4 SYNTAX AND ELEGANCE: ALGOL-60

4.1 DESIGN: SYNTACTIC STRUCTURES

Machine Independence Led to a Free Format

In Chapter 3, Section 3.1, we saw that Algol resulted from an attempt to solve the problem of *software portability*, that is, the problem of making programs work on a wide variety of computers. In other words, Algol had to be *machine independent.* Recall:

> **Portability Principle**
>
> Avoid features or facilities that are dependent on a particular computer or a small class of computers.

This goal affected all aspects of Algol's design, including its syntactic structures. For instance, many different input devices were then in use on computers; some machines used 80- or 90-column punched cards, others used teletypewriter-like devices and punched paper tape, and so forth. Hence, the goal of machine independence ruled out a *fixed format* such as that used in FORTRAN; instead, Algol adopted a *free format*, that is, a format independent of columns or other details of the layout of the program. This convention is familiar from natural languages, such as English, in which the meaning of sentences does not depend on the columns or the number of lines on which they are written. Almost all languages designed after Algol have used a free format.

Since Algol did not force a particular layout on programmers, for the first time programmers had to think about the problem of how a program *should* be formatted. For example, a program could be written in a FORTRAN-like style:

```
sum := 0;
for i := 1 step 1 until N do
begin
real val;
```

```
Read Real (val);
Data[i] := if val < 0 then -val else val
end;
for i := 1 step 1 until N do
sum := sum + Data[i];
avg := sum/N;
```

Or it could be written, English style, in a continuous stream:

```
sum := 0; for i := 1 step 1 until N do begin real val;
Read real (val); Data[i]:= if val < 0 then -val else val
end; for i := 1 step 1 until N do sum := sum + Data[i];
avg := sum/N;
```

Ultimately, this question led to the idea of formatting a program in a way that reflects the structure of the program. That is, the layout of the program should obey the *Structure Principle* by making the program's textual arrangement correspond to its dynamic behavior. For example, a better layout of the above program fragment is

```
sum := 0;
for i := 1 step 1 until N do
  begin real val;
    Read Real (val);
    Data[i] := if val < 0 then -val else val
  end;
for i := 1 step 1 until N do
  sum := sum + Data[i];
avg := sum/N;
```

This format is better because it helps the eye to identify portions of the program that are within the same control and name contexts. The hierarchical structure of the program is made manifest in its layout.

Three Levels of Representation Are Used

The attempt to have a machine-independent language in the face of differing computers and varying national conventions led to a unique specification of Algol's lexical structure. The differing input-output devices available on different computers meant that there was no universal character set that could be used by Algol. Certainly, all computers provided for uppercase letters, digits, and a few punctuation symbols (such as '.', ',', '+'). Beyond this, however, there was little uniformity; some provided lowercase letters, some provided special logical and mathematical symbols ($<$, $\wedge$, $\vee$, etc.), others had various commercial symbols (@, #, etc.), some had format controls (e.g., subscripts and superscripts), and some even had the ability to use two colors (e.g., red and black). Two solutions are apparent: We can either design our language to use just those symbols that are available in all character sets, which will severely restrict the lexics of the language, or we can make the design of our language independent of particular character sets.

The Algol committee took the latter approach. They were forced into this position by the inability to solve a seemingly simple problem. It is conventional in the United States to use a period to represent a decimal point, while in Europe it is conventional to use a comma. When the Algol committee came to the issue of how to write numbers, both sides refused to budge. It seemed that the prospects for an international programming language would be defeated by a decimal point! (It is a folk theorem of programming language design that the more trivial an issue is, the more vehemently people will fight over it!) The solution adopted by the committee was to define three levels of language: a *reference language*, a *publication language*, and multiple *hardware representations*. These languages differed in lexical conventions but had the same syntax.

The *reference language* is the language used in all examples in the Algol Report, as well as in this book. Here is an example of the reference language:

```
a[i + 1] := (a[i] + pi × r↑2)/6.02₁₀23;
```

The *publication language* was intended to be the language used for publishing algorithms in Algol. It was distinguished by allowing various lexical and printing conventions (e.g., subscripts and Greek letters) that would aid readability. For instance, the published form of the above statement could be

$$a_{i+1} \leftarrow \{a_i + \pi \times r^2\}/6.02 \times 10^{23};$$

In practice, the publication language has usually been the same as the reference language.

Finally, the committee left it up to Algol implementors to define *hardware representations* of Algol that would be appropriate to the character sets and input-output devices of their computer systems. For instance, the above example may have to be written in any of these ways:

```
a(/i + 1/) := ( a(/i/) + pi*r**2 ) / 6,02e23;
A(.I + 1.) .= ( A(.I.) + PI * R 'POWER' 2 ) / 6.02'23.,
a[i + 1] := (a[i] + pi*r^2)/6.02E23;
```

The various representations were required to permit any program to be translated into any of the different forms.

Algol Solved Some Problems of FORTRAN Lexics

Algol's distinction among reference, publication, and hardware languages led to the idea that the keywords of the language are indivisible basic symbols. That is, boldface (or underlined) words such as '**if**' and '**procedure**' are considered single symbols, like '≤' and ':=', and have no relation to similarly appearing identifiers such as '`if`' or '`procedure`'. This means that the confusion between keywords and identifiers that we saw in FORTRAN is not possible in Algol. For example, it is perfectly unambiguous (although somewhat confusing) to write:

```
if procedure then until := until + 1 else do := false;
```

There are many possible hardware representations for these compound symbols, including,

```
'if' procedure 'then' until := until + 1 'else' do := 'false';
IF procedure THEN until := until + 1 ELSE do := FALSE;
```

```
#IF PROCEDURE #THEN UNTIL := UNTIL + 1 #ELSE DO := #FALSE;
if Xprocedure then Xuntil := Xuntil + 1 else Xdo := false;
```

The last example illustrates the *reserved word* approach; the symbols used for keywords are not allowed as identifiers. Although this seems to be the most convenient convention, it really violates the spirit of Algol since the keywords can conflict with the identifiers. (Therefore, in this example, we have added an 'X' to the beginnings of the identifiers so they will not conflict with the reserved words.)

It is useful to distinguish the following three lexical conventions for the words of a programming language:

1. *Reserved words.* The words used by the language ('`if`', '`procedure`', etc.) are reserved by the language, that is, they cannot be used by the programmer for identifiers. This is quite readable and is essentially the convention used in natural languages. It may not be too readable to a novice in the language since it does not clearly distinguish those words that are and are not part of the language.
2. *Keywords.* This is the strict Algol approach. The words used by the language are marked in some unambiguous way (e.g., preceding with a '#', surrounding with quotes, or typing in another case or type font). This is usually difficult to type and it is not very readable unless a different font can be used.
3. *Keywords in context.* This is the FORTRAN (and PL/I) approach. Words used by the language are keywords only in those contexts in which they are expected, otherwise they are treated as identifiers. For instance, the following is a legal PL/I statement:

   ```
   IF IF THEN
      THEN = 0;
   ELSE;
      ELSE = 0;
   ```

   The first occurrences of `IF`, `THEN`, and `ELSE` are keywords, while the second occurrences are identifiers. Clearly, this convention can be confusing for both people and compilers. We saw in Chapter 2 how the keyword in context convention made it harder to catch errors in programs.

Most modern programming languages use the *reserved word* convention.

Arbitrary Restrictions Are Eliminated

Algol adheres to the Zero-One-Infinity Principle in many of its design decisions. For example, there is no limit on the number of characters in an identifier (FORTRAN allowed at most six). It also eliminates many other restrictions, for example, any expression is allowed as a subscript, including other array elements:

```
A[2 × B[i] - 1]
Count [if i > 100 then 100 else i]
```

Although these facilities are not often needed, they reduce the number of special cases the programmer must learn and they support the idea that "anything you think you ought to be

able to do, you will be able to do." All of these features contribute to Algol's reputation as a general, regular, elegant, orthogonal language.

■ ***Exercise 4-1*:*** Do you agree that you can do anything you may want to be able to do in Algol? Can you do too much already? Either propose some restrictions that should be eliminated from Algol and show that they are cost effective (i.e., their benefits outweigh their costs) or list some restrictions that should be imposed and show that they are worthwhile.

The if-Statement Had Problems

Algol's syntactic conventions did have some problems. Consider this **if**-statement:

if *B* **then if** *C* **then** *S* **else** *T*

Does the **else** go with the first **if** or the second? That is, is this statement equivalent to

if *B* **then begin if** *C* **then** *S* **else** *T* **end**

or is it equivalent to

if *B* **then begin if** *C* **then** *S* **end else** *T*

This is called the *dangling* **else** *problem*. Algol solved this problem by requiring that the consequent of an **if**-statement be an unconditional statement. Thus, the example, '**if** *B* **then if** *C* **then** *S* **else** *T*, is illegal and must be written in one of the two ways shown, depending on the programmer's intent. Other languages have adopted other solutions to this problem. For example, PL/I says that an **else** goes with the nearest preceding unmatched **if.** We will see in Chapter 8 that newer languages have avoided this problem altogether.

■ ***Exercise 4-2*:*** What do you think is the best solution to the dangling **else** problem? Take and defend a position.

Algol Defined the Style of Almost All Successors

Algol's lexical and syntactic structures became so popular that virtually all languages designed since have been "Algol-like"; that is, they have been hierarchical in structure, with nesting of both environments and control structures. Even PL/I (which started out as FORTRAN VI) adopted these conventions and many other ideas from Algol.

■ ***Exercise 4-3*:*** Pick some language designed after Algol-60 (e.g., Pascal, Ada, PL/I, Algol-68, FORTRAN 77, C) and compare its syntax with Algol's. List the major differences and similarities in both lexical and syntactic structures. Are these improvements or not?

■ ***Exercise 4-4*:*** List at least two programming languages (designed after Algol-60) that are not Algol-like. Are their conventions better than Algol's? Why do you think their designers chose these conventions?

4.2 DESCRIPTIVE TOOLS: BNF

English Descriptions of Syntax Proved Inadequate

An important skill in any design discipline is the use of *descriptive tools* to formulate, record, and evaluate various aspects of the developing design. Although English and other natural languages can often be used for this purpose, most designers have developed specialized languages, diagrams, and notations for representing aspects of their work that would otherwise be difficult to express. Imagine how hard it would be for an architect to work without elevations and floor plans or for an electrical engineer to work without circuit diagrams. In this section we will discuss a very valuable *descriptive tool* for programming language design: BNF.

Recall (Section 3.1) that the decision to use the BNF notation in the Algol-60 report resulted from Peter Naur's realization that his understanding of the Algol-58 description did not agree with that of John Backus. This is true because natural language prose is not sufficiently precise. So that we can see some of the problems inherent in using a natural language to describe syntax, we will work through a simple example.

Algol-60, like most programming languages, allows *numeric denotations* (i.e., numeric literals) to be written in three forms: integers such as '`-273`', fractions such as '`3.14159`', and numbers in scientific notation, such as '$6.02_{10}23$' (meaning 6.02×10^{23}). Here we have defined the format of numeric denotations by giving three representative examples; using examples gives the reader a clear idea of what numbers look like in Algol.

While examples are adequate for many purposes, they do leave a number of questions unanswered. For example: Are leading negative signs allowed on fractional and scientific numbers? Are negative exponents (e.g., '$3.1_{10}-50$') allowed? Is the coefficient of a scientific number required to contain a decimal point or are numbers like '$3_{10}8$' allowed?

One solution to this problem is to give more examples that cover these questionable cases. We could add that '$3.1_{10}-50$', '`-3.14159`', and '$3_{10}8$' are legal numbers, while '`3.`', '$3._{10}8$', '`--273`', '`+-273`', and so forth, are not. Unfortunately, this still does not specify precisely what a numeric denotation is. Are leading decimal points, such as '.5', allowed? How about scientific numbers without a leading coefficient (e.g., '$_{10}-$ 6' meaning 10^{-6})? Is a leading positive sign allowed (e.g., '+ 5')? Is it allowed on the exponent (e.g., '3_{10} + 8')?

We can see that what seemed to be a simple problem, specifying the syntax of a number, actually has many hidden pitfalls. Couldn't we just be more careful in our English description? After all, legal documents, which must be very precise, are written in natural languages. Let's give it a try; here is a precise description of a number.

1. A *digit* is either '`0`', '`1`', '`2`', '`3`', '`4`', '`5`', '`6`', '`7`', '`8`', or '`9`'.
2. An *unsigned integer* is a sequence of one or more *digits*.
3. An *integer* has one of the following three forms: It is either a positive sign ('+') followed by an *unsigned integer*, or a negative sign ('−') followed by an *unsigned integer*, or an *unsigned integer* preceded by no sign.
4. A *decimal fraction* is a decimal point ('.') immediately followed by an *unsigned integer.*
5. An *exponent part* is a subten symbol ('$_{10}$') immediately followed by an *integer.*
6. A *decimal number* has one of three forms: It is either an *unsigned integer*, or a *decimal fraction*, or an *unsigned integer* followed by a *decimal fraction.*

7. An *unsigned number* can take one of three forms: It is either a *decimal number*, or an *exponent part*, or a *decimal number* followed by an *exponent part.*
8. Finally, a *number* can have any one of these three forms: It can be a positive sign ('+') followed by an *unsigned number*, or a negative sign ('−') followed by an *unsigned number*, or an *unsigned number* preceded by no sign.

This is surely an exhaustive (and exhausting!) description of Algol numbers. We have the feeling that we have read a very precise specification of what a number is without having a very clear idea of what they look like. Indeed, we probably will not have a clear idea unless we make up some examples:

• unsigned integer	'273', '3', '02', '14159', '23'
• integer	'3', '+3', '−273'
• decimal fraction	'.02', '.14159'
• exponent part	'$_{10}$23', '$_{10}$−6'
• decimal number	'273', '3.1', '6.02', '3.14159'
• unsigned number	'273', '6.02$_{10}$23', '3.14159', '3$_{10}$8'
• number	'−273', '6.02$_{10}$23', '3.14159', '+3$_{10}$8'

In fact, if these examples were provided with the above description, most people would be inclined to look just at the examples and refer only to the legalistic prose when they were in doubt. The prose, although precise, is so opaque that the language designer who wrote it is likely to have some misgivings about whether it accurately expresses the ideas intended.

Backus Developed a Formal Syntactic Notation

We noted above that examples often give a better idea of a thing than a precise definition of its form. This is the way we learn most concepts, including our native language. The problem with a description by examples is that it is always incomplete. We saw in the *number* example that there was a seemingly endless list of borderline cases and questionable instances that were not handled by our examples. On the other hand, the precise English description dealt with all these borderline cases (we think) but really did not show us what a number is in a very digestible form. What we need is a way of combining the perceptual clarity of examples with the precision of formal prose.

In Chapter 2 we said that FORTRAN restricts subscript expressions to one of the following forms:

c
v
$v + c$ or $v - c$
$c*v$
$c*v + c'$ or $c*v - c'$

where c and c' are integer denotations (*unsigned integers*) and v is an integer variable. What does this have to do with BNF? It turns out that this is exactly the way Backus described

subscript expressions in the original FORTRAN I manual. It combined the advantages of both examples and a formal specification since the formulas above *look like* example expressions (e.g., '25', 'I', 'I + 25', '2*I - 1'), while at the same time it precisely described subscript expressions. For example, the above formulas say that one form of a *subscript expression* is a *c* (i.e., integer denotation), followed by a star ('*'), followed by a *v* (i.e., integer variable), followed by a plus sign ('+'), followed by another *c* (i.e., integer denotation). The formulas make use of the *syntactic categories* *c* and *v*, which stand for integer denotations and integer variables.

This notation is just as precise as our prose but much clearer in its meaning. In fact, these formulas almost draw a picture of the legal subscript expressions. Backus formalized this notation and used it to describe Algol-58.

BNF Is Naur's Adaptation of the Backus Notation

Naur adapted the Backus notation to the Algol-60 Report by making a number of improvements. It is this modified notation that is most widely known; it is called BNF (for Backus–Naur Form) in recognition of Naur's contribution.

To see how BNF works, we will look again at the number problem. We can see that in our formal prose definition of *numbers* we made use of several names for various syntactic components of numbers. For example, *integer* represented strings like '3', '+3', and '-273'; and *decimal fraction* represented strings like '.02' and '.14159'. What we were really doing was defining certain *syntactic categories* of strings, like *decimal number*, in terms of other syntactic categories, like *unsigned integer*. BNF notation represents classes of strings (syntactic categories) by words or phrases in angle brackets such as ⟨decimal fraction⟩ and ⟨unsigned integer⟩.

Let's look at part of a BNF description. In description 3 of the definition of *number*, we said that a *decimal fraction* is a decimal point ('.') followed by an *unsigned integer*. Following the principle that the description should look like the things it describes, BNF represents this as

⟨decimal fraction⟩ ::= .⟨unsigned integer⟩

The '::=' symbol can be read "is defined as"; it is reminiscent of Algol's assignment operator.

Notice that BNF preserves the descriptive advantages of examples by representing particular symbols (such as '.') by themselves, and the concatenation of strings by the juxtaposition of their descriptions. That is, the fact that the decimal point is followed by a member of the class ⟨unsigned integer⟩ is represented by writing

.⟨unsigned integer⟩

The symbols, such as '.', that represent themselves are called *terminal symbols* (*terminus* is the Latin word for limit or end), while those that represent syntactic categories, such as ⟨unsigned integer⟩, are called *nonterminals*. Nonterminal symbols must be defined in terms of other symbols (either terminal or nonterminal), but the process of definition comes to an end with the terminal symbols, because they do not have to be defined. A descriptive language like BNF is called a *metalanguage* (*meta* is Greek for after or beyond) because it is

used to describe another language, the *object language* (Algol in this case). Metalanguages are very important tools for both the language designer and the language user.

BNF Describes Alternates

Most of the definitions in our description of *number* were not as simple as that of *decimal fraction*; most of them listed several *alternative forms* that the things being described can take. For example, an ⟨integer⟩ has three different possible forms:

+ ⟨unsigned integer⟩
− ⟨unsigned integer⟩
⟨unsigned integer⟩

Clearly, *alternation* such as this is very common; BNF provides a means to express it. The symbol '|' is read as "or"; using it ⟨integer⟩ can be defined as follows:

⟨integer⟩ ::= +⟨unsigned integer⟩
| −⟨unsigned integer⟩
| ⟨unsigned integer⟩

We have written the alternatives on separate lines for readability; this is not necessary since BNF, like Algol, is free form.

Recursion Is Used for Repetition

Using the elements of BNF that we have described so far, all of the rules defining *numbers* can be expressed, except one. This is the second rule, which defines an ⟨unsigned integer⟩ to be a sequence of one or more ⟨digit⟩s. How can we express the idea "sequence of one or more of . . ."?

One way to solve a problem like this is to ask how we can generate a member of this syntactic category. In this case, the question is: How can we generate an ⟨unsigned integer⟩? The definition says that any sequence of one or more ⟨digit⟩s is an ⟨unsigned integer⟩, so if we want an ⟨unsigned integer⟩, we can start by picking any ⟨digit⟩. For example, '2' is an ⟨unsigned integer⟩. Now, what do we do if we want a longer ⟨unsigned integer⟩? We can take our string of length one and pick another ⟨digit⟩, say '7', and add it to the end giving the ⟨unsigned integer⟩ '27'. This process can be continued. We can pick another ⟨digit⟩, say '3', and append it to the ⟨unsigned integer⟩ '27' to get the ⟨unsigned integer⟩ '273'.

Thus, we can get an ⟨unsigned integer⟩ in exactly two ways: either by taking a single ⟨digit⟩ or by adding a ⟨digit⟩ to an ⟨unsigned integer⟩ that we already have. It is now straightforward to write this in BNF:

⟨unsigned integer⟩ ::= ⟨digit⟩
| ⟨unsigned integer⟩ ⟨digit⟩

Notice that this is a *recursive* definition, that is, ⟨unsigned integer⟩ is defined in terms of itself. It is not a *circular* definition because the single ⟨digit⟩ provides a *base* or starting point

for the recursion. That is, one of the alternatives in the definition of ⟨unsigned integer⟩ is not recursive. This is called a *well-founded recursive definition.*

Recursive definitions such as this are the usual way of specifying sequences of things in BNF. We will see shortly that some dialects of BNF have additional notations for expressing sequences in a more readable form.

Recursion is also used for describing nested syntax such as Algol's control structures. We will see an example of this later. Before we get to these things, make sure that you thoroughly understand the BNF definition of ⟨number⟩, the syntactic category of numeric denotations, in Figure 4.1.

■ ***Exercise 4-5:*** Show that the following strings satisfy the definition of ⟨number⟩: '-273', '$6.02_{10}23$', '3.14159', '$+3_{10}8$', '$3.1_{10}-50$', '$_{10}-6$', '$.5$', '-0'.

■ ***Exercise 4-6:*** Show that the following strings do not satisfy the definition of ⟨number⟩: '$-\ -273$', '$3.$', '$3._{10}-8$', '$2_{10}0.5$', '$._{10}2$', '$3.-14159$'.

■ ***Exercise 4-7:*** Write a BNF description of identifiers that have the following syntax. An identifier is a string of letters, digits, and underscores ('_'), subject to the following restrictions: (1) an identifier must begin with a letter, (2) consecutive underscores are not permitted, and (3) an identifier may not end with an underscore. *Hint:* Start by writing out some examples of legal and illegal identifiers so that you can identify the pattern.

⟨unsigned integer⟩ ::= ⟨digit⟩
| ⟨unsigned integer⟩ ⟨digit⟩

⟨integer⟩ ::= + ⟨unsigned integer⟩
| − ⟨unsigned integer⟩
| ⟨unsigned integer⟩

⟨decimal fraction⟩ ::= .⟨unsigned integer⟩

⟨exponent part⟩ ::= 10 ⟨integer⟩

⟨decimal number⟩ ::= ⟨unsigned integer⟩
| ⟨decimal fraction⟩
| ⟨unsigned integer⟩ ⟨decimal fraction⟩

⟨unsigned number⟩ ::= ⟨decimal number⟩
| ⟨exponent part⟩
| ⟨decimal number⟩ ⟨exponent part⟩

⟨number⟩ ::= + ⟨unsigned number⟩
| − ⟨unsigned number⟩
| ⟨unsigned number⟩

Figure 4.1 BNF Definition of Numeric Denotations

Extended BNF Is More Descriptive

Several extensions have been made to BNF to improve its readability. Often these directly reflect the words and phrases commonly used to describe syntax. For example, in the prose description of *number*, we defined an *unsigned integer* to be a "sequence of one or more *digits*." This was expressed in BNF through a recursive definition. Although we can get used to recursive descriptions such as this, it would really be more convenient to have a notation that directly expresses the idea "sequence of one or more of. . . ." One such notation is the *Kleene cross*,[1] C^+, which means a sequence of one or more strings from the syntactic category C. Using this notation, the syntactic category ⟨unsigned integer⟩ can be defined:

$$\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle^+$$

A useful variant of this notation is the *Kleene star*, C^*, which stands for a sequence of *zero* or more elements of the class C. For example, an Algol identifier (name) is a ⟨letter⟩ followed by any number of ⟨letter⟩s or ⟨digit⟩s (i.e., a sequence of zero or more ⟨alphanumeric⟩s). This can be written as

$$\langle \text{alphanumeric} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{digit} \rangle$$
$$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \langle \text{alphanumeric} \rangle^*$$

If ⟨alphanumeric⟩ is used in only one place, namely, the definition of ⟨identifier⟩, then it is pointless to give it a name. Rather, it would be better to be able to say directly a "sequence of ⟨letter⟩s or ⟨digit⟩s." Some dialects of BNF permit this by allowing us to substitute '{⟨letter⟩ | ⟨digit⟩}' for ⟨alphanumeric⟩:

$$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \{\langle \text{letter} \rangle \mid \langle \text{digit} \rangle\}^*$$

This can be read, "An identifier is a letter followed by a sequence of zero or more letters or digits." It can be made even more pictorial by stacking the alternatives:

$$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \left\{ \begin{matrix} \langle \text{letter} \rangle \\ \langle \text{digit} \rangle \end{matrix} \right\}^*$$

Let's see if there are any other improvements that can be made to this notation. The rule for an ⟨integer⟩ is

$$\begin{aligned} \langle \text{integer} \rangle \quad ::= \quad & +\langle \text{unsigned integer} \rangle \\ \mid \quad & -\langle \text{unsigned integer} \rangle \\ \mid \quad & \langle \text{unsigned integer} \rangle \end{aligned}$$

One immediate simplification should be apparent; we can use our alternative notation

$$\begin{aligned} \langle \text{integer} \rangle ::= & \left\{ \begin{matrix} + \\ - \end{matrix} \right\} \langle \text{unsigned integer} \rangle \\ \mid \quad & \langle \text{unsigned integer} \rangle \end{aligned}$$

[1] Pronounced "klane-uh" and named after the mathematician and logician S. C. Kleene (1909–).

This is better; but the idea that we really want to express is that the ⟨unsigned integer⟩ is *optionally* preceded by a '+' or '−'. The square bracket is often used for this purpose, that is, '[+ | −]' or

$$\langle \text{integer} \rangle ::= \left[\begin{matrix} + \\ - \end{matrix} \right] \langle \text{unsigned integer} \rangle$$

This is a very graphic notation; it *shows* us what an integer looks like. A two-dimensional metalanguage like this was first used in the 1960s to describe the COBOL language. Figure 4.2 shows the definition of ⟨number⟩ using this extended BNF:

Why is this extended BNF preferable to pure BNF? It is because it adheres better to the Structure Principle, that is, the forms of the extended BNF definitions are closer (visually) to the strings they describe than are the pure BNF definitions. In other words, extended BNF has more of the advantages of examples.

■ ***Exercise 4-8:*** Show that the extended BNF definitions in Figure 4.2 are equivalent to (i.e., describe the same class of strings as) the pure BNF definitions in Figure 4.1.

■ ***Exercise 4-9:*** Write an extended BNF definition of the class of identifiers described in Exercise 4-7.

BNF Led to a Mathematical Theory of Programming Languages

In the late 1950s, Noam Chomsky, a linguist at MIT, was attempting to develop a mathematical theory of natural languages. He produced a mathematical description of four different classes of languages that are now known as

- Chomsky type 0, or *recursively enumerable* languages
- Chomsky type 1, or *context-sensitive* languages
- Chomsky type 2, or *context-free* languages
- Chomsky type 3, or *regular* languages

Each of these classes includes the one below it; for example, all regular languages are context-free, and all context-free languages are context-sensitive.

Chomsky also defined a class of *grammars*, or language descriptions, corresponding to each of these language classes. For example, each *context-free grammar* describes a corre-

$$\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle^{+}$$

$$\langle \text{integer} \rangle ::= \left[\begin{matrix} + \\ - \end{matrix} \right] \langle \text{unsigned integer} \rangle$$

$$\langle \text{decimal number} \rangle ::= \left\{ \begin{matrix} \langle \text{unsigned integer} \rangle \\ [\langle \text{unsigned integer} \rangle]. \ \langle \text{unsigned integer} \rangle \end{matrix} \right\}$$

$$\langle \text{number} \rangle ::= \left[\begin{matrix} + \\ - \end{matrix} \right] \left\{ \begin{matrix} \langle \text{decimal number} \rangle \\ [\langle \text{decimal number} \rangle]_{10} \ \langle \text{integer} \rangle \end{matrix} \right\}$$

Figure 4.2 Extended BNF Definition of Numeric Denotations

sponding context-free language, and, conversely, each context-free language can be described by a context-free grammar. There may be several alternative ways of describing the same language so there may be several grammars corresponding to the same language.

It was quickly realized that Backus's notation is equivalent to Chomsky's context-free (or type 2) grammars, so that the class of languages describable by BNF is exactly the context-free languages. Since Chomsky had developed a mathematical theory of these classes of grammars and languages, the connection between BNF and the Chomsky hierarchy immediately enabled the mathematical analysis of the syntax and grammar of programming languages. This has had important practical benefits since it has permitted the development of automatic parser-generators, thus automating what had been one of the more difficult parts of compiler writing.

Context-Free and Regular Grammars Are the Most Useful

Although a full treatment of formal languages is beyond the scope of this book, we will discuss the significance of the two most important classes of languages and grammars: context-free and regular.

There are mathematical definitions of context-free and regular languages, but the simplest way to see the difference is by reference to the extended BNF notation. A *regular grammar* is one that is written in extended BNF without the use of any recursive rules. For example, the definition of ⟨number⟩ in Figure 4.2 is an example of a regular grammar. A language is formally defined as the set of strings described by a grammar. Therefore, a *regular language* is a language that can be described by a regular grammar. For example, the language defined by ⟨number⟩, which includes the strings '-273', '$6.02_{10}23$', '3.14159', '$+3_{10}8$', is a regular language. Notice that we said that a regular language is one that *can* be described by a regular grammar, not one that *must* be. The distinction is important since the grammar in Figure 4.1 uses recursion and defines the same language as that in Figure 4.2.

A *context-free* grammar is any grammar that can be expressed in extended BNF, possibly including recursively defined nonterminals. Analogously, a *context-free language* is a language that can be described by a context-free grammar. All regular grammars are also context-free grammars, and all regular languages are also context-free languages.

Thus, the difference between regular and context-free grammars is in the use of recursion in the definitions. What are the implications of this? Basically, recursion allows the definition of *nested* syntactic structures. We have already seen the importance of nesting in Algol. Nesting is easy to express in BNF, which probably encouraged the widespread use of nested constructs in Algol. Consider this simplified form of Algol's definition of a ⟨statement⟩:

⟨unconditional statement⟩ ::= { ⟨assignment statement⟩ | **for** ⟨for list⟩ **do** ⟨statement⟩ }

⟨statement⟩ ::=

{ ⟨unconditional statement⟩ | **if** ⟨expression⟩ **then** ⟨unconditional statement⟩ | **if** ⟨expression⟩ **then** ⟨unconditional statement⟩ **else** ⟨statement⟩ }

We can see that ⟨statement⟩ is defined recursively in terms of itself since part of an **if**-statement is itself a ⟨statement⟩. Also, ⟨statement⟩ is defined in terms of ⟨unconditional statement⟩,

which is in turn defined in terms of ⟨statement⟩; this is an example of *indirect recursion.* This recursive definition is not circular because the ⟨assignment statement⟩ alternative (which is not defined in terms of ⟨statement⟩) provides a *base* for the recursion.

It is easy to see that these definitions allow the nesting of statements. For example, since `x := x + 1` is an ⟨assignment statement⟩, it is also an ⟨unconditional statement⟩. Therefore, it can be made part of an **if**-statement, for instance,

```
if x < 0 then x := x + 1
```

Since this **if**-statement is itself a ⟨statement⟩, it can be made the part of a **for**-loop, for example,

```
for i := 1 step 1 until n do if x < 0 then x := x + 1
```

This process of embedding statements into yet larger statements can be continued indefinitely. Before the development of the BNF notation, it would have been much more complicated to describe these recursively nested structures.

Since regular grammars do not permit recursion, it is not possible to specify indefinitely deep nesting in a regular grammar. Thus, the major difference between regular and context-free languages is the possibility of indefinitely deep nesting in the latter. A simple example of this is the language of balanced parentheses. This is a make-believe language whose only strings are sequences of properly balanced parentheses. For example,

```
( ( ( ) ) ) ( ) ( ( ( ) ( ) ) )
```

is an expression in this language, while `( ( ( ( )` is not. This language is easy to describe as a context-free grammar:

⟨expression⟩ ::= ⟨balanced⟩*

⟨balanced⟩ ::= (⟨expression⟩)

(*N.B.* It may look as though there is no base to the recursive definitions above, but that is not the case since the Kleene star notation allows *no* repetitions of ⟨balanced⟩ to be an ⟨expression⟩.)

- ***Exercise 4-10:*** Show that the above grammar describes the balanced and only the balanced strings of parentheses.
- ***Exercise 4-11:*** Show that *limited* nesting does not require context-free rules. *Hint:* Show that strings of parentheses nested at most three (or any fixed number) deep can be defined by a regular grammer.

4.3 DESIGN: ELEGANCE

The Value of Analogies

Programming language design is a comparatively new activity—it has existed for less than half a century, so it is often worthwhile to look to older design disciplines to understand this new activity better. Indeed, this book grew out of a study of teaching methods in architec-

ture, primarily, but also of pedagogy in other disciplines, such as aircraft design. Perhaps you have also seen analogies drawn between programming languages and cars (FORTRAN = Model T, C = dune buggy, etc.).

These analogies can be very informative, and can serve as "intuition pumps" to enhance our creativity, but they cannot be used uncritically because they are, in the end, just analogies. Ultimately our design decisions must be based on more than analogies, since analogies can be misleading as well as informative.

In this section we will consider the role of aesthetics in programming language design, but I will base my remarks on a book about structural engineering, *The Tower and the Bridge*, by David P. Billington (Basic Books, 1983). Although there are many differences between bridges and programming languages, we will find that many ideas and insights transfer rather directly.

Design Has Three Dimensions

According to Billington, there are three values common to many technological activities, which we can call "the three E's": *Efficiency*, *Economy*, and *Elegance*. These values correspond to three dimensions of technology, which Billington calls the *scientific*, *social*, and *symbolic* dimensions (the three S's). We will consider each in turn.

Efficiency Seeks to Minimize Resources Used

In structural engineering, *efficiency* deals with the amount of material used; the basic criterion is safety and the issues are *scientific* (strength of materials, disposition of forces, etc.). Similarly, in programming language design, efficiency is a scientific question dealing with the use of resources. We have seen many examples in which efficiency considerations influenced programming language design. In the early days, the resources to be minimized were often run-time memory usage and processing time, although compile-time resource utilization was also relevant. In other cases the resource economized was programmer typing time, and we have seen cases in which this compromised safety (e.g., FORTRAN's implicit declarations). We have also seen cases in which security (i.e., safety) was sacrificed for the sake of efficiency by neglecting run-time error checking (e.g., array bounds checking).

Efficiency issues often can be quantified in terms of computer memory or time, but we must be careful that we are not comparing apples and oranges. Compile-time is not interchangeable with run-time, and neither is the same as programmer time. Similarly, computer memory cannot be traded off against computer time unless both are reduced to a common denominator, such as money, but this brings in economic considerations, to which we now turn.

Economy Seeks to Maximize Benefit versus Cost

Whereas efficiency is a scientific issue, *economy* is a *social* issue. In structural engineering, economy seeks to maximize social benefit compared to its cost. (This is especially appropriate since structures such as bridges are usually built at public expense for the benefit of the public.) In programming language design, the "public" that must be satisfied is the pro-

gramming community that will use the language and the institutions for which these programmers work.

Economic trade-offs are hard to make because economic values change and are difficult to predict. For example, the shift from the first to the second generation was largely a result of a decrease in the cost of computer time compared to programmer time. We will see that the shift from the second to the third generation involves the increasing cost of residual bugs in programs, and the fourth generation reflects the increasing cost of program maintenance compared to program development.

Other social factors involved in the success or failure of a programming language include whether major manufacturers support the language, whether prestigious universities teach it, whether it is approved in some way by influential organizations (such as the U.S. Department of Defense), whether it has been standardized, whether it comes to be perceived as a "real" language (used by "real programmers") or as a "toy" language (used by novices or dilitantes), and so forth. As can be seen from the historical remarks in this book, social factors are often more important than scientific factors in determining the success or failure of a programming language.

Often economic issues can be quantified in terms of money, but the monetary values of costs and benefits are often unstable and unpredictable because they depend on changing market forces. Also, many social issues, from dissatisfaction with poorly designed software to human misery resulting from system failures, are inaccurately represented by the single dimension of monetary cost. All kinds of "cost" and "benefit" must be considered in seeking an economical design. Recall also that technology is inherently nonneutral; we must face up to its ambivalent advantages and disadvantages (Section 1.4).

Elegance Symbolizes Good Design

"Elegance? Who cares about elegance?" snorts the hard-nosed engineer, but Billington shows clearly the critical role of elegance in "hard-nosed" engineering.

We have already seen the problem that feature interaction poses for language designers and have seen the difficulty of analyzing all the possible interactions of features in a language. Structural engineers face similar problems of analytic complexity, but Billington observes that the best designers do not make extensive use of computer models and calculation.

One reason is that mathematical analysis is always incomplete. The engineer must make a decision about which variables are significant and which are not, and an analysis may lead to incorrect conclusions if this decision is not made well. Also, equations are often simplified (e.g., made linear) to make their analysis feasible, and this is another potential source of error. Because of these limitations, engineers that depend on mathematical analysis may overdesign a structure to compensate for unforseen effects left out of the analysis. Thus the price of safety is additional material and increased cost.

Similarly in programming language design, the limitations of the analytic approach often force us to make a choice between an underengineered design, in which we run the risk of unanticipated interactions, and an overengineered design, in which we have confidence, but which is inefficient or uneconomical.

Many of you will have seen the famous film of the collapse in 1940 of the 4-month-old Tacoma Narrows bridge; it vibrated itself to pieces in a storm because *aerodynamic* stabil-

ity had not been considered in its design. Billington explains that this accident, along with a number of less dramatic bridge failures, was a consequence of an increasing use of theoretical analyses that began in the 1920s. However, the very problem that destroyed the Tacoma Narrows bridge had been anticipated and avoided a century before by bridge designers who were guided by aesthetic principles.

According to Billington, the best structural engineers do not rely on mathematical analysis (although they do not abandon it altogether). Rather, their design activity is guided by a sense of *elegance.* This is because solutions to structural engineering problems are usually greatly underdetermined, that is, there are many possible solutions to a particular problem, such as bridging a particular river. Therefore, expert designers restrict their attention to designs in which the interaction of the forces is easy to see. The design looks unbalanced if the forces are unbalanced, and the design *looks* stable if it *is* stable.

The general principle is that designs that *look* good will also *be* good, and therefore the design process can be guided by aesthetics without extensive (but incomplete) mathematical analysis. Billington expresses this idea by inverting the old architectural maxim and asserting that, in structural design, *function follows form.* He adds, "When the form is well chosen, its analysis becomes astoundingly simple." In other words, the choice of form is open and free, so we should select forms in which elegant design expresses good design (i.e., efficient and economical design). If we do so, then aesthetics can guide design.

The same applies to programming language design. By restricting our attention to designs in which the interaction of features is manifest—in which good interactions look good and bad interactions look bad—we can let our aesthetic sense guide our design and we can be much more confident that we have a good design, without having to check all the possible interactions.

In this case, what is good for the designer is also good for the user. Nobody is comfortable crossing a bridge that looks as if it will collapse any moment, and nobody is comfortable using a programming language in which features may "explode" if combined in the wrong way. The manifest balance of forces in a well-designed bridge gives us confidence when we cross it. So also, the manifestly good design of our programming language should reinforce our confidence when we program in it, because we have (well-justified) confidence in the consequences of our actions.

We accomplish little by covering an unbalanced structure with a beautiful facade. When a bridge is unable to sustain the load for which it was designed, and collapses, its outside beauty will not matter much. So also in programming languages. If the elegance is only superficial, that is, if it is not the manifestation of a deep coherence in the design, then programmers will quickly see through the illusion and lose their (unwarranted) confidence.

In summary, good designers choose to work in a region of the design space in which good designs look good. As a consequence, these designers can rely on their aesthetic sense, as can the users of the structures (bridges or programming languages) they design. We may miss out on some good designs this way, but they are of limited value unless both the designer and the user can be confident that they are good designs. We may summarize the preceding discussion in

The Elegance Principle

Confine your attention to designs that *look* good because they *are* good.

The Programming Language as a Work Environment

There are other reasons that elegance is relevant to a well-engineered programming language. The programming language is something the professional programmer will live with—even live *in.* It should feel comfortable and safe, like a well-designed home or office; in this way it can contribute to the quality of the activities that take place within it. Would you work better in an oriental garden or a sweatshop?

A programming language should be a joy to use. Like any tool, we should be able to work through it, not against it. This will encourage its use and decrease the programmer's fatigue and frustration. The programming language should not be a hindrance, but should serve more as a collaborator, encouraging programmers to do their jobs better. As some automobiles are "driving machines" and work as a natural extension of the driver, so a programming language should be a "programming machine" by encouraging the programmer to acquire the smooth competence and seemingly effortless skill of a virtuoso. The programming language should *invite* the programmer to design elegant, efficient, and economical programs.

Through its aesthetic dimension a programming language symbolizes many values. For example, in the variety of its features it may symbolize profligate excess, sparing economy, or asceticism; the nature of its features may represent intellectual sophistication, down-to-earth practicality, or ignorant crudeness. It inclines us to focus on certain issues and to act in certain ways. Thus a programming language can promote a set of values. By embodying certain values, it encourages us to think about them; by neglecting or negating other values, it allows them to recede into the background and out of our attention. Out of sight, out of mind.

A Sense of Elegance Is Acquired Through Design Experience

Aesthetics is notoriously difficult to teach, so you may wonder how you are supposed to acquire that refined sense of elegance necessary to good design. Billington observes that this sense is acquired through extensive experience in design, which, especially in Europe, is encouraged by a competitive process for choosing bridge designers. Because of it, structural engineers design many more bridges than they build, and they learn from each competition they lose by comparing their own designs with those of the winner and other losers. The public also critiques the competing designs, and in this way becomes more educated; their sense of elegance develops along with that of the designers.

So also, to improve as a programming language designer you should design many languages—design obsessively—and criticize, revise, and discard your designs. You should also evaluate and criticize other people's designs and try to improve them. In this way you will acquire the body of experience you will need when the "real thing" comes along.

- ***Exercise 4-12*:*** What automobiles do you think are analogous to FORTRAN, Algol-60, and some other language with which you are familiar? Defend your analogies and compare them with your classmates' to see what your analogies reveal about your values. That is, explore the characteristics that you choose to emphasize in your analogies.

- ***Exercise 4-13*:*** Develop some other systematic analogy with programming languages. Some ideas are aircraft (see section 5.1 on PL/I as an airplane with 7000 switches etc.),

musical styles ("baroque" languages etc.), musicians, musical instruments (which language is the theater pipe-organ?), buildings (the Parthenon?), and tools (e.g., the Swiss army knife, the sledge hammer, the chain saw). Or select some other domain of interest. (Designed objects tend to produce better analogies to programming languages than do natural objects.)

■ ***Exercise 4-14*:*** Name several of the principles discussed that contribute to the idea that good designs look good, or that show the "balance of forces." Explain how these principles accomplish this goal.

4.4 EVALUATION AND EPILOG

Algol-60 Never Achieved Widespread Use

Algol-60 began with ambitious goals; it was to be a universal programming language for communicating algorithms to both people and machines. Unfortunately, it never achieved widespread use in the United States and was only moderately successful in Europe. What were the reasons for Algol's failure?

Algol Had No Input-Output

One widely cited reason for Algol's lack of success is its lack of input-output statements. This has even been a source of ridicule. For example, people have said, "Algol is the perfect programming language, except that you cannot read in any data or print out any results!" Clearly, Algol's designers realized that a language without input-output was worthless, so why didn't they specify it?

To understand their reasoning, we must remember that Algol-60 was designed at a time when there was little uniformity among input-output conventions; there were few standards. We discussed this situation in Section 4.1, along with its effect on Algol's lexical conventions. Another result of this situation was the input-output statements of FORTRAN I, which were very dependent on the IBM 704. Only later versions of FORTRAN achieved a degree of machine independence.

Faced with this situation, and believing that input-output was not a major part of most scientific programs, Algol's designers decided to omit all input-output statements. They felt that input-output statements would make this intended universal language too machine dependent. Rather, they decided that input-output would be accomplished by calling library procedures. It was their intention that each implementation of Algol would provide a set of procedures appropriate for the machine on which it was implemented.

Eventually several sets of standard input-output procedures were designed for Algol. Some of these were based on experience with later versions of FORTRAN and with COBOL. Since COBOL was a language for commercial programming, it could not avoid the issues of machine-independent input-output. Unfortunately, standardized input-output came too late to Algol to correct the damage done. The momentum of Algol's introduction had been lost and could not be recovered.

Algol Also Directly Competed with FORTRAN

Although there are many other reasons for Algol's failure, one that deserves mention is that it directly competed with FORTRAN. It *was* designed for the same application area—scientific computation. Furthermore, within this application area Algol's unique features (e.g., nested control structures, block structure, and recursion) were not considered very important. Instead, they were seen as causes for inefficiency, which was a very important issue in the early 1960s. This class of users focused more on Algol's *reductive* aspects than on its *ampliative* aspects (Section 1.4).

We should realize that four years passed from the initial excitement accompanying the Algol-58 Report to the Revised Report in 1962, and there was still no input-output. In the meantime, FORTRAN had gained considerable ground and acquired a large user community; standardization efforts were under way and most manufacturers were developing compilers. The coup de grace came when IBM, which had been considering supporting Algol, decided instead to reaffirm its commitment to FORTRAN. Most potential users of Algol saw that the costs of a switch were too great and the benefits too meager. Therefore, use of FORTRAN spread while Algol became an almost exclusively academic language.

Algol Was a Major Milestone in Programming Languages

It is remarkable that although Algol never achieved widespread use, it is one of the major milestones in programming language development. This is partly reflected in the terms it added to the programming language vocabulary, which include type, formal parameter, actual parameter, block, call by value, call by name, scope, dynamic arrays, and global and local variables.[2]

The development of Algol compilers led to a number of concepts that will be discussed in Chapter 6, including activation records, thunks, displays, and static and dynamic chains. Algol not only has introduced programming language terminology, but has motivated much of the work in programming language design and implementation that has occurred since the late 1950s. Perils has said that Algol "was to become a universal tool with which to view, study and proffer solutions to almost every kind of problem in computation."[3] We will discuss some of the indirect accomplishments of Algol.

Naturally, Algol has influenced most succeeding programming languages. For example, there was Algol-68, which was a direct generalization of Algol-60. Since we have seen that in some respects Algol-60 was already *too* general, you will not be surprised to learn that Algol-68 has been even less successful than Algol-60. Nevertheless, Algol-68 has had a considerable impact of its own by introducing new terms and concepts into computer science.

Eventually almost all programming languages became "Algol-like," that is, block-structured, nested, recursive, and free form. Fascination with Algol's technological advances defined a trajectory of increasing generality. Unfortunately, the over-generality of Algol-68 and other later languages, such as PL/I, which included many ideas from Algol, has led some computer scientists to say, "Algol was indeed an achievement; it was a significant advance

---

[2] This list is adapted from Perlis (1978).

[3] Perlis (1978).

on most of its successors."[4] As we will discuss in Chapter 5, this notion led to the quest for simplicity in language design and description that ultimately resulted in Pascal.

Algol has also formed the basis for several direct extensions, one of the most important of which is Simula. Simula is basically Algol-60 with a new feature added, called the *class*, that allows the grouping of related procedures and data declarations. The class is an example of an *encapsulation mechanism*, an important concept discussed in Chapter 7. Simula was also an early example of an *extensible language* (discussed in Chapter 5) and ultimately provided many of the ideas upon which the Smalltalk system and other object-oriented programming languages are based (Chapter 12).

As more and more languages became Algol-like, computer architects began to realize that the effective performance of their computers could be increased by building in mechanisms to support Algol implementation. In the 1960s the Burroughs B5500 was one of the first computers to do this. Now almost all computers, even microcomputers, provide some support for block-structured, recursive programming languages. Thus, Algol has had a direct effect on computer architecture.

Also, as we have discussed, the use of BNF in the Algol Report led to the development of the mathematical theory of formal languages and to automatic means for generating programming language parsers. To date, compilers remain the most successful domain of automatic software generation.

Finally, the imprecision in the English-language descriptions of semantics stimulated a tremendous amount of research into the formal specification of semantics. The goal, which is yet to be reached, is to be able to automate the semantic parts of compiler writing to the same extent as the syntactic parts. Thus, Algol, although a commercial failure, was a scientific triumph. It has remained an inspiration to later language designers. Naur has said, "The language demonstrates what can be achieved if generality and economy of concept is pursued and combined with precision in description."[5]

Characteristic of Second-Generation Programming Languages

Algol-60 was the first *second-generation* programming language, and its characteristics are typical of the entire generation. Broadly, we can say the second-generation structures are elaborations and generalizations of the corresponding first-generation structures.

First, consider the *data structures*, which are very close to first-generation structures. There are some simple generalizations, such as arrays with lower bounds other than 1, and dynamic arrays, but by and large the structures are still linear and closely patterned on machine addressing modes. However, second-generation languages do usually have strong typing of the built-in types (there aren't user-defined types).

The second generation makes one of its biggest contributions in its *name structures*, which are hierarchically nested. This permits both better control of the name space and efficient dynamic memory allocation. The introduction of block structure is perhaps the most characteristic attribute of this language generation.

---

[4] Perlis (1978).

[5] Naur (1978).

Another characteristic of the second generation is its *structured control structures*, which, by hierarchically structuring the control flow, eliminate much of the need for confusing networks of **goto**s. The second generation also elaborated many of the first generation's control structures. Some of these elaborations are important contributions, such as recursive procedures and the idea of a choice of parameter passing modes. Others are more questionable, and the second generation is known for a proliferation of baroque and expensive constructs.

In its *syntactic structures* the second generation saw a shift away from fixed formats, toward free format languages with machine-independent lexical conventions. A number of languages shifted to keyword or reserved word policies, although the keyword-in-context rule was also used (e.g., PL/I).

In general, the second generation can be seen as the full flowering of the technology of language design and implementation. The many new techniques developed in this period encouraged unbridled generalization—with both desirable and undesirable consequences. We will see that the third generation tried to compensate for the excesses while retaining the accomplishments.

EXERCISES

1. Given the widespread use of the ASCII character set, do you think it is still wise to distinguish among reference, publication, and hardware languages? Discuss the pros and cons.

2. Name a part of the syntax of most programming languages that *cannot* be expressed as a regular grammar, that is, that requires a context-free grammar.

3. Write a BNF or extended BNF description of legal FORTRAN IV subscript expressions.

4. Write a BNF or extended BNF description of Algol-60's **for**-loop, including all the different kinds of for-list-elements. Assume that ⟨variable⟩, ⟨statement⟩, ⟨Boolean expression⟩, and ⟨arithmetic expression⟩ are already defined.

5. (Difficult) Write a BNF or extended BNF grammar that describes the solution of the dangling **else** problem that matches an **else** with the nearest preceding unmatched **then**.

6. A very common syntactic pattern in programming language descriptions is "a sequence of one or more . . . separated by . . ." For example, an ⟨actual parameter list⟩ can be defined as a sequence of one or more ⟨expression⟩s separated by commas. Also, a ⟨compound statement⟩ can be defined as a **begin,** followed by one or more ⟨statement⟩s separated by semicolons, followed by an **end.** Show how to express these patterns in BNF and extended BNF. Design a new extension to these notations that simplifies expressing this pattern.

7. Some programming languages are not truly context-free. For example, some languages (such as Ada, Chapters 7 and 8) allow an identifier to be placed on the **end** at the end of a procedure declaration:

```
procedure ⟨name⟩ (⟨formals⟩) is
   ⟨declarations⟩
begin
   ⟨statements⟩
end ⟨name⟩;
```

This is not context-free because the ⟨name⟩ on the **end** is required to match the ⟨name⟩ of the procedure. Invent a syntactic notation for expressing these context-sensitive dependencies.

8. Do you think it is wise of the Algol committee to have omitted input-output from Algol-60? Defend your position and then do one of the following two exercises.

9. Design an extension to Algol-60 to handle input-output.

10. Specify a procedure library that Algol programs can invoke to perform input-output.

11. Suppose you have to write a numerical or scientific program and that both FORTRAN and Algol compilers were available. Describe how you would decide which to use.

12. Read, summarize, and critique Knuth's article, "The Remaining Trouble Spots in ALGOL 60" (*Commun. ACM 10*, 10, 1967).

13. Study and critique Algol-68 (van Wijngaarden et al., "Revised Report on the Algorithmic Language Algol 68," *Acta Inf. 5*, 1975, or *SIGPLAN Notices 12*, 5, 1977). Since orthogonality was a major design goal of Algol-68, you should pay particular attention to orthogonality in your critique.

14. Study the PL/I language (*Am. Natl. Stand. Prog. Lang. PL/I*, ANS X3.53-1976, ANSI, New York, 1976). Identify features and ideas derived from FORTRAN and Algol-60.