Chapter 1
SPECIFYING SYNTAX

Language provides a means of communication by sound and written symbols. Human beings learn language as a consequence of their life experiences, but in linguistics—the science of languages—the forms and meanings of languages are subjected to a more rigorous examination. This science can also be applied to the subject of this text, programming languages. In contrast to the natural languages, with which we communicate our thoughts and feelings, programming languages can be viewed as artificial languages defined by men and women initially for the purpose of communicating with computers but, as importantly, for communicating algorithms among people.

Many of the methods and much of the terminology of linguistics apply to programming languages. For example, language definitions consist of three components:

1. **Syntax** refers to the ways symbols may be combined to create well-formed sentences (or programs) in the language. Syntax defines the formal relations between the constituents of a language, thereby providing a structural description of the various expressions that make up legal strings in the language. Syntax deals solely with the form and structure of symbols in a language without any consideration given to their meaning.

2. **Semantics** reveals the meaning of syntactically valid strings in a language. For natural languages, this means correlating sentences and phrases with the objects, thoughts, and feelings of our experiences. For programming languages, semantics describes the behavior that a computer follows when executing a program in the language. We might disclose this behavior by describing the relationship between the input and output of a program or by a step-by-step explanation of how a program will execute on a real or an abstract machine.

3. **Pragmatics** alludes to those aspects of language that involve the users of the language, namely psychological and sociological phenomena such as utility, scope of application, and effects on the users. For programming languages, pragmatics includes issues such as ease of implementation, efficiency in application, and programming methodology.

Syntax must be specified prior to semantics since meaning can be given only to correctly formed expressions in a language. Similarly, semantics needs to be formulated before considering the issues of pragmatics, since interaction with human users can be considered only for expressions whose meaning is understood. In the current text, we are primarily concerned with syntax and semantics, leaving the subject of pragmatics to those who design and implement programming languages, chiefly compiler writers. Our paramount goal is to explain methods for furnishing a precise definition of the syntax and semantics of a programming language.

We begin by describing a metalanguage for syntax specification called BNF. We then use it to define the syntax of the main programming language employed in this text, a small imperative language called Wren. After a brief look at variants of BNF, the chapter concludes with a discussion of the abstract syntax of a programming language.

At the simplest level, languages are sets of sentences, each consisting of a finite sequence of symbols from some finite alphabet. Any really interesting language has an infinite number of sentences. This does not mean that it has an infinitely long sentence but that there is no maximum length for all the finite length sentences. The initial concern in describing languages is how to specify an infinite set with notation that is finite. We will see that a BNF grammar is a finite specification of a language that may be infinite.

1.1 GRAMMARS AND BNF

Formal methods have been more successful with describing the syntax of programming languages than with explaining their semantics. Defining the syntax of programming languages bears a close resemblance to formulating the grammar of a natural language, describing how symbols may be formed into the valid phrases of the language. The formal grammars that Noam Chomsky proposed for natural languages apply to programming languages as well.

Definition : A grammar $< \Sigma, N, P, S >$ consists of four parts:

1. A finite set Σ of **terminal symbols** , the **alphabet** of the language, that are assembled to make up the sentences in the language.

2. A finite set N of **nonterminal symbols** or **syntactic categories** , each of which represents some collection of subphrases of the sentences.

3. A finite set P of **productions** or **rules** that describe how each nonterminal is defined in terms of terminal symbols and nonterminals. The choice of

nonterminals determines the phrases of the language to which we ascribe meaning.

4. A distinguished nonterminal S, the **start symbol** , that specifies the principal category being defined—for example, sentence or program. ∎

In accordance with the traditional notation for programming language grammars, we represent nonterminals with the form "<category-name>" and productions as follows:

<declaration> ::= **var** <variable list> **:** <type> **;**

where "**var**", "**:**" , and "**;**" are terminal symbols in the language. The symbol "::=" is part of the language for describing grammars and can be read "is defined to be" or "may be composed of". When applied to programming languages, this notation is known as **Backus-Naur For m** or **BNF** for the researchers who first used it to describe Algol60. Note that BNF is a language for defining languages—that is, BNF is a **metalanguage** . By formalizing syntactic definitions, BNF greatly simplifies semantic specifications. Before considering BNF in more detail, we investigate various forms that grammars may take.

The **vocabulary** of a grammar includes its terminal and nonterminal symbols. An arbitrary production has the form $\alpha ::= \beta$ where α and β are strings of symbols from the vocabulary, and α has at least one nonterminal in it. Chomsky classified grammars according to the structure of their productions, suggesting four forms of particular usefulness, calling them type 0 through type 3.

Type 0: The most general grammars, the **unrestricted grammars** , require only that at least one nonterminal occur on the left side of a rule, "$\alpha ::= \beta$"—for example,

a <thing> b ::= b <another thing>.

Type 1: When we add the restriction that the right side contains no fewer symbols than the left side, we get the **context-sensitive grammars**—for example, a rule of the form

<thing> b ::= b <thing>.

Equivalently, context-sensitive grammars can be built using only productions of the form "α <B> $\gamma ::= \alpha\beta\gamma$", where <B> is a nonterminal, α, β, and γ are strings over the vocabulary, and β is not an empty string. These rules are called context-sensitive because the replacement of a nonterminal by its definition depends on the surrounding symbols.

Type 2: The **context-fr ee grammars** prescribe that the left side be a single nonterminal producing rules of the form "<A> ::= α", such as

<expression> ::= <expression> * <term>

where "*" is a terminal symbol. Type 2 grammars correspond to the BNF grammars and play a major role in defining programming languages, as will be described in this chapter.

Type 3: The most restrictive grammars, the **regular grammars**, allow only a terminal or a terminal followed by one nonterminal on the right side—that is, rules of the form "<A> ::= a" or "<A> ::= a <A>". A grammar describing binary numerals can be designed using the format of a regular grammar:

<binary numeral> ::= **0**

<binary numeral> ::= **1**

<binary numeral> ::= **0** <binary numeral>

<binary numeral> ::= **1** <binary numeral>.

The class of regular BNF grammars can be used to specify identifiers and numerals in most programming languages.

When a nonterminal has several alternative productions, the symbol "|" separates the right-hand sides of alternatives. The four type 3 productions given above are equivalent to the following consolidated production:

<binary numeral> ::= **0** | **1** | **0** <binary numeral> | **1** <binary numeral>.

Context-Free Grammars

As an example of a context-free grammar, consider the syntactic specification of a small fragment of English shown in Figure 1.1. The terminal symbols of the language are displayed in boldface. This grammar allows sentences such as "**the girl sang a song.**" and "**the cat surprised the boy with a song.**".

The grammar is context-free because only single nonterminals occur on the left sides of the rules. Note that the language defined by our grammar contains many nonsensical sentences, such as "**the telescope sang the cat by a boy.**". In other words, only syntax, and not semantics, is addressed by the grammar.

In addition to specifying the legal sentences of the language, a BNF definition establishes a structure for its phrases by virtue of the way a sentence can be derived. A derivation begins with the start symbol of the grammar, here the syntactic category <sentence>, replacing nonterminals by strings of symbols according to rules in the grammar.

<sentence> ::= <noun phrase> <verb phrase> **.**
<noun phrase> ::= <determiner> <noun>
| <determiner> <noun> <prepositional phrase>
<verb phrase> ::= <verb> | <verb> <noun phrase>
| <verb> <noun phrase> <prepositional phrase>
<prepositional phrase> ::= <preposition> <noun phrase>
<noun> ::= **boy** | **girl** | **cat** | **telescope** | **song** | **feather**
<determiner> ::= **a** | **the**
<verb> ::= **saw** | **touched** | **surprised** | **sang**
<preposition> ::= **by** | **with**

Figure 1.1: An English Grammar

An example of a derivation is given in Figure 1.2. It uniformly replaces the leftmost nonterminal in the string. Derivations can be constructed following other strategies, such as always replacing the rightmost nonterminal, but the outcome remains the same as long as the grammar is not ambiguous. We discuss ambiguity later. The symbol ⇒ denotes the relation encompassing one step of a derivation.

The structure embodied in a derivation can be displayed by a **derivation tree** or **parse tr ee** in which each leaf node is labeled with a terminal symbol

<sentence> ⇒ <noun phrase> <verb phrase> **.**
⇒ <determiner> <noun> <verb phrase> **.**
⇒ **the** <noun> <verb phrase> **.**
⇒ **the girl** <verb phrase> **.**
⇒ **the girl** <verb> <noun phrase> <prepositional phrase> **.**
⇒ **the girl touched** <noun phrase> <prepositional phrase> **.**
⇒ **the girl touched** <determiner> <noun> <prepositional phrase> **.**
⇒ **the girl touched the** <noun> <prepositional phrase> **.**
⇒ **the girl touched the cat** <prepositional phrase> **.**
⇒ **the girl touched the cat** <preposition> <noun phrase> **.**
⇒ **the girl touched the cat with** <noun phrase> **.**
⇒ **the girl touched the cat with** <determiner> <noun> **.**
⇒ **the girl touched the cat with a** <noun> **.**
⇒ **the girl touched the cat with a feather .**

Figure 1.2: A Derivation

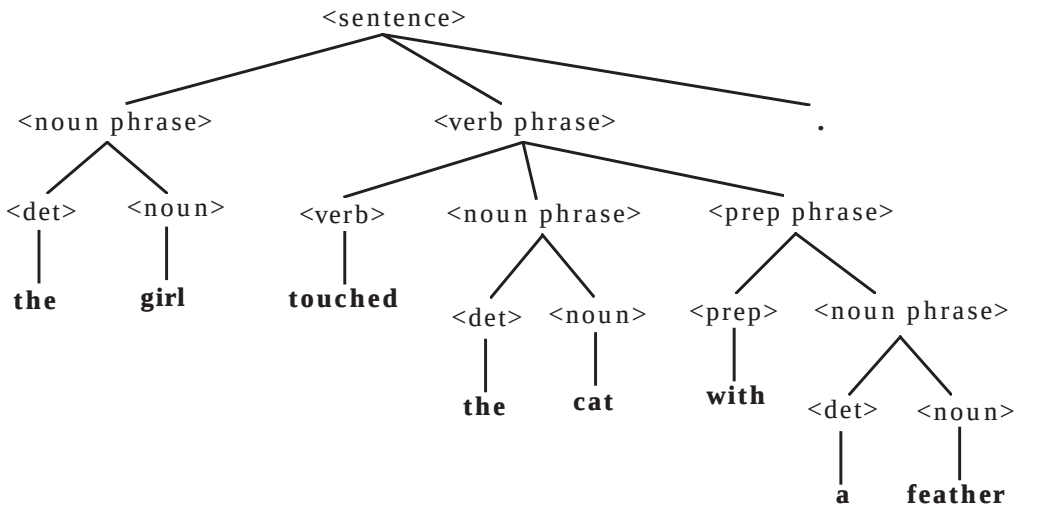


Figure 1.3: A Derivation Tree

and each interior node by a nonterminal whose children represent the right side of the production used for it in the derivation. A derivation tree for the sentence "**the girl touched the cat with a feather.**" is shown in Figure 1.3.

Definition : A grammar is **ambiguous** if some phrase in the language generated by the grammar has two distinct derivation trees. ∎

Since the syntax of a phrase determines the structure needed to define its meaning, ambiguity in grammars presents a problem in language specification. The English language fragment defined in Figure 1.1 allows ambiguity as witnessed by a second derivation tree for the sentence "**the girl touched the cat with a feather .**" drawn in Figure 1.4. The first parsing of the sentence implies that a feather was used to touch the cat, while in the second it was the cat in possession of a feather that was touched.

We accept ambiguity in English since the context of a discourse frequently clarifies any confusions. In addition, thought and meaning can survive in spite of a certain amount of misunderstanding. But computers require a greater precision of expression in order to carry out tasks correctly. Therefore ambiguity needs to be minimized in programming language definitions, although, as we see later, some ambiguity may be acceptable.

At first glance it may not appear that our fragment of English defines an infinite language. The fact that some nonterminals are defined in terms of themselves—that is, using recursion—admits the construction of unbounded strings of terminals. In the case of our English fragment, the recursion is indirect, involving noun phrases and prepositional phrases. It allows the con-

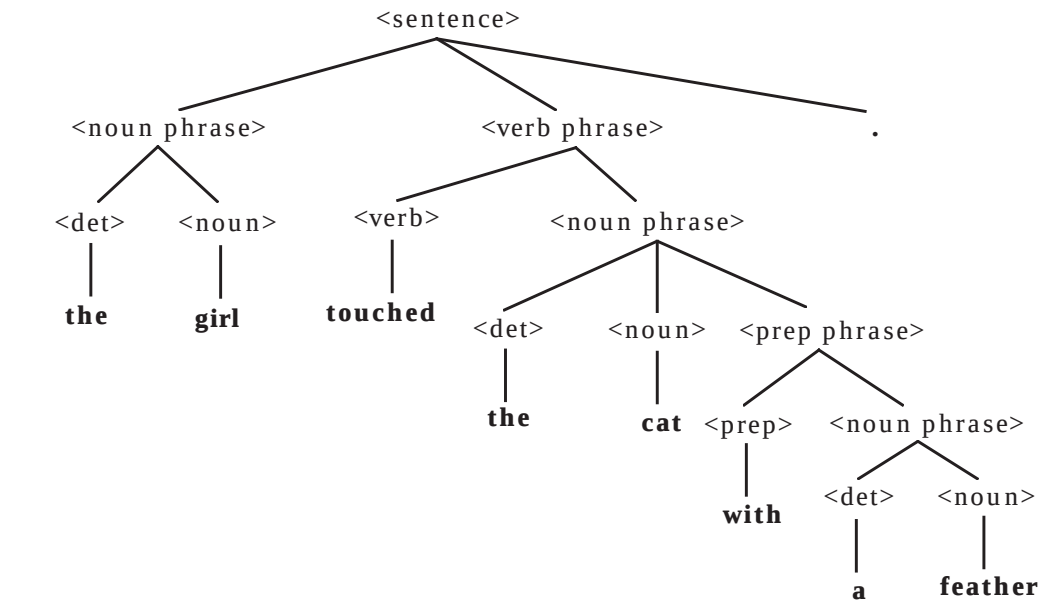


Figure 1.4: Another Derivation Tree

struction of sentences of the form "**the cat saw a boy with a girl with a boy with a girl with a boy with a girl.** " where there is no upper bound on the number of prepositional phrases.

To determine whether a nonterminal is defined recursively in a grammar, it suffices to build a directed graph that shows the dependencies among the nonterminals. If the graph contains a cycle, the nonterminals in the cycle are defined recursively. Figure 1.5 illustrates the **dependency graph** for the English grammar shown in Figure 1.1.

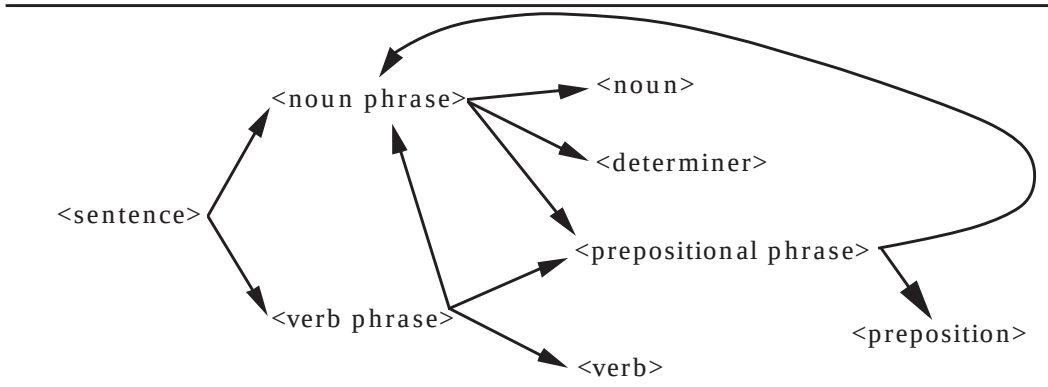


Figure 1.5: The Dependency Graph

Finally, observe again that a syntactic specification of a language entails no requirement that all the sentences it allows make sense. The semantics of the language will decide which sentences are meaningful and which are nonsense. Syntax only determines the correct form of sentences.

Context-Sensitive Grammars

To illustrate a context-sensitive grammar, we consider a synthetic language defined over the alphabet Σ = { **a**, **b**, **c** } using the productions portrayed in Figure 1.6.

<sentence> ::= **abc** | **a**<thing>**bc**
<thing>**b** ::= **b**<thing>
<thing>**c** ::= <other>**bcc**
a<other> ::= **aa** | **aa**<thing>
b<other> ::= <other>**b**

Figure 1.6: A Context-Sensitive Grammar

The language generated by this grammar consists of strings having equal numbers of **a**'s, **b**'s, and **c**'s in that order—namely, the set { **abc**, **aabbcc**, **aaabbbccc**, ... }. Notice that when replacing the nonterminal <thing>, the terminal symbol following the nonterminal determines which rule can be applied. This causes the grammar to be context-sensitive. In fact, a result in computation theory asserts that no context-free grammar produces this language. Figure 1.7 contains a derivation of a string in the language.

<sentence> => **a**<thing>**bc**
=> **ab**<thing>**c**
=> **ab**<other>**bcc**
=> **a**<other>**bbcc**
=> **aabbcc**

Figure 1.7: A Derivation

Exercises

1. Find two derivation trees for the sentence "**the girl saw a boy with a telescope.**" using the grammar in Figure 1.1 and show the derivations that correspond to the two trees.

2. Give two different derivations of the sentence "**the boy with a cat sang a song.**", but show that the derivations produce the same derivation tree.

3. Look up the following terms in a dictionary: *linguistics*, *semiotics*, *grammar*, *syntax*, *semantics*, and *pragmatics*.

4. Remove the syntactic category <prepositional phrase> and all related productions from the grammar in Figure 1.1. Show that the resulting grammar defines a finite language by counting all the sentences in it.

5. Using the grammar in Figure 1.6, derive the <sentence> **aaabbbccc** .

6. Consider the following two grammars, each of which generates strings of correctly balanced parentheses and brackets. Determine if either or both is ambiguous. The Greek letter ε represents an empty string.

   a) <string> ::= <string> <string> | **(** <string> **)** |**[** <string> **]** | ε

   b) <string> ::= **(** <string> **)** <string> | **[** <string> **]** <string> | ε

7. Describe the languages over the terminal set { **a**, **b** } defined by each of the following grammars:

   a) <string> ::= **a** <string> **b** | **ab**

   b) <string> ::= **a** <string> **a** | **b** <string> **b** | ε

   c) <string>::= **a** <B> | **b** <A>
   <A> ::= **a** | **a** <string> | **b** <A> <A>
   <B> ::= **b** | **b** <string> | **a** <B> <B>

8. Use the following grammar to construct a derivation tree for the sentence "**the girl that the cat that the boy touched saw sang a song.** ":

   <sentence> ::= <noun phrase> <verb phrase> **.**

   <noun phrase> ::= <determiner> <noun>
   | <determiner> <noun> <relative clause>

   <verb phrase> ::= <verb> | <verb> <noun phrase>

   <relative clause> ::= **that** <noun phrase> <verb phrase>

   <noun> ::= **boy** | **girl** | **cat** | **telescope** | **song** | **feather**

   <determiner> ::= **a** | **the**

   <verb> ::= **saw** | **touched** |**surprised** | **sang**

   Readers familiar with computation theory may show that the language generated by this grammar is context-free but not regular.

9. Identify which productions in the English grammar of Figure 1.1 can be reformulated as type 3 productions. It can be proved that productions of the form <A> ::= $\mathbf{a}_1\ \mathbf{a}_2\ \mathbf{a}_3 \ldots \mathbf{a}_n$ <B> are also allowable in regular grammars. Given this fact, prove the English grammar is regular—that is, it can be defined by a type 3 grammar. Reduce the size of the language by limiting the terminal vocabulary to **boy**, **a**, **saw**, and **by** and omit the period. This exercise requires showing that the concatenation of two regular grammars is regular.

1.2 THE PROGRAMMING LANGUAGE WREN

In this text, the formal methods for programming language specification will be illustrated with an example language Wren and several extensions to it. Wren is a small imperative language whose only control structures are the **if** command for selection and the **while** command for iteration. The name of the language comes from its smallness and its dependence on the **while** command (**w** in Wren). Variables are explicitly typed as **integer** or **boolean**, and the semantics of Wren follows a strong typing discipline when using expressions.

A BNF definition of Wren may be found in Figure 1.8. Observe that terminal symbols, such as reserved words, special symbols (**:=**, **+**, ...), and the letters and digits that form numbers and identifiers, are shown in boldface for emphasis.

Reserved words are keywords provided in a language definition to make it easier to read and understand. Making keywords reserved prohibits their use as identifiers and facilitates the analysis of the language. Many programming languages treat some keywords as predefined identifiers—for example, "write" in Pascal. We take all keywords to be reserved words to simplify the presentation of semantics. Since declaration sequences may be empty, one of the production rules for Wren produces a string with no symbols, denoted by the Greek letter ε.

The syntax of a programming language is commonly divided into two parts, the **lexical syntax** that describes the smallest units with significance, called **tokens**, and the **phrase-structur e syntax** that explains how tokens are arranged into programs. The lexical syntax recognizes identifiers, numerals, special symbols, and reserved words as if a syntactic category <token> had the definition:

<token> ::= <identifier> | <numeral> | <reserved word> | <relation>
| <weak op> | <strong op> | **:=** | **(** | **)** | **,** | **;** | **:**

where

```
<program> ::= program <identifier> is <block>
<block> ::= <declaration seq> begin <command seq> end
<declaration seq> ::= ε | <declaration> <declaration seq>
<declaration> ::= var <variable list> : <type> ;
<type> ::= integer | boolean
<variable list> ::= <variable> | <variable> , <variable list>
<command seq> ::= <command> | <command> ; <command seq>
<command> ::= <variable> := <expr> | skip
    | read <variable> | write <integer expr>
    | while <boolean expr> do <command seq> end while
    | if <boolean expr> then <command seq> end if
    | if <boolean expr> then <command seq> else <command seq> end if
<expr> ::= <integer expr> | <boolean expr>
<integer expr> ::= <term> | <integer expr> <weak op> <term>
<term> ::= <element> | <term> <strong op> <element>
<element> ::= <numeral> | <variable> | ( <integer expr> ) | – <element>
<boolean expr> ::= <boolean term> | <boolean expr> or <boolean term>
<boolean term> ::= <boolean element>
                 | <boolean term> and <boolean element>
<boolean element> ::= true | false | <variable> | <comparison>
                    | not ( <boolean expr> ) | ( <boolean expr> )
<comparison> ::= <integer expr> <relation> <integer expr>
<variable> ::= <identifier>
<relation> ::= <= | < | = | > | >= | <>
<weak op> ::= + | –
<strong op>  ::=  * | /
<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>
<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m
         | n | o | p | q | r | s | t | u | v | w | x | y | z
<numeral> ::= <digit> | <digit> <numeral>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Figure 1.8: BNF for Wren

```
<reserved word> ::= program | is | begin | end | var | integer
                  | boolean | read | write | skip | while | do | if
                  | then | else | and | or | true | false | not.
```

Such a division of syntax into lexical issues and the structure of programs in terms of tokens corresponds to the way programming languages are normally implemented. Programs as text are presented to a **lexical analyzer** or **scanner** that reads characters and produces a list of tokens taken from the **lexicon**, a collection of possible tokens of the language. Since semantics ascribes meaning to programs in terms of the structure of their phrases, the details of lexical syntax are irrelevant. The internal structure of tokens is immaterial, and only intelligible tokens take part in providing semantics to a program. In Figure 1.8, the productions defining <relation>, <weak op>, <strong op>, <identifier>, <letter>, <numeral>, and <digit> form the lexical syntax of Wren, although the first three rules may be used as abbreviations in the phrase-structure syntax of the language.

Ambiguity

The BNF definition for Wren is apparently free of ambiguity, but we still consider where ambiguity might enter into the syntactic definition of a programming language. Pascal allows the ambiguity of the "dangling else" by the definitions

<command> ::= **if** <boolean expr> **then** <command>
| **if** <boolean expr> **then** <command> **else** <command>.

The string "**if** $expr_1$ **then if** $expr_2$ **then** cmd_1 **else** cmd_2" has two structural definitions, as shown in Figure 1.9. The Pascal definition mandates the second form as correct by adding the informal rule that an **else** clause goes with the nearest **if** command. In Wren this ambiguity is avoided by bracketing the **then** or **else** clause syntactically with **end if**. These examples illustrate that derivation trees can be constructed with any nonterminal at their root. Such trees can appear as subtrees in a derivation from the start symbol <program>.

Another source of ambiguity in the syntax of expressions is explored in an exercise. Note that these ambiguities arise in recursive productions that allow a particular nonterminal to be replaced at two different locations in the definition, as in the production

<command> ::= **if** <boolean expr> **then** <command> **else** <command>.

This observation does not provide a method for avoiding ambiguity; it only describes a situation to consider for possible problems. In fact, there exists no general method for determining whether an arbitrary BNF specification is ambiguousornot.

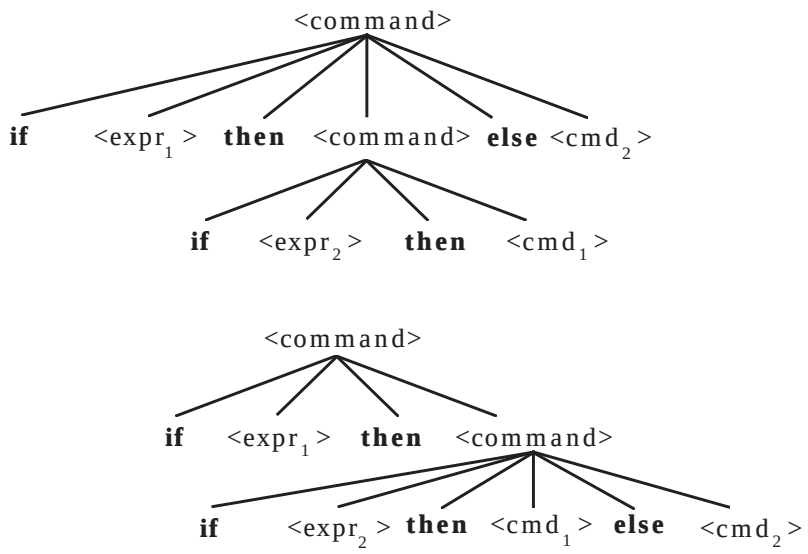


Figure 1.9: Two Structural Definitions

Context Constraints in Wren

Each program in Wren can be thought of as a string of tokens, although not every string of tokens is a legal Wren program. The BNF specification restricts the set of possible strings to those that can be obtained by a derivation from the nonterminal <program>. Even then, illegal programs remain. The BNF notation can define only those aspects of the syntax that are context-free, since each production can be applied regardless of the surrounding symbols. Therefore the program in Figure 1.10 passes the requirements prescribed by the BNF grammar for Wren.

```
program illegal is
    var a : boolean;
begin
    a := 5
end
```

Figure 1.10: An Illegal Wren Program

The error in the program “illegal” involves a violation of the context defined by a declaration. The variable “a” has been declared of Boolean type, but in the body of the program, an attempt is made to assign to it an integer value. The classification of such an error entails some controversy. Language

implementers, such as compiler writers, say that such an infraction belongs to the **static semantics** of a language since it involves the meaning of symbols and can be determined statically, which means solely derived from the text of the program. We argue that static errors belong to the syntax, not the semantics, of a language. Consider a program in which we declare a constant:

```
const c = 5;
```

In the context of this declaration, the following assignment commands are erroneous for essentially the same reason: It makes no sense to assign an expression value to a constant.

```
5 := 66;
```

```
c := 66;
```

The error in the first command can be determined based on the context-free grammar (BNF) of the language, but the second is normally recognized as part of checking the context constraints. Our view is that both errors involve the incorrect formation of symbols in a command—that is, the syntax of the language. The basis of these syntactic restrictions is to avoid commands that are meaningless given the usual model of the language.

Though it may be difficult to draw the line accurately between syntax and semantics, we hold that issues normally dealt with from the static text should be called syntax, and those that involve a program's behavior during execution be called semantics. Therefore we consider syntax to have two components: the **context-free syntax** defined by a BNF specification and the **context-sensitive syntax** consisting of context conditions or constraints that legal programs must obey. While the context-free syntax can be defined easily with a formal metalanguage BNF, at this point we specify the context conditions for Wren informally in Figure 1.11.

1. The program name identifier may not be declared elsewhere in the program.
2. All identifiers that appear in a block must be declared in that block.
3. No identifier may be declared more than once in a block.
4. The identifier on the left side of an assignment command must be declared as a variable, and the expression on the right must be of the same type.
5. An identifier occurring as an (integer) element must be an integer variable.
6. An identifier occurring as a Boolean element must be a Boolean variable.
7. An identifier occurring in a read command must be an integer variable.

Figure 1.11: Context Conditions for Wren

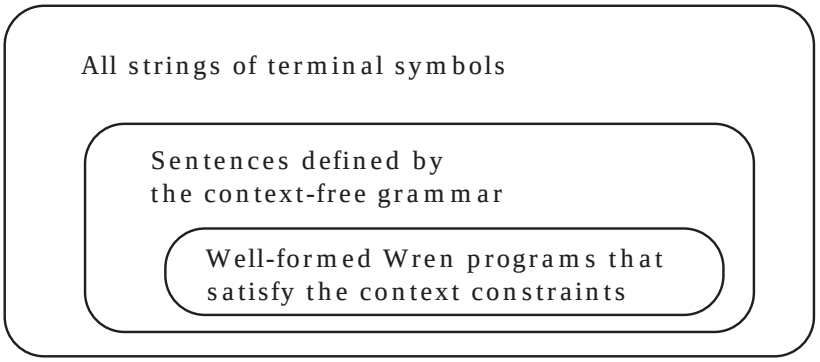
In theory the context conditions can be prescribed using a context-sensitive grammar, but these grammars are unsuitable for several reasons. For one, they bear no resemblance to the techniques that are used to check context conditions in implementing a programming language. A second problem is that the expansion of a node in the derivation tree may depend on sibling nodes (the context). Therefore we lose the direct hierarchical relationships between nonterminals that furnish a basis for semantic descriptions. Finally, formal context-sensitive grammars are difficult to construct and understand. Later in the text, more pragmatic formal methods for defining the context-sensitive aspects of programming languages will be investigated using attribute grammars, two-level grammars, and the methods of denotational semantics and algebraic semantics.

An eighth rule may be added to the list of context conditions for Wren:

8. No reserved word may be used as an identifier.

Since a scanner recognizes reserved words and distinguishes them from identifiers, attaching tags of some sort to the identifiers, this problem can be handled by the requirements of the BNF grammar. If a reserved word occurs in a position where an identifier is expected, the context-free derivation fails. Therefore we omit rule 8 from the list of context conditions.

The relationships between the languages specified in defining Wren are shown in the diagram below:



Semantic Errors in Wren

As any programmer knows, even after all syntax errors are removed from a program, it may still be defective. The fault may be that the program executes to completion but its behavior does not agree with the specification of the problem that the program is trying to solve. This notion of correctness will be dealt with in Chapter 11. A second possibility is that the program does not terminate normally because it has tried to carry out an operation

that cannot be executed by the run-time system. We call these faults **semantic** or **dynamic errors**. The semantic errors that can be committed while executing a Wren program are listed in Figure 1.12.

1. An attempt is made to divide by zero.
2. A variable that has not been initialized is accessed.
3. A **read** command is executed when the input file is empty.
4. An iteration command (**while**) does not terminate.

Figure 1.12: Semantic Errors in Wren

We include nontermination as a semantic error in Wren even though some programs, such as real-time programs, are intended to run forever. In presenting the semantics of Wren, we will expect every valid Wren program to halt.

Exercises

1. Draw a dependency graph for the nonterminal <expr> in the BNF definition of Wren.

2. Consider the following specification of expressions:

   <expr> ::= <element> | <expr> <weak op> <expr>
   <element> ::= <numeral> | <variable>
   <weak op> ::= **+** | **–**

   Demonstrate its ambiguity by displaying two derivation trees for the expression "**a–b–c**". Explain how the Wren specification avoids this problem.

3. This Wren program has a number of errors. Classify them as context-free, context-sensitive, or semantic.

```
program errors was
    var a,b : integer;
    var p,b ; boolean ;
begin
    a := 34;
    if b≠0 then p := true else p := (a+1);
    write p; write q
end
```

4. Modify the concrete syntax of Wren by adding an exponential operator ↑ whose precedence is higher than the other arithmetic operators (including unary minus) and whose associativity is right-to-left.

5. This BNF grammar defines expressions with three operations, *, -, and +, and the variables "a", "b", "c", and "d".

   <expr> ::= <thing> | <thing> ***** <expr>

   <object> ::= <element> | <element> **–** <object>

   <thing> ::= <object> | <thing> **+** <object>

   <element> ::= **a** | **b** | **c** | **d** | **(**<object>**)**

   a) Give the order of precedence among the three operations.

   b) Give the order (left-to-right or right-to-left) of execution for each operation.

   c) Explain how the parentheses defined for the nonterminal <element> may be used in these expressions. Describe their limitations.

6. Explain how the Wren productions for <identifier> and <numeral> can be written in the forms allowed for regular grammars (type 3)—namely, <A> ::= **a** or <A> ::= **a** <B>.

7. Explain the relation between left or right recursion in definition of expressions and terms, and the associativity of the binary operations (left-to-right or right-to-left).

8. Write a BNF specification of the syntax of the Roman numerals less than 100. Use this grammar to derive the string "XLVII".

9. Consider a language of expressions over lists of integers. List constants have the form: [3,-6,1], [86], [ ]. General list expressions may be formed using the binary infix operators

   +, –, *, and @ (for concatenation),

   where * has the highest precedence, + and - have the same next lower precedence, and @ has the lowest precedence. @ is to be right associative and the other operations are to be left associative. Parentheses may be used to override these rules.

   Example: [1,2,3] + [2,2,3] * [5,-1,0] @ [8,21] evaluates to [11,0,3,8,21].

   Write a BNF specification for this language of list expressions. Assume that <integer> has already been defined. The conformity of lists for the arithmetic operations is not handled by the BNF grammar since it is a context-sensitive issue.

10. Show that the following grammar for expressions is ambiguous and provide an alternative unambiguous grammar that defines the same set of expressions.

```
<expr> ::= <term> | <factor>
<term> ::= <factor> | <expr> + <term>
<factor> ::= <ident> | ( <expr> ) | <expr> * <factor>
<ident> ::= a | b | c
```

11. Consult [Naur63] to see how Algol solves the dangling else problem.

12. Explain how the syntactic ambiguity of the term "a(5)" in Ada is resolved. (*Note*: Ada uses parentheses to enclose array subscripts.)

1.3 VARIANTS OF BNF

Several notational variations of BNF are in common usage for describing context-free grammars. First we consider an alternate way of expressing regular grammars—namely, by **regular expr essions** . Each regular expression E denotes some language L(E) defined over an alphabet Σ. Figure 1.13 exhibits the language of regular expressions with ε representing the empty string, lowercase letters at the beginning of the alphabet portraying symbols in Σ, and uppercase letters standing for regular expressions.

Regular Expression	Language Denoted
∅	∅
ε	{ ε }
a	{ a }
(E • F)	{ uv | u∈L(E) and v∈L(F) } = L(E)•L(F)
(E | F)	{ u | u∈L(E) or u∈L(F) } = L(E) ∪ L(F)
(E*)	{ $u_1u_2...u_n$ | $u_1,u_2,...,u_n$∈L(E) and n≥0 }

Figure 1.13: Regular Expressions

The normal precedence for these regular operations is, from highest to lowest, "*" (Kleene closure or star), "•" (concatenation), and "|" (alternation), so that some pairs of parentheses may be omitted. Observe that a language over an alphabet Σ is a subset of Σ*, the set of all finite length strings of symbols from Σ.

The BNF definition of <digit> in Wren is already in the form of a regular expression. Numerals in Wren can be written as a regular expression using

<numeral> ::= <digit> • <digit>*.

The concatenation operator "•" is frequently omitted so that identifiers can be defined by

<identifier> ::= <letter> (<letter> | <digit>)*.

Several regular expressions have special abbreviations:

$E^+ = E \bullet E^*$ represents the concatenation of one or more strings from L(E).

E^n represents the concatenation of exactly $n \geq 0$ strings from L(E).

$E^? = \varepsilon \mid E$ represents zero or one string from L(E).

For example, in a language with signed numerals, their specification can be expressed as

<signed numeral> ::= $(+ \mid -)^?$ <digit>$^+$,

and the context-sensitive language defined in Figure 1.6 can be described as the set $\{ a^n b^n c^n \mid n \geq 1 \}$. Although this set is not regular, it can be described succinctly using this extension of the notation. The new operators "$^+$", "^{n}", and "?" have the same precedence as "*".

The major reason for using the language of regular expressions is to avoid an unnecessary use of recursion in BNF specifications. Braces are also employed to represent zero or more copies of some syntactic category, for example:

<declaration seq> ::= { <declaration> },

<command seq> ::= <command> { **;** <command> }, and

<integer expr> ::= <term> { <weak op> <term> }.

In general, braces are defined by { E } = E*. The braces used in this notation bear no relation to the braces that delimit a set. Since the sequencing of commands is an associative operation, these abbreviations for lists lose no information, but for integer expressions we no longer know the precedence for weak operators, left-to-right or right-to-left. Generally, we use only abbreviations in specifying syntax when the lost information is immaterial. The example of command sequences illustrates a place where ambiguity may be allowed in a grammar definition without impairing the accuracy of the definition, at least for program semantics. After all, a command sequence can be thought of simply as a list of commands. A derivation tree for a command sequence can be represented using a nested tree structure or the multibranch tree illustrated in Figure 1.14.

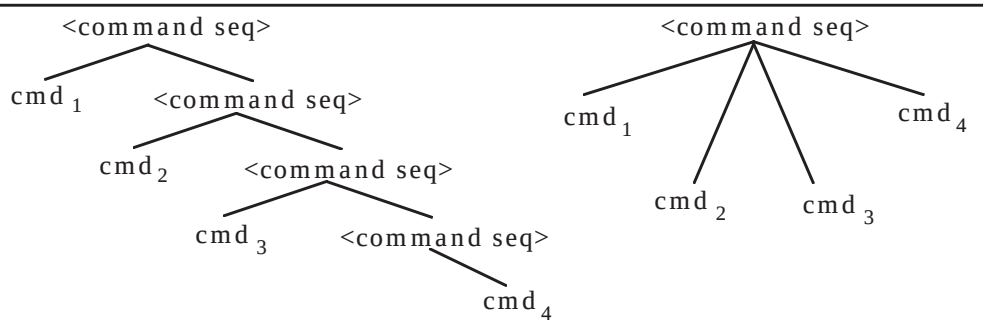


Figure 1.14: Derivation Trees for Command Sequences

Exercises

1. Use braces to replace recursion in specifying variable lists and terms in Wren.

2. Specify the syntax of the Roman numerals less that 100 using regular expressions.

3. Write a BNF grammar that specifies the language of regular expressions in Figure 1.13 over the alphabet $\Sigma = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$. The grammar should enforce the precedence rules for the operators.

4. Investigate possible algebraic laws for the binary operators in regular expressions. Consider associative, commutative, and distributive laws for the operators "•" and "|". Prove properties that hold and give counterexamples for those that do not. Do these binary operations have identities?

5. Prove these special laws for "*":
   a) $E^* = \varepsilon \mid (E \bullet E^*)$
   b) $E^* = \varepsilon \mid (E^* \bullet E)$
   Hint: Show that the languages, sets of strings, denoted by the expressions are equal.

6. Use regular expressions to define the following token classes:
   a) Integer numerals (positive or negative) with no leading zeros.
   b) Fixed point decimal numerals that must have at least one digit before and after the decimal point.
   c) Identifiers that allow lowercase letters and underscores but with the properties that no underscore occurs at the beginning or the end of the identifier and that it contains no two consecutive underscores.

1.4 ABSTRACT SYNTAX

The BNF definition of a programming language is sometimes referred to as the **concrete syntax** of the language since it tells how to recognize the physical text of a program. Software utilities take a program as a file of characters, recognize that it satisfies the context-free syntax for the language, and produce a derivation tree exhibiting its structure. This software usually decomposes into two parts: a **scanner** or **lexical analyzer** that reads the text and creates a list of tokens and a **parser** or **syntactic analyzer** that forms a derivation tree from the token list based on the BNF definition. Figure 1.15 illustrates this process.

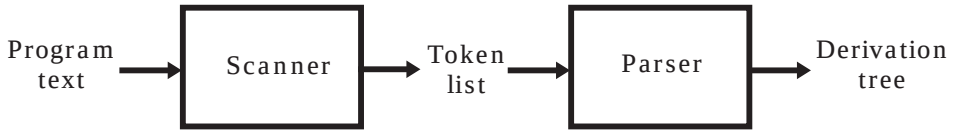


Figure 1.15: The Scanner and Parser

We can think of this process as two functions:

$$\text{scan} : \text{Character}^* \rightarrow \text{Token}^*$$

$$\text{parse} : \text{Token}^* \rightarrow \text{Derivation Tree}$$

whose composition "parse ∘ scan" creates a derivation tree from a list of characters forming the physical program.

The success of this process "parse ∘ scan" depends on the accuracy and detail found in the syntactic specification, the BNF, of the programming language. In particular, ambiguity in a language specification may make it impossible to define this function.

Abstract Syntax Trees

Those qualities of a BNF definition that make parsing possible also create a resulting derivation tree containing far more information than necessary for a semantic specification. For example, the categories of terms and elements are required for accurate parsing, but when ascribing meaning to an expression, only its basic structure is needed. Consider the trees in Figures 1.16 and 1.17.

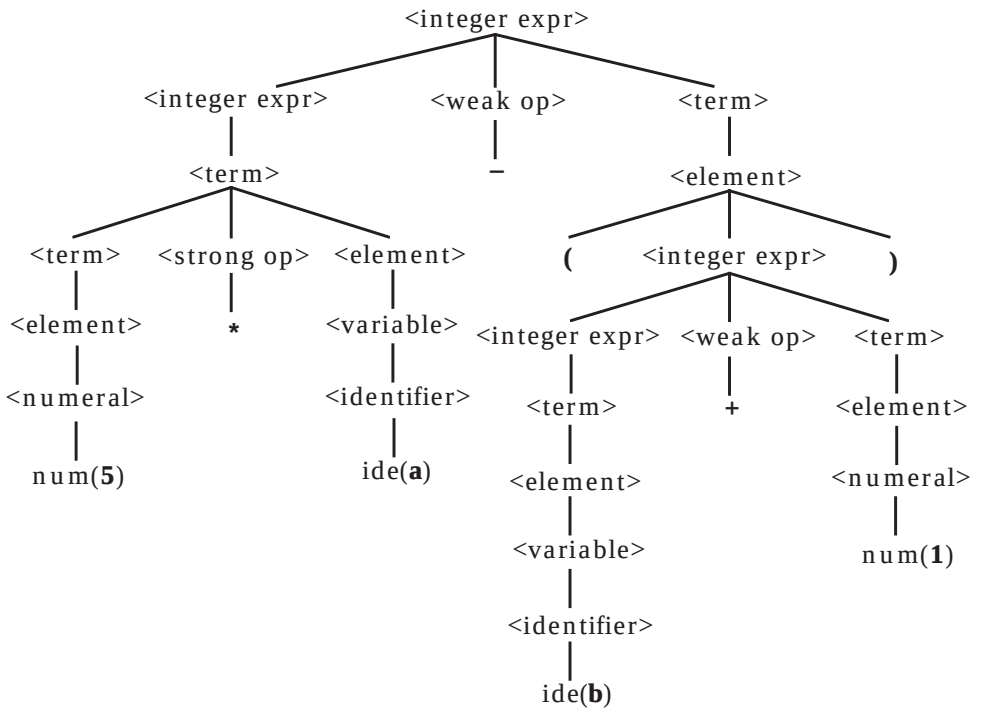


Figure 1.16: A Derivation Tree for **5*a– (b+1)**

The derivation tree retains all the information used in parsing including details that only the parser needs. On the other hand, an **abstract syntax tree** captures the syntactic structure of the expression completely in a much simpler form. After all, the crucial property of the expression "5*a – (b+1)" is that it is a difference of a product and a sum of certain numbers and variables. Any other information is redundant. Figure 1.17 shows two possible abstract syntax trees for the expression. In all three trees, we assume that the text has already been tokenized (scanned).

In transforming a derivation tree into an abstract syntax tree, we generally pull the terminal symbols representing operations and commands up to the root nodes of subtrees, leaving the operands as their children. The second tree in Figure 1.17 varies slightly from this principle in the interest of regularity in expressions. Using this approach, this expression can be thought of as a binary operation and two subexpressions. The choice of the left subtree for the binary operation is arbitrary; it seems to suggest a prefix notation for binary operations, but we are not talking about concrete syntax here, only an abstract representation of certain language constructs. We may choose any representation that we want as long as we can describe the constructs of the language adequately and maintain consistency.

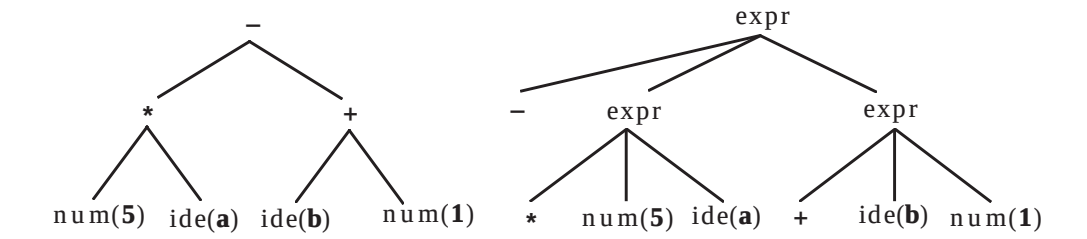


Figure 1.17: Abstract Syntax Trees for **5*a– (b+1)**

The literature of computer science contains considerable confusion between derivation trees and abstract syntax trees; the term parse tree has been used to refer to both kinds of trees. We explain the issue by viewing these trees as abstractions of phrases in a programming language. Derivation trees abstract derivations in the sense that one derivation tree can correspond to several different derivations—for example, leftmost or rightmost. Furthermore, abstract syntax trees abstract derivation trees, since several strings in the language may correspond to the same abstract syntax tree but yet have different derivation trees; for example, "(a+5)–x/2" and "a+5–(x/2)" have the same abstract syntax tree, although their derivation trees are different.

Abstract Syntax of a Programming Language

The point of abstract syntax is simply to communicate the structure of phrases in terms of their semantics in a programming language as trees. Semantics can be defined in terms of derivation trees and actually is with attribute grammars, but most semantic methods are far more understandable when based on a cleaner representation of the phrases in a language. As can be seen from Figure 1.17, designing patterns for abstract syntax allows freedom in format, but for a particular programming language specification, we want uniform templates for the various parts of a language. The blueprints for the abstract syntax trees of a programming language are specified by a collection of syntactic categories or domains and a set of rules telling how categories are decomposed into other categories or tokens.

To design the abstract syntax for a programming language, we need to determine which notions (nonterminals) are fundamental to the language and which basic forms the constructs of the language may take. As an example, consider the expressions in Wren—that is, those language phrases derivable from the nonterminal <expr>. By observing the concrete syntax for Wren (Figure 1.8), we see that expressions ultimately consist of operations (**+**, **–**, **and**, **not**, and so on) applied to numerals, identifiers, and Boolean constants (**true** and **false**). Therefore we reduce the nonterminals used to define expressions into three abstract syntactic categories: Expression, Numeral, and

Identifier. We fold the categories of terms, elements, and comparisons into Expression since they are simply special cases of expressions.

To find the abstract productions that specify the basic patterns of expressions, we first repeat those BNF rules that define expressions in Wren, but with the nonterminals <weak op>, <strong op>, <relation>, and <variable> factored out:

<expr> ::= <integer expr> | <boolean expr>
<integer expr> ::= <term>
| <integer expr> **+** <term> | <integer expr> **–** <term>
<term> ::= <element> | <term> ***** <element> | <term> **/** <element>
<element> ::= <numeral> | <identifier> | **(** <integer expr> **)** | **–** <element>
<boolean expr> ::= <boolean term> | <boolean expr> **or** <boolean term>
<boolean term> ::= <boolean element>
| <boolean term> **and** <boolean element>
<boolean element> ::= **true** | **false** | <identifier> | <comparison>
| **not (** <boolean expr> **)** | **(** <boolean expr> **)**
<comparison> ::= <integer expr> **<=** <integer expr>
| <integer expr> **<** <integer expr>
| <integer expr> **=** <integer expr>
| <integer expr> **>=** <integer expr>
| <integer expr> **>** <integer expr>
| <integer expr> **<>** <integer expr>

Observe that in a derivation

<expr> ⇒ <integer expr> ⇒ <term> ⇒ <element> ⇒ <numeral>,

the only essential information relative to Wren is that an expression can be a numeral. Outside of the parsing problem, the intervening nonterminals play no essential role in describing Wren. Therefore unit rules such as <integer expr> ::= <term>, can be ignored unless they involve basic components of expressions, such as numerals, identifiers, or essential nonterminals. So we select only those rules from the BNF that describe the structure of possible expressions. Omitting parenthesized expressions, the following list results:

<integer expr> **+** <term>
<integer expr> **–** <term>
<term> ***** <element>
<term> **/** <element>
<numeral>

<identifier>
– <element>
<boolean expr> **or** <boolean term>
<boolean term> **and** <boolean element>
true
false
not (<boolean expr> **)**
<integer expr> **<=** <integer expr>
<integer expr> **<** <integer expr>
<integer expr> **=** <integer expr>
<integer expr> **>=** <integer expr>
<integer expr> **>** <integer expr>
<integer expr> **<>** <integer expr>

After the redundant nonterminals are merged into Expression, these basic templates can be summarized by the following abstract production rules:

Expression ::= Numeral | Identifier | **true** | **false**
| Expression Operator Expression | **–** Expression
| **not(** Expression **)**

Operator ::= **+** | **–** | ***** | **/** | **or** | **and** | **<=** | **<** | **=** | **>** | **>=** | **<>**

An abstract syntax for Wren is given in Figure 1.18. This abstract syntax delineates the possible abstract syntax trees that may be produced by programs in the language. To avoid confusion with concrete syntax, we utilize a slightly different notation for abstract production rules, using identifiers starting with uppercase letters for syntactic categories.

Notice that a definition of abstract syntax tolerates more ambiguity since the concrete syntax has already determined the correct interpretation of the symbols in the program text. We investigate a formal description of abstract syntax in Chapter 12, using the terminology of algebraic semantics.

We suggested earlier that parsing a program results in the construction of a derivation tree for the program. As a consequence of adhering to the BNF syntax of a language, any parsing algorithm must at least implicitly create a derivation tree. But in fact we usually design a parser to generate an abstract syntax tree instead of a derivation tree. Therefore the syntax of “parse” is given by

$$\text{parse} : \text{Token}^* \rightarrow \text{Abstract Syntax Tree}.$$

Abstract Syntactic Categories

Program	Type	Operator
Block	Command	Numeral
Declaration	Expression	Identifier

Abstract Production Rules

Program ::= **program** Identifier **is** Block

Block ::= Declaration* **begin** Command^{+} **end**

Declaration ::= **var** Identifier^{+} **:** Type **;**

Type ::= **integer** | **boolean**

Command ::= Identifier **:=** Expression | **skip** | **read** Identifier
| **write** Expression | **while** Expression **do** Command^{+}
| **if** Expression **then** Command^{+}
| **if** Expression **then** Command^{+} **else** Command^{+}

Expression ::= Numeral | Identifier | **true** | **false**
| Expression Operator Expression | **–** Expression
| **not(** Expression**)**

Operator ::= **+** | **–** | ***** | **/** | **or** | **and** | **<=** | **<** | **=** | **>** | **>=** | **<>**

Figure 1.18: Abstract Syntax for Wren

Generally, this parse function will not be one to one. The token lists for the expressions “a+b-c” and “(a+b-c)” map to the same abstract syntax tree. The main point of abstract syntax is to omit the details of physical representation, leaving only the forms of the abstract trees that may be produced. For example, the abstract syntax has no need for parentheses since they are just used to disambiguate expressions. Once this assessment has been done by the parser, the resulting abstract trees have unambiguous meaning, since the branching of trees accurately conveys the hierarchical structure of a phrase. Whereas the concrete syntax defines the way programs in a language are actually written, the abstract syntax captures the pure structure of phrases in the language by specifying the logical relations (relative to the intended semantics) between parts of the language. We can think of an abstract syntax tree as embodying the derivation history of a phrase in the language without mentioning all of the terminal and nonterminal symbols.

When we implement a parser using Prolog in Chapter 2, the parsing operation applied to the token string for the expression “5*a – (b+1)” will create a Prolog structure:

```
expr(minus,expr(times,num(5),ide(a)),expr(plus,ide(b),num(1))),
```

which is a linear representation of one of the abstract syntax trees in Figure 1.17. See Appendix A for a definition of Prolog structures.

In the abstract production rules, lists of declarations, commands, and identifiers are described by means of the closure operators "*" and "$^{+}$". An alternative approach used in many formal methods of specifying semantics involves direct recursion as in:

command = command **;** command | identifier **:=** expression | **skip** | ....

The closure operator "$^{+}$" on commands ignores the occurrence of semicolons between commands, but in abstract syntax semicolons are only cosmetic. Although the abstract production rules in Figure 1.18 use reserved words, these act only as mnemonic devices to help us recognize the phrases being formulated. In fact, not all the reserved words are used in the productions, only enough to suggest the structure of the programming constructs. Note that we have deleted **end if** and **end while** for the sake of conciseness.

An alternative way of describing the abstract production rules is displayed in Figure 1.19 where the definitions are given as tagged record structures. Actually, the notation used to specify the abstract productions is not crucial. The important property of abstract syntax is embodied in the relationships between the categories; for example, a **while** command consists of an expression and a list of commands. As mathematical objects, the various categories are built from aggregations (Cartesian products), alternations (disjoint unions), and list structures. Any notations for these three constructors can serve to define the abstract production rules. We explore these mathematical structures more carefully in Chapter 10.

As an example, consider abstract pattern of the command

while n>0 **do write** n; n:=n-1 **end while** .

Figure 1.20 shows an abstract syntax tree for this command based on the abstract syntax defined in Figure 1.18. Since the body of a while command is a command sequence, we need an extra level in the tree to represent the list of commands. In contrast, following the abstract syntax specification in Figure 1.19 produces a structure representing a similar abstract syntax tree:

while(*expr*(>,ide(n),num(0)),
[*write*(ide(n)),*assign*(ide(n), *expr*(-,ide(n),num(1)))]).

The list of commands (a command sequence) in the body of the while command is represented as a list using brackets in the structure. This notation agrees with that used by Prolog lists in the next chapter—namely, [a, b, c]. The abstract syntax tree of a complete Wren program as a Prolog structure can be found at the beginning of Chapter 2. Notice the lists of variables, declarations, and commands in the representation of that tree.

Abstract Production Rules

Program ::= *prog*(Identifier, Block)

Block ::= *block*(Declaration*, Command$^+$)

Declaration ::= *dec*(Type, Identifier$^+$)

Type ::= *integer* | *boolean*

Command ::= *assign*(Identifier, Expression) | *skip*
| *read*(Identifier) | *write*(Expression)
| *while*(Expression, Command$^+$) | *if*(Expression, Command$^+$)
| *ifelse*(Expression, Command$^+$, Command$^+$)

Expression ::= Numeral | Identifier | *true* | *false* | *not*(Expression)
| *expr*(Operator, Expression, Expression) | *minus*(Expression)

Operator ::= + | – | * | / | *or* | *and* | <= | < | = | > | >= | <>

Figure 1.19: Alternative Abstract Syntax for Wren

Although concrete syntax is essential to implementing programming languages, it is the abstract syntax that lies at the heart of semantic definitions. The concrete syntax is incidental to language specification, but it is important to users since it influences the way they think about a language. This aspect of pragmatics is not of direct concern to us in studying the semantics of programming languages.

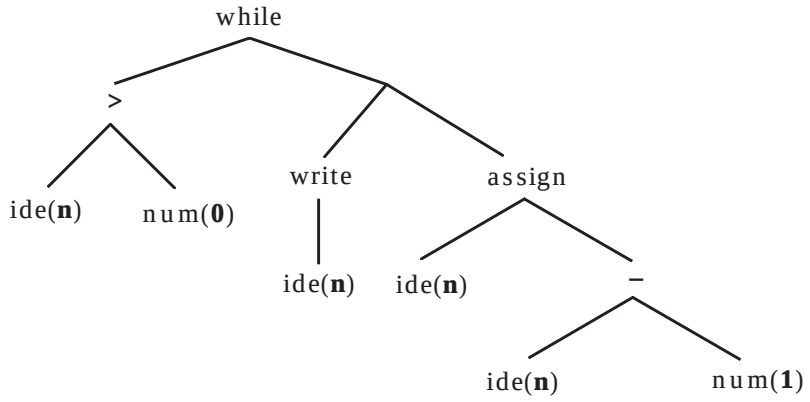


Figure 1.20: Abstract Syntax Tree

It can be argued that when designing a new programming language, we need to formulate the abstract syntax along with the semantics so that the meaning of a program emanates from a mapping

meaning : Abstract Syntax Trees → Semantic Objects

where the semantic objects provide meaning for the various language constructs. Different approaches to semantics depend on the disparate kinds of objects used to define meaning. Later we will see that this discussion is skewed to the denotational approach to semantics, but viewing meaning as a function from syntactic phrases to some sort of semantic objects can be a useful way of organizing formal semantics.

Given the abstract syntax of a programming language, the concrete syntax can be defined by an operation

$$unparse : \text{Abstract Syntax Trees} \rightarrow \text{Concrete Syntax}$$

where Concrete Syntax refers to derivation trees, to lists of tokens, or to lists of characters representing program texts. Since two different phrases following the concrete syntax may produce the same abstract syntax tree, *unparse* may not be a function at all. To ensure that *unparse* is a well-defined function, some canonical representation of concrete phrases must be specified—for example, taking expressions having the fewest parentheses. The correctness of a parsing algorithm can be demonstrated by showing that it is the inverse, in some sense, of the *unparse* function.

Exercises

1. Construct a derivation tree and an abstract syntax tree for the Wren command

   "**if n>0 then a := 3 else skip end if** ".

   Also write the abstract tree as a Prolog structure.

2. Parse the following token list to produce an abstract syntax tree:

   [while, not, lparen, ide(done), rparen, do, ide(n), assign,
   ide(n), minus, num(1), semicolon, ide(done), assign,
   ide(n), greater, num(0), end, while]

3. Draw an abstract syntax tree for the following Wren program:

```
program binary is
    var n,p : integer;
begin
    read n; p := 2;
    while p<=n do p := 2*p end while;
    p := p/2;
    while p>0 do
        if n>= p  then write 1; n := n–p else write 0 end if;
        p := p/2
    end while
end
```

4. Using the concrete syntax of Wren, draw abstract syntax trees or record-like structures following the definition in Figure 1.19 for these language constructs:

   a) (a+7)*(n/2)

   b) **while** n>=0 **do** s:=s+(n*n); n:= n–1 **end while**

   c) **if** a **and** b **or** c **then read** m; **write** m **else** a:=**not(**b **and** c**) end if**

1.5 FURTHER READING

The concepts and terminology for describing the syntax of languages derives from Noam Chomsky's seminal work in the 1950s—for example, [Chomsky56] and [Chomsky59]. His classification of grammars and the related theory has been adopted for the study of programming languages. Most of this material falls into the area of the theory of computation. For additional material, see [Hopcroft79] and [Martin91]. These books and many others contain results on the expressiveness and limitations of the classes of grammars and on derivations, derivation trees, and syntactic ambiguity.

John Backus and Peter Naur defined BNF in conjunction with the group that developed Algol60. The report [Naur63] describing Algol syntax using BNF is still one of the clearest presentations of a programming language, although the semantics is given informally.

Most books on compiler writing contain extensive discussions of syntax specification, derivation trees, and parsing. These books sometimes confuse the notions of concrete and abstract syntax, but they usually contain extensive examples of lexical and syntactic analysis. We recommend [Aho86] and [Parsons92]. Compiler writers typically disagree with our distinction between syntax and semantics, putting context constraints with semantics under the name static semantics. Our view that static semantics is an oxymoron is supported by [Meek90].

Abstract syntax was first described by John McCarthy in the context of Lisp [McCarthy65a]. More material on abstract syntax and other issues pertaining to the syntax of programming languages can be found in various textbooks on formal syntax and semantics, including [Watt91] and [Meyer90]. The book by Roland Backhouse concentrates exclusively on the syntax of programming languages [Backhouse79].