OCCAM 2: IDENTIFIERS AND PROCESSES

OCCAM 2 is case sensitive. Identifiers may be of any length, and may be composed of letters, digits and dots (periods). Examples of valid identifiers:

```
PACKET    LinkOut   NOT.A.NUMBER
vector6   port.in   transputer
fred      Fred      FRED      disCourageD
```

All keywords are in upper case and are reserved. Their list:

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

OCCAM 2 programs are built from processes. There are only five types of primitive processes. Larger processes are built by combining smaller processes into a construction. There are only six kinds of constructions:

PRIMITIVE		*CONSTRUCTIONS*	
Assignment		SEQ	WHILE
Input	SKIP	IF	PAR
Output	STOP	CASE	ALT

OCCAM 2: PRIMITIVE PROCESSES

Assignment:

```
x := y + 2
a, b, c := x, y + 1, z + 2
x, y := y, x  -- swap
```

Input:

```
keyboard ? char
```

Output:

```
screen ! char
```

SKIP Process (performs no action and terminates):

```
SEQ
  keyboard ? char
  SKIP
  screen ! char
```

STOP Process (performs no action and never terminates):

```
SEQ
  keyboard ? char
  STOP
  screen ! char
```


OCCAM 2: SEQ

Used to combine processes to be executed sequentially:

```
SEQ
  screen ! '?'
  keyboard ? char
  screen ! char
  screen ! cr
  screen ! lf
```

is equivalent to:

```
SEQ
  SEQ
    screen ! '?'
    keyboard ? char
  SEQ
    screen ! char
    screen ! cr
    screen ! lf
```

Replicated SEQ

```
SEQ i = 14 FOR 2
  stream ! data.array[i]
```

is equivalent to:

```
SEQ
  stream ! data.array[14]
  stream ! data.array[15]
```

Replication index, base and count must be of type INT. The index may be used in expressions, but cannot be assigned to. If count value is 0, the construction is equivalent to SKIP. A negative count value makes the construction **invalid.**

OCCAM 2: IF

Some IFs may act as STOP:

```
IF                    -- this IF may STOP
  x < y
    x := x + 1
```

but not this one:

```
IF                    -- this IF will never STOP
  x < y
    x := x + 1
  x > y
    x := x - 1
  TRUE
    SKIP
```

An IF construct may be replicated; its guarding Boolean expressions may be IFs too:

```
IF
  IF i = 1 FOR length
    string[i] <> object[i]
      found := FALSE
  TRUE
    found := TRUE
```

Replication index, base and count must be of the type INT. If count value is 0, the replicated IF construction is equivalent to STOP. A negative count value makes the construction **invalid.**

OCCAM 2: CASE

Consider some straightforward examples:

```
CASE direction
  up
    x := x + 1
  down
    x := x - 1
```

Above, the value of the expression `direction` *is being compared against the values of two expressions:* `up` *and* `down`.

Some CASE constructions may act as STOP, like this one:

```
CASE letter
  'a', 'e', 'i', 'o', 'u'
    vowel := TRUE
```

but not this one:

```
CASE letter
  'a', 'e', 'i', 'o', 'u'
    vowel := TRUE
  ELSE
    vowel := FALSE
```

NOTE: All case expressions used in a selection must have distinct constant values. The selector and the case expressions must be of the same data type, which may be either integer or byte data type. A selection can have only one ELSE option.

OCCAM 2: WHILE

All loops in OCCAM 2 are coded using the WHILE construction. Consider:

```
WHILE buffer <> eof
  SEQ
    in  ? buffer
    out ! buffer
```

Searching the array `string` *for a particular character* `char`*:*

```
SEQ
  pointer  := 0
  finished := FALSE
  found    := FALSE
  WHILE NOT finished
    IF
      string[pointer] <> char
        IF
          pointer < end.of.string
            pointer := pointer + 1
          pointer = end.of.string
            finished := TRUE
      string[pointer] = char
        SEQ
          found    := TRUE
          finished := TRUE
```

can be done simpler:

```
IF
  IF i = 0 FOR string.size
    string[i] = char
      found := TRUE
  TRUE
    found := FALSE
```


OCCAM 2: PAR

PARallel construct combines processes to be run concurrently.
Example: Editing a file may be seen as running three processes:

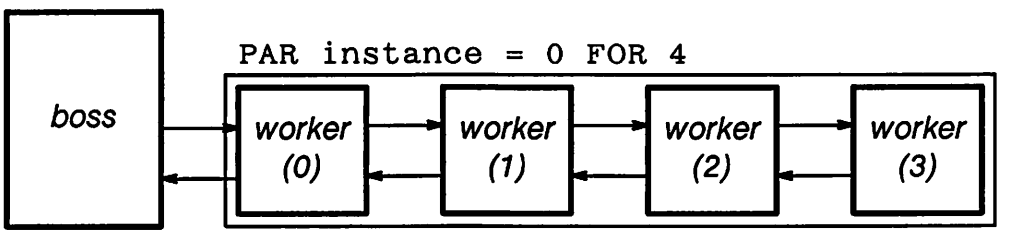
```
PAR
  editor (keyboard, screen)
  input  (keyboard) -- device driver
  output (screen)   -- device driver
```

Example: Concurrent buffering:

```
WHILE next <> eof
  SEQ
    buffer := next
    PAR
      in  ? next
      out ! buffer
```

PAR constructs can be replicated in the usual manner.
For example, we can create a farm of workers, viz.:

```
PAR
  boss()
  PAR instance = 0 FOR 4
    worker(instance)
```



OCCAM 2: PLACED PAR

PLACED PARallel construct is used to distribute programs across a number of processors. Its general format is:

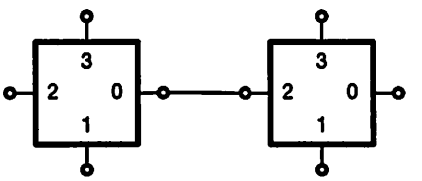
```
PLACED PAR
  PROCESSOR 1 <type>
    P1 -- place some code here
  PROCESSOR 2 <type>,
    P2 -- place some other code here
  ...
  PROCESSOR k <type>,
    Pk -- place some (different) code here
```

Example of use: Suppose we have a program:

```
CHAN OF INT Pipe:
PAR
  WHILE TRUE
    SEQ
      -- some actions including
      Pipe ! x -- for some data value x
  WHILE TRUE
    SEQ
      Pipe ? y
      -- some other actions
```


OCCAM 2: PLACED PAR
(continued)

Suppose further, that we have two transputers of types T800 and T425 plugged into sockets 3 and 7 on the motherboard. We want to connect them as follows:



Transputer 3 (T800) Transputer 7 (T425)

The code accomplishing this should look as follows:

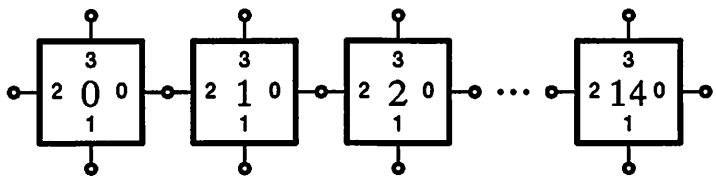
```
CHAN OF INT Pipe:
PLACED PAR
  PROCESSOR 3 T800
    PLACE Pipe AT Linkout0:
    WHILE TRUE
      SEQ
        --some actions including
        Pipe ! x -- for some data value x
  PROCESSOR 7 T425
    PLACE Pipe AT Linkin2:
    WHILE TRUE
      SEQ
        Pipe ? y
        -- some other actions
```


OCCAM 2: PLACED PAR
(continued)

In this example we create a pipeline of processes:

```
VAL INT NProcessors IS 15:
[NProcessors+1]CHAN OF INT Pipe:
PAR i = 0 FOR NProcessors
  process(Pipe[i], Pipe[i+1]
```

If this is to be mapped onto a series of connected transputers of type T800 (numbered 0, 1, ... , 14)



The code should be modified to read as follows:

```
VAL INT NProcessors IS 15:
[NProcessors+1]CHAN OF INT Pipe:
PLACED PAR i = 0 FOR NProcessors
  PROCESSOR i T800
    PLACE Pipe[i] AT Linkin2:
    PLACE Pipe[i+1] AT Linkout0:
    process(Pipe[i], Pipe[i+1]
```


OCCAM 2: PRI PAR

The component processes of a PARallel construct executing on a single processor may be assigned a priority of execution:

```
PRI PAR
  terminal (keyboard, screen)
  editor   (keyboard, screen)
```

Here the process `terminal` *will always be executed in preference to the process* `editor`*. In general, each process belonging to a* `PRI PAR` *construct executes at a separate priority, the first process having the higherst priority, the last process the lowest. Lower priority processes may only execute when all higher priority processes are unable to.*

Replication is also possible:

```
PRI PAR i = 0 FOR 8
  user(keyboard[i], screen[i])
```

The process with the highest index is executed at the lowest priority.

OCCAM 2: ALT

An alternation combines a number of processes guarded by inputs:

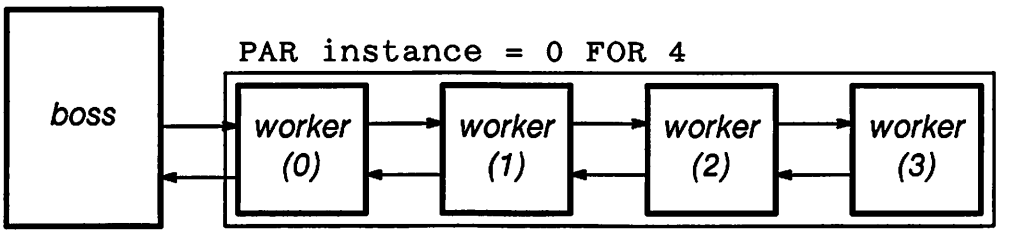
```
ALT
  left ? packet
    stream ! packet
  right ? packet
    stream ! packet
```

An alternation may wait selectively; it may also perform some default actions:

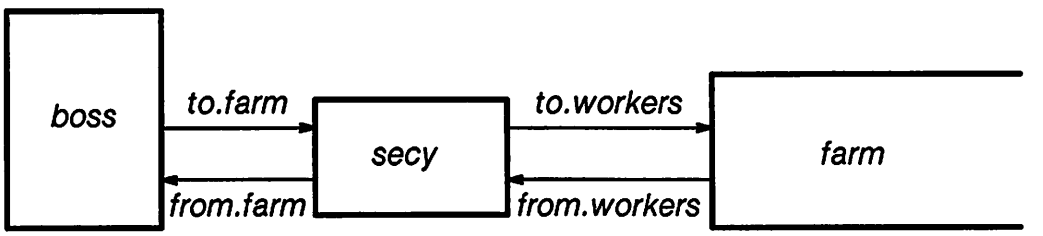
```
ALT
  enabled & left ? packet
    stream ! packet
  right ? packet
    stream ! packet
  sunday & SKIP
    stream ! null.packet
```


OCCAM 2: ALT (continued)

Remember the farm of workers?



Suppose we want to assign the boss a secretary, with the job of regulating the flow of work into the farm:



The code for the secretary would be:

```
SEQ
  idlers := number.of.workers
  WHILE running
    ALT
      from.workers ? result
      SEQ
        from.farm ! result
        idlers := idlers + 1
      (idlers >= 1) & to.farm ? packet
        to.workers ! packet
        idlers := idlers - 1
```


OCCAM 2: ALT
(continued)

The ALT construction can be replicated in the same way as sequences, conditionals and parallels. A replicated alternation constructs a number of similar alternatives.

Suppose that in our farm example, workers are farms themselves. It might then be practical to assign a secretary to each worker. The secretary's code might look like:

```
ALT
  ALT i = 0 FOR number.of.workers
    free.worker[i] & to.farm ? packet
      SEQ
        to.worker[i] ! packet
        free.worker[i] := FALSE
  ALT
    from.worker[i] ? result
      SEQ
        from.farm ! result
        free.worker[i] := TRUE
```


OCCAM 2: PRI ALT

The inputs guarding alternatives in an ALTernation construct may be assigned a selection priority. Consider:

```
PRI ALT
  disk ? block
    disk.driver (block)
  keyboard ? char
    keyboard.driver (char)
```

This construct will input data from the channel `disk` *in preference to the data from the channel* `keyboard`*, whenever both channels are ready.*

Another example:

```
PRI ALT
  stream ? data
    P()
  busy & SKIP
    Q()
```

This construct inputs `data` *whenever the channel* `stream` *is ready and then executes process* `P()`*, otherwise, if Boolean* `busy` *is* `true` *then the process* `Q()` *is executed.*

The replication rules for the PRI ALT construct are identical to those of the ALT construct.

