15

The occam 2 notation

This chapter contains a summary of the programming language, based on that in INMOS's *occam 2 Reference Manual*[1]. The intention is that this chapter will help you over any ambiguities in the informal presentation in the early chapters of the book.

The productions which describe the syntax are not the same as those in the *Reference Manual*, and if regarded as a context-free grammar they generate a smaller language. With the qualifications in the accompanying prose in each, both this chapter and the *Reference Manual* describe the same language.

The notation used in this chapter

The structure of the language is described by productions in a BNF modified to cope with the two-dimensional syntax of occam. Items in rectangular boxes represent literal text, and words in italics stand for syntactic classes.

Vertical bars separate alternative parses of a class. A production like

$$\textit{base.type} \;=\; \boxed{\texttt{BOOL}} \;\Big|\; \boxed{\texttt{BYTE}} \;\Big|\; \textit{int.type} \;\Big|\; \textit{float.type}$$

means that an instance of the syntactic class *base.type* can be either of the symbols `BOOL` or `BYTE`, or it can be an instance of either of the classes *int.type* or *float.type*.

Horizontal juxtaposition means that the components of a parse can appear after each other in a single (logical) line. The production

$$\textit{sequential.definition} \;=\; \boxed{\texttt{PROTOCOL}}\,\textit{name}\,\boxed{\texttt{IS}}\,\textit{sequential.protocol}\,\boxed{\texttt{:}}$$

means that an instance of *sequential.definition* consists of five things: the symbol `PROTOCOL`, a *name*, the symbol `IS`, a *sequential.protocol* and a colon symbol in that sequence, all on the same line. Logical lines may be broken, and redundant spaces may be added between symbols on a line, as detailed later.

As a special case, the only parse of

$$\mathit{empty} \;=\;$$

consists of nothing at all.

Vertical juxtaposition means that the components of a parse must appear above each other, on separate logical lines. The production

$$\mathit{value.proc} \;=\; \begin{array}{l|l} \boxed{\texttt{VALOF}} & \mathit{specification} \\ \quad \mathit{process} & \mathit{value.proc} \\ \quad \boxed{\texttt{RESULT}}\ \mathit{expr.list} & \end{array}$$

means that a *value.proc* could be denoted by the symbol `VALOF` with a *process* laid out below it, and the symbol `RESULT` and an *expr.list* on another line below that. While the `VALOF` has the indentation of the whole *value.proc*, the `RESULT` is indented a further two columns, and the intervening *process* has an additional two spaces indentation on each line. Wherever one line is slightly indented from the line above, the offset is always exactly two additional spaces. On the other hand, a *value.proc* could be a *specification* above a *value.proc*, and in this case both parts of the parse have the same indentation.

There are two abbreviations in our notation for sequences of instances of the same kind.

$$\mathit{case.input} \;=\; \begin{array}{l} \mathit{channel}\,\boxed{\texttt{?}}\boxed{\texttt{CASE}} \\ \quad \{\,\mathit{variant}\,\} \end{array}$$

A *case.input* can have any number of instances of *variant* in it, arranged one above the other at the same indentation, which is in this case two spaces to the right of the indentation of the *channel*.

$$\mathit{fun.heading} \;=\; \{_1 \boxed{\texttt{,}}\ \mathit{base.type}\,\}\boxed{\texttt{FUNCTION}}\,\mathit{name}\,\boxed{\texttt{(}}\{_0 \boxed{\texttt{,}}\ \mathit{fun.formals}\,\}\boxed{\texttt{)}}$$

A *fun.heading* is laid out along a line. It begins with a sequence of instances of *base.type*, at least one of them, separated by commas. Between the parentheses there is a possibly empty sequence of instances of *fun.formals* separated by commas.

Micro-syntax and program 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 comma, semicolon, assignment symbol, `FOR`, `FROM`, `IS` or an operator in an expression. 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 logical line can also be broken after an asterisk in a string denotation; in that case the subsequent physical line must begin with an asterisk preceded by at least as many spaces as the indentation of the logical line. The pair of asterisks and the intervening line break and spaces are not a part of the sequence of characters represented by the string.

A physical line ends at an end-of-line, or at the first occurrence outside matched quotation marks 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-stop characters, the first of which is a letter. Two names are different unless they are the same sequence of characters. The following sequences of upper-case letters are not names, but are reserved for use as symbols

```
AFTER          FOR            OR             RETYPES
ALT            FROM           PAR            ROUND
AND            FUNCTION       PLACE          SEQ
ANY            IF             PLACED         SIZE
AT             IS             PLUS           SKIP
BITAND         INT            PORT           STOP
BITNOT         INT16          PRI            TIMER
BITOR          INT32          PROC           TIMES
BOOL           INT64          PROCESSOR      TRUE
BYTE           MINUS          PROTOCOL       TRUNC
CASE           MOSTNEG        REAL32         VAL
CHAN           MOSTPOS        REAL64         VALOF
ELSE           NOT            REM            WHILE
FALSE          OF             RESULT
```

A sequence of decimal digits is a *decimal* numeral; it stands for the non-negative integer so expressed to base ten. A sequence of hexadecimal digits – decimal digits, or upper-case letters from `A` to `F` inclusive – preceded by either a hash sign, `#`, or a dollar sign, `$`, is a *hexadecimal* numeral; it stands for the integer so expressed to base sixteen.

A *real* numeral consists of two sequences of digits separated by a full stop character, representing the decimal point, followed by an exponent denoted by a letter `E`, either a `+` sign or a `-` sign, and a sequence of decimal digits. The *real* numeral stands for the number with that decimal expansion, multiplied by ten raised to the power of the signed integer represented in decimal in the exponent. The whole exponent may be omitted, in which case `E+0` is assumed.

A character denotation between a pair of single quotes is a *byte* 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; its 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 sequence of character denotations between a pair of double quotes is a *string* constant. Its value is an array of `BYTE`, the length of which is the number of character denotations, and the bytes of which are the values of the character denotations taken in sequence.

Data types

Every expression in occam 2 has a type which constrains the values which it may take.

$$base.type \;=\; \boxed{\texttt{BOOL}} \;\big|\; \boxed{\texttt{BYTE}} \;\big|\; int.type \;\big|\; float.type$$

The Boolean type, denoted `BOOL`, has two values denoting truth and falsity. The byte or character type, denoted `BYTE`, is the range of integers from zero to 255_{10}.

$$int.type \;=\; \boxed{\texttt{INT16}} \;\big|\; \boxed{\texttt{INT32}} \;\big|\; \boxed{\texttt{INT64}} \;\big|\; \boxed{\texttt{INT}}$$

The integer types denote the ranges of integers with twos-complement representations in sixteen, thirty-two or sixty-four bits, respectively. The type `INT` is the range of integers representable in twos-complement in a bit pattern of length determined by each implementation. This would normally be the integer type on which an implementation could most efficiently perform arithmetic.

$$float.type \;=\; \boxed{\texttt{REAL32}} \;\big|\; \boxed{\texttt{REAL64}}$$

The floating-point types denote those approximate reals representable according to ANSI/IEEE Standard 754-1985 in thirty-two bits (sign bit, eight bit exponent, twenty-three bit fraction) and sixty-four bits (sign bit, eleven bit exponent, fifty-two bit fraction) respectively.

$$\textit{data.type} \quad = \quad \textit{base.type} \;\Big|\; \boxed{\texttt{[}}\,\textit{expr}\,\boxed{\texttt{]}}\,\textit{data.type}$$

The data types in occam 2 are these primitive types, and the types of finite homogeneous sequences of other data types. An array type has a *size* or length which is determined by the value of an expression of type `INT` between brackets; the values of an array type are sequences of the given size, each component of the sequence being of the type given after the brackets.

Protocols

Communications in occam 2 conform to protocols in the same way that values have types.

$$\begin{array}{lcl} \textit{protocol} & = & \textit{name} \;\Big|\; \textit{simple.protocol} \;\Big|\; \textit{anarchic} \\ \textit{protocol.definition} & = & \textit{sequential.definition} \;\Big|\; \textit{discriminated.definition} \end{array}$$

A *protocol* may be denoted by a *name* bound in a protocol definition, or may be given as a simple protocol expression.

$$\begin{array}{lcl} \textit{simple.protocol} & = & \textit{data.type} \;\Big|\; \textit{count.type}\,\boxed{\texttt{::}}\,\boxed{\texttt{[}}\boxed{\texttt{]}}\,\textit{data.type} \\ \textit{count.type} & = & \textit{int.type} \;\Big|\; \boxed{\texttt{BYTE}} \end{array}$$

To each data type there corresponds a simple protocol with the same name; a communication conforms to this protocol if and only if both the value sent and the variable into which it is received are of the corresponding type. A counted array protocol is described by a *count.type*, and the *data.type* of the components of an array; a communication conforms to this protocol if it consists of a size of the *count.type* and a segment of that size from an array of components of the *data.type.*

$$\begin{array}{lcl} \textit{sequential.definition} & = & \boxed{\texttt{PROTOCOL}}\,\textit{name}\,\boxed{\texttt{IS}}\,\textit{sequential.protocol}\,\boxed{\texttt{:}} \\ \textit{sequential.protocol} & = & \{_1\boxed{\texttt{;}}\;\textit{simple.protocol}\,\} \end{array}$$

In the scope of a sequential protocol definition, the *name* is bound to a sequential protocol. A sequential protocol is denoted by a semicolon-separated sequence of simple protocols. A communication conforms to the sequential protocol if it is an input or output of a sequence of items each component of which conforms to the corresponding simple protocol.

discriminated.definition = $\boxed{\texttt{PROTOCOL}}$ *name*
$\boxed{\texttt{CASE}}$
{ *tagged.protocol* }
$\boxed{\texttt{:}}$

tagged.protocol = *tag* | *tag* $\boxed{\texttt{;}}$ *sequential.protocol*

tag = *name*

In the scope of a discriminated protocol definition, its *name* denotes a discriminated protocol; the definition also binds in its scope the names of each *tag*, each of which must be distinct from the others. An output conforms to the discriminated protocol if it consists of a *tag* of the protocol, and a possibly empty sequence of values which conforms to any corresponding sequential protocol. An input conforms to the discriminated protocol if it is a case input over some or all of the *tag*s of the protocol, each branch of which is an input that conforms to a *tagged.protocol* of the protocol.

anarchic = $\boxed{\texttt{ANY}}$

An output of a value of any data type conforms to the *anarchic* protocol, as does an input into a variable of any type. An output to a channel with this protocol sends the sequence of bytes which retypes the value; an input from a channel with this protocol assigns a sequence of bytes to that variable as if it had been retyped to a sequence of bytes. It is invalid for a co-operating output and input to send and expect byte sequences of different lengths.

Literals

A *literal* constant denotes a specific value of its type, independently of its context.

literal = *bool.literal* | *byte.literal* | *int.literal* | *float.literal*

There are *literal*s of each of the base data types.

bool.literal = $\boxed{\texttt{TRUE}}$ | $\boxed{\texttt{FALSE}}$

The constants `TRUE` and `FALSE` denote truth and falsity, respectively, and are of type `BOOL`.

byte.literal = *byte* | *byte* $\boxed{\texttt{(}}\boxed{\texttt{BYTE}}\boxed{\texttt{)}}$ | *integer* $\boxed{\texttt{(}}\boxed{\texttt{BYTE}}\boxed{\texttt{)}}$

A *byte* numeral optionally followed by `BYTE` in parentheses represents its value, of type `BYTE`. An *integer* between zero and 255_{10} followed by `BYTE` in parentheses represents the `BYTE` of that value.

int.literal = *integer* | *integer* $\boxed{\texttt{(}}$ *int.type* $\boxed{\texttt{)}}$ | *byte* $\boxed{\texttt{(}}$ *int.type* $\boxed{\texttt{)}}$

integer $=$ *decimal* | *hexadecimal*

A *decimal* numeral followed by the name of an integer type in parentheses represents the value of that constant in that type, provided that it lies between zero and the most positive value of its type.

A *hexadecimal* numeral followed by the name of an integer type in parentheses is also an *int.literal*. It represents the signed integer of the given type with that twos-complement representation in the number of bits appropriate to that type which, when regarded as an unsigned binary number, would have the value of the *hexadecimal* numeral. It is valid only if its value lies between the most negative and most positive values of its type.

A *byte* numeral followed by the name of an integer type in parentheses represents the value of that constant in the given type.

If an *int.literal* appears without a specified type, `(INT)` is assumed.

float.literal $=$ *real* `(` *float.type* `)`

A *real* numeral followed by the name of a floating-point type in parentheses is a constant of that type. Its value is that of the numeral rounded to a value of the type, provided that this is in the range of values representable in the type.

Rounding takes a real number within the range of the type to the floating-point number which differs in value by less than half of the weight of its least significant bit; or if there are two numbers differing from it by exactly half the weight of the least significant bit, to the one with zero as its least significant bit.

Arrays

A *table* denotes a sequence of values all of the same type.

table $=$ `[` $\{_1$ `,` *expr* $\}$ `]` | *string* | *name*

A *table* expression consists of a sequence of expressions all of the same type, separated by commas and enclosed in brackets. Its size is the number of expressions, and its type is that of arrays of that many components of the type of each expression. Its value is the sequence of values of the expressions, taken in order. A *string* is equivalent to the *table* expression with the same value. A *name* bound to a *table* value can be used to stand for the value.

part $=$ *table* | *part* `[` *expr* `]` | `[` *part* `FROM` *expr* `FOR` *expr* `]`

An `INT` indexing expression in brackets after a *part* of array type selects a component from the value of the *part*. The value of the expression must be no less than zero, and strictly less than the size of the array. The value of the selection is that component of the value of the array with the given index, counting from zero for the first component.

A contiguous segment of an array value is specified by giving two `INT` expressions: the first is an index into the array value, which component is the first of the segment; the second is the size of the segment. A segment is valid only if each component of it is a valid component of the array from which it is drawn.

Specification and scope

A *specification* binds some names within its scope.

$$\begin{aligned} \textit{specification} &= \textit{declaration} \mid \textit{abbreviation} \mid \textit{definition} \\ \textit{definition} &= \textit{protocol.definition} \mid \textit{proc.definition} \mid \textit{fun.definition} \end{aligned}$$

In the text of the syntactic entity that follows the *specification* a *name* bound in the *specification* refers to that to which it is bound by that *specification*. The scope of a *specification* is the text of the syntactic entity following the *specification*, excluding any text in the scope of a *specification* of the same *name* contained in that syntactic entity.

Declaration

Every declared object or array of objects has a *type*.

$$\textit{type} = \textit{data.type} \mid \textit{channel.type} \mid \textit{port.type} \mid \textit{timer.type}$$

A *specifier* denotes a range of types.

$$\begin{aligned} \textit{specifier} &= \textit{type} \mid \boxed{\texttt{[}}\, \textit{expr.option} \,\boxed{\texttt{]}}\, \textit{specifier} \\ \textit{expr.option} &= \textit{expr} \mid \textit{empty} \end{aligned}$$

If the *specifier* has the form of a *type*, it consists of just that *type*. A *specifier* preceded by an `INT` expression in brackets contains the types of all arrays with size equal to the value of the expression, and with components in the *specifier*. A *specifier* preceded by an empty pair of brackets consists of the types of all arrays with components that have types in the component *specifier*, irrespective of the sizes of the arrays.

A *declaration* introduces new variables, channels, ports or timers, and gives names to them.

$$\textit{declaration} = \textit{type} \{_1 \boxed{\texttt{,}}\ \textit{name}\} \boxed{\texttt{:}}$$

In each case, a multiple *declaration*

$$\begin{array}{l} \textit{type}\ \textit{name}_1,\ \textit{name}_2,\ \ldots\ ,\ \textit{name}_N\ : \\ \textit{scope} \end{array}$$

is equivalent to

$$\begin{array}{l} \textit{type}'\ \textit{name}_1\ : \\ \textit{type}'\ \textit{name}_2\ : \\ \quad\ldots \\ \textit{type}'\ \textit{name}_N\ : \\ \textit{scope} \end{array}$$

where *type*′ denotes the same type as *type*, but contains no occurrences of the declared names.

A *declaration* of a *name* with a *data.type* introduces a new variable of that type, and binds the *name* to that *variable*:

$$\textit{variable} \;=\; \textit{name} \;\Big|\; \textit{variable}\,\boxed{\texttt{[}}\,\textit{expr}\,\boxed{\texttt{]}} \;\Big|\; \boxed{\texttt{[}}\,\textit{variable}\,\boxed{\texttt{FROM}}\,\textit{expr}\,\boxed{\texttt{FOR}}\,\textit{expr}\,\boxed{\texttt{]}}$$

In the scope of the declaration, the *name* can be used to refer to that *variable* for assignment, or to the value most recently assigned to that *variable*.

A component of a *variable* which has an array type is denoted by following the *variable* by an `INT` expression in brackets. The value of the expression must be non-negative and strictly less than the size of the array; the denoted *variable* is the component of the array with that index, the first component having index zero.

A contiguous segment of a *variable* which has an array type is specified by a pair of `INT` expressions: the value of the first is the index in the *variable* of the first component of the segment; the value of the second is the size of the segment. A segment selection is valid only if each component of it is a valid component of the *variable* array from which it is drawn.

A *declaration* of a *name* with a *channel.type* introduces a new channel or array of channels, and binds the *name* to that new object:

$$\textit{channel.type} \;=\; \boxed{\texttt{CHAN}}\,\boxed{\texttt{OF}}\,\textit{protocol} \;\Big|\; \boxed{\texttt{[}}\,\textit{expr}\,\boxed{\texttt{]}}\,\textit{channel.type}$$

To each *protocol* there corresponds a *channel.type*, all communications on channels having that *channel.type* being required to conform to the *protocol*. The type of an array of channels is denoted by giving the size of the array in brackets before the type of the component.

$$\textit{channel} \;=\; \textit{name} \;\Big|\; \textit{channel}\,\boxed{\texttt{[}}\,\textit{expr}\,\boxed{\texttt{]}} \;\Big|\; \boxed{\texttt{[}}\,\textit{channel}\,\boxed{\texttt{FROM}}\,\textit{expr}\,\boxed{\texttt{FOR}}\,\textit{expr}\,\boxed{\texttt{]}}$$

In the scope of a channel declaration, the bound *name* can be used to refer to the collection of channels. Components and segments of arrays of channels can be selected exactly as with variables. A *channel* which has the *channel.type* of a single channel of some protocol can be used for communications which conform to that protocol.

A *declaration* of a *name* with a *port.type* introduces a new port or array of ports, and binds the *name* to that new object:

$$port.type \;=\; \boxed{\texttt{PORT}}\boxed{\texttt{OF}}\, data.type \;\Big|\; \boxed{\texttt{[}}\, expr\, \boxed{\texttt{]}}\, port.type$$

To each *data.type* there corresponds a *port.type*, it being allowed on ports having that *port.type* to read and write values of that *data.type*. The type of an array of ports is denoted by giving the size of the array in brackets before the type of the component.

$$port \;=\; name \;\Big|\; port\, \boxed{\texttt{[}}\, expr\, \boxed{\texttt{]}} \;\Big|\; \boxed{\texttt{[}}\, port\, \boxed{\texttt{FROM}}\, expr\, \boxed{\texttt{FOR}}\, expr\, \boxed{\texttt{]}}$$

In the scope of a port declaration, the bound *name* can be used to refer to the collection of ports. Components and segments of arrays of ports can be selected exactly as with variables. A *port* which has the *port.type* of a single port of some data type can be used for reading and writing values of that type.

A *declaration* of a *name* with a *timer.type* introduces a new timer or array of timers, and binds the *name* to that new object:

$$timer.type \;=\; \boxed{\texttt{TIMER}} \;\Big|\; \boxed{\texttt{[}}\, expr\, \boxed{\texttt{]}}\, timer.type$$

The type of a single timer is denoted `TIMER` and the type of an array of timers is denoted by giving the size of the array in brackets before the type of the component.

$$timer \;=\; name \;\Big|\; timer\, \boxed{\texttt{[}}\, expr\, \boxed{\texttt{]}} \;\Big|\; \boxed{\texttt{[}}\, timer\, \boxed{\texttt{FROM}}\, expr\, \boxed{\texttt{FOR}}\, expr\, \boxed{\texttt{]}}$$

In the scope of a timer declaration, the bound *name* can be used to refer to the collection of timers. Components and segments of arrays of timers can be selected exactly as with variables. A *timer* which has the *timer.type* of a single timer can be used for reading the time on the corresponding clock, or for delaying execution.

Abbreviation

An *abbreviation* makes a new binding of a name to an existing object or to the value of an expression:

$$abbreviation \;=\; value.abbreviation \;\Big|\; object.abbreviation \;\Big|\; retyping$$

In the scope of a *value.abbreviation* the newly bound *name* can be used to stand for the value of the abbreviated expression:

$$value.abbreviation \;=\; \boxed{\texttt{VAL}}\, specifier.option\; name\, \boxed{\texttt{IS}}\, expr\, \boxed{\texttt{:}}$$

$$\textit{specifier.option} \quad = \quad \textit{specifier} \mid \textit{empty}$$

If a *specifier* is quoted in the abbreviation, the abbreviated expression must have a type in that *specifier*.

The *value.abbreviation* is invalid if the *expr* which it abbreviates is invalid. It is invalid in the scope of a *value.abbreviation* to change the value of any *variable* used in the *expr* being abbreviated.

A *variable*, *channel*, *port*, *timer*, or an array, or array of arrays of these, or a component or segment of such an array, can be given a new name by an abbreviation:

$$\textit{object.abbreviation} \quad = \quad \textit{specifier.option name } \boxed{\texttt{IS}} \textit{ object } \boxed{\texttt{:}}$$

$$\textit{object} \quad = \quad \textit{variable} \mid \textit{channel} \mid \textit{port} \mid \textit{timer}$$

In the scope of the *abbreviation* the newly bound *name* can be used to stand for the abbreviated object. If a *specifier* is quoted, the abbreviated object must have a *type* in that *specifier*.

The *object.abbreviation* is invalid if the *object* which it abbreviates is invalid. It is invalid in the scope of an *object.abbreviation* to refer to the abbreviated object except by means of the *name* bound in the abbreviation. It is also invalid in the scope of an *object.abbreviation* to change the value of any *variable* used in selecting the *object* being abbreviated.

It is invalid in the scope of an abbreviation of a component and segment of an array to refer to any part of the array, except that it may be referred to in other abbreviations of components and segments, provided that the abbreviated components and segments are mutually disjoint.

$$\textit{retyping} \quad = \quad \textit{value.retyping} \mid \textit{object.retyping}$$

A *retyping* conversion makes an implementation-dependent binding of a *name* with one type to an expression or value of another. In the scope of the *retyping*, the type of the *name* bound is in the given *specifier*. It is necessary that the number of bits in the representation of values of both types be the same.

$$\textit{value.retyping} \quad = \quad \boxed{\texttt{VAL}} \textit{ specifier name } \boxed{\texttt{RETYPES}} \textit{ expr } \boxed{\texttt{:}}$$

A *value.retyping* behaves like a *value.abbreviation* of an *expr*, except that the value to which the *name* is bound is that value of the specified type which has the same representation as the value of the *expr* abbreviated.

$$\textit{object.retyping} \quad = \quad \textit{specifier name } \boxed{\texttt{RETYPES}} \textit{ variable } \boxed{\texttt{:}}$$

An *object.retyping* behaves like an *object.abbreviation* of a *variable*, except that the value of the *name* is that value of the specified type which has the same representation as the value of the *variable* abbreviated, and values assigned to the *name* are stored as that value of the type of the abbreviated *variable* which has the same representation.

Expressions

An expression has a type which is fixed and is determined by the types of the constants and variables that occur in it. It has also a value determined by the values of the constants that occur in it and the values, when it is evaluated, of the variables that occur in it.

If a process requires the evaluation of an expression which leads to the evaluation of a sub-expression which is invalid, the process behaves like `STOP`.

$$\begin{array}{lcl}
\textit{expr} & = & \textit{rand} \mid \textit{mon.op rand} \mid \textit{rand rator rand} \mid \textit{conversion} \mid \textit{extremum} \\
\textit{rand} & = & \textit{literal} \mid \textit{part} \mid \textit{variable} \mid \boxed{\texttt{(}}\,\textit{expr}\,\boxed{\texttt{)}} \mid \boxed{\texttt{(}}\,\textit{value.proc}\,\boxed{\texttt{)}} \mid \textit{fun.call} \\
\textit{rator} & = & \textit{number.op} \mid \textit{modulo.op} \mid \textit{bit.op} \mid \textit{logic.op} \mid \textit{shift.op} \mid \textit{relate.op}
\end{array}$$

The type and value of an expression which is a *literal* or *part* are those of the *literal* or *part*. The type of an expression which is a *variable* is that of the *variable*, and its value is the value most recently assigned to the *variable*.

Expressions containing two or more applications of an operator must be grouped by parentheses to specify the order of evaluation.

A *value.proc* or *fun.call* which occurs in an *expr* must return a single value; the type and value of such an expression are those of the returned value.

$$\textit{mon.op} \;=\; \boxed{\texttt{-}} \mid \boxed{\texttt{MINUS}} \mid \boxed{\texttt{\~{}}} \mid \boxed{\texttt{BITNOT}} \mid \boxed{\texttt{NOT}} \mid \boxed{\texttt{SIZE}}$$

If x has any *int.type* or *float.type*, and *zero* is the zero of the same type, then `-`x is the same as *zero* `-` x, and if x has any *int.type*, `MINUS` x is the same as *zero* `MINUS` x.

If x has any *int.type*, and *ones* is the minus one of the same type, represented by a bit pattern every bit of which is one, then `~`x and `BITNOT` x are both the same as *ones* `><` x.

If x has type `BOOL`, so does `NOT`x; and its value is true or false according as the value of x is false or true, respectively.

If x has an array type, $[n]T$, the type of `SIZE` x is `INT`, and its value is the number, n, of components in x.

$$\textit{number.op} \;=\; \boxed{\texttt{+}} \mid \boxed{\texttt{-}} \mid \boxed{\texttt{*}} \mid \boxed{\texttt{/}} \mid \boxed{\texttt{\textbackslash}} \mid \boxed{\texttt{REM}}$$

If x and y have the same *int.type* and $\oplus$ is a *number.op*, then the type of $x \oplus y$ is the same *int.type*. The value of $x + y$ is the sum of the values of x and y, that of $x - y$ is the difference of their values, that of $x * y$ is the product of their values. Provided that y is not zero, the value of x / y is the quotient of x by y truncated to an integer, and the value of $x \setminus y$ is the remainder from that division. If the division is not exact, the sign of the remainder is the same as that of x, so that provided it is valid $((x$ `/` $y)$ `*` $y)$ `+` $(x$ `\` $y)$ has the same value as x.

In each case the expression is invalid if its value has no representation of the type; division or remainder by zero is invalid.

If x and y have the same *float.type* and $\oplus$ is a *number.op*, then the type of $x \oplus y$ is the same *float.type*. In each case the value is rounded to the nearest value of its type, as specified by ANSI/IEEE Standard 754-1985. The value of x `+` y is the sum of the values of x and y, that of x `-` y is the difference of their values, that of x `*` y is the product of their values, and provided that y is not zero that of x `/` y is the quotient of x by y. The value of x `\` y is the remainder on taking from x a number of multiples of y which is x `/` y rounded to an integer.

In each case the expression is invalid if its value is outside the range representable by proper values of the type; division or remainder by zero is invalid. That is, an expression is invalid if the Standard would require its value to be an 'infinity' or a 'not a number' value.

The remainder operation can also be denoted `REM` which is a synonym for `\`.

$$\textit{modulo.op} \;=\; \boxed{\texttt{PLUS}} \;\Big|\; \boxed{\texttt{MINUS}} \;\Big|\; \boxed{\texttt{TIMES}}$$

If x and y have the same *int.type* and $\oplus$ is a *modulo.op*, then the type of $x \oplus y$ is the same *int.type*. In each case, the value is calculated modulo the number of representable values of that type. The value of x `PLUS` y differs from the sum of the values of x and y by a multiple of the range of its type, that of x `MINUS` y from the difference of their values, and that of x `TIMES` y from the product of their values.

Thus, for example, if x and y are of `INT16` the value of x `PLUS` y is that `INT16` which differs from $x + y$ by a multiple of 2^{16}.

$$\textit{logic.op} \;=\; \boxed{\texttt{AND}} \;\Big|\; \boxed{\texttt{OR}}$$

If x and y are both of type `BOOL`, and $\oplus$ is a *logic.op*, then $x \oplus y$ has type `BOOL`. If x is false, then so is x `AND` y and it need not be valid to evaluate y, otherwise its value is that of y. If x is true, then so is x `OR` y and it need not be valid to evaluate y, otherwise its value is that of y.

As a concession to the associativity of `AND` and `OR`, expressions in these operators need not be fully parenthesized; $(x$ `AND` $y)$ `AND` z may be written x `AND` y `AND` z, and $(x$ `OR` $y)$ `OR` z may be written x `OR` y `OR` z.

$$\textit{bit.op} \;=\; \boxed{\texttt{/\textbackslash}} \;\Big|\; \boxed{\texttt{BITAND}} \;\Big|\; \boxed{\texttt{\textbackslash/}} \;\Big|\; \boxed{\texttt{BITOR}} \;\Big|\; \boxed{\texttt{><}}$$

If x and y have the same *int.type* and $\oplus$ is a *bit.op*, then the type of $x \oplus y$ is the same *int.type*. In each case, the value has a representation each bit of which is determined by the corresponding bit in the representations of the values of x and y. A bit in x `/\` y is one when both of the corresponding bits in x and y are one, a bit in x `\/` y is zero when both of the corresponding bits in x and y are zero, and a bit in x `><` y is one when the corresponding bits in x and y differ. `BITAND` is a synonym for `/\`, and `BITOR` is a synonym for `\/`.

$$\textit{shift.op} \;=\; \boxed{\texttt{<<}} \;\big|\; \boxed{\texttt{>>}}$$

If x has an *int.type*, y is of type `INT`, and $\oplus$ is a *shift.op*, then the type of $x \oplus y$ is the same as that of x. In each case, the value has a representation obtained by shifting the representation of the value of x by a number of bit positions determined by the value of y. Bits which are shifted out of the representation are discarded, vacated bit positions are filled by zero bits. The value of x `<<` y is x shifted away from the least significant bit (leftwards) by y bits; that of x `>>` y is x shifted away from the most significant bit (rightwards) by y bits.

It is invalid for the value of y to be negative, or to exceed the number of bits in the representation of the type being shifted.

$$\begin{aligned} \textit{relate.op} &= \textit{equality} \;\big|\; \textit{inequality} \;\big|\; \textit{after} \\ \textit{equality} &= \boxed{\texttt{=}} \;\big|\; \boxed{\texttt{<>}} \end{aligned}$$

If x and y have the same *base.type* and $\oplus$ is an *equality*, then $x \oplus y$ has type `BOOL`. The value of x `=` y is true if x and y are equal, and false otherwise; that of x `<>` y is false if x and y are equal, and true otherwise.

$$\textit{inequality} \;=\; \boxed{\texttt{<}} \;\big|\; \boxed{\texttt{<=}} \;\big|\; \boxed{\texttt{>}} \;\big|\; \boxed{\texttt{>=}}$$

If x and y have the same *int.type* or *float.type*, or are both of type `BYTE`, and $\oplus$ is an *inequality*, then $x \oplus y$ has type `BOOL`. The value of x `<` y is true if x is less than y, and false otherwise; that of x `<=` y is false if x is greater than y, and true otherwise; that of x `>` y is true if x is greater than y, and false otherwise; that of x `>=` y is false if x is less than y, and true otherwise.

$$\textit{after} \;=\; \boxed{\texttt{AFTER}}$$

If x and y have the same *int.type*, then x `AFTER` y has type `BOOL`. If *zero* is the zero of the same type as x and y, the value of x `AFTER` y is the same as that of (x `MINUS` y) `>` *zero*.

$$\begin{aligned} \textit{conversion} &= \textit{base.type rounding rand} \\ \textit{rounding} &= \textit{empty} \;\big|\; \boxed{\texttt{ROUND}} \;\big|\; \boxed{\texttt{TRUNC}} \end{aligned}$$

Data type conversion renders a value of one *base.type* as a numerically similar value of another *base.type*. In each case, the type of the *conversion* is the *base.type* specified. The value is approximately that of the *rand* which must itself be of some *base.type*.

Conversions are exact within and between *int.type*, `BYTE` and `BOOL`. The conversion $T\ x$ is of type T and its value is the same integer as that of x. It is invalid if the value of x is outside the range of values of T. For the purpose of conversion, `FALSE` is treated as zero, and `TRUE` is treated as one. If x has a *float.type*, the conversion `REAL64` x is the `REAL64` with the same value as x.

For other conversions to or from a *float.type*, either rounding or truncation must be specified. The conversion T `ROUND` x is of type T and has the value of x rounded to a value representable in that type. The conversion T `TRUNC` x is of type T and has the value of x truncated by discarding bits less significant than the least significant bit in the representation of the result. Both are invalid if the value of the result of rounding or truncation would be outside the range of values of T.

$$\mathit{extremum} \;=\; \boxed{\texttt{MOSTPOS}}\ \mathit{int.type} \;\Big|\; \boxed{\texttt{MOSTNEG}}\ \mathit{int.type}$$

If T is any *int.type*, the value of `MOSTPOS` T is the largest of that type, and that of `MOSTNEG` T is the smallest (most negative) of that type. In both cases the type of the expression is T.

$$\mathit{value.proc} \;=\; \left.\begin{array}{l} \boxed{\texttt{VALOF}} \\ \quad \mathit{process} \\ \quad \boxed{\texttt{RESULT}}\ \mathit{expr.list} \end{array}\right| \begin{array}{l} \mathit{specification} \\ \mathit{value.proc} \\ \end{array}$$

A *value.proc* returns the values of a list of expressions, evaluated after the execution of the *process* in its body. If a *specification* precedes a *value.proc*, the scope of the *specification* is the *value.proc* including its returned *expr.list*, but not the text of the *specification* itself.

The *process* in a *value.proc* may not assign to variables with a scope extending beyond the *value.proc* and any immediately preceding *specification*; nor may it contain any communication, nor alternation, nor parallel process. If it stops or fails to terminate, then so does any *process* which evaluates the *value.proc*.

Processes

The unit of program structure is the *process*

$$\mathit{process} \;=\; \mathit{atom} \;\Big|\; \mathit{construct} \;\Big|\; \mathit{discrimination} \;\Big|\; \mathit{loop} \;\Big|\; \mathit{instance} \;\Big|\; \mathit{block}$$

Every *process* in a program is either a simple *atom*, or is constructed of a number of simpler processes, or is an invocation of a named process.

$$\mathit{block} \;=\; \left.\begin{array}{l} \mathit{specification} \\ \mathit{process} \end{array}\right| \begin{array}{l} \mathit{allocation} \\ \mathit{process} \end{array}$$

If a *specification* precedes a *process*, the scope of the *specification* is the *process*, but not the text of the *specification* itself. An *allocation* may precede a *process* to locate declared objects in the store of the processor on which the *process* is to be executed.

$$atom \; = \; \boxed{\texttt{SKIP}} \mid \boxed{\texttt{STOP}} \mid assignment \mid input \mid output$$

The process `SKIP` is immediately ready to perform no action, and always terminates. The process `STOP` performs no actions, and never terminates. Implementations of occam may provide assistance in detecting the execution of `STOP`.

Assignment

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

$$\begin{array}{lcl} assignment & = & variable.list\,\boxed{\texttt{:=}}\,expr.list \\ variable.list & = & \{_1\boxed{\texttt{,}}\;variable\,\} \\ expr.list & = & \{_1\boxed{\texttt{,}}\;expr\,\} \mid \boxed{\texttt{(}}\,value.proc\,\boxed{\texttt{)}} \mid fun.call \end{array}$$

An *assignment* stores in a list of variables the values, before the assignment, of a list of expressions; the number of variables and expressions must agree, and the type of each variable must be the same as that of the corresponding expression. An anonymous function or a call on a named function can appear in place of the list of expressions, and must return a list of values of the same number and types as the list of variables.

If there are two or more variables assigned together, they must be detectably disjoint; the meaning of

$$v_1,\ \ldots,\ v_n := e_1,\ \ldots,\ e_n$$

is the same as

```
SEQ
  PAR
    t1 := e1
    ...
    tn := en
  PAR
    v1 := t1
    ...
    vn := tn
```

where $t_1, \ldots, t_n$ are new variables of the same types as the corresponding expressions.

Communication

Output processes evaluate expressions and communicate their values to other processes.

$$\begin{array}{lcl} \textit{output} & = & \textit{channel.output} \mid \textit{port.output} \\ \textit{channel.output} & = & \textit{channel} \boxed{\texttt{!}} \textit{source.list} \mid \textit{channel} \boxed{\texttt{!}} \textit{tagged.source.list} \end{array}$$

An output to a channel selects the *channel*, waits for the corresponding input to be ready, and implements a sequence of communications.

$$\begin{array}{lcl} \textit{tagged.source.list} & = & \textit{tag} \mid \textit{tag} \boxed{\texttt{;}} \textit{source.list} \\ \textit{source.list} & = & \{_1 \boxed{\texttt{;}}\ \textit{source}\,\} \\ \textit{source} & = & \textit{expr} \mid \textit{expr} \boxed{\texttt{::}} \textit{expr} \end{array}$$

In the course of a *channel.output*, each *source* is evaluated. If the list of sources starts with a *tag*, that *tag* is sent; the value of each *source* in the *source.list* is then sent in sequence.

The value of a *source* which is an expression is the value of the expression. The value of a *source* which is a counted array consists of the size, which is the value of the first expression, and of a segment of that number of components from the start of the value of the second expression.

The sequence of an optional *tag* and *source.list* must conform to the protocol of the channel.

$$\textit{port.output} \;=\; \textit{port} \boxed{\texttt{!}} \textit{expr}$$

An output to a port selects the *port*, evaluates its expression, and sends the values of the expression to that *port*. The type of the *port* must be `PORT` of the type of the expression.

$$\textit{input} \;=\; \textit{channel.input} \mid \textit{port.input} \mid \textit{timer.input} \mid \textit{delay}$$

Input processes generally assign to variables some values drawn from outside the process.

$$\textit{channel.input} \;=\; \textit{channel} \boxed{\texttt{?}} \textit{target.list} \mid \textit{channel} \boxed{\texttt{?}} \boxed{\texttt{CASE}} \textit{tagged.list}$$

An input from a channel selects the *channel*, waits for the corresponding output to be ready, and then implements a sequence of communications.

$$\begin{array}{lcl} \textit{tagged.list} & = & \textit{tag} \mid \textit{tag} \boxed{\texttt{;}} \textit{target.list} \\ \textit{target.list} & = & \{_1 \boxed{\texttt{;}}\ \textit{target}\,\} \\ \textit{target} & = & \textit{variable} \mid \textit{variable} \boxed{\texttt{::}} \textit{variable} \end{array}$$

If the list of targets starts with a *tag*, the input process stops unless the received value matches the expected *tag*. When the next *target* is a *variable*, the *variable* is selected and the received value is assigned to it. When the next *target* is a counted array, the two variables are selected, the communicated size is stored in the first *variable*, and the communicated array is stored in a segment of that number of components at the start of the second variable. The sequence of an optional *tag* and *target.list* must conform to the protocol of the channel.

port.input = *port* $\boxed{\texttt{?}}$ *variable*

An input from a port selects the *port*, selects a *variable*, and assigns to that *variable* a value read from that *port*. The type of the *port* must be `PORT` of the type of the *variable*.

timer.input = *timer* $\boxed{\texttt{?}}$ *variable*

An input from a timer selects the *timer*, selects a *variable*, and assigns to that *variable* the reading of the clock corresponding to that *timer*.

delay = *timer* $\boxed{\texttt{?}}\boxed{\texttt{AFTER}}$ *expr*

A *delay* process terminates with no effect once the reading on the selected *timer* has satisfied *reading* `AFTER` *expr*.

Constructed processes

Processes can be constructed from components all of which are executed in sequence or in parallel, or one of which is selected for execution.

construct = *sequence* | *conditional* | *parallel* | *alternation*

In each case, the representation of a constructed processes consists of a constructing keyword followed by a number of components.

For each of these constructions, a regular array of processes can be composed by replication.

replicator = *name* $\boxed{\texttt{=}}$ *expr* $\boxed{\texttt{FOR}}$ *expr*

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 a non-negative `INT`, gives the number of replications. In each case

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, as if by a value abbreviation, to the value of the `INT` expression b. For larger positive c, it is equivalent to

```
cons
  cons i = b FOR 1
    component
  cons i = (b) + 1 FOR (c) - 1
    component
```

Implementations may require that the number of replications of some constructions, in particular *parallel* constructions, be constants determinable by a compiler.

Sequential processes

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

$$\mathit{sequence} \;=\; \begin{array}{l} \boxed{\mathtt{SEQ}} \\ \quad \{\mathit{process}\} \end{array} \;\Bigg|\; \begin{array}{l} \boxed{\mathtt{SEQ}}\ \mathit{replicator} \\ \quad \mathit{process} \end{array}$$

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.

Conditional processes

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

$$\begin{array}{lcl} \mathit{conditional} & = & \begin{array}{l} \boxed{\mathtt{IF}} \\ \quad \{\mathit{choice}\} \end{array} \;\Bigg|\; \begin{array}{l} \boxed{\mathtt{IF}}\ \mathit{replicator} \\ \quad \mathit{choice} \end{array} \\ \mathit{choice} & = & \mathit{guarded.choice} \;\Big|\; \mathit{conditional} \;\Big|\; \begin{array}{l} \mathit{specification} \\ \mathit{choice} \end{array} \\ \mathit{guarded.choice} & = & \begin{array}{l} \mathit{expr} \\ \quad \mathit{process} \end{array} \end{array}$$

When a *conditional* executes it evaluates in textual order the expressions guarding its *guarded.choice*s and those of its components. These expressions must be of type BOOL. A simple *guarded.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 *guarded.choice* executes by executing its *process* part. An unready *conditional* executes exactly like STOP, and fails to terminate.

If a *specification* precedes a *choice*, the scope of the *specification* is the *choice*, but not the text of the *specification* itself.

Homogeneous choice

There are two constructions which express particular forms of conditional discrimination, on the value of a single expression and on the tag of a communication.

$$\begin{array}{lcl}
\textit{discrimination} & = & \textit{selection} \mid \textit{case.input} \\
\textit{selection} & = & \boxed{\texttt{CASE}}\ \textit{expr} \\
 & & \quad \{\ \textit{option}\ \} \\
\textit{option} & = & \begin{array}{c|c|l} \{_1 \boxed{\texttt{,}}\ \textit{expr}\ \} & \boxed{\texttt{ELSE}} & \textit{specification} \\ \quad \textit{process} & \quad\quad \textit{process} & \textit{option} \end{array}
\end{array}$$

The expression at the head of a *selection* is the selector, and must have an integer or byte type. The expressions listed in each *option* must be constants of the same type as the selector, and must be distinct from each other. At most one *option* in each *selection* may be indicated, by the keyword ELSE, to be selected in case no other *option* can be selected.

A *selection* executes by evaluating its selector and executing the *process* part of the *option* listing the constant with the same value. If the value of the selector does not correspond to any listed expression, the *process* part of the default *option* is executed. If the value of the selector does not correspond to any listed expression, and there is no default *option*, the *selection* stops.

If a *specification* precedes an *option*, the scope of the *specification* is the *option*, but not the text of the *specification* itself.

A *case.input* makes a discrimination on the *tag* received by input from a *channel*, which must have a discriminated protocol.

$$\begin{array}{lcl}
\textit{case.input} & = & \textit{channel}\,\boxed{\texttt{?}}\boxed{\texttt{CASE}} \\
 & & \quad \{\ \textit{variant}\ \} \\
\textit{variant} & = & \begin{array}{c|l} \textit{tagged.list} & \textit{specification} \\ \quad \textit{process} & \textit{variant} \end{array}
\end{array}$$

A *case.input* selects its *channel* and performs an input from that *channel*. The input performed is that described by the *tagged.list* which begins with the *tag* received at the beginning of the input. Each *tagged.list* must conform to the protocol of the *channel*, and the *tag* of each must be distinct from the others. If there is no *tagged.list* which begins with the *tag* received, the *case.input* stops.

If a *specification* precedes a *variant*, the scope of the *specification* is the *variant*, but not the text of the *specification* itself.

Parallel processes

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

$$
\mathit{parallel} \;=\; \begin{array}{l} \boxed{\texttt{PAR}} \\ \quad \{\mathit{process}\} \end{array} \;\Bigg|\; \begin{array}{l} \boxed{\texttt{PAR}}\,\mathit{replicator} \\ \quad \mathit{process} \end{array} \;\Bigg|\; \mathit{placed.parallel} \;\Bigg|\; \mathit{pri.parallel}
$$

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 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. It is invalid if there is a port which is used in two of the component processes.

It is invalid for two or more components of a *parallel* to select, using subscripting expressions containing variables, components or segments of a single array of variables or channels or ports, and to use those components or segments in a way which would be invalid were the segments and components not disjoint.

A *parallel* construct may be decorated with configuration directives to indicate that its components be executed on different processors.

$$
\begin{array}{lcl}
\mathit{placed.parallel} & = & \begin{array}{l} \boxed{\texttt{PLACED}}\boxed{\texttt{PAR}} \\ \quad \{\mathit{placement}\} \end{array} \;\Bigg|\; \begin{array}{l} \boxed{\texttt{PLACED}}\boxed{\texttt{PAR}}\,\mathit{replicator} \\ \quad \mathit{placement} \end{array} \\
\mathit{placement} & = & \mathit{singleton} \;\big|\; \mathit{placed.parallel} \\
\mathit{singleton} & = & \begin{array}{l} \boxed{\texttt{PROCESSOR}}\,\mathit{expr} \\ \quad \mathit{process} \end{array}
\end{array}
$$

Each *singleton* is executed by a processor selected, in a way determined by the implementation, by the value of its *expr* which is of type `INT`. Apart from this constraint, a placed *parallel* executes by the concurrent execution of its component processes, in the same way as the corresponding undecorated construct. Particular implementations may require or allow other annotations on a *singleton*, indicating for example the properties of the processor.

A placed *parallel* is invalid if it uses any variables, ports or timers declared in an enclosing scope.

In a process executed on a single processor, declared variables, channels, ports, timers, and arrays of these objects can be fixed at known locations in the store.

$$\mathit{allocation} \;=\; \boxed{\texttt{PLACE}}\,\mathit{name}\,\boxed{\texttt{AT}}\,\mathit{expr}\,\boxed{\texttt{:}}$$

The *expr* in an *allocation* must be of type `INT`. Its value is used as an index into a memory map which is an array of type `[]INT` that occupies, in an implementation-dependent way, the whole store of the processor on which the process including the *allocation* will be executed.

The *name* must be bound to a variable, channel, port, timer, or array of these, and the effect of the *allocation* is to ensure that the named object is at the same absolute store location as the selected component of the memory map. An *allocation* is invalid if it places an object at a location which does not implement that kind of object, or if it causes several variables, channels, ports or timers to occupy the same location.

A *parallel* construct which is to be executed on a single processor may be decorated with directives to indicate the relative priority of its components.

$$\mathit{pri.parallel} \;=\; \begin{array}{l|l} \boxed{\texttt{PRI}}\boxed{\texttt{PAR}} & \boxed{\texttt{PRI}}\boxed{\texttt{PAR}}\,\mathit{replicator} \\ \quad\{\,\mathit{process}\,\} & \quad\mathit{process} \end{array}$$

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 limit on the number of components in an asymmetric *parallel*.

Alternative processes

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.

$$\mathit{alternation} \;=\; \begin{array}{l|l|l} \boxed{\texttt{ALT}} & \boxed{\texttt{ALT}}\,\mathit{replicator} & \mathit{pri.alternation} \\ \quad\{\,\mathit{alternative}\,\} & \quad\mathit{alternative} & \end{array}$$

An *alternation* can be ready or not; it is ready if at least one of its *alternatives* is ready. A ready *alternation* executes by executing exactly one of its ready *alternatives*. An *alternation* which is not ready suspends its execution until it becomes ready, and then executes exactly one of its *alternatives* which became ready.

$$\begin{array}{lcl} \textit{alternative} & = & \textit{simple.alternative} \mid \textit{alternation} \mid \begin{array}{l}\textit{specification}\\ \textit{alternative}\end{array} \\ \textit{simple.alternative} & = & \textit{guarded.alternative} \mid \textit{case.alternative} \end{array}$$

If a *specification* precedes an *alternative*, the scope of the *specification* is the *alternative*, but not the text of the *specification* itself.

$$\begin{array}{lcl} \textit{guarded.alternative} & = & \begin{array}{l}\textit{guard}\\ \quad \textit{process}\end{array} \\ \textit{guard} & = & \boxed{\texttt{SKIP}} \mid \textit{expr}\,\boxed{\texttt{\&}}\,\boxed{\texttt{SKIP}} \mid \textit{input} \mid \textit{expr}\,\boxed{\texttt{\&}}\,\textit{input} \end{array}$$

A simple *guarded.alternative* is ready if its *guard* is ready, and executes by executing the process appearing in its *guard* and its guarded *process* in sequence.

When `SKIP` appears in a *guard*, it is ready immediately the *alternative* of which it is a part is executed. A *guard* which is a *delay* is ready when the *delay* can terminate. A *guard* which is an *input* from a channel is ready when the corresponding output process is executing, even if the *input* expects a *tag* which is not being offered. A *guard* which is an *input* of any other kind is ready immediately the *alternative* of which it is a part is executed.

If the process in a *guard* is preceded by an expression, the type of that expression must be `BOOL`, and the *guard* is ready only if the Boolean is true and the process part is ready. A symmetric *alternation* is not valid if it contains a `SKIP` guard not accompanied by a Boolean expression.

$$\textit{case.alternative} \;=\; \begin{array}{l}\textit{channel}\,\boxed{\texttt{?}}\,\boxed{\texttt{CASE}}\\ \quad \{\,\textit{variant}\,\}\end{array} \;\Bigg|\; \begin{array}{l}\textit{expr}\,\boxed{\texttt{\&}}\,\textit{channel}\,\boxed{\texttt{?}}\,\boxed{\texttt{CASE}}\\ \quad \{\,\textit{variant}\,\}\end{array}$$

An *alternative* which consists of a *case.input* is ready when the corresponding output is executing, even if the *tag* being offered does not correspond to any *variant*. If the *case.input* is preceded by an expression, the type of that expression must be `BOOL`, and the *guard* is ready only if the Boolean is true and the *case.input* is ready. A ready *case.alternative* executes by executing the corresponding *case.input*.

An *alternation* preceded by a `PRI` is asymmetric.

$$\textit{pri.alternation} \;=\; \begin{array}{l}\boxed{\texttt{PRI}}\,\boxed{\texttt{ALT}}\\ \quad \{\,\textit{alternative}\,\}\end{array} \;\Bigg|\; \begin{array}{l}\boxed{\texttt{PRI}}\,\boxed{\texttt{ALT}}\,\textit{replicator}\\ \quad \textit{alternative}\end{array}$$

An asymmetric *alternation* behaves like the corresponding symmetric construct, except that no *alternative* can be selected for execution which is later in the text than the first *alternative* of the *alternation* to become ready.

Unbounded loops

An unbounded loop consists of a Boolean expression and a process.

$$\begin{array}{lcl} \textit{loop} & = & \boxed{\texttt{WHILE}}\ \textit{expr} \\ & & \quad \textit{process} \end{array}$$

It executes as if it were

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

Procedure abstraction

A procedure definition binds a *name* to a process made parametric on a list of names.

$$\begin{array}{lcl} \textit{proc.definition} & = & \textit{proc.heading} \\ & & \quad \textit{process} \\ & & \boxed{\texttt{:}} \\ \textit{proc.heading} & = & \boxed{\texttt{PROC}}\ \textit{name}\ \boxed{\texttt{(}}\{_0 \boxed{\texttt{,}}\ \textit{proc.formals}\}\boxed{\texttt{)}} \end{array}$$

Names free in the *process* and which do not appear in the list of formal parameters are bound by the definitions in the scope of which the *proc.definition* occurs.

The procedure definition is invalid if its body uses for both input and output any channel which appears either as a free name of its body or as formal parameter.

An *instance* of such a procedure executes by executing the *process* in the context of additional bindings to the names of the formal parameters.

$$\textit{proc.formals} \;=\; \textit{specifier}\,\{_1 \boxed{\texttt{,}}\ \textit{name}\} \;\Big|\; \boxed{\texttt{VAL}}\,\textit{specifier}\,\{_1 \boxed{\texttt{,}}\ \textit{name}\}$$

A list of formal parameters with a single *specifier*

$$\textit{spec}\ \textit{name}_1,\ \textit{name}_2,\ \ldots,\ \textit{name}_N$$

is equivalent to

$$\textit{spec}'\ \textit{name}_1,\ \textit{spec}'\ \textit{name}_2,\ \ldots,\ \textit{spec}'\ \textit{name}_N$$

where *spec′* denotes the same *specifier* as *spec*, but contains no occurrences of the names; similarly

$\texttt{VAL}\ spec\ name_1,\ name_2,\ \ldots,\ name_N$

is equivalent to

$\texttt{VAL}\ spec'\ name_1,\ \texttt{VAL}\ spec'\ name_2,\ \ldots,\ \texttt{VAL}\ spec'\ name_N$

An *instance* of a *name* bound to a procedure must be accompanied by as many actual parameters as there are formal parameters to the procedure.

$$\begin{array}{lcl} instance & = & name\ \boxed{\texttt{(}}\{_0\boxed{\texttt{,}}\ proc.actual\,\}\boxed{\texttt{)}} \\ proc.actual & = & object \mid expr \end{array}$$

Its meaning is that of the body of the procedure preceded by a sequence of abbreviations, each binding the name of a formal parameter to the corresponding actual parameter. For the *instance* to be valid these abbreviations must be valid, and the process in the body of the procedure must be valid in the context of these abbreviations.

Function abstraction

A function definition binds a *name* to a list of expressions made parametric on a list of names.

$$\begin{array}{lcl} fun.definition & = & fun.heading\ \boxed{\texttt{IS}}\ expr.list\ \boxed{\texttt{:}} \;\Bigg|\; \begin{array}{l} fun.heading \\ \quad value.proc \\ \boxed{\texttt{:}} \end{array} \\ fun.heading & = & \{_1\boxed{\texttt{,}}\ base.type\,\}\ \boxed{\texttt{FUNCTION}}\ name\ \boxed{\texttt{(}}\{_0\boxed{\texttt{,}}\ fun.formals\,\}\boxed{\texttt{)}} \end{array}$$

Names free in the *value.proc* or *expr.list* and which do not appear in the list of formal parameters are bound by the definitions in the scope of which the *fun.definition* occurs. The form

fun.heading
  value.proc
:

is entirely equivalent to

fun.heading `IS` (*value.proc*
) :

A *fun.call* returns a list of values, the type of each value being the corresponding *base.type* in the *fun.heading*; the values being those of the *expr.list* or those returned by the *value.proc*, evaluated in the context of additional bindings to the names of the formal parameters.

$$fun.formals \;=\; \boxed{\texttt{VAL}}\, specifier \{_1 \boxed{\texttt{,}}\ name\}$$

A list of formal parameters with a single *specifier*

$\texttt{VAL}\ spec\ name_1,\ name_2,\ \ldots,\ name_N$

is equivalent to

$\texttt{VAL}\ spec'\ name_1,\ \texttt{VAL}\ spec'\ name_2,\ \ldots,\ \texttt{VAL}\ spec'\ name_N$

where *spec′* denotes the same *specifier* as *spec*, but contains no occurrences of the names.

$$fun.call \;=\; name\, \boxed{\texttt{(}} \{_0 \boxed{\texttt{,}}\ fun.actual\} \boxed{\texttt{)}}$$
$$fun.actual \;=\; expr$$

A *fun.call* of a *name* bound to a function must be accompanied by as many actual parameters as there are formal parameters to the function.

Its meaning is that of the body of the function, preceded by a sequence of value abbreviations, each binding the name of a formal parameter to the value of the corresponding actual parameter. For the *fun.call* to be valid these abbreviations must be valid, and the body of the function must be valid in the context of these abbreviations.

Index of syntactic classes

Roman face page numbers, like 229, refer to the BNF production which defines the class; references in an italic face, like *229*, are to informal definitions in the prose.