4 Formatted input and output

One of the things that you will probably miss in *occam* if you are used to programming in a typed high level language is the support for text input and output. There are usually either predefined routines, or language constructs, which take your program's data, such as strings, integers, floating point numbers, and translate them into sequences of characters for output to terminals and printers. Similarly, there are usually routines provided for reading sequences of digits, and interpreting them as numbers, and so on. It is almost always possible for you to write your own input and output routines, but those provided for you will usually do.

Since *occam* programs are, at least notionally, to run as 'stand alone' programs, there is no standard operating system or run-time library of such support routines, and the input and output translations must be performed by the program. Moreover, since there is no type information in the program, no standard routine can 'know' that you are interpreting a particular bit-pattern as a character code, or as a signed integer, or perhaps as a floating point number. Each program needs specific processes which translate those types of value whose text representations are input and output by that program.

This section introduces sequential input and output routines used in programs later in the book, and some others which you may find useful.

Output routines

A process for outputting the characters of a string appeared earlier, in the 'Programming structures'

```
PROC write.string(CHAN output, VALUE string[]) =
  -- Write the characters of the string[] to the output
  SEQ character.number = [1 FOR string[BYTE 0]]
    output ! string[BYTE character.number]        :
```

You will also probably need to output bit patterns as decimal numerals. If you have ever written this routine before, there ought to be no difficulty, except that an *occam* process cannot use recursion.

First of all, if *tens* is a power of ten then `'0'` + ((`n`/`tens`)\\`10`) is the digit of that weight in the numeral representing the positive integer n. Notice that `'0'` is just the character code for zero: in the addition it is treated as any other bit-pattern. The result becomes a character again only in as much as that you will choose to treat it as one by, for example, outputting it to a terminal.

To output the whole numeral for n the digit calculation must be performed for each power of ten not exceeding n, and the results are required in sequence in decreasing order of their weight.

```
VAR tens :
SEQ
  tens := 1
  WHILE (n / tens) >= 10
    tens := 10 * tens
  WHILE tens > 0
    SEQ
      output ! '0' + ((n / tens) \ 10)
      tens := tens / 10
```

The division of *tens* by ten always gives an exact answer, excepting the final occasion, when *tens* is one, and the result of the divison is zero. That process works for all positive n and, as a special case, for zero.

It is tempting to try outputting negative numbers by first changing the sign, but this is wrong, because changing the sign of the most negative number gives no defined result. Whatever the effect, it cannot possibly give the right answer, since that is not a representable value. The standard, if perhaps confusing, solution is to treat positive numbers as special cases which are best output by making them negative, or equivalently, to change the sign of *tens*, so that the result of dividing by *tens* is consistently negative.

```
PROC write.signed(CHAN output, VALUE n) =
  -- Write a signed decimal representation of n
  -- to the output channel
  VAR tens :
  SEQ
    IF
      n < 0
        SEQ
          output ! '-'
          tens :=  1
      n >= 0
        tens   := -1
    WHILE (n / tens) <= (-10)
      tens := 10 * tens
   WHILE tens <> 0
     SEQ
       output ! '0' - ((n / tens) \ 10)
       tens := tens / 10                         :
```

Notice that it is a matter of the definition of the division and remainder operators in *occam* that changing the sign either of *tens* or of n just changes the sign of the expression ((`n`/`tens`)\\`10`).

None of the expressions in the process have results outside the range of representable integers: for example, the result of the multiplication `10` $*$ `tens` in the first loop is guaranteed, by the condition on the loop, to be no further from zero than is n, so the multiplication gives the correct result. Similarly, the condition on that loop has to be written in that way, because calculating, say ($-$(`n`/`tens`))`>=10` might involve negating the most negative number.

As a final sophistication to this process, you might want to send leading spaces so that the numeral occupies a fixed number of character spaces, to simplify the laying out of columns of numbers. The simplest way of doing this is to count the digits whilst calculating the value of *tens*.

```
PROC write.signed(CHAN output, VALUE n, field.width) =
  ... Write a signed decimal representation of n
  ... to the output, right justified to occupy
  ... field.width character spaces
```

A coding of this process appears at the beginning of the collection of codes of the programs.

Input routines

Constructing a data object from its textual representation is slightly more difficult because, in general, not all sequences of characters will be legal representations. For example, a process to read a numeral might expect some spaces, perhaps a sign and some more spaces, and then a sequence of digits, followed by something else. If there are no digits, or if the number represented is too large to be encoded as a bit-pattern, then an error has occurred.

The particular action to be taken to recover from an error depends on the circumstances of the conversion: for example, whether the digits are being read from a terminal keyboard, or a magnetic tape, whether the process is running in a desk-top microcomputer or in aircraft auto-pilot equipment. For a general purpose routine, I will settle for returning a Boolean indication of whether the conversion was successful. (Other indications might be possible, for example, a communication on a channel laid aside for indicating errors.)

Ignoring, for the present, the matter of the sign, and the possibility of error, a sequence of digits can be converted into a bit-pattern representing the same number by

```
VAR ch :
SEQ
  n := 0
  input ? ch
  WHILE ('0' <= ch) AND (ch <= '9')
    SEQ
      n := (10 * n) + (ch - '0')
      input ? ch
```

The arithmetic is essentially dual to that in the output routine. If *ch* is the character code of a digit, then it lies between the codes of the characters zero and nine, and (`ch` – `'0'`) is its value as a digit.

As in the case of output, it is necessary to be careful with the most negative integer: it will not do to read negative numerals by reading the digits as if they were those of a positive numeral and then changing the sign of the result. The simplest solution to this problem is to change the sign of each digit before accumulation, keeping n negative throughout.

So as to check for overflow, the new value of n must be compared with either the most positive or the most negative bit-pattern, again being careful to keep all the arithmetic in the expressible range.

```
DEF min = NOT ((NOT 0) >> 1), max = (NOT 0) >> 1 :

IF
  (sign = '+') AND (n <= ((max - (ch - '0')) / 10))
    n := (10 * n) + (ch - '0')
  (sign = '-') AND (((min + (ch - '0')) / 10) <= n)
    n := (10 * n) - (ch - '0')
  otherwise
    ok := FALSE    -- an error has occurred
```

A possible solution to the problem of errors would be to omit the third branch of the conditional entirely, so that the routine would become deadlocked in case of an overflow. This more general solution postpones the decision, giving the caller of the process the option of ignoring the error, or of acting on it in any way he chooses, including the option of stopping.

The appendix contains a routine complementary to *write.signed* which has the following specification:

```
PROC read.signed(CHAN input, VAR n, ok) =
  ... Read an (optionally signed) decimal numeral from
  ... the input returning the corresponding value in n,
  ... and TRUE or FALSE in ok according as the
  ... conversion worked or not
```

In many programming languages a routine like *read.signed* could only be used for conversion of a numeral being read from a peripheral device. In PASCAL, for example, such a routine would be reading from a file, but an entirely different routine would be needed to convert a numeral stored in an array of characters. In *occam*, there is nothing to stop you doing this by putting input and output routines together in parallel. The process

```
CHAN internal :
PAR
  write.string(internal, "-137*C")
  read.signed(internal, n, ok)
```

sets n to -137. This might not look very useful for constant strings, but the same can be done with variable arrays of characters. This means, for example, that it is easy to separate the business of line construction, editing and echoing when reading from a terminal, from whatever data conversion you might want to perform on the input.

For completeness, the appendix also contains a coding of a line construction process suitable for input from a VDU

```
PROC read.line(CHAN keyboard, screen, VAR s[]) =
  ... Construct a string in s[] from the printable
  ... characters read from keyboard and echoed to
  ... screen. The string finishes at a carriage return.
```

As it reads characters from the *keyboard* stream, this process packs them into the byte array *s*[ ] and echoes them to the *screen* stream, allowing the usual sort of line editing. For example, typing backspace

```
SEQ
  keyboard ? ch
  IF
      ⋮
    (ch = backspace) AND (s[BYTE 0] > 0)
      SEQ
        screen ! backspace ; '*s' ; backspace
        s[BYTE 0] := s[BYTE 0] - 1
      ⋮
```

cancels the last character in the line, and removes its echo from the screen by writing a blank in its place.