Programming in Icon; Part II — Programming with Co-Expressions*

Ralph E. Griswold

TR 87-6

June 4, 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; Part II — Programming with Co-Expressions

1. Introduction

This report is the second in a series that deals with various aspects of programming in Icon. The first report [1] dealt with generators. Co-expressions are the topic of this second report for two reasons: (1) they derive their utility from generators, and (2) they seem to present more problems to programmers than other features of Icon.

The first two parts of this report treat the basic aspects of co-expressions and their common uses. The third part describes the use of co-expressions to provide programmer-defined control operations. More advanced uses of co-expressions are covered in the final part. Appendices contain examples illustrating programming techniques and exercises for interested readers.

This report assumes a familiarity with Version 6 of Icon [2], although co-expressions themselves are described in detail here. A good understanding of generators [1] is a prerequisite for the material that follows.

Some features of co-expressions that are described in this report were added in Version 6.4 of Icon. Such features are noted.

2. Basic Features of Co-Expressions

2.1 Motivation

Generators are a central feature of Icon. They contribute more to its distinctive character than any other feature of the language. The results that a generator produces are determined by control structures. For example,

```
every write(find(s1,s2))
```

causes the generator find to produce all of its results. Goal-directed evaluation has no visible representation in the syntax of Icon and is implicit in all expression evaluation, but it also is a control structure. For example,

```
if i = find(s1,s2) then write(i)
```

causes find to produce its results only until one of them is equal to i. If none of the results is equal to i, all results are produced in the process of attempting to satisfy the comparison. In both cases, the situation can be understood in terms of the suspension of a generator when it produces a result and its resumption by the control structure to produce another result.

The use of control structures to cause generators to produce results allows concise and natural programming styles in many situations. A generator can produce its results only at the site of its evaluation, however. Furthermore, its results can be produced only in the order determined by control structures. Nested generators are resumed in a lifo fashion, producing a 'cross-product' form of evaluation. This order of evaluation corresponds to a depth-first production of results, which is natural and useful in problems that involve searching a solution space. However, there is no way of producing the results of generators in parallel using the control structures provided by Icon. For example,

```
every write((4 to 6) + (0 to 2))
```

writes 4, 5, 6, 5, 6, 7, 6, 7, 8, not 4, 6, 8.

Co-expressions are motivated by the need to free generators from the sites where they appear and to obtain their results as needed.

2.2 Co-Expression Creation

A co-expression is a data object that contains a reference to an expression and an environment for the evaluation of that expression. A co-expression is created by the control structure

```
create expr
```

The create expression does not evaluate *expr*. Instead, it produces a value of type co-expression that references *expr*. This value can be assigned to a variable, passed to a procedure, returned from a procedure, and in general handled like any other value. A co-expression contains not only a reference to its argument expression, but also a copy of the dynamic local variables for the procedure in which the create appears. These copied variables have the same values as the corresponding dynamic local variables at the time the create expression is evaluated. This effectively frees *expr* from the place in the program where it appears and provides it with an environment of its own.

An example is

```
procedure writepos(s1,s2,s)
   locs1 := create find(s1,s)
   locs2 := create find(s2,s)
               :
               :
end
```

Here locs1 and locs2 are assigned co-expressions corresponding to the expressions find(s1,s) and find(s2,s), respectively.

The reserved word create has lower precedence than any operator symbol. For example,

```
articles := create "a" | "an" | "the"
```

groups as

```
articles := create ("a" | "an" | "the")
```

Parentheses are used in this report for clarity in most examples, although they usually are unnecessary.

2.3 Activation of Co-Expressions

Control is transferred to a co-expression by *activating* it with the operation @e. At this point, execution continues in the expression referenced by e. When this expression produces a result, control is returned to the activating expression and the result that is returned becomes the result of the activating expression. For example,

```
write(@articles)
```

transfers control to the expression

```
"a" | "an" | "the"
```

which produces the result "a" and returns to the activating expression, where that result is written out.

If the co-expression is activated again, control is transferred to the place in its expression where it last produced a result, and execution continues there. Thus, subsequent to the activation above,

```
second := @articles
```

assigns "an" to second and

```
third := @articles
```

assigns "the" to third. If article is activated again, the activation fails because there are no more results for the expression that is resumed. In general, the activation operation produces at most one result and fails when all results of the co-expression have been produced. Consequently,

```
while write(@locs1)
```

writes out all the positions at which s1 occurs in s and the loop terminates when find(s1,s) has no more results and @locs1 fails. Note that this expression produces the same results as

```
every write(find(s1,s))
```

In general, in the absence of side effects

```
|@e
```

has the same result sequence as the expression referenced by e. Activation may occur at any time and place, however, while producing results by iteration is confined to the site at which the expression occurs.

An important aspect of activation is that it produces at most one result. Therefore, the results of a generator can be produced one at a time, where and when they are needed. For example, the results of generators can be intermingled, as in

```
while write(@locs1," ",@locs2)
```

which writes the locations of s1 and s2 in s, side-by-side in columns. Since activation fails when there are no more results, the loop terminates when either of the generators runs out of results.

The result produced by a co-expression is dereferenced according to the same rules that apply to procedures [2][1]. Specifically, if the result is a dynamic local variable, it is dereferenced.

2.4 Refreshing Co-Expressions

Since activation produces the next result for a co-expression, it has the side effect of changing the 'state' of the co-expression, and effectively consumes a result, much in the way the reading the line of a file consumes that line. Sometimes it is useful to 'start a co-expression over'. Although there is no way to reset the state of a co-expression to its initial value at the time of its creation, the operation ^e produces a 'refreshed' *copy* of a co-expression e. The term 'refresh' is somewhat of a misnomer, since it sounds like e is refreshed; in fact, it does not change e, but instead produces a new co-expression. Typical usage is

```
e := ^e
```

It is worth noting that copy(e) simply returns e; it does not produce a physically distinct copy of e. There is no way to make a physically distinct copy of a co-expression in its current state.

2.5 Number of Values Produced

The 'size' of a co-expression, given by *e, is the number of results it has produced. Each successful activation of a co-expression increments its size (which starts at 0). For example,

```
if *e = 0 then write(@e)
```

writes a result for e, provided it has not yet produced a result. Of course, @e fails if there are no results at all. Similarly,

```
while @e
write(*e)
```

writes the number of results in the result sequence for the expression referenced by e. Such usage obviously is risky, since an expression may have an infinite result sequence.

2.6 Co-Expression Environments

As mentioned earlier, a co-expression is created with *copies* of the dynamic local variables for the procedure in which the create expression occurs. These copies have the values of the corresponding local variables at the time the create expression is evaluated. This aspect of co-expression creation has several implications.

Since every co-expression has its own copies of dynamic local variables, two co-expressions can share a variable only if it is global or static. Copies of local variables may lead to programming mistakes, since the names of the variables in co-expressions are the same, making the variables appear to be the same.

[1] Page 133 of this book states that all results produced by co-expressions are dereferenced. That is incorrect.

When a new co-expression is created by ^e, copies of the dynamic local variables are made again, but with the values they had at the time that e was created. Consider, for example,

```
local i
i := 1
seq1 := create |(i *:= 2)
i := 3
seq2 := create |(i *:= 2)
```

The results produced by successive activations of seq1 are 2, 4, 8, 16, ... , while the results produced by seq2 are 3, 6, 12, 24, ... Then, for

```
seq3 := ^seq1
```

the results produced by seq3 are 2, 4, 8, 16, ... , since the initial value of i in seq1 is 1 and it is not effected by the subsequent assignment of 3 to i — the two variables are distinct.

2.7 String Images of Co-Expressions

The function image(e) produces a string image of the co-expression e with an identifying number and its 'size'. For example, if e is the fifth co-expression created during program execution and it has produced 10 results, then image(e) is

```
co–expression #5 (10)
```

Note: The identifying number was added in Version 6.4 of Icon; prior to that, only the size was given.

3. Using Co-Expressions

As mentioned earlier, co-expressions are useful in situations in which the production of the results of a generator needs to be controlled, instead of occurring automatically as the result of goal-directed evaluation or iteration. Since most of the utility of co-expressions comes from generators, most co-expression applications depend on the use of generators.

3.1 Labels and Tags

In some situations a sequence of labels or tags is needed. For example, an assembler may need a source of unique labels for referencing the code it produces, while a procedure that traverses a graph may need tags to name nodes.

A generator, such as

```
"L" || (1 to limit)
```

is a convenient way of formulating a sequence of labels. However, in an assembler, the need for a new label occurs at different times and places in the program and a single generator such as the one above cannot be used. One solution to this problem is to avoid generators and use a procedure such as

```
procedure label(limit)
   static i
   initial i := 0
   if i = limit then fail
   else return "L" || (i +:= 1)
end
```

Thus, label() produces the next label.

The use of such a procedure gives up the power of expression evaluation in Icon, since it encodes, at the source level, the computation that a generator does internally and automatically. To use a generator, a co-expression such as

```
label := create ("L" || (1 to limit))
```

suffices. Here, @label produces the next label.

3.2 Parallel Evaluation

One of the common paradigms that motivates co-expression usage is the generation of results from generators in parallel. Consider, for example, producing a tabulation showing the decimal, hexadecimal, and octal values for all the characters in &cset, along with the images of the corresponding characters. The values for each column are easily produced by generators:

```
0 to 255

!"0123456789ABCDEF" || !"0123456789ABCDEF"

(0 to 3) || (0 to 7) || (0 to 7)

image(!&cset)
```

However, in order to produce a tabulation, the results of these generators are needed in parallel. This cannot be done by simple expression evaluation. The solution is to create a co-expression for each generator and to activate them in parallel:

```
decimal := create (0 to 255)
hexadecimal := create (!"0123456789ABCDEF" || !"0123456789ABCDEF")
octal := create ((0 to 3) || (0 to 7) || (0 to 7))
character := create image(!&cset)
```

Then an expression such as

```
while write(right(@decimal,3),"    ",@hexadecimal,"    ",@octal,"    ",@character)
```

can be used to produce the tabulation:

```
  0      00      000      "\000"
  1      01      001      "\001"
  2      02      002      "\002"
  3      03      003      "\003"
  4      04      004      "\004"
                  :
 98      61      141      "a"
 99      62      142      "b"
100      63      143      "c"
101      64      144      "d"
                  :
251      FB      373      "\373"
252      FC      374      "\374"
253      FD      375      "\375"
254      FE      376      "\376"
255      FF      377      "\377"
```

Another example of parallel evaluation occurs when the results produced by a generator are to be assigned to a sequence of variables. Suppose, for example, that the first three results for find(s1,s2) are to be assigned to x, y, and z, respectively. This can be done as follows:

```
loc := create find(s1,s2)
every (x | y | z) := @loc
```

Of course, if find(s1,s2) has fewer than three results, not all of the assignments are made.

3.3 Result Sequences

Result sequences [1] describe the capability that expressions have to produce results. For example, the expression 1 to 5 is capable of producing 1, 2, 3, 4, and 5 and has the result sequence {1, 2, 3, 4, 5}. This is written

$\mathcal{S}$(1 to 5) = {1, 2, 3, 4, 5}

Of course, the results that 1 to 5 actually produces depends on the context in which it is evaluated. A result sequence is just an abstraction for the potential results of an expression.

As an abstraction, result sequences provide a useful tool for program formulation. For example, it may be useful to think in terms of an expression that produces the positive integers or one that produces all the five-letter words in a file. Given this approach to conceptualizing a program, the problem becomes one of producing desired sequences.

Several paradigms and numerous examples of expressions that produce various kinds of sequences are given in [1]. Co-expressions provide many ways for producing expressions that have result sequences that cannot be formulated on the basis of generators alone, or that can be formulated using generators only with great difficulty.

The basic relationship between the results produced by an expression *expr* and the co-expression

e := create *expr*

is given by

$\mathcal{S}$(*expr*) = $\mathcal{S}$(|@e)

In order for this relation to hold, *expr* must be *invariant*. That is, its result sequence must be self-contained and time-independent [1] — it must not depend on any factors outside of *expr*, and it must produce the same result sequence whenever it is evaluated. This does not mean that *expr* cannot have side effects, but it does mean its result sequence cannot depend on side-effects. For example,

{i := 0; 1 to i}

is invariant, but

{1 to i}

is not, in general. The formulations that follow assume that all expressions are invariant.

The flexibility that co-expressions provide in the formulation of result sequences is illustrated by one that produces only the even-numbered results in the result sequence for *expr*:

|(@e,@e)

where

e := create *expr*

Similarly, the result sequences for $expr_1$ and $expr_2$ can be interleaved by

|(@e1 | @e2)

where

e1 := create $expr_1$
e2 := create $expr_2$

Note that such formulations depend on the fact that the activation of a co-expression produces at most one result.

4. Programmer-Defined Control Operations

Control structures are provided so that the flow of control during program execution can be modified depending on the results produced by expressions. In Icon, most control structures depend on result sequences. For example, the result sequence for

if $expr_1$ then $expr_2$ else $expr_3$

depends on whether or not the result sequence for $expr_1$ is empty.

Icon's built-in control structures are designed to handle the situations that arise most often in programming. There are many possible control structures in addition to the ones that Icon provides (parallel evaluation is perhaps the most obvious).

Co-expressions make it possible to extend Icon's built-in repertoire of control structures. Consider a simple example of parallel evaluation:

```
procedure parallel(e1,e2)
   local x
   repeat {
      if x := @e1 then suspend x else fail
      if x := @e2 then suspend x else fail
      }
end
```

where e1 and e2 are co-expressions. For example, the result sequence for

```
parallel(create !&lcase, create !&ucase)
```

is {"a", "A", "b", "B", ... "z", "Z"}. In this case, both co-expressions have the same number of results. In general, the result sequence for parallel(e1,e2) terminates when either e1 or e2 runs out of results.

This formulation of parallel evaluation is cumbersome, since the user must explicitly create co-expressions for each invocation of parallel. Icon provides a form of procedure invocation in which the arguments are passed as a list of co-expressions. This form of invocation is denoted by braces instead of parentheses, so that

p{*expr*$_1$, *expr*$_2$, ..., *expr*$_n$}

is equivalent to

p([create *expr*$_1$, create *expr*$_2$, ..., create *expr*$_n$])

Thus, p is called with a single argument, so that of an arbitrary number of co-expressions can be given.

Using this facility, parallel evaluation can be formulated in as follows:

```
procedure Parallel(a)                  # called as Parallel{expr1,expr2}
   local x
   repeat {
      if x := @a[1] then suspend x else fail
      if x := @a[2] then suspend x else fail
      }
end
```

For example, the result sequence for Parallel{!&lcase, !&ucase} is {"a", "A", "b", "B", ... "z", "Z"}.

It is easy to extend parallel evaluation to an arbitrary number of arguments:

```
procedure Parallel(a)                  # called as Parallel{expr1,expr2, ...,exprn}
   local x, e
   repeat
      every e := !a do
         if x := @e then suspend x else fail
end
```

Another example of the use of programmer-defined control operations is a procedure the generalizes alternation to an arbitrary number of expressions:

```
procedure Alt(a)                       # called as Alt{expr1,expr2, ...,exprn}
   local x
   every x := !a do suspend |@x
end
```

Some operations on sequences are more useful if applied in parallel, rather than on the cross product of results. An example is

```
procedure Add(a)                    # called as Add{expr1,expr2}
   suspend |(@a[1] + @a[2])
end
```

String invocation [3] often can be used to advantage in programmer-defined control operations. An example is a procedure that 'reduces' a sequence by applying a binary operation to successive results:

```
procedure Reduce(a)                 # called as Reduce{op, expr}
   local op, opnds, result
   op := @a[1] | fail               # get the operator
   opnds := a[2]                    # get the co-expression for the operands
   result := op(@opnds,@opnds) | fail
   while result := op(result,@opnds)
   return result
end
```

For example, the result of Reduce{"+",1 to 10} is 55.

Another application for programmer-defined control operations is in production of a string representation of a result sequence:

```
procedure Seqimage(a)               # called as Seqimage{expr,i}
   local seq, result, i
   seq := ""
   i := @a[2] | 10                  # limit on number of results
   while result := image(@a[1]) do {
      if *a[1] > i then {
         seq ||:= ", ..."
         break
         }
      else seq ||:= ", " || result
      }
   return "{" || seq[3:0] || "}" | "{}"
end
```

For example, the result produced by Seqimage{!&lcase} is {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", ...}.

There are many other applications of programmer-defined control operations. See [4, 5].

5. Advanced Uses of Co-Expressions

Although co-expressions are motivated by the need to control the results produced by generators, they also can be used as coroutines. A general description of coroutine programming is beyond the scope of this report; see [6-8].

Note: In Versions 6.0 through 6.3 of Icon, the activation of an active co-expression was prohibited because of a problem in the implementation. That restriction was lifted in Version 6.4. Most of the material in the following sections depends on the ability to activate an active co-expression.

5.1 Transfer of Control Among Co-Expressions

As illustrated earlier, a co-expression can transfer control to another co-expression by two means:

- Activating it explicitly, as in @e.
- Returning to it implicitly by producing a result.

Despite the appearance of dissimilarity between these two means for transferring control, they really are symmetric.

It is important to understand that transferring control from one co-expression to another by either means changes the place in the program where execution is taking place and changes the environment in which expressions are

evaluated. Unlike procedure calls, however, transfer of control among co-expressions is not hierarchical.

This is illustrated by the use of co-expressions as coroutines. Consider, for example, the following program:

```
global E1, E2

procedure main()

   E1 := create note(E2,"co–expression E2")
   E2 := create note(E1,"co–expression E1")

   @E1

end

procedure note(e,tag)
   local i

   i := 0

   repeat {
      write("activation ",i +:= 1," of ",tag)
      @e
      }

end
```

When E1 is activated, the procedure note is called with two arguments: the co-expression E2 and a string used for identification. Execution continues in note. A line of output is produced, and E2 is activated. As a result, there is another invocation of note. It writes a line of output and activates E1. At this point, control is transferred to the first call of note at the point it activated E2. Control then transfers back and forth between the two procedure calls, and the output produced is

```
activation 1 of co–expression E2
activation 1 of co–expression E1
activation 2 of co–expression E2
activation 2 of co–expression E1
activation 3 of co–expression E2
activation 3 of co–expression E1
activation 4 of co–expression E2
activation 4 of co–expression E1
activation 5 of co–expression E2
activation 5 of co–expression E1
activation 6 of co–expression E2
activation 6 of co–expression E1
              .
              .
              .
```

This continues endlessly and neither procedure call ever returns.

5.2 Built-In Co-Expressions

There are three built-in co-expressions that facilitate transfer of control among co-expressions: &source, &current, and &main.

The value of &source is the co-expression that activated the currently active co-expression. Thus,

```
@&source
```

'returns' to the activating co-expression.

The value of &current is the co-expression in which execution is currently taking place. For example,

```
process(&current)
```

passes the current co-expression to the procedure process. This could be used to assure return of control to the co-expression that was current when process was called.

The value of &main is the co-expression of the main procedure. This corresponds to the invocation of the main procedure to initiate program execution, which can be viewed as

```
@(create main(a))
```

The co-expression &main is the first co-expression that is created in every program and has the identifying number 1.

If program execution is taking place in any co-expression,

```
@&main
```

returns control to the co-expression for the procedure main at the point a co-expression was activated. Note that this location need not be in the procedure main itself, since main may have called another procedure from which the activation of a co-expression took place.

Note: &current was added in Version 6.4 of Icon.

Transmission

A results can be transmitted to a co-expression when it is activated. Transmission is done by the operation

```
expr @ e
```

where e is activated and the result of *expr* is transmitted to it. In fact, @e is just an abbreviation for

```
&null @ e
```

so that every activation actually transmits a result to the co-expression that is being activated.

On the first activation of a co-expression, the transmitted result is discarded, since there is nothing to receive it. On subsequent activations, the transmitted result becomes the result of the expression that activated the current co-expression.

The use of transmission is illustrated by the following program, which reads in lines from standard input, breaks them up into 'words', and writes out the words on separate lines. Co-expressions are used to isolate the tasks: reading lines, producing the words from the lines, and writing out the words.

```
global words, lines, writer

procedure main()

   words := create word()
   lines := create reader()
   writer := create output()
   @writer

end

procedure word()
   static letters

   initial letters := &lcase || &ucase

   while line := @lines do
      line ? while tab(upto(letters)) do
         tab(many(&lcase)) @ writer

end
```

```
procedure reader()
   while read() @ words
end
procedure output()
   while write(@words)
   @&main
end
```

Note that output activates main to terminate program execution.

Note: This example is designed to illustrate transmission, not a recommended programming technique. The problem above can be solved more simply by using generators and procedure calls, since there is nothing in the problem that requires either the liberation of a generator from its lexical site or coroutine control flow. Coroutine programming generally is appropriate only in large programs that benefit from the organization that coroutines allow. Knuth [6] says 'It is rather difficult to find short, simple examples of coroutines which illustrate the importance of the idea; the most useful coroutine applications generally are quite lengthy', and Marlin [7] remarks ' ... the choice of an example program is ... difficult ... . The programming methodology is intended for programming-in-the-large'.

5.3 Tracing Co-Expressions

Beginning with Version 6.4 of Icon, co-expression activation and return is traced if the value of &trace is non-zero. As for function calls and returns, the value of &trace is decremented for each trace message. The form of co-expression tracing is illustrated by the following program:

```
procedure main()
   local lower, upper
   &trace := -1
   lower := create !&lcase
   upper := create !&ucase
   while write(@lower," ",@upper)
end
```

If this program is in the file trace.icn, the trace output is:

```
trace.icn: 6     | main; #1 : &null @ #2
trace.icn: 4     | main; #2 returned "a" to #1
trace.icn: 6     | main; #1 : &null @ #3
trace.icn: 5     | main; #3 returned "A" to #1
trace.icn: 6     | main; #1 : &null @ #2
trace.icn: 4     | main; #2 returned "b" to #1
trace.icn: 6     | main; #1 : &null @ #3
trace.icn: 5     | main; #3 returned "B" to #1
trace.icn: 6     | main; #1 : &null @ #2
trace.icn: 4     | main; #2 returned "c" to #1
trace.icn: 6     | main; #1 : &null @ #3
                    .
                    .
                    .
trace.icn: 4     | main; #2 returned "z" to #1
trace.icn: 6     | main; #1 : &null @ #3
trace.icn: 5     | main; #3 returned "Z" to #1
trace.icn: 6     | main; #1 : &null @ #2
trace.icn: 4     | main; #2 failed to #1
trace.icn: 7     main failed
```

Note that activation really is transmission of the null value.

Acknowledgements

Steve Wampler designed and implemented the original co-expression facility for Icon. Steve Wampler and Ken Walker provided several of the examples of co-expression usage given in this paper. Steve, Ken, Janalee O'Bagy, Dave Gudeman, and Kelvin Nilsen all provided helpful suggestions on various aspects of co-expressions. In addition, Janalee and Ken read drafts of this report and made several suggestions on improving the presentation.

References

1. R. E. Griswold, *Programming in Icon; Part I — Programming with Generators*, The Univ. of Arizona Tech. Rep. 85-25, 1985.
2. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
3. R. E. Griswold, W. H. Mitchell and J. O'Bagy, *Version 6 of Icon*, The Univ. of Arizona Tech. Rep. 86-10b, 1986.
4. R. E. Griswold and M. Novak, "Programmer-Defined Control Operations", *Computer J. 26*, 2 (May 1983), 175-183.
5. M. Novak and R. E. Griswold, *Programmer-Defined Argument Evaluation Regimes*, The Univ. of Arizona Tech. Rep. 82-16, 1982.
6. D. E. Knuth, *The Art of Computer Programming, Volume I*, Addison-Wesley, 1968, p. 191.
7. C. D. Marlin, *Coroutines; A Programming Methodology, A Language Design and Implementation*, Springer Verlag, 1980.
8. O. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972, pp. 184-193..

Appendix A — A Co-Expression Formulation of the N-Queens Problem

The following program, adapted from one written by Steve Wampler, illustrates the use of co-expressions to solve the problem of *n* non-attacking queens on an *n*-by-*n* chessboard. See [2] for a description of the problem and a solution that does not involve co-expressions.

In this program, the list driver contains *n*+2 co-expressions. Co-expressions 2 through *n*+1 correspond to columns on the chessboard and place the queens. Co-expression *n*+2 displays the chessboard when all *n* queens have been placed. The first co-expression, &main, returns control to the main procedure when there are no more solutions.

```
global queens, driver, solution

procedure main(args)
   local i

   queens := args[1] | 8                    # Default is 8 queens
   driver := list(queens + 2)               # List of queen placement routines
   solution := list(queens)                 # ... and a list of column solutions

   driver[1] := &main                       # 1st co–expression is &main
   every i := 1 to queens do                # 2 to queens + 1 are queen placement
      driver[i+1] := create q(i)            #    co–expressions, one per column.
   driver[queens + 2] := create show()      # queens + 2nd co–expression display the board

   write(queens,"–Queens:")
   @driver[2]                               # Start by placing queen in first column

end

# q(c) – place a queen in column c.

procedure q(c)
   local r
   static up, down, rows

   initial {
      up := list(2 * queens – 1,0)
      down := list(2 * queens – 1,0)
      rows := list(queens,0)
      }

   repeat {
      every (0 = rows[r := 1 to queens] = up[queens + r – c] = down[r + c – 1] &
         rows[r] <– up[queens + r – c] <– down[r + c – 1] <– 1) do {
            solution[c] := r                # Record placement
            @driver[c + 2]                  # Try to place next queen
            }
      @driver[c]                            # Tell previous queen placer "try again"
      }

end

# Show the solution on a chess board.

procedure show()
   static count, line, border
```

```
   initial {
      count := 0
      line := repl("|    ",queens) || "|"
      border := repl("----",queens) || "-"
      }

   repeat {
      write("solution: ", count +:= 1)

      write("   ", border)
      every line[4 * (!solution - 1) + 3] <- "Q" do {
         write("   ", line)
         write("   ", border)
         }

      write()
      @driver[queens + 1]                    # Tell last queen placer to try again
      }

end
```

Appendix B — A Lexical Analyzer and Parser

This program, which was written by Ken Walker, converts expressions in infix form to prefix form. The grammar for the infix form is:

```
<prog>   ::=  <stmt>  |  <prog> ; <stmt>
<stmt>   ::=  <id> := <expr>
<expr>   ::=  <term>  |  <expr> <add_op> <term>
<term>   ::=  <factor>  |  <term> <mult_op> <factor>
<factor> ::=  <id>  |  <integer>  |  ( <expr> )
```

There is a procedure that implements each of the productions above. For example, <prog> is handled by prog(). The classes <id>, <integer>, <add_op>, and <mult_op> are recognized by the lexical analyzer. Tabs and blanks are treated as white space. Comments start with # and continue to the next newline.

The results of the co-expression lex come from two result sequences joined by alternation. The first is the sequence of tokens from the input. The second is an endless sequence of end-of-file tokens. The tokens from the input are produced by a scanning expression. The subject of the scanning expression is the sequence of input lines. For each input line, get_tok() generates the tokens from the line. On an end-of-line, get_tok() fails and the next input line is generated for the subject.

```
global lex                              # co-expression for lexical analyzer
global next_tok                         # next token from input

record token(type, string)

procedure main()
   lex := create ((!&input ? get_tok()) | |token("eof", "eof"))
   prog()
end

#
# get_tok is the main body of lexical analyzer
#
procedure get_tok()
   local tok
   static letters, digits

   initial {
      letters := &ucase ++ &lcase
      digits := '0123456789'
      }

   repeat {                             # skip white space and comments
      tab(many(' \t'))
      if ="#" | pos(0) then fail
```

```
      if any(letters) then                  # determine token type
         tok := token("id", tab(many(letters ++ '_')))
      else if any(digits) then
         tok := token("integer", tab(many(digits)))
      else case move(1) of {
         ";": tok := token("semi", ";")
         "(": tok := token("lparen", "(")
         ")": tok := token("rparen", ")")
         ":": if ="=" then
                 tok := token("assign", ":=")
              else
                 tok := token("colon", ":")
         "+": tok := token("add_op", "+")
         "-": tok := token("add_op", "-")
         "*": tok := token("mult_op", "*")
         "/": tok := token("mult_op", "/")
         default: err("invalid character in input")
         }
      suspend tok
      }
end

#
# The procedures that follow make up the parser
#

procedure prog()
   next_tok := @lex
   stmt()
   while next_tok.type == "semi" do {
      next_tok := @lex
      stmt()
      }
   if next_tok.type ~== "eof" then
      err("eof expected")
end

procedure stmt()
   if next_tok.type ~== "id" then
      err("id expected")
   write(next_tok.string)
   if (@lex).type ~== "assign" then
      err(":= expected")
   next_tok := @lex
   expr()
   write(":=")
end

procedure expr()
   local op
```

```
   term()
   while next_tok.type == "add_op" do {
      op := next_tok.string
      next_tok := @lex
      term()
      write(op)
      }
end

procedure term()
   local op

   factor()
   while next_tok.type == "mult_op" do {
      op := next_tok.string
      next_tok := @lex
      factor()
      write(op)
      }
end

procedure factor()
   case next_tok.type of {
      "id" | "integer": {
         write(next_tok.string)
         next_tok := @lex
         }
      "lparen": {
         next_tok := @lex
         expr()
         if next_tok.type ~== "rparen" then
            err(") expected")
         else
            next_tok := @lex
         }
      default:
         err("id or integer expected")
      }
end

procedure err(s)
   stop(" ** error **  ", s)
end
```

Appendix C — Exercises

1. Write a procedure that generates a sequence of label generators, each with a different letter prefix. Show how this procedure in turn might be used in a co-expression.

2. Discuss the following proposed alternative to the method given at the end of Section 3.2 for assigning the results of an expression to a sequence of variables:

```
loc := create find(s1,s2)
var := create (x | y | z)
while @var := @loc
```

3. Re-write the programmer-defined control operation for parallel evaluation, using repeated alternation in place of the if-then-else control structure.

4. Write a programmer-defined control operation that 'sections' a result sequence, producing only results *i* through *j*.

5. Write a programmer-defined control operation for the Lisp cons control structure.

6. Write a programmer-defined control operation that omits the initial *i* results of a result sequence.

7. Extend Seqimage in Section 4 so that a third argument can be used to specify the inclusion of a specified number of trailing results if the number of results exceeds the limit specified by the second argument.

8. Given an expression *expr* that is invariant, use co-expressions to produce expressions that have the following result sequences:

   (a) The odd-numbered results in the result sequence for *expr*.

   (b) Every third result in the result sequence for *expr*.

   (c) The running sum of the results for *expr*.