The Icon Program Library; Version 6, Release 1*

Ralph E. Griswold

TR 86-13b

June 12, 1986, Last revised October 17, 1986

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

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

The Icon Program Library; Version 6, Release 1

Introduction

This version of the Icon program library is intended for use with Version 6 of Icon. Basic documentation for Version 6 of Icon is contained in the Icon book [1] and a supplementary report [2].

The library contains both complete programs and collections of procedures. The programs range from demonstrations and games to text-processing utilities. The procedures range from straightforward extensions to Icon's function repertoire to such relatively esoteric subjects as programmer-defined control operations. This manual is divided into two main parts according to the composition of the library: complete programs and collections of procedures.

While the library provides some useful application programs and components that may be helpful in building other programs, it also provides examples of Icon programming techniques. In particular, persons who are new to Icon may find it helpful to read the source code for the library to see how experienced persons program in Icon. While not all of the code is the best possible — far from it — it illustrates useful idioms and a variety programming techniques.

In the descriptions that follow, there are pointers to interesting programming techniques as well as several suggestions for extensions and improvements to programs. Such extensions are good exercises persons who are just starting in Icon. Some of these extensions, however, will challenge the most experienced Icon programmer.

Library Format

The root directory of the library is ipl (‘‘Icon program library’’). There are four subdirectories: source, progs, procs, and data. The subdirectory source contains Icon source code for both programs and procedure libraries. Compiled programs are in progs and translated procedures are in procs. The subdirectory data contains sample input for programs in progs. The names of programs and data files generally coincide, with the extensions of data files providing some differentiating identification. For example, the data file csgen.abc is input to the program csgen. There are also several files with the extension .txt that contain English-language text that is suitable as input to any of the programs that process text files.

Disclaimer

The material contained in the Icon program library is provided on an as-is basis. No claim is made that the programs are free of error or that they will function properly. The responsibility for the use of library material resides entirely with the user.

Notes of errors will be appreciated and corrections will be incorporated in future releases of the library.

New Material

Additions are made to the Icon program library from time to time. New material is welcome. Such material should be sent to:

Icon Project
Department of Computer Science
The University of Arizona
Tucson, AZ 87521

Documentation similar in form to that provided in this manual *must* be included and test data should be provided where appropriate. The final decision on inclusion of material in the library resides with the Icon Project.

Acknowledgements

Several persons have contributed programs and procedures to the Icon program library. In addition to the author of this manual, these persons include Allan Anderson, Ward Cunningham, Tom Hicks, William Malloy, Bill Mitchell, Mike Novak, Randal Schwartz, Steve Wampler, and George Yee. See the source files for specific attributions.

Programs

Introduction

Most programs take input from standard input and write output to standard output. Input and output can be redirected and piped in the usual fashion. For example,

```
csgen <../data/csgen.abc | more
```

runs the program csgen on the data file csgen.abc in the parallel subdirectory and pipes the output through more[1].

Many programs take command line arguments, which may be the names of files to process or options that select specific processing functions. An option is prefixed by a dash, sometimes followed by an argument. For example,

```
deal –h 5
```

runs the program deal with the option –h and the argument 5.

If a program is not called with the proper options or arguments, it generally terminates with an error message such as

```
usage: [ –h n ] [ –s n ]
```

which indicates the proper usage. Some programs provide more specific errors messages. Error messages are written to standard error output. Standard error output is always written to the console and cannot be redirected. Consult the descriptions of the programs that follow for details.

The programs that follow are divided into categories by their function.

1. Demonstrations and Games

1.1 Non-Attacking Queens: queens

This program displays the solutions to the non-attacking *n*-queens problem: the ways in which *n* queens can be placed on an *n*-by-*n* chessboard so that no queen can attack another. A positive integer can be given as a command line argument to specify the number of queens. For example,

```
iconx queens 8
```

displays the solutions for 8 queens on an 8-by-8 chessboard. The default value in the absence of an argument is 6. One solution for six queens is:

```
-------------------------
|   | Q |   |   |   |   |
-------------------------
|   |   |   | Q |   |   |
-------------------------
|   |   |   |   |   | Q |
-------------------------
| Q |   |   |   |   |   |
-------------------------
|   |   | Q |   |   |   |
-------------------------
|   |   |   |   | Q |   |
-------------------------
```

Comments: There are many approaches to programming solutions to the *n*-queens problem. This library program is worth reading for its programming techniques. Other solutions may be found in [1] and [5].

[1] Path syntax is system dependent. On some systems, including MS-DOS [3] and VMS [4], compiled programs cannot be run directly, but must be executed using iconx. For MS-DOS, the example above is done as follows:

```
iconx csgen <..\data\csgen.abc | more
```

1.2 Word Intersections: cross

This program takes a list of words and tries to arrange them in cross-word format so that they intersect. Uppercase letters are mapped into lowercase letters on input. For example, the input

```
and
eggplants
elephants
purple
```

produces the output

```
+---------+
| p       |
| u e     |
| r g     |
| p g     |
|elephants|
| e l     |
|   and   |
|   n     |
|   t     |
|   s     |
+---------+
```

Diagnostics: The program objects if the input contains a nonalphabetic character.

Comments: This program produces only one possible intersection and it does not attempt to produce the most compact result. The program is not very fast, either. There is a lot of room for improvement here. In particular, it is natural for Icon to generate a sequence of solutions.

1.3 Bridge Hands: deal

This program shuffles, deals, and displays hands in the game of bridge. An example of the output of deal is

```
---------------------------------

          S: KQ987
          H: 52
          D: T94
          C: T82

S: 3                S: JT4
H: T7               H: J9863
D: AKQ762           D: J85
C: QJ94             C: K7

          S: A652
          H: AKQ4
          D: 3
          C: A653

---------------------------------
```

Options: The following options are available:

–h *n* Produce *n* hands. The default is 1.

–s *n* Set the seed for random generation to *n*. Different seeds give different hands. The default seed is 0.

1.4 Farberisms: farb

Dave Farber, co-author of the original SNOBOL programming language, is noted for his creative use of the English language. Hence the terms ‘‘farberisms’’ and ‘‘to farberate’’. This program produces a randomly selected farberism.

Notes: Not all of the farberisms contained in this program were uttered by the master himself; others have learned to emulate him. A few of the farberisms may be objectionable to some persons. ‘‘I wouldn’t marry her with a twenty-foot pole.’’

2. Random Strings

The programs in this section involve the random generation of strings according to various criteria. These programs are only loosely related to each other.

2.1 Random Sentence Generation: rsg

This program generates randomly selected strings (‘‘sentences’’) from a grammar specified by the user. Grammars are basically context-free and resemble BNF in form, although there are a number of extensions.

The program works interactively, allowing the user to build, test, modify, and save grammars. Input to rsg consists of various kinds of specifications, which can be intermixed:

Productions define nonterminal symbols in a syntax similar to the rewriting rules of BNF with various alternatives consisting of the concatenation of nonterminal and terminal symbols. *Generation specifications* cause the generation of a specified number of sentences from the language defined by a given nonterminal symbol. *Grammar output specifications* cause the definition of a specified nonterminal or the entire current grammar to be written to a given file. *Source specifications* cause subsequent input to be read from a specified file.

In addition, any line beginning with # is considered to be a comment, while any line beginning with = causes the rest of that line to be used subsequently as a prompt to the user whenever rsg is ready for input (there normally is no prompt). A line consisting of a single = stops prompting.

Productions: Examples of productions are:

```
<expr>::=<term>|<term>+<expr>
<term>::=<elem>|<elem>*<term>
<elem>::=x|y|z|(<expr>)
```

Productions may occur in any order. The definition for a nonterminal symbol can be changed by specifying a new production for it.

There are a number of special devices to facilitate the definition of grammars, including eight predefined, built-in nonterminal symbols:

symbol	definition
<lb>	<
<rb>	>
<vb>	|
<nl>	newline
<>	empty string
<&lcase>	any single lowercase letter
<&ucase>	any single uppercase letter
<&digit>	any single digit

In addition, if the string between a < and a > begins and ends with a single quotation mark, it stands for any single character between the quotation marks. For example,

```
<'xyz'>
```

is equivalent to

```
x|y|z
```

Finally, if the name of a nonterminal symbol between the < and > begins with ?, the user is queried during generation to supply a string for that nonterminal symbol. For example, in

```
<expr>::=<?term>|<term>+<expr>
```

if the first alternative is encountered during generation, the user is asked to provide a string for <term>. Note that this is a *strongly* context-sensitive feature.

Generation Specifications: A generation specification consists of a nonterminal symbol followed by a nonnegative integer. An example is

```
<expr>10
```

which specifies the generation of 10 <expr>s. If the integer is omitted, it is assumed to be 1. Generated sentences are written to standard output.

Grammar Output Specifications: A grammar output specification consists of a nonterminal symbol, followed by –>, followed by a file name. Such a specification causes the current definition of the nonterminal symbol to be written to the given file. If the file is omitted, standard output is assumed. If the nonterminal symbol is omitted, the entire grammar is written out. Thus,

```
–>
```

causes the entire grammar to be written to standard output.

Source Specifications: A source specification consists of @ followed by a file name. Subsequent input is read from that file. When an end of file is encountered, input reverts to the previous file. Input files can be nested.

Options: The following options are available:

–s *n* Set the seed for random generation to *n*. The default seed is 0.

–l *n* Terminate generation if the number of symbols remaining to be processed exceeds *n*. There is no default limit.

–t Trace the generation of sentences. Trace output goes to standard error output.

Diagnostics: Syntactically erroneous input lines are noted but are otherwise ignored. Specifications for a file that cannot be opened are noted and treated as erroneous.

If an undefined nonterminal symbol is encountered during generation, an error message that identifies the undefined symbol is produced, followed by the partial sentence generated to that point. Exceeding the limit of symbols remaining to be generated as specified by the –l option is handled similarly.

Caveats: Generation may fail to terminate because of a loop in the rewriting rules or, more seriously, because of the progressive accumulation of nonterminal symbols. The latter problem can be identified by using the –t option and controlled by using the –l option. The problem often can be circumvented by duplicating alternatives that lead to fewer rather than more nonterminal symbols. For example, changing

```
<term>::=<elem>|<elem>*<term>
```

to

```
<term>::=<elem>|<elem>|<elem>*<term>
```

increases the probability of selecting <elem> from 1/2 to 2/3. See [6] for a discussion of the general problem.

Comments: This program is an extension and elaboration of a program described in some detail in [1]. It illustrates many features of Icon, including a combination of string and list processing as well as extensive use of generators. The source code is worth studying.

There are many possible extensions to the program. One of the most useful would be a way to specify the probability of selecting an alternative.

2.2 Context-Sensitive Generation: csgen

This program accepts a context-sensitive production grammar and generates randomly selected sentences from the corresponding language. See [7] for a discussion of such grammars.

Uppercase letters stand for nonterminal symbols and –> indicates the lefthand side can be rewritten by the right-hand side. Other characters are considered to be terminal symbols. Lines beginning with # are considered to be comments and are ignored. A line consisting of a nonterminal symbol followed by a colon and a nonnegative integer i is a generation specification for *i* instances of sentences for the language defined by the nonterminal (goal) symbol. An example of input to csgen is:

```
#    a(n)b(n)c(n)
#    Salomaa, p. 11.
#    Attributed to M. Soittola.
#
X–>abc
X–>aYbc
Yb–>bY
Yc–>Zbcc
bZ–>Zb
aZ–>aaY
aZ–>aa
X:10
```

The output of csgen for this example is

```
aaabbbccc
aaaaaaaaabbbbbbbbbccccccccc
abc
aabbcc
aabbcc
aaabbbccc
aabbcc
abc
aaaabbbbcccc
aaabbbccc
```

A positive integer followed by a colon can be prefixed to a production to replicate that production, making its selection more likely. For example,

```
3:X–>abc
```

is equivalent to

```
X–>abc
X–>abc
X–>abc
```

Option: The –t option writes a trace of the derivations to standard error output.

Limitations: Nonterminal symbols can only be represented by single uppercase letters, and there is no way to represent uppercase letters as terminal symbols.

There can be only one generation specification and it must appear as the last line of input.

Comments: Generation of context-sensitive strings is a slow process. It may not terminate, either because of a loop in the rewriting rules or because of the progressive accumulation of nonterminal symbols. The program avoids deadlock, in which there are no possible rewrites for a string in the derivation.

This program would be improved if the specification of nonterminal symbols were more general, as in rsg.

2.3 Parenthesis-Balanced Strings: parens

This program produces parenthesis-balanced strings in which the parentheses are randomly distributed.

Options: The following options are available:

- –b *n* Bound the length of the strings to *n* left and right parentheses each. The default is 10.
- –n *n* Produce *n* strings. The default is 10.
- –l *s* Use the string *s* for the left parenthesis. The default is (.
- –r *s* Use the string *s* for the right parenthesis. The default is) .
- –v Randomly vary the length of the strings between 0 and the bound. In the absence of this option, all strings are the exactly as long as the specified bound.

For example, the output for

```
parens –v –b 4 –l "begin " –r "end "
```

is

```
begin end
begin end begin end
begin begin end end begin end
begin end begin begin end end
begin end
begin begin end end
begin begin begin end end end
begin end begin begin end end
begin end begin end
begin begin end begin end begin end end
```

Comments: This program was motivated by the need for test data for error repair schemes for block-structured programming langauges. See [8]. A useful extension to this program would be some way of generating other text among the parentheses. In addition to the intended use of the program, it can produce a variety of interesting patterns, depending on the strings specified by –l and –r.

2.4 Shuffled Files: shuffile

This program writes a version of the input file with the lines shuffled. For example, the result of shuffling

```
          On the Future!–how it tells
          Of the rapture that impells
         To the swinging and the ringing
          Of the bells, bells, bells–
       Of the bells, bells, bells, bells,
                 Bells, bells, bells–
  To the rhyming and the chiming of the bells!
```

is

```
  To the rhyming and the chiming of the bells!
         To the swinging and the ringing
                 Bells, bells, bells–
          Of the bells, bells, bells–
          On the Future!–how it tells
       Of the bells, bells, bells, bells,
          Of the rapture that impells
```

Option: The option –s *n* sets the seed for random generation to *n*. The default seed is 0.

Limitation: This program stores the input file in memory and shuffles pointers to the lines; there must be enough memory available to store the entire file.

3. Text Tabulation

3.1 Character Tabulation: tablc

This program tabulates characters and lists each character and the number of times it occurs. Characters are written using Icon's escape conventions. Line termination characters and other control characters are included in the tabulation.

Options: The following options are available:

–a	Write the summary in alphabetical order of the characters. This is the default.
–n	Write the summary in numerical order of the counts.
–u	Write only the characters that occur just once.

3.2 Word Tabulation: tablw

This program tabulates words and lists number of times each word occurs. A word is defined to be a string of consecutive upper- and lowercase letters with at most one interior occurrence of a dash or apostrophe.

Options: The following options are available:

–a	Write the summary in alphabetical order of the words. This is the default.
–i	Ignore case distinctions among letters; uppercase letters are mapped into to corresponding lowercase letters on input. The default is to maintain case distinctions.
–n	Write the summary in numerical order of the counts.
–l *n*	Tabulate only words longer than *n* characters. The default is to tabulate all words.
–u	Write only the words that occur just once.

4. Mailing Labels

4.1 Produce Mailing Labels: labels

This program produces labels using coded information taken from the input file. In the input file, a line beginning with # is a label header. Subsequent lines up to the next header or end-of-file are accumulated and output so as to be centered horizontally and vertically on label forms. Lines beginning with ∗ are treated as comments and are ignored.

Options: The following options are available:

–c *n*	Print *n* copies of each label.
–s *s*	Select only those labels whose headers contain a character in *s*.
–t	Format for curved tape labels (the default is to format for rectangular mailing labels).
–w *n*	Limit line width to *n* characters. The default width is 40.
–l *n*	Limit the number of printed lines per label to *n*. The default is 8.
–d *n*	Limit the depth of the label to *n*. The default is 9 for rectangular labels and 12 for tape labels (–t).
–f	Print the first line of each selected entry instead of labels.

Options are processed from left to right. If the number of printed lines is set to a value that exceeds the depth of the label, the depth is set to the number of lines. If the depth is set to a value that is less than the number of printed lines, the number of printed lines is set to the depth. Note that the order in which these options are specified may affect the results.

Printing Labels: Label forms should be used with a pin-feed platen. For mailing labels, the carriage should be adjusted so that the first character is printed at the leftmost position on the label and so that the first line of the output is printed on the topmost line of the label. For curved tape labels, some experimentation may be required to get the text positioned properly.

Diagnostics: If the limits on line width or the number of lines per label are exceeded, a label with an error message is written to standard error output.

4.2 Zip Code Sorting: zipsort

This program sorts labels produced by labels in ascending order of their postal zip codes.

Option: The option –d *n* sets the number of lines per label to *n*. The default is 9. This value must agree with the value used to format the labels.

Zip Codes: The zip code must be the last nonblank string at the end of the label. It must consist of digits but may have an embedded dash for extended zip codes. If a label does not end with a legal zip code, it is placed after all labels with legal zip codes. In such a case, an error messages also is written to standard error output.

5. Laminated Files

5.1 Laminating Files: lam

This program laminates files named on the command line onto the standard output, producing a concatenation of corresponding lines from each file named. If the files are different lengths, empty lines are substituted for missing lines in the shorter files. A command line argument of the form – *s* causes the string *s* to be inserted between the concatenated file lines.

Each command line argument is placed in the output line at the point that it appears in the argument list. For example, lines from file1 and file2 can be laminated with a colon between each line from file1 and the corresponding line from file2 by the command

```
lam file1 –: file2
```

File names and strings may appear in any order in the argument list. If – is given for a file name, standard input is read at that point. If a file is named more than once, each of its lines will be duplicated on the output line, except that if standard input is named more than once, its lines will be read alternately. For example, each pair of lines from standard input can be joined onto one line with a space between them by the command

```
lam – "– " –
```

while the command

```
lam file1 "– " file1
```

replicates each line from file1.

5.2 Delaminating Files: delam

This program delaminates standard input into several output files according to the specified fields. It writes the fields in each line to the corresponding output files as individual lines. If no data occurs in the specified position for a given input line an empty output line is written. This insures that all output files contain the same number of lines as the input file.

If – is used for the input file, the standard input is read. If – is used as an output file name, the corresponding field is written to the standard output.

The fields are defined by a list of field specifications, separated by commas, colons, or semicolons, of the following form:

n	the character in column *n*
n–*m*	the characters in columns *n* through *m*
n+*m*	*m* characters beginning at column *n*

where the columns in a line are numbered from 1 to the length of the line.

The use of delam is illustrated by the following examples. The command

```
delam 1–10,5 x.txt y.txt
```

reads standard input and writes characters 1 through 10 to file x.txt and character 5 to file y.txt. The command

```
delam 10+5:1–10:1–10:80 mid x1 x2 end
```

writes characters 10 through 14 to mid, 1 through 10 to x1 and x2, and character 80 to end. The command

```
delam 1–80;1–80 – –
```

copies standard input to standard output, replicating the first eighty columns of each line twice.

5.3 Delaminating Files by Separators: delamc

This program delaminates standard input into several output files according to the separator characters specified by the string following the –t option. It writes the fields in each line to the corresponding output files as individual lines. If no data occurs in the specified position for a given input line an empty output line is written. This insures that all output files contain the same number of lines as the input file.

If – is used as an output file name, the corresponding field is written to the standard output. If the –t option is not used, an ascii horizontal tab character is assumed as the default field separator.

The use of delamc is illustrated by the following examples. The command

```
delamc labels opcodes operands
```

writes the fields of standard input, each of which is separated by a tab character, to the output files labels, opcodes, and operands. The command

```
delamc –t: scores names matric ps1 ps2 ps3
```

writes the fields of standard input, each of which are separated by a colon, to the indicated output files. The command

```
delamc –t,: oldata f1 f2
```

separates the fields using either a comma or a colon.

6. Icon Program Utilities

6.1 Icon Program Cross Reference: ipxref

This program cross-references Icon programs. It lists the occurrences of each variable by line number. Variables are listed by procedure or separately as globals. The options specify the formatting of the output and whether or not to cross-reference quoted strings and non-alphanumerics. Variables that are followed by a left parenthesis are listed with an asterisk following the name. If a file is not specified, then standard input is cross-referenced.

Options: The following options change the format defaults:

–c *n*	The column width per line number. The default is 4 columns wide.
–l *n*	The starting column (i.e. left margin) of the line numbers. The default is column 40.
–w *n*	The column width of the whole output line. The default is 80 columns wide.

Normally only alphanumerics are cross-referenced. These options expand what is considered:

–q	Include quoted strings.
–x	Include all non-alphanumerics.

Note: This program assumes the subject file is a valid Icon program. For example, quotes are expected to be matched.

6.2 Sort Icon Declarations: ipsort

This program reads an Icon program and writes an equivalent program with the procedures sorted alphabetically. Global, link, and record declarations come first in the order they appear in the original program. The main procedure comes next followed by the remaining procedures in alphabetical order.

Comments and white space between declarations are attached to the next following declaration.

Limitations: This program only recognizes declarations that start at the beginning of a line.

Comments and interline white space between declarations may not come out as intended.

6.3 Icon Program Splitting: ipsplit

This progam reads an Icon program and writes each procedure to a separate file. The output file names consist of the procedure name with .icn appended. If the –g option is specified, any global, link, and record declarations are written to that file. Otherwise they are written in the file for the procedure that immediately follows them.

Comments and white space between declarations are attached to the next following declaration.

Notes: The program only recognizes declarations that start at the beginning of lines. Comments and interline white space between declarations may not come out as intended.

If the –g option is not specified, any global, link, or record declarations that follow the last procedure are discarded.

7. Miscellaneous Utilities

7.1 Line Lengths: ll

This program prints the lengths of the shortest and longest lines in files named on the command line. If there is no command line argument, the standard input is used. The argument – may be used to explicitly specify the standard input.

7.2 Trimming Lines: trim

This program copies lines from standard input to standard output, truncating the lines at *n* characters and removing any trailing blanks. The default value for *n* is 80. For example,

```
trim 70 <grade.txt >grade.fix
```

copies grade.txt to grade.fix, with lines longer than 70 characters truncated to 70 characters and the trailing blanks removed from all lines.

The –f option causes all lines to be *n* characters long by adding blanks to short lines; otherwise, short lines are left as is.

7.3 Sorting Groups of Lines: grpsort

This program sorts input containing ‘‘records’’ defined to be groups of consecutive lines. Output is written to standard output. Each input record is separated by one or more repetitions of a demarcation line (a line beginning with the separator string). The first line of each record is used as the key.

If no separator string is specified on the command line, the default is the empty string. Because all input lines are trimmed of whitespace (blanks and tabs), empty lines are default demarcation lines. The separator string specified can be an initial substring of the string used to demarcate lines, in which case the resulting partition of the input file may be different from a partition created using the entire demarcation string.

The –o option sorts the input file but does not produce the sorted records. Instead it lists the keys (in sorted order) and line numbers defining the extent of the record associated with each key.

The use of grpsort is illustrated by the following examples. The command

```
grpsort "catscats" <x >y
```

sorts the file x, whose records are separated by lines containing the string "catscats", into the file y placing a single line of "catscats" between each output record. Similarly, the command

```
grpsort "cats" <x >y
```

sorts the file x as before but assumes that any line beginning with the string "cats" delimits a new record. This may or may not divide the lines of the input file into a number of records different from the previous example. In any case, the output records will be separated by a single line of "cats". Another example is

```
grpsort –o <bibliography >bibkeys
```

which sorts the file bibliography and produces a sorted list of the keys and the extents of the associated records in bibkeys. Each output key line is of the form:

```
[s–e] key
```

where

s	is the line number of the key line
e	is the line number of the last line
key	is the actual key of the record

Introduction

Collections of translated procedures are in the distribution directory procs. These files can be linked into other programs. For example, if the Icon program library resides in /usr/icon/v6/ipl, the procedures in gener.icn can be linked with an Icon program by using the link declaration [2]:

```
link "/usr/icon/v6/ipl/procs/gener"
```

The IPATH environment variable [2] can be used to have the directory containing the translated procedures searched automatically. For example, if IPATH is set to /usr/icon/v6/ipl/procs, the link declaration need only be

```
link gener
```

1. Math Procedures: math

The following procedures compute standard trigonometric functions. The arguments are in radians.

sin(x)	sine of x
cos(x)	cosine of x
tan(x)	tangent of x
asin(x)	arc sine of x in the range $-\pi/2$ to $\pi/2$
acos(x)	arc cosine of x in the range 0 to π
atan(x)	arc tangent of x in the range $-\pi/2$ to $\pi/2$
atan2(y,x)	arc tangent of x/y in the range $-\pi$ to π

The following procedures convert from degrees to radians and conversely:

dtor(d)	radian equivalent of d
rtod(r)	degree equivalent of r

The following additional procedures are available:

sqrt(x)	square root of x
exp(x)	exponential function of x
log(x)	natural logarithm of x
log10(x)	base-10 logarithm of x
floor(x)	largest integer not greater than x
ceil(x)	smallest integer nor less than x

Failure Conditions: asin(x) and acos(x) fail if the absolute value of x is greater than one. sqrt(x), log(x), and log10(x) fail if x is less than zero.

2. Bit Operations: bitops

The following procedures perform operations on characters strings of zeros and ones (‘‘bit strings’’).

and(b1, b2)	logical ‘‘and’’ of b1 and b2
bitstring(i)	convert integer i to bit string
bsum(b1, b2)	arithmetic sum of b1 and b2 (used by other procedures)
decimal(b)	convert b to integer

[2] In MS-DOS, backslashes can be used in place of slashes in such link declarations, but they must be escaped, as in

```
link "\\usr\\icon\\v6\\ipl\\procs\\gener"
```

exor(b1, b2)	‘‘exclusive-or’’ of b1 and b2
neg(b)	negation of b
or(b1, b2)	logical ‘‘or’’ of b1 and b2

Note: If i in bitstring(i) is negative, the value produced is the corresponding unsigned 32-bit bit string.

Bugs: Integer values that exceed those allowable in Icon may produce bogus results or spurious diagnostics.

3. Radix Conversions: radcon

The following procedures convert numbers from one radix to another. The letters from a to z are used for ‘‘digits’’ greater than 9. All the conversion procedures fail if the conversion cannot be made.

exbase10(i, j)	convert base-10 integer i to base j
inbase10(s, i)	convert base-i integer s to base 10
radcon(s, i, j)	convert base-i integer s to base j

Limitation: The maximum base allowed is 36.

4. Complex Arithmetic: complex

The following procedures perform operations on complex numbers.

complex(r, i)	create complex number with real part r and imaginary part i
cpxadd(x1, x2)	add complex numbers x1 and x2
cpxdiv(x1, x2)	divide complex number x1 by complex number x2
cpxmul(x1, x2)	multiply complex number x1 by complex number x2
cpxsub(x1, x2)	subtract complex number x2 from complex number x1
cpxstr(x)	convert complex number x to string representation
strcpx(s)	convert string representation s of complex number to complex number

5. Collated Strings: collate

These procedures collate (interleave) respective characters of two strings and decollate such strings by selecting every other character of a string. produce a string consisting of interleaved characters of s1 and s2.

collate(s1, s2) collate the characters of s1 and s2. For example,

```
collate("abc", "def")
```

produces "adbecf".

decollate(s, i) produce a string consisting of every other character of s. If i is odd, the odd-numbered characters are selected, while if i is even, the even-numbered characters are selected.

Diagnostics: Run-time error 208 occurs if the arguments to collate are not of the same size.

6. Emphasized Text: bold

These procedures produce text with interspersed characters suitable for printing to produce the effect of boldface (by overstriking) and underscoring (using backspaces).

bold(s)	bold version of s
uscore(s)	underscored version of s

7. Shuffling: shuffle

The procedure shuffle(x) shuffles a string or list. In the case that x is a string, a corresponding string with the characters randomly rearranged is produced. In the case that x is a list, the values in the list are randomly rearranged.

8. Segmented Strings: segment

The procedure segment(s, c) generates consecutive substrings of s consisting of characters that respectively do/do not occur in c. For example,

```
segment("Not a sentence.", &lcase ++ &ucase)
```

generates

```
"Not"
" "
"a"
" "
"sentence"
"."
```

9. String Utilities: strutil

These procedures perform simple operations on strings.

compress(s, c)	compress consecutive occurrences of characters in c that occur in s
delete(s, c)	delete all occurrences of characters in c that occur in s
rotate(s, i)	rotate s i characters to the left (negative i produces rotation to the right); the default value of i is 1

10. Structure Utilities: structs

These procedures manipulate trees and acyclic graphs (dags). The structures are represented with lists. See [1].

depth(t)	compute maximum depth of tree t
eq(x, y)	compare list structures x and y
ldag(s)	construct a dag from the string s
ltree(s)	construct a tree from the string s
stree(t)	construct a string from the tree t
tcopy(t)	copy tree t
teq(t1, t2)	compare trees t1 and t2
visit(t)	visit, in preorder, the nodes of the tree t

Note: The procedure ldag has a second argument that is used on internal recursive calls; a second argument must not be supplied by the user.

11. Icon Literal Escapes: escape

The procedure escape(s) produces a string in which Icon quoted literal escape conventions in s are replaced by the corresponding characters. For example, escape("\\143\\141\\164") produces the string "cat".

12. Images of Icon Values: image

The procedure Image(x) produces a string image of the value x. The value produced is a generalization of the value produced by the Icon function image(x), providing detailed information about structures.

Tags are used to uniquely identify structures. A tag consists of a letter identifying the type followed by an integer. The tag letters are L for lists, R for records, S for sets, and T for tables. The first time a structure is encountered, it is imaged as the tag followed by a colon, followed by a representation of the structure. If the same structure is encountered again, only the tag is given.

An example is

```
a := ["x"]
push(a, a)
t := table()
push(a, t)
t[a] := t
t["x"] := []
t[t] := a
write(Image(t))
```

which produces

```
T1:["x"–>L1:[], L2:[T1, L2, "x"]–>T1, T1–>L2]
```

Note that a table is represented as a list of entry and assigned values separated by –>.

13. List Mapping: lmap

The procedure lmap(a1, a2, a3) maps elements of a1 according to a2 and a3. This procedure is the analog for lists of the built-in string-mapping function map(s1, s2, s3). Elements in a1 that are the same as elements in a2 are mapped into the corresponding elements of a3. For example, given the lists

```
a1 := [1, 2, 3, 4]
a2 := [4, 3, 2, 1]
a3 := ["a", "b", "c", "d"]
```

then

```
lmap(a1, a2, a3)
```

changes a1 to

```
["d", "c", "b", "a"]
```

Note that the value of a1 is modified.

Lists that are mapped can have any kinds of elements. The operation

```
x === y
```

is used to determine if elements x and y are equivalent.

All cases in lmap are handled as they are in map, except that no defaults are provided for omitted arguments. As with map, lmap can be used for transposition as well as substitution.

Warning: If lmap is called with the same lists a2 and a3 as in the immediately preceding call, the same mapping is performed, even if the values in a2 and a3 have been changed. This improves performance, but it may cause unexpected effects.

Comments: It is easy to change lmap to produce a new list instead of modifying a1; this is a good exercise for beginning Icon programmers. The ‘‘caching’’ of the mapping table based on a2 and a3 also can be removed easily to avoid the potential problem mentioned in the warning above.

14. Snapshots of Scanning: snapshot

The procedure snapshot() writes a snapshot of the state of string scanning, showing the value of &subject and &pos. For example,

```
"((a+b)–delta)/(c*d))" ? {
    tab(bal('+–/*'))
    snapshot()
    }
```

produces

```
-------------------------------------
|                                   |
| &subject = "((a+b)-delta)/(c*d))" |
|                          |        |
-------------------------------------
```

Note that the bar showing the &pos is positioned under the &posth character (actual positions are between characters). If &pos is at the end of &subject, the bar is positioned under the quotation mark delimiting the subject. For example,

```
"abcdefgh" ? (tab(0) & snapshot())
```

produces

```
-------------------------
|                       |
| &subject = "abcdefgh" |
|                     | |
-------------------------
```

Escape sequences are handled properly. For example,

```
"abc\tdef\nghi" ? (tab(upto('\n')) & snapshot())
```

produces

```
------------------------------
|                            |
| &subject = "abc\tdef\nghi" |
|                     |      |
------------------------------
```

15. Miscellaneous Generators: gener

These procedures generate sequences of results.

hex()	sequence of hexadecimal codes for numbers from 0 to 255
label(s, i)	sequence of labels with prefix s starting at i
octal()	sequence of octal codes for numbers from 0 to 255
star(s)	sequence consisting of the closure of s starting with the empty string and continuing in lexical order as given in s

16. Result Sequences: seqimage

The procedure Seqimage{e, i, j} produces a string image of the result sequence for the expression e. The first i results are printed. If i is omitted, there is no limit. If there are more than i results for e, ellipses are provided in the image after the first i. If j is specified, at most j results from the end of the sequence are printed after the ellipses. If j is omitted, only the first i results are produced.

For example, the expressions

```
Seqimage{1 to 12}
Seqimage{1 to 12,10}
Seqimage{1 to 12,6,3}
```

produce, respectively,

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...}
{1, 2, 3, 4, 5, 6, ..., 10, 11, 12}
```

Warning: If j is not omitted and e has a infinite result sequence, Seqimage does not terminate.

17. SNOBOL4 Pattern Matching: patterns

These procedures provide procedural equivalents for most SNOBOL4 patterns and some extensions. See [9-11]. Procedures and their pattern equivalents are:

Any(s)	ANY(S)
Arb()	ARB
Arbno(p)	ARBNO(P)
Arbx(i)	ARB(I)
Bal()	BAL
Break(s)	BREAK(S)
Breakx(s)	BREAKX(S)
Cat(p1, p2)	P1 P2
Discard(p)	/P
Exog(s)	\S
Find(s)	FIND(S)
Len(i)	LEN(I)
Limit(p, i)	P \ i
Locate(p)	LOCATE(P)
Marb()	longest-first ARB
Notany(s)	NOTANY(S)
Pos(i)	POS(I)
Replace(p, s)	P ≡ S
Rpos(i)	RPOS(I)
Rtab(i)	RTAB(I)
Span(s)	SPAN(S)
String(s)	S
Succeed()	SUCCEED
Tab(i)	TAB(I)
Xform(f, p)	F(P)

The following procedures relate to the application and control of pattern matching:

Apply(s, p)	S ? P

Mode()	anchored or unanchored matching (see Anchor and Float)
Anchor()	&ANCHOR = 1 if Mode := Anchor
Float()	&ANCHOR = 0 if Mode := Float

In addition to the procedures above, the following expressions can be used:

p1() | p2()	P1 | P2
v <– p()	P . V (approximate)
v := p()	P $ V (approximate)
fail	FAIL
=s	S (in place of String(s))
p1() || p2()	P1 P2 (in place of Cat(p1, p2))

Using this system, most SNOBOL4 patterns can be satisfactorily transliterated into Icon procedures and expressions. For example, the pattern

```
SPAN("0123456789") $ N "H" LEN(*N) $ LIT
```

can be transliterated into

```
(n <– Span('0123456789')) || ="H" ||
   (lit <– Len(n))
```

Concatenation of components is necessary to preserve the pattern-matching properties of SNOBOL4. See the documents referenced above for details and limitations.

Caveats: Simulating SNOBOL4 pattern matching using the procedures above is inefficient.

18. Defined Control Operations: pdco

These procedures use co-expressions to used to model the built-in control structures of Icon and also provide new ones. See [12].

Alt{e1, e2}	models e1 | e2
Colseq{e1, e2, …}	produces results of e1, e2, … alternately
Comseq{e1, e2}	compares result sequences of e1 and e2
Cond{e1, e2, …}	models the generalized Lisp conditional
Every{e1, e2}	models every e1 do e2
Galt{e1, e2, …}	models generalized alternation: e1 | e2 | …
Lcond{e1, e2, …}	models the Lisp conditional
Limit{e1, e2}	models e1 \ e2
Ranseq{e1, e2, …}	produces results of e1, e2, … at random
Repalt{e}	models |e
Resume{e1, e2, e3}	models every e1 \ e2 do e3
Select{e1, e2}	produces results from e1 by position according to e2

Comments: Because of the handling of the scope of local identifiers in co-expressions, expressions in programmer-defined control operations cannot communicate through local identifiers. Some constructions, such as break and return, cannot be used in arguments to programmer-defined control operations.

19. Defined Control Regimes: pdae

These procedures use co-expressions to model the built-in argument evaluation regime of Icon and also provide new ones. See [13].

Allpar{e1, e2, …}	parallel evaluation with last result used for short sequences
Extract{e1, e2, …}	extract results of even-numbered arguments according to odd-numbered values
Lifo{e1, e2, …}	models standard Icon ‘‘lifo’’ evaluation
Parallel{e1, e2, …}	parallel evaluation terminating on shortest sequence
Reverse{e1, e2, …}	left-to-right reversal of lifo evaluation
Rotate{e1, e2, …}	parallel evaluation with shorter sequences re-evaluated
Simple{e1, e2, …}	simple evaluation with only success or failure

Comments: Because of the handling of the scope of local identifiers in co-expressions, expressions in programmer-defined argument evaluation regimes cannot communicate through local identifiers. Some constructions, such as break and return, cannot be used in arguments to programmer-defined argument evaluation regimes.

At most 10 arguments can be used in the invocation of a programmer-defined argument evaluation regime. This limit can be increased by modifying Call, a utility procedure that is included.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
2. R. E. Griswold, W. H. Mitchell and J. O'Bagy, *Version 6 of Icon*, The Univ. of Arizona Tech. Rep. 86-10b, 1986.
3. R. E. Griswold, *Version 7.5 of Icon for MS-DOS*, The Univ. of Arizona Icon Project Document IPD47c, 1988.
4. G. M. Townsend, *Using Version 6 of Icon on VMS*, The Univ. of Arizona Icon Project Document IPD11, 1986.
5. R. E. Griswold, *Programming in Icon; Problems and Solutions from the Icon Newsletter*, The Univ. of Arizona Tech. Rep. 86-2a, 1986.
6. C. S. Wetherwell, ''Probabilistic Languages: A Review and Some Open Questions'', *Computing Surveys 12*, 4 (1980), 362-379.
7. A. Salomaa, *Formal Languages*, Academic Press, 1973.
8. D. B. Anderson and M. R. Sleep, ''Uniform Random Generation of Balanced Parenthesis Strings'', *ACM Trans. Prog. Lang. and Systems 2*, 1 (1980), 122-128.
9. R. E. Griswold, *Pattern Matching in Icon*, The Univ. of Arizona Tech. Rep. 80-25, 1980.
10. R. E. Griswold, *Models of String Pattern Matching*, The Univ. of Arizona Tech. Rep. 81-6, 1981.
11. A. C. Fleck, ''Formal Models for String Patterns'', in *Current Trends in Programming Methodology; Data Structuring*, vol. IV, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978, 216-240.
12. R. E. Griswold and M. Novak, ''Programmer-Defined Control Operations'', *Computer J. 26*, 2 (May 1983), 175-183.
13. M. Novak and R. E. Griswold, *Programmer-Defined Argument Evaluation Regimes*, The Univ. of Arizona Tech. Rep. 82-16, 1982.