Programming in Icon; Problems and Solutions from the Icon Newsletter*

Ralph E. Griswold

TR 86-2b

January 1, 1986; last revised March 10, 1987

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8401831.

Programming in Icon; Problems and Solutions from the Icon Newsletter

The Icon Newsletter was started in 1978 as a vehicle for distributing information about the Icon programming language. One continuing feature of the Newsletter is a ''programming corner'' that presents programming techniques, problems, and solutions. This report is a compilation of that material. It starts with Icon Newletter #5, since earlier Newsletters contained material related to versions of Icon that are now obsolete.

Most of the material that follows appears as it was originally published, although there are some editorial changes to correct errors, to improve continuity, and to bring earlier program material up to the syntax of Version 5.

1. Icon Newsletter #5

This programming corner comes in the form of puzzles and posed questions, with solutions and answers to appear in the next Newsletter.

1. What is the output produced by each of the following expressions?

```
every write((0 | 0) to 7)
every write(0 to 3,0 to 7,0 to 7)
every write(1 | 2 to 3 | 4 by 1 | 2)
every 1 to 3 do every write(1 to 3)
```

2. For arbitrary procedures f(x,y) and g(x,y), what is the sequence of calls produced by

```
every (f | g)(1 to 3, 4 | 5)
```

3. Given

```
s1 := "aeiou"
s2 := "abecaeioud"
```

what are the outcomes of

```
(find | upto)(s1,s2)
(find | upto)(s2,s1)
(if size(s1) > size(s2) then upto else find)(s1,s2)
```

4. What are the outcomes of the following expressions? (Note any that produce errors.)

```
(x | y) := 3
(x & y) := 3
(1 & x) := 3
(x & 1) := 3
(x + 1) := 3
```

5. Given the procedure

```
procedure drive(x)
  fail
end
```

What is the output produced by

```
drive(write(1 to 7))
drive(write(0 to 7,0 to 7))
```

6. What does execution of the following program do?

```
procedure main()
   f(f := write)
end
```

7. The following procedure is proposed as a generator of ''words'' — strings of consecutive letters — in the lines of the input file. It does not work properly, however. What does it actually do and what is the cause of the problem? Rewrite the procedure to work properly.

```
procedure genword()
   local line
   static letters
   initial letters := &lcase ++ &ucase
   while line := read() do
      line ? while tab(upto(letters))
            do suspend tab(many(letters))
end
```

2. Icon Newsletter #6

2.1 An Idiom

Every programming language has a number of particularly apt idioms. Consider the expression

```
x <- x
```

At first sight, this expression appears to be a curiosity. However, when used in a conjunction expression, it serves as a stack with automatic pushing of the value of x when it is evaluated and automatic popping of the value of x during backtracking. Thus, in

$expr_1$ & (x <- x) & $expr_2$

if $expr_1$ succeeds, the value of x is pushed and $expr_2$ is evaluated. If $expr_2$ fails, the value of x is popped and $expr_1$ is reactivated.

In situations in which several expressions are connected by conjunction to obtain the first-in, last-out sequencing provided by goal-directed evaluation, this reversible-assignment idiom is both concise and (once it is understood) clearly indicates its purpose.

2.2 Solutions to Questions Posed in Newsletter #5

In the programming corner of Newsletter #5, several programming questions were posed. These questions are restated below with their answers.

Problem 1:

Q: What is the output produced by each of the following expressions?

```
(a)    every write((0 | 0) to 7)
(b)    every write(0 to 3,0 to 7,0 to 7)
(c)    every write(1 | 2 to 3 | 4 by 1 | 2)
(d)    every 1 to 3 do every write(1 to 3)
```

A: These expressions illustrate the use of every to force generators through all their results. The left-to-right, last-in first-out order of results is shown by the output below. Ellipses are used to compress long sequences where the output follows an obvious pattern.

```
(a)  0      (b)  000     (c)  1      (d)  1
     1           001          2           2
     2            .           3           3
     3            .           1           1
     4            .           3           2
     5           076          1           3
     6           077          2           1
     7           100          3           2
     0           101          4           3
     1            .           1
     2            .           3
     3            .           2
     4           176          3
     5           177          2
     6           200          2
     7           201          3
                  .           4
                  .           2
                  .           4
                 276
                 277
                 300
                 301
                  .
                  .
                  .
                 376
                 377
```

Problem 2:

Q: For arbitrary procedures `f(x,y)` and `g(x,y)`, what is the sequence of calls produced by

```
every (f | g)(1 to 3, 4 | 5)
```

A:

```
f(1,4)
f(1,5)
f(2,4)
f(2,5)
f(3,4)
f(3,5)
g(1,4)
g(1,5)
g(2,4)
g(2,5)
g(3,4)
g(3,5)
```

Problem 3:

Q: Given

```
s1 := "aeiou"
s2 := "abecaeioud"
```

what are the outcomes of

```
(a)    (find | upto)(s1,s2)
(b)    (find | upto)(s2,s1)
(c)    (if size(s1) > size(s2) then upto else find)(s1,s2)
```

A: The answer to this question illustrates that functions are data objects and that function application involves applying the value of a (function-valued) expression, such as (find | upto). In addition, goal-directed evaluation applies to such expressions themselves. In fact, an expression such as *expr*(*expr*$_1$, ..., *expr*$_n$) involves the mutual goal-directed evaluation of *expr*, *expr*$_1$, ..., *expr*$_n$ in which the value of *expr* is applied to *expr*$_1$, ..., *expr*$_n$. The outcomes for the expressions above are

```
(a)    5
(b)    1
(c)    5
```

Problem 4:

Q: What are the outcomes of the following expressions? (Note any that produce errors.)

```
(a)    (x | y) := 3
(b)    (x & y) := 3
(c)    (1 & x) := 3
(d)    (x & 1) := 3
(e)    (x + 1) := 3
```

A: The term *outcome* is used in the technical sense here. As indicated, the outcomes of the first three expressions are variables, since assignment returns its left operand as a variable.

(a) x (assigned the value 3)
(b) y (assigned the value 3)
(c) x (assigned the value 3)
(d) *error* (variable expected)
(e) *error* (variable expected)

Problem 5:

Q: Given the procedure

```
procedure drive(x)
  fail
end
```

What is the output produced by

```
(a)    drive(write(1 to 7))
(b)    drive(write(0 to 7,0 to 7))
```

A: This problem illustrates the relationship between goal-directed evaluation and the control structure every, which forces generators to produce all their results. The same effect can be produced by a procedure that only fails, hence forcing goal-directed evaluation to produce all the results of its argument.

(a)
```
1
2
3
4
5
6
7
```

(b)
```
00
01
02
.
.
.
06
07
10
11
12
.
.
.
.
.
.
66
67
70
71
72
.
.
.
76
77
```

Problem 6:

Q: What does the execution of the following program do?

```
procedure main()
   f(f := write)
end
```

A: This one is tricky. Since the program has no procedure declaration for f, one might suppose the execution of the program is an error. Recall Problem 3 above, however, noting that function and procedure applications are evaluated the same way. Furthermore variables are not dereferenced until all the arguments are evaluated. This applies to the "zeroth" argument, which is the function or procedure to be applied. Evaluation of the first argument assigns a function value to f (i.e., the value of write). Hence this expression is equivalent to write(write) and produces the output

```
function write
```

(The form of the output is a consequence of "imaging" a non-string value for the purposes of output. This is the same imaging that is used in tracing procedure calls.)

Problem 7:

Q: The following procedure is proposed as a generator of "words" — strings of consecutive letters — in the lines of the input file. It does not work properly, however. What does it actually do and what is the cause of the problem? Rewrite the procedure to work properly.

```
procedure genword()
   local line
   static letters
   initial letters := &lcase ++ &ucase
   while line := read() do
      line ? while tab(upto(letters))
            do suspend tab(many(letters))
end
```

A: The problem lies in the suspend expression. suspend is like every — it forces its argument to generate all its results. Although neither tab nor many have alternative results, tab does restore the value of &pos if it is resumed to produce a second result. Hence, &pos is always restored to its position prior to the first word and this procedure loops, continually returning the first word of the first line of input!

There are two ways of circumventing this problem: use of an auxiliary identifier or explicitly preventing generation of alternatives in the suspend expression, and hence the backtracking done by tab. Thus, the while loop can be rewritten as

```
while tab(upto(letters)) do {
  t := tab(many(letters))
  suspend t
  }
```

or

```
while tab(upto(letters)) do
   suspend tab(many(letters)) \ 1
```

Incidentally, this problem is sufficiently insidious that it deserves attention in the design of Icon. The subtlety of the problem lies in the fact that, except for reversible assignments, Icon does data backtracking only in tab and move.

3. Icon Newsletter #7

Write the shortest possible self-reproducing Icon program; i.e., a program, which when run, writes its own text.

SNOBOL4 buffs may be interested in the shortest known self-reproducing SNOBOL4 program:

```
S = ′ OUTPUT = " S = 0" S "0"; OUTPUT = REPLACE(S,+"","0");END′
OUTPUT = " S = ′" S "′"; OUTPUT = REPLACE(S,+"","′");END
```

4. Icon Newsletter #8

There are relatively few responses to the request for self-reproducing Icon programs, although a number of self-reproducing programs in other languages were offered. Viktors Berstis pointed out that the self-reproducing SNOBOL4 program given in Newsletter #7 could be shortened and made more easily usable by replacing OUTPUT by PUNCH. He also suggested the following program:

```
PUNCH = REWIND(5) INPUT;END
```

This solution is certainly shorter than the one given, although it is more of a ''self-copying'' program than a self-reproducing one, since it relies on facilities of the operating system (including the capacity to rewind the standard input file).

The shortest self-reproducing Icon program was supplied by Steve Wampler (an inside job):

```
procedure main();x:="write(\"procedure main();x:=\",image(x));write(x);end"
write("procedure main();x:=",image(x));write(x);end
```

5. Icon Newsletter #9

The following shuffling procedure was contributed by Ward Cunningham of Tektronix Computer Research Laboratory:

```
procedure shuffle(x)
   every !x :=: ?x
   return x
end
```

Note that this is not the standard algorithm for shuffling as given, for example, by Knuth. Comments on the ''effectiveness'' of the procedure above are welcome.

Observe that x may be a string, list, table, or record. *Question:* why does this procedure not work for csets?

6. Icon Newsletter #10

6.1 Implicit Type Conversion

In Newsletter #9, the following shuffling procedure was given:

```
procedure shuffle(x)
   every !x :=: ?x
   return x
end
```

It was noted there that x may be a string, list, table, or record, but not a cset. The reason is that the operations !x and ?x do not apply to csets directly.

If x is a cset, its value is first converted to a string and then the operation is applied. Consequently,

```
every write(!x)
```

writes the characters in the cset x as expected. However, the implicit type conversion does not change the value of x. The expression above is therefore equivalent to

```
every write(!string(x))
```

Similarly,

```
!x :=: ?x
```

is equivalent to

```
!string(x) :=: ?string(x)
```

This is much like

```
!"abc" :=: ?"abc"
```

Neither argument of the exchange operation is a variable, and a run-time error results.

6.2 Result Sequences

The result sequence for an expression in Icon consists of the results the expression is *capable* of producing. For example, the result sequence for 1 to 5 is {1, 2, 3, 4, 5}. The results that an expression actually produces depend on the context in which the expression is used. For example

```
every write(1 to 5)
```

causes all the results for 1 to 5 to be produced, but in

```
(1 to 5) = 2
```

only the first two results of 1 to 5 are produced.

Result sequences are interesting in themselves, independent of the context in which they are used. This subject is explored in Steve Wampler's doctoral dissertation and in the forthcoming Icon book. For example, the result sequence for

```
expr1 | expr2
```

is simply the result sequence for *expr1* followed by the result sequence for *expr2* (the concatenation of the result sequences).

Similarly, the result sequence for repeated alternation

```
|expr
```

is the repeated concatenation of the result sequences for *expr*.

From this, it follows that the result sequence for

```
(1 to 4) | (7 to 10)
```

is { 1, 2, 3, 4, 7, 8, 9, 10} and the result sequence for

```
|1
```

is { 1, 1, 1, ... }, which is infinite. Similarly, the result sequence for

```
(i := 1) | |(i +:= 1)
```

is { 1, 2, 3, ... }.

An expression that fails has an empty, zero-length result sequence, { }, by definition. Empty result sequences take the place of the Boolean value *false* in control structures such as **while-do** and **if-then-else**. The empty result sequence also terminates the result sequence for repeated alternation. Thus, the result sequence for

```
|read()
```

is the sequence of lines from the input file. This sequence terminates when read() fails at the end of the file.

The use of expressions that have infinite result sequences does not necessarily result in failure of the program to terminate. The generation of results from an expression can be controlled in several ways. The most direct method of controlling generation is the limitation control structure:

```
expr \ i
```

which limits *expr* to at most i results. For example, the result sequence for

```
((i := 1) | |(i +:= 1)) \ j
```

is { 1, 2, 3, ..., *j*}, assuming that the value of j is a positive integer *j*. This provides an easy way of inspecting result sequences; a typical test has the form

```
every write(expr) \ 10
```

These observations on result sequences lead to the following exercises:

1. Write expressions that have the following result sequences (do *not* use procedures):

   (1) The squares of the positive integers: { 1, 4, 9, 16, ... }

   (2) The factorials: { 1, 2, 6, 24, 120, ... }

   (3) The Fibonacci numbers: { 1, 1, 2, 3, 5, 8, 13, ... }

   (4) All nonempty substrings of a string s. For "abc" the result sequence is { "a", "ab", "abc", "bc", "c"}.

   (5) All the odd-sized substrings of s.

2. What are the result sequences for the following expressions? (*Warning*: take appropriate precautions if you try to run these.)

(1) `!&lcase || !&ucase`

(2) `(1 to 3) + (1 to 3)`

(3) `(1 to 3) \ (1 to 3)`

(4) `(1 to 5) = (4 to 9)`

(5) `1 = |0`

7. Icon Newsletter #11

7.1 Solutions to the Exercises in Newsletter #10

The first exercise asked for expressions that produced certain result sequences. There are several ways of doing these, of which one set follows. Some of the parentheses are unnecessary and are included only for clarity.

(1) The squares of the positive integers: `(i := 1) | |((i +:= 1) ^ 2)`

(2) The factorials: `(j := i := 1) | |(j *:= (i +:= 1))`

(3) The Fibonacci numbers: `((i | j) := 1) | (|(i | j) := i + j)`

(4) All nonempty substrings of a string `s`: `s[(i := 1 to *s):((i + 1) to (*s + 1))]`

(5) All the odd-sized substrings of `s`: `s[(i := 1 to *s):((i + 1) to (*s + 1) by 2)]`

The expression for generating the Fibonacci numbers is due to Bill Mitchell.

The second exercise turned the issue around and asked what result sequences were produced by certain expressions. The answers are:

(1) `!&lcase || !&ucase`: {"aA", "aB", "aC", ..., "zX", "zY", "zZ"} (26^2 strings in all)

(2) `(1 to 3) + (1 to 3)`: {2, 3, 4, 3, 4, 5, 4, 5, 6}

(3) `(1 to 3) \ (1 to 3)`: {1, 1, 2, 1, 2, 3}

(Note that the expression that limits a result sequence can, itself, be a generator.)

(4) `(1 to 5) = (4 to 9)`: {4, 5}

(5) `1 = |0`: this is a ‘‘black hole’’! It never produces a result, but its evaluation does not terminate. The reason is that the right argument produces 0, which is not equal to 1. The resulting failure cause the the right argument to be resumed. Since it is a repeated alternation, it produces 0 again, and so on. This phenomenon led to the decision to make repeated alternation terminate if its argument ever has an empty result sequence. Otherwise, for example, `|read()` would turn into a black hole when the end of the input file is reached. Fortunately, expressions such as the one above do not seem to occur in ordinary programming contexts. This black-hole phenomenon should not be disturbing — it is no worse than an expression such as

```
until 1 = 0
```

7.2 Limitation

In a recent letter, John Polstra commented that the limitation control structure prevents reversal of assignment in reversible-assignment expressions. For example, in

```
(x <- y) \ 1
```

the assignment to x is not reversed. This is intentional and not an implementation or design error. There are two factors involved. In the first place, the limitation control structure effectively stands between the expression it limits and the surrounding context. In this role, the limitation control structure simply limits the number of times the expression may be resumed. In the example above, the reversible assignment can be resumed only once (''resumed'' as used here includes the initial evaluation, which assigns the value of y to x). On the other hand, reversal of the assignment occurs when the reversible assignment operation is resumed the second time. Although reversible assignment does not produce a second result, its resumption gives it the opportunity to reverse the assignment. The same thing is true of reversible exchange, tab(i), and move(i).

7.3 N Queens

The non-attacking 8-queens problem is used to the point of boredom in demonstrating backtrack programming. There is a solution in the Icon book. A more difficult problem is producing a program that will solve the *n*-queens problem, where *n* is a parameter. Try this one — it is not trivial.

7.4 A Puzzle

What does the following program do, and why?

```
procedure main()
   write(
      "abcde" ? {
         p() := "x"
         write(&subject)
         &subject
         }
      )
end

procedure p()
   suspend &subject[2:3]
end
```

8. Icon Newsletter #12

8.1 Solutions to the Problems in Newsletter #11

N Queens: Numerous solutions to the 8-queens problem have been presented to show different backtracking techniques. Icon lends itself nicely to such problems as illustrated by the following program*:

```
procedure main()
   write(q(1),q(2),q(3),q(4),q(5),q(6),q(7),q(8))
end
```

*Ralph E. Griswold and Madge T. Griswold, *The Icon Programming Language*, © 1983, page 153. Reprinted by permission of Prentice–Hall, Inc., Englewood Cliffs, New Jersey.

```
procedure q(c)
   suspend place(1 to 8,c)               # look for a row
end

procedure place(r,c)
   static up, down, row
   initial {
      up := list(15,0)
      down := list(15,0)
      row := list(8,0)
      }
   if row[r] = down[r + c - 1] = up[8 + r - c] = 0
   then suspend row[r] <- down[r + c - 1] <-
      up[8 + r - c] <- r                 # place if free
end
```

The heart of this solution lies in the mutual goal-directed evaluation of q(1), q(2), ..., q(8). Each set of values for which these procedure calls mutually succeed corresponds to a solution. Note that all the queens are ''equal''.

Clearly, however, this type of solution does not lend itself to parameterization, since each of the q(i) must appear explicitly in the program.

One approach to the general problem is to use a list to contain the solutions and a hierarchical, recursive scheme in which each queen controls the invocation of the next one. Thus, the first queen dominates the second, and so on. A program of this kind, due to Steve Wampler, is

```
global n, solution

procedure main(x)
   n := x[1]                             # number of queens
   solution := list(n)                   # list of column solutions
   every q(1)                            # start by placing queen in first col.
end

procedure q(c)
   local r
   static up, down, rows
   initial {
      up := list(2 * n - 1,0)
      down := list(2 * n - 1,0)
      rows := list(n,0)
      }
   every 0 = rows[r := 1 to n] = up[n + r - c] = down[ r + c - 1] &
      rows[r] <- up[n + r - c] <- down[r + c - 1] <- 1 do {
         solution[c] := r                # record placement
         if c = n then show()            # all queens placed, show positions
         else q(c + 1)                   # try to place next queen
         }
end

procedure show()
   every writes(!solution)
   write()
end
```

Here the value of *n* is supplied on the command line when the program is run. The version of the program given here is stripped down to conserve space. The complete program with error checking and a display of the queens on a chessboard is included in the Icon program library.

This approach can also be cast in terms of co-expressions, as illustrated by the following program by Steve Wampler:

```
global n, nthq, solution

procedure main(x)
   local i
   n := x[1]
   nthq := list(n + 2)                         # list of queens
   solution := list(n)                         # list of solutions
   nthq[1] := &main                            # 1st queen is main routine
   every i := 1 to n do                        # 2 to n + 1 are real queen placements
      nthq[i + 1] := create q(i)               # one co-expression per column
   nthq[n + 2] := create show()                # n + 2nd queen is display routine
   @nthq[2]                                    # start by placing queen in first col.
end

procedure q(c)
   local r
   static up, down, rows
   initial {
      up := list(2 * n - 1, 0)
      down := list(2 * n - 1, 0)
      rows := list(n, 0)
      }
   repeat {
      every 0 = rows[r := 1 to n] = up[n + r - c] = down[r + c - 1] &
         rows[r] <- up[n + r - c] <- down[r + c - 1] <- 1 do {
            solution[c] := r
         @nthq[c + 2]                          # try to place next queen
         }
      @nthq[c]                                 # tell last queen try again
      }
end

procedure show()
   repeat {
      every writes(!solution)
      write()
      @nthq[n + 1]                             # tell last queen to try again
      }
end
```

A somewhat different form of solution — and one that requires a better understanding of Icon — is illustrated by the following solution to the *n*-rooks problem by Tom Slone.

```
global n

procedure main(x)
   n := x[1]                                   # number of rooks
   every show(rook(n))
end
```

```
procedure show(a)
   every writes(!a)
   write()
end

procedure rook(i)
   static a
   initial a := list(n)
   if i = 0 then return
   if i < n then
      suspend (a[i] := r()) & rook(i - 1)
   else suspend (a[i] := r()) & rook(i - 1) & a
end

procedure r()
   suspend place(1 to n)
end

procedure place(r)
   static col
   initial col := list(n, 0)
   if col[r] = 0 then suspend col[r] <- r
end
```

A Puzzle: In the last Newsletter, the behavior of the following program was posed:

```
procedure main()
   write(
      "abcde" ? {
         p() := "x"
         write(&subject)
         &subject
         }
      )
end

procedure p()
   suspend &subject[2:3]
end
```

In the first place, the argument of write in main is a scanning expression. The scanning expression consists of three sub-expressions. The first is a call to p that returns a substring of &subject. Since this is a substring of a global variable (&subject), it is not dereferenced when p returns, and the subsequent assignment changes the value of &subject to "axcde", which is written. The last sub-expression of the scanning expression is simply &subject; this is the result of the scanning expression and hence the argument of the outer write. Since &subject is a global variable, it is not dereferenced when it is produced by the scanning expression. Hence write gets the *variable* &subject. However, when the scanning expression returns, it restores the former value of &subject (in this case, the empty string, the initial default value of &subject). So, by the time write dereferences &subject, its value is the empty string. Thus, this program writes axcde followed by an blank line!

This example illustrates two things: (1) dereferencing is a serious language design problem, and (2) programmers may encounter some mysterious results if they write programs that rely on the side effects of assignment to global variables like &subject.

Superficially, this program looks like it ought to write axcde twice (actually, the write in the scanning expression got there as a result of trying to find out why the outer write produced a blank line). The problem can be solved by explicitly dereferencing &subject in the scanning expression, as in

```
procedure main()
   write(
      "abcde" ? {
         p() := "x"
         write(&subject)
         .&subject
         }
      )
end
```

Now the value returned by the scanning expression is "axcde" — the value of &subject before its former value is restored at the end of scanning.

8.2 New Problems

Scanning an Entire File: Since Icon has a string data type and many operations on strings, it is natural to process strings as whole objects, rather than character by character as in most lower-level languages. String scanning raises the operations on strings to an even higher level, providing a subject on which scanning functions operate, matching functions that move the position of attention in the subject, and backtracking to the previous position in case a matching function fails.

One annoying problem in trying to process strings as whole objects occurs in situations like lexical analysis, in which the object to be processed is really a file with interspersed line terminators. The result of read in Icon is a string consisting of a line up to a line terminator. Therefore it is natural to process files in terms of their lines. However, a construction to be processed may span several lines (comments are typical). Thus, the natural input to string scanning may not contain the whole string to be processed. Although it is possible to make the entire input into a single string, this is generally inefficient and often impractical.

There is a device that often can be used to overcome these problems. If an expected match fails (for example, searching for the closing token of a comment), the subject of scanning can be reset *during* scanning to be the next line of input. This is illustrated by the following program by Tom Slone for stripping ''white space'' out of Pascal programs:

```
procedure main()
   local line
   while line := read() do
      line ? {
         remwhite()
         write("" ~== tab(0))
         }
end

procedure remwhite()
   while tab(many(' \t')) | (="(*" & comment()) | (pos(0) & newline())
end

procedure comment()
   while not ="*)" do
      if pos(0) then newline()
      else move(1) & tab(many(~'*'))
   return
end
```

```
procedure newline()
   (&subject := read()) | stop("unexpected end-of-file.")
   return
end
```

This technique is applicable only if the part of the subject already scanned can be discarded — once a new value has been assigned to the subject, the old value of the subject is lost and the automatic backtracking in string scanning does not apply to it.

The program above is not complete — it has at least two defects. Correcting these is left as an exercise.

Random Numbers: A linear congruence method is used for generating pseudo-random numbers in Icon. There is only one sequence, which is used for producing random integers, real numbers, and elements of strings and structures. Thus, the operation ?x is valid for values of x of different types. The pseudo-random sequence is computed from the value of &random, which is initially zero and which changes with each use of ?x. &random also can be set, so that a sequence can be repeated or started at an arbitrary place. A problem arises, however, if two or more independent random sequences are needed — the use of one inevitably affects the other — or does it?

For starters, write a procedure random() whose result sequence consists of the successive values of &random as they are produced by successive uses of ?x (note that is it not necessary to know how Icon actually performs the pseudo-random computation). Assume that there is no other use of random generation in the program in which random() is used.

Now remove the assumption, so that the result sequence produced by random() is not affected by uses of ?x elsewhere in the program.

9. Icon Newsletter #13

Random Numbers: In the last Newsletter, the question of independent random sequences was raised. One of the problems posed there was to write a procedure random() whose result sequence consists of the successive values of &random as it changes as a side effect of the evaluation of ?x, assuming there are no other uses of random generation in the program in which random() is used. This is easily done, and such a procedure is

```
procedure random()
   repeat {
      suspend &random
      ?0
      }
end
```

The only point here is the observation that each evaluation of ?0 (or any other valid form of ?*expr*) changes &random.

Now the problem is how to achieve this same sequence of results if there are other occurrences of ?*expr* in the program. Some method of remembering the value of &random is needed. This is not hard to do in a procedure, but it requires a little care. A solution is

```
procedure random()
   local i
   suspend i := 0
   repeat {
      &random :=: i
      ?0
      i :=: &random
      suspend i
      }
end
```

Here, it is necessary to know that the initial value of &random is 0, since there may be random generation before random() is called. The local identifier i keeps track of the value of &random, which may be changed between suspensions and resumptions of random(). The values of &random and i are exchanged so that random() does not affect other uses of ?*expr* — independence works two ways.

So far, so good. But what about an *expression* that generates the successive values of &random without using a procedure? The procedure provides a loop from which values are produced, but there is no corresponding expression form in Icon. For example, there is no way to deliver a value out of a **while** or **every** loop. To do this with an expression requires rephrasing the procedural solution as a mutual evaluation expression, in which backtracking is used to assure that &random has the correct value on each resumption. Such an expression is

```
(i := 0) | (i :=: &random, |?0, i <-> &random)
```

The first expression in the alternation sets i to the initial value of &random which is known to be 0, and also produces this as the first value of the whole expression. The second expression in the alternation is a mutual evaluation of three expressions. The first of these expressions exchanges the values of i and &random, so that i now contains whatever value &random had outside the expression and &random is properly initialized. The second expression changes the value of &random (the reason for repeated alternation will become apparent in a moment). Next, the values of i and &random are exchanged again. Since the exchange operation returns its left argument, the value of the entire mutual evaluation expression is the desired result.

The interesting aspect of this expression occurs when it is resumed to produce another value. The last expression, the reversible exchange, is resumed. This causes the values of i and &random to be restored to what they were before the reversible exchange was evaluated. Specifically, &random is restored to the value it had after the preceding evaluation of |?0, regardless of what happened while the mutual evaluation expression was suspended. Since the reversible exchange does not produce a second result when it is resumed, but only reversed the exchange, |?0 is resumed next. Here the reason for repeated alternation is evident — it always produces another result and as a side effect advances the random sequence. With this new result, the reversible exchange is *evaluated* again to produce the next value of &random, and so on. Note that the first expression in the mutual evaluation is never resumed; it serves only as initialization and the repeated alternation provides a barrier to backtracking, since it always produces another result.

Admittedly, this mutual evaluation expression is somewhat arcane. However, once the concepts are grasped, such techniques become idiomatic.

Getting away from the problem of independently generating the values of &random, which was chosen only to make the problem simple, there are cases in which other independent random sequences are useful. One is a random ''production/consumption'' kind of process. This is illustrated in the following program, which creates randomly positioned stars on a terminal screen, building up an initial field of stars, after which the oldest stars are destroyed in the order in which they were created. Finally, creation ceases and destruction continues until all the stars are gone. To accomplish this, two identical sequences of co-ordinate positions are used, one for creation, and one for destruction. Creation is started up first and allowed to proceed until the destruction process is started. There is no need to provide storage for the co-ordinate positions, since this is done in the expressions as described above. Co-expressions are used so that the creation and destruction of stars can be controlled. The program is:

```
procedure main(x)
   local i, j, r, ran1, ran2
   i := x[1] | 10                        # time for creation/destruction (default 10)
   j := x[2] | 50                        # steady state time (default 50)
   r := 0
   ran1 := create (r := 0, &random :=: r, rplot("*"), &random <-> r)
   ran2 := create (r := 0, &random :=: r, rplot(" "), &random <-> r)
   clear()                               # clear the screen
   every 1 to i do @ran1                 # create the universe
   every 1 to j do {                     # steady state condition
      @ran2
      @ran1
      }
   every 1 to i do @ran2                 # destroy the universe
   home()                                # home the cursor (screen is clear)
end
```

Note that the times for startup/destruction and the steady-state times can be provided optionally as command line arguments. The identifiers r in the co-expressions for ran1 and ran2 are distinct, since the creation of a co-expression creates independent copies of local identifiers.

The procedures rplot(s), clear(), and home() are terminal-dependent. The last two clear the screen and home the cursor, respectively. The interesting routine is rplot(s), which is a generator that on successive resumptions writes s at randomly selected spaces on the screen. For the DataMedia 3025, rplot is:

```
procedure rplot(s)
   static row, col
   initial {
      row := string(&cset[33+:24])
      col := string(&cset[33+:80])
      }
   suspend |writes("\^[Y", col[?80], row[?24], s)
end
```

Question: Can the repeated alternation in

```
   suspend |writes("\^[Y", col[?80], row[?24], s)
```

be placed at any other position other than in front of writes?

10. Icon Newsletter #14

10.1 Answer to a Previous Query

In Newsletter #13, the expression

```
suspend |writes("\^[Y", col[?80], row[?24], s)
```

was used to generate a sequence of displays of s at randomly selected positions on a terminal screen. The question that was posed was whether the repeated evaluation in the expression could be placed at any place other than in front of writes.

In fact, the repeated evaluation can be placed anywhere in the expression as long as both random selection operations are evaluated subsequently to the evaluation of the repeated alternation. One possibility is

```
suspend writes(|"\^[Y", col[?80], row[?24], s)
```

When the argument of the **suspend** expression is resumed to produce another result, the arguments of writes are resumed from right to left. None produce a result until |"\^[Y" is resumed, which produces "\^[Y" again. The arguments to its right are then evaluated again, giving new positions, after which writes is called again to write s at another position. The repeated alternation in this case serves as an endless generator of a constant value for an argument, serving as a kind of barrier for the left-to-right resumption. Other possible locations for the repeated alternation are

```
suspend writes("\^[Y", |col[?80], row[?24], s)
suspend writes("\^[Y", col[|?80], row[?24], s)
suspend writes("\^[Y", col[?|80], row[?24], s)
```

The results are the same in all cases. The last expression is the most efficient. Why?

Note that the following expression does not produce the desired effect:

```
suspend writes("\^[Y", col[?80], | row[?24], s)
```

Although a new row position is produced, the previous column position is used again and s is always written in the same column. The barrier occurs too far to the right.

On the other hand,

```
|suspend writes("\^[Y", col[?80], row[?24], s)
```

does not produce the desired effect at all. In fact, it generates only one display. Why?

10.2 Returning More than One Value from a Procedure

Persons have asked about ways to return multiple values from a procedure.

Icon argument transmission is strictly by-value and there is no way, *per se*, to return multiple values from a procedure. One way around this problem is to return a structure that contains several values as in

```
procedure p()
       .
       .
       .
   return [expr1, expr2, expr3, ... exprn]
end
```

In some cases, a record may be more appropriate than a list.

The values then may be obtained from the structure that is returned, as in

```
a := p()
x := a[1]
y := a[2]
z := a[3]
   .
   .
   .
```

Note that this approach allows a procedure not only to produce multiple values, but also to produce an arbitrary number of values, possibly varying from call to call. In this case, a more general method must be used to access the values that are returned.

Since structures in Icon are pointers to data objects, something akin to call-by-reference can be obtained by passing a structure as an argument to a procedure, as in

```
a := list(n)
p(a)
```

with

```
procedure p(a)
        .
        .
        .
    a[1] := expr1
    a[2] := expr2
    a[3] := expr3
        .
        .
        .
    a[n] := exprn
    return
end
```

When p returns, the values are in the list a and can be accessed as before.

An entirely different approach to returning multiple values is to generate them in sequence:

```
procedure p()
        .
        .
        .
    suspend expr1 | expr2 | expr3 | ... | exprn
end
```

This method fits naturally into Icon's expression evaluation mechanism. The values can be put into a list, if desired, by

```
    a := []
    every put(a, p())
```

The following expression does the same thing:

```
    put(a := [], p())
```

Note that the number of values that p produces need not be known. On the other hand, there is a problem if the generated values are to be assigned to separate variables. This can be done by making explicit assignments as above, using

```
    x := a[1]
    y := a[2]
    z := a[3]
        .
        .
        .
```

but it is tempting to avoid the list and to use iteration, as in

```
    every (x | y | z | ...) := p()
```

This does not work as intended, since evaluation is left-to-right and resumption is right-to-left. Thus, the alternation expression first produces x and p() is called, producing the value of *expr*$_1$, which is assigned to x. However, p() is resumed next, producing the value of *expr*$_2$, which is also assigned to x, and so on. (What values are assigned to y and z?)

The lack of "parallel" evaluation in Icon can be circumvented by using a co-expression:

```
    e := create p()
    every (x | y | z | ...) := @e
```

The resumption of a co-expression activation does not produce another value. Consequently, the first value is assigned to x, the alternation is resumed, y is produced, and the second value produced by p() is assigned to it by the second activation of e, and so on.

While this approach is a bit oblique for this simple situation, it illustrates the control that co-expressions provide over the production of results.

10.3 Initial Assigned Values in Tables

When a table is created, as in the expression

```
t := table(x)
```

the value of x is the initial assigned value for new entries in t. This initial assigned value is used for references to entries that are not already in the table. For example, if a count is being kept of strings, the initial assigned value might be 0, as in

```
count := table(0)
```

Now suppose the following expression is evaluated:

```
count[s] +:= 1
```

This expression produces the same result as

```
count[s] := count[s] + 1
```

Suppose s is not in the table. Then the reference to

```
count[s]
```

in the addition operation produces 0, the initial assigned value. (The augmented assignment operation is preferable to addition and assignment, since the latter expression requires that s be looked up twice in the table.)

But suppose that the initial assigned value is not known. How can it be determined? The problem is that there is not, in general, any way of knowing what entries there are in a table, short of converting it to a list by sorting it.

One approach is to pick some unlikely entry value (perhaps a value that is not a string, assuming all the entries in the table are strings). This may work in practice, but for an arbitrary table with arbitrary entries, what value can be *guaranteed* not to be in the table?

The answer is easy — use an entry value that cannot have *existed* before the test. Any newly created structure will do, but the following expression is particularly simple:

```
x := t[[]]
```

Since a list creation expression creates a new structure, the entry value [] cannot be in t and the expression above assigns the initial entry value of t to x — absolutely guaranteed!

10.4 Matching Expressions

Matching expressions, which are analogous to patterns in SNOBOL4, provide a way to elevate string scanning in Icon to a higher conceptual level. By definition, a matching expression is an expression that may change &pos and always returns the substring of &subject between the new and old values of &pos. A matching expression that fails also must restore &pos to its old value. For example, tab(i) and move(i) are built-in matching expressions, but &pos := i is not a matching expression, since it does not return the substring of &subject between the former and new values of &pos.

Suppose that *expr$_1$* and *expr$_2$* are matching expressions. Then which of the following also are matching expressions?

> *expr$_1$* & *expr$_2$*
> *expr$_1$* | *expr$_2$*
> *expr$_1$* || *expr$_2$*
> x := *expr$_1$*
> if *expr* then *expr$_1$* else *expr$_2$*
> while *expr$_1$*
> every *expr$_1$*

A matching procedure is a procedure whose call is a matching expression. An example is

```
procedure Arb()
   local pos, s
   pos := &pos
   every s := &subject[pos:&pos := &pos to *&subject + 1] do suspend s
   &pos := pos
end
```

which corresponds to the SNOBOL4 pattern ARB. This procedure can be written considerably more compactly; what is the most concise possible form?

The requirements that an expression must meet in order to be a matching expression can be modified to produce a variety of different classes of ''patterns''. One possibility is suggested by John Polstra:

> A useful extension to the idea of the matching procedure is what I call the transforming procedure. A transforming procedure is just like a matching procedure, except that it returns a *variable* which is a substring of &subject. Assignment to a call of a transforming procedure can then be used for modifying the subject, as could assignment to tab(i) and move(i) before Version 5 of Icon. A useful general transforming procedure is the following:
>
> ```
> procedure xform(p)
>    suspend &subject[.&pos:p() & &pos]
> end
> ```
>
> If p is a parameterless matching procedure, then p() is the corresponding matching expression and xform(p) is the corresponding transforming expression. Using co-expressions, a more general version (allowing parameters) can be written.

The remark ''before Version 5'' refers to Version 4 of Icon, which is now obsolete.

10.5 Problems with Dereferencing

The arguments of functions are not dereferenced until all arguments are evaluated. Consequently, the expression

```
write(s,s := "a")
```

writes aa, regardless of the value that s had prior to the evaluation of this expression.

Such expressions generally are considered to be bad style and they rarely occur in programs, at least in such an obvious form. More subtle and perplexing problems may occur because subscripting expressions in Icon are variables and are polymorphic. Consider

```
x[y] := z
```

Here x may be a string, a list, a table, or even a record. Now consider the expressions

```
x := "hello world"
x[3] := (x := "abc")
```

What happens when the value of x changes between the time it is subscripted and the time the assignment to the subscripted variable is made? What about

```
x := "hello world"
x[3] := (x := "ab")
```

Here the subscript of x is in range when it appeared on the left side of the assignment but is out of range when the assignment is made? What about

```
x := "hello world"
x[3] := (x := [1,2,3,4])
```

where the type of the value of x is changed? Or even

```
x := "hello world"
x[3] := (x := 397)
```

where the type is changed, but to one that is coercible to the previous type?

Granted that such expressions are unlikely to occur in ''real'' programs, they must be accounted for by the semantics of Icon and implementations must handle them properly.

10.6 Syntactic Pitfalls

The Icon translator automatically inserts semicolons between expressions on adjacent lines in cases where the token at the end of the first line is legitimate as the end of an expression (an ''ender'') and the token at the beginning of the second line is legitimate as the beginning of an expression (a ''beginner''). Thus, semicolon insertion makes it possible to write programs without having to put semicolons at the ends of expressions.

All prefix operators are beginners. Since many prefix operators also are legal in infix operators, semicolon insertion sometimes can produce unexpected results. For example,

```
x
| y
```

is translated as if it had been written

```
x; | y
```

not as

```
x | y
```

(Identifiers are both beginners and enders.) Note that

```
x; | y
```

is syntactically correct, if a bit unlikely. It is advisable to guard against unexpected translations of infix operations by putting the infix operator at the end of the first line, not at the beginning of the second. Thus,

```
x |
y
```

is translated as

```
x | y
```

since no operator is an ender and hence a semicolon is not inserted.

The semicolon insertion mechanism does not take into account situations in which a line ends with an ender and the next line begins with a beginner, but in which the lines do not form complete expressions. Thus,

```
write(i
+ j)
```

is translated as if it were

```
write(i;
+j)
```

and is diagnosed as a syntactic error, although

```
i
+ j
```

is translated as if it were

```
i;
+ j
```

which is syntactically correct.

In the case of expressions with optional arguments, semicolon insertion may produce mysterious effects if care is not taken. For example,

```
return
   x
```

is translated as if it were

```
return;
x
```

This occurs because the argument of **return** is optional, making it an ender. When this code segment is evaluated, the null value is returned and x is never evaluated. If the problem is not recognized, it appears that x always has the null value, even though it may obviously have a different value.

As another example, consider the following code segment that was produced by a SNOBOL4 programmer who is used to having an omitted right argument of assignment default to the empty string:

```
s[1] :=
return s
```

On the face of it, this is erroneous in Icon. However, since := is not an ender, a semicolon is not inserted and this code segment is translated as if it were

```
s[1] := return s
```

Although this is a strange expression, it is both syntactically and semantically correct. When the right argument of the assignment is evaluated, a return occurs and the assignment is never completed. Unless the translator's interpretation of this expression is recognized, the effect during program execution may appear mysterious.

Fortunately, semicolon insertion works well in practice. Observation of the rules of program layout given above avoids all these problems.

10.7 Trivia Corner

What is the longest string of distinct prefix operators which, when applied to a value, might compute a meaningful result? (You may assume any value that you wish.) What if the prefix operators need not be distinct?

11. Icon Newsletter #15

11.1 Assignment to Subscripted Strings

In the last Newsletter, the semantics of expressions such as

x[i:j] := (x := *expr*)

were posed, where x is string-valued when the subscripting expression is evaluated, but in which *expr* changes the value of x before the (left) assignment is made to replace the subscripted string. (Expressions such as x[i] and x[i+:j] are just special cases of x[i:j].)

While expressions like this are uncommon (and generally are considered to be in poor style), they are legal and therefore must be well defined and handled properly in the implementation. (It should be no surprise that all the possibilities were not considered in the initial design and that there were several bugs related to these matters in the early versions of the implementation.)

This is a case where efficiency and implementation considerations influenced language design. The problem is that the translator cannot, *in general*, determine whether an expression such as x[i:j] will have a value assigned to it. Even if x[i:j] is the target of an assignment operation, the assignment never may be made because of failure in evaluation elsewhere in the expression. In the case of

```
return x[i:j]
```

the translator has even less information, since the use of the returned expression depends on the context in which the function containing this return is called. For these reasons, the translator treats all expressions such as x[i:j] in the same way*. When an expression like x[i:j] is evaluated, if the value of x[i:j] is a string, a *trapped variable* is

*There is the potential here for an implementation optimization, since there are many situations in which the translator could determine that a subscripting expression is not the target of an assignment.

produced. A trapped variable is a special kind of variable that points to a small block of data which contains enough information to assign a new value to x if an assignment is made to x[i:j]. This information consists of the variable x and the location of the substring in x. Every string subscripting expression produces a trapped variable. Although the block of data that is created usually is used only transiently, it causes a certain amount of storage throughput.

Now consider what happens if the value of x is changed before an assignment is made to x[i:j]. Since the value of x can be changed to anything, the assignment cannot be made blindly — the position of the replaced string might be out of range, even if the new value of x is a string. Consequently, the type of x is checked when assignment is about to be made to x[i:j]. If the value of x is a string, its length is checked to be sure the substring specified by i:j is still in range. If it is, the assignment is made, even if the value of x is different from what it was when x[i:j] was evaluated. Thus, in

```
x := "hello world"
x[3] := (x := "abc")
```

the value of x becomes "ababc". On the other hand, if the value of x is a string, but it is too short, run-time error 205 (value out of range) occurs, as in

```
x := "hello world"
x[3] := (x := "ab")
```

One might well argue that assignment to x[i:j] should be an error if the value of x has changed, even if the substring is still in range. After all, such a situation seems more likely to be an error than an intentional computation. Here, however, there is an efficiency consideration. In order to be able to detect that the value of x has changed, it would be necessary to save the value of x as well as the variable x in the trapped variable. Furthermore, this would have to be done for every evaluation of a string subscripting expression. The result would be substantially higher storage throughput just to treat a pathological case more elegantly.

From a language design viewpoint, a somewhat more radical alternative would be to bind the value of x to x at the time x[i:j] is evaluated, so that

```
x := "hello world"
x[3] := (x := "abc")
```

would change the value of x to "heabclo world". This solution also would require saving the value of x in the trapped variable — additional overhead that again does not seem justified for such a pathological situation.

Returning to the situation as it actually is handled, given that any string value for x that is long enough is acceptable, the next question is what to do if the value of x is not a string when the assignment is made to x[i:j]? In consonance with Icon's general philosophy of converting types automatically whenever possible, if the value of x can be converted to a string, it is. Thus,

```
x := "hello world"
x[3] := (x := 397)
```

changes the value of x to "39397". Weird, maybe, but consistent with the result of concatenating two integers — which is, after all, what this expression amounts to.

If the value of x cannot be converted to a string, a run-time error (103) occurs, as in

```
x := "hello world"
x[3] := (x := [1,2,3,4])
```

Note that these problems are essentially problems of dereferencing — when and how the value of x is determined when assignment is made to x[i:j]. There are a number of other situations in Icon in which dereferencing is a problem. One is string scanning, which will be discussed in the next Newsletter, along with more material on matching expressions.

11.2 Trivia Corner

In the last Newsletter, the following problem was posed:

> What is the longest string of distinct prefix operators which, when applied to a value, might compute a meaningful result? (You may assume any value that you wish.) What if the prefix operators need not be distinct?

For distinct prefix operators, one possibility is

```
|+=-?*~\@^!x
```

It might go like this: Let x be a list of co-expressions. Generate one, refresh and activate it, being sure the result is nonnull. Assuming the result is a cset, use the size of its complement to provide a range for a randomly selected integer. Negate this integer. Match the equivalent string in &subject and convert the result back to an integer. Repeat the whole process (whatever that means). *Enough!*

Strictly speaking, repeated alternation is a control structure, not an operator, but it is denoted with operator syntax. Note that the prefix operators . and / are not included in the expression above. They can be added, but not in a "meaningful" way.

If the prefix operators do not have to be distinct, there is no specific limit on the number that can occur. Consider, for example,

```
=== ... ==x
```

The expression =x matches x in &subject, ==x matches two consecutive occurrences of x, and so on.

What about expressions such as

```
??? ... ??x
```

11.3 Pitfalls

Steve Wampler contributes the following program, in which the procedure tally echos its argument and tallies it in the table count. In the main procedure, empty input lines are converted into the more prominent marker <empty line> . Or are they? What does this program actually do? What does it take to fix the problem?

```
global count

procedure main()
   count := table(0)
   while line := read() do
      tally(("" ~== line) | "<empty line>")
                    :
end

procedure tally(s)
   count[s] +:= 1
   write(s)
end
```

12. Icon Newsletter #16

12.1 Old Business

Consider the program at the end of the last Newsletter. If line is not empty, tally(line) is called and line is written. However, since tally has no explicit return, it fails by flowing off the end of the procedure body. Since the call fails, the argument

```
("" ~=== line) | "<empty line>")
```

is resumed, resulting in the call tally("<empty line>"). Consequently, <empty line> is written after every nonempty line, as well as in place of empty lines.

The cure is simple — insert a return at the end of the procedure body for tally. (What is another way of fixing the problem?) The lesson is a more general one — be careful to provide a return at the end of a procedure body unless calls of the procedure are supposed to fail.

12.2 Chosing Programming Techniques in Icon

There often are several ways of doing the same thing in Icon. While this is true of most programming languages, it is exaggerated in Icon, since its expression evaluation mechanism is more general than the expression evaluation mechanisms of "Algol-like" languages, such as Pascal and C. Thus, in Icon, there is often an Algol-like solution and also a solution that makes use of generators. The fact that Icon has both low-level string operations and higher-level string scanning complicates the situation.

The novice Icon programmer (and even the more advanced one) is faced with choices that may be confusing or even bewildering. Most programmers develop a fixed set of techniques and often fail to use the full potential of the language.

In the discussion that follows, a simple text processing problem is approached from a variety of ways to illustrate and compare different programming techniques in Icon. The problem is to count the number of times the string s occurs in the file f, which is formulated in terms of a procedure scount(s, f).

The first attempt at such a procedure might be

```
# scount1

procedure scount(s,f)
   count := 0
   while line := read(f) do
      while i := find(s,line) do {
         count +:= 1
         line := line[i + 1:0]
         }
   return count
end
```

This solution is very Algol-like in nature and explicitly examines successive portions of each input line. It does not take advantage of the third argument of find, which allows the starting position for the examination to be specified. Using this feature, the procedure becomes

```
# scount2

procedure scount(s,f)
   count := 0
   while line := read(f) do {
      i := 1
      while i := find(s,line,i) + 1 do
          count +:= 1
      }
   return count
end
```

While this solution is shorter than the previous one, it is still Algol-like and uses only low-level string processing. Using string scanning, an alternative solution is

```
# scount3

procedure scount(s,f)
   count := 0
   while line := read(f) do
      line ? while tab(find(s) + 1) do
         count +:= 1
   return count
end
```

This approach eliminates the auxiliary identifier i and uses scanning to move through the string. None of the solutions above uses the capacity of find to *generate* the positions of successive instances of s, however. If this capacity is used, it is not necessary to tab through the string, and the following solution will do:

```
# scount4

procedure scount(s,f)
   count := 0
   while line := read(f) do
      line ? every find(s) do
         count +:= 1
   return count
end
```

At this point, it becomes clear that string scanning provides no advantage and it can be eliminated in favor of the following solution:

```
# scount5

procedure scount(s,f)
   count := 0
   while line := read(f) do
      every find(s,line) do
         count +:= 1
   return count
end
```

Finally, the hard-core Icon programmer may want to get rid of the while loop, making use of the fact that the expression !f generates the input lines from f. The solution then becomes

```
# scount6

procedure scount(s,f)
   count := 0
   every find(s,!f) do
      count +:= 1
   return count
end
```

The relative merits of these different solutions are arguable on stylistic grounds. Certainly the last (scount6) is the most concise, but scount5 is probably easier to understand.

But what about efficiency. Is scount6 more efficient than scount5? In fact, how much difference in performance is there among all the solutions?

The relative efficiency varies considerably, depending on the data — how many lines there are in f, how long the lines are, how many times s occurs in f, and so forth. The following figures are typical, however. The figures are normalized so that the fastest solution has the value 1.

scount1	2.96
scount2	2.14
scount3	2.03
scount4	1.14
scount5	1.04
scount6	1.00

The fact that the last solution is the fastest may not be surprising — it is the shortest and uses the features of Icon that are most effective in internalizing computation. Nor should it be surprising that scount2 is significantly faster than scount1, since scount2 avoids the formation of substrings. (It is worth noting, however, that substring formation is relatively efficient in Icon — no new strings are constructed, only pointers to portions of old ones.)

It might be surprising, however, to discover that the use of string scanning in scount3 provides a significant advantage over scount2. Evidently, the internalization of the string and position that scanning provides more than overcomes the fact that scount3 produces substrings (by tab).

The real gain in efficiency comes with the use of generators in scount4, where the state of the computation is maintained in find for all the positions in any one line.

Getting rid of string scanning, which serves no useful purpose in scount5, produces an expected improvement, although perhaps not as much as might be expected. The last step of reducing the nested loops to a single loop in scount6 also produces a slight improvement in performance.

What might be learned from these examples is that there may be a very substantial difference in performance in Icon, depending on the technique used — a factor of nearly 3 between the naive solution of scount1 and the sophisticated one of scount6. The value of using the capabilities of generators is also evident, both in performance and in the conciseness of the solutions. It is notable that string scanning is not as expensive as one might imagine. It can be used without the fear that it will degrade performance substantially.

12.3 Different Ways of Looking at Things

Steve Wampler contributes the following interesting note on programming in Icon:

How do you test to see if the value of i is not between 1 and the length of the string s? I would write:

```
if not (1 <= i <= *s) then ...
```

but my students wrote:

```
if *s < i < 1 then ...
```

13. Icon Newsletter #17

13.1 Old Business

A number of readers objected that the following two expressions from the last Newsletter are not equivalent:

```
if not (1 <= i <= *s) then ...

if *s < i < 1 then ...
```

Actually, no claim of equivalence was made. As an exercise, determine in what situations the two expressions *are* equivalent.

13.2 Anagramming

There is an easy way in Icon, given a string s, to produce another string that contains the characters of s in alphabetical order with duplicate characters removed:

```
string(cset(s))
```

Problem: Write a procedure anagram(s) that produces a string consisting of the characters in s in alphabetical order, but without the deletion of duplicates. For example,

```
anagram("hello")
```

should produce ehllo.

Experiment with different techniques to try to find the fastest method.

14. Icon Newsletter #18

There were several interesting solutions to the anagramming problem posed in the last Newsletter. The most efficient solution, at least when working on a large amount of data, follows:

```
procedure anagram(s)
   local c, s1
   s1 := ""                        # start with the empty string
   every c := !cset(s) do          # for every character in s
      every find(c,s) do           # and every time it occurs in s
         s1 ||:= c                 # append one
   return s1
end
```

The heart of this solution is to use cset(s) to obtain an ordered set of the characters in s. It is interesting that it is noticeably more efficient to use find instead of upto in the solution. The difference does not lie primarily in the functions themselves. What else is at work here?

It is also faster to append the characters one at a time than to count the number of each and append them in groups, as in

```
procedure anagram(s)
   local c, i, s1
   s1 := ""                           # start with the empty string
   every c := !cset(s) do {           # for every character in s
      i := 0
      every(find(c, s)) do            # count the number of times it occurs
         i +:= 1
      s1 ||:= repl(c, i)              # and append that many copies to the result
      }
   return s1
end
```

Why should this be?

Randal Schwartz submitted a number of interesting solutions in addition to one similar to the first solution above. One of his solutions uses a table of characters and their counts:

```
procedure anagram(s)
   local c, t
   c := table(0)
   s ?:= {
      while c[move(1)] +:= 1
      ""
      }
   every t := !sort(c, 1) do
      s ||:= repl(t[1], t[2])
   return s
end
```

This solution is about twice as slow, when working on a large amount of data, as the first one above, probably because of time spent in table lookup and garbage collection (due to the larger amount of storage needed for tables).

An even more interesting, albeit admittedly inefficient, solution from Randal Schwartz is the following one that puts each character of s in a separate table element:

```
procedure anagram(s)
   local t
   t := table()
   s ?:= {
      while t[[]] := move(1);
      ""
      }
   every s ||:= (!sort(t, 2))[2]
   return s
end
```

Do you see why there is a separate table element for each character?

15. Icon Newsletter #20

15.1 String Scanning

The string scanning expression in Icon,

$expr_1$? $expr_2$

is often regarded by programmers as somewhat of a mystery. This expression is technically a control structure, since it cannot be cast as a procedure call. The reason for this is that actions must be taken after *$expr_1$* is evaluated but before *$expr_2$* is evaluated, while in a procedure call, all arguments are evaluated before the procedure gains control.

The actions taken between the evaluation of $expr_1$ and $expr_2$ relate to the global variables &subject and &pos. These "state variables" for scanning are set to the string value of $expr_1$ and 1, respectively. If these variables were not set before $expr_2$ was evaluated, $expr_2$ could not "operate on" the value of $expr_1$.

As mentioned above, &subject and &pos are global variables and their values are accessible throughout the program. They are not affected by procedure calls. If they were, it would not be possible to write "matching procedures" to extend the built-in repertoire of matching functions (tab and move).

A string scanning expression, however, cannot just set the values of these variables. In order for nested scanning and scanning in mutual evaluation to work properly, the current values of the state variables must be saved before new values are set and then restored when the scanning expression is complete. Thus, the scope of the scanning variables can be thought of as being dynamic with respect to scanning expressions.

Although the string scanning expression cannot be cast as a procedure call, it can be cast as nested procedure calls:

$expr_1$? $expr_2$ $\rightarrow$ Escan(Bscan($expr_1$), $expr_2$)

In the nested call, $expr_1$ is evaluated first. Bscan is then called before $expr_2$ is evaluated. Thus, Bscan can manipulate the state variables before $expr_2$ is evaluated, first saving the current "outer" values, then setting the new "inner" values for $expr_2$ to operate on. If $expr_2$ produces a result, Escan is called. It restores the outer values before producing the result provided by $expr_2$. On the other hand, if $expr_2$ fails, Bscan is resumed and can restore the outer values before it, too, fails.

Note that Escan must have access to the outer values saved by Bscan. This is accomplished by passing these values as the result produced by Bscan. Since there are two values, a record can be used. Procedures to model the string scanning expression in this fashion are:

```
record ScanEnvir(subject, pos)

procedure Bscan(e1)
   local OuterEnvir
   OuterEnvir := ScanEnvir(&subject, &pos)
   &subject := e1
   &pos := 1
   suspend OuterEnvir
   &subject := OuterEnvir.subject
   &pos := OuterEnvir.pos
   fail
end

procedure Escan(OuterEnvir, e2)
   local InnerEnvir
   InnerEnvir := ScanEnvir(&subject, &pos)
   &subject := OuterEnvir.subject
   &pos := OuterEnvir.pos
   suspend e2
   &subject := InnerEnvir.subject
   &pos := InnerEnvir.pos
   fail
end
```

Note that if $expr_2$ produces a result, Escan suspends. If it is resumed, Escan restores the inner values and fails, forcing $expr_2$ to be resumed. This allows $expr_2$ to produce a sequence of results.

This is all there is to the string scanning expression — the maintenance of state variables. All of string analysis and "pattern matching" comes from matching functions in $expr_2$ that examine &subject and change &pos. These functions are simple. For example, tab(i) can be modeled as a procedure as follows:

```
procedure tab(i)
   suspend .&subject[.&pos:&pos <- i]
end
```

This leaves the question of where all of the power of string scanning comes from. It derives from the expression evaluation mechanism of Icon: generators and goal-directed evaluation.

Exercises:

1. There is a slight flaw in the procedures given above as a model for string scanning. What is it? *Hint:* It has nothing to do with the maintenance of state variables.

2. Write a model of string scanning that uses co-expressions and a programmer-defined control operation Scan{$expr_1$, $expr_2$} in place of $expr_1$? $expr_2$.

3. Why are the explicit dereferencing operations necessary in the procedure for tab?

15.2 A Programming Idiom

Version 5.10 of Icon has a new sort option for tables that produces a single list of alternating entry and assigned values. For example,

```
a := sort(t,3)
```

assigns such a list to a. This option is more convenient and much more efficient in many cases than the standard sort option

```
a := sort(t,1)
```

that assigns to a a list of two-element lists.

Consider the problem of writing the entry and assigned values of a table, side-by-side in two columns, using the new option. The obvious approach is

```
a := sort(t,3)
every i := 1 to *a - 1 by 2 do
   write(a[i],"\t",a[i + 1])
```

Students in our class on string and list processing techniques came up with a different approach:

```
a := sort(t,3)
while write(get(a),"\t",get(a))
```

To be sure, this approach ‘‘destroys’’ the list a, but that normally makes no difference. And not only is this approach simpler than the ‘‘obvious’’ one, it is faster too!

Exercise: The elements produced by sort(t,3) are in increasing order of the entry values. How could the approach above be modified to write the output in descending order of the entry values?

15.3 Teasers

Steve Wampler provides two more teasers. Suppose that

```
t := table([])
```

Also suppose that getword is a procedure that produces a word from a line of text taken from a file in which lineno is the current line number.

1. Why does the following program segment never increase the size of the table t?

```
while word := getword() do
   put(t[word]),lineno)
```

2. Why does the following program segment increase the size of the table when the previous one does not?

```
while word := getword() do
   t[word] |||:= [lineno]
```

15.4 Other Exercises

1. What is the upper bound on the number of results that find(s1, s2) can produce?

2. The two code segments

```
return x
```

and

```
suspend x
fail
```

are often thought of as being operationally equivalent. Is this really true? If not, explain the difference.

3. It is frequently claimed that the outcome of looping expressions such as

while *$expr_1$* do *$expr_2$*

is failure — that is, that the loop itself produces no results. Is this always true? If not, give a counter example.

15.5 Trivia Corner

Write the shortest possible Icon program whose translation produces at least one instance of every different ucode instruction in Icon's intermediate language. (See TR 85-19 for a description of ucode.) Interpret ''shortest possible'' to mean the fewest number of characters.

16. Icon Newsletter #21

16.1 Solutions to Previous Problems

1. If the default value for a table is an empty list, as in

```
t := table([ ])
```

unexpected things may happen if operations are performed on this list. It is important to understand that there is only *one* default value associated with a table. For example, if a program changes the *contents* of the default value, as in

```
while word := getword() do
   put(t[word],lineno)
```

no new elements are added to t; instead every reference to t[word] produces the default value, to which the value of lineno is added. There is never any assignment to t[word].

On the other hand, in

```
while word := getword() do
   t[word] |||:= [lineno]
```

the list concatenation operation creates a *new* list and assigns it to t[word] every time the do clause is evaluated. The first time this happens for a particular value of word, the empty list default value is concatenated with a list containing the value of lineno and this new list is assigned. The default value itself is never modified.

2. The upper bound on the number of results that find(s1, s2) can produce is *max*(*s2 − *s1 + 1, 0).

3. The difference between

```
return x
```

and

```
suspend x
fail
```

is simply an extra resumption in the second case if another result is needed in the context in which the corresponding procedure is called. This extra resumption is detectable in trace output, but otherwise does not affect program behavior. However, if the identifier x is replaced by an arbitrary expression, the two cases may produce very different results. Even if this expression itself only produces a single result, it is resumed in the second case, and this resumption may produce side effects. For example,

```
return tab(i)
```

and

```
suspend tab(i)
fail
```

may behave very differently. In the first case, assuming tab(i) itself succeeds, &pos is left set to i. In the second case, if the call is resumed, tab(i) is resumed and it restores &pos to its former value. The latter form generally is used in matching procedures to assure that scanning state variables are restored to conform to the matching protocol. In such cases, return tab(i) would be an error. Another way to think about it is that return limits its argument to at most one result. Thus,

return *expr*

and

suspend *expr* \ 1
fail

are equivalent (except for the extra resumption).

4. The outcome of a looping expression such as

while $expr_1$ do $expr_2$

need not be failure. If a break expression in either $expr_1$ or $expr_2$ is evaluated, the outcome of the looping expression is the outcome of the argument of the break expression.

It is common to omit the argument of a break expression. In this case, the argument defaults to a null value. Consequently, if the break expression in

```
while expr1 do {
        :
    break
        :
    }
```

is evaluated, the outcome of the looping expression is the null value. In fact, if this effect is not wanted,

```
break &fail
```

can be used to assure the outcome of the looping expression is failure.

However, the argument of a break expression can be a generator. For example, if

```
break 1 to 5
```

is evaluated in a looping expression, the result sequence for the looping expression is { 1, 2, 3, 4, 5 }.

5. Icon programmers usually have no interest in the intermediate ''ucode'' that is produced by the translator to serve as input to the interpreter for Icon's virtual machine. However, getting into the internals of the implementation at this level can give insight into what goes on when an Icon program is actually executed. Hence the posed problem of finding the shortest possible Icon program whose translation contains at least one instance of every different

ucode instruction.

In Version 5.10 of Icon, there are 80 different executable ucode instructions. (There also are ucode declarations, which are not of interest here.) Some of these instructions are simple stack-manipulation operations. The bulk correspond to Icon's operators — there is a different ucode instruction for every different source-language operator, except for the augmented-assignment operators.

As a start, therefore, any program that produces every different ucode instruction must have at least one instance of every operator. Beyond that, there are ucode instructions related to control structures and various specialized constructions. In the absence of a list of all the ucode instructions, they can be determined empirically. Since ucode is printable text, experimentation is easy. For example,

```
icont -c inst.icn
```

produces the ucode file `inst.u1`. Getting at the ucode instruction set this way is a good exercise – it illuminates aspects of Icon that few programmers ever think about.

Once all the ucode instructions are determined, the problem becomes one of finding a minimal program that produces all of them. Some things about the syntax of Icon programs can be learned by trying this.

Here is the shortest known program that produces at least one instance of every different ucode instruction under Version 5.10 of Icon (Version 6.0 has a slightly different ucode instruction set):

```
procedure y();initial|0.0[suspend(,)to"":create+-?~=!@^*./\x/x*x%x^x<x<=x=x>=x>x~=x++x--x**x||
x<<x<<=x==x>>=x>>x~==x|||x+:=x<-x:=:x<->x~===x&x.x?:=x\&pos["]]-case[]of{1:return};end
```

The program actually consists of a single 182-character line to avoid increasing its size with newline characters. We have broken it into separate lines here to fit it on the page.

Of course, if this program is executed, it immediately terminates with a run-time error message, but that is not the issue here.

To compound this lunacy, other questions can be posed. What Icon program produces the smallest *ucode* file that contains at least one instance of every different ucode instruction? Does the program above do this? Is it possible to simultaneously minimize the sizes of the Icon program and its corresponding ucode file?

As an aside, the size of a ucode file in Version 5.10 depends on the name of the file in which its source code is contained. What is the shortest file name for a source-language program that `icont` will accept?

17. Icon Newsletter #22

17.1 Archiving Programs

Our readers frequently ask for some examples of simple programs that do not require an extensive knowledge of Icon to understand. Here are two that are quite simple and lend themselves to extensions that provide good exercises for beginning Icon programmers.

Most computer systems have some kind of a facility for combining files within a common file that can be used for transporting a large number of small files from one place to another or just to keep track of them. Such archiving facilities have many features, but most of them are system-dependent.

Here are two simple, relatively portable, Icon programs for archiving files and de-archiving them. The idea is simply to concatenate the files to be archived to make one large file. A header precedes each file, giving its name. An rather arbitrary string, "!!!!!", is chosen to distinguish headers; it cannot occur in any file to be archived (but see the exercises).

The archiving program takes a list of file names to be archived from standard input:

```
procedure main()
   while name := read() do {
      input := open(name) | stop("cannot open \"", name, "\"")
      write("!!!!!", name)
      while write(read(input))
      close(input)
      }
end
```

Note the use of an alternative to produce an error message in case a named file cannot be opened. On most systems it is important to close a file once it is no longer needed, as shown, to release space used by the i/o system for reuse.

The de-archiving program is just a little more complicated:

```
procedure main()
   while line := read() do {
      line ? if ="!!!!!" then {
         close(\out)
         out := open(name := tab(0), "w") | stop("cannot open ", name)
         }
      else write(out, line)
      }
end
```

As lines are read in, they are examined for headers. For archives constructed by the program above, the first line is always a header. (What would happen if it were not?) When a header is found, the previous file is closed, the new file name is taken from the rest of the line, and the new file is opened. If the line is not a header, however, it is written to the current file.

There is one 'trick' used here that is, in fact, good idiomatic style in Icon — the file is closed only if it is not null-valued. This happens only when the first header is encountered. At this point, out is null-valued because no assignment has been made to it yet. Thus, \out fails and close is not called. Subsequently, out is assigned a (non-null) file value and \out succeeds. This idiom takes advantage of the fact that the initial values of variables in Icon are null. It saves special coding for the start-up case.

The two programs above are crude, but generally workable. There are all kinds of possibilities for improvements and extensions:

- Modify the archiving program so that the heading string can be specified by the user.
- Modify the de-archiving program to figure out the heading string.
- Modifying the archiving program so that the heading string contains creation date and time information.
- Modify the de-archiving program so that only specified files are extracted.
- Add the facility to list the contents of an archive file without extracting the files.
- Extend the archiving facility to handle arbitrary binary files.
- Combine all the archiving and de-archiving facilities in one program that prompts the user for commands.

A Simple Calculator

The following program, based on one written by Steve Wampler when his hand-held calculator broke, is a good illustration of the use of lists as stacks and of the motivation for 'string invocation' of operations:

```
procedure main()
   local stack, token, arg1, arg2
   stack := []
   while token := read() do
      if numeric(token) then push(stack,token)
      else if proc(token,2) then {
         arg2 := pop(stack) | {
            write(&errout,"*** empty stack ***")
            next
            }
         arg1 := pop(stack) | {
            write(&errout,"*** empty stack ***")
            next
            }
         push(stack,result := token(arg1,arg2)) |
             write(&errout,"*** operation failed ***")
         write(result)
      }
      else write(&errout,"*** invalid entry ***")
end
```

The calculator takes successive lines of input as numerical computations in reverse-polish notation. If the input token is numeric, it is pushed. Otherwise, there is a check to see if the token is a binary operator: proc(token,2). This function, which is an extension to Version 5 of Icon, fails if token is not a string representing a binary operator. For example, "+" is such a string, but "$" is not. If the token represents a binary operator, two arguments are popped off the stack and the operator is applied to them. The result is pushed and the loop continues.

As exercises, consider the following:

- Why is the result of the computation assigned to result, pushed, and then written out separately?
- Consider the effect of different numeric types — such as integer, floating-point, and mixed-mode computations. Modify the calculator so that it performs only floating-point arithmetic.
- Rewrite the program to make it as short as possible and using the least number of identifiers.
- The program is designed to handle only binary operators. Modify it to handle unary ones also. Take care to consider operator symbols such as "−" that are both binary and unary.
- What happens if a token is the name of a function or procedure instead of an operator? Consider how this can be used to extend the usefulness of the calculator.
- There is no particular reason why the calculator should be limited to numeric data. Extend it to handle strings.
- Add a facility for commands that do not affect the stack but instead alter the mode of computation.

18. Icon Newsletter #23

18.1 A Program to Deal and Display Bridge Hands

The choice of data representations often is one of the most important aspects in the design of programs that perform nonnumerical computations. Not only do data representations affect program speed and space requirements, but they also may play a central role in the difficulty or ease of writing the program.

Icon offers an unusually wide variety of data types. While it provides more flexibility than is found in some other programming languages, it also presents the programmer with more choices. Sometimes the best choice is not the obvious one.

The following program, which is an adaptation of one from the Icon program library, illustrates a compact data representation that often is useful in programs that manipulate a small number of objects. This data representation also gives computational efficiency, since it allows the use of built-in operations that otherwise might have to be provided as procedures.

The problem is to produce and display hands in the game of bridge. The basic operations are shuffling the deck, dealing the cards to the players, and displaying the results. Not surprisingly, displaying the results is the most difficult part of the program.

In the game of bridge, the deck consists of 52 cards with 13 denominations in four suits. The suits are clubs, diamonds, hearts, and spades, and the denominations are 2 through 10, jack, queen, king, and ace.

There are lots of possible ways of representing the cards. Since a card has two attributes — its suit and its denomination — a record type with these fields is a possibility. This representation presents the problem (among others) of how to represent the deck — that is, how to keep track of the cards. A list of the records is a possibility. A simpler, if less elegant, choice of data representation is simply a list of 52 elements in which the position encodes the attributes of the individual card. In this case, the real representation of the card is its index in the list. In other words, 52 integers are all that are necessary, and the list is used to keep them together. Other representations are possible. For example, the string "8C" might represent the eight of clubs, and so on.

There is clearly some advantage in having a simple object to represent a card. The method used in the following program is to associate a unique character with each card. This allows groups of cards to be represented by strings, and cset operations can be used to operate on groups of cards.

Since there are 52 cards, 52 different characters are needed. For the program that follows, any 52 characters will do, but a convenient choice (purely by coincidence) is

```
deckimage := &lcase || &ucase
```

This choice has the additional advantage of facilitating debugging.

The next question is which character corresponds to which card. This decision can be made in many ways. The one chosen here is to consider the deck to be a concatenation of the suits in order, with the first 13 characters corresponding to the clubs, the next 13 to the diamonds, and so on. Thus, the characters abc ... m are clubs, the characters nop ... z are diamonds, the characters ABC ... M are hearts, and the characters NOP ... Z are spades. Except for possible debugging, however, these explicit correspondences never come up. The specific order of denominations is rather arbitrary, but it turns out to be convenient for display purposes to rank the cards according to the order of characters in the following string:

```
rank := "AKQJT98765432"
```

Thus, the character a is the ace of clubs, the character z is the two of diamonds, and so on. Again, this never comes up explicitly in the program.

It might appear that encoding the card deck as a string of characters would introduce all sorts of problems, especially in figuring out which card is which and producing output that gives an understandable representation. Actually, most of the operations in the program do not require such determinations. For example, shuffling is insensitive to the suits or denominations of cards — it simply is the rearrangement of a number of objects that are as anonymous as the backs of real cards are supposed to be.

Shuffling is a good place to start:

```
procedure shuffle(deck)
   local i
   every i := *deck to 2 by -1 do
      deck[?i] :=: deck[i]
   return deck
end
```

This procedure is an implementation of a method given by Knuth in his book, *Seminumerical Algorithms*. It operates by starting at the end of the deck, exchanging that card with a randomly chosen one, and then working down toward the beginning, chosing the exchange card from the remainder of the deck. Whether or not this produces a 'good' shuffle is somewhat of an open question, but it seems to work well in practice.

Once the deck is shuffled, it is customary to distribute the cards to the four players by dealing them one-by-one to the four players in turn. This method of distribution is more of a convention than a necessity and is motivated partly by social considerations. If the deck really is shuffled properly, it is good enough to give the first 13 cards to the first player, the next 13 to the next player, and so on. It is also a lot easier to program.

The real fun comes in displaying the results of the deal. In bridge, it is customary to separate the cards in each hand into suits and to arrange the cards in each suit from higher to lower denomination. Here is where Icon's cset and mapping operations can be used to advantage. The idea is to extract from a hand of 13 cards all of the cards of a given suit by mapping the cards of the desired suit into themselves and mapping all other cards into a single character that is not in the deck, effectively throwing away the cards that are not in the desired suit. The blank character is useful for this elimination. For example, the mapping string to discard all cards that are not clubs is constructed as follows:

```
denom := deckimage[1+:13]
blanks := repl(" ",13)
Cmap := denom || repl(blanks,3)
```

The strings denom and blanks are used here in place of a more direct construction of Cmap, since they are useful in producing maps for the other suits.

Now,

```
clubs := map(hand,deckimage,Cmap)
```

assigns to clubs a string in which all the characters corresponding to clubs are left unchanged, while all other characters are blanks. This string still is 13 characters long, and probably contains a lot of blanks. The clubs can be obtained by constructing a cset with the blank removed:

```
clubs --:= ' '
```

(There's an augmented assignment operation you don't see very often. It does not appear in the actual program, where the result is computed in a single expression.)

At this point, clubs contains all the clubs in the hand, but they are in a cset and unordered. The desired string with the clubs in order and mapped into their denominations is produced by

```
clubs := map(clubs,denom,rank)
```

The result comes out correctly, since the automatic conversion of the cset to a string in the first argument to map puts the characters in alphabetical order.

That's about all there is to it, except for the mechanics of handling all of the suits in all of the hands and formatting the output in a manner that is customary, with the four hands arranged according to the points of the compass. Here's the complete program, arbitrarily set up to print five sets of hands. Comments have been removed to save space.

```
global deck, deckimage, handsize
global suitsize, denom, rank, blanks

procedure main()
   deck := deckimage := &lcase || &ucase
   handsize := suitsize := *deck / 4
   rank := "AKQJT98765432"
   blanks := repl(" ",suitsize)
   denom := &lcase[1+:suitsize]

   every 1 to 5 do display()
end

procedure display()
   local layout, i
   static bar, offset

   bar := "\n" || repl("-",33)
   offset := repl(" ",10)
```