1

An introduction to occam

The occam notation was designed to simplify the programming of parallel computers. Arranging that several things happen 'at the same time' in a program is as simple in occam as is arranging that they happen in any particular order. The language was designed to be used as a low-level programming language and as a compiler target for machines built of INMOS transputers.

The language, occam 2, described and used in this book is a development of an earlier occam language, variously called 'proto-occam' or just 'occam'. It provides facilities for describing and checking the types of data items used and communicated by a program, and for structuring the communications between processes. It has also a richer set of program structures.

There is an unfortunate confusion of different languages all called occam. Where it is necessary to distinguish, this book will refer to the earlier language, that of the *Programming Manual*[4], as proto-occam. At the time of writing, the occam compilers available for transputer-based machines accept a language which is neither proto-occam, nor occam 2 as described by the *Reference Manual*[1]. The language used in this book is, as far as possible, occam 2 as described in the *Reference Manual*.

An occam 2 program is a process, that is, it is a description of some actions possibly with a description of the data on which those actions are performed. Indeed, the principal difference between proto-occam and occam 2 is that there is in the newer language a richer sub-language for describing the types and structures of the data on which a program operates. Where there are new program structures in occam 2, these reflect the new data types and structures.

Accordingly, this discussion of occam 2 takes the following sequence: the data and the types of data acted on by an occam program; the channels and the protocols on channels by which programs communicate; the processes and structures of processes from which programs are built. Because it might be a bit confusing (and rather tedious) to come at things in this order, the rest of this chapter is an outline of the key peculiarities of occam. All this will come around again in its proper order, but is here so that the next chapters will make sense in the order in which they appear.

First of all, just as in other procedural languages, there are assignments in occam; the assignment

```
x := y
```

changes the value of x to be the same as that of y. Conventional, sequential programs can be written, although the syntax is a bit unusual: what might, in another language like PASCAL or ALGOL, have been written

```
BEGIN x := y; y := 3 * x + 2; z := x + y + 1 END
```

is written in occam as

```
SEQ
  x := y
  y := (3 * x) + 2
  z := (x + y) + 1
```

This means that the value of y should be stored in x; that the new value of x is used to calculate $3x + 2$, and the value of that expression stored in y; and only the new values of x and y be used to store a new value in z.

The format is rigidly fixed: each process (in this case, each assignment) starts on a line to itself, and in a column indented a little (two columns) from the start of the keyword `SEQ` (for *seq*uential).

As a concession, if a line gets too long for comfort, it can be broken in most of the places where the first part cannot be confused with a complete line, provided that the continuation is indented at least as far as the first part of the line. Comments are introduced by a pair of dashes, `--`, and extend to the end of the line. So as to avoid making the layout obscure, a comment must not start to the left of the next line of program.

```
SEQ             -- this is a keyword
  x := y
  -- notice that there is no precedence
  -- between the operators in expressions

  y := (3 * x) +
        2       -- this is a continuation

  -- notice that + is not associative
  z := (x + y) + 1
```

Expressions have to be written with enough parentheses to make the evaluation order explicit. There is no precedence between, say, multiplication and addition; addition is not associative, because there might be an arithmetic overflow in one order of evaluation, but not in another.

The reason for the fuss about `SEQ` introducing a sequentially executed list of processes is that there are also concurrent, or *par*allel, lists of processes. The notation is similar

```
PAR
  x := y + 1
  z := y + 2
```

This process executes by executing both of its components, but it is not defined which of them happens first. Indeed they are allowed to happen 'at the same time'. A process like this is only legal if the components are *disjoint*: none of them changes a variable which appears in one of the others.

Branches of a parallel process can communicate by input (written with a `?` mark) and output (written with an `!` mark) over a common channel

```
PAR
  c ! x + y
  c ? z
```

which executes so as to have the same effect as `z:=x+y`.

Parallel and sequential constructions can be nested without any limitation,

```
SEQ
  PAR
    p1
    SEQ
      p2.1
      SEQ
        p2.2.1
        p2.2.2
      p2.3
    p3
  q
  SEQ
    r1
    PAR
    r3
```

Notice that the last of those `PAR` keywords, with nothing indented after it, is like a pair of brackets with nothing between them: it has an empty list of processes. To execute such an empty parallel is to do no-things, all at once. In the same way, a sequential list of no processes does nothing, and is immediately completed. The same effect can be achieved more clearly by writing `SKIP`. This is a process which does nothing, and then terminates.

Conditional processes make decisions about which of a collection of processes will be executed. What might, in other notations, have been written

```
IF odd(x) THEN x := 3 * x + 1 ELSE x := x / 2
```

is written in occam as

```
IF
  odd(x)
    x := (3 * x) + 1
  NOT odd(x)
    x := x / 2
```

A conditional process contains a list of pairs of expressions and processes, each of the expressions being either true or false. It executes by executing the first of the processes paired with a true expression.

It would be wrong for there not to be a true expression, and in that case none of the components can be executed, but neither can the conditional terminate. An invalid conditional like this, and in particular an `IF` with no components, does nothing but it is not like `SKIP` because it does not terminate successfully. This turns out also to be such a useful process that it has a name: `STOP`.

The other thing that a programmer must be able to specify is that some piece of program be executed repeatedly. The notation in occam for what is written

```
WHILE even(x) DO x := x / 2
```

in other languages is

```
WHILE even(x)
  x := x / 2
```

and it means the same as

```
IF
  even(x)
    SEQ
      x := x / 2
      WHILE even(x)
        x := x / 2
  NOT even(x)
    SKIP
```

Something to notice about loops that do not terminate, like for example

```
WHILE TRUE
  SKIP
```

is that although this is a process which does nothing, it is not the same as SKIP because it never terminates; on the other hand, it is not the same as STOP, because you can expect an implementation of occam to do something intelligent with STOP.

Do not confuse STOP with a non-terminating loop: STOP is used to explain what is meant to happen in an occam program that goes wrong; it is a potentially detectable failure. For example, a process that divides by zero, or which accesses an array outside its bounds, is meant to 'behave like STOP'. A non-terminating loop, on the other hand, can carry on not terminating for a long time without an implementation of the language having (or indeed being able) to notice.