nonterminals and terminals. For each nonterminal there are one or more productions with strings of zero or more nonterminals and terminals on the right hand side as described in the BNF description. There is one special nonterminal called the start symbol of the grammar.

2.3.1 The Infix Expression Grammar

A context-free grammar for infix expressions can be specified as $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, T, F\}$
$\mathcal{T} = \{identifier, number, +, -, *, /, (,)\}$
$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid identifier \mid number$$

2.4 Derivations

A *sentence* of a grammar is a string of tokens from the grammar. A sentence belongs to the language of a grammar if it can be derived from the grammar. This process is called constructing a derivation. A *derivation* is a sequence of sentential forms that starts with the start symbol of the grammar and ends with the sentence you are trying to derive. A *sentential form* is a string of terminals and nonterminals from the grammar. In each step in the derivation, one nonterminal of a sentential form, call it A, is replaced by a string of terminals and nonterminals, β, where $A \rightarrow \beta$ is a production in the grammar. For a grammar, G, the language of G is the set of sentences that can be derived from G and is usually written as *L(G)*.

2.4.1 A Derivation

Here we prove that the expression $(5 * x) + y$ is a member of the language defined by the grammar given in Sect. 2.3.1 by constructing a derivation for it. The derivation begins with the start symbol of the grammar and ends with the sentence.

$$\underline{E} \Rightarrow \underline{E} + T \Rightarrow \underline{T} + T \Rightarrow \underline{F} + T \Rightarrow (\underline{E}) + T \Rightarrow (\underline{T}) + T \Rightarrow (\underline{T} * F) + T$$
$$\Rightarrow (\underline{F} * F) + T \Rightarrow (5 * \underline{F}) + T \Rightarrow (5 * x) + \underline{T} \Rightarrow (5 * x) + \underline{F} \Rightarrow (5 * x) + y$$

Each step is a sentential form. The underlined nonterminal in each sentential form is replaced by the right hand side of a production for that nonterminal. The derivation proceeds from the start symbol, E, to the sentence $(5 * x) + y$. This proves that $(5 * x) + y$ is in the language *L(G)* as G is defined in Sect. 2.3.1.

Practice 2.1 Construct a derivation for the infix expression $4 + (a - b) * x$.
You can check your answer(s) in Section 2.17.1.

2.4.2 Types of Derivations

A sentence of a grammar is *valid* if there exists at least one derivation for it using the grammar. There are typically many different derivations for a particular sentence of a grammar. However, there are two derivations that are of some interest to us in understanding programming languages.

- Left-most derivation - Always replace the left-most nonterminal when going from one sentential form to the next in a derivation.
- Right-most derivation - Always replace the right-most nonterminal when going from one sentential form to the next in a derivation.

The derivation of the sentence $(5 * x) + y$ in Sect. 2.4.1 is a left-most derivation. A right-most derivation for the same sentence is:

$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + y \Rightarrow T + y \Rightarrow F + y \Rightarrow (E) + y \Rightarrow (T) + y$$
$$\Rightarrow (T * F) + y \Rightarrow (T * x) + y \Rightarrow (F * x) + y \Rightarrow (5 * x) + y$$

Practice 2.2 Construct a right-most derivation for the expression $x * y + z$.
You can check your answer(s) in Section 2.17.2.

2.4.3 Prefix Expressions

Infix expressions are expressions where the operator appears between the operands. Another type of expression is called a prefix expression. In prefix expressions the operator appears before the operands. The infix expression $4 + (a - b) * x$ would be written $+4 * -abx$ as a prefix expression. Prefix expressions are in some sense simpler than infix expressions because we don't have to worry about the precedence of operators. The operator precedence is determined by the order of operations in the expression. Because of this, parentheses are not needed in prefix expressions.

2.4.4 The Prefix Expression Grammar

A context-free grammar for prefix expressions can be specified as $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E\}$
$\mathcal{T} = \{identifier, number, +, -, *, /\}$
$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow + \; E \; E \mid - \; E \; E \mid * \; E \; E \mid / \; E \; E \mid identifier \mid number$$

Practice 2.3 Construct a left-most derivation for the prefix expression $+4 * -abx$.
You can check your answer(s) in Section 2.17.3.

2.5 Parse Trees

A grammar, G, can be used to build a tree representing a sentence of $L(G)$, the language of the grammar G. This kind of tree is called a *parse tree*. A parse tree is another way of representing a sentence of a given language. A parse tree is constructed with the start symbol of the grammar at the root of the tree. The children of each node in the tree must appear on the right hand side of a production with the parent on the left hand side of the same production. A program is syntactically valid if there is a parse tree for it using the given grammar.

While there are typically many different derivations of a sentence in a language, there is only one parse tree. This is true as long as the grammar is not ambiguous. In fact that's the definition of ambiguity in a grammar. A grammar is *ambiguous* if and only if there is a sentence in the language of the grammar that has more than one parse tree.

The parse tree for the sentence derived in Sect. 2.4.1 is depicted in Fig. 2.1. Notice the similarities between the derivation and the parse tree.

Practice 2.4 What does the parse tree look like for the right-most derivation of $(5 * x) + y$?
You can check your answer(s) in Section 2.17.4.

Practice 2.5 Construct a parse tree for the infix expression $4 + (a - b) * x$.
HINT: What has higher precedence, "+" or "*"? The given grammar automatically makes "*" have higher precedence. Try it the other way and see why!
You can check your answer(s) in Section 2.17.5.

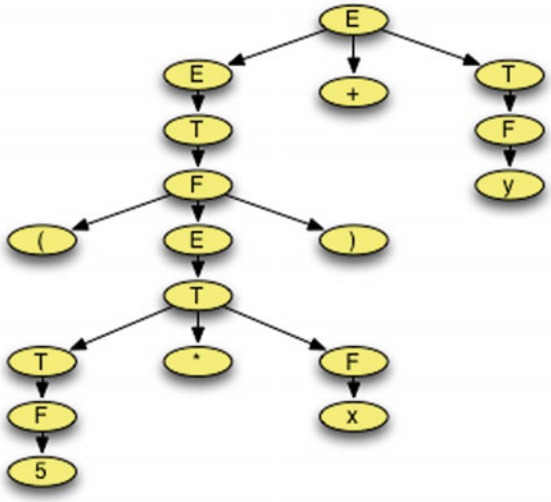


Fig. 2.1 A parse tree

Practice 2.6 Construct a parse tree for the prefix expression $+4 * -abx$.
You can check your answer(s) in Section 2.17.6.

2.6 Abstract Syntax Trees

There is a lot of information in a parse tree that isn't really needed to capture the meaning of the program that it represents. An abstract syntax tree is like a parse tree except that non-essential information is removed. More specifically,

- Nonterminal nodes in the tree are replaced by nodes that reflect the part of the sentence they represent.
- Unit productions in the tree are collapsed.

For example, the parse tree from Fig. 2.1 can be represented by the abstract syntax tree in Fig. 2.2. The abstract syntax tree eliminates all the unnecessary information and leaves just what is essential for evaluating the expression. Abstract syntax trees, often abbreviated ASTs, are used by compilers while generating code and may be used by interpreters when running your program. Abstract syntax trees throw away superfluous information and retain only what is essential to allow a compiler to generate code or an interpreter to execute the program.

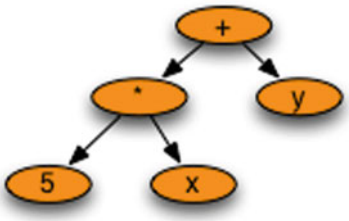


Fig. 2.2 An AST

Practice 2.7 Construct an abstract syntax tree for the expression $4+(a-b)*x$.
You can check your answer(s) in Section 2.17.7.

2.7 Lexical Analysis

The syntax of modern programming languages are defined via grammars. A grammar, because it is a well-defined mathematical structure, can be used to construct a program called a parser. A language implementation, like a compiler or an interpreter, has a parser that reads the program from the source file. The parser reads the tokens, or terminals, of a program and uses the language's grammar to check to see if the stream of tokens form a syntactically valid program.

For a parser to do its job, it must be able to get the stream of tokens from the source file. Forming tokens from the individual characters of a source file is the job of another program often called a tokenizer, or scanner, or lexer. Lex is the Latin word for *word*. The words of a program are its tokens. In programming language implementations a little liberty is taken with the definition of *word*. A *word* is any terminal or token of a language. It turns out that the tokens of a language can be described by another language called the language of regular expressions.

2.7.1 The Language of Regular Expressions

The language of regular expression is defined by a context-free grammar. The context-free grammar for regular expressions is $RE = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, T, K, F\}$
$\mathcal{T} = \{character, *, +, ., (,)\}$
$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T.K \mid K$$
$$K \rightarrow F* \mid F$$
$$F \rightarrow character \mid (\ E\ )$$

The + operator is the *choice* operator, meaning either E or T, but not both. The dot operator means that T is *followed by* K. The $*$ operator, called *Kleene Star* for the mathematician that first defined it, means *zero or more occurrences* of F. The grammar defines the precedence of these operators. Kleene star has the highest precedence followed by the dot operator, followed by the choice operator. At its most primitive level, a regular expression may be just a single character.

Frequently, a choice between many different characters may be abbreviated with some sensible name. For instance, *letter* may be used to abbreviate $A + B + \cdots + Z + a + b + \cdots z$ and digit may abbreviate $0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$. Usually these abbreviations are specified explicitly before the regular expression is given.

The tokens of the infix grammar are identifier, number, +, −, $*$, /, (, and). For brevities sake, assume that letter and digit have the usual definitions. We'll also put each operator character in single quotes so as not to confuse them with the metalanguage. Then, these tokens might be defined by the regular expression

letter.letter* + digit.digit* + '+' + '−' + '*' + '/' + '(' + ')'

From this regular expression specification a couple of things come to light. Identifiers must be at least one character long, but can be as long as we wish them to be. Numbers are only non-negative integers in the infix expression language. Floating point numbers cannot be specified in the language as the tokens are currently defined.

Practice 2.8 Define a regular expression so that negative and non-negative integers can both be specified as tokens of the infix expression language.

You can check your answer(s) in Section 2.17.8.

2.7.2 Finite State Machines

A finite state machine is a mathematical model that accepts or rejects strings of characters for some regular expression. A finite state machine is often called a finite state automaton. The word *automaton* is just another word for machine. Every regular expression has at least one finite state machine and vice versa, every finite state machine has at least one matching regular expression. In fact, there is an algorithm that given any regular expression can be used to construct a finite state machine for it.

Formally a finite state automata is defined as follows.

$M = (\Sigma, S, F, s_0, \delta)$ where Σ (pronounced sigma) is the input alphabet (the characters understood by the machine), S is a set of states, F is a subset of S usually written as $F \subseteq S$, s_0 is a special state called the start state, and δ (pronounced delta) is a function that takes as input an alphabet symbol and a state and returns a new state. This is usually written as $\delta : \Sigma \times S \rightarrow S$.

A finite state machine has a current state which initially is the start state. The machine starts in the start state and reads characters one at a time. As characters are read, the finite state machine changes state. Each state has transitions to other states based on the last character read. Each time the machine transitions to a new state, another character is read from the stream of characters.

After reading all the characters of a token, if the current state is in the set of final states, F, then the token is accepted by the finite state machine. Otherwise, it is rejected. Finite state machines are typically represented graphically by drawing the states, transitions, start state, and final states. States in a graphical representation are depicted as nodes in a graph. The start state has an arrow going into it with nothing at the back side of the arrow. The transitions are represented as arrows going from one state to another and are labelled with the characters that trigger the given transition. Finally, final or accepting states are denoted with a double circle.

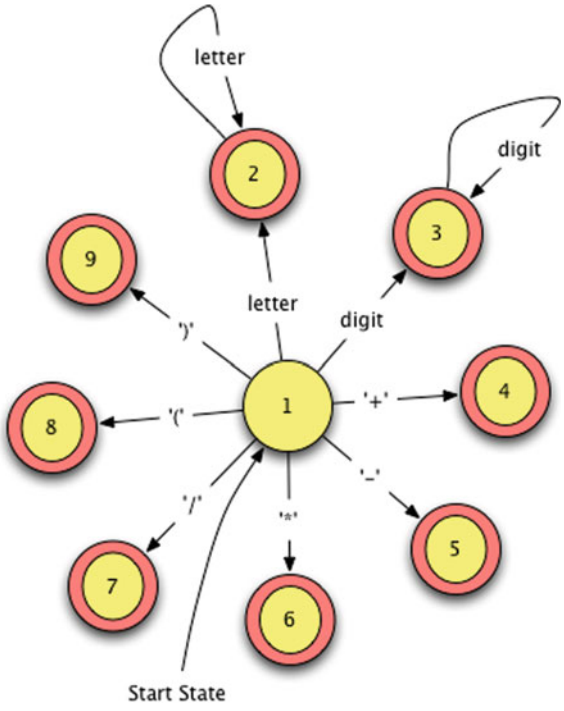


Fig. 2.3 A finite state machine

Figure 2.3 depicts a finite state machine for the language of infix expression tokens. The start state is 1. Each of states 2 through 9 are accepting states, denoted with a double circle. State 2 accepts identifier tokens. State 3 accepts number tokens. States 4 to 9 accept operators and the parenthesis tokens. The finite state machine accepts one token at a time. For each new token, the finite state machine starts over in state 1.

If, while reading a token, an unexpected character is read, then the stream of tokens is rejected by the finite state machine as invalid. Only valid strings of characters are accepted as tokens. Characters like spaces, tabs, and newline characters are not recognized by the finite state machine. The finite state machine only responds with *yes* the string of tokens is in the language accepted by the machine or *no* it is not.

2.7.3 Lexer Generators

It is relatively easy to construct a lexer by writing a regular expression, drawing a finite state machine, and then writing a program that mimics the finite state machine. However, this process is largely the same for all programming languages so there are tools that have been written to do this for us. Typically these tools are called lexer generators. To use a lexer generator you must write regular expressions for the tokens of the language and provide these to the lexer generator.

A lexer generator will generate a lexer program that internally uses a finite state machine like the one pictured in Fig. 2.3, but instead of reporting *yes* or *no*, for each token the lexer will return the string of characters, called the *lexeme* or *word* of the token, along with a classification of the token. So, identifiers are categorized as *identifier* tokens while '+' is categorized as an *add* token.

The *lex* tool is an example of a lexical generator for the C language. If you are writing an interpreter or compiler using C as the implementation language, then you would use *lex* or a similar tool to generate your lexer. *lex* was a tool included with the original *Unix* operating system. The Linux alternative is called *flex*. Java, Python, Standard ML, and most programming languages have equivalent available lexer generators.

2.8 Parsing

Parsing is the process of detecting whether a given string of tokens is a valid sentence of a grammar. Every time you compile a program or run a program in an interpreter the program is first parsed using a parser. When a parser isn't able to parse a program the programmer is told there is a *syntax error* in the program. A *parser* is a program that given a sentence, checks to see if the sentence is a member of the language of the given grammar. A parser usually does more than just answer *yes* or *no*. A parser frequently builds an abstract syntax tree representation of the source program. There are two types of parsers that are commonly constructed.

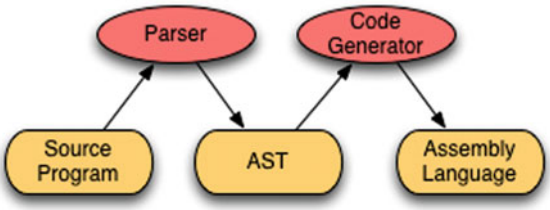


Fig. 2.4 Parser data flow

- A *top-down parser* starts with the root of the parse tree.
- A *bottom-up parser* starts with the leaves of the parse tree.

Top-down and bottom-up parsers check to see if a sentence belongs to a grammar by constructing a derivation for the sentence, using the grammar. A parser either reports success (and possibly returns an abstract syntax tree) or reports failure (hopefully with a nice error message). The flow of data is pictured in Fig. 2.4.

2.9 Top-Down Parsers

Top-down parsers are generally written by hand. They are sometimes called recursive descent parsers because they can be written as a set of mutually recursive functions. A top-down parser performs a left-most derivation of the sentence (i.e. source program).

A top-down parser operates by (possibly) looking at the next token in the source file and deciding what to do based on the token and where it is in the derivation. To operate correctly, a top-down parser must be designed using a special kind of grammar called an LL(1) grammar. An LL(1) grammar is simply a grammar where the next choice in a left-most derivation can be deterministically chosen based on the current sentential form and the next token in the input. The first *L* refers to scanning the input from left to right. The second *L* signifies that while performing a left-most derivation, there is only *1* symbol of lookahead that is needed to make the decision about which production to choose next in the derivation.

2.9.1 An LL(1) Grammar

The grammar for prefix expressions is LL(1). Examine the prefix expression grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E\}$
$\mathcal{T} = \{identifier, number, +, -, *, /\}$
$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow + \; E \; E \mid - \; E \; E \mid * \; E \; E \mid / \; E \; E \mid identifier \mid number$$

While constructing any derivation for a sentence of this language, the next production chosen in a left-most derivation is going to be obvious because the next token of the source file must match the first terminal in the chosen production.

2.9.2 A Non-LL(1) Grammar

Some grammars are not LL(1). The grammar for infix expressions is not LL(1). Examine the infix expression grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, T, F\}$
$\mathcal{T} = \{identifier, number, +, -, *, /, (,)\}$
$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow E \; + \; T \mid E \; - \; T \mid T$$
$$T \rightarrow T \; * \; F \mid T \; / \; F \mid F$$
$$F \rightarrow (\; E \;) \mid identifier \mid number$$

Consider the infix expression $5 * 4$. A left-most derivation of this expression would be

$$E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow 5 * F \Rightarrow 5 * 4$$

Consider looking at only the 5 in the expression. We have to choose whether to use the production $E \rightarrow E \; + \; T$ or $E \rightarrow T$. We are only allowed to look at the 5 (i.e. we can't look beyond the 5 to see the multiplication operator). Which production do we choose? We can't decide based on the 5. Therefore the grammar is not LL(1).

Just because this infix expression grammar is not LL(1) does not mean that infix expressions cannot be parsed using a top-down parser. There are other infix expression grammars that are LL(1). In general, it is possible to transform any context-free grammar into an LL(1) grammar. It is possible, but the resulting grammar is not always easily understandable.

The infix grammar given in Sect. 2.9.2 is left recursive. That is, it contains the production $E \rightarrow E + T$ and another similar production for terms in infix expressions. These rules are left recursive. Left recursive rules are not allowed in LL(1) grammars. A left recursive rule can be eliminated in a grammar through a straightforward transformation of its production.

Common prefixes in the right hand side of two productions for the same nonterminal are also not allowed in an LL(1) grammar. The infix grammar given in Sect. 2.9.2 does not contain any common prefixes. Common prefixes can be eliminated by introducing a new nonterminal to the grammar, replacing all common prefixes with the new nonterminal, and then defining one new production so the new nonterminal is composed of the common prefix.

2.9.3 An LL(1) Infix Expression Grammar

The following grammar is an LL(1) grammar for infix expressions. $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, RestE, T, RestT, F\}$
$\mathcal{T} = \{identifier, number, +, -, *, /, (,)\}$
$\mathcal{P}$ is defined by the set of productions

$$
\begin{aligned}
&E \rightarrow T\ RestE \\
&RestE \rightarrow +\ T\ RestE \mid -\ T\ RestE \mid \epsilon \\
&T \rightarrow F\ RestT \\
&RestT \rightarrow *\ F\ RestT \mid /\ F\ RestT \mid \epsilon \\
&F \rightarrow (\ E\ ) \mid identifier \mid number
\end{aligned}
$$

In this grammar the ϵ (pronounced epsilon) is a special symbol that denotes an empty production. An empty production is a production that does not consume any tokens. Empty productions are sometimes convenient in recursive rules.

Once common prefixes and left recursive rules are eliminated from a context-free grammar, the grammar will be LL(1). However, this transformation is not usually performed because there are more convenient ways to build a parser, even for non-LL(1) grammars.

Practice 2.9 Construct a left-most derivation for the infix expression $4 + (a - b) * x$ using the grammar in Sect. 2.9.3, proving that this infix expression is in L(G) for the given grammar.

You can check your answer(s) in Section 2.17.9.

2.10 Bottom-Up Parsers

While the original infix expression language is not *LL(1)* it is *LALR(1)*. In fact, most grammars for programming languages are LALR(1). The *LA* stands for *look ahead* with the *1* meaning just one symbol of look ahead. The *LR* refers to scanning the input from left to right while constructing a right-most derivation. A bottom-up parser constructs a right-most derivation of a source program in reverse. So, an LALR(1) parser constructs a reverse right-most derivation of a program.

Building a bottom-up parser is a somewhat complex task involving the computation of item sets, look ahead sets, a finite state machine, and a stack. The finite state machine and stack together are called a *pushdown automaton*. The construction of the pushdown automaton and the look ahead sets are calculated from the grammar. Bottom-up parsers are not usually written by hand. Instead, a parser generator is used

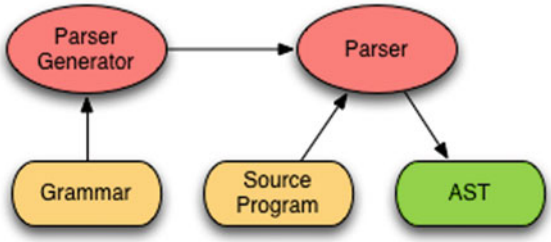


Fig. 2.5 Parser generator data flow

to generate the parser program from the grammar. A parser generator is a program that is given a grammar and builds a parser for the language of the grammar by constructing the pushdown automaton and lookahead sets needed to parse programs in the language of the grammar.

The original parser generator for Unix was called *yacc*, which stood for *yet another compiler compiler* since it was a compiler for grammars that produced a parser for a language. Since a parser is part of a compiler, *yacc* was a compiler compiler. The Linux version of yacc is called *Bison*. Hopefully you see the pun that was used in naming it *Bison*. The *Bison* parser generator generates a parser for compilers implemented in C, C++, or Java. There are versions of yacc for other languages as well. Standard ML has a version called ml-yacc for compilers implemented in Standard ML. ML-yacc is introduced and used in Chap. 6.

Parser generators like Bison produce what is called a bottom-up parser because the right-most derivation is constructed in reverse. In other words, the derivation is done from the bottom up. Usually, a bottom-up parser is going to return an AST representing a successfully parsed source program. Figure 2.5 depicts the dataflow in an interpreter or compiler. The parser generator is given a grammar and runs once to build the parser. The generated parser runs each time a source program is parsed.

A bottom-up parser parses a program by constructing a reverse right-most derivation of the source code. As the reverse derivation proceeds the parser *shifts* tokens from the input onto the stack of the pushdown automaton. Then at various points in time it reduces by deciding, based on the look ahead sets, that a reduction is necessary.

2.10.1 Parsing an Infix Expression

Consider the grammar for infix expressions as $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, T, F\}$
$\mathcal{T} = \{identifier, number, +, -, *, /, (,)\}$
$\mathcal{P}$ is defined by the set of productions

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow number$
(6) $F \rightarrow (E)$

Now assume we are parsing the expression $5 * 4 + 3$. A right-most derivation for this expression is as follows.

$$E \Rightarrow E+T \Rightarrow E+F \Rightarrow E+3 \Rightarrow T+3 \Rightarrow T*F+3 \Rightarrow T*4+3 \Rightarrow F*4+3 \Rightarrow 5*4+3$$

A bottom-up parser does a right-most derivation in reverse using a pushdown automaton. It can be useful to look at the stack of the pushdown automaton as it parses the expression as pictured in Fig. 2.6. In step A the parser is beginning. The dot to the left of the 5 indicates the parser has not yet processed any tokens of the source program and is looking at the 5. The stack is empty. From step A to step B one token, the 5 is shifted onto the stack. From step B to C the parser looks at the multiplication operator and realizes that a *reduction* using rule 5 of the grammar must be performed. It is called a reduction because the production is employed in reverse order. The reduction pops the right hand side of rule 5 from the stack and replaces it with the nonterminal F. If you look at this derivation in reverse order, the first step is to replace the number 5 with F.

The rest of the steps of parsing the source program follow the right-most derivation either shifting tokens onto the stack or reducing using rules of the grammar. In step O the entire source has been parsed, the stack is empty, and the source program is accepted as a valid program. The actions taken while parsing include shifting and reducing. These are the two main actions of any bottom-up parser. In fact, bottom-up parsers are often called shift-reduce parsers.

Practice 2.10 For each step in Fig. 2.6, is there a shift or reduce operation being performed? If it is a reduce operation, then what production is being reduced? If it is a shift operation, what token is being shifted onto the stack?

You can check your answer(s) in Section 2.17.10.

Practice 2.11 Consider the expression $(6 + 5) * 4$. What are the contents of the pushdown automaton's stack as the expression is parsed using a bottom-up parser? Show the stack after each shift and each reduce operation.

You can check your answer(s) in Section 2.17.11.

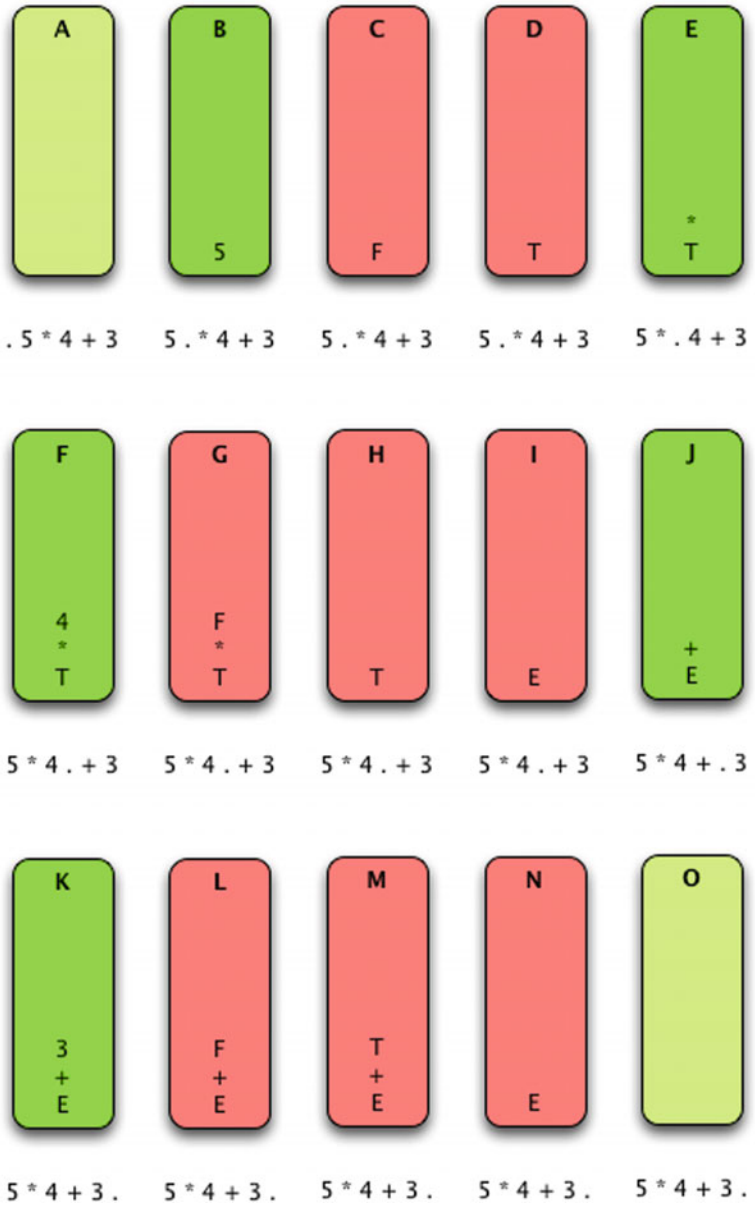


Fig. 2.6 A pushdown automaton stack

2.11 Ambiguity in Grammars

A grammar is ambiguous if there exists more than one parse tree for a given sentence of the language. In general, ambiguity in a grammar is a bad thing. However, some ambiguity may be allowed by parser generators for LALR(1) languages.

A classic example of ambiguity in languages arises from nested if-then-else statements. Consider the following Pascal statement:

```
if a<b then
   if b<c then
      writeln("a<c")
else
   writeln("?")
```

Which *if* statement does the *else* go with? It's not entirely clear. The BNF for an if-then-else statement might look something like this.

```
<statement> ::= if <expression> then <statement> else <statement>
              | if <expression> then <statement>
              | writeln ( <expression> )
```

The recursive nature of this rule means that if-then-else statements can be arbitrarily nested. Because of this recursive definition, the *else* in this code is dangling. That is, it is unclear if it goes with the first or second *if* statement.

When a bottom-up parser is generated using this grammar, the parser generator will detect that there is an ambiguity in the grammar. The problem manifests itself as a conflict between a shift and a reduce operation. The first rule says when looking at an *else* keyword the parser should *shift*. The second rule says when the parser is looking at an *else* it should *reduce*. To resolve this conflict there is generally a way to specify whether the generated parser should shift or reduce. The default action is usually to shift and that is what makes the most sense in this case. By shifting, the *else* would go with the nearest *if* statement. This is the normal behavior of parsers when encountering this if-then-else ambiguity.

2.12 Other Forms of Grammars

As a computer programmer you will likely learn at least one new language and probably a few during your career. New application areas frequently cause new languages to be developed to make programming applications in that area more convenient. Java, JavaScript, and ASP.NET are three languages that were created because of the world wide web. Ruby and Perl are languages that have become popular development languages for database and server side programming. Objective C is another language made popular by the rise of iOS App programming for Apple products. A recent trend in programming languages is to develop domain specific languages for particular embedded platforms.

Programming language manuals contain some kind of reference that describes the constructs of the language. Many of these reference manuals give the grammar of the language using a variation of a context free grammar. Examples include CBL (Cobol-like) grammars, syntax diagrams, and as we have already seen, BNF and EBNF. All these syntax metalanguages share the same features as grammars. They all have some way of defining parts of a program or syntactic categories and they all have a means of defining a language through recursively defined productions. The definitions, concepts, and examples provided in this chapter will help you understand a language reference when the time comes to learn a new language.

2.13 Limitations of Syntactic Definitions

The concrete syntax for a language is almost always an incomplete description. Not all syntactically valid strings of tokens should be regarded as valid programs. For instance, consider the expression $5 + 4/0$. Syntactically, this is a valid expression, but of course cannot be evaluated since division by zero is undefined. This is a semantic issue. The meaning of the expression is undefined because division by zero is undefined. This is a semantic issue and semantics are not described by a syntactic definition. All that a grammar can ensure is that the program is syntactically valid.

In fact, there is no BNF or EBNF grammar that generates only legal programs in any programming language including C++, Java, and Standard ML. A BNF grammar defines a context-free language: the left-hand side of each rules contains only one syntactic category. It is replaced by one of its alternative definitions regardless of the context in which it occurs.

The set of programs in any interesting language is not context-free. For instance, when the expression $a + b$ is evaluated, are a and b type compatible and defined over the $+$ operator? This is a context sensitive issue that can't be specified using a context-free grammar. Context-sensitive features may be formally described as a set of restrictions or context conditions. Context-sensitive issues deal mainly with declarations of identifiers and type compatibility. Sometimes, context-sensitive issues like this are said to be part of the *static semantics* of the language.

While a grammar describes how tokens are put together to form a valid program the grammar does not specify the semantics of the language nor does it describe the static semantics or context-sensitive characteritics of the language. Other means are necessary to describe these language characteristics. Some methods, like type inference rules, are formally defined. Most semantic characteristics are defined informally in some kind of English language description.

These are all context-sensitive issues.

- In an array declaration in C++, the array size must be a nonnegative value.
- Operands for the && operation must be boolean in Java.
- In a method definition, the return value must be compatible with the return type in the method declaration.

- When a method is called, the actual parameters must match the formal parameter types.

2.14 Chapter Summary

This chapter introduced you to programming language syntax and syntactic descriptions. Reading and understanding syntactic descriptions is worthwhile since you will undoubtedly come across new languages in your career as a computer scientist. There is certainly more that can be said about the topic of programming language syntax. Aho, Sethi, and Ullman [2] have written the widely recognized definitive book on compiler implementation which includes material on syntax definition and parser implementation. There are many other good compiler references as well. The Chomsky hierarchy of languages is also closely tied to grammars and regular expressions. Many books on Discrete Structures in Computer Science introduce this topic and a few good books explore the Chomsky hierarchy more deeply including an excellent text by Peter Linz [13].

In the next chapter you put this knowledge of syntax definition to good use learning a new language: the JCoCo assembly language. JCoCo is a virtual machine for interpreting Python bytecode instructions. Learning assembly language helps in having a better understanding of how higher level languages work and Chap. 3 provides many examples of Python programs and their corresponding JCoCo assembly language programs to show you how a higher level language is implemented.

2.15 Review Questions

1. What does the word syntax refer to? How does it differ from semantics?
2. What is a token?
3. What is a nonterminal?
4. What does BNF stand for? What is its purpose?
5. What kind of derivation does a top-down parser construct?
6. What is another name for a top-down parser?
7. What does the abstract syntax tree for $3 * (4 + 5)$ look like for infix expressions?
8. What is the prefix equivalent of the infix expression $3 * (4 + 5)$? What does the prefix expression's abstract syntax tree look like?
9. What is the difference between lex and yacc?
10. Why aren't all context-free grammars good for top-down parsing?
11. What kind of machine is needed to implement a bottom-up parser?
12. What is a context-sensitive issue in a language? Give an example in Java.
13. What do the terms *shift* and *reduce* apply to?

2.16 Exercises

1. Rewrite the BNF in Sect. 2.2.1 using EBNF.
2. Given the grammar in Sect. 2.3.1, derive the sentence $3*(4+5)$ using a right-most derivation.
3. Draw a parse tree for the sentence $3 * (4 + 5)$.
4. Describe how you might evaluate the abstract syntax tree of an expression to get a result? Write out your algorithm in English that describes how this might be done.
5. Write a regular expression to describe identifier tokens which must start with a letter and then can be followed by any number of letters, digits, or underscores.
6. Draw a finite state machine that would accept identifier tokens as specified in the previous exercise.
7. For the expression $3 * (4 + 5)$ show the sequence of shift and reduce operations using the grammar in Sect. 2.10.1. Be sure to say what is shifted and which rule is being used to reduce at each step. See the solution to practice problem 2.1 for the proper way to write the solution to this problem.
8. Construct a left-most derivation of $3 * (4 + 5)$ using the grammar in Sect. 2.9.3.

2.17 Solutions to Practice Problems

These are solutions to the practice problems. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

2.17.1 Solution to Practice Problem 2.1

This is a left-most derivation of the expression. There are other derivations that would be correct as well.

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow 4 + T \Rightarrow 4 + T * F \Rightarrow 4 + F * F \Rightarrow 4 + (E) * F$$
$$\Rightarrow 4 + (E - T) * F \Rightarrow 4 + (T - T) * F \Rightarrow 4 + (F - T) * F \Rightarrow 4 + (a - T) * F \Rightarrow$$
$$4 + (a - F) * F \Rightarrow 4 + (a - b) * F \Rightarrow 4 + (a - b) * x$$

2.17.2 Solution to Practice Problem 2.2

This is a right-most derivation of the expression $x * y + z$. There is only one correct right-most derivation.

$$E \Rightarrow E+T \Rightarrow E+F \Rightarrow E+z \Rightarrow T+z \Rightarrow T*F+z \Rightarrow T*y+z \Rightarrow F*y+z \Rightarrow x*y+z$$

2.17.3 Solution to Practice Problem 2.3

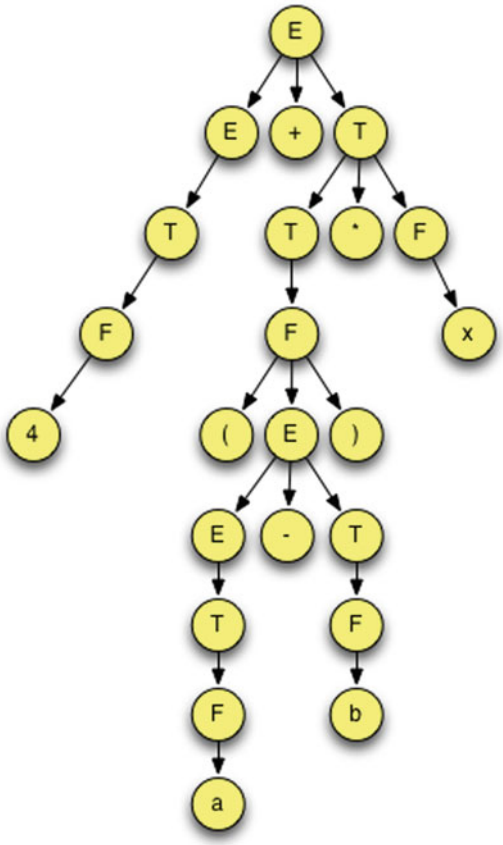
This is a left-most derivation of the expression $+4 * -abx$.

$$E \Rightarrow +EE \Rightarrow +4E \Rightarrow +4 * EE \Rightarrow +4 * -EEE \Rightarrow +4 * -aEE \Rightarrow +4 * -abE \Rightarrow +4 * -abx$$

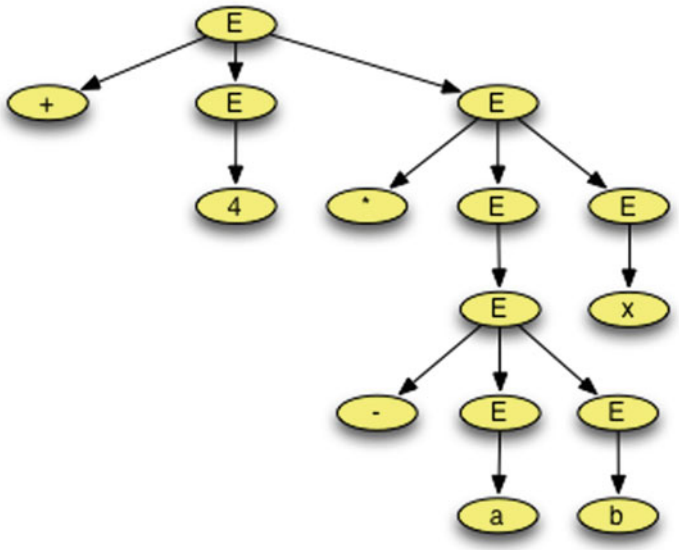
2.17.4 Solution to Practice Problem 2.4

Exactly like the parse tree for any other derivation of $(5 * x) + y$. There is only one parse tree for the expression given this grammar.

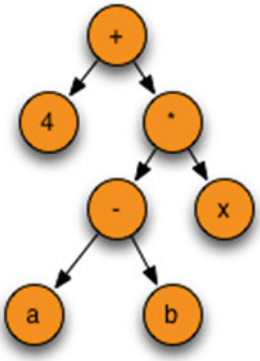
2.17.5 Solution to Practice Problem 2.5



2.17.6 Solution to Practice Problem 2.6



2.17.7 Solution to Practice Problem 2.7



2.17.8 Solution to Practice Problem 2.8

In order to define both negative and positive numbers, we can use the choice operator.

letter.letter* + digit.digit* + '-'.digit.digit* '+' + '-' + '*' + '/' + '(' + ')'

2.17.9 Solution to Practice Problem 2.9

$$E \Rightarrow T\ Rest E \Rightarrow F\ Rest T\ Rest E \Rightarrow 4\ Rest T\ Rest E \Rightarrow 4\ Rest E \Rightarrow$$
$$4 + T\ Rest E \Rightarrow 4 + F\ Rest T\ Rest E \Rightarrow 4 + (E)\ Rest T\ Rest E \Rightarrow 4 + (T\ Rest E) Rest T\ Rest E$$
$$\Rightarrow 4 + (F\ Rest T\ Rest E)\ Rest T\ Rest E \Rightarrow 4 + (a\ Rest T\ Rest E) Rest T\ Rest E \Rightarrow$$
$$4 + (a\ Rest E)\ Rest T\ Rest E \Rightarrow 4 + (a - T\ Rest E)\ Rest T\ Rest E \Rightarrow$$
$$4 + (a - F\ Rest E)\ Rest T\ Rest E \Rightarrow 4 + (a - b\ Rest E) \Rightarrow 4 + (a - b)\ Rest T\ Rest E$$
$$\Rightarrow 4 + (a - b) * F\ Rest T\ Rest E \Rightarrow 4 + (a - b) * x\ Rest T\ Rest E \Rightarrow 4 + (a - b) * x\ Rest E$$
$$\Rightarrow 4 + (a - b) * x$$

2.17.10 Solution to Practice Problem 2.10

In the parsing of $5 * 4 + 3$ the following shift and reduce operations: step A initial condition, step B shift, step C reduce by rule 5, step D reduce by rule 4, step E shift, step F shift, step G reduce by rule 5, step H reduce by rule 3, step I reduce by rule 2, step J shift, step K shift, step L reduce by rule 5, step M reduce by rule 4, step N reduce by rule 1, step O finished parsing with dot on right side and E on top of stack so pop and complete with success.

2.17.11 Solution to Practice Problem 2.11

To complete this problem it is best to do a right-most derivation of $(6 + 5) * 4$ first. Once that derivation is complete, you go through the derivation backwards. The difference in each step of the derivation tells you whether you shift or reduce. Here is the result.

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * 4 \Rightarrow F * 4 \Rightarrow (E) * 4 \Rightarrow (E + T) * 4 \Rightarrow (E + F) * 4$$
$$\Rightarrow (E + 5) * 4 \Rightarrow (T + 5) * 4 \Rightarrow (F + 5) * 4 \Rightarrow (6 + 5) * 4$$

We get the following operations from this. Stack contents have the top on the right up to the dot. Everything after the dot has not been read yet. We shift when we must move through the tokens to get to the next place we are reducing. Each step in the reverse derivation provides the reduce operations. Since there are seven tokens there should be seven shift operations.

1. Initially: . (6 + 5) * 4
2. Shift: (. 6 + 5) * 4
3. Shift: (6 . + 5) * 4
4. Reduce by rule 5: (F . + 5) * 4
5. Reduce by rule 4: (T . + 5) * 4
6. Reduce by rule 2: (E . + 5) * 4
7. Shift: (E + . 5) * 4

8. Shift: (E + 5 .) ∗ 4
9. Reduce by rule 5: (E + F .) ∗ 4
10. Reduce by rule 4: (E + T .) ∗ 4
11. Shift: (E + T) . ∗ 4
12. Reduce by rule 1: (E) . ∗ 4
13. Reduce by rule 6: F . ∗ 4
14. Reduce by rule 4: T . ∗ 4
15. Shift: T ∗ . 4
16. Shift: T ∗ 4 .
17. Reduce by rule 5: T ∗ F .
18. Reduce by rule 3: T .
19. Reduce by rule 2: E .