# 5

# Abbreviation and abstraction

This chapter is largely about the business of taking parts of a program and giving names to them. We have got so far without saying what a name is, largely because the rules for which are allowable names are much what they are in most programming languages. To be precise, a name is a sequence of letters, digits and full-stop symbols; the first must be a letter. Two names are distinct unless they are exactly the same sequence of characters – the case of letters is significant – so each of the following twenty-four names

```
x           y          z          X          Y          Z
left        right      up         down       seq        par
x0          x1         x2         x.         x..        x...
Imre.Nagy              G.G.Arnold            d2V.by.dr2
Beatrice7              dff.39c               OX1.3DW
```

can validly be used as a name in an `occam` program. The only exception to this description is that those sequences of upper case letters which are keywords – like `SEQ`, `PAR` – are reserved for this purpose and cannot be used as names. There is a list of reserved words on page 206 in the language summary at the back of this book.

Formally, all the names used in a program have to be in the scope of a declaration of the right kind. In practice, an implementation will have libraries of pre-declared names which behave as though they had already been defined. For the present, these can be treated as though they were declared immediately before the text of every program.

Declarations of variables, protocols and channels, and abbreviations of values, variables and channels can appear in front of any process, any branch of a conditional, branch of an alternative, branch of a case discrimination, branch of a discriminated input, or anonymous function. The scope of the name introduced by the declaration or abbreviation is the process or branch, and any declarations or abbreviations appearing between them.

Very roughly, what this means is that everywhere after the declaration or abbreviation and before the end of the subsequent process or branch the new name refers to the declared or abbreviated object. There are two kinds of exception: either another declaration of the same name can knock a hole in the scope of the declaration, or there may be constraints which mean that although a name is in scope it is not valid to use it.

The first kind of exception follows a conventional scoping rule in all ALGOL-like languages. An example of this kind is that in

```
VAL INT x IS 27 :
SEQ
  c ! 1; x
  VAL BYTE x IS '$' :
  d ! 2; x
  c ! 3; x
```

because the $x$ in `d!2;x` is in the scope of the inner abbreviation, it is taken out of the scope of the outer one; the whole process is equivalent to

```
SEQ
  c ! 1; 27
  d ! 2; '$'
  c ! 3; 27
```

Notice that only the one branch of the sequence is in the scope of the inner abbreviation. Deliberate and unnecessary re-use of names like this is to be discouraged, since it is likely to confuse a human reader – and often the writer – of the program. It is valid to write

```
VAL x IS 63 :
VAL x IS 2 * x :
VAL x IS x + 1 :
c[x] ! x
```

but it would be altogether clearer to write

```
VAL x IS 63 :
VAL y IS 2 * x :
VAL z IS y + 1 :
c[z] ! z
```

which is what it means. On the other hand, it is as well that this scoping rule makes it safe to use a name locally irrespective of an unrelated use that happens to have an enclosing scope.

The second kind of exception is less familiar; in most programming languages, it is simply considered good practice never to have two names for the same variable. For example, it is possible but frowned upon to have a variable passed as an argument to a PASCAL procedure, and to use it as a free variable of that procedure. If there are two different names for a single piece of storage, it becomes difficult to predict the effect of assigning to one of them. An experienced programmer will know at a glance that

```
SEQ
  Casnewydd := Casnewydd >< Newport
  Newport   := Casnewydd >< Newport
  Casnewydd := Casnewydd >< Newport
```

has the effect of interchanging the values of the integer variables *Casnewydd* and *Newport*, admittedly rather less efficiently than

```
Casnewydd, Newport := Newport, Casnewydd
```

(It is an old trick for exchanging the contents of two processor registers without recourse to the use of a third, or of main store, as a temporary storage location.) Yet, there is an unstated assumption that they are different variables: were it allowable to write

```
Newport IS Casnewydd :
SEQ
  Casnewydd := Casnewydd >< Newport
  Newport   := Casnewydd >< Newport
  Casnewydd := Casnewydd >< Newport
```

then presumably this would be equivalent to

```
SEQ
  Casnewydd := Casnewydd >< Casnewydd
  Casnewydd := Casnewydd >< Casnewydd
  Casnewydd := Casnewydd >< Casnewydd
```

which sets *Casnewydd* to zero, three times. Since there might be any amount of text between the abbreviation and its use, it might be difficult for a reader to spot that *Casnewydd* and *Newport* shared the same location.

In occam such dangerous bad style is outlawed by the language definition; such programs are not valid occam, and an implementation of occam must check for violation of these rules. This compulsion is partly because the potential for confusion is enhanced by the possibility of concurrent accesses to shared variables. In part it reflects a growing acceptance that it is a good thing to restrict the programmer's freedom to make a fool of himself, so as to make it easier to check that a program means what it seems to mean.

The rule for channel and variable abbreviations is essentially that when a new name has been given to a variable or channel, use of the old name is no longer valid. In order to be realistically checkable without having to execute the program, the rule for abbreviating components and segments of arrays is that any use of the name of the array is invalid, except for further abbreviations; and that abbreviated parts of an array must be non-overlapping.

The rule for value abbreviations is that it is invalid in the scope of a value abbreviation to appear to change the value of a variable which appears in the abbreviated expression. This means that

```
VAL INT forty.two IS (zero * x) + one.three.seven :
IF
  forty.two = 42
    x := 17
  forty.two <> 42
    SKIP
```

would be invalid, even if *forty.two* were not 42. A variable appears to be changed if there is in the text an assignment to it, an input to it, or a variable abbreviation of it, even though that part of the text might happen not to be executed.

The rule for sharing variables between the branches of a parallel process is essentially that either none of them appears to change it, or exactly one of them can change it and none of the others can access it at all. It might help to remember this rule if you think of it as: either each of the branches can safely take a copy of the initial value of the variable, and it does not change; or one of the branches can take sole possession of the storage represented by the variable. The rule for channels is similar: at most one branch of a parallel process can use a channel for input, at most one of them can use it for output.

Again, to make the disjointness of the branches of a parallel process efficiently checkable by a compiler, there is an additional rule for arrays of variables or channels. If a branch of a parallel process indexes an array with subscripting expressions that include variables, that branch is treated as if it used the whole of the array. Of course, it is valid to abbreviate several disjoint segments of an array and to index those with variable subscripts in different branches of a parallel process.

## Procedure definition

A *procedure* definition gives a name to a process, so that in the scope of the definition, the name can be written to have the effect of having written out the whole of the process. In the scope of

```
PROC another()
  number := number + 1
:
```

you can use the *call*

```
another()
```

to stand for the process in the body of the procedure

```
number := number + 1
```

As with other definitions, the scope of a procedure definition is the part of the program *after* the definition, so does not include the process in the definition itself. In occam, procedures cannot call themselves.

A word of caution about names: in the rather bizarre program

```
INT number :
PROC another()
  number := number + 1
:
SEQ
  number := 0
  INT number :
  SEQ
    another()
    number := 0
  c ! number
```

the *number* assigned to by *another*() is the one in scope where the procedure is defined, not the one where it is used. The example is the same as

```
INT number :
SEQ
  number := 0
  INT new.number :
  SEQ
    number := number + 1
    new.number := 0
  c ! number
```

The reason for the empty parentheses after the name of *another* is that a procedure definition need not give a name to a simple process. The procedure can be made to depend on abbreviations which are different at each call of the procedure.

```
PROC write.string(CHAN OF BYTE c, VAL []BYTE s)
  SEQ i = 0 FOR SIZE s
    c ! s[i]
:
```

is a procedure which sends an as yet unspecified sequence of bytes along an as yet unspecified character channel. An instance like

```
write.string(output, "Bootifrolo")
```

is effectively

```
CHAN OF BYTE c IS output :
[]BYTE s IS "Bootifrolo" :
SEQ i = 0 FOR SIZE s
  c ! s[i]
```

The formal parameter list in a procedure definition consists of the left hand sides of a sequence of abbreviations, and the actual parameter list in an instance consists of the corresponding right hand sides.

There must be the same number of components in each list of actual parameters as in the formal parameter list, and they must be the right sort of expression, variable, or channel, and of the right type for the equivalent abbreviation to be correct. Since the whole point of procedure definition is to take a piece of the program text and make it self-contained, you cannot leave the type specification from the formal parameter list as you can from an actual abbreviation. On the other hand, where there is a sequence of parameters of the same type and kind, you can leave out the second and subsequent specifiers:

```
PROC set.min.max(INT min, max, VAL INT a, b)
  IF
    a <= b
      min, max := a, b
    a >= b
      max, min := a, b
:
```

is exactly the same definition as if it had been introduced by

```
PROC set.min.max(INT min, INT max, VAL INT a, VAL INT b)
```

Since the meaning of parameters is defined in terms of abbreviation, they are governed by the same rules as abbreviation: a variable may not be passed twice as two variable parameters, nor a channel twice as two channel parameters; neither should a variable which appears in an expression which is passed as a value parameter also be passed as a variable parameter. An instance like

```
set.min.max(a, b, a, b)
```

is not valid, because the argument list abbreviates the variables $a$ and $b$ before they appear as value arguments. This is a problem which cannot be cured by changing the order of the arguments:

```
PROC set.min.max(VAL INT a, b, INT min, max)
  ...
:
set.min.max(a, b, a, b)
```

would be invalid because $a$ and $b$ are abbreviated after they appear in value abbreviations.

In addition, of course, the parameter abbreviations must not interfere with free names – those names like *free* in

```
PROC remember(VAL INT arg)
  free := free >< arg
:
```

which appear in a procedure definition and are not defined within it, but refer to definitions in an enclosing scope. An invocation like *remember*(137) is all right, but *remember*(*free*) would have been invalid; to be precise, in

```
INT free :
PROC remember(VAL INT arg)
  free := free >< arg
:
SEQ
  ...
  INT expensive :
  SEQ
    ...
    remember(expensive)   -- valid
  remember(free)        -- invalid
  INT nothing IS free :
  SEQ
    ...
    remember(something) -- invalid
  INT free :
  SEQ
    ...
    remember(free)      -- valid
```

the first instance is all right; the second is invalid because it means

```
INT free :
...
VAL INT arg IS free :
free := free >< arg
```

which assigns to a variable mentioned in an abbreviated expression; the third is invalid because it means

```
INT free :
...
INT nothing IS free :
...
free := free >< something
```

which assigns to an abbreviated variable; and the fourth is valid because it is the same as

```
INT free :
...
INT new.free :
...
VAL INT arg IS new.free :
free := free >< arg
```

because the apparent clash of names is entirely accidental. Notice that – as in all ALGOL-like languages – the assignment is always to the same variable irrespective of what names are in scope at the instance.

Similarly the procedure

```
PROC invalid(CHAN OF INT channel)
  PAR
    channel ! 0
    INT dummy :
    channel ? dummy
:
```

is invalid even though the channel is used for input in the one case and output in the other. This is because of an additional rule that a channel argument or free channel of a procedure may not be used both for input and for output. Of course, this causes no great hardship to the programmer, since such a channel might as well have been declared locally to the body of the procedure.

## Anonymous functions

An anonymous function contains a process which calculates a value; for example

```
INT sum :
VALOF
  SEQ
    sum := 0
    SEQ i = 0 FOR SIZE v
      sum := sum + v[i]
  RESULT sum
```

is an `INT` expression – its value is the value of the variable *sum*, which appears as its `RESULT`, after the execution of the process which is its body:

```
SEQ
  sum := 0
  SEQ i = 0 FOR SIZE v
    sum := sum + v[i]
```

and so is the sum of all the components of the array $v$. (In INMOS documentation, anonymous functions are called *value processes*, but notice that they are expressions, and not processes.)

In order for an anonymous function to be side-effect free it is forbidden for its body to assign to free variables. This means that in practice, the result will almost always consist of variables declared immediately in front of the `VALOF`. Additionally, the body may not input or output, and must contain no parallel or alternative processes. This means that the value of the expression is fully determined by the values of its free variables.

When an anonymous function appears in an expression, it must be enclosed in a pair of parentheses one above the other, and the whole of its text must be to the right of these parentheses.

```
sum.of.sums := (INT sum :
                 VALOF
                   SEQ
                     sum := 0
                     SEQ i = 0 FOR SIZE v
                       sum := sum + v[i]
                   RESULT sum
               ) + sum.of.sums
```

If the process in the body of an anonymous function fails, for example if it stops, then the expression using the anonymous function is just like any other invalid expression: a process which uses it stops.

As well as returning a single value for use in an expression, an anonymous function may also return a list of values, for example

```
INT min, max :
VALOF
  IF
    a <= b
      min, max := a, b
    a >= b
      max, min := a, b
  RESULT min, max
```

which returns two `INT`s. These functions can be used on the right hand side of a multiple assignment

```
less, more := (INT min, max :
               VALOF
                 IF
                   a <= b
                     min, max := a, b
                   a >= b
                     max, min := a, b
                 RESULT min, max
              )
```

again, between and to the right of vertically disposed parentheses. In practice, anonymous functions will rarely be written in expressions and assignments: they are usually only used as the bodies of function definitions. You can expect some compilers not to implement anonymous functions in expressions.

### Function definitions

Function definition generalizes value abstraction, in a way like procedure definition. For example, in the scope of

```
REAL64 FUNCTION square(VAL REAL64 a) IS a * a :
```

you can write

```
r.square := square(x) + square(y)
```

to mean

```
VAL REAL64 a IS x :
VAL REAL64 b IS y :
r.square := (a * a) + (b * b)
```

or, rather, something more like

```
r.square := (VAL REAL64 a IS x :
             VALOF
               SKIP
               RESULT a * a
            ) + (VAL REAL64 a IS y :
                 VALOF
                   SKIP
                   RESULT a * a
                )
```

The formal parameters of a function definition must all be value parameters, and the value of an instance of the function is the value of the expression in the body of its definition, in the scope of the value abbreviations corresponding to its parameters.

Notice that there is an important difference between

```
VAL INT initial.x IS x :
```

and the definition of a 'constant' function like

```
INT FUNCTION recent.x() IS x :
```

In the scope of a value abbreviation it is invalid to try to change the value of the abbreviated expression; the value of the abbreviating name is therefore fixed. Each instance of a constant function is evaluated afresh:

```
INT FUNCTION recent.x() IS x :
SEQ
  x := 42
  c ! recent.x()
  x := 137
  c ! recent.x()
```

has the same meaning as

```
SEQ
  x := 42
  c ! 42
  x := 137
  c ! 137
```

As a syntactic concession, if the body of a function definition is an anonymous function, it can be written to look as though it were a procedure definition,

```
INT FUNCTION max(VAL INT a, b)
  INT max :
  VALOF
    IF
      a <= b
        max := b
      a >= b
        max := a
    RESULT max
:
```

Remember that the process in the body of such a function definition, being the body of an anonymous function, cannot assign to free variables, communicate, or make arbitrary choices. This ensures that the defined function is free from side-effects.

More generally, a function can return a list of values, for example

```
INT64, INT64 FUNCTION divide (VAL INT64 x, y) IS x/y, x\y :
```

and instances of such functions can be used as the right hand side of a multiple assignment or as the `RESULT` of an anonymous function.

```
x, y := ( INT64 p, q :
          VALOF
            p, q := divide(x, y)
            RESULT divide(p, y)
        )
```

Notice that calling a function which returns multiple results can be more efficient than calling several functions. The assignment

```
INT, INT FUNCTION min.max(VAL INT a, b)
  INT min, max :
  VALOF
    IF
      a <= b
        min, max := a, b
      a >= b
        max, min := a, b
    RESULT min, max
:
a, b := min.max(a, b)
```

makes only one comparison of $a$ against $b$, whereas

```
a, b := min(a, b), max(a, b)
```

necessarily makes two comparisons.

Since a function with multiple results can ultimately only be used on the right hand side of a multiple assignment, it might seem that you would be as well to have written a procedure with variable parameters. For example

```
x, y := min.max(a, b)
```

being the same as

```
set.min.max(x, y, a, b)
```

Notice, however, that

```
a, b := min.max(a, b)
```

is valid, whereas

```
set.min.max(a, b, a, b)
```

is not.