10 The occam notation

This section contains a summary of the programming language, based on that in INMOS's *occam Programming Manual* (Prentice-Hall International, 1984). It codifies the language used in this book, which matches neither any implementation known to me, nor (in every detail) any other documentation. I have followed the definition in the *Programming Manual* except where the baroqueness of the quirks in the language that it describes suggest error rather than intention.

Micro-syntax and layout

The layout of an *occam* program is significant in a way unusual in a modern programming language. A program is written as a sequence of *logical lines*, each line preceded by an indentation the depth of which is significant to the structure of the program.

A logical line can be broken into several *physical lines* by breaking it after a binary operator, comma, semicolon or ampersand. The indentation of the logical line is the number of leading spaces on its first physical line, and each of its physical lines must begin with at least this many spaces.

A physical line ends at an end-of-line, or at the first occurrence of two consecutive minus signs, which introduce a comment that extends to the end-of-line. Within a physical line, spaces are significant only between quotes, or where they separate two words or numbers that might otherwise form a single word or number, or two symbols that might be a single symbol.

A *name* is a sequence of letters and digits and full-stops, the first of which is a letter. Two names are different unless they are the same sequences of characters. The following sequences are not names, but are reserved as symbols

$\boxed{\texttt{AFTER}}$	$\boxed{\texttt{CONST}}$	$\boxed{\texttt{PAR}}$	$\boxed{\texttt{TABLE}}$
$\boxed{\texttt{ALT}}$	$\boxed{\texttt{DEF}}$	$\boxed{\texttt{PLACED}}$	$\boxed{\texttt{TIME}}$
$\boxed{\texttt{AND}}$	$\boxed{\texttt{FALSE}}$	$\boxed{\texttt{PRI}}$	$\boxed{\texttt{TRUE}}$
$\boxed{\texttt{ANY}}$	$\boxed{\texttt{FOR}}$	$\boxed{\texttt{PROC}}$	$\boxed{\texttt{VALUE}}$
$\boxed{\texttt{AT}}$	$\boxed{\texttt{IF}}$	$\boxed{\texttt{SEQ}}$	$\boxed{\texttt{VAR}}$
$\boxed{\texttt{BYTE}}$	$\boxed{\texttt{NOT}}$	$\boxed{\texttt{SKIP}}$	$\boxed{\texttt{WHILE}}$
$\boxed{\texttt{CHAN}}$	$\boxed{\texttt{OR}}$	$\boxed{\texttt{STOP}}$	

A sequence of decimal digits is a *numeral*, whose value is the corresponding non-negative integer. A sequence of decimal digits preceded by a sign is a *signednumeral*, whose value is the corresponding integer. In either case, the twos-complement representation of the integer must fit into the word of the implementing machine.

A hash sign, '`#`', followed by a sequence of hexadecimal digits is a *numeral* whose value is the corresponding bit pattern, adjusted to the word size of the implementing machine by adding or removing zeroes at the left. The case of the letters in a hexadecimal numeral is insignificant.

A *character denotation* between a pair of single quotes is also a numeral. Except for the single quote, double quote and asterisk characters, any printable ASCII character, that is one with a code greater than 31_{10} and less than 127_{10}, is a character denotation, whose value is the corresponding code. The following are also character denotations, whose values are the codes of these characters

`*c` or `*C`	carriage return	(13_{10})
`*n` or `*N`	new line	(10_{10})
`*s` or `*S`	space	(32_{10})
`*t` or `*T`	horizontal tab	(9_{10})
`*'`	single quote	(39_{10})
`*"`	double quote	(34_{10})
`**`	asterisk	(42_{10})

Any number between zero and 255_{10} also has a corresponding character denotation consisting of an asterisk, followed by a hash sign and two hexadecimal digits.

A (possibly empty) sequence of no more than 255_{10} character denotations between a pair of double quotes is an abbreviation for a *table* of `BYTE` expressions. The value of the first byte of the table is the number of character denotations, and the remaining bytes are the values of the character denotations in sequence.

Notation

The syntax is described by productions in a BNF modified to cope with the two-dimensional syntax of *occam*. Items enclosed in rectangular boxes represent literal text, and words in italics represent syntactic categories. The production

$$\begin{array}{rl} \textit{repetition} = & \boxed{\texttt{WHILE}}\ \textit{expression} \\ & \quad \textit{process} \end{array}$$

means that a *repetition* consists of the word `WHILE` followed on the same line by an *expression*, with a *process* below it and indented two columns further than the keyword. Vertical bars separate alternative decompositions of classes, brackets surround optional parts, and braces parts which may be omitted or repeated any number of times. For example, the productions

$$\begin{array}{rcl} \textit{expr} & = & \textit{atom}\ [\boxed{\texttt{-}}\ \textit{atom}]\ \Big|\ \textit{atom}\ \{\boxed{\texttt{+}}\ \textit{atom}\} \\ \textit{atom} & = & \boxed{\texttt{(}}\,\textit{expr}\boxed{\texttt{)}}\ \Big|\ \boxed{\texttt{E}} \end{array}$$

admit each of the following as an *expr*

```
E               E - E             E + E
( E )           E - ( E - E )     E + E + E
( E + E )       ( E - E ) - E     E + E + E + E
```

A set of horizontal braces represents a component which may be omitted, or may be repeated any number of times in a vertical column, each instance having the same indentation. The production

$$\begin{array}{rl} \textit{sequence} = & \boxed{\texttt{SEQ}} \\ & \quad \overbrace{\underbrace{\textit{sequence}}} \end{array}\ \Bigg|\ \boxed{\texttt{SKIP}}$$

admits each of

```
SEQ          SEQ          SEQ          SEQ
               SEQ          SKIP         SEQ
                            SEQ            SKIP
                                           SEQ
                                         SEQ
```

as a *sequence*.

The notation is unambiguous except in the case of

$$\begin{aligned} \textit{process} = \ & \textit{declaration}\ \boxed{:} \\ & \textit{process} \end{aligned}$$

where the *declaration* can take up more than one line. In that case, I mean that a colon follows the last thing on the last line of the *declaration*. The awkwardness of that statement is reflected by a corresponding awkwardness in every *occam* parser that I have seen.

Processes

The unit of program structure is the process

$$\textit{process} = \ \textit{action} \,\Big|\, \textit{construct} \,\Big|\, \textit{substitution} \,\Big|\, \textit{block}$$

Processes may be one of a family of primitive *action*s, may be constructed from smaller processes, or may be invocations of named processes. Declarations may precede any process which is then the scope of these declarations.

The primitive processes may change the values of variables, communicate with other processes, or suspend execution.

$$\textit{action} = \ \boxed{\texttt{SKIP}} \,\Big|\, \boxed{\texttt{STOP}} \,\Big|\, \textit{assignment} \,\Big|\, \textit{input} \,\Big|\, \textit{output} \,\Big|\, \textit{timer} \,\Big|\, \textit{delay}$$

`SKIP` is the process that has no effect: it always executes to completion, and never does anything else. `STOP` is similar, in that it does nothing, but differs in that it never terminates.

Assignment

Assignment processes compute the values of expressions, and store them in variables.

$$\textit{assignment} = \ \textit{variable}\ \boxed{\texttt{:=}}\ \textit{expression} \,\Big|\, \textit{vectorvar}\ \boxed{\texttt{:=}}\ \textit{vectorexpr}$$

A simple assignment selects a *variable*, computes the value of the *expression*, and stores that value in that *variable*. Nothing else is changed.

A slice assignment copies values from its *vectorexpr* to the corresponding variables of its *vectorvar*. For a vector assignment to be valid, both vectors must be of the same size, both must be byte-vectors or both word-vectors, and they may not overlap.

Communications

Communication processes co-operate to implement distributed assignments by synchronizing two concurrent processes. The computation of the values is carried out in *output* processes, and the storing of values in *input* processes.

$$input = \quad channel\ \boxed{?}\ target\ \left\{\boxed{;}\ target\right\}$$
$$target = \quad variable \Big| vectorvar \Big| \boxed{\mathtt{ANY}}$$

An *input* process selects a *channel*, selects its *variable* or vector of variables, and suspends execution until an *output* process is executing using the same channel. The notation

$$channel\,?\ target\ ;\ targets$$

abbreviates

```
SEQ
  channel ? target
  channel ? targets
```

No variable in any *target* may appear in the *channel*.

$$output = \quad channel\ \boxed{!}\ source\ \left\{\boxed{;}\ source\right\}$$
$$source = \quad expression \Big| vectorexpr \Big| \boxed{\mathtt{ANY}}$$

An *output* process selects a *channel*, evaluates its *expression* or selects its vector of expressions, and suspends execution until an *input* process is executing using the same channel. The notation

$$channel\ !\ source\ ;\ sources$$

abbreviates

```
SEQ
  channel ! source
  channel ! sources
```

When an *input* and an *output* process are both executing using the same channel, the value of the *source* is stored in the *target* as though by the assignment

$$target\ \mathtt{:=}\ source$$

and both *input* and *output* terminate. Byte-vector outputs must correspond to byte-vector inputs, word-vector outputs to word-vector outputs, and the sizes of the both vectors must be the same. If the *target* is `ANY`, the assignment is not performed; if the *source* is `ANY`, an arbitrary value may be assigned.

Real-time processes

The *timer* process sets a selected *variable* to a value, notionally the present reading on a clock.

$$\mathit{timer} = \boxed{\texttt{TIME}}\ \boxed{\texttt{?}}\ \mathit{variable}$$

The clock reading changes with the passage of time, changing at regular intervals and counting up cyclically through the whole range representable numbers at a constant rate.

A *delay* suspends execution until the reading on the clock represents a time after a given one, to within the range of the clock.

$$\mathit{delay} = \boxed{\texttt{TIME}}\ \boxed{\texttt{?}}\ \boxed{\texttt{AFTER}}\ \mathit{expression}$$

It terminates with no other effect at some time after the reading on the clock has satisfied the relation *clock* `AFTER` *expression*.

Constructed processes

Compound processes execute either by executing their components in sequence or in parallel, or by selecting a component to be executed.

$$\mathit{construct} = \mathit{sequence} \,\big|\, \mathit{conditional} \,\big|\, \mathit{parallel} \,\big|\, \mathit{alternative} \,\big|\, \mathit{repetition}$$

Processes constructed by `SEQ` execute by executing all their components in sequence.

$$\mathit{sequence} = \left.\begin{array}{l} \boxed{\texttt{SEQ}} \\ \quad \overbrace{\underbrace{\mathit{process}}} \end{array}\right| \begin{array}{l} \boxed{\texttt{SEQ}}\ \mathit{replicator} \\ \quad \mathit{process} \end{array}$$

A *sequence* starts with the execution of its first component process, each subsequent process starts if and when its predecessor has terminated, and the whole sequence terminates when and only when the last component has terminated.

A *sequence* is invalid if there is a channel which may be used only for input by one of its components, and only for output by another of its components.

Processes constructed by `IF` execute by executing one of their components, the selection of the component depending on the values of variables.

$$\begin{array}{lcl} \mathit{conditional} & = & \left.\begin{array}{l} \boxed{\texttt{IF}} \\ \quad \overbrace{\underbrace{\mathit{choice}}} \end{array}\right| \begin{array}{l} \boxed{\texttt{IF}}\ \mathit{replicator} \\ \quad \mathit{choice} \end{array} \\ \mathit{choice} & = & \left.\begin{array}{l} \mathit{expression} \\ \quad \mathit{process} \end{array}\right| \mathit{conditional} \end{array}$$

When a *conditional* executes it evaluates the *expressions* guarding its *choices* and those of its components. The expressions should evaluate to `TRUE` or to `FALSE`. A simple *choice* is ready if its condition is `TRUE`, and a *conditional* is ready if it has a ready component. A ready *conditional* executes by executing the first of its ready components, and a ready simple *choice* executes by executing its *process* part. An unready *conditional* executes exactly like `STOP`, and fails to terminate.

Processes constructed by `PAR` execute by the concurrent execution of all their components.

$$\mathit{parallel} = \left.\begin{array}{l}\boxed{\texttt{PAR}}\\ \quad\overbrace{\underbrace{\mathit{process}}}\end{array}\right| \begin{array}{l}\boxed{\texttt{PAR}}\ \mathit{replicator}\\ \quad \mathit{process}\end{array}$$

When a *parallel* construct executes, each of its component processes is executed. The whole construct terminates when and only when all of the components have terminated.

A *parallel* construct is invalid if any of its components may change the value of a variable which may be used in any other of its components. It is similarly invalid if there is a channel which may be used by two of the component processes, both using the channel for input, or both using it for output.

Processes constructed by `ALT` execute by executing one of their components, the selection of the component depending on the readiness of other processes to communicate.

$$\begin{array}{lcl}
\mathit{alternative} & = & [\boxed{\texttt{PRI}}]\ \mathit{symmetricalt}\\
\mathit{symmetricalt} & = & \left.\begin{array}{l}\boxed{\texttt{ALT}}\\ \quad\overbrace{\underbrace{\mathit{guarded}}}\end{array}\right| \begin{array}{l}\boxed{\texttt{ALT}}\ \mathit{replicator}\\ \quad \mathit{guarded}\end{array}\\
\mathit{guarded} & = & \left.\begin{array}{l}\mathit{guard}\\ \quad\mathit{process}\end{array}\right| \mathit{symmetricalt}
\end{array}$$

A simple *guarded* process is ready if its *guard* is ready, and a *symmetricalt* is ready if it has a ready *guarded* component. A ready *symmetricalt* executes by executing one of its ready components, and a ready simple *guarded* process executes by executing its *guard* and its *process* part in sequence. Executing an unready *symmetricalt* suspends execution until it should become ready. One of its components is then selected from amongst those that become ready simultaneously with the first to become ready.

If the whole *alternative* is preceded by `PRI` the first ready component is chosen whenever a choice is made, and each of its *symmetricalt* components behaves as though they too were preceded by `PRI`.

The readiness of a simple *guarded* process is determined by the readiness of its *guard*.

$$
\begin{aligned}
\textit{guard} &= \left[\textit{expression}\ \boxed{\texttt{\&}}\right]\ \textit{atomic}\\
\textit{atomic} &= \textit{input}\,\Big|\,\textit{delay}\,\Big|\,\boxed{\texttt{SKIP}}
\end{aligned}
$$

The *expression* should evaluate to `TRUE` or to `FALSE`, and a *guard* is ready if and only if its *expression* is `TRUE` and its *atomic* action is ready. An omitted *expression* is treated as if it were the constant `TRUE`. A `SKIP` is always ready, a *delay* is ready when the clock reading has satisfied its condition, and an *input* is ready when the *output* corresponding to it has begun. Thus, an *atomic* action is ready when it can be executed to completion, or in the case of a multiple *input* when the first of its inputs can be executed to completion.

A *guard* executes by executing the *atomic* process part.

Regular arrays of processes are composed by replication from the constructors above.

$$
\textit{replicator} = \ \textit{name}\ \boxed{\texttt{=}}\ \boxed{\texttt{[}}\textit{expression}\ \boxed{\texttt{FOR}}\ \textit{expression}\boxed{\texttt{]}}
$$

A *replicator* attached to a constructor replicates the single component of that construct, introducing in each copy a new binding for its *name*. The value of the second *expression*, which must be non-negative, gives the number of replications.

For each array constructor `CONS`,

```
CONS i = [b FOR c]
  component
```

is the same as an empty `CONS` construction if the value of c is zero. If the value of c is one, it is the same as a `CONS` construction with one component the same as *component* except that i is bound to the value of b. For larger positive c, it is equivalent to

```
CONS
  CONS i = [b FOR 1]
    component
  CONS i = [(b)+1  FOR (c)-1]
    component
```

The only form of unbounded loop is the sequential loop

repetition = $\boxed{\texttt{WHILE}}$ *expression*
process

which executes exactly as though it were

```
IF
  expression
    SEQ
      process
      WHILE expression
        process
  NOT (expression)
    SKIP
```

Declarations

Declarations and definitions bind identifiers to constants, variables, channels and named processes.

block = *declaration* $\boxed{:}$
process

declaration = *constdefs* | *vardecls* | *chandecls* | *procdecl*

If a *declaration* precedes the *process*, then free occurrences in that *process* of the name which it introduces are bound by that *declaration*.

For each kind declaration for which it is legal,

```
DECL declaration, declarations :
process
```

abbreviates

```
DECL declaration :
DECL declarations :
process
```

A constant definition makes a new binding of a *name* to a constant value or vector of constant values.

constdefs = $\boxed{\texttt{CONST}}$ *constdef* $\{\boxed{,}$ *constdef*$\}$

constdef = *name* $\boxed{=}$ *expression* | *name* $\boxed{=}$ *vectorexpr*

In the scope of the definition, the *name* can be used as if it were a constant or vector of constants as appropriate. Its value is that of the *expression* or *vectorexpr*.

An expression is a compilation constant if every name in it is bound by a constant definition. All expressions appearing in constant definitions must be compilation constants.

A variable declaration introduces a new variable or vector of variables, and binds a *name* to that variable or vector.

$$\begin{aligned} vardecls &= \boxed{\texttt{VAR}}\ vardecl \ \{\boxed{,}\ vardecl\} \\ vardecl &= name \,\big|\, name\boxed{\texttt{[}}\,[\boxed{\texttt{BYTE}}]\, expression\boxed{\texttt{]}} \end{aligned}$$

A declaration consisting of a *name* alone introduces a simple variable.

If the *name* is followed by an *expression* in brackets, a vector of new variables is introduced. The value of the *expression*, which must be positive, is the number of variables in the vector. The word `BYTE` inside the left bracket indicates a vector of byte variables.

$$\begin{aligned} variable &= name \,\big|\, vectorvar\ selector \\ vectorvar &= name \,\big|\, vectorvar\ slicer \end{aligned}$$

If a value is to be stored in a byte variable selected from a vector, the value is truncated to its least significant eight bits before being stored.

A channel declaration introduces a new channel or vector of channels, and binds a name to the new channel or vector.

$$\begin{aligned} chandecls &= \boxed{\texttt{CHAN}}\ chandecl \ \{\boxed{,}\ chandecl\} \\ chandecl &= name \,\big|\, name\ \boxed{\texttt{[}}\, expression\boxed{\texttt{]}} \end{aligned}$$

A declaration consisting of a *name* alone introduces a new channel. If the *name* is followed by an *expression* in brackets, a vector of new channels is introduced. The value of the *expression*, which must be positive, is the number of channels in the vector.

A *channel* is denoted by a *name* bound to a channel, or by selecting one from a vector of channels.

$$channel = name \,\big|\, name\ \boxed{\texttt{[}}\, expression\boxed{\texttt{]}}$$

A channel selection is valid if the *name* is bound to a vector of channels, and the value of the *expression* in brackets is non-negative and less than the number of channels in that vector. The channels in a vector are indexed consecutively from zero for the first, and the channel selected is that indexed by the value of the *expression*.

Named processes

A name may be bound to a process

$$\begin{aligned} \textit{procdecl} &= \boxed{\texttt{PROC}}\ \textit{name}\ [\textit{formals}]\ \boxed{\texttt{=}} \\ &\quad\ \textit{process} \\ \textit{substitution} &= \textit{name}\ [\textit{actuals}] \end{aligned}$$

so that in the scope of

```
PROC name =
  process
```

the *name* may appear with the same meaning as the *process*. Names free in the text of the *process* and not bound by parameters of the process declaration are bound in the environment of the declaration.

The *process* is the scope of bindings of those names which appear as formal parameters in the declaration

$$\begin{aligned} \textit{formals} &= \boxed{\texttt{(}}\ \textit{formal}\ \{\boxed{\texttt{,}}\ \textit{formal}\}\ \boxed{\texttt{)}} \\ \textit{formal} &= \textit{constformals} \mid \textit{varformals} \mid \textit{chanformals} \end{aligned}$$

to the meanings of actual parameters that appear at the point of each substitution

$$\begin{aligned} \textit{actuals} &= \boxed{\texttt{(}}\ \textit{actual}\ \{\boxed{\texttt{,}}\ \textit{actual}\}\ \boxed{\texttt{)}} \\ \textit{actual} &= \textit{constactual} \mid \textit{varactual} \mid \textit{chanactual} \end{aligned}$$

The number and kinds of formal and actual parameters must correspond. In each case,

KEY *formal*, *formals*

abbreviates

KEY *formal*, KEY *formals*

Constant parameters bind names to the values or vectors of values.

$$\begin{aligned} \textit{constformals} &= \boxed{\texttt{VALUE}}\ \textit{constformal}\ \{\boxed{\texttt{,}}\ \textit{constformal}\} \\ \textit{constformal} &= \textit{name} \mid \textit{name}\ \boxed{\texttt{[}}\boxed{\texttt{]}} \\ \textit{constactual} &= \textit{expression} \mid \textit{vectorexpr} \end{aligned}$$

A *name* appearing alone as a *constformal* must be bound to the value of an *expression*, and a *name* followed by a pair of brackets must be bound to a vector of values.

Variable parameters bind names to variables or vectors of variables.

$$\begin{aligned} \mathit{varformals} &= \boxed{\texttt{VAR}}\ \mathit{varformal}\ \{\boxed{,}\ \mathit{varformal}\} \\ \mathit{varformal} &= \mathit{name} \,\big|\, \mathit{name}\ \boxed{\texttt{[}}\boxed{\texttt{]}} \\ \mathit{varactual} &= \mathit{variable} \,\big|\, \mathit{vectorvar} \end{aligned}$$

A *name* appearing alone as a *varformal* must be bound to a simple variable, which must not be a byte variable selected from a vector. A *name* followed by a pair of brackets must be bound to a vector of variables, which must not be a byte slice of a vector.

Channel parameters bind names to channels or vectors of channels.

$$\begin{aligned} \mathit{chanformals} &= \boxed{\texttt{CHAN}}\ \mathit{chanformal}\ \{\boxed{,}\ \mathit{chanformal}\} \\ \mathit{chanformal} &= \mathit{name} \,\big|\, \mathit{name}\ \boxed{\texttt{[}}\boxed{\texttt{]}} \\ \mathit{chanactual} &= \mathit{channel} \,\big|\, \mathit{name} \end{aligned}$$

A *name* appearing alone as a *chanformal* must be bound to a simple channel, and a *name* followed by a pair of brackets must be bound to a vector of channels.

Expressions

The simplest expressions are literal constants

$$\mathit{literal} = \mathit{numeral} \,\big|\, \boxed{\texttt{TRUE}} \,\big|\, \boxed{\texttt{FALSE}}$$

The value of a *numeral* was described informally above. The value of `TRUE` is a word in which every bit is one, and the value of `FALSE` is a word in which every bit is zero.

$$\mathit{rand} = \mathit{name} \,\big|\, \mathit{literal} \,\big|\, \mathit{selection} \,\big|\, \boxed{\texttt{(}}\mathit{expression}\boxed{\texttt{)}}$$

A *name* may appear as an operand if it is bound to a constant, in which case its value is that constant; if it is bound to a value by a *replicator*, or by a `VALUE` parameter, then that value; if it is bound to a variable in which a value has been stored, then the value most recently stored in that variable. The value of a *selection* is the selected component of the vector of values. Any *expression* in brackets can appear as an operand, having the value of the *expression*.

$$\begin{aligned} \mathit{expression} &= \mathit{rand}\ \left[\mathit{rator}\ \mathit{rand}\right] \Big| \mathit{rand}\ \{\mathit{assoc}\ \mathit{rand}\} \Big| \mathit{signed} \\ \mathit{signed} &= \mathit{signednumeral} \Big| \mathit{monop}\ \mathit{rand} \end{aligned}$$

Expressions are fully bracketed combinations of operands and operators

$$\begin{aligned} \mathit{rator} &= \mathit{arithmetic} \Big| \mathit{comparison} \Big| \mathit{logical} \Big| \mathit{boolean} \Big| \mathit{shift} \\ \mathit{monop} &= \boxed{\texttt{-}} \Big| \boxed{\texttt{NOT}} \end{aligned}$$

except that the brackets may be omitted from sequences of operands joined by the same associative operator

$$\mathit{assoc} = \boxed{\texttt{+}} \Big| \boxed{\texttt{*}} \Big| \mathit{logical} \Big| \mathit{boolean}$$

The *arithmetic* operators treat their operands as signed twos-complement representations of integers.

$$\mathit{arithmetic} = \boxed{\texttt{+}} \Big| \boxed{\texttt{-}} \Big| \boxed{\texttt{*}} \Big| \boxed{\texttt{/}} \Big| \boxed{\texttt{\\}}$$

The value of `a + b` is the sum of (the values of) a and b
`a - b` the difference of a and b
`a * b` the product of a and b
`a / b` the result of dividing a by b
`a \ b` the remainder on dividing a by b
provided that the value has a twos-complement representation as a word-sized bit pattern. The result of division is an integer and is rounded towards zero in case it cannot be exact, so that provided that b is not zero, nor is a the most negative representable integer and b minus one,

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

and the sign of the remainder is that of a.

The monadic subtraction operator negates its argument, so that

```
(- a) = (0 - a)
```

The value of an expression in a comparison operator is `TRUE` or `FALSE` according to a test on the values of the operands.

$$\textit{comparison} = \boxed{\texttt{<}} \,\Big|\, \boxed{\texttt{<=}} \,\Big|\, \boxed{\texttt{=}} \,\Big|\, \boxed{\texttt{>}} \,\Big|\, \boxed{\texttt{>=}} \,\Big|\, \boxed{\texttt{<>}} \,\Big|\, \boxed{\texttt{AFTER}}$$

The value of	`a <  b` is `TRUE`	if a is less than b
	`a <= b`	if a is not more than b
	`a =  b`	if a is the same as b
	`a >= b`	if a is not less than b
	`a >  b`	if a is more than b
	`a <> b`	if a is not the same as b

and is `FALSE` otherwise.

Let h be the smallest positive integer that is too large to have a twos-complement representation as a word sized bit-pattern.

The value of `a AFTER b` is `TRUE` if $a < b$ and $(a - b) < h$
or $a > b$ and $(b - a) > h$
and is `FALSE` otherwise.

Each bit in the value of an expression in a logical operator

$$\textit{logical} = \boxed{\texttt{/\textbackslash}} \,\Big|\, \boxed{\texttt{\textbackslash/}} \,\Big|\, \boxed{\texttt{><}}$$

is determined by applying the operator to corresponding bits of the operands, according to the rules

```
0 /\ b = 0
1 /\ b = b
0 \/ b = b
1 \/ b = 1
0 >< b = b
b >< 0 = b
1 >< 1 = 0
```

where b is either 0 or 1. The monadic operator `NOT` complements each bit of its argument.

Expressions in the Boolean operators

$$\textit{boolean} = \boxed{\texttt{AND}} \,\Big|\, \boxed{\texttt{OR}}$$

are evaluated consistently with the rules

```
TRUE AND b  = b
FALSE AND b = FALSE
TRUE OR b   = TRUE
FALSE OR b  = b
```

The right operand may be an expression with no defined meaning (such as a division by zero) in case the value of the whole expression does not depend on it.

The value of an expression in a *shift* operator is the same bit pattern as the left operand, displaced by a number of bits given by the right operand.

$$shift = \boxed{\texttt{<<}} \mid \boxed{\texttt{>>}}$$

The value of `a << b` is a displaced left by b bit positions
`a >> b` a displaced right by b bit positions

provided that the value of b is not negative. Bits in a pattern shifted left are moved to more significant positions, and in a pattern shifted right are moved to less significant positions. Vacant bit positions are filled by zero bits, and bits shifted past the boundary of the word are discarded.

Vector expressions

A vector expression has a value which is a finite sequence of values.

$$vectorexpr = name \mid string \mid table \mid slice$$

A *name* has a vector value if it is bound by a constant definition to a constant vector, in which case the value is that constant vector; by a `VALUE` vector formal parameter, in which case the value is that of the corresponding actual parameter; by a variable vector declaration or by a `VAR` vector formal, in which case the value is the vector of values most recently stored in the variables.

The value of a *string* is the constant byte vector described above. The value of a *table* expression

$$table = \boxed{\texttt{TABLE}}\ \boxed{\texttt{[}}\left[\boxed{\texttt{BYTE}}\right] expression\ \left\{\boxed{\texttt{,}}\ expression\right\}\boxed{\texttt{]}}$$

is also a constant vector, with as many components as there are component expressions, the value of each component being the value of the corresponding expression. Each expression must be a compilation constant. If the word `BYTE` appears inside the left bracket, the value of the *table* is a vector of byte constants, the value of each component of which is the least significant eight bits of the value of the corresponding *expression*.

A single value can be selected from a vector by subscription

$$selection = vectorexpr\ selector$$

$$selector = \boxed{\texttt{[}}\left[\boxed{\texttt{BYTE}}\right] expression\boxed{\texttt{]}}$$

The value of the selecting *expression* must be non-negative and less than the number of components in the vector. The components of a vector are indexed consecutively from zero at the first component, and the value of a *selection* is that component of the value of the *vector* indexed by the value of the *expression*.

A *selector* selects a word from a word vector, unless `BYTE` appears inside the left bracket in which case it selects a byte from a byte vector.

A *slice* of a vector is also a vector.

slice = *vectorexpr slicer*

slicer = $\boxed{\texttt{[}}\,[\boxed{\texttt{BYTE}}]$ *expression* $\boxed{\texttt{FOR}}$ *expression*$\boxed{\texttt{]}}$

A word vector sliced by a word slicer is a word vector, and a byte slice, with `BYTE` inside the left bracket, of a byte vector is a byte vector. The first component of the value of the *slice* is that indexed by the value of the first *expression* of the *slicer*. The number of components in the value of a *slice* is the value of the second *expression* in the *slicer*, which must be positive. The correspondence is such that

```
p[q FOR r][s]             = p[(q) + (s)]
p[BYTE q FOR r][BYTE s]   = p[BYTE (q) + (s)]
```

For a *slice* to be valid, each valid selection from it must be a valid selection from the vector, according to these rules.

Implementation dependencies

An implementation may require some expressions to have values which are readily determinable during compilation. Any name which occurs in such a compilation constant must be bound by a constant definition. The expressions in constant definitions and tables must always be constants. Conventional implementations will require the sizes of declared vectors of variables and channels, and the replication counts of `PAR` replicators also to be constants.

The language described in the *occam Programming Manual* allows of byte selection and slicing of word vectors and vice versa. The meanings of programs which use these depend on an implementation parameter `N` which is the number of bytes in a word. The values of word and byte vectors are identified in such a way that

```
p[BYTE q] = (p[(q)/N] >> ((q)\N)) /\#FF
```

and consistently for slices.

Configuration directives

Although an *occam* program may simply be a process, the *Programming Manual* defines constructions for configuring stand-alone programs.

$$\mathit{program} = \begin{array}{l|l|l} \mathit{system} & \mathit{allocation} & \mathit{global} \\ & \mathit{singleton} & \mathit{program} \end{array}$$

A *program* may be a *system* of processes to be run on different processors, or it may be a *singleton* to be run on a single processor. (An *allocation* includes the information necessary for a compiler to fit the *singleton* to its processor. Although the *Programming Manual* describes this, it is of no interest in this book so left undescribed.)

Declarations which precede a distributed program must themselves be distributable

$$\mathit{globals} = \begin{array}{l|l|l} \mathit{constdefs} & \mathit{chandecls} & \mathit{procdecl} \end{array}$$

so may not declare any variables. Their meaning is exactly the same as it would have been in an ordinary *block*.

A *system* of distributed processes is divided between processors by indicating the distribution of the outermost parallel constructs.

$$\mathit{system} = \begin{array}{l|l} \boxed{\texttt{PLACED}}\ \boxed{\texttt{PAR}} & \boxed{\texttt{PLACED}}\ \boxed{\texttt{PAR}}\ \mathit{replicator} \\ \quad \underbrace{\overbrace{\mathit{program}}} & \quad \mathit{program} \end{array}$$

A *system* executes by the concurrent execution of its component programs. The annotation `PLACED` does not alter the execution of a *parallel* except to indicate that each component is to be executed by a different (set of) processors.

A *singleton* is to be executed by a single processor.

$$\mathit{singleton} = \begin{array}{l|l|l} \mathit{process} & \boxed{\texttt{PRI}}\ \boxed{\texttt{PAR}} & \mathit{declaration}\ \boxed{\texttt{:}} \\ & \quad \underbrace{\overbrace{\mathit{process}}} & \mathit{singleton} \end{array}$$

A *singleton* may either be a *process* to be executed by its processor, or an asymmetric *parallel*.

An asymmetric *parallel* executes by the concurrent execution of its component processes, in the same way as the corresponding symmetric construct, except that it is guaranteed that no component executes unless all syntactically preceding components are unable to proceed, either because they are waiting for communication or because they have terminated. An implementation may impose a small limit on the number of components in an asymmetric *parallel*.

A *declaration* preceding an asymmetric *parallel* has exactly the same meaning as if the *parallel* were symmetric.

Index of definitions