1 An introduction to occam

As far as will concern us here, an *occam* program is simply a process, which may have some free identifiers that have specific meanings dependent on the computer on which the program is to be run. A process describes some actions that are to be performed: that is, it is the expression of an algorithm. Each process may be either a primitive process, or a composite process consisting of a number of definitions and simpler component processes bound together by process constructors. The structure of constructed processes is indicated by a fixed layout of the source text, with each component appearing on a new line, slightly indented from the keyword that introduced the whole construction.

Processes that do nothing

The simplest of the primitive processes is `SKIP`, which is the process that does nothing at all. In many programming languages, you are obliged to write nothing (that is, not to write anything at all) if you want 'nothing' done; you will see later that `SKIP` serves as useful a purpose in *occam* as that of zero in the decimal notation for numbers.

The process `STOP` also does 'nothing', but unlike `SKIP` it fails to terminate. You can think of it as being what happens when something goes wrong, like a deadlock, or some illegal machine operation. Nothing can happen in a sequential process after it has stopped, but things can happen in parallel with a stopped process. You might not expect to write the `STOP` process very often in your programs, but it is the rational thing to do when something unexpected happens, because it ensures that the part of a network of processes that has failed is brought to a standstill without affecting other processes, at least until they come to depend on the broken part.

It is also useful to have `STOP` around so as to be able to describe the effect of compound processes that 'go wrong', for example by becoming deadlocked. A process is said to be deadlocked if there is nothing which it is able to do next, but it has not finished properly. A parallel program which becomes deadlocked typically does so because each of its processes is waiting for one of the others to do something.

Sequential processes

In programs which execute sequentially, the work is done by assigning values to variables, and subsequently basing decisions on the values of those variables. The *occam* assignment has the form

```
variable := expression
```

Each expression has a value, as explained later, which is just a bit pattern of the size of a 'word' on your computer, and every variable is capable of storing a word sized bit pattern. Of course, there are operators which treat these bit patterns as if they represented numbers, or truth values, or characters, but no type distinction is enforced by the language or its compilers.

A sequence of operations is described by writing them one under the other, under and slightly indented from the keyword `SEQ`. The sequence executes by executing each of its components in the order in which they are written. Of course, if there are no components at all, then the sequence does nothing, and behaves just like `SKIP`. Thus

```
SEQ
  x := 3
  y := x + 7
  x := x + 6
  z := x + 1
  x := (y + z) / 2
```

has the overall effect of setting each of the variables x, y, z to 10, albeit in a somewhat perverse way: first x is set to three, then y to seven more than x, then x is changed to nine, z is set to one more than it, and finally x is set to the average of y and z, which is of course also ten.

Decisions based on the values of variables are made by a conditional process. This construction consists of the keyword `IF` written above a list of components, each slightly indented. Each component is either another conditional nested within the first, or consists of an expression (the condition) and, below the condition and a little further indented, a process. The whole conditional executes by looking down the list of components, and the components of nested conditionals, until a condition is found whose value is `TRUE`. If one is found then the corresponding process and only that process is executed, and the whole conditional terminates. It is an error for no condition to evaluate to `TRUE`, for example if there are no components, so in that case the conditional stops.

```
IF
  n < 0
    sign := -1
  n = 0
    sign := 0
  n > 0
    sign := 1
```

sets the value of the variable *sign* to one of minus one, zero, or plus one, according as the variable n has a negative, zero, or positive value.

Since it is defined that the textually first of the processes corresponding to a `TRUE` condition is selected, the process

```
IF
  n = 0
    sign := 0
  n <= 0
    sign := -1
  TRUE
    sign := 1
```

describes exactly the same effect as the former, but is much less clear. In general, it is good style to use constraints that describe as precisely as is convenient the conditions under which a process is to be executed.

Parallel processes

Just as a list of actions can be described as happening in a strict sequence, so in *occam* it is possible to specify that each of a list of actions is to happen, without specifying an order in which they must happen. Such a parallel composition is indicated by writing the actions one under the other, under and slightly indented from the keyword PAR. The parallel composition executes by executing each of its components, until each has terminated. Of course, if there are no components, the effect is the same as SKIP, and if any of the components fails to terminate, for example by stopping, then the composition cannot terminate.

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

sets the values of x and z to be one less than, and one more than, respectively, the value of y. There is, however, no guarantee that the assignments will not happen in the other order, or at precisely the same moment. Because of this, for a parallel composition to be legal, none of its components may change any variable which is used in any of the other branches: it would be wrong, for example, to try to write

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

because x is used in the second component, but changed in the first component.

If such mutual interference is not allowed, how then are concurrent processes to communicate? The answer is that they do so by input and output over channels. The output process

```
channel ! expression
```

sends the value of the expression over the channel. Similarly, the input process

```
channel ? variable
```

receives a value from the channel and stores it in the variable. Each of these communications waits for the other, so that an output does not happen until the corresponding input happens, and vice versa.

There is an abbreviation which is useful when complex data are to be transmitted, for example where many streams of data are multiplexed along one channel, and some identification must be sent with each item. A sequence of inputs (or of outputs) along one channel may be written as a single primitive process, separating the target variables (or expressions) by semicolons

```
channel ? variable.1; variable.2; ... variable.n
```

This has precisely the same meaning as

```
SEQ
  channel ? variable.1
  channel ? variable.2
          :
  channel ? variable.n
```

In combination, an input and an output behave just like an assignment, except that the expression and variable are in different, concurrently executing processes. In particular, the assignment

```
variable := expression
```

is exactly the same as

```
PAR
  channel ! expression
  channel ? variable
```

provided that it is legal to write the latter in the particular context. Just as there are rules about the use of variables in concurrently executing processes, so also each end (the input end, or the output end) of a channel may be used in only one of the components of a `PAR` construction.

Decisions may also be distributed across several processes, using an alternative, which is similar to a conditional except in that the choice can depend on whether another process is performing an output. An alternative is written with the keyword `ALT` above a list of components, each of which is either an alternative, or consists of a guard with below that a process which is indented a little further. A guard may be an input process, or `SKIP`, or either of these simple guards preceded by an expression and an ampersand sign, as for example

```
ALT
  red.selected & red ? x
    out ! x
  green.selected & green ? x
    out ! x
  NOT (red.selected OR green.selected) & SKIP
    out ! default.value
```

An alternative waits until there is a guard which is 'ready'. An input guard becomes ready when a corresponding output is possible; a `SKIP` guard is always ready; and a guard preceded by an expression can only become ready if both the value of the expression is `TRUE`, and the process part of the guard is ready. When some guard has become ready, one of the ready guards is selected and executed, followed by the corresponding process. After this the alternative terminates. At most one of the branches of an alternative is selected; if no guard ever becomes ready (for example, if there are no components in the alternative) then the whole alternative is deadlocked, like `STOP`.

The example alternative above will accept an input from either the *red* or the *green* channel, provided that the value of the corresponding variable, *red.selected* or *green.selected*, is `TRUE`, and having accepted that input into the variable x, it then re-outputs the value down the channel *out*. In case neither variable is `TRUE`, only the third guard is ready, so the alternative sends the *default.value* and then terminates.

Data declarations

Each name used in an *occam* program must be declared before it can be used. There are declarations which allow you to give names to constant values, to variables, and to channels.

Constant definitions do not introduce any new objects into the program, they serve merely to give names to particular values. They are written with the keyword `DEF` followed by some definitions of the form

```
name = constant.expression
```

each definition separated from the next by a comma, and with a colon at the end. The effect is to allow you to use the *name* in the process that follows, wherever you want to refer to the value of the *constant.expression*. As an example, the declarations

```
DEF red = 0, red.and.amber = 1, green = 2, amber = 3 :
DEF first.state = red, last.state = amber :
```

might appear in a traffic light controller. It would then be possible to test, for example, whether

```
current.state = last.state
```

rather than comparing *current.state* for equality with 3, which would be altogether more obscure.

Variables, as has already been indicated, are capable of storing bit patterns of some fixed size. They are declared by listing their names after the keyword VAR, above the process that will use them. The names are separated by commas, and there is a colon at the end of the list. The traffic light controller might well include, for example,

```
VAR current.state, queue.size :
```

Initially, a variable has no defined value, but once it has been set by an assignment or an input, its value is the last value that was stored into it. The usual place to find variable declarations is accordingly just before the keyword SEQ. Components of a sequence communicate with each other by one component leaving a value in a variable, and a later component reading that value. The traffic light controller might begin

```
DEF first.state = red, last.state = amber :
VAR current.state, queue.size :
SEQ
  current.state := first.state
  queue.size := 0
        ⋮
```

Channels are declared in the same way as variables, except that the keyword is CHAN. Somewhere within the scope of each channel declaration there will be two concurrent processes, one of which sends output to the channel, the other of which takes input from it. It is therefore usual to find channel declarations immediately in front of PAR constructions.

Arrays

There is only one device for structuring data in *occam*: you may make one-dimensional arrays of constants, variables or channels. An array of variables is declared by giving a constant expression in brackets after the name.

```
name[count]
```

This indicates that a number, equal to the value of the *count*, of variables are to be declared, which can be referred to by indexing the name of the array with expressions whose values range from zero up to $count - 1$. For example, in the scope of the definition

```
VAR a[137] :
```

there are declared 137 variables, called $a[0]$, $a[1]$, $a[2]$, $\ldots$, $a[136]$, each variable independent of the others. Arrays of channels are similar.

An array of constants is called a table, and is denoted by an expression of the form

```
TABLE[expression.zero, expression.one,
                    expression.two, ... , expression.n]
```

(This is an example of a broken line: long lines of *occam* can be can be split in places where it is obvious that they are not yet complete, so long as the second part of the line does not start to the left of the first part. Be warned that the rules will probably not agree with you about which are the places at which a line is obviously incomplete!)

Table expressions can be used in `DEF` declarations, or can appear to the left of an index, so that the value of

```
TABLE[expression.zero, expression.one,
                              expression.two, ... ][2]
```

is the same as the value of *expression.two*, for example.

In case the values of the components are always going to be in the range zero to 255, as for example when the values represent characters, *occam* allows you to specify that the values are to be packed one to a byte in the store of the machine. An array of byte variables is declared by adding the keyword `BYTE` after the opening bracket of the declarations; a byte table is denoted by including the keyword `BYTE` after the opening bracket of the denotation. A byte array of constants or variables is indexed by putting the keyword `BYTE` after the opening bracket of the index. For the present, the only reason for being concerned with byte arrays is that *occam* has a convenient denotation for byte arrays of characters: a string, which is written as a sequence of characters enclosed in double quotes,

```
"This is a string"
```

is a byte array, the first (zeroth) byte of which contains the length, and the subsequent bytes contain the characters in sequence. This string represents the same array of constants as does

```
TABLE[BYTE 16, 'T', 'h', 'i', 's','*s', 'i', 's','*s',
               'a','*s', 's', 't', 'r', 'i', 'n', 'g']
```

A character between single quotes denotes a constant bit-pattern just like any other, corresponding to the ASCII code for that character, and the sequence `'*s'` represents the space character, that is 32. I could have written this as a space between quotes, `' '`, but the asterisk form is clearer. There are asterisk-sequences for space (`*s`), carriage-return (`*c`), the newline character (`*n`), the quote characters (`*'` and `*"`), and of course for asterisk (`**`), which can be used either in character (single) quotes, or as elements of a string. In the scope of the declaration

```
DEF s = "This*sis*sa*sstring" :
```

(which is yet again the same byte array as before) the value of `s[BYTE 0]` is 16, and the value of `s[BYTE 16]` is 103, which is the ASCII code of the character '*g*'.

Process declarations

Names may be given to whole processes by means of `PROC` declarations. These are introduced by a line of the form

```
PROC name =
```

which is followed by the process to be called *name*, indented slightly, and terminated by a colon. The effect is that anywhere in the process that follows the declaration you can write *name* to mean the whole of the named process. (That means the named process is not in the scope of the declaration, so it is not possible to invoke it from within itself.)

A named process may have a list of formal parameters included after its name in the declaration. The nature of each parameter is indicated by one of the keywords `VALUE`, `VAR`, or `CHAN`, meaning a value (run-time constant), variable, or channel. You may omit the keyword in front of a second or subsequent parameter of a particular kind. Arrays are indicated by writing an empty pair of brackets after the name of the array.

The process is invoked by putting a corresponding list of actual parameters after the name at the point of call. The effect is the same as if the body of the named process had been written in place of the call, but with the actual parameters substituted for the formal parameters. Thus, for example

```
PROC assign.character(VAR x, VALUE s[], i) =
  x := s[BYTE i]        :

VAR ch :
SEQ
  assign.character(ch, "This is a string", 13)
        ⋮
```

has the effect of assigning the code of '*r*', that is the 13^{th} character of the string, to the variable *ch*, exactly as though the program had been written

```
VAR ch :
SEQ
  ch := "This is a string" [13]
      ⋮
```

Loops and arrays of processes

There are two kinds of loops in *occam*: unbounded `WHILE` loops, and indexed, bounded `FOR` loops. Unbounded loops are necessarily sequential in *occam*, but there are many forms of `FOR` loop representing different sorts of regular activity.

An unbounded loop is written with the keyword `WHILE`, followed by an expression (the condition), with a process (the body) below and slightly indented from it. It executes by testing the value of the condition and then, provided that its value is `TRUE`, executing the body. When the body has terminated, the condition is re-tested, so that the body is executed a number of times, in sequence, for as long as the condition remains `TRUE`. The whole `WHILE` loop terminates when the condition is tested and found to be `FALSE`.

```
SEQ
  x := 0
  v[n] := key
  WHILE  v[x] <> key
    x := x + 1
```

sets x to be the index of the first variable in the array v which contains the value *key*, by first posting a sentinel at $v[n]$. ('`<>`' is the *occam* notation for 'not equal'.)

A bounded loop may be thought of as being an array of processes. Loops can be made from each of the `ALT`, `IF`, `PAR` and `SEQ` constructions, by putting a replicator of the form

```
name = [base FOR count]
```

after the keyword, and then writing a single component (of the kind appropriate to the construction) below and slightly indented from the keyword. The *base* and *count* are expressions, and the meaning of such a `FOR` loop is the same as that of a construction formed from the same keyword followed by *count* copies of the component with the name taking on the values $base$, $base+1$, $\ldots$, $base+count-1$ in successive copies.

A `FOR` loop stands for a repetition of the constructor with which it is made. In the same way that

$$\prod_{year=1280}^{1341} entity_{year}$$

stands for the multiplication of sixty-two values,

$$entity_{1280} \times entity_{1281} \times \ldots \times entity_{1341}$$

so too the `SEQ-FOR` loop

```
SEQ year = [1280 FOR 62]
  celebrate.Christmas(year)
```

stands for the sequential composition of sixty-two processes

```
SEQ
  celebrate.Christmas(1280)
  celebrate.Christmas(1281)
             ⋮
  celebrate.Christmas(1341)
```

Thus, `SEQ-FOR` loops are just like `FOR` loops in languages like ALGOL or PASCAL; but that you may not assign to the loop index, and it is not declared outside the body of the loop.

The bodies of parallel `PAR-FOR` loops are executed concurrently, so such loops behave like arrays of parallel processes. A conditional loop, written with `IF` and `FOR`, performs a bounded search, so

```
SEQ
  v[n] := key
  IF i = [0 FOR n + 1]
    v[i] = key
      x := i
```

has precisely the same effect as that of the `WHILE` loop above. The same search can be done, without the use of a sentinel, by writing

```
DEF otherwise = TRUE, not.found = n :
IF
  IF i = [0 FOR n]
    v[i] = key
      x := i
  otherwise
    x := not.found
```

Here, x is set to *not.found* precisely when there is no occurrence of the value *key* in the array. You should now see the reason for allowing conditionals as components of conditionals, and alternatives as components of alternatives!

Some dialects of *occam* allow you to construct `FOR` loops not only from constructed processes, but also from the primitive assignments, inputs and outputs. An expression like

```
name [base FOR count]
```

is called a slice, and denotes all of the variables (or values)

```
name[base], name[base+1], ..., name[base+count-1]
```

Slices may be assigned

```
dest[dest.base FOR count]:= srce[srce.base FOR count]
```

or they may be communicated

```
PAR
  channel ? dest[dest.base FOR count]
  channel ! srce[srce.base FOR count]
```

In each case, both of the slices concerned must be of the same size, and the effect is to set each of the *count* variables in *dest*, from *dest*[*dest.base*] upwards, to the values of the *count* variables in *srce*, again, counting from *srce*[*srce.base*] upwards. Do not confuse slice communications with the semicolon abbreviation for a sequence of communications: single slice operations must match with other slice operations; the semicolon denotes a sequence of unrelated communications.

There are also byte slices which are denoted by

```
name [BYTE base FOR count]
```

and which may be assigned or communicated into other byte slices.

Expressions

Every simple expression in *occam* denotes a bit pattern the size of the word on the computer that is executing the program. There is no defined precedence between the various operators, so parentheses are generally needed to disambiguate an expression with more than one operator. The only exception to this rule is that associative operators do not need parentheses. All expressions are built of operators, constants, variables, and indexed arrays, so that evaluating an expression cannot possibly have a side-effect.

A number of operators are best understood by regarding each bit pattern as the twos complement representation of an integer: the values of

```
a + b        a - b        a * b        a / b        a \ b
```

are the sum, difference, product, quotient and remainder of a and b. The result of a division is an integer, rounded towards zero, and the remainder is given by

```
(a \ b)      =      (a - (b * (a / b)))
```

provided b is not zero.

The usual six relational operators:

```
a < b    a <= b    a = b    a >= b    a > b    a <> b
```

compare their operands as though they represented signed integers, and return one of the values `TRUE` or `FALSE`. There is one other relational operator intended for comparing the values of a clock which counts cyclically through all possible word sized bit-patterns: the expression

```
a AFTER b
```

is `TRUE` or `FALSE` according as a would be reached sooner by successively incrementing b, ignoring overflow, than by decrementing it. (Almost half of all bit patterns are `AFTER` any given bit-pattern.)

Expressions made up with the following operators have Boolean values.

```
NOT a
```

is `TRUE` when a is `FALSE` and `FALSE` when a is `TRUE`. Provided that each of a and b is either `TRUE` or `FALSE`, the values of

```
a AND b          a OR b          a >< b
```

are `TRUE` if both of a and b are `TRUE`, if either one is `TRUE`, or if exactly one is `TRUE`, repectively; they are `FALSE` otherwise. The `AND` and `OR` operators evaluate their left argument first, and then the right argument only should it be needed to decide the result. This means that, for example, bounds checks should precede array accesses

```
(0 <= i) AND (i < size) AND (a[i] = x)
```

The effects of some operators are most easily described by thinking of the operands as bit patterns. The values of

```
a \/ b          a /\ b          a >< b
```

are the bit by bit 'logical or', 'logical and', and 'exclusive or' of the patterns a and b.

```
NOT a
```

is the bit by bit (ones) complement of a.

Since the values `TRUE` and `FALSE` are, respectively, a word of one-bits and a word of zero-bits

```
(x /\ TRUE)  =  x
(x /\ FALSE) =  0
(x \/ 0)     =  x
```

This means that you can write conditional expressions, such as

```
(a /\ (p < q)) \/ (b /\ (p = q)) \/ (c /\ (p > q))
```

the value of which is one of a, b, or c according as p is less than, equal to, or greater than q.

The shift operators displace a pattern a by b number of bits, so that this many bits are lost from one end of the pattern, and the same number of zero bits are shifted in at the other end of the pattern. The values of

```
a << b          a >> b
```

are the same as the pattern a but shifted left and right respectively by b bit positions.

There is nothing to stop you constructing bit patterns by bit manipulation, and then treating these patterns as twos complement integers. Provided that *logarithm* is less than the number of bits in a word, the value of

```
1 << logarithm
```

is a word with only the *logarithm*$^{\mathrm{th}}$ bit set, that is $2^{logarithm}$. The value of

```
(NOT 0) >> 1
```

is a word of one bits, excepting that the most significant bit, that is the sign bit, is zero; this value is therefore the largest positive integer in the range that can be represented on your particular machine, written in a way that is independent of the word length of the machine.

```
NOT ((NOT 0) >> 1)
```

is similarly a word of zero bits, excepting for a one in the sign bit, so is the most negative integer that can be represented. Another construction which I will use later is

```
NOT ((NOT 0) << logarithm)
```

which is the bit pattern in which the least significant *logarithm* number of bits are set. This is useful for making masks, so that, for example, in the scope of

```
DEF control = NOT ((NOT 0) << 5) :
```

the value of

```
control /\ 'G'
```

is seven, which is the character code known as 'control-and-G'.