Differences Between Versions 2 and 5 of Icon*

Ralph E. Griswold

TR 83-5

March 12, 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.

Differences Between Versions 2 and 5 of Icon

1. Introduction

The Icon programming language has evolved through several versions with somewhat different language features. The first two versions were implemented in Ratfor and were designed to be portable over a wide range of computers. Starting with Version 3 and continuing through the current Version 5, Icon has been implemented in C and tailored to run under the UNIX* operating. system. The Ratfor implementation remains at Version 2 and there are no plans to bring it up to date. Nonetheless, Version 2 is implemented on several operating systems and computers for which Version 5 is not available. Consequently, both Version 2 and Version 5 are in use. Versions 1, 3, and 4, on the other hand, are obsolete.

There is a reference manual for Version 2 [1] that describes the features of the language. It does not include program examples or discuss programming in Icon. The Icon book [2], which contains many program examples as well as chapters on programming techniques, describes Version 5. This report is a guide to that book for users of Version 2.

2. Overview of Version 2 and Version 5 Differences

There are quite a few differences between Versions 2 and 5. Some of these differences are only syntactic. Others have to do with the repertoire of operations. In some cases, there are substantial semantic differences, although it is possible to recast most Version 5 programs in Version 2.

The major semantic differences between Versions 2 and 5 are:

- The null value is treated quite differently in Versions 2 and 5.
- Version 2 does not have the mutual evaluation, repeated alternation, and limitation control structures of Version 5. A number of common control structures are treated somewhat differently in the two versions.
- Version 2 has a stack data type that does not exist in Version 5.
- The operations on lists in Versions 2 and 5 are considerably different.
- Version 2 does not have co-expressions.

The following descriptions of differences are keyed to the Icon book and pages numbers refer to that book. Differences between Versions 2 and 5 that affect programs significantly are marked by asterisks. The Appendix contains the procedures from the Icon book, rewritten for Version 2. Comparison of these procedures with those in the Icon book reveals the patterns of differences between the two versions more clearly than the detailed description that follows.

Because of the many minor differences between Versions 2 and 5, there are undoubtedly some omissions and errors in the following list. Please notify the author of any errors in this report.

3. List of Differences

The following list covers differences in the part of the Icon book that describes the language: Chapters 1-13 and the Appendices. No attempt is made here to provide changes to all places in the book where there are

*UNIX is a trademark of Bell Laboratories.

differences, generally only the first occurrence of a difference is noted

Chapter 1 — Getting Started

Page 1: Version 2 does not require parentheses in procedure declarations that have no parameters Thus

```
procedure main
```

is acceptable in Version 2

Page 3: Upper- and lowercase letters are equivalent in Version 2 For example, the identifiers label and Label are equivalent in Version 2

Chapter 2 — Control Structures

Page 12*: The **do** clauses in the **while** and **until** expressions are not optional in Version 2

Pages 12-13*: The prefix **not** expression in Version 5 is represented by the suffix **fails** expression in Version 2 For example, the Version 5 expression

```
if not expr1 then expr2
```

is written in Version 2 as

```
if expr1 fails then expr2
```

Page 13: The **break** and **next** expressions are not allowed in the control clauses of looping control structures in Version 2

Page 14*: The expression

```
repeat expr
```

terminates in Version 2 if *expr* fails Note that

```
while expr
```

in Version 5 is equivalent to

```
repeat expr
```

in Version 2

Page 16: The values in case clauses must be literals in Version 2 — they cannot be expressions However, several literals may appear in one case clause, separated by commas, as in

```
1, 2 : write("low")
```

This case clause is selected if the value of the case control expression is 1 or 2

Page 16: Failure of the evaluation of the control expression in a **case** control structure in Version 2 is an error

Chapter 3 — Numbers

Page 20: The remaindering operation is represented in Version 2 as mod(i, j), not as i % j as in Version 5

Page 21: There is no absolute value function in Version 2

Page 22*: There are no augmented assignment operations in Version 2 However, Version 2 has suffix

operations for incrementing and decrementing the values of variables by 1 That is, i+ in Version 2 is equivalent to i +:= 1 in Version 5

Page 23 The random number operation ?i of Version 5 is represented in Version 2 as random(i) In Version 2, random(0) is an error, it does not produce a randomly selected real number between 0 0 and 1 0

Chapter 4 — Character Sets and Strings

Page 25*: The characters \, |, and ! are equivalent in Version 2 and all indicate escapes in quoted literals Consequently, "||" represents a single vertical bar

Page 25: In Version 2, the escape sequence for newline is \l

Page 25: There are no cset literals in Version 2, csets can only be obtained from operations that produce them or by implicit conversion of strings to csets In Version 2, string literals can be enclosed in either single or double quoatation marks Thus 'aeiou' and "aeiou" are equivalent in Version 2 and both represent strings Strings are converted to csets automatically according to context in Version 2 as in Version 5 Generally speaking, string literals can be used in Version 2 in places that cset literals might be used in Version 5

Page 25: In Version 2, the values of &ascii, &lcase, and &ucase are strings, not csets as they are in Version 5

Page 28*: The size operation *s in Version 5 is represented by the function size(s) in Version 2

Pages 29-31*: Version 2 does not have range specifications for substrings Instead it has functions that produce substrings The equivalents are

Version 5	Version 2
s[i:j]	section(s, i, j)
s[i+:j]	substr(s, i, j)
s[i-:j]	substr(s, i, -j)

In these functions, an omitted position is equivalent to 0 Thus section(s, i) is equivalent to s[i:0]

Page 31: Version 2 has a function pos(i, s) that returns the positive equivalent of position i in s, but fails if i does not correspond to a position in s

Page 32: The operation ?s is not available in Version 2

Page 33: The Version 5 lexical comparison operations, except for lexical equality and inequality, are represented in Version 2 by functions as follows

Version 5	Version 2
s1 << s2	llt(s1, s2)
s1 <<= s2	lle(s1, s2)
s2 >> s2	lgt(s1, s2)
s2 >>= s2	lge(s1, s2)

Pages 42-43: The function bal(c1, c2, c3, s) must match at least one character in Version 2

Chapter 5 — Structures

Page 48*: The syntax for list construction in Version 2 uses angular brackets instead of square brackets, as in

$$<expr_1, expr_2, \quad , expr_n>$$

Page 49*: The Version 5 function list(i, x) is represented in Version 2 in the form

```
list(i) initial x
```

and list is a reserved word in Version 2 Both lower and upper bounds can be specified for lists in Version 2 The syntax is

```
list(i:j)
```

where i is the lower bound and j is the upper bound

Page 49*: In Version 2, <> is a list consisting of one null value, not the empty list

Page 50: Nonpositive specifications cannot be used for list referencing in Version 2

Page 52: The operation ?a is not available in Version 2

Page 52: Version 2 does not have list concatenation

Page 53: Version 2 does not have list sections

Pages 53-54*: Version 2 does not have stack and queue access functions for lists There is, however, a stack data type in Version 2 that supports push(k,x) and pop(k) in a fashion similar to the stack access functions for lists in Version 2 In Version 2, a stack is created by

```
stack(i)
```

where i is a limit on the size of the stack A size specification of 0 creates a stack with unlimited size stack is a reserved word in Version 2 The function top(k) references the top element of k Stacks can be referenced by position like lists

Lists also can be opened for expansion in Version 2, so that elements can be added at the right end The function

```
open(a)
```

opens a for expansion and also returns a The function

```
close(a)
```

closes a so that it cannot be expanded and also returns a Empty lists are open when they are created, all other lists are closed when they are created An element is added to the right end of an open list by specifying an index that is one greater than the current size of the list

Page 56*: A table is created in Version 2 by an expression of the form

```
table(i)
```

where i specifies the maximum size of the table (0 indicates a table of unlimited size) table is a reserved word in Version 2 The initial assigned value for entry values that are not in a table is always the null value in Version 2 In Version 2, tables can be opened and closed in the fashion of lists Tables are always open when they are created Reference to a nonexistent entry value in a closed table fails

Page 57: The operation ?t is not available in Version 2

Page 58: The syntax for record declaration in Version 2 is

record *name* $field_1$, $field_2$, ..., $field_n$ end

Unlike Version 5, records in Version 2 must have at least one field. Record names are reserved words in Version 2.

Page 60: The operation ?x is not available in Version 2.

Chapter 6 — Data Types

Page 61. There is no co-expression data type in Version 2.

Page 61. Type names are reserved words in Version 2.

Page 63*: The empty string is convertible to 0 in Version 2.

Page 64: In addition to the type conversion functions of Version 5, null(x) is a type conversion function in Version 2. The values that that are convertible to the null value are the integer 0, the real number 0.0, the empty string, and the null value itself.

Page 65*: The null value is legal in computations in Version 2 and is converted to 0, 0.0, or the empty string depending on context. Thus it is often unnecessary to initialize identifiers in Version 2.

Page 66*: The operations /x and \x are not available in Version 2.

Chapter 7 — Procedures

Page 68: The parentheses in procedure declarations are unnecessary in Version 2 for procedures with no parameters.

Page 69: Version 2 also has the declarations local static and local dynamic with obvious meanings.

Page 70: Version 2 has the declarations implicit local and implicit error that govern the interpretation of undeclared identifiers. implicit local, which is the default, corresponds to the treatment of undeclared identifiers in Version 5. implicit error causes undeclared identifiers to be treated as erroneous.

Page 73. Version 2 has a **return** expression

succeed *expr*

that returns the null value if *expr* fails. An omitted *expr* defaults to the null value.

Page 73*: An implicit fail is not provided at the end of procedure bodies in Version 2. Instead, flowing off the end of a procedure in Version 2 causes the return of the null value.

Page 74: In Version 2, functions, unlike procedures, are not values.

Chapter 8 — Expression Evaluation

Page 80: Version 2 does not have the keyword &fail.

Page 80: In Version 2, braces that enclose an expression or sequence of expressions limit the last expression in braces to at most one result.

Page 81: In Version 2, if $expr_1$ fails in

if $expr_1$ then $expr_2$

the expression returns the null value instead of failing as in Version 5

Page 81*: In Version 2 looping control structures return the null value instead of no result, as in Version 5

Page 81: In Version 2, the **break** expression does not take an argument

Page 84: In Version 2, if the argument of a **suspend** expression is a local identifier, it is not dereferenced, while in Version 5, it is

Page 84*: In Version 2, arguments are dereferenced as they are evaluated, rather than after all of them are evaluated Version 2 does not have an explicit dereferencing operation

Pages 86-87*: Version 2 does not have the mutual evaluation expression

Chapter 9 — Input and Output

Page 88 Version 2 does not have the keyword &errout or the concept of a standard error output file

Page 92 Files cannot be changed in midstream in the write and writes functions in Version 2 An argument that is not convertible to a string in Version 2 is converted to its string image, instead of being an error as in Version 5

Page 93 In Version 2, write and writes return the null value rather than their last argument

Chapter 10 — Miscellaneous Operations

Page 96: In Version 2, the exchange operation returns its right argument instead of its left argument, as in Version 5

Pages 98-99: String images are somewhat different in Version 2 than they are in Version 5 The image of a cset c is given as cset(c), rather than with single quotes Files are given in single quotes Records are given without the designation record, and so on

Page 99 Trace output goes to the standard output file in Version 2 The format of trace messages is different in Versions 2 and 5

Page 100: In Version 2, the function display does not have a second argument and display output goes to the standard output file The format of the display is somewhat different in Versions 2 and 5

Pages 101-102: In Version 2, the format of the value of &date is mm/dd/yy The keywords &dateline, &host, and &version do not exist in Version 2

Chapter 11 — Generators

Page 105: In Version 2, the **every-do** control structure produces the null value when it terminates, as opposed to producing no value as in Version 5

Page 111* Version 2 does not have the limitation control structure

Page 112* Version 2 does not have the repeated alternation control structure

Pages 113-114 In Version 2, if a is an open list, !a generates one element beyond the current size of the list

Page 117: In Version 2, the **suspend** expression produces the null value when it terminates, as opposed to producing no value as in Version 5

Page 119: In Version 2, control backtracking is prevented in the control expressions of **case** control structures as well as in **if-then-else** control structures

Page 120: In Version 2, the reversible exchange operation returns its right argument instead of its left argument as in Version 5

Chapter 12 — String Scanning

Page 123*: The scanning control structure in Version 2 is represented by the reserved wors syntax

scan s using *expr*

instead of

s ? *expr*

as in Version 5

Page 124*: In Version 2, the result of

scan s using *expr*

is the value of &subject, rather than the result of *expr* as in Version 5 In Version 2, s is limited to one result

Page 127*: In Version 2, pos(i) returns the positive equivalent of i with respect to &subject, while in Version 5, pos(i) compares the positive equivalent of i with respect to &subject to &pos as well

Chapter 13 — Co-Expressions

Version 2 does not have co-expressions

Appendix A — Icon Syntax

Page 221: Version 2 has an **include** declaration that causes a named file to be included in the program

Page 221: Version 2 has an **implicit** declaration that governs the interpretation of undeclared identifiers See the note regarding page 70

Page 222: Version 2 does not have an **external** declaration

Page 222: The syntax of record declarations is different in Versions 2 and 5 See the note regarding page 58

Page 222: In Version 2, parentheses are not required in procedure headers if there are no parameters

Page 222: Version 2 has additional local declaration forms See the note regarding page 69

Page 223: Version 2 does not have **limitation**, **transmission**, or **create** expressions Version 2 has reserved-word syntax for creating structures See the notes regarding pages 49, 53-54, and 56

Page 224: The syntax for **list** expressions is different in Versions 2 and 5 See the note regarding page 49

Page 224: Version 2 does not have range specifications

Page 225: In Version 2, the expression at the beginning of an **invocation** expression is not optional

Page 225: Version 2 does not have the prefix **not** expression However, Version 2 has the suffix expressions

expression fails
expression +
expression −

Page 226: Version 2 does not have the infix operators % or |||

Pages 227-228: Version 2 does not have augmented assignment operations

Page 228: The scanning operation has a reserved-word syntax in Version 2 See the note regarding page 123

Page 228: Version 2 has the additional **succeed** expression See the note regarding page 73

Page 228: The **break** expression in Version 2 does not take an argument

Page 229: Case clauses in Version 2 are different from case clauses in Version 5 See the note regarding page 16

Page 229: The **do** clauses in the **while** and **until** control structures are not optional in Version 2

Page 229: Upper- and lowercase letters are equivalent in Version 2

Page 230: Version 2 lacks the reserved words create, external, and not, but has the reserved words cset, error, fails, fail implicit, include, integer list, real, scan, stack, string, succeed, and table

Page 231: Version 2 does not have cset literals

Page 232: Version 2 does not have the escape sequence \n nor the control code escape sequences, but Version 2 has the following escape sequences that are not in Version 5

\<	{
\>	}
\(	[
\)	]

In Version 2, the hexadecimal control sequence must contain two digits

Page 234: Version 2 is designed to run on computers that do not use the ASCII character set The following syntactic equivalences exist in Version 2 programs

lowercase and corresponding uppercase letters
blank and tab (as in Version 5)
^ and ~
[, {, and $(
], }, and $)
|, \, and !

In quoted literals, only the last equivalences apply, other characters being distinct.

Appendix B — Machine Dependencies and Limits

Pages 238-239: Most of the material in this appendix is not relevant to Version 2, but depends on the computer on which Version 2 is implemented. See the appropriate Version 2 user's guide

Appendix C — Running an Icon Program

Pages 240-242: The organization of the Version 2 implementation is different from that of Version 5 and most of the material here is not relevant to Version 2. See the appropriate Version 2 user's guide. Version 2 does not support separate compilation.

Pages 242-243: Version 2 does not have external procedures.

Page 243: The options do not apply to Version 2

Page 244: Environment variables do not apply to Version 2.

Page 244-245: Version 2 does not have an interpreter.

Page 245: Version 2 does not support command-line arguments as the value of an argument to the main procedure

Page 245: Version 2 does not have the system function

Page 245: In Version 2, the exit function does not have an argument and there is no concept of exit status. In some implementations of Version 2, exit() produces an executable core image of the program.

Appendix D — Errors

Pages 246-250: The causes of errors and the error messages are somewhat different in Versions 2 and 5 The error messages for Version 2 follow:

Errors During Translation

assignment to nonvariable
cannot open include file
duplicate declaration for local identifier
duplicate field name
extraneous closing brace
extraneous end
global name previously declared
identifier too long
integer character larger than base
invalid character
invalid construction
invalid context for break
invalid context for next
invalid declaration
invalid escape specification
invalid field name
invalid function call
invalid global declaration

invalid implicit declaration
invalid integer base
invalid integer literal
invalid keyword
invalid keyword construction
invalid operator
invalid real literal
invalid reference
invalid use of field name
misplaced declaration
missing argument
missing closing brace in case expression
missing closing parenthesis
missing colon in case expression
missing declaration
missing do in while or until expression
missing literal in case expression
missing main procedure
missing of in case expression
missing open brace in case expression
missing opening parenthesis
missing procedure end
missing procedure name
missing quote
missing record end
missing record field
missing record name
missing semicolon or operator
missing then in if-then expression
missing using in scan expression
multiple defaults in case expression
multiple implicit declarations
numeric literal too long
procedure name previously declared
string literal too long
unclosed list
unexpected end-of-file

Overflow Conditions

overflow in character table
overflow in global identifier table
overflow in integer literal table
overflow in local identifier table
overflow in nested include files
overflow in parse tree
overflow in procedure block table
overflow in procedure labels
overflow in real literal table
overflow in record table
overflow in string literal table
overflow in translator stack

Program Error Messages

Category 1: Invalid Type or Form

101	integer expected
102	real expected
103	numeric expected
104	string expected
105	cset expected
106	file expected
107	procedure expected
108	record expected
109	stack expected
111	invalid type to size
112	invalid type to close
121	variable expected

Category 2: Invalid Argument or Computation

201	division by zero
202	zero second argument to mod
203	integer overflow
204	real overflow
205	real underflow
206	negative first argument in real exponentiation
207	invalid value to random or &random
210	invalid field name
211	negative second argument to repl
212	negative second argument to left
213	negative second argument to right
214	negative second argument to center
215	second and third arguments to map of unequal length
216	erroneous list bounds
217	negative stack size
218	negative table size
219	invalid first argument to sort
220	invalid second argument to sort
221	invalid second argument to open
222	invalid second argument to reads
230	case expression failure

Category 3: Invalid Structure Operation

301	table size exceeded
302	stack size exceeded

Category 4: Input/Output Errors

401	cannot close file
402	attempt to read file not open for reading
403	attempt to write file not open for writing
411	input string too long

Category 5: Capacity Exceeded

501	insufficient storage
502	control stack overflow

Appendix E — Summary of Built-In Operations

No attempt is made here to recapitulate the differences between the built-in operations of Version 2 and Version 5 One point that is not mentioned elsewhere in the book deserves note, however

Page 255: In Version 2, there are no defaults for the second and third arguments of map

References

1 Griswold, Ralph E and David R Hanson *Reference Manual for the Icon Programming Language, Version 2* Technical Report TR 79-1a, Department of Computer Science, The University of Arizona 1980

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

Appendix — Icon Book Procedures for Version 2

Procedures from the book that work properly in Version 2 without modification are omitted here without comment. Procedures that cannot be converted to Version 2 are also omitted, but noted. All the procedures that follow have been tested under Version 2. They are available from the author in machine-readable form upon request.

Chapter 2

Page 14:

```
procedure main()
   repeat {                                  # exit repeat if eof
      if (headline := read()) fails then break
      count := 1
      while line := read() do {              # exit while if "end"
         count := count + 1
         if match("end", line) then break
         }
      write(headline, " : ", count)
      }
end
```

Chapter 4

Page 27: The second procedure on this page cannot be converted, since Version 2 does not have augmented assignment operations.

Page 29:

```
procedure wordlist()
   wlist := ""  # initialize wlist
   while word := read() do
      wlist := wlist || word || ","
   return wlist
end
```

Page 30:

```
procedure main()
   while line := read() do {
      line := section(line, 1, 61)          # truncate
      write(line)
      }
end
```

Page 34:

```
procedure main()
   min := max := read()      # initial min and max
   while line := read() do
      if lgt(line, max) then max := line
      else if llt(line, min) then min := line
   write("lexically largest line is: ", max)
   write("lexically smallest line is: ", min)
end
```

Page 34: The second procedure on this page cannot be converted, since Version 2 does not have augmented assignment operations

Page 35:

```
procedure main()
   i := 0
   while i < 10 do {
      i+
      write(right(i, 5), right(i ^ 2, 8), right(i ^ 3, 8),
         right(i ^ 4, 8))
      }
end
```

Page 38:

```
procedure main()
   s2 := &cset || "AEIOUaeiou"
   s3 := repl(" ", size(&cset)) || repl("\|", 10)
   while line := read() do {
      write(line)
      write(map(line, s2, s3))
      }
end
```

Note the necessity of the escape sequence for the vertical bar

Page 39:

```
procedure lmark(s)
   while line := read() do {
      write(line)
      write(repl(" ", find(s, line) - 1), "\|")
      }
   return
end
```

Page 41:

```
procedure main()
   wchar := &lcase ++ &ucase
   pchar := '., :;?! '
   while line := read() do
                                    # get to first letter
      if line := section(line, upto(wchar, line))
      then write(section(line, 1, upto(pchar, line)))
end
```

Page 42:

```
procedure main()
   wchar := &lcase ++ &ucase
   while line := read() do
      if line := section(line, upto(wchar, line))
      then write(section(line, 1, many(wchar, line)))
end
```

Page 43:

```
procedure main()
   while line := read() do {
      write(line)
      write(repl(" ", bal('+-*/', ,, line) - 1), "\|")
      }
end
```

Page 45:

```
procedure main()
   wchar := &lcase ++ &ucase ++ '\'-'
   while line := read() do {
      i := 1
      while j := upto(wchar, line, i) do {
         i := many(wchar, line, j)
         write(section(line, i, j))
         }
      }
end
```

Chapter 5

Page 51:

```
procedure main()
   wchar := &lcase ++ &ucase ++ '\'-'
   wordlength := list(10) initial 0      # initial zero counts
   while line := read() do {
      i := 1
      while j := upto(wchar, line, i) do {
         i := many(wchar, line, j)
         wordlength[size(section(line, i, j))]+
         }
      }
   write("word length    count:\n")
   i := 0
   while i < size(wordlength) do {
      i+
      write(left(i || ":", 12), right(wordlength[i], 3))
      }
end
```

Page 52:

```
procedure array(i, j, x)
   a  = list(i) initial 0
   k := 0
   repeat a[k+] := list(j) initial x
   return a
end
```

Page 53:

```
procedure words()
   wchar := &lcase ++ &ucase ++ '\'-'
   wordlist := list(0)
   index := 0
   while line := read() do {
      i := 1
      while j := upto(wchar, line, i) do {
         i := many(wchar, line, j)
         wordlist[index+] := section(line, i, j)  # add to list
         }
      }
   return close(wordlist)
end
```

Page 54: The procedure on this page cannot be converted, since Version 2 does not have queue access functions for lists. If the procedure on the previous page is rewritten using stacks, then this procedure works as it stands in the book.

Pages 57-58:

```
procedure tabwords()
   wchar := &lcase ++ &ucase ++ '\'-'
   words := table(0)
   while line := read() do {
      i := 1
      while j := upto(wchar, line, i) do {
         i := many(wchar, line, j)
         words[section(line, i, j)]+                    # increment count
         }
      }
   return words
end

procedure main()
   wlist := sort(tabwords())                            # get sorted list
   i := 0
   while pair := wlist[i+] do
      write(left(pair[1], 12), right(pair[2], 3))
end
```

Chapter 7

Page 69:

```
procedure exor(s1, s2)
   local count, line
   count := 0
   while line := read() do
      if find(s1, line) then {
         if find(s2, line) fails then count+
         }
      else if find(s2, line) then count+
   return count
end
```

Page 71:

```
procedure main()
   repeat write(fword())
end

procedure fword()
   local wchar, line
   wchar := &lcase ++ &ucase ++ '\'-'
   while line := read() do
      if line := section(line, upto(wchar, line), 0)
      then return section(line, 1, many(wchar, line))
   fail
end
```

Page 71:

```
procedure fword()
   static wchar
   local wchar, line
   initial wchar := &lcase ++ &ucase ++ '\'-'
   while line := read() do
      if line := section(line, upto(wchar, line), 0)
      then return section(line, 1, many(wchar, line))
   fail
end
```

Page 72:

```
procedure nword()
   static wchar, line, i, j
   initial {
      wchar := &lcase ++ &ucase ++ '\'-'
      if line := read() then i := 1 else fail
      }
   repeat {
      while j := upto(wchar, line, i) do {
         i := many(wchar, line, j)
         return section(line, i, j)
         }
      if line := read() then i := 1 else fail
      }
end
```

Page 75:

```
procedure fib(i)
   static fibmem
   local j
   initial {
      fibmem := table()
      fibmem[1] := fibmem[2] := 1
      }
   if null(j := fibmem[i]) fails then return j
   else return fibmem[i] := fib(i - 1) + fib(i - 2)
end
```

Page 77:

```
procedure expression()
   return case random(2) of {
      1  :  term()
      2  :  term() || "+" || expression()
      }
end
```

```
procedure term()
   return case random(2) of {
      1  :  element()
      2  :  element() || "*" || term()
      }
end

procedure element()
   return case random(4) of {
      1  :  "x"
      2  :  "y"
      3  :  "z"
      4  :  "(" || expression() || ")"
      }
end
```

Page 78:

```
procedure expression()
   return case random(3) of {
      1  :  term()
      2  :  term()
      3  :  term() || "+" || expression()
      }
end
```

Chapter 8

Page 82: The procedure on this page cannot be converted, since in Version 2, the **break** expression does not take an argument.

Page 84:

```
procedure maxel(a)
   local i, j, max
   j := i := 1
   max := a[1]
   while i < size(a) do {
      i+
      if max < a[i] then {max := a[i]; j := i}
      }
   return a[j]
end
```

Chapter 9

Page 90:

```
procedure main()
   if (intext := open("shaw.txt")) fails
   then stop("cannot open input file")
   repeat write(read(intext))
end
```

Page 92:

```
procedure main()
   if (intext := open("shaw.txt")) fails
   then stop("cannot open input file")
   if (outtext := open("shaw.cpy", "w")) fails
   then stop("cannot open output file")
   while line := read(intext) do {
      write(line)
      write(outtext, line)
      }
end
```

Pages 92-93:

```
procedure main()
   writes("specify input file: ")
   repeat {
      if intext := open(read()) then break
      writes("cannot open input file, respecify: ")
      }
   writes("specify output file: ")
   repeat {
      if outtext := open(read(), "w") then break
      writes("cannot open output file, respecify: ")
      }
   repeat write(outtext, read(intext))
end
```

Chapter 10

Page 97:

```
procedure shuffle(s)
   local i
   i := size(s)
   while i >= 2 do {
      s[random(i)] :=: s[i]
      i-
      }
   return s
end
```

Page 101:

```
procedure main()
   local intext
   if (intext := open("shaw.txt")) fails
   then stop("cannot open input file")
   write(linecount(intext))
end

procedure linecount(file)
   local count, line
   count := 0
   while line := read(file) do
      if find("stop", line) then break
      else count+
   display()
   return count
end
```

Chapter 11

Page 105:

```
procedure mark(s)
   local line, marker, i
   while line := read() do {
      marker := ""                                  # initialize marker
      every i := find(s, line) do
         marker := left(marker, i - 1) || "\|"
      write(line, "\n", marker)                     # write line and marker
      }
   return
end
```

Page 115:

```
procedure main()
   while line := read() do {
      marker := ""
      every i := bal('+-/*', ,, line) do
         marker := left(marker, i - 1) || line[i]
      write(line, "\n", marker)    # write line and marker
      }
end
```

Page 116:

```
procedure To_(i, j)
   while i <= j do {
      suspend i
      i+
      }
   fail
end
```

Note the insertion of the **fail** expression to prevent an unwanted null value from being returned by flow off the procedure body.

Page 117:

```
procedure genword()
   static wchar
   local line, i, j
   initial wchar := &lcase ++ &ucase ++ '\'-'
   while line := read() do {
      i := 1
      while j := upto(wchar, line, i) do {
         i := many(wchar, line, j)
         suspend section(line, i, j)       # produce word
         }
      }
   fail
end
```

Page 117:

```
procedure rtl(s)
   suspend s[size(s) to 1 by -1]
   fail
end
```

Page 123:

```
procedure tabwords()
   static wchar
   local words, line
   initial wchar := &lcase ++ &ucase ++ '\'-'
   words := table(0)
   while line := read() do
      scan line using while tab(upto(wchar)) do
         words[tab(many(wchar))]+
   return words
end
```

Chapter 12

Page 129:

```
procedure words()
   local wchar, line, word
   wchar := &lcase ++ &ucase ++ '\'-'
   while line := read() do
      scan line using while tab(upto(wchar)) do {
         word := tab(many(wchar))
         suspend word
         }
   fail
end
```

Chapter 13

The procedures in Chapter 13 cannot be converted to Version 2.

Chapter 14

Page 144: The procedure on this page cannot be converted, since Version 2 does not have the limitation control structure.

Page 145:

```
procedure posint()
   local i
   i := 0
   repeat suspend i+
end
```

Page 146: The procedures on this page cannot be converted, since Version 2 does not have the limitation control structure or co-expressions.

Page 147:

```
procedure cross(word1, word2)
   local i, j
   if i := upto(word2, word1)
   then {
      j := upto(word1[i], word2)
      every write(right(word2[1 to j - 1], i))
      write(word1)
      every write(right(word2[j + 1 to size(word2)], i))
      write()
      }
   return
end
```

Page 148:

```
procedure cross(word1, word2)
   local i, j
   every i := upto(word2, word1) do
      every  j := upto(word1[i], word2) do {
         every write(right(word2[1 to j - 1], i))
         write(word1)
         every write(right(word2[j + 1 to size(word2)], i))
         write()
         }
  return
end
```

Page 153:

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

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

procedure place(r, c)
   static up, down, row
   initial {
      up := list(-7:7) initial 0
      down := list(2:16) initial 0
      row := list(8) initial 0
      }
   if row[r] = down[r + c] = up[r - c] = 0
   then suspend row[r] <- down[r + c] <-
      up[r - c] <- r                 # place if free
   fail
end
```

Chapter 15

Page 160:

```
procedure tab(i)
   suspend section(&subject, &pos, &pos <- i)
   fail
end
```

Page 160:

```
procedure arb()
   suspend section(&subject, &pos, &pos <- &pos to size(&subject) + 1)
   fail
end
```

Page 160:

```
procedure rarb()
   suspend section(&subject, &pos,
      &pos <- ((size(&subject) + 1) to &pos by -1))
   fail
end
```

Page 161:

```
procedure lmatch(slist)
   suspend =!slist
   fail
end
```

Page 161:

```
procedure arbno(p)
   suspend "" | (p() || arbno(p))
   fail
end
```

Page 161:

```
procedure shades()
   suspend arb() || lmatch(<"black", "white", "gray">)
   fail
end
```

Pages 163-164:

```
procedure main()
   while writes(line := read()) do
      if scan line using (X() & (pos(0) = &pos)) then write(" accepted")
      else write(" rejected")
end

procedure X()
   suspend T() | (T() || "+" || X())
   fail
end

procedure T()
   suspend E() | (E() || "*" || T())
   fail
end
```

```
procedure E()
   suspend ="x" | ="y" | ="z" | (="(" || X() || =")")
   fail
end
```

Page 165:

```
procedure main()
   while writes(line := read()) do
      if scan line using (ABC("") & (pos(0) = &pos)) then write(" accepted")
      else write(" rejected")
end

procedure ABC(X)
   suspend =X | (="a" || ABC("b" || X) || ="c")
   fail
end
```

Pages 167-168:

```
procedure main()
   while writes(line := read()) do
      if scan line using ((a := X()) & (pos(0) = &pos)) then {
         write(" accepted")
         write(image(a))
         }
      else write(" rejected")
end

procedure T()
   suspend <"T", E()> | <"T", E(), ="*", T()>
   fail
end

procedure E()
   suspend <"E", =("x" | "y" | "z")> | <"E", ="(", X(), =")">
   fail
end

procedure X()
   suspend <"X", T()> | <"X", T(), ="+", X()>
   fail
end
```

Page 172:

```
procedure ltree(stree)
   local a, i
   scan stree using
      if a := <tab(upto('('))> then {                    # start with value
         move(1)
         open(a)
         i := 1
                                                         # add subtrees
         while a[i+] := ltree(tab(bal(',')'))) do
            move(1)
         }
      else a  = <tab(0)>                                 # leaf
   return close(a)
end
```

Page 173:

```
procedure stree(ltree)
   local s, s1
   if size(ltree) = 1 then return ltree[1]               # start with leaf
   s := ltree[1] || "("                                  # append value
                                                         # append stree
   every s1 := stree(ltree[2 to size(ltree)]) do
      s := s || s1 || ","
   return section(s, 1, -1) || ")"
end
```

This procedure deserves special note It might appear that the Version 5 code segment

```
every s ||:= stree(ltree[2 to *ltree]) || ","
```

could be rewritten in Version 2 as

```
every s  = s || stree(ltree[2 to *ltree]) || ","
```

However, since arguments in Version 2 are dereferenced as they are evaluated (see note regarding page 84), the value of s as the first argument of the concatenation remains the same throughout the computation — the assignments to s do not change it Hence the result of the proposed rewriting above is not what is desired In Version 2, it is good policy not to perform computations with side effects in **every** clauses

Page 173:

```
procedure visit(ltree)
   suspend ltree | visit(ltree[2 to size(ltree)])
   fail
end
```

Page 174:

```
procedure teq(a1, a2)
   local i
   if size(a1) ~= size(a2) then fail                    # check sizes
   if a1[1] ~=== a2[1] then fail                        # check values
   every i := 2 to size(a1) do                          # check subtrees
      if teq(a1[i], a2[i]) fails then fail
   return a2
end
```

Page 174:

```
procedure eq(x, y)
   local i
   if x === y then return y                             # succeed if identical
   if type(x) == type(y) == "list" then {
      if size(x) ~= size(y) then fail                   # check sizes
      every i := 1 to size(x) do                        # check subtrees
         if eq(x[i], y[i]) fails then fail
      return y
     }
end
```

Page 176:

```
procedure ldag(stree, done)
   local a, i
   if null(done) then done := table()
                                                        # return list if done
   if null(a := done[stree]) fails then return a
   scan stree using
      if a := <tab(upto('('))> then {
         move(1)
         i := 1
         open(a)
         while a[i+] := ldag(tab(bal(',')), done) do
            move(1)
         }
      else a := <tab(0)>
   return done[stree] := close(a)                       # put in table
end
```

Page 179:

```
procedure lgraph(sgraph)
   local nodes, ndescr, nlist, a, name, i
   nodes := table()                              # table of nodes
   scan sgraph using
      while ndescr := tab(many(~';')) do {
         move(1)
         scan ndescr using {                     # process one node
            a := list(0)                         # new list goes in table
            nodes[tab(upto(':'))] := a
            move(1)
            i := 0
                                                 # add value and names
            while a[i+] := tab(many(~', ')) do
               move(1)
            }
         }
   every name := !nodes do                       # change names to lists
      every i := 2 to size(name) do
         name[i] := nodes[name[i]]
   return nodes
end
```

Chapter 17

Page 183:

```
procedure reverse(s)
   static labels, trans, max
   initial {
      labels := "abcdefghijklmnopqrstuvwxyz"
      trans := "zyxwvutsrqponmlkjihgfedcba"
      max := size(labels)
      }
   if size(s) <= max then
      return map(right(trans, size(s)), left(labels, size(s)), s)
   else return reverse(right(s, size(s) - max)) ||
      map(trans, labels, left(s, max))
end
```

Page 185:

```
procedure swap(s)
   static labels, trans, max
   initial {
      labels := "12"
      trans := "21"
      max := size(labels)
      trans := swap(string(&cset))
      labels := string(&cset)
      max := size(labels)
      }
   if size(s) <= max then
      return map(left(trans, size(s)), left(labels, size(s)), s)
   else return swap(left(s, size(s) - max)) ||
      map(trans, labels, right(s, max))
end
```

Page 187:

```
procedure shuffle(deck)
   every !deck :=: deck[random(size(deck))]
   return deck
end
```

Page 188:

```
procedure disp(deck)
   static fresh, suits, denoms
   initial {
      fresh := &ucase || &lcase
      suits := repl("C", 13) || repl("D", 13) || repl("H", 13) ||
         repl("S", 13)
      denoms := repl("A23456789TJQK", 4)
      }
   write(map(deck, fresh, suits))                    # suits
   write(map(deck, fresh, denoms))                   # denominations
end
```

Page 190:

```
procedure successors(graph, nodes)
   local snodes
   snodes := ''                                      # start with none
   scan graph using repeat {
      if tab(any(nodes)) then snodes := snodes ++ move(1)
      else move(2) | break                           # exit at end of string
      }
   return snodes
end
```

```
procedure closure(graph, nodes)
   local snodes
   snodes := nodes                                  # start with given nodes
   while snodes ~===
      (nodes := nodes ++ successors(graph, nodes)) do
         snodes := nodes                            # update if change
   return nodes
end
```

Chapter 18

Page 195:

```
global base, segsize

procedure add(s1, s2, carry)
   local siz, sum
   if size(s1) > size(s2) then s1 :=: s2
   siz := size(s2)
   if siz <= segsize then return s1 + s2 + carry
   s1 := right(s1, siz, "0")
   sum := right(s1, segsize) + right(s2, segsize) + carry
   return add(left(s1, siz - segsize),
      left(s2, siz - segsize), sum / base) ||
      right(mod(sum, base), segsize, "0")
end
```

Pages 196-197:

```
global base, segsize

procedure large(s)
   local a, i
   a := list(0)
   scan s using {
      &pos := 0                                     # start at right end
      i := 0
      repeat a[i+] := integer(move(-segsize))
                                                    # add remaining digits
      if &pos ~= 1 then a[i] := integer(tab(1))
      }
   return a
end
```

```
procedure add(a1, a2)
   local carry, d, i, a3
   carry := 0
   if size(a1) < size(a2) then a1 :=: a2
   i := size(a2)
   while size(a2) < size(a1) do a2[i+] := 0
   a3 := list(size(a1)) initial 0
   open(a3)
   every i := 1 to size(a1) do {
      d := a1[i] + a2[i] + carry
      carry := d / base
      a3[i] := mod(d, base)
      }
   if carry > 0 then a3[i+] := carry
   return a3
end

procedure lstring(a)
   local s
   s := ""
   every s := right(!a, segsize, "0") || s
   scan s using (tab(upto(~'0') | -1) & (s :=  tab(0)))
   return s
end
```

The procedure add given above is not really a conversion of the procedure given in the book, which uses operations on lists that are not in Version 2. Instead, it is a more conventional approach to manipulating large integers using lists. The procedure in the book can be converted to Version 2 by using stacks instead of lists.

Page 198:

```
record largint coeff, link end
global base, segsize

procedure add(g1, g2, carry)
   local sum
   if null(carry) then carry := largint(0)
   if null(g1) & null(g2) then return if carry.coeff ~= 0 then carry
   else &null
   if null(g1) then return add(carry, g2)
   if null(g2) then return add(g1, carry)
   sum := g1.coeff + g2.coeff + carry.coeff
   carry := largint(sum / base)
   return largint(mod(sum, base), add(g1.link, g2.link, carry))
end

procedure large(s)
   if size(s) <= segsize then return largint(integer(s))
   else return largint(right(s, segsize),
      large(left(s, size(s) - segsize)))
end
```

```
procedure lstring(g)
   local s
   if null(g.link) then s := g.coeff
   else s := lstring(g.link) || right(g.coeff, segsize, "0")
   scan s using (tab(upto(~'0') | -1) & (s := tab(0)))
   return s
end
```

Page 199:

```
procedure mpy(g1, g2)
   local prod
   if null(g1 | g2) then return &null    # zero product
   prod := g1.coeff * g2.coeff
   return largint(mod(prod, base),
      add(mpy(largint(g1.coeff), g2.link), mpy(g1.link, g2),
      largint(prod / base)))
end
```

Chapter 19

Page 205:

```
procedure main()
   repeat write(fix(read()))
end

procedure fix(exp)
   repeat scan exp using (
      ="(" & (exp := tab(bal(')'))) & (pos(-1) = &pos)
      )
   return lassoc(exp, '+-' | '*/') | rassoc(exp, '^') | exp
end

procedure lassoc(exp, op)
   local j
   scan exp using {
      every j := bal(op)
      if null(j) fails then exp := form(tab(j), move(1), tab(0))
      }
   if null(j) then fail else return exp
end

procedure rassoc(exp, op)
   if scan exp using (exp := form(tab(bal(op)), move(1), tab(0)))
   then return exp else fail
end

procedure form(arg1, op, arg2)
   return op || "(" || fix(arg1) || "," || fix(arg2) || ")"
end
```

Page 206:

```
procedure form(arg1, op, arg2)
   return <op, fix(arg1), fix(arg2)>
end
```

Page 206:

```
procedure form(arg1, op, arg2)
   arg1 := fix(arg1)
   arg2 := fix(arg2)
   return case op of {
      "+"  :  add(arg1, arg2)
      "-"  :  sub(arg1, arg2)
      "*"  :  mpy(arg1, arg2)
      "/"  :  div(arg1, arg2)
      "^"  :  rse(arg1, arg2)
      }
end
```

Page 207:

```
procedure add(arg1, arg2)
   local i
   return {
      if i := integer(arg1) + integer(arg2) then i
      else if arg1 == "0" then arg2
      else if arg2 == "0" then arg1
      else if arg1 == arg2 then symop("2", "*", arg2)
      else symop(arg1, "+", arg2)
      }
end
```

Page 209:

```
procedure drv(exp, var)
   local arg1, op, arg2
   if scan exp using {
      op := tab(upto('(')) &
      move(1) &
      arg1 := tab(bal(',')) &
      move(1) &
      arg2 := tab(bal(')'))
      }
   then return case op of {
      "+" : add(drv(arg1, var), drv(arg2, var))
      "-" : sub(drv(arg1, var), drv(arg2, var))
      "*" : add(mpy(arg1, drv(arg2, var)),
                mpy(arg2, drv(arg1, var)))
      "/" : div(sub(mpy(arg2, drv(arg1, var)),
                mpy(arg1, drv(arg2, var))), rse(arg2, "2"))
      "^" : mpy(mpy(arg2, rse(arg1, sub(arg2, "1"))),
                drv(arg1, var))
      }
   else return if exp == var then "1" else "0"
end
```

Chapter 20

Pages 217-218:

```
global defs

record nonterm name end

procedure main()
   local line
   defs := table()
   while line := read() do
      (define | generate)(line)
end

procedure define(line)
   return scan line using
      defs[(="<" & tab(find(">::=")))] := (move(4) & alts(tab(0)))
end

procedure alts(defn)
   local alist
   alist := stack()
   scan defn using while push(alist, syms(tab(many(~'\|')))) do move(1)
   return alist
end
```

```
procedure syms(alt)
   local slist
   slist := stack()
   scan alt using repeat push(slist, tab(many(~'<')) |
      puterm(="<", tab(upto('>')), move(1)))
   return slist
end

procedure puterm(x, y, z)
   return nonterm(y)
end

procedure generate(line)
   local goal, count, i
   scan line using {
      ="<" &
      goal := tab(upto('>')) &
      move(1) &
      count := tab(0)
      }
   i := 0
   every write(gener(goal)) do {
      i+
      if i > count then break
      }
   return
end

procedure gener(goal)
   local pending, genstr, symbol
   pending := stack()
   repeat {
      push(pending, nonterm(goal))
      genstr := ""
      while symbol := pop(pending) do
         if type(symbol) == "string" then genstr := genstr || symbol
         else {
            x := defs[symbol.name]
            y := x[random(size(x))]
            every push(pending, !y)
            }
      suspend genstr
      }
end
```

Appendix F

Page 282:

```
procedure exor(s1, s2)
   count := 0
   while line := read() do
      if find(s1, line) then {
         if find(s2, line) fails then count := count + 1
         }
      else if find(s2, line) then count := count + 1
   return count
end
```

Page 283:

```
procedure main()
   sum := 0.0
   count := 0
   while sum := sum + read() do
      count+
   write(sum / count)
end
```

Page 283:

```
procedure fact(i)
   j := 1
   while i > 0 do {
      j := j * i
      i-
      }
   return j
end
```

Page 284:

```
procedure delete(s, c)
   repeat s[upto(c, s)] := ""
   return s
end
```

Page 284:

```
procedure delete(s, c)
   while i := upto(c, s) do
      section(s, i, many(c, s, i)) := ""
   return s
end
```

Page 284:

```
procedure main()
   wchar := &lcase ++ &ucase ++ '\'-'
   while line := read() do {
      i := 1
      dashes := repl(" ", size(line))
      while j := upto(wchar, line, i) do {
         i := many(wchar, line, j)
         section(dashes, i, j) := repl("-", i - j)
         }
      write(line)
      write(dashes)
      }
end
```

Page 284:

```
procedure main()
   local i
   lines := list(0)
   repeat lines[i+] := read()
   repeat write(lines[i-])
end
```

Pages 285-286:

```
record complex rpart, ipart end

procedure strcpx(s)
   i := upto('+-', s, 2)
   return complex(section(s, 1, i), section(s, i, -1))
end

procedure cpxstr(x)
   if x.ipart < 0 then return x.rpart || x.ipart || "i"
   else return x.rpart || "+" || x.ipart || "i"
end

procedure cpxadd(x1, x2)
   return complex(x1.rpart + x2.rpart, x1.ipart + x2.ipart)
end

procedure cpxsub(x1, x2)
   return complex(x1.rpart - x2.rpart, x1.ipart - x2.ipart)
end

procedure cpxmul(x1, x2)
   return complex(x1.rpart * x2.rpart - x1.ipart * x2.ipart,
      x1.rpart * x2.ipart + x1.ipart * x2.rpart)
end
```

```
procedure cpxdiv(x1, x2)
   denom := x2.rpart ^ 2 + x2.ipart ^ 2
   return complex((x1.rpart * x2.rpart + x1.ipart * x2.ipart) /
      denom, (x1.ipart * x2.rpart - x1.rpart * x2.ipart) /
      denom)
end
```

Page 287:

```
procedure acker(i, j)
   local args, k
   static ackermem
   initial ackermem := table()
   args := i || "," || j
   if null(k := ackermem[args]) fails then return k
   if i = 0 then return ackermem[args] := j + 1
   if j = 0 then return ackermem[args] := acker(i - 1, 1)
   return ackermem[args] := acker(i - 1, acker(i, j - 1))
end
```

Page 287: Exercise 8.3 cannot be solved in Version 2, since Version 2 does not have the mutual evaluation control structure

Page 287:

```
procedure main()
   repeat writes(reads(, 100))
end
```

Page 288:

```
procedure main()
   local chars, charlist, i, pair
   chars := table(0)
   repeat chars[reads()]+
   charlist := sort(chars)
   i := 0
   while pair := charlist[i+] do
      write(left(image(pair[1]), 6), right(pair[2], 6))
end
```

Page 288:

```
procedure ackertrace(i, j)
   static level
   local result
   initial level := 0
   write(repl("x", level+))
   if i = 0 then result := j + 1
   else if j = 0 then result := ackertrace(i - 1, 1)
   else result := ackertrace(i - 1, ackertrace(i, j - 1))
   level-
   return result
end
```

Page 289:

```
procedure genpos(a, x)
   local i
   every i := 1 to size(a) do
      if a[i] === x then suspend i
   fail
end
```

Page 290:

```
procedure allbal(c, s)
   local i
   every i := 1 to (size(s) - 1) do
      suspend section(s, i, bal(c, ,, s, i))
   fail
end
```

Page 290:

```
procedure enrepl(s)
   local c, s1
   s1 := ""
   scan s using while c := move(1) do {
      i := 1 + (size(tab(many(c))) | 0)
      if i > 4 then s1 := s1 || c || "(" || i || ")"
      else s1 := s1 || repl(c, i)
      }
   return s1
end
```

```
procedure derepl(s)
   local c, s1
   s1 := ""
   scan s using {
      while s1 := s1 || tab(upto('(') - 1) do {
         c := move(1)
         move(1)
         s1 := s1 ||  repl(c, tab(upto(')')))
         move(1)
         }
      s1 := s1 || tab(0)
      }
  return s1
end
```

Page 291:

```
procedure allbal(c, s)
   scan s using repeat {
      suspend tab(bal(c))
      move(1) | break
      }
  fail
end
```

Pages 291-293: The exercises from Chapter 13 cannot be done in Version 2.

Page 293:

```
procedure main()
   every print(<q(1), q(2), q(3), q(4), q(5), q(6), q(7), q(8)>)
end

procedure q(r)
   suspend place(1 to 8, r)
   fail
end

procedure place(c, r)
   static up, down, col
   initial {
      up := list(-7:7) initial 0
      down := list(2:16) initial 0
      col := list(8) initial 0
      }
   if col[c] = up[r - c] = down[r + c] = 0
   then suspend col[c] <- up[r - c] <-
      down[r + c] <- c
   fail
end
```

```
procedure print(a)
   static line, bar
   initial {
      line := repl("+-", 8) || "+"
      bar := repl("\| ", 8) || "\|"
      }
   every bar[!a*2] <- "Q" do {
      write(line)
      write(bar)
      }
      write(line, "\n\n")
   return
end
```

Page 294:

```
procedure allbal(c, s)
   scan s using suspend arb() & tab(bal(c))
   fail
end
```

Page 295:

```
procedure main()
   while writes(line := read()) do
      if scan line using (ABCD("", "", "", "") & (pos(0) = &pos))
      then write(" accepted")
      else write(" rejected")
end

procedure ABCD(A, B, C, D)
   suspend (=A || =B || =C || =D) |
      (="a" || ABCD(A, "b" || B, C || "c", D) || ="d")
   fail
end
```

Page 295:

```
procedure depth(ltree)
   local count, i
   count := 0
   every i := 1 + depth(ltree[2 to size(ltree)]) do
      if count < i then count := i
   return count
end
```

Page 296:

```
record bnode value, left, right end

procedure btree(stree)
   local x
   scan stree using if x := bnode(tab(upto('('))) then {
      move(1)
      x.left := btree(tab(bal(',')))
      move(1)
      x.right := btree(tab(bal(')')))
      }
      else x := bnode(tab(0))
   return x
end

procedure stree(btree)
   local s
   if null(btree.left) then return btree.value
   s := btree.value || "(" ||
      stree(btree.left) || "," || stree(btree.right) || ")"
   return s
end
```

Page 297:

```
procedure boldface(s)
   local c
   static labels, trans, max
   initial {
      labels := "1"
      trans := "1\b1\b1\b1\b1"
      max := size(labels)
      trans := boldface(string(&cset -- '\b'))
      labels := string(&cset -- '\b')
      max := size(labels)
      }
   if size(s) <= max then
      return map(left(trans, 9 * size(s)), left(labels, size(s)), s)
   else return boldface(left(s, size(s) - max)) ||
      map(trans, labels, right(s, max))
end
```

Page 298:

```
procedure cdigit(s)
   local s1
   s1 := ""
    scan s using {
      &pos := 0
      repeat s1 := "," || move(-3) || s1
      if pos(1) = &pos then s1 := section(s1, 2)
      else s1 := tab(1) || s1
      }
   return s1
end
```

Page 300:

```
procedure fix(exp)
   repeat scan exp using (
      ="(" & (exp := tab(bal(')'))) & (pos(-1) = &pos)
      )
   return lassoc(exp, '+-' | '*/') | rassoc(exp, '^') |
      func(exp) | exp
end

procedure func(exp)
   scan exp using exp := (tab(upto('(') + 1) || fix(tab(-1)) || tab(0))
   return exp
end
```

Page 301:

```
procedure generate(line)
   local goal, count, i
   if scan line using {
      ="<" &
      goal := tab(upto('>')) &
      move(1) &
      count := (0 <= integer(tab(0)))
      }
   then {
      i := 0
      every write(gener(goal)) do {
         i+
         if i > count then break
         }
      return
      }
   else fail
end
```