# The Implementation of an Experimental Language for Manipulating Sequences*

*Ralph E. Griswold*

TR 83-20

December 31, 1983

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS81-01916.

# The Implementation of an Experimental Language for Manipulating Sequences

## 1. Introduction

Seque [1] is an experimental programming language for manipulating sequences as data objects. Such sequences are designed to be used in both storage-oriented and production-oriented modes. In the storage-oriented mode, sequences resemble vectors and are accessed by position, as in

```
X!i
```

which produces the ith element of the sequence X. In the production-oriented mode, sequences are characterized by values that are generated successively by a computation, as in

```
gen{1 to 10}
```

which is a sequence consisting of the integers from 1 to 10.

This report describes the implementation of Seque, which is embedded in Icon [2]. When a Seque program is run, it first is translated into a standard Icon program. This Icon program is translated and linked with a run-time library of Icon procedures. The implementation makes extensive use of co-expressions [3]. Code for Seque is given in appendices.

## 2. Seque Language Features

Sequences are data objects of type Sequence. In the sections that follow, identifiers that begin with uppercase letters indicate sequence-valued expressions.

The following global identifiers have predefined sequences as values:

| | |
|---|---|
| Phi | $\Phi \equiv \{\}$ |
| Izero | $\mathcal{I}_0 \equiv \{0, 1, 2, 3, \ldots\}$ |
| Iplus | $\mathcal{I}_+ \equiv \{1, 2, 3, 4, \ldots\}$ |

The following expressions are available for manipulating sequences:

| *feature* | *formal notation* | *Seque syntax* |
|---|---|---|
| explicit sequence | $\{x_1, x_2, x_3, \ldots\}$ | seq{x1, x2, x3, ... } |
| concatenation | $X \oplus Y$ | Cat(X, Y) |
| subsequence | $X_{i:j}$ | Subseq(X, i, j) |
| pre-truncation | $X \downarrow i$ | X %% i (%X is equivalent to X %% 1) |
| post-truncation | $X \uparrow i$ | X ^^ i |
| generator | $[I : \lambda(j)X]$ | gen[ : I : lambda(j) X ] |
| sequence length | $\mid X \mid$ | Length(X) |
| element selection | $X!i$ | X!i |
| reduction | $Red_p(X)$ | Red(X, p) |

The following procedures related to sequences also are available in Seque:

- Compress(X) converts a sequence containing sequences as elements to a sequence of scalar values.
- Copy(X) creates a copy of the sequence X.
- Empty(X) succeeds and produces X if X is an empty sequence but fails otherwise.
- Image(X, i) produces a string image of the first i values of X.
- Read(f) produces a sequence of values resulting from reading file f.
- Trace(X, i) writes the result of Image(X, i) and produces the value of X.
- Write(X) writes all the elements of X with separating linefeeds.
- Writes(X) writes all the elements of X without separating linefeeds.

In addition, gen{*expr*} produces a sequence corresponding to the Icon result sequence for *expr*.

To differentiate gen[*expr*] from gen{*expr*}, the former is referred to as a Seque generator, while the latter is referred to as an Icon generator.

Seque also has recurrence declarations that allow recurrence relations to be specified concisely. See Reference 1 for the details of recurrence declarations and their use.

There are two kinds of expression evaluation available in Seque programs: ordinary Icon evaluation with the usual operations and functions, and Seque evaluation, in which expressions are limited to at most one result and operations and functions are extended to apply to values of type Sequence. The type of evaluation used depends on the context, which is determined as follows:

- Seque evaluation applies in procedures that are declared with the reserved word procedure. Such procedures are called *Seque procedures*.
- Within a Seque procedure, Seque evaluation applies in all expressions with operator syntax and in function and procedure calls in which the function or procedure is given by an identifier that begins with an initial lowercase letter.
- Icon evaluation applies in procedures that are declared with the reserved words icon procedure. Such procedures are called *Icon procedures*. For example,

```
icon procedure main()
        :
        :
end
```

  declares the main procedure to be an Icon procedure. The main procedure need not be an Icon procedure.
- Icon evaluation applies in recurrence declarations.
- Seque evaluation applies in explicit sequences and Seque generators, regardless of the context in which such expressions occur.
- Icon evaluation applies in Icon generators, regardless of the context in which such expressions occur.
- Icon evaluation applies all assignment operations, regardless of where they occur.
- Control structures behave in the way they do in Icon, regardless of the context in which they appear.

### 3. The Representation of Sequences

Since sequences are designed to be used in both storage and production paradigms, a hybrid implementation is used. Lists are used to store values that are produced by the activation of co-expressions.

A sequence is a record of type Sequence, as given by the declaration

```
record Sequence(a, e)
```

The a field is a list and the e field is a co-expression. For example, the sequence Iplus is given by

```
Iplus := Sequence([], create seq())
```

where seq() is a function in the experimental extensions to Icon [4] that produces the positive integers in

sequence. Similarly,

```
Tens := gen{10 to 1000 by 10}
```

assigns

```
Sequence([], create 10 to 1000 by 10)
```

to Tens.

An essential aspect of every newly created sequence is that its a field contains an empty list, while its e field contains a newly created co-expression. Thus a newly created sequence has no stored elements but has the capability for producing them.

When an element of a sequence is needed, as in

```
Tens!2
```

the list is first examined to see if it contains the specified element. If it does, that element is returned. In this example, that amounts to

```
if x := (Tens.a)!2 then return x
```

If the element is not in the list, the co-expression is activated to produce values, which are put in the list until the desired element has been produced. The expression that "transfers" values from the co-expression for a sequence X to its list has the form

```
while expr do
   put(X.a, @X.e) | fail
```

where *expr* is an expression that controls the transfer.

As a sequence is referenced with progressively larger indices, its list increases in size and its co-expression is depleted. This is essentially a value-on-demand strategy that keeps the use of storage to a minimum, subject to the constraint that a value that is once produced is not discarded. Note that there is no way to compute the $i$th element in a sequence without computing the values for $1, 2, \ldots, i-1$.

Because a co-expression produces a value every time that it is activated, that value must be stored for possible future reference. In addition, the same co-expression cannot be used in two different sequences. For example, if a sequence X were copied by

```
Y := Sequence(copy(X.a), X.e)
```

there would be potentially disastrous consequences, since both X and Y would share the same co-expression; the production of a result in one of these sequences would cause that result to be skipped in the other sequence. Instead, the form of the copy is

```
Y := Sequence([], ^X.e)
```

Note that ^X.e creates a co-expression that is physically distinct from X.e and is reset to the beginning of the result sequence. Icon has no mechanism for producing a physically distinct copy of a co-expression with its state intact.

### 4. Code Generated by the Translator

The translator converts Seque programs to standard Icon programs. The specific tasks of the translator are:

- Insert a link declaration so that the Seque run-time library is automatically included when the translated program subsequently is run.
- Produce code to assign initial values for the built-in procedures and global identifiers that are used in the run-time library.

- Translate Seque expressions that have special syntax into standard Icon syntax.
- Convert recurrence declarations into standard Icon procedure declarations.

Function calls, such as Cat(X, Y) and Empty(X), are passed through unchanged by the translator.

The insertion of the link declaration and the generation of code to initialize global identifiers is done when the main procedure is encountered by the translator. The declaration

```
procedure main()
  body
end
```

is translated into

```
link "/usr/icon/ibin/seqlibe"

procedure main()
   Undef_ := Undef()
   X_ := []
   Iplus := Sequence([], create seq(1))
   Izero := Sequence([], create seq(0))
   Phi := Sequence([], create &fail)
   body
end
```

The identifiers Undef_ and X_ are used in the run-time library; see Sec. 6.2.

The code generated for Seque expressions depends on the type of the expression, as follows:

*Built-In Sequences*:

In order to avoid catastrophic effects that might result from changing the value of Phi, Izero, or Iplus during program execution, the explicit dereferencing operation is prepended to references to them. For example, Phi is translated into .Phi.

*Explicit Sequences*:

The expression

seq{$expr_1$, $expr_2$, ..., $expr_n$}

is translated into

Sequence([],create ($expr_1$) \ 1 | ($expr_2$) \ 1 | ... | ($expr_n$) \ 1)

The expressions are limited to one result, since Seque evaluation applies in explicit sequences.

*Pre-Truncation*:

The expression

$expr_1$ %% $expr_2$

is translated into the procedure call

Shift_(($expr_1$) \ 1, ($expr_2$) \ 1)

while

%*expr*

is translated into

Shift_((*expr*) \ 1, 1)

*Post-Truncation*:

The expression

$expr_1$ ^^ $expr_2$

is translated into

Lim_((*expr*$_1$) \ 1, (*expr*$_2$) \ 1)

*Element Selection*:

The expression

$expr_1$!$expr_2$

is translated into

Ref_((*expr*$_1$) \ 1, (*expr*$_2$) \ 1)

*Seque Generators*:

The expression

gen [ : *expr*$_1$ : lambda(j) *expr*$_2$ ]

is translated into

Sequence([], create (Pp_() & j := Gen_(*expr*$_1$) & 1(Gen_(*expr*$_2$), Qq_())))

The procedures Pp_ and Qq_ are used by the heuristic for terminating generation and are described in Sec. 6.2. Gen_ is a procedure in the run-time library that generates the results from the sequence X. Note that the outcome of activating the co-expression in a Seque generator is the outcome of Gen_(*expr*$_2$).

The bound variable j usually appears in $expr_2$, so the effect of the expression

j := Gen_(*expr*$_1$)

is to assign successive values from the generation sequence to the bound variable, which in turn controls the successive values generated from $expr_2$. The entire expression produces the results from Gen_(*expr*$_2$).

In case lambda(j) is omitted, i is provided by default. Similarly, Iplus is supplied in case $expr_1$ is omitted. Thus

gen[*expr*]

is translated into

Sequence([], create (Pp_() & i := Gen_(Iplus) & 1(Gen_(*expr*), Qq_())))

*Icon Generators*:

The expression gen{*expr*} is translated into

Sequence([], create *expr*)

*Operations in Seque Evaluation Contexts*:

In Seque evaluation contexts, operations and function calls are translated into calls on procedures in the run-time library. For example, the expression

$expr_1$ + $expr_2$

is translated into

Binop_("+", $expr_1$ \ 1, $expr_2$ \ 1)

and the expression

$expr(expr_1, expr_2)$

is translated into

P_([$expr$, $expr_1$, $expr_2$] \ 1)

See Sec. 6.3 and Appendix B for a description of Binop_ and P_.

It is an essential characteristic of such procedures that they produce at most one result. The explicit limitation to one result in the arguments prevents control structures, such as alternation, from introducing more results.

## 5. Recurrence Declarations

Recurrence declarations have the form

```
recur name ( generation variable ; [ parameters ] ; [ constant ] ; [ initial values ] )
   expr
end
```

A recurrence declaration is translated into an Icon procedure that is a generator for the values in the sequence for the recurrence. The procedure consists of three parts:

- a preamble
- expressions that generate the initial values
- an expression that generates the rest of the values

The generation variable is declared to be local and is assigned the initial value 0. The identifier m_, which is also local, is assigned a table whose default assigned value is the constant specified in the recurrence declaration. For example, the recurrence declaration

```
recur Fibs(j; ; ""; "a", "b")
   Fibs(j - 1) || Fibs(j - 2)
end
```

produces the preamble

```
procedure Fibs()
   local j, m_
   j := 0
   m_ := table("")
```

Each initial value, $expr_i$, is translated into an expression of the form

```
suspend m_[j +:= 1] := expr_i
```

where j is the generation variable. For example, the recurrence declaration given above produces

```
suspend m_[j +:= 1] := "a"
suspend m_[j +:= 1] := "b"
```

The translation of the expression given in the recurrence declaration into an expression to generate the rest of the values is more complicated. The general form is

```
suspend m_[j +:= 1] := expr'
```

where *expr'* is derived from *expr* by replacing calls to the recurrence by corresponding references to the table m_. Thus, for Fibs, the expression

```
Fibs(i - 1) || Fibs(i - 2)
```

produces

```
suspend m_[j +:= |1] := m_[j - 1] || m_[j - 2]
```

Note that if an entry value for m_ has not been previously computed, the constant value is produced.

The complete procedure for Fibs is

```
procedure Fibs()
   local j, m_
   j := 0
   m_ := table("")
   suspend m_[j +:= 1] := "a"
   suspend m_[j +:= 1] := "b"
   suspend m_[j +:= |1] := m_[j - 1] || m_[j - 2]
end
```

The situation is more complicated if a recurrence declaration has parameters. Such parameters become formal parameters of the corresponding Icon procedure and calls of this procedure provide values for the parameters in the recurrence. The complication occurs in the reference to the table m_, which in this case involves not only the value of the generation variable, but also the values of the parameters. This problem is handled by converting these values into a single value, using the procedure C_ in the run-time library. For example, the recurrence declaration

```
recur Gk(i; k; 0; )
   i - Gk(Gk(i - k, k), k)
end
```

is translated into

```
procedure Gk(k)
   local i, m_
   i := 0
   m_ := table(0)
   suspend m_[C_([i +:= |1, k])] := i - m_[C_([m_[C_([i - k, k])], k])]
end
```

The procedure C_ is

```
procedure C_(a)                          # identifying "subscript" for
   local s                               # recurrence lookup
   s := a[1]
   every s ||:= "." || image(a[2 to *a])
   return s
end
```

For example, C_([1,"a"]) produces the string

```
1."a"
```

The value produced by C_ is intended to be a unique representation of the values of the generation variable (which is an integer) and the parameters (which may be of any type). This procedure can be defeated by parameters whose values are structures. If this becomes a problem in practice, a more sophisticated technique may be needed.

## 6. The Run-Time Library

The run-time library is divided into three sections: basic operations, Seque evaluation, and user procedures. The basic operations and Seque evaluation procedures lie at the heart of Seque and ordinarily are not of interest to the Seque programmer.

### 6.1 Basic Operations

The basic operations are listed in Appendix A. A few of the procedures deserve special note and are described here.

In many situations it is necessary to coerce a value to a Sequence. This is done by coercing the value to a co-expression, which is then used in the construction of a sequence. The procedure used is

```
procedure Expr_(X)                          # return refreshed co-expression for X
   if type(X) == "Sequence" then return ^X.e
   else if /X then return Phi.e
   else return create X
end
```

If the value of X is a sequence, a refreshed copy of its co-expression is produced. If X is not a sequence, but is null-valued, the co-expression for Phi is produced. This corresponds to coercing the null value to the empty sequence. Note that it is not necessary to produce a refreshed copy of this co-expression, since it can never produce a value. For any other value, a corresponding co-expression is produced.

The pre- and post-truncation operations illustrate the use of co-expressions. Both of these operations construct sequences that contain co-expressions in which procedures control the production of results:

```
procedure Shift_(X, i)                      # X %% i
   local e
   e := Expr_(X)
   return Sequence([], create Pre_(e, i))
end

procedure Lim_(X, i)                        # X ^^ i
   local e
   e := Expr_(X)
   return Sequence([], create Post_(e, i))
end
```

The procedures Pre_ and Post_ do the actual truncation:

```
procedure Pre_(e, i)                        # skip first i values of X
   e := ^e
   every 1 to i
      do @e | fail
   suspend |@e
end

procedure Post_(e, i)                       # limit X to at most i values
   e := ^e
   suspend |@e \ i
end
```

### 6.2 The Termination Heuristic

A termination heuristic is used in Seque generators to prevent nontermination in situations like

```
gen[ |!i = 0 ]
```

where generation sequence is infinite and no value of I may be zero. In the absence of some termination

mechanism, such a generator would never produce a value and evaluation would continue within it endlessly. To avoid this possibility, a Seque generator terminates if an element selection expression, X!i, fails during evaluation of the generator.

As described in Sec. 4, the code produced for a Seque generator contains a co-expression within which the order of evaluation is

```
Pp_() & ... Qq_()
```

The procedures Pp_ and Qq_ serve to maintain a stack with respect to Seque generation. On this stack a state is maintained for each Seque generator, indicating whether or not an element selection operation in the generator has failed. The expression Pp_() is evaluated when generation begins:

```
procedure Pp_()                          # push/pop undefined marker
   push(X_, &null)
   suspend
   pop(X_)
   fail
end
```

X_ is a list that is created in the initialization code for the main procedure (see Sec. 4). Pushing the null value corresponds to establishing a new level of generation in which element selection has not failed. Pp_ then suspends, and the generation code is evaluated. Should the generation fail, Pp_ is resumed, the current level is popped, and Pp fails, transmitting the failure of the generation.

If the generation succeeds, the last expression evaluated is Qq_():

```
procedure Qq_()                          # pop/push undefined marker
   pop(X_)
   suspend
   push(X_, &null)
   fail
end
```

Qq_ is the inverse of Pp_: it pops the current state and suspends, transmitting the success of the generator. The desired result is produced by embedding Qq_ in

```
1(Gen_(expr2),Qq_())
```

so that the result is the result of Gen_($expr_2$).

The selection operator changes the current level of X_ to a unique nonnull value if it fails:

```
procedure Ref_(X, i)                     # X!i
   local x
   if i < 1 then {
      X_[1] := Undef_                    # termination heuristic
      fail
      }
   if not S_(X) then return \X | Phi
   if i > *X.a then
      every 1 to i - *X.a do
         put(X.a, @X.e) | {
            X_[1] := Undef_              # termination heuristic
            fail
            }
   return .X.a[i]
end
```

Gen_, in turn, checks the top of stack for this nonnull value:

```
procedure Gen_(X)                          # generate elements of X
   local i, x
   if X_[1] === Undef_ then fail           # termination heuristic
   if not S_(X) then return \X             # fails if X is null-valued
   every i := seq() do {
      if x := X.a[i] then                  # produce stored values first
         suspend x
      else {                               # transfer remaining values
         put(X.a, @X.e) | fail
         if x := X.a[i] then suspend x else fail
         }
      if X_[1] === Undef_ then fail        # termination heuristic
      }
end
```

Note that any nonnull value would suffice for these tests. Undef_, which is initialized in the main procedure, is used to allow possible elaboration of the heuristic.

### 6.3 Seque Evaluation

In Seque evaluation contexts, all operations are translated into procedures in the run-time library that serve to simulate the operation. These procedures have two purposes: the production of sequences if at least one argument is a sequence, and the limitation of the operation to at most one result. A typical procedure is

```
procedure Binop_(op, X1, X2)               # op(X1, X2)
   local e1, e2
   if not S_(X1 | X2) then return op(X1, X2)
   else {
      e1 := Expr_(X1)
      e2 := Expr_(X2)
      return Sequence([], create |op(@e1, @e2))
      }
end
```

Note that if neither argument is a sequence, the operation is performed using

```
op(X1, X2)
```

in which the value of op is a string representing the binary operator (see Sec. 4). The string invocation facility of the experimental extensions to Icon [4] invokes the corresponding binary operation, and the value is returned (suspension would allow the operation to produce more than one result). If either argument is a sequence, a sequence is produced in which the operation is embedded in a co-expression.

Function calls are handled by

```
procedure P_(a)                            # limited evaluation
   return case *a of {
      2 :  Unop_(a[1], a[2])
      3 :  Binop_(a[1], a[2], a[3])
      4 :  Triop_(a[1], a[2], a[3], a[4])
      5 :  Quadop_(a[1], a[2], a[3], a[4], a[5])
      default :  stop("Too many arguments in parallel evaluation")
      }
end
```

Unop_, Triop_, and Quadop_ are similar in structure to Binop_. See Appendix B. Because of the way that the Icon translator operates, function calls always have at least one (possibly null-valued) argument, so the list a always has at least two elements. Function calls with more than four arguments are not supported in Seque evaluation contexts; to do so would only require additional procedures.

### 6.4 User Procedures

User procedures are generally straightforward, relying on the basic operations and the use of co-expressions in the manner already described. Cat(X, Y) is typical:

```
procedure Cat(X1, X2)                    # concatenation of X1 and X2
   local e1, e2
   e1 := Expr_(X1)
   e2 := Expr_(X2)
   return Sequence([], create |@e1 | |@e2)
end
```

Note that alternation is used to produce the concatenation of result sequences.

The care that must be taken in handling empty sequences is illustrated by Empty(X):

```
procedure Empty(X)                       # is X an empty sequence?
   if /X then return Phi
   if not(S_(X)) | (*X.a > 0) | put(X.a, @X.e) then fail
   else return Phi
end
```

The reduction of a sequence over an operation illustrates other aspects of manipulating sequences:

```
procedure Red(X, op)                     # reduction of X over op
   local x, y, i
   x := Ref_(X, 1) | fail
   i := 1
   while y := Ref_(X, i +:= 1) do
      x := op(x, y)                      # op(x, y) may fail, not changing x
   X_[1] := &null                        # undo spurious heuristic
   return x
end
```

Note that Red removes the spurious setting of the termination heuristic that inevitably results from the use of element selection.

## 7. Translator Details

The translation of a Seque program into an Icon program is accomplished by a variant of the source-to-source Icon translator described in Reference 5. The specification of this variant translator consists of four parts:

- modifications to the lexical analyzer
- modifications to the Yacc grammar
- macro definitions for semantic actions in the parser
- functions for semantic actions in the parser

The modifications to the lexical analyzer consist of two new operators, ^^ and %%, and five new reserved words, gen, icon, lambda, recur, and seq. These modifications are straightforward; see Reference 5 for details.

The syntax of Seque is essentially upward compatible with that of Icon. The additions to the Yacc grammar are extensive, however, because of the number of details involved. Appendix D lists the grammar for the Seque translator, with changes to the Icon grammar identified by comments of the form /* Seque */ in the right margin.

Aside from generating code for Seque constructions, the major additions relate to determining Seque and Icon evaluation contexts and producing appropriate translations accordingly. The context switch tswitch is set to SEQUEV or ICONV, depending on the evaluation context. This switch is kept on a stack, since evaluation contexts can be nested.

Within recurrence relations, rswitch is set to LOOKUP, indicating that calls on the recurrence name, recname, are to be translated into table references. The argument to a table reference depends on whether or not the recurrence has parameters (see Sec. 5) and is determined by the value of xargs.

The specification of code generated by the parser is split between macro definitions (for simple code) and parser functions (for more complex code). See Appendices E and F.

**Acknowledgements**

Dave Hanson, Bill Mitchell, and Steve Wampler provided a number of helpful suggestions concerning the presentation of the material in this report.

**References**

1. Griswold, Ralph E. *Seque: An Experimental Language for Manipulating Sequences*, Technical Report TR 83-16, Department of Computer Science, The University of Arizona. 1983.

2. Griswold, Ralph E. and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1983.

3. Wampler, Stephen B. and Ralph E. Griswold. "Co-Expressions in Icon", *The Computer Journal*, Vol. 26, No. 1 (February 1983). pp. 72-78.

4. Griswold, Ralph E. and William H. Mitchell. *Experimental Extensions to Version 5.8 of Icon*, technical report, Department of Computer Science, The University of Arizona. 1983.

5. Griswold, Ralph E. *The Construction of Variant Translators for Icon*, Technical Report TR 83-19, Department of Computer Science, The University of Arizona. 1983.

## Appendix A — Run-Time Library; Basic Operations

```
link "/usr/icon/ibin/uops"                    # user operations
link "/usr/icon/ibin/call"                    # Seque evaluation

record Sequence(a, e)                         # sequence data type
record Undef()                                # for unique "undefined" value

global Phi, Iplus, Izero, Undef_, X_

procedure C_(a)                               # identifying "subscript" for
   local s                                    # recurrence lookup
   s := a[1]
   every s ||:= "." || image(a[2 to *a])
   return s
end

procedure Collapse_(X)                        # generate scalars from X
   if S_(X) then
      suspend Collapse_(Gen_(X))
  else return X
end

procedure Expr_(X)                            # return refreshed co-expression for X
   if type(X) == "Sequence" then return ^X.e
   else if /X then return Phi.e
   else return create X
end

procedure Gen_(X)                             # generate elements of X
   local i, x
   if X_[1] === Undef_ then fail              # termination heuristic
   if not S_(X) then return \X                # fails if X is null-valued
   every i := seq() do {
      if x := X.a[i] then                     # produce stored values first
         suspend x
      else {                                  # transfer remaining values
         put(X.a, @X.e) | fail
         if x := X.a[i] then suspend x else fail
         }
      if X_[1] === Undef_ then fail           # termination heuristic
      }
end

procedure Generic_(p, X)                      # apply p to X
   if /X then fail
   return if S_(X) then every p(Gen_(X)) else p(X)
end

procedure Lim_(X, i)                          # X ^^ i
   local e
   e := Expr_(X)
   return Sequence([], create Post_(e, i))
end

procedure Post_(e, i)                         # limit X to at most i values
   e := ^e
   suspend |@e \ i
end
```

```
procedure Pp_()                          # push/pop undefined marker
   push(X_, &null)
   suspend
   pop(X_)
   fail
end

procedure Pre_(e, i)                     # skip first i values of X
   e := ^e
   every 1 to i
      do @e | fail
   suspend |@e
end

procedure Qq_()                          # pop/push undefined marker
   pop(X_)
   suspend
   push(X_, &null)
   fail
end

procedure Ref_(X, i)                     # X!i
   local x
   if i < 1 then {
      X_[1] := Undef_                    # termination heuristic
      fail
      }
   if not S_(X) then return \X | Phi
   if i > *X.a then
      every 1 to i - *X.a do
         put(X.a, @X.e) | {
            X_[1] := Undef_              # termination heuristic
            fail
            }
   return .X.a[i]
end

procedure S_(x)                          # is X a sequence?
   return type(x) == "Sequence"
end

procedure Shift_(X, i)                   # X %% i
   local e
   e := Expr_(X)
   return Sequence([], create Pre_(e, i))
end
```

## Appendix B — Run-Time Library; Seque Evaluation

```
procedure Unop_(op, X)                          # op(X)
   local e
   if S_(X) then {
      e := Expr_(X)
      return Sequence([], create op(|@e))
      }
   else return op(X)
end

procedure Binop_(op, X1, X2)                    # op(X1, X2)
   local e1, e2
   if not S_(X1 | X2) then return op(X1, X2)
   else {
      e1 := Expr_(X1)
      e2 := Expr_(X2)
      return Sequence([], create |op(@e1, @e2))
      }
end

procedure Triop_(op, X1, X2, X3)                # op(X1, X2, X2)
   local e1, e2, e3
   if not S_(X1 | X2 | X3) then return op(X1, X2, X3)
   else {
      e1 := Expr_(X1)
      e2 := Expr_(X2)
      e3 := Expr_(X3)
      return Sequence([], create |op(@e1, @e2, @e3))
      }
end

procedure Quadop_(op, X1, X2, X3, X4)           # op(X1, X2, X3, X4)
   local e1, e2, e3, s4
   if not S_(X1 | X2 | X3 | X4) then return op(X1, X2, X3, X4)
   else {
     e1 := Expr_(X1)
     e2 := Expr_(X2)
     e3 := Expr_(X3)
     e4 := Expr_(X4)
     return Sequence([], create |op(@e1, @e2, @e3, @e4))
     }
end

procedure P_(a)                                 # limited evaluation
   return case *a of {
      2 :  Unop_(a[1], a[2])
      3 :  Binop_(a[1], a[2], a[3])
      4 :  Triop_(a[1], a[2], a[3], a[4])
      5 :  Quadop_(a[1], a[2], a[3], a[4], a[5])
      default :  stop("Too many arguments in parallel evaluation")
      }
end
```

## Appendix C — Run-Time Library; User Procedures

```
procedure Cat(X1, X2)                          # concatenation of X1 and X2
   local e1, e2
   e1 := Expr_(X1)
   e2 := Expr_(X2)
   return Sequence([], create |@e1 | |@e2)
end

procedure Compress(X)                          # compression of X to scalar sequence
   return Sequence([], create Collapse_(Copy(X)))
end

procedure Copy(X)                              # copy X
   local e
   e := Expr_(X)
   return Sequence([], e)
end

procedure Empty(X)                             # is X an empty sequence?
   if /X then return Phi
   if not(S_(X)) | (*X.a > 0) | put(X.a, @X.e) then fail
   else return Phi
end

procedure Image(X, i)                          # image of X to i values
   local s, t, j
   if S_(X) then {
      /i := 5
      j := 0
      s := "{"
      every t := (Gen_(X) \ i) do {
         s ||:= Image(t, i) || ", "
         j +:= 1
         }
      if j = 0 then return "{}"
      if Ref_(X, i + 1) then s ||:= "..., "
      else X_[1] := &null                      # undo spurious heuristic
      s[-1] := "}"
      return s
      }
   else return image(X)
end

procedure Length(X)                            # length of X
   while put(X.a, @X.e)
   return *X.a
end

procedure Read(f)                              # sequence from file f
   return Sequence([], create |read(f))
end
```

```
procedure Red(X, op)                                    # reduction of X over op
   local x, y, i
   x := Ref_(X, 1) | fail
   i := 1
   while y := Ref_(X, i +:= 1) do
      x := op(x, y)                                     # op(x, y) may fail, not changing x
   X_[1] := &null                                       # undo spurious heuristic
   return x
end

procedure Subseq(X, i, j)                               # subsequence of X from i to j
   local e
   e := Expr_(X)
  return Sequence([], create Post_(create Pre_(e, i - 1), j - i + 1))
end

procedure Trace(X, i)                                   # image of X, returning X
   write(Image(X, i))
   return X
end

procedure Write(X)                                      # write elements in X with linefeeds
   return Generic_(write, X)
end

procedure Writes(X)                                     # write elements in X without linefeeds
   return Generic_(writes, X)
end
```

## Appendix D — Seque Grammar

```
/*          Seque Translator          */                          /* Seque */

/* primitive tokens */

%token      CSETLIT EOFX IDENT INTLIT REALLIT STRINGLIT

/* reserved words */

%token      BREAK BY CASE CREATE DEFAULT DO DYNAMIC ELSE END EVERY EXTERNAL
            FAIL GLOBAL IF INITIAL LINK LOCAL NEXT NOT OF PROCEDURE RECORD REPEAT
            RETURN STATIC SUSPEND THEN TO UNTIL WHILE

/* new Seque reserved words */                                     /* Seque */

%token      GEN ICON LAMBDA RECUR SEQ                              /* Seque */

/* operators */

%token      ASSIGN AT AUGACT AUGAND AUGEQ AUGEQV AUGGE AUGGT AUGLE AUGLT
            AUGNE AUGNEQV AUGSEQ AUGSGE AUGSGT AUGSLE AUGSLT AUGSNE
            BACKSLASH BANG BAR CARET CARETASGN COLON COMMA CONCAT
            CONCATASGN CONJUNC DIFF DIFFASGN DOT EQUIV INTER INTERASGN LBRACE
            LBRACK LCONCAT LCONCATASGN LEXEQ LEXGE LEXGT LEXLE LEXLT LEXNE
            LPAREN MCOLON MINUS MINUSASGN MOD MODASGN NOTEQUIV NUMEQ NUMGE
            NUMGT NUMLE NUMLT NUMNE PCOLON PLUS PLUSASGN QMARK RBRACE RBRACK
            REVASSIGN REVSWAP RPAREN SCANASGN SEMICOL SLASH SLASHASGN STAR
            STARASGN SWAP TILDE UNION UNIONASGN

/* new operators */                                               /* Seque */

%token      CARETCARET MODMOD                                      /* Seque */

%{
#define q(x) strcpy(malloc(strlen(x)+1), x)
#define Str(x) q(STR0(x))

extern int tswitch;                 /* evaluation context switch */   /* Seque */
extern int rswitch;                 /* recurrence context switch */   /* Seque */
extern char *xargs;                 /* recurrence parameters */       /* Seque */
extern char *recname;               /* recurrence name */             /* Seque */
extern char *argname;               /* recurrence variable */         /* Seque */

#include "tdefs.h"
#include "itran.h"
#include "sym.h"
#include "tree.h"
#define YYSTYPE nodeptr
#define YYMAXDEPTH 500
#define SEQUEV 0                    /* Seque evaluation */            /* Seque */
#define ICONV 1                     /* Icon evaluation */             /* Seque */
#define LOOKUP 1                    /* recurrence table look up */    /* Seque */
#define NOLOOKUP 0                  /* no recurrence table look up */ /* Seque */
%}

%%
```

```
%{
int tswitch = SEQUEV;                                                    /* Seque */
int rswitch = NOLOOKUP;                                                  /* Seque */
int tstack[50];              /* Seque/Icon context stack */              /* Seque */
int stacktop = 0;                                                        /* Seque */
char *recname;                                                           /* Seque */
char *argname;                                                           /* Seque */
char *xargs;                                                             /* Seque */
%}

program     : decls EOFX ;

decls       : {$$ = Null;} ;
            | decls decl ;

decl        : record ;
            | proc ;
            | recur ;                                                    /* Seque */
            | global ;
            | link ;

link        : LINK lnklist {
                        Link($2);
                        free($2);
                        loc_init();
                        } ;

lnklist     : lnkfile ;
            | lnklist COMMA lnkfile {$$ = Linklist($1, $3);} ;

lnkfile     : IDENT {$$ = Linkident($1);} ;
            | STRINGLIT {$$ = Linkstring($1);} ;

global      : GLOBAL idlist {
                        Global($2);
                        free($2);
                        } ;
            | EXTERNAL idlist {
                        External($2);
                        free($2);
                        } ;

record      : RECORD IDENT LPAREN arglist RPAREN {
                        Record($2, $4);
                        free($4);
                        loc_init();
                        } ;

proc        : prochead SEMICOL locals initial procbody END {
                        Proc($1, $3, $4, $5);
                        free($1);
                        free($3);
                        free($4);
                        free($5);
                        treeinit();
                        loc_init();
                        } ;

prochead    : ptype IDENT LPAREN arglist RPAREN {$$ = Prochead($2, $4);} ;   /* Seque */

ptype       : PROCEDURE {tswitch = SEQUEV;} ;                            /* Seque */
            | ICON PROCEDURE {tswitch = ICONV;} ;                        /* Seque */
```

```
arglist     : {$$ = Null;} ;
            | idlist ;

idlist      : IDENT {$$ = Ident($1);} ;
            | idlist COMMA IDENT {$$ = Idlist($1, $3);} ;

locals      : {$$ = Null;} ;
            | locals retention idlist SEMICOL {$$ = Locals($1, $2, $3);} ;

retention   : LOCAL {$$ = Local;} ;
            | STATIC {$$ = Static;} ;
            | DYNAMIC {$$ = Dynamic;} ;

initial     : {$$ = Null;} ;
            | INITIAL expr SEMICOL {$$ = Initial($2);} ;

procbody    : {$$ = Null;} ;
            | nexpr SEMICOL procbody {$$ = Procbody($1, $3);} ;

recur       : recurh LPAREN rarg xarglist ispec SEMICOL expr SEMICOL END {      /* Seque */
                        Recur($1, $3, $4, $5, $7);                             /* Seque */
                        free($4);       /* free allocated strings */           /* Seque */
                        free($5);                                              /* Seque */
                        free($7);                                              /* Seque */
                        treeinit();                                            /* Seque */
                        loc_init();                                            /* Seque */
                        tswitch = SEQUEV;                                      /* Seque */
                        rswitch = NOLOOKUP;                                    /* Seque */
                        recname = "";                                          /* Seque */
                        argname = "";                                          /* Seque */
                        xargs = "";                                            /* Seque */
                        } ;                                                    /* Seque */

rarg        : IDENT SEMICOL {                                                  /* Seque */
                        argname = Str($1);                                     /* Seque */
                        $$ = $1;                                               /* Seque */
                        } ;                                                    /* Seque */
recurh      : RECUR IDENT {                                                    /* Seque */
                        tswitch = ICONV;                                       /* Seque */
                        rswitch = LOOKUP;                                      /* Seque */
                        recname = Str($2);                                     /* Seque */
                        $$ = $2;                                               /* Seque */
                        } ;                                                    /* Seque */

xarglist    : SEMICOL {$$ = Null;} ;                                           /* Seque */
            | idlist SEMICOL {                                                 /* Seque */
                        xargs = $1;                                            /* Seque */
                        $$ = $1;                                               /* Seque */
                        } ;                                                    /* Seque */

ispec       : nexpr SEMICOL slist RPAREN {$$ = Ispec($1, $3);} ;               /* Seque */

slist       : {$$ = Null;} ;                                                   /* Seque */
            | rinit ;                                                          /* Seque */
            | rinit COMMA slist {$$ = Slist($1, $3);} ;                        /* Seque */

rinit       : expr {$$ = Rinit($1);} ;                                         /* Seque */

nexpr       : {$$ = Null;} ;
            | expr ;

expr        : expr1a ;
            | expr CONJUNC expr1a {$$ = Bamper($1, $3);} ;
```

```
expr1a      : expr1 ;
            | expr1a QMARK expr1 {$$ = Bques($1, $3);} ;

expr1       : expr2 ;
            | expr2 SWAP expr1 {$$ = Bswap($1, $3);} ;
            | expr2 ASSIGN expr1 {$$ = Bassgn($1, $3);} ;
            | expr2 REVSWAP expr1 {$$ = Brswap($1, $3);} ;
            | expr2 REVASSIGN expr1 {$$ = Brassgn($1, $3);} ;
            | expr2 DIFFASGN expr1 {$$ = Bdiffa($1, $3);} ;
            | expr2 UNIONASGN expr1 {$$ = Buniona($1, $3);} ;
            | expr2 PLUSASGN expr1 {$$ = Bplusa($1, $3);} ;
            | expr2 MINUSASGN expr1 {$$ = Bminusa($1, $3);} ;
            | expr2 STARASGN expr1 {$$ = Bstara($1, $3);} ;
            | expr2 INTERASGN expr1 {$$ = Bintera($1, $3);} ;
            | expr2 SLASHASGN expr1 {$$ = Bslasha($1, $3);} ;
            | expr2 MODASGN expr1 {$$ = Bmoda($1, $3);} ;
            | expr2 CARETASGN expr1 {$$ = Bcareta($1, $3);} ;
            | expr2 AUGEQ expr1 {$$ = Bauqeq($1, $3);} ;
            | expr2 AUGEQV expr1 {$$ = Baugeqv($1, $3);} ;
            | expr2 AUGGE expr1 {$$ = Baugge($1, $3);} ;
            | expr2 AUGGT expr1 {$$ = Bauggt($1, $3);} ;
            | expr2 AUGLE expr1 {$$ = Baugle($1, $3);} ;
            | expr2 AUGLT expr1 {$$ = Bauglt($1, $3);} ;
            | expr2 AUGNE expr1 {$$ = Baugne($1, $3);} ;
            | expr2 AUGNEQV expr1 {$$ = Baugneqv($1, $3);} ;
            | expr2 AUGSEQ expr1 {$$ = Baugseq($1, $3);} ;
            | expr2 AUGSGE expr1 {$$ = Baugsge($1, $3);} ;
            | expr2 AUGSGT expr1 {$$ = Baugsgt($1, $3);} ;
            | expr2 AUGSLE expr1 {$$ = Baugsle($1, $3);} ;
            | expr2 AUGSLT expr1 {$$ = Baugslt($1, $3);} ;
            | expr2 AUGSNE expr1 {$$ = Baugsne($1, $3);} ;
            | expr2 CONCATASGN expr1 {$$ = Baugcat($1, $3);} ;
            | expr2 LCONCATASGN expr1 {$$ = Bauglcat($1, $3);} ;
            | expr2 SCANASGN expr1 {$$ = Baugques($1, $3);} ;
            | expr2 AUGAND expr1 {$$ = Baugamper($1, $3);} ;
            | expr2 AUGACT expr1 {$$ = Baugact($1, $3);} ;

expr2       : expr3 ;
            | expr2 TO expr3 {$$ = To2($1, $3);} ;
            | expr2 TO expr3 BY expr3 {$$ = To3($1, $3, $5);} ;

expr3       : expr4 ;
            | expr4 BAR expr3 {$$ = Alt($1, $3);} ;

expr4       : expr5 ;
            | expr4 LEXEQ expr5 {$$ = Bseq($1, $3);} ;
            | expr4 LEXGE expr5 {$$ = Bsge($1, $3);} ;
            | expr4 LEXGT expr5 {$$ = Bsgt($1, $3);} ;
            | expr4 LEXLE expr5 {$$ = Bsle($1, $3);} ;
            | expr4 LEXLT expr5 {$$ = Bslt($1, $3);} ;
            | expr4 LEXNE expr5 {$$ = Bsne($1, $3);} ;
            | expr4 NUMEQ expr5 {$$ = Beq($1, $3);} ;
            | expr4 NUMGE expr5 {$$ = Bge($1, $3);} ;
            | expr4 NUMGT expr5 {$$ = Bgt($1, $3);} ;
            | expr4 NUMLE expr5 {$$ = Ble($1, $3);} ;
            | expr4 NUMLT expr5 {$$ = Blt($1, $3);} ;
            | expr4 NUMNE expr5 {$$ = Bne($1, $3);} ;
            | expr4 EQUIV expr5 {$$ = Beqv($1, $3);} ;
            | expr4 NOTEQUIV expr5 {$$ = Bneqv($1, $3);} ;

expr5       : expr6 ;
            | expr5 CONCAT expr6 {$$ = Bcat($1, $3);} ;
            | expr5 LCONCAT expr6 {$$ = Blcat($1, $3);} ;
```

```
expr6       : expr7 ;
            | expr6 PLUS expr7 {$$ = Bplus($1, $3);} ;
            | expr6 DIFF expr7 {$$ = Bdiff($1, $3);} ;
            | expr6 UNION expr7 {$$ = Bunion($1, $3);} ;
            | expr6 MINUS expr7 {$$ = Bminus($1, $3);} ;

expr7       : expr8 ;
            | expr7 STAR expr8 {$$ = Bstar($1, $3);} ;
            | expr7 INTER expr8 {$$ = Binter($1, $3);} ;
            | expr7 SLASH expr8 {$$ = Bslash($1, $3);} ;
            | expr7 MOD expr8 {$$ = Bmod($1, $3);} ;
            | expr7 MODMOD expr8 {$$ = Shift2($1, $3);} ;                    /* Seque */

expr8       : expr9 ;
            | expr9 CARET expr8 {$$ = Bcaret($1, $3);} ;
            | expr9 CARETCARET expr8 {$$ = Limit($1, $3);} ;                 /* Seque */

expr9       : expr10 ;
            | expr9 BACKSLASH expr10 {$$ = Blim($1, $3);} ;
            | expr9 AT expr10 {$$ = Bact($1, $3);} ;

expr10      : expr10a ;                                                      /* Seque */
            | AT expr10 {$$ = Uat($2);} ;
            | MOD expr10 {$$ = Shift1($2);} ;                                /* Seque */
            | MODMOD expr10 {$$ = Shift1(Shift1($2));} ;                     /* Seque */
            | NOT expr10 {$$ = Not($2);} ;
            | BAR expr10 {$$ = Ubar($2);} ;
            | CONCAT expr10 {$$ = Ubar(Ubar($2));} ;
            | LCONCAT expr10 {$$ = Ubar(Ubar(Ubar($2)));} ;
            | DOT expr10 {$$ = Udot($2);} ;
            | BANG expr10 {$$ = Ubang($2);} ;
            | DIFF expr10 {$$ = Uminus(Uminus($2));} ;
            | PLUS expr10 {$$ = Uplus($2);} ;
            | STAR expr10 {$$ = Ustar($2);} ;
            | SLASH expr10 {$$ = Uslash($2);} ;
            | CARET expr10 {$$ = Ucaret($2);} ;
            | CARETCARET expr10 {$$ = Ucaret(Ucaret($2));} ;                 /* Seque */
            | INTER expr10 {$$ = Ustar(Ustar($2));} ;
            | TILDE expr10 {$$ = Utilde($2);} ;
            | MINUS expr10 {$$ = Uminus($2);} ;
            | NUMEQ expr10 {$$ = Ueq($2);} ;
            | NUMNE expr10 {$$ = Utilde(Ueq($2));} ;
            | LEXEQ expr10 {$$ = Ueq(Ueq($2));} ;
            | LEXNE expr10 {$$ = Utilde(Ueq(Ueq($2)));} ;
            | EQUIV expr10 {$$ = Ueq(Ueq(Ueq($2)));} ;
            | UNION expr10 {$$ = Uplus(Uplus($2));} ;
            | QMARK expr10 {$$ = Uques($2);} ;
            | NOTEQUIV expr10 {$$ = Utilde(Ueq(Ueq(Ueq($2))));} ;
            | BACKSLASH expr10 {$$ = Ubacksl($2);} ;

expr10a     : expr11                                                         /* Seque */
            | expr10a BANG expr11 {$$ = Refer($1, $3);} ;                    /* Seque */
```

```
expr11      : literal ;
            | section ;
            | return ;
            | if ;
            | case ;
            | while ;
            | until ;
            | every ;
            | repeat ;
            | CREATE expr {$$ = Create($2);} ;
            | IDENT {$$ = Idmap($1);} ;                                  /* Seque */
            | NEXT {$$ = Next;} ;
            | BREAK nexpr {$$ = Break($2);} ;
            | LPAREN exprlist RPAREN {$$ = Paren($2);} ;
            | SEQ LBRACE RBRACE {$$ = Phi;} ;                            /* Seque */
            | seqhead seqlist RBRACE {                                   /* Seque */
                        $$ = Seq($2);                                    /* Seque */
                        tswitch = pops();                                /* Seque */
                        rswitch = pops();                                /* Seque */
                        } ;                                              /* Seque */
            | LBRACE compound RBRACE {$$ = Brace($2);} ;
            | LBRACK exprlist RBRACK {$$ = Bracket($2);} ;
            | genh1 expr RBRACE {                                        /* Seque */
                        $$ = Seq($2);                                    /* Seque */
                        tswitch = pops();                                /* Seque */
                        } ;                                              /* Seque */
            | genh2 xseq lambda expr RBRACK {                            /* Seque */
                        $$ = Gener($2, $3, $4);                          /* Seque */
                        tswitch = pops();                                /* Seque */
                        rswitch = pops();                                /* Seque */
                        } ;                                              /* Seque */
            | expr11 LBRACK expr RBRACK {$$ = Subscr($1, $3);} ;
            | expr11 LBRACE exprlist RBRACE {$$ = Pdco($1, $3);} ;
            | expr11 LPAREN exprlist RPAREN {$$ = Invoke($1, $3);} ;
            | expr11 DOT IDENT {$$ = Field($1, $3);} ;
            | CONJUNC FAIL {$$ = Kfail;} ;
            | CONJUNC IDENT {$$ = Keyword($2);} ;

while       : WHILE expr {$$ = While1($2);} ;
            | WHILE expr DO expr {$$ = While2($2, $4);} ;

until       : UNTIL expr {$$ = Until1($2);} ;
            | UNTIL expr DO expr {$$ = Until2($2, $4);} ;

every       : EVERY expr {$$ = Every1($2);} ;
            | EVERY expr DO expr {$$ = Every2($2, $4);} ;

repeat      : REPEAT expr {$$ = Repeat($2);} ;

return      : FAIL {$$ = Fail;} ;
            | RETURN nexpr {$$ = Return($2);} ;
            | SUSPEND nexpr {$$ = Suspend($2);} ;

if          : IF expr THEN expr {$$ = If2($2, $4);} ;
            | IF expr THEN expr ELSE expr {$$ = If3($2, $4, $6);} ;

case        : CASE expr OF LBRACE caselist RBRACE {$$ = Case($2, $5);} ;

caselist    : cclause ;
            | caselist SEMICOL cclause {$$ = Clist($1, $3);} ;

cclause     : DEFAULT COLON expr {$$ = Default($3);} ;
            | expr COLON expr {$$ = Cclause($1, $3);} ;
```

```
exprlist        : nexpr {Null;} ;
                | exprlist COMMA nexpr {$$ = Exprlist($1, $3);} ;

literal         : INTLIT {$$ = Iliter($1);} ;
                | REALLIT {$$ = Rliter($1);} ;
                | STRINGLIT {$$ = Sliter($1);} ;
                | CSETLIT {$$ = Cliter($1);} ;

section         : expr11 LBRACK expr COLON expr RBRACK {$$ = Sect($1, $3, $5);} ;
                | expr11 LBRACK expr PCOLON expr RBRACK {$$ = Psect($1, $3, $5);} ;
                | expr11 LBRACK expr MCOLON expr RBRACK {$$ = Msect($1, $3, $5);} ;

compound        : nexpr ;
                | nexpr SEMICOL compound {$$ = Semi($1, $3);} ;

seqhead         : SEQ LBRACE {                                          /* Seque */
                        pushs(rswitch);                                 /* Seque */
                        pushs(tswitch);                                 /* Seque */
                        tswitch = SEQUEV;                               /* Seque */
                        rswitch = NOLOOKUP;                             /* Seque */
                        } ;                                             /* Seque */

snexpr          : {$$ = Nullkey;} ;                                     /* Seque */
                | expr {$$ = Lone($1);} ;                               /* Seque */

seqlist         : snexpr;                                               /* Seque */
                | seqlist COMMA snexpr {$$ = Alt($1, $3);} ;            /* Seque */

genh1           : GEN LBRACE {                                          /* Seque */
                        pushs(tswitch);                                 /* Seque */
                        tswitch = ICONV;                                /* Seque */
                        } ;                                             /* Seque */

genh2           : GEN LBRACK {                                          /* Seque */
                        pushs(rswitch);                                 /* Seque */
                        pushs(tswitch);                                 /* Seque */
                        tswitch = SEQUEV;                               /* Seque */
                        rswitch = NOLOOKUP;                             /* Seque */
                        } ;                                             /* Seque */

xseq            : {$$ = Defseq;} ;                                      /* Seque */
                | COLON expr COLON {$$ = $2;} ;                         /* Seque */

lambda          : {$$ = Defvar;} ;                                      /* Seque */
                | LAMBDA LPAREN IDENT RPAREN {$$ = Ident($3);} ;        /* Seque */

program         : error decls EOFX ;
proc            : prochead error procbody END ;
expr            : error ;
%%
#include "cater.c"              /* string concatenation */
#include "ulibe.c"              /* Seque parser functions */            /* Seque */
```

## Appendix E — Specifications for Translator Macros

```
#               remove macro definitions in favor of functions
#
Proc(x, y, z, w)
Invoke(x, y)
#
#               new macro definitions for Seque
#
Defseq          "Iplus"
Defvar          "i"
Ispec(x, y)     "m_ := table("           x          ")\n"              y
Limit(x, y)     "Lim_((("               x          ") \\ 1, "         y          ")"
Lone(x)         "("                     x          ") \\ 1"
Nullkey         "&null"
Phi             ".Phi"
Refer(x, y)     "Ref_(("                x          ") \\ 1, "         y          ")"
Rinit(x)        "suspend m_["           Vrep()     "] := "            x
Seq(x)          "Sequence([], create "  x          ")"
Shift1(x)       "Shift_(("              x          ") \\ 1, 1)"
Shift2(x, y)    "Shift_(("              x          ") \\ 1, "         y          ")"
Slist(x, y)     x                       "\n"       y
<bop>(x, y)     Binop(x, <bop>, y)
<uop>x          Unop(<uop>, x)
```

## Appendix F — Parser Functions

```
char *Binop(x, y, z)                    /* translate binary operation */
   char *x, *y, *z;
   {
   if (tswitch == SEQUEV)
      return cat(q("Binop_(\""), y, q("\", ("), x, q(") \\ 1, ("), z, q(") \\ 1)"));
   else return cat(x, q(" "), y, q(" "), z);
   }

char *Gener(x, y, z)                    /* translate Seque generator */
   char *x, *y, *z;
   {
   return cat(q("Sequence([], create (Pp_() &\n"), y, q(" := Gen_("),
      x, q(") & 1(Gen_("), z, q("), Qq_()))"));
   }

char *Idmap(x)                          /* translate reference to identifier */
   {
   char *s;
   s = Str(x);
   if (strcmp(s, "Iplus") == 0) return q(".Iplus");
   else if (strcmp(s, "Izero") == 0) return q(".Izero");
   else if (strcmp(s, "Phi") == 0) return q(".Phi");
   else return s;
   }

char *Invoke(x, y)                      /* translate function call */
   char *x, *y;
   {
   if ((rswitch == LOOKUP) && (strcmp(x, recname) == 0) && (strcmp(xargs, "") == 0))
      return cat(q("m_["), y, q("]"));
   else if ((rswitch == LOOKUP) && (strcmp(x, recname) == 0) && (strcmp(xargs, "") != 0))
      return cat(q("m_[C_(["), y, q("])]"));
   else if ((tswitch == SEQUEV) && (x[0] >= 'a' && x[0] <= 'z'))
      return cat(q("P_(["), x, q(", "), y, q("] \\ 1)"));
   else return cat(x, q("("), y, q(")"));
   }

char *Proc(x, y, z, w)                  /* translate procedure declaration */
   char *x, *y, *z, *w;
   {
   if (strncmp(x, "procedure main(", 15) == 0) {
      printf("link \"/usr/icon/ibin/seqlibe\"\n\n");
      printf("%s;\n%s%s", x, y, z);
      printf("Undef_ := Undef()\n");
      printf("X_ := []\n");
      printf("Iplus := Sequence([], create seq(1))\n");
      printf("Izero := Sequence([], create seq(0))\n");
      printf("Phi := Sequence([], create &fail)\n");
      printf("%send\n", w);
      }
   else printf("%s;\n%s%s%send\n", x, y, z, w);
   }
```

```
char *Recur(x, y, z, u, v)                    /* translate recurrence declaration */
  char *x, *y, *z, *u, *v;
   {
   printf("procedure %s(%s)\n", x, z);
   printf("local %s, m_\n", y);
   printf("%s := 0\n", y);
   printf("%s\n", u);
   if (strcmp(z, "") == 0) printf("suspend m_[%s +:= |1] := %s\n", y, v);
   else printf("suspend m_[C_([%s +:= |1, %s])] := %s\n", y, z, v);
   printf("end\n");
   }

char *Unop(x, y)                              /* translate unary operation */
   char *x, *y;
   {
   if (tswitch == SEQUEV) return cat(q("Unop_(\""), x, q("\", ("), y, q(") \\ 1)"));
   else return cat(x, y);
   }

char *Vrep()                                  /* translate recurrence initialization */
   {
   if (strcmp(xargs, "") == 0) return cat(q(argname), q(" +:= 1"));
   else return cat(q("C_(["), q(argname), q(" +:= 1, "), q(xargs), q("])"));
   }

pushs(i)                                      /* push i on evaluation context stack */
int i;
   {
   tstack[++stacktop] = i;
   }

pops()                                        /* pop value from evaluation context stack */
   {
   int i;
   i = tstack[stacktop--];
   return i;
   }
```