Appendix K

EBNF

Context-free grammars are a method of describing the syntax of programming languages. Context-free grammars are most often written in Backus-Naur Form, or BNF, in honor of the work done by John Backus and Peter Naur in creating the Algol 60 report. In this book, we use extended BNF, or simply EBNF, which makes it easier to specify optional and repeated items (Wirth 1977a).

An EBNF grammar consists of a list of grammar *rules*. Each rule has the form:

$A ::= \alpha$

where A is a *nonterminal symbol*, and α is a collection of alternatives separated by vertical bars. Each alternative is a sequence, and in the simple case, each element of the sequence is either a nonterminal symbol or *literal text* (in `typewriter font`).

A non-terminal symbol represents all the phrases in a syntactic category. Thus, *toplevel* represents all legal top-level inputs, *exp* all legal expressions, and *name* all legal names. Literal text, on the other hand, represents characters that appear *as is* in syntactic phrases.

Consider the rule for *toplevel* in Impcore:

toplevel ::= *exp*
  | `(use` *file-name*`)`
  | `(val` *variable-name exp*`)`
  | `(define` *function-name* `(`*formals*`)` *exp*`)`

This rule can be read as asserting that a legal *toplevel* input is exactly one of the following:

- A legal *exp*
- A left parenthesis followed by the word `use`, then a file name, and then a right parenthesis
- A left parenthesis followed by the word `val`, then a variable name, then a legal *exp*, and then a right parenthesis
- A left parenthesis followed by the word `define`, then a function name, a left parenthesis, whatever is permitted as *formals*, a right parenthesis, an *exp*, and finally a right parenthesis

A set of such rules is called a *context-free grammar*. It describes how to form the phrases of each syntactic category, in one or more ways, by combining phrases of other categories and specific characters in a specified order.

For another example, the phrase

```
(set x 10)
```

is a *toplevel* input by the following reasoning:

- An *input* can be an *expression*.
- An *expression* can be a left parenthesis and the word `set`, followed by a *variable* and an *expression*, followed by a right parenthesis.
- A *variable* is a *name*, and a *name* is a sequence of characters which may be the sequence "x" (we appeal to the English description of this category).
- An *expression* can be a *value*, a *value* is an *integer*, and `10` is an *integer*.

The explanation above is not the whole story. In addition to a nonterminal symbol or literal text, a sequence may contain a collection of alternatives in brackets. EBNF offers three kinds of brackets:

- Parentheses $(\cdots)$ stand for a choice of exactly one of the bracketed alternatives.
- Square brackets $\big[\cdots\big]$ stand for a choice of either nothing (the empty sequence), or exactly one of the bracketed alternatives.
- Braces $\big\{\cdots\big\}$ stand for a sequence of zero or more items, each of which is one of the bracketed alternatives.

In each case, alternatives within brackets are separated by a vertical bar $(|)$.

For example, this rule shows that *formals* stands for a sequence of zero or more variable names:

formals ::= $\big\{$*variable-name*$\big\}$

Similarily, the EBNF phrase "(*function-name* $\big\{$*exp*$\big\}$)" stands for a function name followed by a sequence of zero or more argument expressions, all in parentheses.

The topic of context-free grammars is an important one in computer science. It should be covered in depth in almost any introductory theory or compiler-construction book. Good sources include those from Aho, Sethi, and Ullman (1986), Barrett et al. (1986), and Hopcroft and Ullman (1979).