

## BASICS

All programming languages start with a syntactic definition, or grammar. Following this grammar enables programmers to write valid programs that can be compiled (or interpreted) to produce results when the program is executed. For formal semantics, however, we are not interested in building real compilers or interpreters. In fact, one of the goals is get away from the real machine so that the semantic definition is platform independent. We can therefore forget about defining a *concrete* syntax with all its complexity of punctuation, operators and dependencies, and instead use a much simpler grammar form called *abstract* syntax.

The easy way to think of this is to use the minimum amount of concrete syntax that can tell us what semantic form is intended. Another way is to think of parse trees (the usual output of a syntactic parser) as the subject of the grammar, rather than character strings, which are the subject of concrete syntax definitions. As an example of the difference, take a definition of a floating point number. A concrete syntax would worry about whether the decimal point can be omitted, or can start a number, or whether scientific notation is allowed. The following are the choices made by Pascal:

```
FPnumber ::= digit-sequence '.' fractional-part ['e' scale-factor]
          | digit-sequence 'e' scale-factor
fractional-part ::= digit-sequence
digit-sequence ::= digit { digit }
digit ::= 0|1|2|3|4|5|6|7|8|9
```

However, if we think semantically, a floating-point number is just a number. It differs only from an integer in having a fractional part, but each number in the set is a distinct object with two parts, a mantissa and an exponent (to use the IEEE formulation) or a whole number part and a fractional part. Either way, the syntax is much simpler:

```
Fpnumber ::= mantissa exponent | whole fraction
```

We call this an *abstract* syntax because we are not concerned with punctuation and character strings, but with structure, i.e. the order and type of the parts defined by the syntax rule. In fact, we can write an abstract syntax of a language without using terminal symbols at all. The non-terminals used correspond directly to semantically meaningful objects. Sometimes, however, we add back in some (unnecessary) symbols to make the syntax more readable by human beings:

```
Fpnumber ::= mantissa 'e' exponent | whole '.' fraction
```

Another way to look at the difference between concrete and abstract syntax is that the abstract form is the ‘real’ form, since it is semantically oriented; the concrete form is then an realization of it in character strings. This is why concrete syntax definitions can be ambiguous – we are linearizing a tree structure in a non-reversible way. The classic case is that of arithmetic expressions involving constants. An abstract definition is:

```
Exp ::= Num | Exp1 Op Exp2
Op ::= plus | minus | mult | div
Num ::= zero | one | ...
```

The reason this is not ambiguous is because it describes trees, not character strings. Thus we can use this grammar to describe the computation “add one to two and multiply the result by four” accurately, or we can use the same grammar to describe the computation “add two to the result of multiplying one by four”. However, the string approximation:  $2+1*4$  that follows from this grammar *is* ambiguous unless the concrete syntax makes a distinction in the precedence of operators, and adds parentheses – a much more complex grammar:

```
Exp ::= Exp LowOp Term | Term
Term ::= Term HighOp Factor | Factor
Factor ::= Num | ( Exp )
LowOp ::= + | -
HighOp ::= * | /
```