Programming in 'occam'

A tourist guide to parallel programming

This note is intended as an introduction to a style of concurrent programming. The occam notation is described in enough detail to make this presentation self-contained. There then follow descriptions of a number of programs which make use of the novel structures possible in occam programs.

Programming Research Group
Technical Monograph PRG-43

March 1985

Geraint Jones
Oxford University Computing Laboratory
8-11, Keble Road
Oxford OX1 3QD

Programming Research Group,
8–11, Keble Road,
Oxford, *OX1 3QD*

Here, I saw, was a genuine innocent, writing odds and ends of verse about odds and ends of things, living quite out of the world in which such things are usually done, and knowing no better (or rather no worse) than to get his book made by the appropriate craftsman and hawk it around like any other ware.

Programming in occam

Over the past twenty years, theoretical computer scientists have expended a great deal of effort on the study of concurrency and synchronization. A great many mechanisms have been suggested for the taming of concurrency, such as semaphores, data monitors, condition queues, critical regions, remote procedure calls and rendezvous, even the disciplined use of shared store. There are, of course, at least as many programming language designs as there are programming language designers.

At the same time, the problems needing solution have become more numerous and difficult. The management of concurrency used to be a task to be undertaken on the large scale: the creation of new processes, and the synchronization and scheduling of their actions were once necessarily expensive in comparison to the real work done. These are conditions that seem to persist in the construction of operating systems and continent-wide communication networks, where processes are consequently long-lived, and interact as infrequently as can be contrived. Such are not the conditions experienced by programmers of multi-processor computers, where it becomes feasible to create large numbers of ephemeral processes, and desirable that they communicate frequently.

The occam language inherits the tradition of theoretical study, being more than reminiscent of recent work on the mathematics of synchronization. It is intended by its devisers as the 'assembly code' of the inmos transputer, a microprocessor designed to be used in relatively large numbers to make a single machine, yet capable of managing a large number of concurrent tasks within the one processor. The programmer is therefore encouraged to think of process creation, and of synchronisation and scheduling operations, as being as cheap as any other 'primitive' actions. Even on a conventional processor, neither process creation nor scheduling need be any more expensive than, say, procedure invocation. This startling scale of costs gives the programmer much greater freedom of expression, and leads to an unaccustomed programming style, which is the proper subject of this monograph.

It must be allowed that occam does have some compensating disadvantages: it shares many of the practical difficulties of other assembly level programming notations, and is certainly not the ideal means of expression for the programmer. Anyone who has used a modern programming language will miss the assistance of type-checking, and feel confined by the absence of implicitly handled recursion. The scope control in occam is no better than in any twenty year old language, and the fixed format of occam source texts is disconcerting unless you are blessed with a helpful text editor. Nevertheless, I know of no other programming notation that handles concurrent execution of tasks with the facility of occam.

I begin with a Cook's tour of occam which is sufficient to understand the examples that follow. A number of program fragments are then coded in occam and discussed in sufficient detail to give a flavour of the language in familiar and unfamiliar contexts. Finally, there are examples of real programs; they are, of course, not large programs but they were written as genuine engineering solutions to problems.

As far as will concern us here, an occam program is simply a process, which may have some free identifiers that have specific meanings dependent on the computer on which the program is to be run. A process describes some actions that are to be performed: that is, it is the expression of an algorithm. Each process may be either a primitive process, or a composite process consisting of a number of definitions and simpler component processes bound together by process constructors. The structure of constructed processes is indicated by a fixed layout of the source text, with each component appearing on a new line, slightly indented from the keyword that introduced the whole construction.

Processes that do nothing

The simplest of the primitive processes is skip, which is the process that does nothing at all. In many programming languages, you are obliged to write nothing (that is, not to write anything at all) if you want 'nothing' done; you will see later that skip serves as useful a purpose in occam as that of zero in the decimal notation for numbers.

The process stop also does 'nothing', but unlike skip it fails to terminate. You can think of it as being what happens when something goes wrong, like a deadlock, or some illegal machine operation. Nothing can happen in a sequential process after it has stopped, but things can happen in parallel with a stopped process. You might not expect to write the stop process very often in your programs, but it is the rational thing to do when something unexpected happens, because it ensures that the part of a network of processes that has failed is brought to a standstill without affecting other processes, at least until they come to depend on broken part.

It is also useful to have stop around so as to be able to describe the effect of compound processes that 'go wrong', for example by becoming deadlocked. A process is said to be deadlocked if there is nothing which it is able to do next, but it has not finished properly. Typically, a parallel program becomes deadlocked because each of its processes is waiting for one of the others to do something.

Sequential processes

In programs which execute sequentially, the work is done by assigning values to variables, and subsequently basing decisions on the values of those variables. The occam assignment has the form

```
variable := expression
```

Each expression has a value, as explained later, which is just a bit pattern of the size of a 'word' on your computer, and every variable is capable of storing a word sized bit pattern. Of course, there are operators which treat these bit patterns as if they represented numbers, or truth values, or characters, but no type difference is enforced by the language or its compilers.

A sequence of operations is described by writing them one under the other, under and slightly indented from the keyword seq. The sequence executes by executing each of its components in the order in which they are written. Of course, if there are no components at all, then the sequence does nothing, and behaves just like skip. Thus

```
seq
  x := 3
  y := x + 7
  x := x + 6
  z := x + 1
  x := (y + z) / 2
```

has the overall effect of setting each of the variables x, y, z to 10, albeit in a somewhat

perverse way: first x is set to three, then y to seven more than x, then x is changed to nine, z is set to one more than it, and finally x is set to the average of y and z, which is of course also ten.

Decisions based on the values of variables are made by a conditional process. This construction consists of the keyword if written above a list of components, each slightly indented. Each component is either another conditional nested within the first, or consists of an expression (the condition) and, below the condition and a little further indented, a process. The whole conditional executes by looking down the list of components, and the components of nested conditionals, until a condition is found whose value is true. If one is found then the corresponding process and only that process is executed, and the whole conditional terminates. It is an error for no condition to evaluate to true, for example if there are no components, so the conditional stops. As an example

```
if
  n < 0
    sign := -1
  n = 0
    sign := 0
  n > 0
    sign := 1
```

sets the value of the variable sign to one of minus one, zero, or plus one, according as the variable n contains a negative, zero, or positive value.

Since it is defined that the textually first of the processes corresponding to a true condition is selected, the process

```
if
  n = 0
    sign := 0
  n <> 0
    sign := -1
  true
    sign := 1
```

describes exactly the same effect as the former, but is much less clear. In general, it is good style to use constraints that describe precisely the conditions under which a process is to be executed.

Parallel processes

Just as a list of actions can be described as happening in a strict sequence, so in occam it is possible to specify that each of a list of actions is to happen, without specifying an order in which they must happen. Such a parallel composition is indicated by writing the actions one under the other, under and slightly indented from the keyword par. The parallel composition executes by executing each of its components, until each has terminated. Of course, if there are no components, the effect is the same as skip, and if any of the components fails to terminate, for example by stopping, then the composition cannot terminate. As an example

```
par
  x := y - 1
  z := y + 1
```

sets the values of x and z to be one less than, and one more than, respectively, the value of y. There is, however, no guarantee that the assignments will not happen in the other order, or at precisely the same moment. Because of this, for a parallel composition to be legal, none of its

components may change any variable which is used in any of the other branches: it would be wrong, for example, to try to write

```
par
  x := y - 1
  z := x + 2
```

because x is used in the second component, but changed in the first component.

If such mutual interference is not allowed, how then are concurrent processes to communicate? The answer is that they do so by input and output over channels. The output process

```
channel ! expression
```

sends the value of the expression over the channel. Similarly, the input process

```
channel ? variable
```

receives a value from the channel and stores it in the variable. Each of these communications waits for the other, so that an output does not happen until the corresponding input happens, and vice versa.

There is an abbreviation which is useful when complex data are to be transmitted, for example where many streams of data are multiplexed along one channel, and some identification must be sent with each item. A sequence of inputs (or of outputs) along one channel may be written as a single primitive process, separating the target variables (or expressions) by semicolons

```
channel ? variable.1; variable.2; ... variable.n
```

This has precisely the same meaning as

```
seq
  channel ? variable.1
  channel ? variable.2
    :
  channel ? variable.n
```

In combination, an input and an output behave just like an assignment, except that the expression and variable are in different, concurrently executing processes. In particular, the assignment

```
variable := expression
```

is exactly the same as

```
par
  channel ! expression
  channel ? variable
```

provided that it is legal to write the latter in the particular context. Just as there are rules about the use of variables in concurrently executing processes, so also each end (that is the input end and the output end) of a channel may be used only in one of the components of a par construction.

Decisions may also be distributed across several processes, using an alternative, which is similar to a conditional except in that the choice can depend on whether another process is performing an output. An alternative is written with the keyword alt above a list of components, each of

which is either an alternative, or consists of a guard with below that a process which is indented a little further. A guard may be an input process, or skip, or either of these simple guards preceded by an expression and an ampersand sign, as for example

```
alt
  red.selected & red ? x
    out ! x
  green.selected & green ? x
    out ! x
  not (red.selected or green.selected) & skip
    out ! default.value
```

An alternative waits until there is a guard which is 'ready'. An input guard becomes ready when a corresponding output is possible; a skip guard is always ready; and a guard preceded by an expression can only become ready if both the value of the expression is true, and the process part of the guard is ready. When some guard has become ready, one of the ready guards is selected and executed, followed by the corresponding process. After this the alternative terminates. At most one of the branches of an alternative is selected; if no guard ever becomes ready (for example, if there are no components in the alternative) then the whole alternative is deadlocked, like stop.

The example alternative above will accept an input from either the red or the green channel, provided that the value of the corresponding variable, red.selected or green.selected, is true, and having accepted that input into the variable x, it then re-outputs the value down the channel out. In case neither variable is true, only the third guard is ready, so the alternative sends the default.value and then terminates.

Data declarations

Each name used in an occam program must be declared before it can be used. There are declarations which allow you to give names to constant values, to variables, and to channels.

Constant definitions do not introduce any new objects into the program, they serve merely to give names to particular values. They are written with the keyword def followed by some definitions of the form

```
name = constant.expression
```

each definition separated from the next by a comma, and with a colon at the end. The effect is to allow you to use the name in the process that follows, wherever you want to refer to the value of the constant.expression. As an example, the declarations

```
def red = 0, red.and.amber = 1, green = 2, amber = 3 :
def first.state = red, last.state = amber :
```

might appear in a traffic light controller. It would then be possible to test, for example, whether

```
current.state = last.state
```

rather than comparing current.state for equality with 3, which would be somewhat more obscure.

Variables, as has already been indicated, are capable of storing bit patterns of some fixed size. They are declared by listing their names after the keyword var, above the process that will use them. The names are separated by commas, and there is a colon at the end of the list. The traffic light controller might well include, for example,

```
var current.state, queue.size :
```

Initially, a variable has no defined value, but once it has been set by an assignment or an input, its value is the last value that was stored into it. The usual place to find variable declarations is accordingly just before the keyword seq. Components of a sequence communicate with each other by one component leaving a value in a variable, and a later component reading that value. The traffic light controller might begin

```
def first.state = red, last.state = amber :
var current.state, queue.size :
seq
  current.state := first.state
  queue.size := 0
    :
```

Channels are declared in the same way as variables, except that the keyword is chan. Somewhere within the scope of each channel declaration there will be two concurrent processes, one of which sends output to the channel, the other of which takes input from it. It is therefore usual to find channel declarations immediately in front of par constructions.

Arrays

There is only one device for structuring data in occam: you may make one-dimensional arrays of constants, variables or channels. An array of variables is declared by giving a constant expression in brackets after the name.

```
name[count]
```

This indicates that a number, equal to the value of the count, of variables are to be declared, which can be referred to by indexing the name of the array with expressions whose values range from zero up to count-1. For example, in the scope of the definition

```
var a[137] :
```

there are declared 137 variables, called a[0], a[1], a[2], ... a[136], each variable independent of the others. Arrays of channels are similar.

An array of constants is called a table, and is denoted by an expression of the form

```
table [expression.zero, expression.one, expression.two, ..., expression.n]
```

Table expressions can be used with def declarations, or can appear to the left of an index, so that the value of

```
table [expression.zero, expression.one, expression.two, ...] [2]
```

is the same as the value of expression.two, for example.

In case the values of the components are always going to be in the range zero to 255, as for example when the values represent characters, occam allows you to specify that the values are to be packed one to a byte in the store of the machine. An array of byte variables is declared by adding the keyword byte after the opening bracket of the declarations; a byte table is denoted by including the keyword byte after the opening bracket of the denotation. A byte array of constants or variables is indexed by putting the keyword byte after the opening bracket of the index. For the present, the only reason for being concerned with byte arrays is that occam has a convenient denotation for byte arrays of characters: a string, which is written as a sequence of characters enclosed in double quotes,

```
"This is a string"
```

is a byte array, the first (zeroth) byte of which contains the length, and the subsequent bytes contain the characters in sequence. This string represents the same array of constants as does

```
table [byte 16, 'T', 'h', 'i', 's', '*s', 'i', 's', '*s', 'a', '*s', 's', 't', 'r', 'i', 'n', 'g']
```

A character denotes a constant bit-pattern like any other, corresponding to the ASCII code for that character, and the sequences '*s' represent the space character, that is 32. I could have written this as a space between quotes, ' ', but the asterisk form is clearer. There are asterisk-sequences for space (*s), carriage-return (*c), the newline character (*n), the quote characters (*' and *"), and of course for asterisk (**), which can be used either in character (single) quotes, or as elements of a string. In the scope of the declaration

```
def s = "This*sis*sa*sstring" :
```

(which is the same byte array as before) the value of s[byte 0] is 16, and the value of s[byte 16] is 103, which is the ASCII code of the character 'g'.

Process declarations

Names may be given to whole processes by means of proc declarations. These are introduced by a line of the form

```
proc name =
```

which is followed by the process to be called name, indented slightly, and terminated by a colon. The effect is that anywhere in the process that follows the declaration you can write name to mean the whole of the named process. (Of course, the named process is not in the scope of the declaration, so it is not possible to invoke it from within itself.)

A named process may have a list of fomal parameters included after its name in the declaration. The nature of each parameter is indicated by one of the keywords value, var, or chan, meaning a value (run-time constant), variable, or channel. You may omit the keyword in front of a second or subsequent parameter of a particular kind. Arrays should be indicated by writing an empty pair of brackets after the name of the array. The process is invoked by putting a corresponding list of actual parameters after the name at the point of call. The effect is the same as if the body of the named process had been written in place of the call, but with the actual parameters substituted for the formal parameters. Thus, for example

```
proc assign.character(var x, value s[], i) =
  x := s[byte i]                              :

var ch :
seq
  assign.character(ch, "This is a string", 13)
      :
```

has the effect of assigning the code of 'r', that is the 13th character of the string, to the variable ch, exactly as though the program had been written

```
var ch :
seq
  ch := "This is a string" [13]
      :
```

Loops and arrays of processes

There are two kinds of loops in occam: unbounded while loops, and indexed, bounded for loops.

Unbounded loops are necessarily sequential in occam, but there are many forms of for loop representing different forms of regular activity.

An unbounded loop is written with the keyword while, followed by an expression (the condition), with a process (the body) below and slightly indented from it. It executes by testing the value of the condition and then, provided that its value is true, executing the body. When the body has terminated, the condition is re-tested, so that the body is executed a number of times, in sequence, for as long as the condition remains true. The whole while loop terminates when the condition is tested and found to be false. As an example, the process

```
seq
  x := 0
  v[n] := key
  while  v[x] ≠ key
    x := x + 1
```

sets x to be the index of the first variable in the array v which contains the value key, by first posting a sentinel at v[n].

A bounded loop may be thought of as being an array of processes. Loops can be made from each of the alt, if, par and seq constructions, by putting a replicator of the form

```
name = [base for count]
```

after the keyword, and then writing a single component (of the kind appropriate to the construction) below and slightly indented from the keyword. The base and count are expressions, and the meaning of such a for loop is the same as that of a construction formed from the same keyword followed by count copies of the component with the name taking on the values base, base+1, ..., base+count−1 in successive copies.

A for loop stands for a repetition of the constructor with which it is made. In the same way that

$$\sum_{year = 1280}^{1341} f(year)$$

stands for the addition of sixty-two values,

$$f(1280) + f(1281) + \ldots + f(1341)$$

so too the seq-for loop

```
seq year = [1280 for 62]
  celebrate.christmas(year)
```

stands for the sequential composition of sixty-two processes

```
seq
  celebrate.christmas(1280)
  celebrate.christmas(1281)
          ⋮
  celebrate.christmas(1341)
```

Thus, seq-for loops are just like for loops in languages like algol or pascal; but that you may not assign to the loop index, and it is not declared outside the body of the loop. The bodies of par-for loops are executed concurrently, so such loops behave like arrays of parallel processes. The conditional loop, if-for, performs a bounded search, so for example the process

```
seq
  v[n] := key
  if i = [0 for n + 1]
    v[i] = key
      x := i
```

has precisely the same effect as that of the while loop above. The same search can be done, without the use of a sentinel, by writing

```
def otherwise = true, not.found = n :
if
  if i = [0 for n]
    v[i] = key
      x := i
  otherwise
    x := not.found
```

Here, x is set to not.found precisely when there is no occurrence of the value key in the array. Now you should see the reason for allowing conditionals as components of conditionals, and alternatives as components of alternatives!

Some dialects of occam allow you to construct for loops not only from constructed processes, but also from the primitive assignments, inputs and outputs. An expression of the form

```
name [base for count]
```

is called a slice, and denotes all of the variables (or values)

```
name[base], name[base+1], ... name[base+count-1]
```

Slices may be assigned

```
destination[destination.base for count] := source[source.base for count]
```

or they may be communicated

```
par
  channel ? destination[destination.base for count]
  channel ! source[source.base for count]
```

In each case, both of the slices concerned must be of the same size, and the effect is to set each of the count variables in destination, from destination.base upwards, to the values of the count variables in source, again, counting from source.base upwards. Do not confuse slice communications with the semicolon abbreviation for a sequence of communications: single slice operations must match with other slice operations; the semicolon denotes a sequence of unrelated communications.

There are, of course, also byte slices which are denoted by

```
name [byte base for count]
```

and which may be assigned or communicated into other byte slices.

Expressions

Every simple expression in occam denotes a bit pattern the size of the word on the computer which is executing the program. There is no defined precedence between the various operators,

so parentheses are generally needed to disambiguate an expression with more than one operator. The only exception to this rule is that associative operators do not need parentheses. All expressions are built of operators, constants, variables, and indexed arrays, so that evaluating an expression cannot possibly have a side-effect.

There are a number of operators that can best be explained by regarding each bit pattern as the twos complement representation of an integer:

a + b	is the sum of a and b
a − b	is the difference of a and b
a × b	is the product of a and b
a ÷ b	is the result of dividing a by b and rounding towards zero
a mod b	is the remainder on dividing a by b
− a	is a with its sign changed

The usual six relational operators:

a < b a ≤ b a = b a ≥ b a > b a ≠ b

compare their operands as though they represented signed integers, and return one of the values true or false. There is one other relational operator intended for comparing the values of a clock which counts cyclically through all possible word sized bit-patterns: the expression

a after b

is true or false according as a would be reached sooner by successively incrementing b, ignoring overflow, than by decrementing it. (Almost half of all bit patterns are after any given bit-pattern.)

The Boolean values can be manipulated by the following logical operators. Provided that each of a and b is either true or false

a and b	is true if both of a and b are true, and false otherwise
a or b	is true if either of a and b is true, and false otherwise
a ⊕ b	is true if exactly one of a and b is true, and false otherwise
not a	is true if a is false, and vice versa

The and and or operators evaluate their left argument first, and then the right argument, should it be needed to decide the result. This means that, for example, bounds checks should precede array accesses, thus

(0 ≤ i) and (i < size) and (a[i] = x)

Finally, there are some operators whose effect is most easily described by thinking of the operands as bit patterns

a ∨ b	is the bit by bit or of a and b
a ∧ b	is the bit by bit and of a and b
a ⊛ b	is the bit by bit modulo two sum of a and b
not a	is the ones (bit by bit) complement of a
a ≪ b	is the pattern a shifted left by b bits
a ≫ b	is the pattern a shifted right by b bits

The shift operators displace the pattern a by b number of bits, so that this many bits are lost from one end of the pattern, and the same number of zero bits are shifted in at the other end of the pattern.

Some of the most useful applications of the bit-manipulation operators are in idioms which achieve effects that in typed languages would use additional expression forms. Since the values true and false are, respectively, a word of one-bits and a word of zero-bits

```
(x ∧ true)  = x
(x ∧ false) = 0
(x ∨ 0)     = x
```

This means that you can write conditional expressions, such as

```
(a ∧ (p < q)) ∨ (b ∧ (p = q)) ∨ (c ∧ (p > q))
```

the value of which is one of a, b, or c according as p is less than, equal to, or greater than q.

Again, there is nothing to stop you constructing bit patterns with the bit manipulators, and then treating them as twos complement integers. The value of

```
(not 0) >> 1
```

is a word of one bits, excepting that the most significant bit, that is the sign bit, is zero; this value is therefore the largest positive integer in the range that can be represented on your particular machine, represented in a way that is independent of the word length of the machine.

```
not ((not 0) >> 1)
```

is a word of zero bits, excepting for a one in the sign bit, so is the most negative integer that can be represented. Another construction which I will often use is

```
not ((not 0) << logarithm)
```

which is the bit pattern in which the least significant logarithm number of bits are set. This is useful for making masks, so that, for example, in the scope of

```
def control = not ((not 0) << 5) :
```

the value of

```
control ∧ 'G'
```

is seven, that is the character code normally known as 'control-G'.

The representation of occam programs

This section describes the liberties that I have taken with the concrete syntax of occam. You will not need to read it to understand any of the rest of the document.

First of all, so as to making programs more readable, I have used a number of special symbols not in ISO-7 character sets. In the standard language, these are written using the following sequences of characters:

≤	as	<	=	less sign, equal sign
≥	as	>	=	greater sign, equal sign
≠	as	<	>	less sign, greater sign
⊕	as	>	<	greater sign, less sign
∧	as	/	\	slash, backslash
∨	as	\	/	backslash, slash
«	as	<	<	less sign, twice
»	as	>	>	greater sign, twice
—	as	-	-	minus sign, twice
:=	as	:	=	colon, equal sign
×	as	*		asterisk
÷	as	/		slash
mod	as	\		backslash
both " and "	as	"		the double-quote character
both ' and '	as	'		the single-quote character

I have written the keywords in bold face characters, whereas in the standard language they are written using the same characters as are used for identifiers, and so are reserved words. In some implementations of occam, the keywords are only reserved if written entirely in capitals, in other implementations, case is not significant.

The layout of programs is as described here, where 'slightly indented' is taken always to mean 'indented by a further two spaces'. Each line begins with an even number of spaces, two spaces indented from the line to which it is subordinate. Layout within the line is at your discretion, except that spaces are needed to punctuate sequences of letters and digits which might otherwise be misinterpreted, for example to distinguish

```
d[byte 0 for 3] := s[byte 0 for 3]
```

which is a slice assignment, from

```
d[byte0for3] := s[byte0for3]
```

which is a simple assignment. Additionally, long lines may be broken in any place where there is a sufficiently strong indication that the break is deliberate, such as after a comma in a list of declarations, or after an operator in an expression. Be warned that the language definition is unlikely to agree with your idea of what constitutes a sufficiently strong indication! The continuation line follows and must be indented at least as far as the first part of the line. Blank lines are ignored.

The first authoritative reference on the occam notation and its representation is the

occam Programming Manual, (authors) INMOS Limited
Prentice/Hall International, 1984

which defines a dialect called proto-occam. This has been variously modified, for example by recent occam programming system manuals, which I have followed. There are substantial differences between this language and the proposed occam 2.0.

In the next few pages, the various pipes and joints of occam are demonstrated in some small plumbing exercises. Although these examples may seem unrealistic or overly elaborate for their size, they are intended to show some practical programming techniques.

Simple sequential processes

It is almost traditional that the first program anyone writes in a new programming language is one that writes "Hello", or some equally imaginative greeting, to the screen of their terminal. In an occam environment, the terminal screen is likely to be accessible as a channel: values output to the channel being displayed on the screen as characters. A first, unexciting attempt at the "Hello" program is

```
seq
  terminal.screen ! 'H'
  terminal.screen ! 'e'
  terminal.screen ! 'l'
  terminal.screen ! 'l'
  terminal.screen ! 'o'
```

Looking for ways to generalize the program, we would naturally write a loop that outputs each of the characters of an occam string. Recall that a string is a byte array, with the number of characters being string[byte 0] so that a program to write string should behave like

```
seq
  output ! string[byte 1]
  output ! string[byte 2]
    :
  output ! string[byte string[byte 0]]
```

This is patently a candidate for a seq-for loop, which can be written

```
seq character.number = [1 for string[byte 0]]
  output ! string[byte character.number]
```

This process can now be packaged up as a named process, which corresponds to a procedure in a language such as pascal, for writing the characters of a string to a channel

```
proc write.string(chan output, value string[]) =
  -- Output the characters of the string along the channel output
  seq character.number = [1 for string[byte 0]]
    output ! string[byte character.number]          :
```

The line with the hyphen on it is a comment: these can appear at the end of any line in an occam program, even blank ones as here. Writing comments summarising the behaviour of named processes is probably a good habit to cultivate.

At the Programming Research Group, people have come to expect the computer to greet them with the shibboleth "Bootifrolo". This might be done by using write.string, as follows

```
proc write.string(chan output, value string[]) =
  -- Output the characters of the string along the channel output
  seq character.number = [1 for string[byte 0]]
    output ! string[byte character.number]          :
write.string(terminal.screen, "Bootifrolo")
```

Simple parallel processes

The simplest thing that you can usefully want a process to be doing, at the same time as another process is doing something else, is to copy data from one channel to another. This is just a matter of repeatedly taking input from one channel, storing it in a local variable, and tben sending the value of the variable along another channel.

```
var local :
seq
  source ? local
  sink ! local
```

Why you might possibly want this done should be apparent: the local variable acts as a buffer in the data stream passing along the two channels. This copying process can be packaged up as a named process that can be used to buffer any unbounded data stream passing along a channel between two processes

```
proc buffer(chan source, sink) =
  while true
    var local :
    seq
      source ? local
      sink ! local              :
```

Now whereas the producer and the consumer process are tightly synchronized in a program of the form

```
proc producer(chan output.stream) =
  while true
    var datum :
    seq
      ... calculate a new datum
      output.stream ! datum          :

proc consumer(chan input.stream) =
  while true
    var datum :
    seq
      input.stream ? datum
      ... calculate using the datum    :

chan data.stream :
par
  producer(data.stream)
  consumer(data.stream)
```

with neither able to get ahead of the other, by adding a buffer

```
chan data.from.producer, data.to.consumer :
par
  producer(data.from.producer)
  buffer(data.from.producer, data.to.consumer)
  consumer(data.to.consumer)
```

the two are slightly decoupled. The producer is now able to run up to one item of data ahead

of the consumer. ('Magic buffers' which would allow the consumer to run an item ahead of the producer are a mite more difficult to implement, even in occam.)

More buffering is easily provided by inserting more buffers in the data path, in a structure analogous to 'fall-through' first-in-first-out stores, where each item of data is passed along a chain until it reaches the last unoccupied location in the chain. Several items can be in independent 'free fall' at once if the buffer is fairly empty.

```
chan data.stream[number.of.buffers + 1] :
par
  producer(data.stream[0])
  par index = [0 for number.of.buffers]
    buffer(data.stream[index], data.stream[index + 1])
  consumer(data.stream[number.of.buffers])
```

There is, of course, nothing to stop you programming a buffer with an array of variables governed by one process, just as in any conventional programming language.

Synchronization by control signals

You could try putting the buffer process into the stream that goes to the terminal screen from the "Bootifrolo" program

```
chan internal.stream :
par
  write.string(internal.stream, "Bootifrolo")
  buffer(internal.stream, terminal.screen)
```

but this is not quite right. The buffer initially performs well, and copies all of the characters to the screen. Eventually, however, all of the string has been sent, and the write.string process terminates. This leaves the buffer in a somewhat undignified state, trying to perform an input on internal.stream when there will never again be a corresponding output. The program is deadlocked.

Some way is needed of telling the buffer that it should not expect any more input along its source channel, and that it should accordingly terminate. The process

```
proc copy.characters(chan source, end.of.source, sink) =
  -- Copy characters from source to sink
  -- until there is a signal on end.of.source
  var more.characters.expected :
  seq
    more.characters.expected := true
    while more.characters.expected
      var ch :
      alt
        source ? ch
          sink ! ch
        end.of.source ? any
          more.characters.expected := false    :
```

behaves just like the buffer, copying from source to sink, except that it may also take input from the channel end.of.source. The keyword any just means that the actual value received is immaterial, so it need not be stored in a variable. When an input signal is received on end.of.source, the variable more.characters.expected is set false, so the while loop terminates.

The right way of buffering the output of write.string is therefore to send its output to a copy.characters process, and to send a termination signal after the whole string has been sent.

Since the value received as a termination signal is ignored, it does not matter what is sent: outputting any has the effect of sending something unspecified.

```
chan internal.stream, end.of.internal.stream :
par

  proc write.string(chan output, value string[]) =
    ... Output the characters of the string along the channel output
  seq
    write.string(internal.stream, "Bootifrolo")
    end.of.internal.stream ! any


  proc copy.characters(chan source, end.of.source, sink) =
    ... Copy characters from source to sink until a signal on end.of.source
  copy.characters(internal.stream, end.of.internal.stream, terminal.screen)
```

The ... are not a part of the syntax of occam: they are just meant to save you the trouble of re-reading the code that they save me the trouble of re-writing!

In the case of the copy.characters process, there is no need to use a signal on an extra channel, because there is spare capacity on the source channel going in the right direction. You could select some value, say

```
def end.of.stream = -1
```

which is not a possible character value, and send that after the last real character of the message

```
def end.of.stream = -1 :
chan internal.stream :
par

  proc write.string(chan output, value string[]) =
    ... Output the characters of the string along the channel output
  seq
    write.string(internal.stream, "Bootifrolo")
    internal.stream ! end.of.stream

  proc copy.characters(chan source, sink) =
    -- Copy characters from source to sink until end.of.stream received
    var more.characters.expected :
    seq
      more.characters.expected := true
      while more.characters.expected
        var ch :
        seq
          source ? ch
          if
            ch <> end.of.stream
              sink ! ch
            ch = end.of.stream
              more.characters.expected := false    :

  copy.characters(internal.stream, terminal.screen)
```

In this particular case, there is little to choose between the two styles: the latter program may be marginally more efficient.

In many cases there will be no convenient data-stream going in the right direction. The example of a circular buffer implemented using an array of variables is of this kind. Assume that the array is declared by

```
var datum[size] :
```

and that the variables

```
var reader, writer :
```

have values in the range zero to size−1, so that the oldest value to leave the buffer will be found at datum[reader], and the next to enter the buffer will be written to datum[writer]. It will be convenient to keep track of the number of unoccupied locations in the buffer by a further variable

```
var count :
```

whose value ranges from zero, for a full buffer, to size for an empty one.

There are two activities in which the buffer must be able to participate: provided that it is not full, that is that count>0, it must be possible to add another value to the buffer

```
seq
  source ? datum[writer]
  writer := (writer + 1) mod size
  count  := count - 1
```

and provided that count<size it must be possible for the oldest value to be read from the buffer

```
seq
  sink ! datum[reader]
  reader := (reader + 1) mod size
  count  := count + 1
```

The buffer must allow the producing and consuming processes to control its activity, selecting between writing and reading, provided only that there is room to write, or something to read, respectively. It is tempting to try writing

```
alt
  count > 0   &   source ? datum[writer]
    seq
      writer := (writer + 1) mod size
      count  := count - 1
  count < size   &   sink ! datum[reader]
    seq
      reader := (reader + 1) mod size
      count  := count + 1
```

but output processes cannot be used to guard alternatives. The solution to this problem is to have a control signal from the consuming process indicating that it is ready to accept an input from sink. There is no need for the corresponding request before a write to the buffer, because the input along source serves perfectly well in place of a control signal.

```
proc circular.buffer(chan source, request, sink) =
  -- Copy from source to sink, buffering up to size items.
  -- A signal is required on request before each item is read.
  var reader, writer, count, datum[size] :
  seq
    reader := 0
    writer := 0
    count  := size
    while true
      alt
        count > 0   &   source ? datum[writer]
          seq
            writer := (writer + 1) mod size
            count  := count - 1
        count < size   &   request ? any
          seq
            sink ! datum[reader]
            reader := (reader + 1) mod size
            count  := count + 1                    :
```

It is the responsibility of the consumer, whenever it reads from the buffer, to perform two communications in sequence

```
seq
  request ! any
  source ? ...
```

This burden can be removed by the consumer at the expense of an extra process, executing concurrently with the circular buffer

```
proc multiple.buffer(chan source, sink) =
  -- Copy from source to sink, buffering up to size + 1 items.
  chan request, data :
  par
    circular.buffer(source, request, data)

    while true
      var datum :
      seq
        request ! any
        data ? datum
        sink ! datum                    :
```

The resulting multiple.buffer process has a behaviour which is indistinguishable from that of a chain of size+1 single-item buffer processes acting in parallel.

Processes that evaluate expressions

Suppose now that you have a need to calculate the parity of the characters that are being sent to the terminal. (The parity of a character is an indication of whether there is an even or an odd number of 'one' bits in the binary representation of its code.) In a language like pascal, you might write a function to calculate the parity of a character which was given to it as a parameter, but in occam (there being no 'function's) the natural construction is a named process

```
proc calculate.parity(value ch, var parity)
```

which returns the result by way of a var parameter. A representation will be necessary for parity values: the natural thing to do is to choose the truth values

```
def even = true, odd = not even :
```

In fact, the process will be independent of the actual bit pattern chosen to represent even.

Calculating the parity of ch involves considering each bit of ch: the simplest thing to do is to take them one at a time in sequence. (Expert bit twiddlers may care to code an algorithm logarithmic in the number of bits in the character.) The expression

```
ch ∧ (1 ≪ bit.number)
```

is either zero or not according as the bit.number'th bit of ch, counting from zero at the least significant end, is zero or not. The loop

```
seq bit.number = [0 for number.of.bits.in.character]
  if
    (ch ∧ (1 ≪ bit.number)) = 0
      skip
    (ch ∧ (1 ≪ bit.number)) ≠ 0
      parity := not parity
```

complements the value of parity as often as there are 'one' bits in ch. If parity is initialized to even, then its final value indicates the parity of ch.

```
seq
  parity := even
  seq bit.number = [0 for number.of.bits.in.character]
    if
      (ch ∧ (1 ≪ bit.number)) = 0
        skip
      (ch ∧ (1 ≪ bit.number)) ≠ 0
        parity := not parity
```

Since exclusive-or behaves like a conditional complement operation, the conditional process in the middle of this seq-for loop can be abbreviated to a simple assignment which has the same effect

```
parity := parity ⊕ ((ch ∧ (1 ≪ bit.number)) ≠ 0)
```

and the whole can be packaged as a reasonable implementation of the named process calculate.parity

```
def even = true, odd = not even :

proc calculate.parity(value ch, var parity) =
  -- Return the parity of ch in parity
  def number.of.bits.in.character = 8 :
  seq
    parity := even
    seq bit.number = [0 for number.of.bits.in.character]
      parity := parity ⊕ ((ch ∧ (1 ≪ bit.number)) ≠ 0)    :
```

Using parallelism as a tool for program modularity

If for some reason you wanted to modify the write.string process so that it wrote only the even parity characters from its argument, ignoring the rest, you could write

```
proc write.even.parity.string(chan output, value string[]) =
  seq character.number = [1 for string[byte 0]] var parity :
    seq
      calculate.parity(string[byte character.number], parity)
        if
          parity = even
            output ! string[byte character.number]
          parity = odd
            skip                                          :
```

This process is perhaps a little specialized: it performs its task well enough, but there are no recognizable separate components performing subtasks, which you might be able to use again in other programs. The code for selecting characters according to their parity is mixed in with the code for turning a string into a sequence of output processes. A more modular program might use a process which splits a stream of characters into two streams according to their parities.

```
proc divide.on.parity(chan source, end.of.source, even.sink, odd.sink) =
  -- Copy the even parity characters from source to even.sink, odd
  -- parity characters to odd.sink, until signalled on end.of.source
  var more.characters.expected :
  seq
    more.characters.expected := true
    while more.characters.expected
      var ch :
      alt
        source ? ch
          var parity :
          seq
            calculate.parity(ch, parity)
            if
              parity = even
                even.sink ! any
              parity = odd
                odd.sink ! any
        end.of.source ? any
          more.characters.expected := false                :
```

This process is not specialized to the application in hand, but can be used to filter out the odd or even parity character codes from any data stream. The unwanted stream must be discarded

```
proc consume(chan source, end.of.source) =
  var more.characters.expected :
  seq
    more.characters.expected := true
    while more.characters.expected
      alt
        source ? any
          skip
        end.of.source ? any
          more.characters.expected := false   :
```

Using divide.on.parity the "Bootifrolo" program might be written

```
proc write.string(chan output, value string[]) =
  ... Output the characters of string along the channel output
proc divide.on.parity(chan source, end.of.source, even.sink, odd.sink) =
  ... Copy characters from source to even.sink or odd.sink
proc consume(chan source, end.of.source) =
  ... discard characters from source, until end.of.source

chan both.parities, end.of.both.parities, odd.parity, end.of.odd.parity :
par
  seq
    write.string(both.parities, "Booting from Floppy")
    end.of.both.parities ! any
  seq
    divide.on.parity(both.parities, end.of.both.parities, terminal.screen, odd.parity)
    end.of.odd.parity ! any
  consume(odd.parity, end.of.odd.parity)
```

In this particular case, the gain in modularity may not seem adequate to justify the expense, both in programming effort and execution time. The advantage is clearer in cases where the program must perform a number of tasks each of which divides its input data into chunks, and where the boundaries of these components do not coincide.

Using parallelism to resolve structure clash

A structure clash happens whenever a program must perform operations on data that must be divided into mutually overlapping components. In a text processing program, for example, it may prove necessary to do something to every line of a document, and something else to every sentence. The natural way to code each of these tasks, individually, is to write programs whose structure reflects the structure of the document. To perform an action on every line:

```
while ... there are more lines
  seq
    ... read a line
    ... process the line
```

and to perform an action on every sentence:

```
while ... there are more sentences
  seq
    ... read a sentence
    ... process the sentence
```

Since sentences do not need to contain only complete lines, nor lines complete sentences, it is impossible to combine these two programs into a single sequential program. The somewhat unsatisfactory best that can be done in a sequential program is to treat the document as a sequence of words, these being the largest common sub-components of both lines and sentences.

```
while ... there are more words
  seq
    ... read a word
    ... if this completes a line process the line
    ... if this completes a sentence process the sentence
```

In a parallel program, the structure of both component processes can be retained by performing the two divisions of the document concurrently

```
chan lines, sentences :
par
  ... copy the document to lines and sentences

  while ... there is more document
    seq
      ... read a line from lines
      ... process the line

  while ... there is more document
    seq
      ... read a sentence from sentences
      ... process the sentence
```

The simplest case of a structure clash arises from attempting to pack data into fixed sized blocks that will not accommodate an exact whole number of items. It might be necessary, for example, to pack a stream of characters into half-kilobyte blocks for transmission or storage on a medium which accepts only such blocks. Consider first a case in which there is no structure clash: the medium is represented as a channel that accepts only slice outputs of half a kilobyte, and characters are represented by codes in the range from 0 to 255, so that a whole number of characters exactly fill a block.

The way to perform actions sequentially on the components of an array of bytes declared by

```
var buffer[byte bytes.in.a.block] :
```

is to use a sequential 'array' of processes created by the constructor

```
seq byte.number = [0 for bytes.in.a.block]
```

so this packing might be done by a process of the form

```
proc pack.bytes.into.blocks(chan byte.source, end.of.source, block.sink) =
  var more.bytes.expected :
  seq
    more.bytes.expected := true
    while more.bytes.expected
      var buffer[byte bytes.in.a.block] :
      seq
        seq byte.number = [0 for bytes.in.a.block]
          alt
            more.bytes.expected & byte.source ? buffer[byte byte.number]
              skip
            more.bytes.expected & end.of.source ? any
              more.bytes.expected := false
            not more.bytes.expected & skip
              skip
      block.sink ! buffer[byte 0 for bytes.in.a.block]                    :
```

The branch of the alternative that does all the work is the first, that guarded by an input from byte.source which inputs the next byte into the particular component of the buffer which is being considered. Since the guard does all the work, there is nothing left to be done in the

guarded process, so this is skip. Notice the use of a skip guard in the alternative inside this for loop: the condition ensures that this guard is ready when and only when there are no more bytes to be packed into the last block.

This process always sends a partly or completely empty block as its last output. The sending of a completely empty block could be prevented by looking ahead for the next byte

```
def bytes.in.a.block = 512 :

proc pack.bytes.into.blocks(chan byte.source, end.of.source, block.sink) =
  -- Copy data from byte.source to block.sink in complete blocks
  -- until there is a signal on end.of.source
  var next.byte :
  alt
    byte.source ? next.byte                -- Read ahead the first byte
      var more.bytes.to.pack :
      seq
        more.bytes.to.pack := true
        while more.bytes.to.pack
          var buffer[byte bytes.in.a.block + 1] :
          seq
            buffer[byte 0] := next.byte
            seq byte.number = [1 for bytes.in.a.block]
              alt
                more.bytes.to.pack & byte.source ? buffer[byte byte.number]
                  skip
                more.bytes.to.pack & end.of.source ? any
                  more.bytes.to.pack := false
                not more.bytes.to.pack & skip
                  skip
            block.sink ! buffer[byte 0 for bytes.in.a.block]
            next.byte := buffer[byte bytes.in.a.block]

    end.of.source ? any                    -- No bytes at all
      skip                                                    :
```

Even so, in case the entire message does not exactly fill a whole number of blocks, it has to be possible for a process that unpacks the characters from the blocks to deduce from those characters that it has reached the actual end of the character stream before the end of the last block.

Now consider the problem of trying to achieve a higher packing density, given that only character codes less than 128 are going to be sent, so that seven bits will suffice rather than eight. Seven bit values will not fit neatly into bytes, nor into half-kilobyte blocks. The problem can, however, be decomposed into two simpler separate problems in which there is no structure clash: turning seven bit character values into a sequence of bits, and packing a sequence of bits into blocks.

The packing of bits into blocks can be done in almost exactly the same way as that suggested for packing bytes into blocks. A byte can be considered to be an array of bits, indexed by using the bit-pattern manipulating operations. The assignment

```
buffer[byte byte.number] := buffer[byte byte.number] ∧ (not (1 << bit.number))
```

sets the bit.number'th bit of the byte.number'th byte of buffer to zero, whilst

```
buffer[byte byte.number] := buffer[byte byte.number] ∨ (1 << bit.number)
```

sets that same bit to one, so the conditional

```
if
  bit = 0
    buffer[byte byte.number] := buffer[byte byte.number] ∧ (not (1 ≪ bit.number))
  bit = 1
    buffer[byte byte.number] := buffer[byte byte.number] ∨ (1 ≪ bit.number)
```

stores the given bit in the bit.number'th bit of the byte.number'th byte of the buffer. The buffer is, in effect being treated as a two-dimensional array, and the process that packs the buffer is a two-dimensional seq-for array of processes.

```
def bits.in.a.byte = 8, bytes.in.a.block = 512 :

proc pack.bits.into.blocks(chan bit.source, end.of.source, block.sink) =
  -- Copy data from bit.source to block.sink in complete blocks
  -- until there is a signal on end.of.source
  var next.bit :
  alt
    bit.source ? next.bit                          -- Read ahead the first bit
      var more.bits.to.pack :
      seq
        more.bits.to.pack := true
        while more.bits.to.pack
          var buffer[byte bytes.in.a.block] :
          seq
            seq byte.number = [0 for bits.in.a.block]
              if
                more.bits.to.pack
                  seq bit.number = [0 for bits.in.a.byte]
                    if
                      more.bits.to.pack
                        seq
                          if
                            next.bit = 0
                              buffer[byte byte.number] := buffer[byte byte.number] ∧
                                                          (not (1 ≪ bit.number))
                            next.bit = 1
                              buffer[byte byte.number] := buffer[byte byte.number] ∨
                                                          (1 ≪ bit.number)
                          alt
                            bit.source ? next.bit
                              skip
                            end.of.source ? any
                              more.bits.to.pack := false
                      not more.bits.to.pack
                        skip
                not more.bits.to.pack & skip
                  skip
            block.sink ! buffer[byte 0 for bytes.in.a.block]
    end.of.source ? any                            -- No bits at all
      skip                                                    :
```

Turning seven bit characters into a sequence of bits is also a simple task, since there is again

no structure clash. The value of the expression

```
(character >> bit.number) /\ 1
```

is zero or one according to the value of the bit.number'th bit of the value of character, so the character code can be treated as though it were an array of seven bits. The following process turns a stream of characters into a stream of the bits which make up their codes, least significant bit of the character first.

```
def bits.in.a.character = 7 :

proc unpack.bits.from.characters(chan char.source, end.of.source, bit.sink) =
  -- Copy characters from char.source to bit.sink, a bit at a time
  -- until there is a signal on end.of.source
  var more.characters.expected :
  seq
    more.characters.expected := true
    while more.characters.expected
      var character :
      alt
        char.source ? character
          seq bit.number = [0 for bits.in.a.character]
            bit.sink ! (character >> bit.number) /\ 1
        end.of.source ? any
          more.characters.expected := false                    :
```

The task of packing seven bit characters into half-kilobyte blocks is now easily done by performing each of these subtasks in parallel

```
def bits.in.a.character = 7, bits.in.a.byte = 8, bytes.in.a.block = 512 :

proc pack.characters.into.blocks(chan char.source, end.of.source, block.sink) =

  proc unpack.bits.from.characters(chan char.source, end.of.source, bit.sink) =
    ... Send the bits of character codes from char.source along bit.sink

  proc pack.bits.into.blocks(chan bit.source, end.of.source, block.sink) =
    ... Pack bits from bit.source into blocks sent along block.sink

  chan bit.stream, end.of.bit.stream :
  par
    seq
      unpack.bits.from.characters(char.source, end.of.source, bit.stream)
      end.of.bit.stream ! any

    pack.bits.into.blocks(bit.stream, end.of.bit.stream, block.sink)    :
```

Substantially the same program structure can clearly be used to turn the stream of blocks back into a stream of seven bit character codes, since that is just another, similar packing problem. The solution to each packing problem is of one of the three forms that I have shown here: grouping small objects to make larger ones; dividing large objects to make small ones; or a problem in which a structure clash requires that both the input data and the output data be divided into common subcomponents.

Local time

There are many applications of programmed devices where it is necessary for the program to be able to refer to, or to measure, the passage of time: for example, in long-range communication, the participants are usually prepared to wait for replies for a limited time only, before taking action to recover from the loss of messages. To accommodate these needs, there are two primitive processes by which occam programs may refer to the changing state of a local clock. I mention them here to complete the presentation, but they will hardly be used in the programs which follow: you may want to pass by this section on a first reading.

The clock reading process

```
time ? variable
```

sets the value of the variable to the current reading on the clock. This is a word-sized bit pattern which changes at a uniform implementation-dependent rate with the passage of time. It counts up cyclically through a set of values distributed through the whole range of bit patterns, the most negative reading following after the most positive one. Notice that it is misleading for this process to look like an input process: the sequence of characters time ? is indivisible, time is not a channel, nor are clock reading processes governed by the rules that control the legal uses of channels: many concurrent processes may legally read the time from the same clock.

The clock delay process

```
time ? after expression
```

is another process that does nothing, like skip, except that it suspends execution. It does not terminate until the reading on the clock has satisfied the condition

```
reading after expression
```

I have been careful with the wording of that last sentence: notice that there can be no guarantee about the value of variable after the execution of

```
seq
  time ? after expression
  time ? variable
```

As before, the sequence of characters time ? after is atomic, a delay is not an input process, but it is allowed to stand in the place of an input process as a guard of an alternative. Such a guard becomes ready as soon as the delay process may terminate.

The operator after is intended for the comparison of readings taken from the clock. Provided that two times are separated by less than half the time that it takes for the clock to count around the complete cycle of its readings, one time is after the other if readings taken from the clock at those times are similarly related by after. The cycle time of the clock depends on the word size, on the amount by which the reading is incremented at each tick, and the frequency of the clock ticks. Each of these depends on the particular implementation, and I will assume that you can supply a definition

```
def second = ...
```

in any program that needs it, indicating by how much the reading changes in one second. (This assumption will be unjustified if the clock cycle time is two seconds or less, as will be the case for some proposed transputer devices.)

Any two readings being compared, either directly, or by the delay process, should be taken from the same clock: the language does not guarantee any relationship between readings taken in

different branches of a par construct. Notionally, the clock is a register on a transputer, and no connection is to be expected between that register and the registers of any other transputers participating in the execution of a program. There is no mechanism in the language which maintains a global time, and it is the programmer's responsibility to implement one if it is needed. Similarly, if needed it is the programmer's task to provide a mechanism, using the clock, for timing long periods (those in excess of half a clock cycle time).

There are three idioms that, in combination, encompass almost all uses of the clock. First of all, to suspend execution for a fixed time, say ten seconds

```
var started :
seq
  time ? started
  time ? after started + (10 * second)
```

This might happen as a once-only action in a program while starting or stopping some mechanical peripheral device.

If an action is to be performed at regular intervals, say once every ten seconds, then

```
var next.dead.line :
seq
  time ? next.dead.line
  while ...
    seq
      next.dead.line := next.dead.line + (10 * second)
      time ? after next.dead.line
      ... perform action
```

will do this (provided that the action can be completed in under ten seconds!) Notice that each deadline is set relative to the previous deadline, so as to avoid slippage.

Finally, using delay guards allows a program to limit the time for which it is prepared to wait for input.

```
var prompted, ch :
seq
  write.string(terminal.screen, "Yer wot? ")
  time ? prompted
  alt
    terminal.keyboard ? ch
      to.program ! ch
    time ? after prompted + (30 * second)
      to.program ! operator.asleep.or.dead
```

Provided that the input from the terminal.keyboard arrives within thirty seconds of the clock being read, the alternative will select its first guard. After that time, the other guard is ready and the process is no longer obliged to wait for its input.

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. This means that each program will need specific processes which translate those types of value whose text representations are input and output by that program.

This section describes routines to be used by the programs described later.

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) mod 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 if you choose to treat it as such by, for example, outputting it to a terminal.

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

```
var tens :
seq
  tens := 1
  while (n / tens) > 10
    tens := 10 * tens
  while tens > 0
    seq
      output ! '0' + ((n / tens) mod 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 this is not a representable value. The standard, if 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
  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) mod 10)
        tens := tens ÷ 10                    :
```

Notice that it is a matter of the definition of division and the mod operator in occam that changing the sign either of tens or of n just changes the sign of the expression

```
((n ÷ tens) mod 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. This would 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 appendix that contains the 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 on 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 signal on a special channel 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
```

where the arithmetic is essentially similar to that in the output routine. As in that case, you will have to be careful with the most negative integer: it will not do to read negative numerals by reading the digits as if of a positive numeral and changing the sign of the result. The simplest solution 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, 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. The general solution postpones the decision, giving the caller of the process the option of ignoring the error, or acting on it in any way he chooses, including the option of stopping.

The appendix contains a routine complementary to the write.signed which has this 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 space in its place.

Where once were only interrupts

It used to be that programmers only met concurrently executing processes if they had to code interrupt routines, or to write code which shared store with interrupt routines. An interrupt is a mechanism designed to make small amounts of processing power available at short notice to handle urgent tasks, when it would be unreasonably expensive to make that processing capability permanently available. To this extent, it separates two concerns: an applications programmer wanting to send characters to a lineprinter need only supply them to an interrupt handler; it is the responsibility of the interrupt handler to transmit them to the printer at the precise times that the printer indicates that it is ready for them. In this way, the programmer is relieved of the burden of making frequent checks on the state of the printer, and the structure of his program can be unaffected by the timing constraints imposed by the printer.

Interrupt routines are notoriously difficult to code and to use. In addition to assuming responsibilities of meeting real-time deadlines, the interrupt routine must maintain the programmer's illusion that the application program has exclusive use of the processor and store. This imposes rigorous discipline on the use of registers, and of store locations, both to avoid conflicts, and in the management of those shared variables by which program and interrupt routine communicate. Moreover, the interrupt routine has usually to be programmed as a 'subroutine' (rather than a 'coroutine') invoked once by each interrupt, which means that any state that is to persist from the handling of one interrupt to that of the next must be saved in store and reconstructed at the next interrupt. To make matters worse, high level programming languages are rarely able to offer convenient abstractions for coding interrupt handlers, which are inherently machine-dependent, and it is usual for interrupt routines to be written in machine code.

It is tempting to claim that the concurrently executed processes of occam are the right tools for writing interrupt routines. To do so would be misleading: concurrent processes are right for a task for which interrupt routines have always been inadequate! The task is in two parts: that of writing code to meet real-time deadlines; and that of isolating their effects, so as to keep the rest of the program simple.

In occam, sustaining the illusion that the application program has exclusive use of the processor and store is easily done, since each and every process of every occam program operates under this very illusion. The illusion is sufficiently strong that a programmer need never know whether or not any particular process is executed on its own dedicated processor.

Meeting real-time deadlines remains a problem that must be solved by ensuring that each processor is fast enough, and that the code is short enough. Apart from this concern with urgency, an interrupt handler coded in occam can be written in exactly the same way as any other process, and communicates with the application program in the same way as any other processes communicate with each other.

Managing terminal input

To take a concrete example, consider managing the traffic to and from a terminal. Every time a key is struck at the keyboard, there will be a corresponding event (traditionally an interrupt) in the computer, and some action must be taken to read information about the key before the next key is struck, lest the information be lost. Quite often the action taken will be to store the character corresponding to the key in a buffer, from which it will subsequently be read at the leisure of the program which is consuming the terminal input. The capacity of the buffer determines how many characters can be 'typed ahead' of the demand from the program.

In occam, an 'event in the computer' is represented by a communication on a special channel. Special channels are declared by noting some implementation-dependent value (such as the store address of the relevant peripheral controller) in the declaration,

```
chan keystroke.in at 32540 :
```

Programs use special channels just as they would use other channels, except that they use each

channel only for input, or only for output, with the other half of the communication being performed by the peripheral controller.

In the case of the terminal example, it would be possible, every time a key was struck, for the program to perform an input

```
keystroke.in ? ch
```

so a reasonable interrupt handler might be

```
circular.buffer(keystroke.in, request, reply)
```

using the circular buffer coded in the 'Programming structures' section. This process has the disadvantage that, were the buffer to become full through the coincidence of a fast typist and a slow program, the process would no longer be prepared to accept input from keystroke.in. Since there is, fortunately, no mechanism built into current terminals to suspend the execution of the typist while the computer is busy, this would mean that keystrokes made whilst the buffer was full would be lost, without warning.

An improved scheme would be to code the interrupt handler in such a way that it was always prepared to acknowledge the keystroke, and to take some remedial action in case there were no room left in the buffer.

```
proc keyboard.handler(chan request, sink, error) =
  -- Characters typed at the keyboard can be read from sink.
  -- A signal is required on request before each item is read.
  -- If more than type.ahead are typed-ahead, there is an error signal.
  chan keystrokes.in at ... :
  var reader, writer, count :
  seq
    reader := 0
    writer := 0
    count := type.ahead
    var datum[type.ahead] :
    while true
      alt
        count = 0   &   keystrokes.in ? any
          error ! any
        count > 0   &   keystrokes.in ? datum[writer]
          seq
            writer := (writer + 1) mod type.ahead
            count  := count - 1
        count < type.ahead   &   request ? any
          seq
            sink ! datum[reader]
            reader := (reader + 1) mod type.ahead
            count  := count + 1                    :
```

This process signals on the error channel if an attempt is made to overfill the type-ahead buffer; later, I will use this signal to ring the bell on the terminal.

Notice that the keyboard.handler is written in such a way that, provided

* the outputs to error and sink are never delayed for more than a fixed time
* this process executes at a known rate within a known short time of becoming ready

then it is possible to put a bound on the length of time before the process next becomes ready to accept an input from keystroke.in. Bounds of this kind are what you would need to demonstrate that no interrupts were lost.

Managing terminal output

For the purpose of this example, suppose that the outgoing traffic to the terminal screen consists of a sequence of bytes passing along the special channel screen.out to be displayed as characters on a screen, or acted upon in some other way by the terminal. The terminal may then become busy for some short time, before again being ready to accept output. The screen handling process has to accept characters from the user's program, and to pass them on; additionally, it must accept urgent error signals from the process handling the type-ahead buffer, and send a 'bell' character to the terminal when an error is flagged.

```
def control = not ((not 0) << 5) :

proc screen.handler(chan outgoing, error) =
  def bell.character = control /\ 'G' :
  chan screen.out at ... :
  while true
    var ch :
    alt
      outgoing ? ch
        screen.out ! ch
      error ? any
        screen.out ! bell.character          :
```

It might appear that there are no timing constraints on the behaviour of the screen.handler, but recall that the performance of the keyboard.handler depends on its error signals not being delayed unduly. As written, the error-guard in the screen.handler might indeed be delayed indefinitely, even were it guaranteed that the screen.handler was executed immediately either of the guards became ready. It might be that the process that sends characters along the outgoing channel is able to send a new character in less time than it takes the screen.handler to execute the body of its while loop once. In that case, the outgoing-guard would always be ready every time the alternative was executed, and since an alternative can choose any one of the ready guards, it is possible that the error-guard might be ignored indefinitely, even were it ready. Notice, particularly, that this behaviour is not caused by my having written the outgoing-guard first: the order of the components of an alternative is immaterial to its meaning.

For just this reason some dialects of occam have an additional constructor, pri alt, which breaks the symmetry. (pri is to be read 'prioritized'!) The components of an asymmetric alternative are the same as those of the symmetric construct, but the meaning differs in that earlier components are treated more favourably than later ones. The alternative waits until one of its guards is ready, then the earliest (nearest to the top of the paper) of the ready guards is selected. Execution of the selected component is then the same as it would be in a symmetric alternative. This means that if the screen.handler were re-written

```
proc screen.handler(chan outgoing, error) =
  def bell.character = control /\ 'G' :
  chan screen.out at ... :
  while true
    var ch :
    pri alt
      error ? any
        screen.out ! bell.character
      outgoing ? ch
        screen.out ! ch                      :
```

then an error signal could not be delayed for longer that it takes to execute the body of the while loop once. Discharging the responsibility to accept these signals in a fixed time reduces to

showing that

* the outputs to screen.out are never delayed for more than a fixed time
* this process executes at a known rate within a known short time of becoming ready

The first requirement is met by the terminal, by assumption; to the second I will return later. Notice that there is no constraint on the timing of transactions on the outgoing channel; I am building a firewall around the terminal, beyond which meeting real-time deadlines will no longer be a concern.

A particular program that uses the terminal may contain a large number of processes, each needing to send characters to the terminal screen. Since the outgoing channel is now the only way out to the terminal, and since only one process is able to send along that channel, a process must be written to interleave the many output streams, and send the interleaving along outgoing.

```
def release = -1 :

proc output.multiplexer(chan from[], value width, chan outgoing) =
  while true
    var ch :
    alt selected.process = [0 for width]
      from[selected.process] ? ch
        while ch ≠ release
          seq
            outgoing ! ch
            from[selected.process] ? ch                    :
```

This process interleaves messages from each of the from[...] channels, in an arbitrary order, each message being terminated by the release value. The most interesting property of this process, for present purposes, is that it is outside the firewall: there are no constraints on the speed with which it executes, nor on the times at which other processes communicate with it.

Managing echoing

The time-dependency firewall is not yet complete: there remains the problem of reading from the type-ahead buffer. Recall that, having issued a request signal, the reader assumes a responsibility to accept the reply from the sink channel within a fixed time. This means that the reader must be within the firewall. Here is a suitable process, which reads characters from the type-ahead buffer, and to which I have given the job of 'echoing' the printable characters to the terminal screen as they are read by the program using the keyboard input.

```
proc echo.handler(chan request, reply, echo, inward) =
  def enter = control ∧ 'M' :
  while true
    var ch :
    seq
      request ! any
      reply ? ch
      inward ! ch          -- Transmit character to user
      if
        (' ' ≤ ch) and (ch ≤ '~')
          echo ! ch        -- Send visible input back to terminal screen
        ch = enter
          echo ! release   -- Release screen at end of line of input
        true
          skip                                             :
```

The only timing constraint on this process is that it execute sufficiently rapidly that the input

from reply is accepted within a permissible time of the preceding request being accepted by the keyboard.handler. There being no constraints on communication on the echo and inward channels, these may cross the firewall: the echo channel is to be one of the array from[] going to the screen.multiplexer, and the inward channel can be used directly by the process that consumes keyboard input.

Notice that the screen sharing strategy is implemented by an 'ordinary' process not subject to any timing constraints. Since each line of echoed characters from the type-ahead buffer is sent as a message to the screen as an indivisible message, there is no problem about input characters being mixed in with output, but neither is there any need for the echo.handler to be concerned with screen allocation.

If the program that used the terminal were written as a named process, user, then the whole could be put together with the terminal handler

```
def type.ahead = ...,  control = not ((not 0) << 5),  release = -1 :

proc keyboard.handler(chan request, sink, error) =
  ...
proc echo.handler(chan request, reply, echo, inward) =
  ...
proc output.multiplexer(chan from[], value width, chan outgoing) =
  ...
proc screen.handler(chan outgoing, error) =
  ...
proc user(chan terminal.keyboard, terminal.screen) =
  ...

chan request, reply, error, outgoing, from.keyboard :
def from.echo.handler = 0, from.user = 1, number.of.outputs = 2 :
chan to.screen[number.of.outputs] :

par
  keyboard.handler(request, reply, error)                                  -- *
  echo.handler(request, reply, to.screen[from.echo.handler], from.keyboard)  -- *
  output.multiplexer(to.screen, number.of.outputs, outgoing)
  screen.handler(outgoing, error)                                          -- *
  user(from.keyboard, to.screen[from.user])
```

Configuration directives

Ignoring, for the moment, the timing constraints imposed by the proper handling of the terminal interrupts, checking the correctness of this program can be done in two parts. First of all, there are properties of individual processes that can be checked in isolation from the other processes: for example, that the echo.handler performs a cycle of four communications in a fixed order, behaviour that is unaffected by the other processes. Secondly, there are some properties that are inherently global, notably freedom from deadlock, which may depend on the behaviour of every one of the processes.

The same is the case with the timing constraints: the argument thus far has been about each of the component processes, more or less in isolation. Had I settled on a particular implementation of occam on a particular computer, and on a particular set of terminal characteristics, then I could have calculated the 'fixed times' within which actions must occur as so many seconds of processor time, so much communication time, and so on. It remains, however, to be demonstrated that there will always be sufficient processor time available when it is required.

One way of achieving this would be to dedicate a processor to the execution of each of the five components of the program. Dialects of occam intended for writing such multi-processor programs have a variant of the parallel constructor, placed par, for indicating such a division

of labour. Were this used in place of the par in the present program, then the processor occupancy times calculated for the three starred processes would be actual elapsed times, each independent of the processor loading of the other processes. In this particular case, such a solution seems excessive, since the tasks are each fairly simple, and the traffic is light. It would be a more reasonable way of dealing with, say, the traffic to and from a fast disk, where a whole transputer might be allocated to managing the large volumes of data, and the potentially intricate calculations required to make efficient accesses to the disk.

More realistically, this particular program would probably be run on a single processor, say one transputer. As it stands, in order to be able to guarantee sufficient speed of execution in the starred processes, I must know details of the behaviour of the unstarred processes: for example, that the user process does not require more than a known propertion of the processor's time. This being unsatisfactory, there is another dialectal variant of par, one which distinguishes more and less urgent tasks. As with asymmetric alternative the asymmetric parallel construct, constructed with pri par, is made of the same components as the symmetric variant, but differs in execution by favouring its earlier components. For example, the process

```
pri par
  p
  q
  r
```

executes by the concurrent execution of its three components, but q can only execute when p is prevented from doing so because it is waiting for a communication or has already terminated. Similarly, r can only execute when both p and q are blocked, and execution of r will rapidly be suspended should either of the higher priority processes become ready.

The asymmetric parallel constructor, if used, must be the outermost constructor of a uni-processor program, or the outermost constructor of one of the branches of a placed parallel construct. In occam programs to be executed on currently proposed transputers, asymmetric parallel constructs can have no more than two components, corresponding to the two process-queues in the transputer. For that machine, the right way to organize the terminal handler would be

```
pri par
  par                  -- High priority process
    keyboard.handler(request, reply, error)
    echo.handler(request, reply, to.screen[from.echo.handler], from.keyboard)
    screen.handler(outgoing, error)
  par                  -- Low priority process
    output.multiplexer(to.screen, number.of.outputs, outgoing)
    user(from.keyboard, to.screen[from.user])
```

Now it cannot matter what the user or output.multiplexer processes do: if any of the urgent processes is able to execute, then one of them will do so within a very short time. This latency will be determined and guaranteed by the implementation, so again if I had a particular implementation in mind, this would be known. The total waiting time, for any of the 'interrupt' processes, between becoming ready and beginning to execute, is bounded by the sum of one latency time and the sum of the longest execution time of each of the other interrupt processes.

That completes the analysis of the timing of the program. All that is needed in the case of a particular implementation is to calculate the times, a matter of counting instructions, which task could and should he delegated to the compiler. Substituting the figures for the waiting and execution times allows a check to be made that the required response times are achieved.

Parallel matrix multiplication

In systems which manipulate and display geometrical data, one of the common routine tasks is the application of linear transformations to the data. A system containing a representation of a three-dimensional object may need to rotate or displace that representation so as to select a point of view from which to project a two-dimensional picture of the object onto a terminal screen, or a plotter. If the positions of the parts of the object are represented by a sequence of Cartesian co-ordinates, then these rotations and displacements can be achieved by matrix multiplication. For each point, with co-ordinates ⟨x[0], x[1], x[2]⟩ it is necessary to calculate the corresponding transformed co-ordinates ⟨y[0], y[1], y[2]⟩ given by

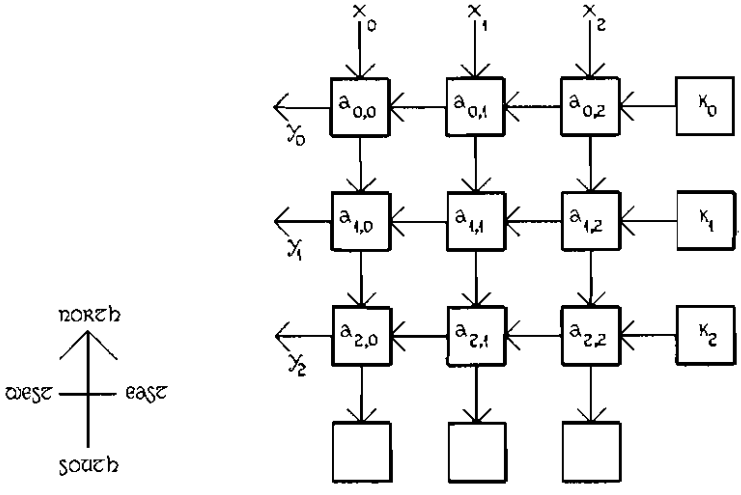
$$y[i] = \sum_{l=0}^{2} (a[i, j] \times x[j]) + k[i]$$

This requires nine multiplications and nine additions for each point in the representation of the object.

If the transformation is being applied once to an object with a view to printing an image on a slow, hard copy device such as a pen plotter, then the time taken to do the transformation is probably not important, and it does not matter much how the matrix multiplication is organized. On the other hand, if the image is being displayed on a cathode ray tube, and the observer is allowed to change his point of view from the console, then speed is important. Ideally, the transformation should be applied to every relevant point of the object as the position of that point is required to refresh the display, so that observer sees the effect of a change in the transformation as soon as possible.

If there are of the order of a thousand points in the representation of the image, then this means something of the order of a hundred thousand matrix multiplications in a second. For practical purposes, this requires that special hardware be dedicated to performing the matrix multiplications on a stream of co-ordinates on its way to the display. In such an arrangement, the time taken to perform the nine individual multiplications will dominate the time taken by all of the communications and additions involved. There is therefore an advantage in arranging that as many as possible of the multiplications can happen at once.

A natural configuration of processors to perform this task is a square array, mimicking the matrix a, one processor being responsible for each element of the matrix, and performing the multiplication by that element.



Successive values of each x co-ordinate are poured into the array from the top, passing down along the north to south channels, and successive values of the transformed y co-ordinates emerge from the east to west channels at the left of the array. In this diagram, each processor is labelled with the parameter for which it takes responsibility. For simplicity, the transformation is assumed to be constant: a mechanism for changing the parameter values might involve a further array of channels at right angles to the plane of the array of processes connecting each relevant processor to a controlling process.

Each multiplier cell has three tasks to perform during each complete matrix multiplication: getting the next co-ordinate, x[j], from its northern neighbour and passing it on to its southern neighbour; performing its own multiplication; getting a partial sum from its eastern neighbour, adding its own contribution and passing the sum on to the west. These tasks can be performed sequentially

```
var xj, aij.times.xj, yi :
while true
  seq
    seq
      north ? xj
      south ! xj

    aij.times.xj := aij * xj

    seq
      east ? yi
      west ! yi + aij.times.xj
```

Because the condition on the loop is a constant true, this process never terminates; it repeatedly performs the three tasks in strict sequence. Since this is a design for highly parallel hardware, it should be worth extracting a little more parallelism

```
proc multiplier(value aij, chan north, south, west, east) =
  var xj, aij.times.xj, yi :
  seq
    north ? xj
    while true
      seq
        par
          south ! xj
          aij.times.xj := aij * xj
          east ? yi
        par
          west ! yi + aij.times.xj
          north ? xj                                    :
```

Since different components of the multiplier would be used by each of the communications and the arithmetic, the branches of the par constructs naturally execute simultaneously.

Notice that the multiplier process does not need to know where it is in the array - it is independent of i and j. This means that the hardware could use nine identical circuits.

In order to complete the multiplier, a source of the k[i] offset values is needed along the eastern border

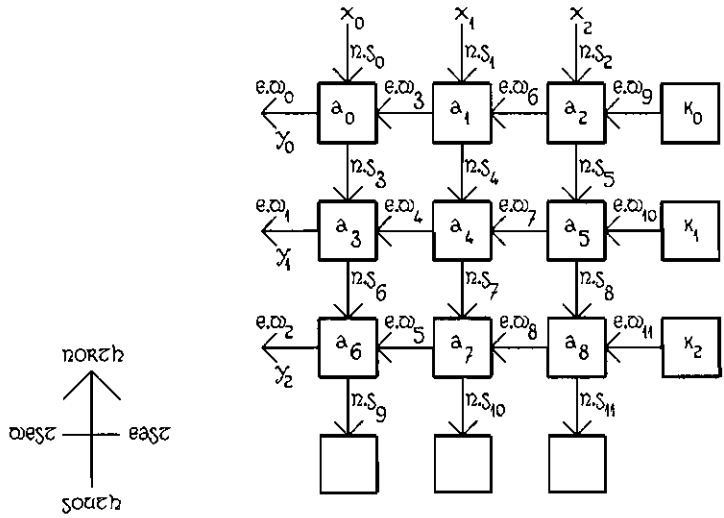
```
proc offset(value ki, chan west) =
  while true
    west ! ki                    :
```

and a sink must be provided at the southern end of each column of multipliers to receive the redundant x[j] from the southernmost multiplier processes

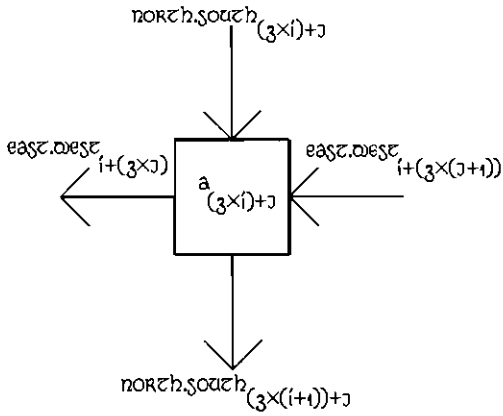
```
proc sink(chan north) =
  while true
    north ? any        :
```

Although the sink does nothing with the values received, its input actions are necessary so that the corresponding output can happen in its neighbouring multiplier. A row of sink processes yields a simpler solution than one which involves two kinds of multiplier process, one for the north of the array, and another for the southernmost row.

Connecting these components to form the matrix multiplier is a matter of choosing an enumeration for the channels, and using chan arrays, suitably indexed. One solution is



The a parameters have also been enumerated, so as to correspond to the elements of an occam one-dimensional array. The channels connected to a typical multiplier process in this diagram are



so the whole multiplier can be described by the program

```
def n = 3 :
var a[n × n], k[n] :
seq
  -- initialise a and k

  chan north.south[(n + 1) × n], east.west[n × (n + 1)] :
  par
    par j = [0 for n]                          -- producer of co-ordinates x[j]
      produce.xj(j, north.south[j])

    par                                        -- the matrix multiplier
      par i = [0 for n]
        offset(k[i], east.west[(n × n) + i])
      par i = [0 for n]
        par j = [0 for n]
          multiplier( a[(n × i) + j],
                      north.south[(n × i) + j], north.south[(n × (i + 1)) + j],
                      east.west   [i + (n × j)], east.west   [i + (n × (j + 1))] )
      par j = [0 for n]
        sink(north.south[(n × n) + j])

    par i = [0 for n]                          -- consumer of transformed co-ordinates
      consume.yi(i, east.west[i])
```

It is the task of each produce.xj process to output successive values of the co-ordinate corresponding to its first parameter, and that of each consume.yi process to input successive values of the transformed co-ordinate.

By devising suitable definitions for the produce.xj and consume.yi processes, this program can be used on any occam implementation as a simulation of the parallel matrix multiplier hardware. Of course, if it is executed on a single processor computer, then it will be very much slower than a simpler sequential program, because of the additional work in communicating and scheduling. On the special hardware for which it is designed, however, it would be very much faster. The longest data path from input to output is that traversed by x[2] on the way to contributing to y[2]. This path involves six communications, three additions, and a single multiplication, all of which must happen in sequence. The program is designed on the assumption that the time taken for the multiplication would dominate all others, under which assumption it would be almost nine times faster than a sequential multiplier.

The matrix multiplier example appears in essentially this form in

Communicating Sequential Processes, C.A.R. Hoare
in Communications of the ACM, 21 (8), August 1978, pp 666-677

Parallel sorting

Sorting is a candidate problem for parallel solution because many algorithms have an element of divide-and-conquer. That means the task is carried out by dividing it into some number of smaller, simpler tasks each of which is repeatedly divided until only trivial tasks remain. Such a strategy rapidly identifies independent parts of the original problem, which can be tackled concurrently.

I make no claims for the sorting algorithm used here, beyond its simplicity. Although a parallel sorting program is described, the subject is how to observe a parallel program in operation. With small changes to the sorting program itself, its activity can be displayed on a vdu screen, turning the program into a simulator of its own behaviour.

Sorting strategy

The program consists of a number of simple processes linked together in a tree shaped structure. As in the case of the matrix multiplier, no process need ever know where it is in the tree: there will be only two types of process: leaves, and internal nodes. Again, each process is independent of the size of the problem, and need never store more than two values and some flags, no matter how many values are being sorted. A bigger problem demands a bigger tree, but the components are unchanged.

The strategy is to distribute the numbers upwards from the root of the tree, until they are spread out, one to each leaf. Each process is then responsible for sending back to its parent the sequence of numbers which it has received, but sorted into ascending order. For a leaf, the task is simple, since its one number already constitutes a sequence in ascending order. Each internal node, relying on the sorted subsequences that it will receive from its children, merges two ascending sequences to generate its output sequence.

Each leaf process needs two-way communication with its parent,

```
proc leaf(chan up, down) =
  ...
```

and each internal node needs six channels, two to provide two-way communication with its parent, and two each to and from each of its children.

```
proc fork(chan up, down, left.down, left.up, right.down, right.up) =
  ...
```

For simplicity, the root process is treated as an internal node, with a virtual root process acting as the parent of the root

```
proc driver(chan up.to.tree, down.from.tree) =
  ...
```

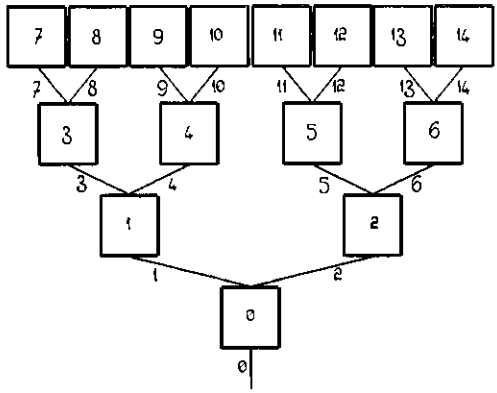
and acting as a driver to control the activity of the tree.

To connect these process, they have to be indexed, so as to correspond to linear arrays of channels. For simplicity, I have made the program a complete balanced tree

```
def depth.of.tree = 3 :

def number.of.leaves = 1 << depth.of.tree ,
    number.of.forks = number.of.leaves - 1 ,
    number.of.processes = number.of.forks + number.of.leaves ,
    number.of.channels = number.of.processes :
```

then, numbering the processes breadth-first, upwards from the root




```
def root = 0 ,
    first.fork = root ,
    first.leaf = first.fork + number.of.forks :
```

the children of the internal node process i are indexed (2×i)+1 and (2×i)+2. If channels indexed i are used to connect process i to its parent, these same formulae will give the indexes of the channels to and from the children of internal node process i.

```
chan up[number.of.channels], down[number.of.channels] :
par
  driver(up[root], down[root])
  par i = [first.fork for number.of.forks]
    fork(up[i], down[i], down[(2×i)+1], up[(2×i)+1], down[(2×i)+2], up[(2×i)+2])
  par i = [first.leaf for number.of.leaves]
    leaf(up[i], down[i])
```

Components of the sorter

There are two phases of activity in the tree: first the sequence of numbers is distributed; then sorted sequences are gathered and merged. Each component process passes through the same two phases.

```
proc fork(chan up, down, left.down, left.up, right.down, right.up) =
  seq
    fork.distribute(up, left.up, right.up)
    fork.gather(down, left.down, right.down)  :
```

During the distribution phase, each internal node receives a sequence of numbers from its parent. Notice that since a fork process knows neither where it is in the tree, nor how big the tree is, it cannot know how many numbers to expect. Accordingly, the sequence is passed around with each number preceded by a true value, and the last followed by a false value. Such a sequence can be read by

```
var more :
seq
  up ? more
  while more
    var next :
    up ? next; more
```

The simplest way of distributing the sequence amongst the children, without foreknowledge of

its length, is to send one-for-left, one-for-right, alternately.

```
proc fork.distribute(chan up, left.up, right.up) =
  def leftward = 0, rightward = not leftward :
  var more, inclination :
  seq
    inclination := leftward
    up ? more
    while more
      var number :
      seq
        up ? number
        if
          inclination = leftward
            left.up ! true; number
          inclination = rightward
            right.up ! true; number
        up ? more
        inclination := not inclination
    par
      left.up ! false
      right.up ! false                    :
```

Notice that this process passes the guarantee of correctly interpolated true and false values on to its children.

Since I assumed that the component processes would serve in an arbitrarily large tree, they should not count the numbers as they pass upwards. This means that during the merging phase there are again sequences of unknown length to be read, and I will use a similar protocol.

In order to do the merging, preserving ordering, numbers must be compared, and this requires that each merging process have at least two registers holding numbers. Since each child sends its sequence in ascending order, the head of each sequence is the minimum of those to come, so the merging process compares the heads, passes on the smaller, and draws one more value from the selected sequence, continuing until the sequences are exhausted.

```
proc fork.gather(chan down, left.down, right.down) =
  var left.more, left.minimum, right.more, right.minimum :
  seq
    par
      left.down ? left.more; left.minimum
      right.down ? right.more; right.minimum
    while left.more or right.more
      if
        left.more and ((not right.more) or (left.minimum < right.minimum))
          seq
            down ! true; left.minimum
            left.down ? left.more; left.minimum
        right.more and ((not left.more) or (left.minimum > right.minimum))
          seq
            down ! true; right.minimum
            right.down ? right.more; right.minimum
    down ! false; any                     :
```

Notice a final any sent downwards at the end of the sequence, which accounts for the parent process being able to receive a pair of values, ...more, ...minimum, even when the first is

false. This trick is simpler than making the parent's behaviour conditional on the first value, and there is very little penalty since after sending down its false value there is nothing left for the child process to do.

The driver process must generate and absorb sequences of numbers, stuffing and stripping the protocol.

```
proc driver(chan up, down) =
  seq
    seq i = [0 for number.of.leaves]
      var number :
      seq
        ... think of a number
        up ! true; number
    up ! false
    seq i = [0 for number.of.leaves]
      var number :
      seq
        down ? any; number
        ... do something with the number
    down ? any; any                          :
```

The missing code controls the behaviour of the whole program. It might, for example, read numbers from the terminal keyboard, and write them back, in ascending order, to the terminal screen.

Finally the leaf process must be designed to simulate the behaviour of an internal node that only handles a sequence of one number

```
proc leaf(chan up, down) =
  var number :
  seq
    up ? any; number; any
    down ! true; number; false; any :
```

That completes the sorting program which, whilst it may look overcomplex for a single processor implementation, would look better on an array of number.of.processes simple processors. Notice, particularly, that once the numbers have started to emerge from the tree in ascending order, each is available only one comparison time after its predecessor. The advantage would be more obvious were the sorter managing more complex data, where the comparison time might be very large.

Monitoring strategy

The program as it stands may be run on a single processor to simulate the activity of the ideal multiprocessor implementation. By writing the missing code in the driver, you could observe numbers going into and coming out of the tree, checking that the program sorts particular sequences of numbers. That tells you nothing about what goes on inside the tree: just as in spring, it might be edifying to be able to observe the activity up in the branches.

By analogy with the testing of electronic circuits, the idea is to probe the components of the circuit, rather than just watching the signals that pass into and out of the terminals. There are two techniques: breaking connections to measure the current flowing through them corresponds to tapping the channels to watch the traffic; attaching probes to measure the potential at various points corresponds to noting state changes in the processes.

In order to observe the traffic on a channel, a process must be added which duplicates the traffic along a monitoring channel, something like

```
proc duplicate(chan source, sink, copy) =
  while true
    var datum :
    seq
      source ? datum
      par
        copy ! datum
        sink ! datum                    :
```

This process can be inserted into a data stream passing along a channel

```
chan channel :
par
  producer(channel)
  consumer(channel)
```

allowing the data to be read by another process

```
chan channel.a, channel.b, test.data :
par
  producer(channel.a)
  duplicate(channel.a, channel.b, test.data)
  consumer(channel.b)

  monitor(test.data)
```

Of course, the observation is not perfect: it may affect the behaviour of the program. First of all, the duplicate process acts as an additional buffer in the data stream. In this example it cannot matter, but were there some other communication, possibly through a third party, between the producer and consumer, it might matter that the output from the producer could proceed, despite the corresponding demand not being made in the consumer. Secondly, the duplicate process, as written, does not terminate, so unless it is used to observe an infinite data stream, the program will eventually become deadlocked, even had it previously terminated correctly.

In both of these ways you must be careful to design monitoring code that does not interfere excessively with the action being observed. In general, it is necessary for the behaviour of the monitoring processes to depend on the data passing through them, and this in-stream technique should be avoided if there are many data paths between pairs of processes in the program being observed.

In order to make internal state visible, it is necessary to add code to the processes being observed. Just as observing traffic involves adding new output processes in parallel with the observed program, so observing state requires that new output processes be set in sequence with the code being observed. In order to observe the changing value of a variable

```
proc p(...) =
  var x :
  seq
    ⋮
    x := e
    ⋮
    c ? x
    ⋮
```

each assignment to that variable should be followed by an output process signalling the change

```
proc p(..., chan test.data) =
  var x :
  seq
      ⋮
    seq
      x := e
      test.data ! x
      ⋮
    seq
      c ? x
      test.data ! x
      ⋮
```

on a channel which passes out to the monitoring code. Again, the observation is invasive: you must be aware that the observed process may be delayed by executing the new output processes.

In the example of the parallel sorter, I will use both types of monitoring: the explanation of the behaviour of the merging is in terms of the sequences of values passing along channels, so the traffic along the channels will be watched; the leaves are used as storage locations, so it is appropriate to observe their state.

The result of adding this monitoring code is a number of channels emerging from the tree, each carrying signals indicating the presence or absence of a number. Each will be treated similarly, either to write a number to or to remove it from a position on the screen which will represent the place in the program which is being watched. Since changes to the screen must be made in sequence, it is appropriate to multiplex the test data from the tree, and process each new test signal in sequence.

These decisions lead to the following, changed, program structure

```
def number.of.probes = number.of.channels + number.of.leaves :

chan up.a[number.of.channels], down.a[number.of.channels],
     up.b[number.of.channels], down.b[number.of.channels],
     probe[number.of.probes], all.probes :

par
  driver(up.a[root], down.b[root])

  par i = [first.fork for number.of.forks]
    fork(up.b[i], down.a[i], down.b[(2×i)+1], up.a[(2×1)+1], down.b[(2×1)+2], up.a[(2×1)+2])

  par i = [first.leaf for number.of.leaves]
    leaf(up.b[i], down.a[i], probe[number.of.channels + (i - first.leaf)])

  par i = [root for number.of.channels]
    monitor(up.a[i], down.a[i], up.b[i], down.b[i], probe[i])

  multiplex(probe, all.probes)

  display(all.probes, terminal.screen)
```

Each monitor process copies data from its ...a channels to its ...b channels, duplicating the activity along the corresponding probe. Every leaf is modified to indicate its state with similar messages. All of these messages are multiplexed onto a single channel, and then translated into sequences of instructions to display the changing state of the program on the terminal screen.

Component processes

There are three types of message to be sent along the probe channels: messages indicating the presence of a number, messages indicating the absence of a number, and a final termination message. Each of these will be indicated by starting it with one of three values

```
def display.number = 1, display.empty = 2, display.stop = 3 :
```

Sending an explicit termination signal means that the behaviour of the tree can be altered without the monitoring code having to be changed.

To start with the leaf process, all that is needed is to indicate the arrival and departure of the stored number.

```
proc leaf(chan up, down, probe) =
  var number :
  seq
    up ? any; number
    probe ! display.number; number
    up ? any
    down ! true; number
    probe ! display.empty
    down ! false; any
    probe ! display.stop          :
```

The monitor process must copy the sequences of numbers passing first up and then down the tree using the correct protocol for each case. The necessary monitoring code is then just what you would need to record changes of state in this buffering processes

```
proc monitor(chan up.a, down.a, up.b, down.b, probe) =
  seq
    var more :
    seq
      up.a ? more
      while more
        var number :
        seq
          up.a ? number
          probe ! display.number; number
          up.b ! more; number
          probe ! display.empty
          up.a ? more
      up.b ! more
    var more, number :
    seq
      down.a ? more; number
      while more
        seq
          probe ! display.number; number
          down.b ! more; number
          probe ! display.empty
          down.a ? more; number
      down.b ! false; any
      probe ! display.stop                    :
```

The multiplex process simply gathers together all of the probe signals, tagging them with the

corresponding index number for later identification.

```
proc multiplex(chan probe[], all.probes) =
  var more, more.from[number.of.probes] :
  seq
    more := number.of.probes
    seq i = [0 for number.of.probes]
      more.from[i] := true
    while more > 0
      var instruction :
      alt i = [0 for number.of.probes]
        more.from[i] & probe[i] ? instruction
          if
            instruction = display.number
              var number :
              seq
                probe[i] ? number
                all.probes ! instruction; i; number
            instruction = display.empty
              all.probes ! instruction; i
            instruction = display.stop
              seq
                more.from[i] := false
                more := more - 1
    all.probes ! display.stop                    :
```

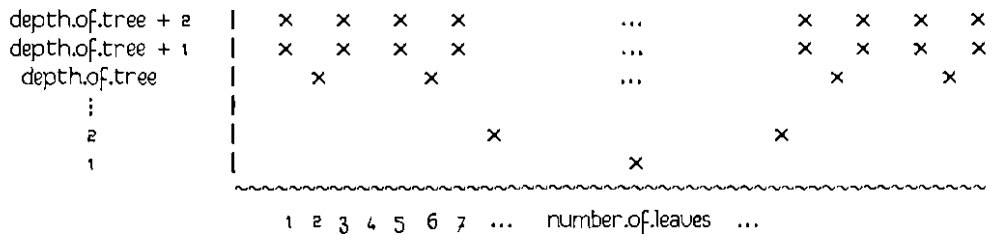
Once a display.stop is received from a particular probe, no more signals are read from it, and the whole multiplexer terminates when all probes have been shut off.

Display management

It remains only to translate the stream of probe messages into a stream of terminal screen control messages. The first thing to do is to translate the probe numbers into positions on the screen. This happens in two stages: first the numbers are translated into positions in a terminal-independent space; then that space is mapped onto the terminal screen.

```
proc display(chan source, sink) =
  chan internal :
  par
    independent(source, internal)
    dependent(internal, sink)      :
```

The terminal independent space has right-handed co-ordinates, with the leaves evenly spread across the top, and the root at the middle of the bottom line.



Messages from probes with index less than number.of.channels are from probes within the tree, and those with higher indices are from the leaves. The top line, representing the states of the leaves, is clearly not a part of the pattern in the rest of the tree, so is dealt with differently. For the channel probes, the simplest solution is to count up from the root. There are ((1 << line) - 1) probes represented on the bottom line lines of the display, so the right line for a particular probe is the first for which its index lies below this number. The right column is calculated by discounting the ((1 << (line - 1)) - 1) probes displayed on the lower lines and multiplying by a factor which accounts for the exponential separation of nodes at different depths

```
proc make.cartesian(value index, var x, y) =
  if
    if line = [1 for depth.of.tree + 1]
      index < ((1 << line) - 1)
        var c :
        seq
          c := index - ((1 << (line - 1)) - 1)
          x := ((2 x c) + 1) x (number.of.leaves >> (line - 1))
          y := line
    index > number.of.channels
      seq
        x := (2 x (index - number.of.channels)) + 1
        y := depth.of.tree + 2                          :
```

The make.cartesian process translates a probe index into an x, y pair

$$1 \leq x \leq (2 \times \text{number.of.channels}) - 1 \quad \text{and} \quad 1 \leq y \leq \text{depth.of.tree} + 1$$

The other terminal independent part of the translation is to turn the numbers into digits. All the numbers are written in a fixed width field

```
proc independent(chan source, sink) =
  var instruction :
  seq
    source ? instruction
    while instruction ≠ display.stop
      seq
        sink ! true
        var index, x, y :
        seq
          source ? index
          make.cartesian(index, x, y)
          sink ! x; y
        if
          instruction = display.number
            var number :
            seq
              source ? number
              write.signed(sink, number, field.width)
          instruction = display.empty
            seq i = [0 for field.width]
              sink ! '*s'
        source ? instruction
    sink ! false                                        :
```

The output from this process consists of a sequence of packets, each beginning with an x, y pair, followed by field.width number of characters to be displayed there. Each packet is preceded by a true value, and the whole sequence is terminated with a false.

If the terminal has cursor addressing, then the task is almost complete. Here, for example, is the necessary terminal dependent part of the display process for a digital VT52 terminal

```
def virtual.height = depth.of.tree + 1, virtual.width = (2 × number.of.leaves) − 1 :

proc dependent(chan source, terminal) =
  -- terminal dependent code for driving VT52

  def screen.height = 24, screen.width = 80 :
  def control = not ((not 0) ≪ 5), escape = control ∧ '[' :

  proc clear.screen(chan terminal) =
    -- clear screen sequence for a VT52
    terminal ! escape ; 'H' ; escape ; 'J'          :

  proc goto.xy(chan terminal, value x, y) =
    -- lefthanded co-ordinates, origin 0, 0 at top left
    terminal ! escape ; 'Y' ; '%s' + y ; '%s' + x    :

  var more :
  seq
    clear.screen(terminal)
    source ? more
    while more
      seq
        var x, y :
        seq
          source ? x; y
          goto.xy(terminal, (x − 1) × (screen.width ÷ virtual.width),
                            (virtual.height − y) × (screen.height ÷ virtual.height))
        seq i = [1 for field.width]
          var ch :
          seq
            source ? ch
            terminal ! ch
        source ? more
    goto.xy(terminal, 0, screen.height − 1)          :
```

The divison of work is such that, if it is at all reasonable to draw such pictures on a particular terminal, the program can be modified to do so simply by writing the appropriate dependent process. Even should the terminal not have full cursor control, but only the ability to move the cursor in small steps, dependent can be made to keep track of the position of the cursor.

For the purpose of the simulator, the simplest coding of the driver process invents a random sequence of numbers for input to the tree. A common way of generating an unpredictable sequence of numbers is to use a linear feedback shift register with an uncontrolled initial state

```
def mask = not ((not 0) ≪ 9) :

proc shift(var state) =
  seq i = [1 for 9]
    state := ((state ≪ 1) ∧ mask) ∨ (((state ≫ 4) ⊕ (state ≫ 8)) ∧ 1)  :
```

An arbitrary initial state may be obtained by reading the real-time clock. Since the shift-register will not change state if the initial state is all zeros, the time is \/-ed with a one to guarantee a non-zero intial state.

This coding of the driver pauses after injecting each number into the tree, and after removing each number from the tree, so as to give you time to see what is happening. There is nothing scientific about the choice of a one second pause: I adjusted it to get a good display from the particular implementation that I was using.

```
proc driver(chan up, down) =
  seq
    var event, number :
    seq
      time ? event
      number := (event /\ mask) \/ 1
      seq i = [0 for number.of.leaves]
        seq
          event := event + second
          shift(number)
          up ! true; number
          time ? after event
      up ! false
    var event :
    seq
      time ? event
      seq i = [0 for number.of.leaves]
        seq
          event := event + second
          down ? any; any
          time ? after event
      down ? any; any                    :
```

The driver discards the result of the sort, because all the information has already been displayed, as it passes out of the root process.

Conway's game of 'Life'

Lest you be misled by the name, 'Life' is neither a competitive game between several players, nor yet a solitaire game in which a player competes against the collusion between the rules and the roll of the dice. The game is more a simulation, in which the evolution of a system is fully determined by a set of rules.

To be precise, Life is played on an infinite square board: that means that there are a number of squares, or 'cell's, each of which has four immediate neighbours and four diagonal neighbours, in the fashion of a chess board. That the board is infinite means simply that every cell in which you will be interested is one with a full complement of neighbours, so that you need never worry about what happens at the edges. There will be only a finite number of interesting cells to think about at any one time. Each cell may be in one of two states: occupied (alive) or unoccupied (dead), and only finitely many will be alive at any time.

The rules describe the succession of states of each cell in terms of earlier states of that cell and its eight near neighbours. Each cell passes through a sequence of generations, with the state of the cell in the next generation being determined by its state in this generation, and by the number of cells adjacent to it which are either alive or dead in this generation. If a cell is currently alive, and if it has less than two live neighbours, it is deemed to die of loneliness, and will be dead in the next generation. A live cell with two or three neighbours alive in the same generation survives into the next generation, but if it has four or more contemporaries, it will be dead from overcrowding by the next generation. A dead cell with exactly three live neighbours in this generation will give birth and be alive in the next generation, otherwise it will remain barren.

Notice that the rules determine the state of the whole board in the next generation in terms of its state in the present generation. Moreover, the rules are expressed in purely local terms, and the property of Life that makes it interesting is that these local rules can control the evolution of global structures. A number of patterns of live cells are known to pass through cycles of growth and decline, some are known to grow without limit, whilst others die out.

Although the rules of evolution are simple, applying them to a pattern large enough to be interesting, for more than one or two generations, is a tedious business. Machine assistance makes it possible to watch the long term development of substantial colonies, and Life was once a popular way of consuming otherwise unused machine cycles! More practically, a Life board is a particularly simple and symmetrical example of a systolic cellular array. These are studied by VLSI designers seeking algorithms with fast but simple implementations in highly parallel hardware. A systolic array is characterized by the achievement of global co-operation through many simultaneous calculations organized by local communications. Ideally, the components of the array are, like the cells of a Life board, all of a few basic types, have a small finite amount of state, and need never know where they are in the array.

The program described here is, as with the parallel sorter, a simulation in two parts: there is a plane of parallel processes in which the cells of a Life board are represented, one cell to a process; added to this is an essentially sequential mechanism for guiding and watching the evolution of the colony. Perhaps it is worth pointing out at the outset that the resulting program, run on a single processor, is far from the fastest way of playing Life. There are, for example, a number of optimizations that require each process to have a more global view of the state of the board, and naturally give rise to a sequential program. This program is here for two reasons: firstly as an intricate example of the interconnection of processes, showing how to separate this from the workings of the processes themselves; secondly, it is an example of a general method of adding global synchronization to a loosely coupled system in order to observe its behaviour.

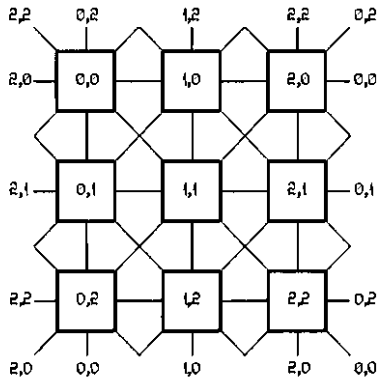
The Life board

There is no problem in selecting a representation the board. Each cell of the board has a state, so is represented by a process which administers the variable in which that state is stored. There is no reason why each of these processes should not be identical. Each cell is distinguished only

by the particular eight other cells which are close enough to influence its state in the next generation. The neighbours of a cell process are connected to it, each by a pair of channels, one in each direction.

The first problem that arises is one of representing an infinite board on what must necessarily be a finite array of processes. As suggested earlier, the requirement of an infinite board is made so that the behaviour of a cell will not be influenced by its being at an edge of the board. Unless a colony grows without limit, or moves en masse in some direction, a finite board will do, since the evolution of a colony is unaffected by any amount of dead space around it.

One solution, and the one that I have adopted here, is to take a finite sized board and wrap it around a torus, so that the cells on the top edge have neighbours on the bottom edge, and



those on the right have neighbours on the left. There are now no edges to worry about. You may think of this toroidal board in either of two ways. Looking at it as a flat board with tricky edges, it correctly implements the rules of healthy living until one or more of the edge cells gives birth, from which point on it is possible for things to go wrong, with miraculous conceptions and unexplained deaths happening in ways not predicted by the rules. Another way of thinking about it is that the toroidal board is behaving as if it were a fragment of a truly infinite flat board on which the real finite colony that you can see is repeated, in the fashion of a wallpaper pattern, at regular intervals in the horizontal and vertical directions. The boundary effects are now explicable, since they are the effects (predicted by the rules) of a neighbouring copy of the colony coming close enough to influence the visible part of the board.

To be more definite about the program, it consists of a rectangular array of cell processes

```
def array.width = ... , array.height = ... :

par x = [0 for array.width]
  par y = [0 for array.height]
    ... process representing cell x, y
```

The neighbours of cell x, y are those indexed

```
((x ± 1) + array.width) mod array.width
((y ± 1) + array.height) mod array.height
```

with the mod operator taking care of the proximity of cells at the edges of the board. Notice that the numerator of the mod operator has to be made positive, since in occam it is defined that

```
((-1) mod w) = (-1)
```

The next thing to decide is the arrangement of the channels connecting these processes. As in the matrix multiplier example, it would be possible to allocate one channel array to account for all of the data flowing in each compass direction. The result would be that each cell process would be connected to eight individually named channels carrying data inwards, and eight individually named channels carrying data outwards. This is to ignore the symmetry with which the rules of living treat the neighbours of a process. A cell does not distinguish between its neighbours according to their compass direction, but treats them uniformly. The symmetry should be represented by a for loop in the cell processes, the body being executed eight times, once for each neighbour. That suggests that an array of eight channels is needed, indexed by the eight directions.

Since there are, in occam, neither channel variables nor channel pointers, the only neat solution to this problem is to allocate all of the channels from a single large array. Each cell then needs to be told which eight subscripts it should use to select its incoming channels, and which eight to select its outgoing ones.

```
def radius = 1 ,  -- of the 'sphere of influence'
    diameter = (2 * radius) + 1 ,
    neighbours = (diameter * diameter) - 1 :

def number.of.cells = array.height * array.width ,
    number.of.links = neighbours * number.of.cells :

proc initialize(value x, y, var in[], out[]) =
  ... initialize in[...] and out[...]

proc cell(chan link[], value in[], out[]) =
  ... cell using link[in[...]] and link[out[...]]

chan link[number.of.links] :
par x = [0 for array.width]
  par y = [0 for array.height]
    var in[neighbours], out[neighbours] :
    seq
      initialize(x, y, in, out)
      cell(link, in, out)
```

Perhaps this is the place to note that I remain unsatisfied by this solution because of the generality of the variable arrays in[...] and out[...]. A mechanical checker, such as might be a part of an occam compiler is unlikely to be able to verify that the cell makes only legal use of the link channels, since the uses appear to be dynamically determined. A mechanically checkable program would most probably have to recompute the subscripts at the point of use. It is because the effort of recomputing complex subscript expressions would dominate all of the other activity in the program that I have adopted this solution. (Carroll Morgan first showed me the application of this indirection strategy in a Life program on which mine is based.)

The remainder of the board configuration is in the initialization of the indirection array. To do this, an enumeration of the processes and the channels must be chosen. I have chosen to count the processes in the usual way: along the rows then down the colums, from zero at process x zero, y zero in the top left

```
this.process := x + (array.width * y)
```

and to allocate the first eight channels to carry data out of the first process, the next eight out of the next process, and so on. This, of course, accounts for all of the channels, exactly once, since every channel is outward bound from some process.

To settle on a particular enumeration of channels, the eight neighbours of a process must be put in some order. I choose order of increasing direction as computed by the loop

```
seq delta.x = [-radius for diameter]
  seq delta.y = [-radius for diameter]
    var direction :
    seq
      direction := delta.x + (diameter * delta.y)
      ... consider neighbour x+delta.x  y+delta.y
```

which is (except at the top and left edges) the order of increasing process number. The direction of a neighbour characterizes it, and lies in the range

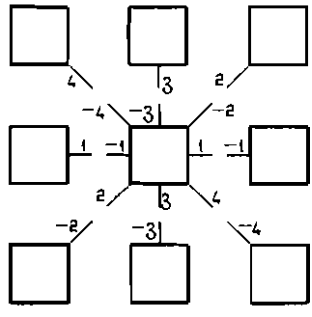
$$-(neighbours \div 2) \leq direction \leq +(neighbours \div 2)$$

with the zero value corresponding to the cell at x, y itself. To fill an array with the neighbour consecutive subscripts of outward going channels the non-zero values of direction must be mapped onto consecutive indices for out, and a group of eight consecutive channel numbers

```
if
  direction <> 0
    var this.index :
    seq
      this.index := (neighbours + direction) mod (neighbours + 1)
      out[this.index] := this.index + (neighbours * this.process)
  direction = 0
    skip
```

The value of this.index so constructed ranges from zero to neighbours-1, taking on each value exactly once, in the course of a scan of the neighbours.

Now the question arises of which are the correct subscripts to use to select the incoming links. Incoming links at this.process are, if looked at from the other end, the outgoing links from the



neighbours of this.process. The simplest, brute force, solution to the problem of enumerating them is to put yourself in the position of the processes at the other end of each channel, and to ask which link that process would be using to talk to this.process, as one of its neighbours. The process at x, y is a neighbour of each of its own neighbours; in particular it is the neighbour in the -direction direction of the process which is its neighbour in the direction direction. (You can see this because direction is linear in delta.x and in delta.y.) This means that the inward channel from the neighbour in the direction direction is the one that, at that other process would be described as the outward channel in the -direction direction.

```
        var other.x, other.y, other.process, other.index :
          seq
            other.x := (x + delta.x + array.width) mod array.width
            other.y := (y + delta.y + array.height) mod array.height
            other.process := other.x + (array.width × other.y)
            other.index := (neighbours - direction) mod (neighbours + 1)
            in[other.index] := other.index + (neighbours × other.process)
```

These fragments being gathered together, the configuration process is complete

```
proc initialize(value x, y, var in[], out[]) =
  -- initialize the link indirection arrays for the cell at x,y
  seq delta.x = [-radius for diameter]
    seq delta.y = [-radius for diameter]
      var direction :
      seq
        direction := delta.x + (diameter × delta.y)
        if
          direction ≠ 0
            var index, process :
            seq
              process := x + (array.width × y)
              index   := (neighbours + direction) mod (neighbours + 1)
              out[index] := index + (neighbours × process)
              process := ((x + delta.x + array.width) mod array.width) +
                         (array.width × ((y + delta.y + array.height) mod array.height))
              index   := (neighbours - direction) mod (neighbours + 1)
              in[index] := index + (neighbours × process)
          direction = 0
            skip                                              :
```

All of the tricky code being now dealt with, the code of the cell process is relatively simple. It records the state of the cell, that it is either dead or alive, and controls the evolution of the state

```
def dead = 0, alive = not dead :

proc cell(chan link[], value in[], out[]) =

  proc broadcast.present.state(chan link[], value out[], state) =
    ... tell neighbours about the state of this cell
  proc calculate.next.state(chan link[], value in[], state, var next.state) =
    ... evolve in keeping with the rules

  var state :
  seq
    state := ...                    -- set an initial state
    while true
      var next.state :
      seq
        par
          broadcast.present.state(link, out, state)
          calculate.next.state(link, in, state, next.state)
        state := next.state                                   :
```

I postpone the matter of the initial state which determines the type of colony being watched.

In each generation, the cell must learn the state of each of its neighbours, so as to count up the number of adjacent occupied cells.

```
proc calculate.next.state(chan link[], value in[], state, var next.state) =
  var count :          -- number of living neighbours
  seq
    var state.of.neighbour[neighbours] :
    seq
      par i = [0 for neighbours]
        link[in[i]] ? state.of.neighbour[i]
      count := 0
      seq i = [0 for neighbours]
        if
          state.of.neighbour[i] = alive
            count := count + 1
          state.of.neighbour[i] = dead
            skip
    if
      count < 2
        next.state := dead    -- death from isolation
      count = 2
        next.state := state   -- this cell is stable
      count = 3
        next.state := alive   -- stable if alive, a birth if dead
      count > 3
        next.state := dead  : -- death from overcrowding
```

Notice that although the input processes are written in a parallel for loop, the counting of live neighbours has to be sequential, since the count variable may not be shared. Whilst the simplest of mechanical checkers would be justified in drawing the programmer's attention to the shared array state.of.neighbour[...], it is clear that no element of the array is shared.

There is a corresponding obligation on a cell to tell each of its neighbours about its own current state

```
proc broadcast.present.state(chan link[], value out[], state) =
  par i = [0 for neighbours]
    link[out[i]] ! state                    :
```

Observation and control

As with the parallel sorter, having completed the highly parallel part of the program, I have still to design a means for controlling and watching what happens. This task demands substantially sequential code, since there is only one terminal keyboard and one terminal screen involved. The observation will impose more synchronization on the array of cells: there is, so far, nothing to prevent widely separated processes from working on as widely separated generations, but the display should be capable of showing the state of one generation at a time, across the whole of the board.

There are three intructions that the controlling process will need to issue to each cell on the board: it may ask for the cell to assume a new state, so as to initialize, and subsequently edit, the state of the board; it may instruct the cell to evolve for one generation; and it may tell the cell process to terminate.

```
def set.state = 1, ask.state = 2, terminate = 3 :
```

In response to instructions to evolve, the cell should yield up its new state. To carry these messages, a channel is needed into each cell, and one from each cell.

```
chan link[number.of.links], control[number.of.cells], sense[number.of.cells] :
par
  controller(keyboard, screen, control, sense)      -- control process
  par x = [0 for array.width]
    par y = [0 for array.height]
      var in[neighbours], out[neighbours] :
      seq
        initialize(x, y, in, out)
        cell(link, in, out, control[x + (array.width * y)], sense[x + (array.width * y)])
```

The cell process must respect the instructions received on its control channel, thus

```
proc cell(chan link[], value in[], out[], chan control, sense) =
  var state, instruction :
  seq
    state := dead        -- the whole board starts off dead
    control ? instruction
    while instruction ≠ terminate
      seq
        if
          instruction = set.state
            control ? state
          instruction = ask.state
            var next.state :
            seq
              par
                broadcast.present.state(link, out, state)
                seq
                  calculate.next.state(link, in, state, next.state)
                  sense ! (state ≠ next.state); next.state
              state := next.state
        control ? instruction                              :
```

At the end of each generation, the cell process sends not only its new state, but an indication of whether the state has changed in this generation. This makes the task of the controlling process simpler.

The controlling process is essentially sequential. Under the control of input from the terminal keyboard, it can either modify the state of cells on the board by issuing set.state instructions

```
proc edit(chan keyboard, screen, control[]) =
  ... modify the colony on the board
```

or it is able to drive the whole board through the evolution of a single generation by scanning the board, issuing ask.state instructions, and reading back the new states.

```
proc generation(chan screen, control[], sense[], var active) =
  ... cause the colony on the board to move on a generation
```

The active parameter returns an indication of whether any changes have happened during the generation: if a colony remains unchanged from one generation to the next, then it is stable, and will never change again.

The normal activity of the controller, free.running, is to cause a sequence of invocations of generation so that the colony is continually evolving. If the colony becomes stable, then the controller becomes idle. Between any two generations, the keyboard has an opportunity to change the activity

```
def idle = 1, editing = 2, single.stepping = 3, free.running = 4, terminated = 5 :

proc controller(chan keyboard, screen, control[], sense[]) =
  var activity :
  seq
    activity := idle
    initialize.display(screen)
    while activity ≠ terminated
      seq
        display.activity(screen, activity)
        var ch :
        pri alt
          (activity ≠ editing) & keyboard ? ch      -- provided not editing type ...
            if
              (ch = 'q') or (ch = 'Q')              -- ... Q to finish program
                activity := terminated
              (ch = 's') or (ch = 'S')              -- ... S to halt evolution
                activity := idle
              (ch = 'e') or (ch = 'E')              -- ... E to start editing
                activity := editing
              (ch = 'r') or (ch = 'R')              -- ... R to start evolution
                activity := free.running
              otherwise                             -- ... or anything else to make
                activity := single.stepping         --     just one step of evolution
          (activity = editing) & skip
            seq
              edit(keyboard, screen, control)
              activity := idle
          (activity = free.running) or (activity = single.stepping) & skip
            var changing :
            seq
              generation(screen, control, sense, changing)
              if
                (activity = single.stepping) or (not changing)
                  activity := idle
                (activity = free.running) and changing
                  skip
    display.activity(screen, activity)
    seq cell = [0 for number.of.cells]
      control[cell] ! terminate
    clean.up.display(screen)                        :
```

The alternative has to be asymmetric, because a sequence of calls to generation might otherwise go on indefinitely without ever allowing pending keyboard input to be accepted.

The single.step activity, entered by typing almost anything on the keyboard, causes an evolution of precisely one generation. This makes it easier to follow the details of a history. Notice that the code of the board is entirely unaffected by the detailed design of the single stepping mechanism, or even the details of the editor.

Each cell starts an evolutionary advance in response to an ask.state instruction, on its control

channel. Of course, it cannot complete the advance unless its neighbours are also on the move.

```
proc generation(chan screen, control[], sense[], var active) =
  seq
    seq cell = [0 for number.of.cells]
      control[cell] ! ask.state
    active := false
    seq cell = [0 for number.of.cells]
      var changed, next.state :
      seq
        sense[cell] ? changed; next.state
        if
          changed
            seq
              display.state(screen, cell mod array.width, cell / array.width, next.state)
              active := true
          not changed
            skip                    :
```

One invocation of the process generation scans the whole array once, inviting each cell to proceed with a single evolution. The new states are gathered, and any changes are notified on the display.

To settle on the details of the display, the process display.state must be supplied. Assuming a digital VT52 type terminal, and a Life board some tens on a side, I have mapped cells onto contiguous screen locations, with the first cell at the top left

```
def control = not ((not 0) << 5), escape = control /\ 'I' :

proc move.cursor(chan screen, value x, y) =
  -- move to column x of line y (of a VT52 screen)
  screen ! escape ; 'Y' ; '*s' + y ; '*s' + x :

proc display.state(chan screen, value x, y, state) =
  seq
    move.cursor(screen, x, y)
    if
      state = alive
        screen ! '**'
      state = dead
        screen ! '*s'                    :
```

A live cell shows as an asterisk, and a dead cell as a blank space.

To make the initial screen consistent with the initial state of the board, which is entirely dead, it suffices to clear the screen

```
proc initialize.display(chan screen) =
  screen ! escape ; 'H' ; escape ; 'J' :  -- clear the screen (of a VT52)
```

and to clean up at the end of the program, the cursor is moved to the left of the line below the image of the board

```
proc clean.up.display(chan screen) =
  move.cursor(screen, 0, array.height) :
```

Assuming that there is some spare room on the screen to the right of the image of the board, the activity of the controller can be displayed there

```
proc display.activity(chan screen, value activity) =
  seq
    move.cursor(screen, array.width + 1, array.height ÷ 2)
    if
      activity = idle
        write.string(screen, "Idle")
      activity = editing
        write.string(screen, "Edit")
      activity = single.stepping
        write.string(screen, "Step")
      activity = free.running
        write.string(screen, "Busy")
      activity = terminated
        write.string(screen, "Done")                    :
```

All that remains is to supply an editor. Here is a simple process that allows a cursor to be moved around the board image, and allows the state of the cell under the cursor to be set

```
proc edit(chan keyboard, screen, control[]) =

  def left.key   = ctrl ∧ 'H',   right.key  = ctrl ∧ 'L',  up.key    = ctrl ∧ 'K',
      down.key = ctrl ∧ 'J',   uproot.key = '*s',         plant.key = '**'          :

  var x, y, editing, ch :
  seq
    x := array.width ÷ 2
    y := array.height ÷ 2
    editing := true
    while editing
      seq
        move.cursor(screen, x, y)
        keyboard ? ch
        if
          (ch = left.key) and (x > 0)
            x := x - 1
          (ch = right.key) and (x < (array.width - 1))
            x := x + 1
          (ch = up.key) and (y > 0)
            y := y - 1
          (ch = down.key) and (y < (array.height - 1))
            y := y + 1
          (ch = uproot.key) or (ch = plant.key)
            var state :
            seq
              state := (dead ∧ (ch = uproot.key)) ∨ (alive ∧ (ch = plant.key))
              control[x + (array.width × y)] ! set.state; state
              display.state(screen, x, y, state)
          (ch = 'q') or (ch = 'Q')
            editing := false
          otherwise
            skip                                        :
```

Editing continues until a character 'Q' is typed. The cursor control keys move the cursor vertically and horizontally over the board, the space bar kills the occupant of a cell, and the asterisk key plants a new occupant. For simplicity, any other character, or an attempt to pass over the boundary of the board image is ignored without complaint.

Life

A brief word seems to be in order about the game of Life itself. Life first became widely known through Martin Gardner's column 'Mathematical Games' in the Scientific American magazine, in October 1970 (PP120-123) and May 1971 (PP112-117). The former article explains the rules, and introduces some of the jargon of the subject: for example, the speed of light, which is one cell width per generation, the greatest rate at which information can pass across the board; and the glider, a small, fixed size, moving colony

```
 *
  *
* * *
```

The glider is one of the small, simple colonies whose evolution is fully known: it moves across the board in the direction in which it appears to be pointing, at a quarter of the speed of light, passing through a fixed sequence of four distinct forms.

The second article describes more complicated examples, drawn from the readers' experience of wasting both machine cycles and mathematical ingenuity. Here you will find the curiosities of the subject: Garden of Eden colonies, which are ones that cannot possibly have come about as a result of an evolutionary advance from a former state; the glider gun, a huge structure which grows without limit, by firing an unending stream of gliders from one of its extremities; and a glider-gobbler which, although stable in itself, can also swallow a stream of gliders such as that given off by the gun, to no ill effect. There are viruses, which disrupt regular structures, and regular structures which can restore their symmetry after withstanding a virus attack.

Huffman minimum redundancy coding

It has become usual to store data and transmit messages using fixed length codes such as ASCII. The character set is represented by some number of codewords, each of the same length, which in the case of ASCII is seven binary digits. The result is that it takes the same number of bits to store, or the same bandwith to transmit, all messages with the same number of characters. Of course, if you know in advance that your message is in, say, English, then you know that it is much less likely to contain letter 'z's than letter 'e's. This means that if you use a shorter codeword to represent 'e' than 'z', you can expect to use less store, or bandwidth, for the average message.

In ASCII, the message 'easily' is encoded

e	a	s	i	l	y
1100101	1100001	1110011	1101001	1101100	1111001

requiring forty-two bits, whereas by using a code which included the following representations

a	⇒	1001
e	⇒	0
i	⇒	1010
l	⇒	11001
s	⇒	11010
y	⇒	1011

the same message may can be encoded

e	a	s	i	l	y
0	1001	11010	1010	11001	1011

in only twenty-one bits. The codewords must be chosen in such a way that none is a prefix of any of the others, so that there can be only one way of decoding a particular coded text.

In a classic paper, published in 1952, David Huffman described an algorithm for choosing a code that would minimize the expected length of a message, given that the probability of each character were known. Essentially, his method decides the lengths of codewords, giving the longest to the least likely characters. It then remains only to create an arbitrary unambiguous code with codewords of the right lengths.

The terminology of Huffman's paper is a little different from that in use today, as indicated in the brackets. He uses the term 'message' to mean an individual character. First of all, the ensemble [= character set] is sorted in decreasing order of probability:

> [It is] necessary that the two least probable messages [= characters] have codes [= codewords] of equal length ... [and that] there be only two of the messages with coded length $l(n)$ which are identical except for their last digits. The final digits of these two codes will be one of the two binary digits, 0 and 1. It will be necessary to assign these two message codes to the nth and $(n-1)$st messages [= two least probable characters] since at this point it is not known whether or not other codes of length $l(n)$ exist. Once this has been done, these two messages are equivalent to a single composite message. Its code (as yet undetermined) will be the common prefixes of order $l(n)-1$ of these two messages. Its probability will be the sum of the probabilities of the two messages from which it was created. The ensemble containing this composite message in the place of its two component messages will be called the first auxiliary message ensemble.
>
> This newly created ensemble contains one less message than the original. Its

members should be rearranged if necessary so that the messages are again ordered according to their probabilities. It may be considered exactly as the original ensemble was. ...

This procedure is applied again and again until the number of messages in the most recently formed auxiliary ensemble is reduced to two. One of each of the binary digits is assigned to each of these two composite messages. These messages are then combined to form a single composite message with probability unity, and the coding is complete. ...

Having now decided proper lengths of code for each message, the problem of specifying the actual digits remains. Since the combining of messages into their composites is similar to the successive confluences of trickles, rivulets, brooks, and creeks into a final large river, the procedure thus far described might be considered analogous to the placing of signs by a water-borne insect at each of these junctions as he journeys downstream. ... the code we desire is that one which the insect must remember in order to work his way back upstream.

A method for the construction of minimum-redundancy codes, David A. Huffman
in Proc I.R.E., 40 (9), September 1952, pp 1098-1101

Restated more prosaically, the final paragraph identifies the unambiguous set of codewords with a (binary) tree. Each leaf of the tree corresponds to one of the characters. The depth of that leaf, that is its distance from the root, is the length of that character's codeword. The digits of the codeword are the 'address' of the leaf, that is a sequenee of instructions for getting to the leaf from the root, say 0 for 'go to the left' and 1 for 'go to the right'.

Representing a coding tree

As usual, the task of representing a data structure in occam amounts to choosing an enumeration for the component parts, so as to map the structure onto a linear array. The structure in question this time is a binary tree similar to that in the sorting example, but this tree may be severely imbalanced, and is of unpredictable depth. This means that the simple fixed enumeration, with the children of node i being nodes (2×i)+1 and (2×i)+2, would be unreasonably wasteful of store, so is unsuitable. A better representation, in this case, uses an array children[] to record the index of the offspring of a node, so that the children of node i are indexed children[i] and children[i]+1.

If the root of the tree is taken to be the node indexed by zero

```
def root = 0 :
```

then, since the root is by definition not the child of any node,

```
children[node]  =  root
```

can be used to signify that node is a leaf of the tree. In the case of the leaves, it will be necessary to know to which character they correspond. This is most readily recorded in another array of the same size as children[] in which the value of character[node] is the character corresponding to the node, if it is a leaf.

The array children[] makes it easy to pass 'upstream' from the root of the tree to the leaves. In order to make the 'downstream' journey as efficient, it will be useful to record the inverse of children[], in an array parent[], such that

```
parent[children[node]]  =  parent[children[node]+1]  =  node
```

for each non-leaf node, and the inverse of character[] in an array representative[], which records the index of the leaf corresponding to each character.

It remains to be decided how big these arrays must be. This, of course, depends on the size of the character set being encoded. For the purposes of this example, the (unencoded) character set

will be signed, eight-bit significant values,

$$-128 \leq ch < 128$$

This allows room for the normal seven-bit characters in the non-negative half range, and room for another, negative, character set which can be used for control information, indicating such things as the end of a message.

```
def bits.in.character      = 8,
    number.of.characters = 1 << bits.in.character,
    number.of.codes      = number.of.characters,
    character.mask       = not ((not 0) << bits.in.character) :
```

The character.mask consists of bits.in.character number of one bits, and is for mapping signed characters onto non-negative array indexes, so that, for example,

```
ch      =      character[ representative[ch & character.mask] ]
```

Now if there are number.of.codes leaves in a binary tree, then there will be one less than that number of non-leaf nodes, so the total number of nodes is given by

```
def size.of.tree = (2 * number.of.codes) - 1 :
```

and the declarations of the arrays for representing the tree are

```
var children[size.of.tree],   parent[size.of.tree],
    character[size.of.tree], representative[number.of.characters] :
```

Constructing a coding tree

Huffman's algorithm proceeds in two stages. First the character set is sorted into descending order of probability of the character's occurrence. Each of the characters will correspond to a leaf of the tree, so you can think of this stage of the process as constructing number.of.codes number of leaves. These leaves will be sub-trees of the final coding tree. Since each is just a leaf, they are disjoint, in the sense that they share no nodes with each other, and they are maximal, in the sense that there is not yet any bigger tree of which any is a member.

The second stage of the algorithm repeatedly reduces the size of the collection of maximal disjoint sub-trees, by combining the two lightest trees to make one new composite tree. By 'lightest' I mean of least weight where the weight of a leaf is the probability of the corresponding character, and the weight of a larger tree is the sum of the weights of its leaves. Notice that during this second stage, it is guaranteed that any pair of siblings - children of a common parent - are already adjacent in descending order of weight.

This observation, which I take from to Robert Gallager

> A prefix condition code is a code with the property that no codeword is a prefix of any other codeword. A binary tree has the sibling property if each node (except the root) has a sibling, and if the nodes of the tree can be arranged in order of non-increasing probability with each node being adjacent to its sibling. A binary prefix condition code is a Huffman code iff the code tree has the sibling property.
>
> Variations on a Theme by Huffman, Robert G. Gallager
> In IEEE Trans. Information Theory, IT-24(6), 1978, pp 668-674

is in fact a non-algorithmic characterization of Huffman codes. It also shows that in the representation chosen for the coding tree, which allocates adjacent elements of the arrays to siblings, it is possible to keep the arrays sorted in descending order of weight. Gallager's proof

that this property holds is, essentially, an informal proof of correctness of Huffman's algorithm.

Keeping the arrays sorted by weight of node in this way simplifies the finding of the two lightest sub-trees, and if the arrays are filled from the high-index, light, end towards the root, then sub-trees once constructed need not be moved again.

I have divided the algorithm into three parts

```
proc construct.tree(value probability[]) =
  var left.limit, right.limit, weight[size.of.tree] :

  proc construct.leaves =
    ... build the leaves of the tree

  proc construct.other.nodes =
    ... join pairs of subtrees until only one tree remains

  proc invert.representation =
    ... set parent[] and representative[]

  seq
    left.limit  := size.of.tree + 1
    right.limit := size.of.tree + 1

  -- left.limit = (size.of.tree + 1)  and  (right.limit - left.limit) = 0

    construct.leaves

  -- left.limit = number.of.codes  and  (right.limit - left.limit) = number.of.codes

    construct.other.nodes

  -- left.limit = root             and  (right.limit - left.limit) = 1

    invert.representation                :
```

Throughout, the collection of maximal disjoint sub-trees consists of those trees rooted at nodes for which

$$\text{left.limit} \quad < \quad \text{node} \quad < \quad \text{right.limit}$$

The initialization of the limits makes this collection empty. The process construct.leaves introduces a new sub-tree into the collection for each of the characters of the character set, setting its weight according to the probability of the character, maintaining the arrangement of the leaves in descending order, so that

$$\text{left.limit} < i < j < \text{size.of.tree} \quad \Rightarrow \quad \text{weight}[i] > \text{weight}[j]$$

The process construct.other.nodes combines the two lightest leaves, nearest to right.limit, introducing a new node with the combined weight of these two, adjusting the limits of the collection, and filling in the shape of the tree in children[]. Finally, the process invert.representation constructs the arrays parent[] and representative[].

Each of construct.leaves and construct.other.nodes repeatedly creates a new node of some given weight, and inserts it into the right place between the limits to maintain the weight ordering of the nodes. The determination of this right place, and the consequent adjustment of the lighter nodes is done by

```
proc insert.new.node( var new.node, value weight.of.new.node,
                      var left.limit, value right.limit        ) =
  var weight.limit :
  seq
    if
      if node = [left.limit for right.limit - left.limit]
        weight[node] ≤ weight.of.new.node
          weight.limit := node
      true
        weight.limit := right.limit
    seq node = [left.limit for weight.limit - left.limit]
      seq
        character[node - 1] := character[node]
        children[node - 1]  := children[node]
        weight[node - 1]    := weight[node]
      left.limit := left.limit - 1
      new.node := weight.limit - 1
      weight[new.node] := weight.of.new.node                  :
```

Recall that the collection of maximal disjoint sub-trees of the coding tree so far constructed consists of those rooted at nodes

$$\text{left.limit} \leq \text{node} < \text{right.limit}$$

and that they are in descending order of weight. This means that the conditional sets the weight.limit so that

$$\text{left.limit} \leq \text{node} < \text{weight.limit} \Rightarrow \text{weight[node]} > \text{weight.of.new.node}$$
$$\text{weight.limit} \leq \text{node} < \text{right.limit} \Rightarrow \text{weight.of.new.node} \geq \text{weight[node]}$$

The sequential loop then displaces each of the heavier nodes one place to the left to make room for the new.node, and the left.limit of the collection is adjusted to compensate. This shift does not make it necessary to adjust any of the values in children[] because

$$\text{node} < \text{weight.limit} \Rightarrow \text{node} < \text{right.limit}$$

and the tree is so constructed that

$$(\text{children[node]} = \text{root}) \text{ or } (\text{children[node]} > \text{right.limit})$$

so that none of the nodes being moved is yet a child.

Using this process, insert.new.node, the process that creates the leaf nodes can be written

```
proc construct.leaves =
  def minimum.character = - (number.of.characters ÷ 2) :
  seq ch = [minimum.character for number.of.characters]
    var new.node :
    seq
      insert.new.node(new.node, probability[ch ∧ character.mask], left.limit, right.limit)
      children[new.node]  := root
      character[new.node] := ch                                :
```

This inserts a new leaf into the collection, increasing the size of the collection by decreasing the left.limit. The process to combine the leaves into a tree

```
proc construct.other.nodes =
  while (right.limit - left.limit) ≠ 1
    var new.node :
    seq
      right.limit := right.limit - 2
      insert.new.node(new.node, weight[right.limit] + weight[right.limit+1],
                                                    left.limit, right.limit)
      children[new.node] := right.limit                                    :
```

first removes the two lightest sub-trees from the collection, by decreasing right.limit, then joins them under a parent whose weight is the sum of their individual weights. Notice that the assignment to children[new.node] maintains the property that there are no children to the left of the right.limit. The process is complete when only one tree remains.

Inverting the representation of the tree is a simple task, which involves assigning to representative[] the indexes of the leaf nodes, and to parent[] the indexes of the nodes that are not leaves, thus

```
proc invert.representation =
  seq node = [root for size.of.tree]
    if
      children[node] = root
        representative[character[node] ∧ character.mask] := node
      children[node] ≠ root
        seq child = [children[node] for 2]
          parent[child] := node                                            :
```

Encoding and decoding using a coding tree

The encoding of any given character ch is the sequence of 'go left' and 'go right' instructions that Huffman's insect must follow to pass upstream from the root node to the representative node of that character. It is easy enough to construct this code backwards, since floating downstream involved passing from node to parent[node] in succession from the representative node until the root is reached. The process

```
seq
  length := 0
  node := representative[ch ∧ character.mask]
  while node ≠ root
    seq
      encoding[length] := node - children[parent[node]]
      length := length + 1
      node := parent[node]
```

establishes the condition that

$$\forall\, i.\ 0 \leqslant i < \text{length} \quad \Rightarrow \quad \text{node}(i) = (\text{children}[\text{node}(i+1)] + \text{encoding}[i])$$

$$\text{where} \quad \text{node}(0) = \text{representative}[\text{ch} \wedge \text{character.mask}]$$
$$\text{node}(\text{length}) = \text{root}$$

so that the encoding of ch can be transmitted in the right order by

```
seq i = [1 for length]
  output ! encoding[length - i]
```

It remains only to decide how much room needs to be allocated to store the encoding[] whilst it is being constructed. Assume that you are decoding a Huffman encoded character. Before you receive the first bit of the encoding, there are number.of.codes possible codes that you might be about to receive. Each bit that you receive divides the set of possible characters into two non-empty sub-sets, those that are still possible, those that are now precluded. This means that at most number.of.codes−1 bits will suffice. In fact, in the worst case, this limit is achieved: if each character is twice as probable as the next most probable, then the Huffman codes are, in decreasing order of probability

0, 10, 110, 1110, 11110, ...

with the two least probable characters both having encodings number.of.codes−1 bits long. With this knowledge, the encoding process is written

```
proc encode.character(chan output, value ch) =
  -- Transmit the encoding of ch along output
  def size.of.encoding = number.of.codes - 1 :
  var encoding[size.of.encoding], length, node :
  seq
    length := 0
    node   := representative[ch ∧ character.mask]
    while node ≠ root
      seq
        encoding[length] := node - children[parent[node]]
        length := length + 1
        node   := parent[node]
    seq i = [1 for length]
      output ! encoding[length - i]                :
```

Decoding a stream of bits to determine the character consists of following the 'go left' and 'go right' instructions as they arrive, passing 'upstream' from the root node until a leaf is reached. That leaf indicates the decoded character

```
proc decode.character(chan input, var ch) =
  var node :
  seq
    node := root
    while children[node] ≠ root
      var bit :
      seq
        input ? bit
        node := children[node] + bit
    ch := character[node]                :
```

I will assume that the probabilities of the characters are fixed in advance, say by considering an average over many messages of the type to be sent.

```
def probability = table[ ... ] :   -- indexed by [0 for number.of.characters]
```

In order to keep all the arithmetic in integers, the probabilities should be scaled and rounded so that the total of the 'probabilities'

$$\sum_{ch} \text{probability}[ch]$$

is a large integer. If it is possible to read the message through before sending it, then you can count actual character frequencies, and produce an optimal Huffman code for the message, but of course, you will have to transmit a description of the code with your message!

If one of the character codes is laid aside to indicate the end of the transmitted message, then

```
def end.of.message = -1 :

proc copy.encoding(chan source, end.of.source, sink) =
  -- Read characters from source, sending their encodings along
  -- sink, until a signal is received along end.of.source.
  var more.characters.expected :
  seq
    construct.tree(probability)
    more.characters.expected := true
    while more.characters.expected
      var ch :
      alt
        source ? ch
          encode.character(sink, ch)
        end.of.source ? any
          more.characters.expected := false
    encode.character(sink, end.of.message)          :
```

will translate a stream of characters into a stream of bits representing their Huffman encodings, and mark the end of the stream by sending the encoding of end.of.message. The corresponding decoding process would be

```
proc copy.decoding(chan source, sink) =
  -- Read a bit stream from source, decoding it into characters
  -- and send these along sink until end.of.message is decoded
  var more.characters.expected :
  seq
    construct.tree(probability)
    more.characters.expected := true
    while more.characters.expected
      var ch :
      seq
       decode.character(source, ch)
       if
         ch ≠ end.of.message
           sink ! ch
         ch = end.of.message
           more.characters.expected := false    :
```

These processes can be used at the opposite ends of a serial communications medium

```
proc copy.over.serial.medium(chan source, end.of.source, sink) =
  -- Copy characters from source to sink until end.of.source
  chan serial.medium :
  par
    copy.encoding(source, end.of.source, serial.medium)
    copy.decoding(serial.medium, sink)                    :
```

or a blocked medium, such as a magnetic tape. Here is a process for encoding a message and

packing it into blocks, using a component from the 'Programming structures' section,

```
proc encode.into.blocks(chan source, end.of.source, block.sink) =
  chan bit.stream, end.of.bit.stream :
  par
    seq
      copy.encoding(source, end.of.source, bit.stream)
      end.of.bit.stream ! any
    pack.bits.into.blocks(bit.stream, end.of.bit.stream, block.sink)  :
```

Decoding the characters from the stream of blocks is a slightly trickier task, since the end of the message is determined by the decoded data. The most elegant solution, as seems common in parallel programs, involves a process that throws away unwanted information

```
proc discard(chan source, end.of.source) =
  var more.expected :
  seq
    more.expected := true
    while more.expected
      alt
        source ? any
          skip
        end.of.source ? any
          more.expected := false       :
```

This inputs successively from source, ignoring the values that it receives, until a signal is sent to it on end.of.source. With this, the process for decoding the bits in a stream of blocks can be written

```
proc decode.from.blocks(chan block.source, sink) =
  chan end.of.block.source, bit.stream, end.of.bit.stream :
  par
    seq
      unpack.bits.from.blocks(block.source, end.of.block.source, bit.stream)
      end.of.bit.stream ! any                          -- 'feed-forward'

    seq
      copy.decoding(bit.stream, sink)
      par
        discard(bit.stream, end.of.bit.stream)
        end.of.block.source ! any              :     -- 'feed-back'
```

When copy.decoding decodes an end.of.message it terminates, causing a signal to be offered for output on end.of.block.stream, which is a feed-back path to the block unpacking process. At the same time, discard is absorbing any bits that were left in the last block of the message. When all of the bits of the last block have gone, unpack.bits.from.blocks accepts the end.of.block.source signal, and terminates, causing an end.of.bit.stream signal to be sent to terminate the discard process.

Adapting the code to the message

So far, I have accepted Huffman's assumption that the code is predetermined and remains fixed throughout the transmission of a given message. This is reasonable in case the probability distribution of the characters in the message is known in advance, or if the message can be read through in advance. Gallager suggests an alternative encoding that tends in the long run towards

the fixed Huffman encoding, but which starts with no knowledge of the probability distribution of the characters, and adapts the code as the message is being sent.

Each character is encoded with a Huffman code that would be optimal for a message consisting of all those characters that have gone before it. This encoding technique has the startling property that, since the decoder has already decoded the preceding characters, it can deduce from the received message what code should be used to decode each character. There is no longer a problem in communicating the code as well as the message!

As I have presented it, it might seem that Gallager's adaptive Huffman coder requires that a new coding tree be constructed for each character of the transmitted and received message. Fortunately, this is not the case: the accumulated character frequencies change little, so the shape of the tree tends to settle down; successive trees are sufficiently similar that it is fairly easy to construct each from its predecessor.

The idea is to write a process increment.frequency(ch) which modifies the coding tree so as to be consistent with a frequency distribution with one more occurrence of the character ch than previously. The encoding process becomes

```
proc copy.encoding(chan source, end.of.source, sink) =
  -- Read characters from source, sending their encodings along
  -- sink, until a signal is received along end.of.source.
  var more.characters.expected :
  seq
    construct.tree
    more.characters.expected := true
    while more.characters.expected
      var ch :
      alt
        source ? ch
          seq
            encode.character(sink, ch)
            increment.frequency(ch)
        end.of.source ? any
          more.characters.expected := false
    encode.character(sink, end.of.message)          :
```

and the corresponding decoding process would be

```
proc copy.decoding(chan source, sink) =
  -- Read a bit stream from source, decoding it into characters
  -- and send these along sink until end.of.message is decoded
  var more.characters.expected :
  seq
    construct.tree
    more.characters.expected := true
    while more.characters.expected
      var ch :
      seq
       decode.character(source, ch)
       if
         ch ≠ end.of.message
           seq
             sink ! ch
             increment.frequency(ch)
         ch = end.of.message
           more.characters.expected := false    :
```

To keep track of the accumulated frequencies, the weight[] must become a permanent part of the representation of the tree

```
var weight[size.of.tree] :
```

In order to increment the recorded frequency of a character, it is necessary to increment the weight of its representative leaf

```
var node :
seq
  node := representative[ch ∧ character.mask]
  weight[node] := weight[node] + 1
```

There are two ways in which this may have damaged the structure of the tree. First of all, unless the tree has only the one node, the weight of the parent of node is no longer the sum of the weights of its children: it will be necessary to increment the weights of the parent of the node, and all of its ancestors up to the root

```
var node :
seq
  node := representative[ch ∧ character.mask]
  while node ≠ root
    seq
      weight[node] := weight[node] + 1
      node := parent[node]
  weight[root] := weight[root] + 1
```

Secondly, each time the weight of a node, be that the original leaf or one of its ancestors, is increased there is a danger that the ordering of the weights may be upset. If this is the case then it is time to reorganize the tree, and change the encoding.

Assuming that the tree is initially properly ordered, then the ordering will first fail when

$$weight[node-1] = weight[node]$$

and the weight of node is about to be incremented. Now, the trees rooted at nodes of equal weight must be disjoint trees, that is either the nodes are siblings, or they have ancestors which are siblings. This follows from the fact that the weight of a node is always less than that of its ancestors, and greater than that of its descendants, so another node with the same weight is neither an ancestor nor a descendant.

To preserve the ordering on the nodes, you could try exchanging the trees rooted at node and node−1, and then try to increment the weight of the light node in its new position. Since there might be many nodes with the same weight, however, you would have to do this repeatedly, shuffling the imminently overweight node leftwards in the tree.

```
while weight[node - 1] = weight[node]
  seq
    swap.trees(node, node - 1)
    node := node - 1
```

An alternative solution is to look for the leftmost node of the given weight, and exchange with that node, directly. The same argument about the weight of a node being less than that of its ancestors shows that there is always a sequence of nodes for which

$$weight[(node - i) - 1] > weight[node - i] = \ldots = weight[node]$$

This leftmost node, indexed node − i, is identified, and the exchange performed, by

```
if i = [1 for (node − root) − 1]
  weight[(node − i) − 1] > weight[node]
    seq
      swap.trees(node, node − i)
      node := node − i
```

Having moved the node, it is possible to increment its weight, and that of each of its ancestors.

```
var node :
seq
  node := representative[ch ∧ character.mask]
  while node ≠ root
    if
      weight[node−1] > weight[node]
        seq
          weight[node] := weight[node] + 1
          node := parent[node]
      weight[node−1] = weight[node]
        if i = [1 for (node − root) − 1]
          weight[(node − i) − 1] > weight[node]
            seq
              swap.trees(node, node − 1)
              node := node − i
  weight[root] := weight[root] + 1
```

The process for exchanging a pair of disjoint sub-trees is simply coded

```
proc swap.trees(value i, j) =
  -- Exchange disjoint sub-trees rooted at i and j

  proc swap.words(var p, q) =
    -- Exchange values stored in p and q
    var t :
    seq
      t := p
      p := q
      q := t                :

  proc adjust.offspring(value i) =
    -- Restore downstream pointers to node i
    if
      children[i] = root
        representative[character[i] ∧ character.mask] := i
      children[i] ≠ root
        seq child = [children[i] for 2]
          parent[child] := i                                  :

  seq
    swap.words(children[i], children[j])
    swap.words(character[i], character[j])
    adjust.offspring(i)
    adjust.offspring(j)                                       :
```

First, the 'upstream' pointers, children[] and character[], to the nodes are exchanged, then the process adjust.offspring restores the 'downstream' pointers that are no longer correct. There is, of course, no need to exchange the weights of the nodes, since they are known to be equal.

The only remaining problem is to decide the shape of the initial coding tree: what encoding should be used to send the first character? The simplest solution would be to construct the initial tree on the assumption that all characters are equally likely to turn up, that is

```
children[node] = root    ⇒    weight[node] = 1
```

This means that, to begin with, the code is a fixed length one, each character being encoded by bits.in.character number of bits.

An alternative technique is to keep in the coding tree only representations of characters that have actually been sent and received. Whenever a character is to be sent for the first time in the message, the code of a special escape character is sent, followed by some standard representation of the new character, say its ASCII code. A new leaf must then be added to the tree to represent the new character.

In order to accommodate the escape character, the space allocated for the tree must be enlarged

```
def number.of.codes = number.of.characters + 1 :
```

and, since the tree grows, some way must be found of recording that size. As each escape is the representation of a character that has never occurred at all (you may not yet know which character, but you do know this), it should be given a very low weight. This means that it is reasonable to represent it by the rightmost (least likely) leaf of the tree. Doing this means that a single variable

```
var escape :
```

serves the purpose of recording which node represents the escape, and which is the rightmost node of the tree.

Since the value of escape changes, it will not do as an initial value for representative[]. Define, instead,

```
def not.a.node = size.of.tree :
```

then creating the initial tree is just a matter of making the escape leaf, and initializing the array of representatives

```
proc construct.tree =
  seq
    escape := root
    weight[escape]   := 1          -- minimum legal weight
    children[escape] := root       -- it is a leaf
    seq ch = [0 for number.of.characters]
      representative[ch] := not.a.node              :
```

Encoding using the new tree is substantially unchanged, excepting in that some provision must he made for sending escaped characters. First of all, the encoding is potentially larger by the bits.in.character number of bits in the unencoded representation, so

```
def size.of.encoding = bits.in.character + (number.of.codes - 1) :
```

The bits of the unencoded character representation can then be stored before the encoding of

escape, to be transmitted after it.

```
proc encode.character(chan output, value ch) =
  -- Transmit the encoding of ch along output
  def size.of.encoding = bits.in.character + (number.of.codes - 1) :
  var encoding[size.of.encoding], length, node :
  seq
    if
      representative[ch /\ character.mask] <> not.a.node
        seq
          length := 0
          node   := representative[ch /\ character.mask]
      representative[ch /\ character.mask] = not.a.node
        seq
          seq i = [0 for bits.in.character]
            encoding[i] := (ch >> i) /\ 1        -- i'th bit of unencoded ch
          length := bits.in.character
          node   := escape
    while node <> root
      seq
        encoding[length] := node - children[parent[node]]
        length := length + 1
        node   := parent[node]
    seq i = [1 for length]
      output ! encoding[length - i]              :
```

The very first character to be sent will be escaped, and since the representative node for escape is initially root the encoding of the escape will be the null sequence of bits. This means that the first transmitted bit will be the first bit of the unencoded character representation.

Decoding is also as before, excepting that on receipt of the encoding of escape, the bits of the unencoded escaped character must be read and the character reassembled

```
proc decode.character(chan input, var ch) =
  -- Receive an encoding along input and store the character in ch
  var node :
  seq
    node := root
    while children[node] <> root
      var bit :
      seq
        input ? bit
        node := children[node] + bit
    if
      node < escape
        ch := character[node]
      node = escape
        var bit :
        seq
          input ? bit
          ch := - bit
          seq i = [2 for bits.in.character - 1]
            seq
              input ? bit
              ch := (ch << 1) \/ bit               :
```

The first bit of an escaped sequence is the sign bit of the character code, so the assignment

```
ch := - bit
```

extends the sign bit to the left, and the loop shifts the subsequent bits in from the right.

In order to increment the frequency of a character not yet in the tree, it is necessary to be able to construct a new leaf to be the representative of the new character. This process divides the escape leaf into two leaves and their parent, thus

```
proc create.leaf(var new.leaf, value ch) =
  -- Extend the tree by fision of the escape leaf into two new leaves
  var new.escape :
  seq
    new.leaf              := escape + 1
    new.escape            := escape + 2

    children[escape]      := new.leaf      -- escape is the new parent

    weight[new.leaf]      := 0
    children[new.leaf]    := root
    parent[new.leaf]      := escape
    character[new.leaf]   := ch
    representative[ch ∧ character.mask] := new.leaf

    weight[new.escape]    := 1
    children[new.escape]  := root
    parent[new.escape]    := escape

    escape                := new.escape          :
```

The new leaf has no weight when created, so does not affect the weights of its ancestors. Its weight must be incremented just as for any other leaf

```
proc Increment.frequency(value ch) =
  var node :
  seq
    if
      representative[ch ∧ character.mask] ≠ not.a.node
        node := representative[ch ∧ character.mask]
      representative[ch ∧ character.mask] = not.a.node
        create.leaf(node, ch)
    while node ≠ root
      if
        weight[node - 1] > weight[node]
          seq
            weight[node] := weight[node] + 1
            node := parent[node]
        weight[node - 1] = weight[node]
          if i = [1 for (node - root) - 1]
            weight[(node - 1) - 1] > weight[node]
              seq
                swap.trees(node, node - 1)
                node := node - 1
    weight[root] := weight[root] + 1                    :
```

Notice that a brand new leaf having no weight, the data invariant - that no node has the same weight as its parent - is breached by the escape node and its parent. In order to show that the tree exchanging is correct, the statement of this invariant must be strengthened: no node, excepting the escape node has the same weight as its parent. This is sufficient, because you will never require to exchange with the degenerate tree rooted at escape.

That completes the adaptive coder. Notice that, since the processes copy.encoding and copy.decoding have the same interfaces as the corresponding processes in the fixed-code coder, they may be substituted into the example programs. There is no need to change the processes that convey the bit stream from encoder to decoder.

If you have read what went before, then you may think that I have been trying to tell you how to write concurrent programs. Be sure that others will always have different ways. There are some decisions which it is my weakness to need to justify before closing.

Sequential or parallel?

Excepting for the parallel matrix multiplier, these programs were all written for execution on a single processor computer. This has affected the design, for example, in places where either seq or par would have done, I have tended to write the former, in the knowledge that it is 'cheaper' on such a machine.

Sequentially composed processes can rely on the state left behind by their predecessors, and to write par would be to imply that you were not relying on such residual state. Concurrently composed processes can rely on being able to communicate with their contemporaries, and to write seq would be to imply that you were not using such communications. If you required neither sequencing nor guaranteed contemporaneousness, then the choice between seq and par could only be made on grounds of efficiency (or whim).

By the skin of my conscience, I shall avoid making this into an argument for an ambiguous constructor, which might be translated either into seq or into par as the implementor would see fit - he being best able to judge relative efficiencies (and just as capable of whimsey).

Folding

Experienced occam programmers who have used the tools provided by inmos to support occam programming, tools such as the occam programming system (ops), may find my programs unexpectedly rich in proc declarations. The ops editor has a text structuring capability called 'folding' which, by rolling up a whole screenful of program onto a single line of the terminal screen, allows you to consider the structure of a very large piece of code a little at a time. ("What you see is what you are thinking about.")

The tendency is, when writing programs with a folding editor, to write proc bodies in-line at the point of call, and to fold the text to keep it in manageable chunks. Given the support tools, this is as good a way, if not a better one, of modularizing the code. My excuse for using proc declarations here is that the technology of hierarchical folding, although described in terms of a paper metaphor, is altogether less successful on paper, and makes binding the book rather difficult.

Typing

Finally, there is the matter of data typing. Few self-respecting authors of tutorial papers on programming style would now choose an untyped language like bcpl as their vehicle. Looking back over the descriptions of programs here, there seems to be a great deal of argument given over to the basic data types, such as arrays of bits, and trees. Much, although not all, of this could be factored out by adopting some variable and channel typing scheme from a sequential language, such as pascal. My excuse for not doing so is that the designers of occam, with laudable caution, have yet to make this leap themselves, and I am loath to go before them. This matter is addressed in a language christened, with originality, occam 2.0, to which I trust you will be able in due course to adapt any good ideas which you may have found here.

A number of people have contributed to this monograph beside myself, although they may not all have been aware of doing so. I thank particularly: Tony Hoare for showing me the light; Paul Fertig, Michael Goldsmith, and Bernard Sufrin for drawing my attention to those of its failings which I was prepared to admit.

Codes of the programs

```
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]          :



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
  var tens, width :                 -- tens will be a signed power of ten
  seq
    if
      n > 0
        seq
          tens  := -1
          width := 1                -- count a minimum of one digit
      n < 0
        seq
          tens  := 1
          width := 2                -- count a sign and a minimum of one digit

    while (n ÷ tens) <= (- 10)      -- set tens so that  0 < (- (n ÷ tens)) <= 10
      seq                           -- or, if n = 0 then tens = 1
        tens  := 10 × tens
        width := width + 1

    while width < field.width       -- pad with spaces to field.width characters
      seq
        output ! '*s'
        width := width + 1

    if                              -- output a sign for negative n
      n > 0
        skip
      n < 0
        output ! '-'

    while tens ≠ 0                  -- output the digits of n, most significant first
      seq
        output ! '0' - ((n ÷ tens) mod 10)
        tens := tens ÷ 10                       :
```

```
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

  def min = not ((not 0) >> 1), max = (not 0) >> 1 :
  def otherwise = true :

  var ch, sign :
  seq

    input ? ch
    while ch = '*s'                    -- skip leading spaces
      input ? ch

    if
      (ch = '+') or (sign = '-')       -- read a possible sign
        seq
          sign := ch
          input ? ch
      (ch ≠ '+') and (sign ≠ '-')
        sign := '+'

    while ch = '*s'                    -- skip any spaces after the sign
      input ? ch

    n := 0
    ok := ('0' ≤ ch) and (ch ≤ '9')    -- check for the presence of digits

    while ('0' ≤ ch) and (ch ≤ '9')    -- and read a sequence of them
      seq
        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                -- number out of representable range
        input ? ch                                  :
```

```
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.

  def control    = not ((not 0) << 5),
      otherwise  = true,
      backspace  = control /\ 'H',
      bell       = control /\ 'G',
      cancel     = control /\ 'U',
      delete     = not ((not 0) << 7),
      max.length = not ((not 0) << 8) :

  seq
    s[byte 0] := 0                          -- byte zero contains the length of the string
    while s[byte s[byte 0]] <> '*C'
      var ch :
      keyboard ? ch
      if
        ('*S' <= ch) and (ch < delete) and (s[byte 0] < (max.length - 1))
          seq                               -- 'printable' characters are
            screen ! ch                     --     echoed
            s[byte 0] := s[byte 0] + 1
            s[byte s[byte 0]] := ch         --     and added to the string
        ch = '*C'
          seq                               -- carriage return
            s[byte 0] := s[byte 0] + 1      --    is added to the string
            s[byte s[byte 0]] := ch         --    and terminates the loop
        (ch = backspace) and (s[byte 0] > 0)
          seq                               -- backspace
            screen ! backspace ; '*S' ; backspace   --   overwrites the last character echoed
            s[byte 0] := s[byte 0] - 1      --   and removes it from the string
        ch = cancel
          while s[byte 0] > 0               -- cancel
            seq                             --   back-spaces over the whole line
              screen ! backspace ; '*S' ; backspace
              s[byte 0] := s[byte 0] - 1
        otherwise                           -- anything else is an error
          screen ! bell                     :
```

```
def type.ahead = ...,  control = not ((not 0) << 5),  release = -1 :


proc keyboard.handler(chan request, sink, error) =
  -- Characters typed at the keyboard can be read from sink.
  -- A signal is required on request before each item is read.
  -- If more than type.ahead are typed-ahead, there is an error signal.
  chan keystrokes.in at ... :
  var reader, writer, count :
  seq
    reader := 0                          -- index of next item to be read from buffer
    writer := 0                          -- index of next free location in buffer
    count  := type.ahead                 -- number of spare locations in buffer
    var datum[type.ahead] :
    while true
      alt
        count = 0   &   keystrokes.in ? any          -- if something typed but no room
          error ! any                                --    then signal an error
        count > 0   &   keystrokes.in ? datum[writer] -- if something typed when room
          seq
            writer := (writer + 1) mod type.ahead    --   then store it in the buffer
            count  := count - 1
        count < type.ahead   &   request ? any       -- if something requested
          seq
            sink ! datum[reader]                     --   then read from the buffer
            reader := (reader + 1) mod type.ahead
            count  := count + 1                  :


proc echo.handler(chan request, reply, echo, inward) =
  def enter = control! ^ 'M' :
  while true
    var ch :
    seq
      request ! any
      reply ? ch
      inward ! ch                        -- Transmit character to user
      if
        (' ' < ch)  and  (ch < '~')
          echo ! ch                      -- Send visible input back to terminal screen
        ch = enter
          echo ! release                 -- Release screen at end of line of input
        true
          skip                                   :
```

```
proc output.multiplexer(chan from[], value width, chan outgoing) =
  while true
  var ch :
  alt selected.process = [0 for width]
    from[selected.process] ? ch          -- take a message from any from channel
      while ch ≠ release                 -- and copy it to completion
        seq
          outgoing ! ch
          from[selected.process] ? ch                :
```

```
proc screen.handler(chan outgoing, error) =
  def bell.character = control ∧ 'G' :
  chan screen.out at ... :
  while true
    var ch :
    pri alt
      error ? any                        -- signal errors by ringing the bell
        screen.out ! bell.character
      outgoing ? ch                      -- and send on outgoing characters
        screen.out ! ch                  :
```

```
proc user(chan terminal.keyboard, terminal.screen) =
  ...
```

```
def  from.echo.handler = 0, from.user = 1, number.of.outputs = 2 :
chan outgoing, from.keyboard, to.screen[number.of.outputs] :

pri par

  chan request, reply, error :       -- High priority process
  par
    keyboard.handler(request, reply, error)
    echo.handler(request, reply, to.screen[from.echo.handler], from.keyboard)
    screen.handler(outgoing, error)

  par                                -- Low priority process
    output.multiplexer(to.screen, number.of.outputs, outgoing)
    user(from.keyboard, to.screen[from.user])
```

```
proc produce.xj(value j, chan south) =          -- north row: source of X values
  while true
    south ! any                    :

proc consume.yi(value i, chan east) =           -- west column: sink for Y values
  while true
    east ? any                     :

proc offset(value ki, chan west) =              -- east column: source of k offsets
  while true
    west ! ki                      :

proc multiplier(value aij, chan north, south, west, east) =
  var xj, aij.times.xj, yi :                    -- middle: responsible for a values
  seq
    north ? xj
    while true
      seq
        par
          south ! xj
          aij.times.xj := aij × xj
          east ? yi
        par
          west ! yi + aij.times.xj
          north ? xj                                    :

proc sink(chan north) =                         -- south row: sink for unused outputs
  while true
    north ? any        :

def n = 3 :
var a[n × n], k[n] :
seq
  -- initialise a and k

  chan north.south[n × (n + 1)], east.west[n × (n + 1)] :
  par
    par j = [0 for n]
      produce.xj(j, north.south[j])

    par i = [0 for n]
      offset(k[i], east.west[(n × n) + i])
    par i = [0 for n]
      par j = [0 for n]
        multiplier( a[(n × i) + j],
                    north.south[(n × i) + j], north.south[(n × (i + 1)) + j],
                    east.west   [i + (n × j)], east.west   [i + (n × (j + 1))] )
    par j = [0 for n]
      sink(north.south[(n × n) + j])

    par i = [0 for n]
      consume.yi(i, east.west[i])
```

```
proc fork.distribute(chan up, left.up, right.up) =
  -- share out a sequence of numbers as two sequences, to the left, to the right
  def leftward = 0, rightward = not leftward :
  var more, inclination :
  seq
    inclination := leftward
    up ? more
    while more
      var number :
      seq
        up ? number
        if
          inclination = leftward
            left.up ! true; number
          inclination = rightward
            right.up ! true; number
        up ? more
        inclination := not inclination
    par
      left.up ! false
      right.up ! false                          :


proc fork.gather(chan down, left.down, right.down) =
  -- merge two ascending sequences, from left and right, into one ascending sequence
  var left.more, left.minimum, right.more, right.minimum :
  seq
    par
      left.down ? left.more; left.minimum
      right.down ? right.more; right.minimum
    while left.more or right.more
      if
        left.more and ((not right.more) or (left.minimum < right.minimum))
          seq
            down ! true; left.minimum
            left.down ? left.more; left.minimum
        right.more and ((not left.more) or (left.minimum > right.minimum))
          seq
            down ! true; right.minimum
            right.down ? right.more; right.minimum
    down ! false; any                                   :


proc fork(chan up, down, left.down, left.up, right.down, right.up) =
  -- actions for a medial node in the sorting tree
  seq
    fork.distribute(up, left.up, right.up)
    fork.gather(down, left.down, right.down)  :
```

```
def display.number = 1, display.empty = 2, display.stop = 3 :


proc leaf(chan up, down, probe) =
  -- actions for a terminal node in the sorting tree
  var number :
  seq
    up ? any; number                    -- expect a sequence of one number
    probe ! display.number; number      -- pass the number to the monitoring code
    up ? any
    down ! true; number                 -- return it as an ascending sequence
    probe ! display.empty               -- indicating its departure
    down ! false; any
    probe ! display.stop          :


proc monitor(chan up.a, down.a, up.b, down.b, probe) =
  -- in-channel monitoring code, in the form of a buffer
  seq
    var more :
    seq                           -- first watch an upward-bound sequence of values
      up.a ? more
      while more
        var number :
        seq
          up.a ? number
          probe ! display.number; number
          up.b ! more; number
          probe ! display.empty
          up.a ? more
      up.b ! more
    var more, number :
    seq                           -- then watch a downward-bound sequence
      down.a ? more; number
      while more
        seq
          probe ! display.number; number
          down.b ! more; number
          probe ! display.empty
          down.a ? more; number
      down.b ! more; number
      probe ! display.stop                    :
```

```
def depth.of.tree = 4 :

def number.of.leaves     = 1 << depth.of.tree ,
    number.of.forks      = number.of.leaves - 1 ,
    number.of.processes  = number.of.forks + number.of.leaves ,
    number.of.channels   = number.of.processes ,
    number.of.probes     = number.of.channels + number.of.leaves :


proc make.cartesian(value index, var x, y) =
  -- turn a probe index into Cartesian co-ordinates in a terminal-independent space
  if
    if line = [1 for depth.of.tree + 1]
      index < ((1 << line) - 1)            -- then probe is from a channel at this depth
        var c :
        seq
          c := index - ((1 << (line - 1)) - 1)
          x := ((2 x c) + 1) x (number.of.leaves >> (line - 1))
          y := line
    index >= number.of.channels            -- then probe is from a leaf
      seq
        x := (2 x (index - number.of.channels)) + 1
        y := depth.of.tree + 2                          :



def field.width = 3 :


proc independent(chan source, sink) =
  var instruction :
  seq
    source ? instruction
    while instruction <> display.stop
      seq                                          -- turn every probe signal into ...
        sink ! true                                -- ... a true value
        var index, x, y :
        seq
          source ? index
          make.cartesian(index, x, y)
          sink ! x; y                              -- ... a co-ordinate-pair
        if                                         -- and field.width number of characters:
          instruction = display.number
            var number :
            seq
              source ? number
              write.signed(sink, number, field.width)      -- either a numeral
          instruction = display.empty
            seq i = [0 for field.width]
              sink ! '*s'                                  -- or that many blanks
        source ? instruction
    sink ! false                           :
```

```
def virtual.height = depth.of.tree + 1, virtual.width = (2 × number.of.leaves) − 1 :


proc dependent(chan source, terminal) =
  -- terminal dependent code for driving a VT52

  def screen.height = 24, screen.width = 80 :

  def control = not ((not 0) << 5), escape = control ∧ 'I' :

  proc clear.screen(chan terminal) =
    -- clear screen sequence for a VT52
    terminal ! escape ; 'H' ; escape ; 'J'          :

  proc goto.xy(chan terminal, value x, y) =
    -- lefthanded co-ordinates, origin 0, 0 at top left
    terminal ! escape ; 'Y' ; '%s' + y ; '%s' + x    :

  var more :
  seq
    clear.screen(terminal)
    source ? more
    while more
      seq
        var x, y :
        seq
          source ? x; y
          goto.xy(terminal, (x − 1) × (screen.width ÷ virtual.width),
                            (virtual.height − y) × (screen.height ÷ virtual.height))
        seq i = [1 for field.width]
          var ch :
          seq
            source ? ch
            terminal ! ch
        source ? more
    goto.xy(terminal, 0, screen.height − 1)          :
```

```
proc display(chan source, sink) =
  chan internal :
  par
    independent(source, internal)
    dependent(internal, sink)      :
```

```
proc multiplex(chan probe[], all.probes) =
  -- gather all probe signals onto a single channel
  var more, more.from[number.of.probes] :
  seq
    more := number.of.probes
    seq i = [0 for number.of.probes]
      more.from[i] := true
    while more > 0                              -- while not all probes are dead
      var instruction :
      alt i = [0 for number.of.probes]
        more.from[i] & probe[i] ? instruction   -- take a probe instruction
          if
            instruction = display.number        -- if this is a number
              var number :
              seq
                probe[i] ? number                       -- copy the number, and tag
                all.probes ! instruction; i; number     -- it with the probe number
            instruction = display.empty         -- if this is a blank instruction
              all.probes ! instruction; i               -- tag it with the probe number
            instruction = display.stop          -- if the probe is dead
              seq
                more.from[i] := false           -- then expect no more signals from it
                more := more - 1                -- and decrease the count of working ones
    all.probes ! display.stop                   :
```

```
proc driver(chan up, down) =

  def mask = not ((not 0) << 9) :

  proc shift(var state) =
    seq i = [1 for 9]
      state := ((state << 1) ∧ mask) ∨ (((state >> 4) ⊕ (state >> 8)) ∧ 1)  :

  seq
    var event, number :                          -- first fill the tree
    seq
      time ? event
      number := (event ∧ mask) ∨ 1               -- initialize the random number
      seq i = [0 for number.of.leaves]
        seq
          event := event + second
          shift(number)                          -- pick a new number
          up ! true; number                      -- send it into the tree
          time ? after event                     -- and wait for a second before the next
      up ! false
    var event :                                  -- then empty the tree
    seq
      time ? event
      seq i = [0 for number.of.leaves]
        seq
          event := event + second
          down ? any; any                        -- take a number from the tree
          time ? after event                     -- once a second
      down ? any; any                  :


def root = 0 ,
    first.fork = root ,
    first.leaf = first.fork + number.of.forks :

chan up.a[number.of.channels], down.a[number.of.channels],
     up.b[number.of.channels], down.b[number.of.channels],
     probe[number.of.probes], all.probes  :

par
  driver(up.a[root], down.b[root])

  par i = [first.fork for number.of.forks]
    fork(up.b[i], down.a[i], down.b[(2×i)+1], up.a[(2×i)+1], down.b[(2×i)+2], up.a[(2×i)+2])

  par i = [first.leaf for number.of.leaves]
    leaf(up.b[i], down.a[i], probe[number.of.channels + (i - first.leaf)])

  par i = [root for number.of.channels]
    monitor(up.a[i], down.a[i], up.b[i], down.b[i], probe[i])

  multiplex(probe, all.probes)

  display(all.probes, terminal.screen)
```

```
def dead = 0, alive = not dead :          -- possible states of each cell


def radius    = 1 ,                       -- radius of the 'sphere of influence'
  diameter    = (2 × radius) + 1 ,
  neighbours = (diameter × diameter) - 1 :    -- consequent number of neighbours of



proc calculate.next.state(chan link[], value in[], state, var next.state) =
  var count :          -- number of living neighbours
  seq
    var state.of.neighbour[neighbours] :
    seq
      par i = [0 for neighbours]          -- receive present state from each neighbour
        link[in[i]] ? state.of.neighbour[i]
      count := 0
      seq i = [0 for neighbours]
        if
          state.of.neighbour[i] = alive
            count := count + 1            -- and count the number alive this generation
          state.of.neighbour[i] = dead
            skip
      if
        count < 2                         -- if too few
          next.state := dead              --    this cell dies from isolation
        count = 2                         -- if exactly two
          next.state := state             --    this cell is stable
        count = 3                         -- if exactly three
          next.state := alive             --    this cell is gives birth if dead
        count > 3                         -- if too many
          next.state := dead  :           --    this cell dies from overcrowding



proc broadcast.present.state(chan link[], value out[], state) =
  -- satisfy each neighbour's requirement to know this cell's state
  par i = [0 for neighbours]
    link[out[i]] ! state                  :
```

```
def set.state = 1, ask.state = 2, terminate = 3 :


proc cell(chan link[], value in[], out[], chan control, sense) =
  -- calculate the state of a single cell on the board
  var state, instruction :
  seq
    state := dead                          -- the whole board starts off dead
    control ? instruction
    while instruction ≠ terminate
      seq
        if                                 -- on instruction
          instruction = set.state
            control ? state                --     accept a new state
          instruction = ask.state
            var next.state :
            seq                            --     or calculate the next state
              par
                broadcast.present.state(link, out, state)
                seq
                  calculate.next.state(link, in, state, next.state)
                  sense ! (state ≠ next.state); next.state
                                           --     announce this to the controller
              state := next.state          --     and move on a generation
        control ? instruction                    :


def array.width = 50, array.height = 20 :
def number.of.cells  = array.height × array.width ,
    number.of.links = neighbours × number.of.cells :

proc initialize(value x, y, var in[], out[]) =
  -- initialize the link indirection arrays for the cell at x,y
  seq deltax = [-radius for diameter]          -- offset of neighbour
    seq deltay = [-radius for diameter]        -- in two dimensions
      var direction :
      seq
        direction := deltax + (diameter × deltay)      -- -4 ≤ direction ≤ +4
        if
          direction ≠ 0
            var index, process :
            seq
              -- select outgoing channel in this direction
              process := x + (array.width × y)
              index   := (neighbours + direction) mod (neighbours + 1)
              out[index] := index + (neighbours × process)
              -- and select the corresponding incoming channel
              process := ((x + deltax + array.width) mod array.width) +
                           (array.width × ((y + deltay + array.height) mod array.height))
              index   := (neighbours - direction) mod (neighbours + 1)
              in[index] := index + (neighbours × process)

          direction = 0        -- this cell is not its own neighbour
            skip                                   :
```

```
def control = not ((not 0) << 5), escape = control \/ '[' :


proc move.cursor(chan screen, value x, y) =
  -- move to column x of line y (of a VT52)
  screen ! escape; 'Y'; '*s' + y; '*s' + x :

proc initialize.display(chan screen) =
  -- clear the screen (of a VT52)
  screen ! escape; 'H' ; escape ; 'J' :

proc clean.up.display(chan screen) =
  -- move away from board
  move.cursor(screen, 0, array.height) :

proc display.state(chan screen, value x, y, state) =
  -- display the state of one cell
  seq
    move.cursor(screen, x, y)
    if
      state = alive              -- live cells show as an asterisk
        screen ! '**'
      state = dead               -- dead ones as a blank space
        screen ! '*s'                    :



proc generation(chan screen, control[], sense[], var active) =
  -- cause the colony on the board to move on one generation
  seq
    seq cell = [0 for number.of.cells]     -- invite each cell
      control[cell] ! ask.state            --     to make evolutionary progress
    active := false
    seq cell = [0 for number.of.cells]     -- for each cell on the board
      var changed, next.state :
      seq
        sense[cell] ? changed; next.state  --     receive its new state
        if
          changed                          --     and cause it to be displayed
            seq
              display.state(screen, cell mod array.width, cell / array.width, next.state)
              active := true
          not changed
            skip                                 :
```

```
proc edit(chan keyboard, screen, control[]) =
  -- modify the colony on the board

  def ctrl = not ((not 0) << 5), otherwise = true :
  def left.key = ctrl /\ 'H',      right.key = ctrl /\ 'L',
      up.key   = ctrl /\ 'K',      down.key  = ctrl /\ 'J',
      uproot.key = '*s',           plant.key = '**' :

  var x, y, editing, ch :
  seq
    x := array.width / 2           -- set co-ordinates of cursor to centre of board
    y := array.height / 2
    editing := true
    while editing
      seq
        move.cursor(screen, x, y)
        keyboard ? ch
        if
          (ch = left.key) and (x > 0)
            x := x - 1
          (ch = right.key) and (x < (array.width - 1))
            x := x + 1
          (ch = up.key) and (y > 0)
            y := y - 1
          (ch = down.key) and (y < (array.height - 1))
            y := y + 1
          (ch = uproot.key) or (ch = plant.key)
            var state :            -- change state of the cell under the cursor
            seq
              state := (dead /\ (ch = uproot.key)) \/ (alive /\ (ch = plant.key))
              control[x + (array.width * y)] ! set.state; state
              display.state(screen, x, y, state)   -- keeping the display in step
          (ch = 'q') or (ch = 'Q')
            editing := false
          otherwise                -- ignoring anything that is not understood
            skip                                  :


def idle = 1, editing = 2, single.stepping = 3, free.running = 4, terminated = 5 :

proc display.activity(chan screen, value activity) =      -- display state of the controller
  seq
    move.cursor(screen, array.width + 1, array.height / 2) -- to the right of the board
    if
      activity = idle
        write.string(screen, "Idle")
      activity = editing
        write.string(screen, "Edit")
      activity = single.stepping
        write.string(screen, "Step")
      activity = free.running
        write.string(screen, "Busy")
      activity = terminated
        write.string(screen, "Done")                  :
```

```
proc controller(chan keyboard, screen, control[], sense[]) =
  -- control the activity of the colony on the board under direction from the keyboard
  var activity :
  seq
    activity := idle
    initialize.display(screen)
    while activity ≠ terminated
      seq
        display.activity(screen, activity)
        var ch :
        pri alt
          (activity ≠ editing) & keyboard ? ch  -- provided not editing, typing ...
            if
              (ch = 'q') or (ch = 'Q')          -- ... Q stops the program
                activity := terminated
              (ch = 's') or (ch = 'S')          -- ... S stops the evolutionary process
                activity := idle
              (ch = 'e') or (ch = 'E')          -- ... E invokes the editor
                activity := editing
              (ch = 'r') or (ch = 'R')          -- ... R sets evolution in train
                activity := free.running
              otherwise                         -- ... anything else causes evolution
                activity := single.stepping     --     for just one (more) generation
          (activity = editing) & skip
            seq
              edit(keyboard, screen, control)
              activity := idle
          (activity = free.running) or
          (activity = single.stepping) & skip   -- if evolving but nothing typed
            var changing :
            seq
              generation(screen, control, sense, changing)      -- move on a generation
              if
                (activity = single.stepping) or (not changing)
                  activity := idle
                (activity = free.running) and changing
                  skip
    display.activity(screen, activity)
    seq cell = [0 for number.of.cells]
      control[cell] ! terminate
    clean.up.display(screen)                    :



chan link[number.of.links], control[number.of.cells], sense[number.of.cells] :
par
  controller(keyboard, screen, control, sense)  -- control process

  par x = [0 for array.width]                   -- board
    par y = [0 for array.height]
      var in[neighbours], out[neighbours] :
      seq
        initialize(x, y, in, out)
        cell(link, in, out, control[x + (array.width × y)], sense[x + (array.width × y)])
```

```
def bits.in.character      = 8,
    number.of.characters = 1 << bits.in.character,
    number.of.codes      = number.of.characters,
    character.mask       = not ((not 0) << bits.in.character) :


def root = 0,  size.of.tree = (2 * number.of.codes) - 1 :


var children[size.of.tree],   parent[size.of.tree],
    character[size.of.tree], representative[number.of.characters] :


proc insert.new.node( var new.node, value weight.of.new.node,
                      var left.limit, value right.limit        ) =
  var weight.limit :
  seq
    if
      if node = [left.limit for right.limit - left.limit]
        weight[node] < weight.of.new.node
          weight.limit := node
      true
        weight.limit := right.limit
    seq node = [left.limit for weight.limit - left.limit]
      seq
        character[node - 1] := character[node]
        children[node - 1]  := children[node]
        weight[node - 1]    := weight[node]
      left.limit := left.limit - 1
      new.node := weight.limit - 1
      weight[new.node] := weight.of.new.node              :
```

```
proc construct.tree(value probability[]) =
  var left.limit, right.limit, weight[size.of.tree] :

  proc construct.leaves =
    -- build the leaves of the tree
    def minimum.character = - (number.of.characters ÷ 2) :
    seq ch = [minimum.character for number.of.characters]
      var new.node :
      seq
        insert.new.node(new.node, probability[ch ∧ character.mask], left.limit, right.limit)
        children[new.node]   := root
        character[new.node] := ch                              :


  proc construct.other.nodes =
    -- join pairs of subtrees until only one tree remains
    while (right.limit - left.limit) ≠ 1
      var new.node :
        seq
          right.limit := right.limit - 2
          insert.new.node(new.node, weight[right.limit] + weight[right.limit+1],
                                                                    left.limit, right.limit)
          children[new.node] := right                  :


  proc invert.representation =
    -- set parent[] and representative[]
    seq node = [root for size.of.tree]
      if
        children[node] = root
          representative[character[node] ∧ character.mask] := node
        children[node] ≠ root
          seq child = [children[node] for 2]
            parent[child] := node                                :



seq
  left.limit   := size.of.tree + 1
  right.limit := size.of.tree + 1

-- left.limit = (size.of.tree + 1) and  (right.limit - left.limit) = 0

  construct.leaves

-- left.limit = number.of.code  and  (right.limit - left.limit) = number.of.codes

  construct.other.nodes

-- left.limit = root            and  (right.limit - left.limit) = 1

  invert.representation                    :
```

```
proc encode.character(chan output, value ch) =
  -- Transmit the encoding of ch along output
  def size.of.encoding = number.of.codes - 1 :
  var encoding[size.of.encoding], length, node :
  seq
    length := 0
    node   := representative[ch ∧ character.mask]
    while node ≠ root
      seq
        encoding[length] := node - children[parent[node]]
        length := length + 1
        node   := parent[node]
    seq i = [1 for length]
      output ! encoding[length - i]                :
```

```
proc decode.character(chan input, var ch) =
  var node :
  seq
    node := root
    while children[node] ≠ root
      var bit :
      seq
        input ? bit
        node := children[node] + bit
    ch := character[node]                :
```

```
def probability = table[ ... ] :  -- indexed by [0 for number.of.characters]


def end.of.message = -1 :


proc copy.encoding(chan source, end.of.source, sink) =
  -- Read characters from source, sending their encodings along
  -- sink, until a signal is received along end.of.source.
  var more.characters.expected :
  seq
    construct.tree(probability)
    more.characters.expected := true
    while more.characters.expected
      var ch :
      alt
        source ? ch
          encode.character(sink, ch)
        end.of.source ? any
          more.characters.expected := false
    encode.character(sink, end.of.message)            :


proc copy.decoding(chan source, sink) =
  -- Read a bit stream from source, decoding it into characters
  -- and send these along sink until end.of.message is decoded
  var more.characters.expected :
  seq
    construct.tree(probability)
    more.characters.expected := true
    while more.characters.expected
      var ch :
      seq
        decode.character(source, ch)
        if
          ch ≠ end.of.message
            sink ! ch
          ch = end.of.message
            more.characters.expected := false    :
```

```
def bits.in.character      = 8,
    number.of.characters = 1 << bits.in.character,
    number.of.codes      = number.of.characters + 1,
    character.mask       = not ((not 0) << bits.in.character) :


def root = 0  size.of.tree = (2 * number.of.codes) - 1,  not.a.node = size.of.tree, :


var escape, weight[size.of.tree],
    children[size.of.tree],   parent[size.of.tree],
    character[size.of.tree], representative[number.of.characters] :



proc construct.tree =
  -- Create a tree for the encoding in which every character is escaped
  seq
    escape := root
    weight[escape]   := 1
    children[escape] := root     -- it is a leaf
    seq ch = [0 for number.of.characters]
      representative[ch] := not.a.node              :


proc create.leaf(var new.leaf, value ch) =
  -- Extend the tree by fision of the escape leaf into two new leaves
  var new.escape :
  seq
    new.leaf             := escape + 1
    new.escape           := escape + 2

    children[escape]     := new.leaf      -- escape is the new parent

    weight[new.leaf]     := 0
    children[new.leaf]   := root
    parent[new.leaf]     := escape
    character[new.leaf]  := ch
    representative[ch /\ character.mask] := new.leaf

    weight[new.esape]    := 1
    children[new.escape] := root
    parent[new.escape]   := escape

    escape               := new.escape            :
```

```
proc swap.trees(value i, j) =
  -- Exchange disjoint sub-trees rooted at i and j

  proc swap.words(var p, q) =
    -- Exchange values stored in p and q
    var t :
    seq
      t := p
      p := q
      q := t                :

  proc adjust.offspring(value i) =
    -- Restore downstream pointers to node i
    if
      children[i] = root
        representative[character[i] ∧ character.mask] := i
      children[i] ≠ root
        seq child = [children[i] for 2]
          parent[child] := i                              :

  seq
    swap.words(children[i], children[j])
    swap.words(character[i], character[j])
    adjust.offspring(i)
    adjust.offspring(j)                                   :


proc increment.frequency(value ch) =
  -- Adjust the weights of all relevant nodes to account for one more occurrence
  -- of the character ch, and adjust the shape of the tree if necessary
  var node :
  seq
    if
      representative[ch ∧ character.mask] ≠ not.a.node
        node := representative[ch ∧ character.mask]
      representative[ch ∧ character.mask] = not.a.node
        create.leaf(node, ch)
    while node ≠ root
      if
        weight[node-1] > weight[node]
          seq
            weight[node] := weight[node] + 1
            node := parent[node]
        weight[node-1] = weight[node]
          if i = [1 for (node - root) - 1]
            weight[(node - 1) - 1] > weight[node]
              seq
                swap.trees(node, node - 1)
                node := node - 1
    weight[root] := weight[root] + 1 :
```

```
proc encode.character(chan output, value ch) =
  -- Transmit the encoding of ch along output
  def size.of.encoding = bits.in.character + (number.of.codes - 1) :
  var encoding[size.of.encoding], length, node :
  seq
    if
      representative[ch ∧ character.mask] ≠ not.a.node
        seq
          length := 0
          node   := representative[ch ∧ character.mask]
      representative[ch ∧ character.mask] ≠ not.a.node
        seq
          seq i = [0 for bits.in.character]
            encoding[i] := (ch >> i) ∧ 1        -- i'th bit of unencoded ch
          length := bits.in.character
          node   := escape
    while node ≠ root
      seq
        encoding[length] := node - children[parent[node]]
        length := length + 1
        node   := parent[node]
    seq i = [1 for length]
      output ! encoding[length - i]                :
```

```
proc decode.character(chan input, var ch) =
  -- Receive an encoding along input and store the corresponding character in ch
  var node :
  seq
    node := root
    while children[node] ≠ root
      var bit :
      seq
        input ? bit
        node := children[node] + bit
    if
      node < escape
        ch := character[node]
      node = escape
        var bit :
        seq
          input ? bit
          ch := - bit
          seq i = [2 for bits.in.character - 1]
            seq
              input ? bit
              ch := (ch << 1) ∨ bit                    :
```

```
def end.of.message = -1 :


proc copy.encoding(chan source, end.of.source, sink) =
  -- Read a stream of characters from source, until signalled on end.of.source,
  -- and transmit their encodings in sequence along sink, followed by that of
  -- end.of.message, maintaining throughout the encoding tree for the encoding
  -- determined by the cumulative frequencies of the characters transmitted
  var more.characters.expected :
  seq
    construct.tree
    more.characters.expected := true
    while more.characters.expected
      var ch :
      alt
        source ? ch
          seq
            encode.character(sink, ch)
            increment.frequency(ch)
        end.of.source ? any
          more.characters.expected := false
    encode.character(sink, end.of.message)            :


proc copy.decoding(chan source, sink) =
  -- Read the encodings of a stream of characters, up to and including the
  -- encoding of end.of.message, from source and transmit the corresponding
  -- characters along sink, maintaining the encoding tree for the encoding
  -- determined by the cumulative frequencies of the characters received
  var more.characters.expected :
  seq
    construct.tree
    more.characters.expected := true
    while more.characters.expected
      var ch :
      seq
        decode.character(source, ch)
        if
          ch ≠ end.of.message
            seq
              sink ! ch
              increment.frequency(ch)
          ch = end.of.message
            more.characters.expected := false :
```