G.1 Syntax in context

The following tables present the syntax of occam 2. Each syntactic object appears in context. However, the following BNF should not be read in isolation. The syntactic objects are kept to a minimum, and must be considered in association with the semantic rules given in the definition. Thus, for example, the use of *primitive.type* and *type* in the syntax

simple.protocol = *primitive.type*`::[]`*type*

is clarified by the semantics which point out that the *primitive.type* must be an integer or byte type, and that *type* must be a data type.

G.1.1 Processes

```
process      =  SKIP | STOP
             |  action
             |  construction
             |  instance
action       =  assignment | input | output

assignment   =  variable := expression

input        =  channel ? variable
output       =  channel ! expression

assignment      =  variable.list := expression.list
variable.list   =  {1 , variable }
expression.list =  {1 , expression }
```

G.1.2 Construction

```
construction =  sequence | conditional | selection | loop
             |  parallel | alternation

sequence     =  SEQ
                  { process }

conditional     =  IF
                     { choice }
choice          =  guarded.choice | conditional
guarded.choice  =  boolean
                     process
boolean         =  expression

selection       =  CASE selector
                     { option }
option          =  {1 , case.expression }
                     process
                |  ELSE
                     process
selector        =  expression
case.expression =  expression

loop         =  WHILE boolean
                  process

parallel     =  PAR
                  { process }
```

```
alternation          =    ALT
                            { alternative }
alternative          =    guarded.alternative | alternation
guarded.alternative  =    guard
                            process
guard                =    input
                        |  boolean & input
                        |  boolean & SKIP
```

G.1.3 Replicator

```
sequence     =    SEQ replicator
                    process
conditional  =    IF replicator
                    choice
parallel     =    PAR replicator
                    process
alternation  =    ALT replicator
                    alternative
replicator   =    name = base FOR count
base         =    expression
count        =    expression
```

G.1.4 Types

```
type             =    primitive.type
                    |  array.type
primitive.type   =    CHAN OF protocol
                    |  TIMER
                    |  BOOL
                    |  BYTE
                    |  INT
                    |  INT16
                    |  INT32
                    |  INT64
                    |  REAL32
                    |  REAL64
array.type       =    [ expression ] type
```

G.1.5 Literal

```
literal      =    integer
                |  byte
                |  integer(type)
                |  byte(type)
                |  real(type)
                |  string
                |  TRUE | FALSE
integer      =    digits | #hex.digits
byte         =    'character'
real         =    digits.digits | digits.digitsEexponent
exponent     =    +digits | -digits
digit        =    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
hex.digit    =    digit | A | B | C | D | E | F
```

G.1.6 Declaration

```
declaration = type {1 , name } :
```

G.1.7 Protocol

```
definition = PROTOCOL name IS simple.protocol :
           | PROTOCOL name IS sequential.protocol :
protocol = name

simple.protocol = type
                | primitive.type: : [ ] type
input = channel ? input.item
input.item = variable
           | variable : : variable
output = channel ! output.item
output.item = expression
            | expression : : expression
protocol = simple.protocol

sequential.protocol = {1 ; simple.protocol }
input = channel ? {1 ; input.item }
output = channel ! {1 ; output.item }

definition = PROTOCOL name
               CASE
                   { tagged.protocol }
               :
tagged.protocol = tag
                | tag ; sequential.protocol
tag = name

output = channel ! tag
       | channel ! tag ; {1 ; output.item }

case.input = channel ? CASE
               { variant }
variant = tagged.list
            process
        | specification
          variant
tagged.list = tag
            | tag ; {1 ; input.item }
process = case.input
input = channel ? CASE tagged.list

alternative = channel ? CASE
                { variant }
            | boolean & channel ? CASE
                { variant }
```

G.1.8 Timer access

```
input = timer.input
      | delayed.input
timer.input = timer ? variable
delayed.input = timer ? AFTER expression
```

G.1.9 Element

element = *element* [*subscript*]
| [*element* **FROM** *subscript* **FOR** *subscript*]
| *name*
subscript = *expression*
variable = *element*
channel = *element*
timer = *element*

G.1.10 Expression

operand = *element*
| *literal*
| *table*
| (*expression*)
expression = *monadic.operator operand*
| *operand dyadic.operator operand*
| *conversion*
| *operand*
table = *table* [*subscript*]
| [{1 , *expression* }]
| [*table* **FROM** *subscript* **FOR** *count*]
expression = **MOSTPOS** *type*
| **MOSTNEG** *type*
conversion = *primitive.type operand*
| *primitive.type* **ROUND** *operand*
| *primitive.type* **TRUNC** *operand*

G.1.11 Abbreviation

abbreviation = *specifier name* **IS** *element* :
| *name* **IS** *element* :
| **VAL** *specifier name* **IS** *expression* :
| **VAL** *name* **IS** *expression* :
specifier = *primitive.type*
| [] *specifier*
| [*expression*] *specifier*

G.1.12 Scope

process = *specification*
process
choice = *specification*
choice
option = *specification*
option
alternative = *specification*
alternative
variant = *specification*
variant
valof = *specification*
valof
specification = *declaration* | *abbreviation* | *definition*

G.1.13 Procedure

```
definition      =   PROC name ( {0 , formal })
                      procedure.body
                    :
formal          =   specifier {1 , name }
                  | VAL specifier {1 , name }
procedure.body  =   process
instance        =   name ( {0 , actual } )
actual          =   element
                  | expression
```

G.1.14 Function

```
value.process   =   valof
valof           =   VALOF
                      process
                      RESULT expression.list
                  | specification
                    valof
operand         =   ( value.process
                    )
expression.list =   ( value.process
                    )

definition      =   {1 , primitive.type } FUNCTION name ( {0 , formal })
                    function.body
                    :
function.body   =   value.process
operand         =   name ( {0 , expression } )
expression.list =   name ( {0 , expression } )
definition      =   {1 , primitive.type } FUNCTION name ( {0 , formal }) IS expression.list :
```

G.1.15 Configuration

```
placedpar   =   PLACED PAR
                  { placedpar }
            |   PLACED PAR replicator
                  placedpar
            |   PROCESSOR expression
                  process
parallel    =   placedpar

parallel    =   PRI PAR
                  { process }
            |   PRI PAR replicator
                  process

alternation =   PRI ALT
                  { alternative }
            |   PRI ALT replicator
                  alternative

process     =   allocation
                process
allocation  =   PLACE name AT expression :

definition  =   specifier name RETYPES element :
            |   VAL specifier name RETYPES expression :

primitive.type  =   PORT OF type
port        =   element
input       =   port ? variable
output      =   port ! expression
protocol    =   ANY
```

G.2 Ordered syntax

The following tables present the syntax of occam with each syntactic object placed in alphabetical order.

```
abbreviation = specifier name IS element :
             | name IS element :
             | VAL specifier name IS expression :
             | VAL name IS expression :

action = assignment
       | input
       | output

actual = element
       | expression

allocation = PLACE name AT expression :

alternation = ALT
                { alternative }
            | ALT replicator
                alternative
            | PRI ALT
                { alternative }
            | PRI ALT replicator
                alternative

alternative = guarded.alternative | alternation
            | specification
              alternative
            | channel ? CASE
                { variant }
            | boolean & channel ? CASE
                { variant }

array.type = [ expression ] type

assignment = variable := expression
           | variable.list := expression.list

base = expression

boolean = expression

byte = 'character'

case.expression = expression

case.input = channel ? CASE
               { variant }

channel = element

choice = guarded.choice | conditional
       | specification
         choice
```

```
conditional = IF
                { choice }
            | IF replicator
                choice

construction = sequence | conditional | selection | loop
             | parallel | alternation

conversion = primitive.type operand
           | primitive.type ROUND operand
           | primitive.type TRUNC operand

count = expression

declaration = type {1 , name } :

definition = PROTOCOL name IS simple.protocol :
           | PROTOCOL name IS sequential.protocol :
           | PROTOCOL name
               CASE
                 { tagged.protocol }
             :
           | PROC name ( {0 , formal })
               procedure.body
             :
           | {1 , primitive.type } FUNCTION name ( {0 , formal })
               function.body
             :
           | {1 , primitive.type } FUNCTION name ( {0 , formal }) IS expression.list :
           | specifier name RETYPES element :
           | VAL specifier name RETYPES expression :

delayed.input = timer ? AFTER expression

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

element = element [ subscript ]
        | [ element FROM subscript FOR subscript ]
        | name

exponent = +digits | -digits

expression = monadic.operator operand
           | operand dyadic.operator operand
           | conversion
           | operand
           | MOSTPOS type | MOSTNEG type

expression.list = ( value.process
                  )
                | name ( {0 , expression } )
                | {1 , expression }

formal = specifier {1 , name }
       | VAL specifier {1 , name }

function.body = value.process
```

```
guard = input
      | boolean & input
      | boolean & SKIP

guarded.alternative = guard
                        process

guarded.choice = boolean
                   process

hex.digit = digit | A | B | C | D | E | F

input = channel ? variable
      | channel ? input.item
      | channel ? {1 ; input.item }
      | channel ? CASE tagged.list
      | timer.input
      | delayed.input
      | port ? variable

input.item = variable
           | variable : : variable

instance = name ( {0 , actual } )

integer = digits | #hex.digits

literal = integer
        | byte
        | integer (type)
        | byte (type)
        | real (type)
        | string
        | TRUE | FALSE

loop = WHILE boolean
         process

operand = element
        | literal
        | table
        | (expression)
        | ( value.process
          )
        | name ( {0 , expression } )

option = {1 , case.expression }
           process
       | ELSE
           process
       | specification
         option

output = channel ! expression
       | channel ! output.item
       | channel ! {1 ; output.item }
       | channel ! tag
       | channel ! tag ; {1 ; output.item }
       | port ! expression
```

```
output.item = expression
            | expression :: expression

parallel = PAR
             { process }
         | PAR replicator
             process
         | PRI PAR
             { process }
         | PRI PAR replicator
             process
         | placedpar

placedpar = PLACED PAR
              { placedpar }
          | PLACED PAR replicator
              placedpar
          | PROCESSOR expression
              process

port = element

primitive.type = CHAN OF protocol
               | TIMER
               | BOOL
               | BYTE
               | INT
               | INT16
               | INT32
               | INT64
               | REAL32
               | REAL64
               | PORT OF type

procedure.body = process

process = SKIP | STOP
        | action
        | construction
        | instance
        | case.input
        | specification
          process
        | allocation
          process

protocol = name
         | simple.protocol
         | ANY

real = digits.digits | digits.digitsEexponent

replicator = name = base FOR count

selection = CASE selector
              { option }

selector = expression
```

```
sequence = SEQ
             { process }
         | SEQ replicator
             process

sequential.protocol = {1 ; simple.protocol }

simple.protocol = type
                | primitive.type : : [ ] type

specification = declaration | abbreviation | definition

specifier = primitive.type
          | [ ] specifier
          | [ expression ] specifier

subscript = expression

table = table [ subscript ]
      | [ {1 , expression } ]
      | [ table FROM subscript FOR count ]

tag = name

tagged.list = tag
            | tag ; {1 ; input.item }

tagged.protocol = tag
                | tag ; sequential.protocol

timer = element

timer.input = timer ? variable

type = primitive.type
     | array.type

valof = VALOF
          process
          RESULT expression.list
      | specification
        valof

value.process = valof

variable = element

variable.list = {1 , variable }

variant = tagged.list
            process
        | specification
          variant
```