Icon Observed Coding Laws and Standard Techniques (Icon OCLAST)

Cary A. Coutant

(Not issued formally as a technical report;
offered to readers of Icon Newsletter #5)

May 1979
Revised August 1980

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

Icon Observed Coding Laws and Standard Techniques
(Icon OCLAST)

This document describes a coding style for the Icon programming language. This style is not presented as the best or only way to write programs. Rather, it is a blend of commonly-used conventions (largely influenced by many Ratfor programs) that have proven to be practical, and thus is a suitable style for use in a group project.

1. Overall Program Structure

An Icon source file consists of a sequence of record, global, and procedure declarations. Record and global declarations should be grouped together at the beginning of the program, and should be separated from one another by one blank line. Procedure declarations should follow, each separated by two blank lines.

Record declarations should assume the following form:

```
record name(field1, field2)
```

Global variables should each be declared on a separate line, with a comment beside each explaining the use of the variable.

Procedure declarations should consist of two parts, a preamble of comments followed by the source code. The first line of the preamble should begin with two sharp signs (##), for both visual and automated aid in finding procedures, followed by the name of the procedure and its arguments, and a description of the procedure's function and method. Continuation lines should begin with a single sharp sign, and two blanks to align the left edge of the comment text.

Normally, the preamble for each procedure should contain all the comments necessary to understand that procedure. In cases where a tricky construction is used, or a comment in the code is deemed necessary, the comment should appear on the right side of the code. Comments should not appear intermixed with the code, unless a procedure consists of several logically independent steps, in which case a comment (preceded by a blank line) may identify the beginning of each step.

All local (and static) variables used in a procedure should be declared on one or more lines immediately following the procedure header. The declarations should be indented three spaces, and the word **local** or **static** should be followed by a comma-separated list of variable names. (There should be spaces after the commas in this list.) If the list is too long for one line, the continuation should be aligned underneath the first variable name in the list.

One blank line should be used after declarations within a procedure, and after an initial clause.

An example of a procedure appears below.

```
## gcd(n, m) — Compute greatest common divisor
#  of n and m, using Euclid's algorithm.

procedure gcd(n, m)
   local rem

   if (rem := n % m) = 0 then return m
   return gcd(m, rem)
end
```

The rest of this report describes the style for the procedure source code.

2. Indenting Conventions

In Icon, any arbitrary program structure may be nested within any control structure Without a consistent convention for indenting, a program quickly becomes unintelligible Any nested structure should be indented underneath the structure within which it occurs, and a structure should generally be thought of as continuing until a statement below it appears at the same indentation level Three or four spaces for indentation have proven to be good fewer spaces render the structure difficult to follow, more spaces (or tabs) produce overly long lines

When a compound expression is nested in a control structure, the braces surrounding the expression should be placed so that they do not complicate the code The opening brace should appear at the end of the line containing the governing control structure, each statement within the compound expression should begin on a separate line, indented the same amount The closing brace should appear on a line of its own indented the same amount as the body of the expression, since it is part of that expression

3. Formatting Control Structures

Each control structure should obey the rules of indenting as described above Nonetheless, there are several ways to arrange complex statements This section gives one or more forms for each Icon control structure

3.1 **if** expr1 **then** expr2 [**else** expr3]

If there is no **else** clause and **expr2** is short, the **if** structure may be written on one line

```
if expr1 then expr2
```

If the **else** clause is present, the structure can be treated as two separate statements:

```
if expr1 then expr2
else expr3
```

If either **expr1** or **expr2** is long or compound, the structure should be written on several lines

```
if expr1 then
   expr2
else {
   expr3

   }
```

Nested **else-if** constructions should never accumulate indentation, each **if** should be written immediately following the **else**. For example

```
if expr1 then
   expr2
else if expr3 then {
   expr4

   }
else if expr5 then {
   expr6

   }
```

Use of the **if** structure within an expression is possible, but an alternative style is usually clearer In particular,

```
if expr1 then
   a := expr2
else
   a := expr3
```

is preferable to

```
a := if expr1 then expr2 else expr3
```

In cases where the **if** structure is reasonably used in an expression, the entire expression should fit on one line. (If it will not fit nicely on one line, it is unreasonable.)

3.2 **while** expr1 **do** expr2 (**until** expr1 **do** expr2)

If expr2 is compound or long, the **while** or **until** structure should be written on several lines:

```
while expr1 do {
   expr2
   ...
   }
```

or

```
while expr1 do
   expr2
```

Otherwise, the structure may be written on one line as above.

3.3 **repeat** expr

If expr is compound, this structure should be written on several lines:

```
repeat {
   expr
   ...
   }
```

Otherwise, the structure may be written on one line.

3.4 **case** expr **of** { ... }

This structure should always be written on several lines. The case labels should appear stacked, indented the standard three spaces under the first line of the structure. If each case clause is a simple expression (and fits on one line), each clause should follow its label. The case clauses should line up vertically, with two spaces between the longest label and its clause. For example:

```
case expr of {
   label1:           clause1
   label2 | label3:  clause2
   label4:           clause3
   default:          clause4
   }
```

If any of the case clauses is compound, or if a long list of labels would push the case clauses too far right, each case clause should be indented below its label:

```
case expr of {
   label1 | label2 | label3 | label4:
      clause1
   label5: {
      clause2
      ...
      }
   }
```

For **case** structures with few labels, an **else-if** construction may prove clearer.

3.5 *scan* s *using* expr

This structure should be formatted like **while** (Section 3.2).

3.6 **initial** expr

This structure should be formatted like **repeat** (Section 3.3).

4. Continuation Conventions

When an expression is too long to fit on one line, it should be broken at an operator of low precedence so that the resulting halves of the expression are as similar as possible. For example, if condition3 of the following **if** structure does not fit,

```
if condition1 & condition2 | condition3 then
   expr1
```

it should be written as

```
if condition1 & condition2 |
  condition3 then
   expr1
```

A long expression is often a sequence of terms separated by one of the operators |, &, ||, or +. When this is the case, the most logical place to break the expression is after each occurrence of the operator:

```
if condition1 |
  condition2 |
  condition3 then
   expr1
```

When one of the terms is too long to fit on a single line, the term itself should be broken up similarly. For example:

```
if ((condition1 | condition2) &
   (condition4 | condition3 | condition5)) |
  condition6 then
   expr1
```

Note the use of parentheses and the extra indentation to help group the terms visually.

Procedure calls with several arguments are often candidates for splitting across lines. In this case, the final arguments should be aligned underneath the first argument. For example:

```
result := map(string1 || string2,
              &ucase || &lcase || "()<>[]{}",
              &lcase || &ucase || ")(><][}{") ||
          map(string3, &ucase, &lcase)
```

Note that the first map procedure call is, in this case, the first term of a concatenation. The second term is aligned underneath the first, as above.

5. Use of Blanks

Blanks normally should surround all binary operators that are not within subscripts, list constructors, or argument lists. Commas within these constructions should not be followed by blanks. For example:

```
i := i * r + d
day := calendar(month,date,"19"||year) || "day"
s[n+1][r] := s[n][r-1] - n * s[n][r]
res := expr("*", [2,x])
```

Blanks should, however, surround operators within vertically stacked list constructions or argument lists (see Section 4).

Appendix — Sample Program

```
record graph(nodes, arcs)


## topological sort program — reads a list of graphs from
#  standard input, and topologically sorts each one.  The
#  graphs are input on one line each, as follows:  All the
#  nodes in the graph are listed first (each node is
#  identified by a single character), followed by a colon,
#  followed by a list of arcs (each arc is a pair of
#  characters identifying the beginning and end of the arc).
#  The graph is represented internally in a record, whose
#  first field contains a cset of all the nodes, and whose
#  second field contains a string of all the arcs.

procedure main()
   local line, i, c

   while line := trim(read()) do {
      i := upto(":", line)
      g := graph(cset(line[1:i]), line[i+1:0])
      write()
      write(line)
      write(disg(g))
      write("sorted")
      write("   ", tsort(g) | "the graph has a cycle")
      }
end


## disg(g) — return a display of the graph g.
#  The arcs (the second element of the list) are returned using
#  mapg to format them, followed by the nodes (the first element)
#  which are not contained in any arcs.

procedure disg(g)
   local result, c

   result := mapg(g.arcs)
   every c := !(g.nodes -- g.arcs) do
      result := result || "   " || c || "\n"
   return result
end
```

```
## mapg(arcs) — map the arcs given as the argument into a string
#  with each arc on a newline, and inserting "->" as a frill.

procedure mapg(arcs)
   static image, object

   # Bootstrap the image and object strings used for displaying
   # the arcs of a graph.
   initial {
      object := "  1->2\n"
      image := "12"
      object := mapg(&lcase || &ucase)
      image := &lcase || &ucase
      }

   if size(arcs) <= size(image) then
      return map(object[1 +: 7*size(arcs)/2],
                 image[1 +: size(arcs)], arcs)
   return map(object, image, arcs[1 +: size(image)]) ||
         mapg(arcs[size(image)+1 : 0])
end


## tsort(g) — perform a topological sort on the graph g.
#  All nodes which have no predecessors are appended to the
#  result string, then all arcs from each of those nodes
#  are deleted.  This process is repeated until there are
#  no nodes left with no predecessors.  Hopefully, there
#  will be no arcs left then either, or the graph has a
#  cycle in it, in which case, tsort fails.

procedure tsort(g)
   local result, nodes, arcs, roots

   nodes := g.nodes
   arcs := g.arcs
   until null(roots := nodes -- snodes(arcs)) do {
      result := result || roots
      arcs := delarcs(arcs, roots)
      nodes := nodes -- roots
      }
   if null(arcs) then
      return result
   else fail
end
```

```
## snodes(arcs) — return all the even characters of arcs, which
#  really returns all the nodes which have predecessors.  The
#  cset is taken here since a cset of nodes is more meaningful
#  than a string.

procedure snodes(arcs)
   static dascii

   # Form a string of doubled characters from &ascii used for
   # extracting the even characters from a string.
   initial {
      every c := !&ascii do
         dascii := dascii || c || c
      }

   if size(arcs) <= size(dascii) then
      return cset(map(&ascii[1 +: size(arcs)/2],
                      dascii[1 +: size(arcs)], arcs))
   return cset(map(&ascii, dascii, arcs[1 +: size(dascii)]) ||
               snodes(arcs[size(dascii)+1:0]))
end


## delarcs(arcs, roots) — delete all arcs from the string arcs
#  which arc from a node contained in the cset roots.

procedure delarcs(arcs, roots)
   local result, i

   every i := 1 to size(arcs)-1 by 2 do
      if any(roots, arcs, i) fails then
         result := result || arcs[i +: 2]
   return result
end
```