

SYNTAX EXAMPLES

We will present two complete examples of an abstract syntax definition. One is for a simple imperative language (with assignment as its main computational device) and one for a simple functional language (with no assignment).

AN IMPERATIVE LANGUAGE

First, the semantic categories. They are:

$P \in \text{Programs}$
 $C \in \text{Commands}$
 $E \in \text{Expressions}$
 $O \in \text{Operators}$
 $I \in \text{Identifiers}$
 $N \in \text{Numerals}$
 $B \in \text{Booleans}$

Next, the grammar:

```
P ::= C
C ::= nop | C1;C2 | I := E | if B then C1 else C2 end | while B do C end
E ::= N | I | E1 OA E2
OA ::= plus | minus | mult | divide
B ::= true | false | I OR E
OR ::= equal | greater | less
I ::= ... any identifier ...
N ::= zero | one | two ...
```

The first rule says that a program is simply a command. The second rule says that a command (sometimes called a statement) is either an assignment, a no-operation, or is one of the three basic structured programming forms, sequence (signified by the semicolon), conditional, signified by if-then-else or a while loop. The no-operation is included so that either the then or else part of the conditional can be no statement. The third rule, for expressions, says that the right hand side of an assignment can be a constant number (N), a simple identifier, I, or an arithmetic expression involving two subexpressions and one of the four arithmetic operators. The fifth and sixth rules handle boolean expressions for conditionals and loop exits. Again we should emphasize that the ‘noise words’ (colon-equals, semicolon, if, then, else, while, do, end) are not necessary for the abstract syntax, but help readability, as do the subscripts on E and C in rules where there are two similar non-terminals. This is, of course, only one way to set up such a language, but it is a complete language in that it can handle complex tasks in arithmetic. Additions like declarations, types and I/O are complications we can do without for now.

A sample program is:

```
x := three;
y := two;
while x greater zero do
  if y equal x then
    nop
  else
    x := x minus one
  end
end
```

You might have noticed that under the ‘normal’ semantics of these kinds of languages, that this program never terminates. It is not the job of abstract syntax to prevent this (actually it couldn’t anyway) but the job of a semantic definition to handle this case and others like it.

A FUNCTIONAL LANGUAGE

First, the semantic categories. They are:

$P \in$ Programs

$F \in$ Functions

$E \in$ Expressions

$I \in$ Identifiers

$N \in$ Numerals

Next, the grammar:

```
P ::= F1 F2 ... Fn E
F ::= I (I1, I2, ... In) = E
E ::= N | I | E1 O E2 | if E1 then E2 else E3 |
      let I = E1 in E2 | I (E1, E2, ... En)
O ::= plus | minus | mult | divide | equal | greater | less
```

The first rule says that a program consists of any number of function definitions and an expression to evaluate in the context of these definitions. The second says that a function contains three parts – a name (the first I), a number of parameters (the Is in parentheses and an expression which is the definition's body. The third rule is the one for expressions. An expression can be a number (constant), a simple identifier, a binary expression where the operator can be arithmetic or relational, a conditional, a let expression which introduces a new identifier and binds it to a value, or a function call in standard notation.

The intended semantics, which we shall of course, make plain later, are that the E in the program rule, calls one of the defined functions, passing a value or values to it, which in turn calls other functions. Each function, when it exits, returns a value which eventually is passed all the way out as the result of the program. Consider this program:

```
even(x) = if x then odd(x minus one) else one
odd(x) = if x then even(x minus one) else zero
even(four)
```

even(four) calls odd(three) which calls even(two) which calls odd(one) which calls even(zero) which returns one. In this language zero is used for false (as it is in C) and one for true.