The Icon Analyst

In-Depth Coverage of the Icon Programming Language

February 1994
Number 22

In this issue …

Procedures with Memory (continued)

In the last issue of the Analyst, we described how memory can be added to procedures to avoid unnecessary recomputation. As illustrated in that article, the savings in time can be dramatic. In this article, we'll consider the space requirements for procedures with memory and look at alternate formulations.

Space Requirements

It obviously takes space to provide the memory that saves redundant computations. Trade-offs between speed and space are common in computing. In the situation described here, the trade-offs are much in the favor of speed, but space is nonetheless a significant consideration.

The examples in the previous article used tables for memory. Tables provide the great advantage of flexibility. They can be subscripted with any kind of value, they grow automatically, and there's no worry about out-of-bounds subscripts.

As easy as tables are to use, however, they take more storage than the obvious alternative, lists. The exact amount of storage that tables and lists requires is complicated to determine and depends to some extent on the history of their use [1]. In both cases, however, the amount of space is dominated by the space for the elements they contain, so this is the important consideration for large tables and lists. A table element requires about 3.5 times the amount of space as a list element (the exact amount depends on the size of the table and how it is constructed).

There are several reasons for this. One is that in tables, keys must be stored along with the values. For lists, on the other hand, keys are just positions and are implicit. Another reason tables are larger than lists is that information is stored in tables to provide rapid look-up.

Whether or not a factor of 3.5 is important depends on the amount of memory a procedure needs and on the platform on which the program runs. On modern workstations, the amount of memory needed for the kinds of procedures we've described usually is not a significant consideration. For a personal computer with limited RAM, the amount of memory needed may be the limiting factor in what can be done.

We'll therefore consider how lists can be used in place of tables. The procedure to compute values in the chaotic sequence provides a good case to study. A version that uses tables is:

```
procedure q(i)
  static memory

  initial {
    memory := table()
    memory[1] := memory[2] := 1
    }

  return \memory[i] |
    (memory[i] := q(i – q(i – 1)) + q(i – q(i – 2)))

end
```

Back Issues

Back issues of The Icon Analyst are available for $5 each. This price includes shipping in the United States, Canada, and Mexico. Add $2 per order for airmail postage to other countries.

It's relatively easy to substitute a list for the table; nothing needs to be changed except the initialization:

```
initial {
   memory := list(1000)
   memory[1] := memory[2] := 1
   }
```

The problem with this is, of course, that the size of the list limits the values that can be computed. The size of 1,000 used in the example above, is, of course, somewhat arbitrary. If the size of the list is exceeded, \memory[i] fails because memory[i] fails, and the procedure calls fails. That may not be so bad; it even could be called a design decision. However, if the size of the list is not big enough for the intended use, the procedure must be modified to handle the problem, or the size of the list must be specified as an argument to the procedure.

Making the list as large as possible is not a good solution, since a list requires space for all its elements, even if they never are used. An obvious solution to these problems is to start with a list of modest size and increase its size if needed. It might seem like Icon's ability to add an element to a list would be the easiest thing to do, but there's no way to predict the argument with which the procedure is called — many additional elements might be required for any call, and putting them on the list one at a time is inefficient. And, if the list needs to be increased in size, it's probably worth making a substantial increase. An unimaginative but workable approach to list-expansion is:

```
procedure q(i)
   static memory, size, delta

   initial {
      size := 1000
      delta := 1000
      memory := list(size)
      memory[1] := memory[2] := 1
      }

   if i > size then {
      memory |||:= list(i – size + delta)
      size := *memory
      }

   return \memory[i] |
    (memory[i] := q(i – q(i – 1)) + q(i – q(i – 2)))

end
```

If i is greater than the current size, enough elements are appended to the list to provide for i as well as some "breathing room".

You might wonder about the relative speed of the table and list approaches to providing procedural memory. List look-up certainly is faster than table look-up, but by how much? And what's the penalty for having to check the list bound every time the procedure is called?

Here are timings on a Sparc IPX for computing q(1) through q(1000) for three cases: using a table, using a list without bounds checking, and using a list with bounds checking.

tables:	400 ms.
list without check:	366 ms.
list with check:	483 ms.

You may be surprised that table look-up is not much slower than list look-up. That's part of what you get in return for the larger size of table elements. And with bounds checking, using a list actually is slower than using a table. Actually, for the case timed here, the list is not increased in size. Presumably, the overall time to do that is small in any case, since it occurs infrequently.

Generators with Memory

In developing a procedure like this, it's easy to overlook a better approach. In the case of a sequence, values are most likely to be wanted in order. For this, a generator that is called once and repeatedly resumed is more natural formulation than a procedure that is called many times.

In the case off a generator, the amount of memory needed can be determined from an argument that limits the generation. Here's a version of the chaotic sequence:

```
procedure qgen(limit)
   local memory, i

   memory := list(limit, 1)

   suspend 1 | 1 | {
      memory[i := 3 to limit] :=
         memory[i – memory[i – 1]] +
            memory[i – memory[i – 2]])
      }

end
```

The correctness of this procedure relies on the fact that *q(i)* always depends on previously computed values in the sequence.

Thus, qgen(1000) generates the first 1,000 values in the chaotic sequence using a list with only one element more than the number actually needed. (Saving that one element is hardly worth the effort and computational time.)

The interesting thing about this formulation is how fast it is: Generating 1,000 values takes only 163 ms.

Procedures with Several Arguments

Adding memory to procedures that have more than one argument presents different problems. Consider the procedure for computing Ackermann's function as given in the previous article:

```
procedure a(i, j)

  if i = 0 then return j + 1
  else if j = 0 then return a(i – 1, 1)
  else return a(i – 1, a(i, j – 1))

end
```

If we really wanted to compute values Ackermann's function for values of *i* < 5, we could use the formulas given in the last article. We'll ignore that possibility here, since we want to illustrate how to add memory to procedures that we may not know to how compute more directly.

One way to add memory to such a procedure is to use a table of tables in the fashion that can be used to implement sparse arrays [2]:

```
procedure a(i, j)
  static memory

  initial memory := table()

  if i = 0 then return j + 1

  /memory[i] := table()

  if j = 0 then
    return \memory[i][j] |
      (memory[i][j] := a(i – 1, 1))
  else
    return \memory[i][j] |
      (memory[i][j] := a(i – 1, a(i, j – 1)))

end
```

A list of lists would require less space, but performing bounds checking and doing list expansion for such an approach would be time-consuming and messy. Since it's known that computing Ackermann's function for anything but small argument values of *i* is intractable, a list of tables can be used with little risk:

```
procedure a(i, j)
  static memory

  initial {
    memory := list(6)
    every !memory := table()
    }

  if i = 0 then return j + 1

  if j = 0 then
    return \memory[i][j] |
      (memory[i][j] := a(i – 1, 1))
  else
    return \memory[i][j] |
      (memory[i][j] := a(i – 1, a(i, j – 1)))

end
```

As with the alternative approaches to the implementation of sparse arrays, one option is to use a single table with a composite key composed from the two arguments:

```
procedure a(i, j)
  static memory
  local key

  initial memory := table()

  if i = 0 then return j + 1

  key :=  ...          # function of i and j

  if j = 0 then
    return \memory[key] |
      (memory[key] := a(i – 1, 1))
  else
    return \memory[key] |
      (memory[key] := a(i – 1, a(i, j – 1)))

end
```

A key, of course, must be a unique function of the arguments; every different pair of arguments must produce a different key. Some care is needed here.

Since it's impractical to compute *a(i, j)* for *i* > 6, *i* and *j* can be packed into a single integer, as in:

```
key := 8 * j + i
```

To be safe, the value of *i* should be checked to be sure it's not too large. Of course, *j* might be so large that multiplying it by 8 would cause integer overflow for implementations of Icon that don't support arbitrarily large integers.

A more general approach that does not depend on restrictions on the size of the values is concatenation, as in

```
key := i || "," || j
```

The separator can be any nonnull string that is not a digit.

This approach has the advantage of being very general. It can be used for many cases where arguments are not integers and for cases in which there are many arguments. An even more general approach can be used for cases where the arguments are strings that might contain the separator character, as in

```
key := image(s1) || "," || image(s2)
```

This works for many other types of arguments, such as lists, because `image(x)` produces a unique identification for `x` in almost all cases (do you know when it may not?)

Here are timings on a Sparc IPX for `a(3, 7)` for the methods described above:

no memory:	51400 ms.
table of tables:	516 ms.
list of tables:	466 ms.
integer keys:	500 ms.
concatenated keys:	533 ms.

The advantage of using memory is obvious. It's not so obvious which of the memory methods is best. Using a table of tables is simple, and only a little speed is gained by using a list of tables. It should not be surprising that integer keys are faster than concatenated keys, but the difference is less than you might expect.

Another factor that needs to be considered is the amount of storage allocated for the different forms of memory:

table of tables:	88284 bytes
list of tables:	88504 bytes
integer keys:	88408 bytes
concatenated keys:	98409 bytes

Except in the case of concatenated keys, all the storage allocated is for structures. The larger amount for concatenated keys is for string allocation. By the way, the timings given earlier do not include time for garbage collection; we set the region sizes large enough to avoid garbage collection. Nonetheless, that factor needs to be kept in mind.

Incidentally, there's an extra benefit in using memory for procedures like this: `a(3, 7)` goes to a depth of 355 in recursion without memory but only 260 with it.

Conclusions

As we mentioned at the beginning of the first article, the examples we've given, however interesting they may be, are not the kinds of things you're likely to encounter in your day-to-day programming.

You may, however, find situations in which a procedure is called repeatedly with the same arguments. These situations usually occur when a definition is recursive. Look particularly for cases where strings are defined recursively, as in a grammar, or where structures are defined recursively, as for trees.

You also may want to look at a related use of memory in procedures to assure that structures are built only once [3].

So as not to leave you hanging with the question about `image()`, it doesn't give a unique identification for files if two by the same name are open.

References

1. "Memory Utilization", Icon Analyst 4, pp. 7-10.

2. "Sparse Arrays", Icon Analyst 16, pp. 9-12.

3. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, pp. 193-195.

Color in X-Icon

Color is one of the most important and potentially rewarding components of computer graphics — and one of the most difficult. Color often is used just to make an application visually attractive. But color has many important uses beyond purely decorative ones: to attract attention to important events or situations, to distinguish between different kinds of objects, and so on.

Using color effectively requires much more than just a technical mastery of rendering color. There are difficult issues related to color vision, the human cognitive system, the psychology of color, and even artistic taste. We won't attempt to discuss these issues here, but if you're interested in digging deeper into such topics, see References 1-6.

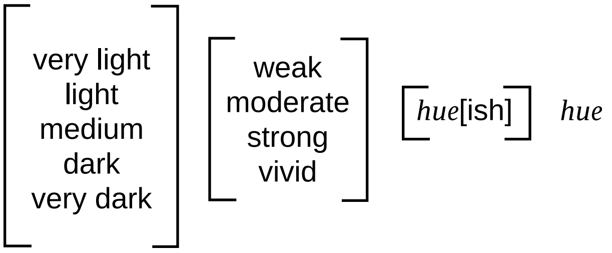
In what follows we're assuming hardware that supports a color display of at least a few colors; otherwise what follows is purely academic.

Specifying Colors

There are two ways you can specify colors in X-Icon: by name or by numerical specification.

Color Names: Icon supports a color naming system for the most commonly used colors. These names consists of simple English phrases that specify hue, lightness, and saturation values of the desired colors. Hue distinguishes among different colors, such as red, cyan, and purple**.** Lightness measures the perceived intensity of a color. Saturation is a measure of the purity of a color — how far it is from a gray of equal intensity. Bright red is highly saturated, while pink is comparatively unsaturated.

The syntax of a color name is



where choices enclosed in brackets are optional and *hue* can be one of black, gray, white, pink, violet, brown, red, orange, yellow, green, cyan, blue, purple, or magenta. A single hyphen or space separates each word from its neighbor.

Conventional English spelling is used. When adding ish to a hue ending in e, the e is dropped. For example, purple becomes purplish. The ish form of red is reddish. Some examples are

```
"dark–blue"
"very light greenish–blue"
"very dark purplish blue"
"vivid orange"
```

When two hues are supplied and the first hue has no ish suffix, the resulting hue is halfway between the two named hues. When a hue with an ish suffix precedes a hue, the resulting hue is three-fourths of the way from the ish hue to the main hue. The default lightness is medium and the default saturation is vivid.

Mixing radically different hues such as yellow and purple usually does not produce the expected result. It's also worth noting that the human perception of color varies widely, as do the actual colors produced by these names on different monitors. The program colrbook allows you to see what different color names produce. See Figure 1 on the next page.

If a color name does not conform to the naming system described above, the name is passed to the underlying window system, which may recognize other names.

Our X servers support many other color names, some of which are ones in common use, like chartreuse, pink, and maroon. Other names strike us as fanciful, not ones we'd think of on our own — names like papaya whip and gainsboro. (Note that color names can contain blanks.) There are a large number of grays — 115 distinct ones on our X server. These presumably are useful for grayscale monitors.

You might think that with all those names, you could easily find a color you want. Don't count on it. You may not find a name you'd expect. For example, our X server doesn't have a name for teal blue, which is one of our favorite colors. Perhaps more disturbing is the fact that the color you get for a color name may not be what you expect, or even close to it. There are numerous reasons for this, including differences in hardware and calibration.

If you're running X, you can experiment with its range of color names using the program colrname in the Icon program library. See Figure 2 on the next page. If you click on a color name, the background of the window turns to the corresponding color and the color name appears in the title of the window. (If you pick a color that's dark, you may not be able to read the names; if this

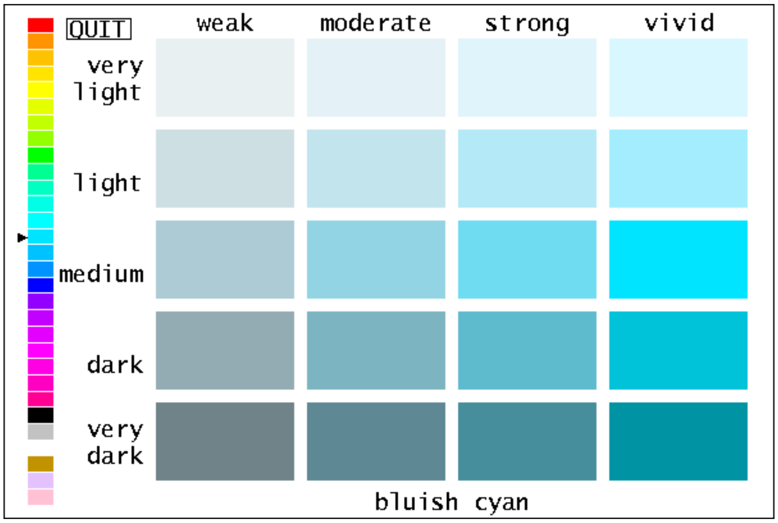


Figure 1. Standard Color Names

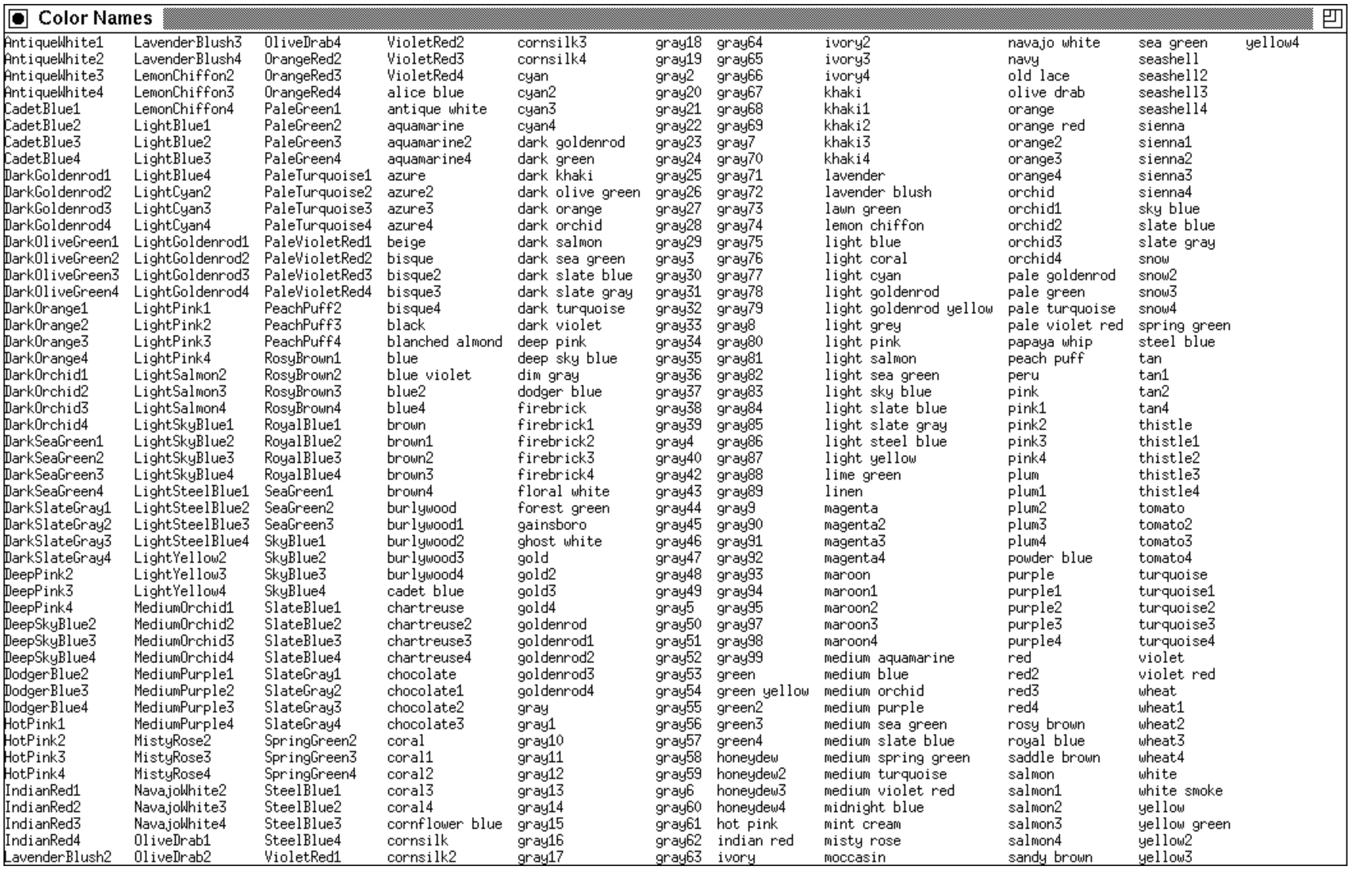
happens, just click here and there until you get a light color background.)

Numerical Specifications: Numerical specifications are given in terms of the red, green, and blue (RGB) components of light that are used to produce color on most modern monitors. Red, green, and blue in this context are primary *additive colors*. The intensity of the red, green, and blue components determines the color. At zero intensity for all components, the color produced is black — at least in theory; most monitor screens don't appear to be completely black in the absence of any illumination. At maximum intensity for all components, the color produced is white — again, in theory; the screens of most monitors appear to be light gray when fully illuminated. And, in general, equal intensities of all components produce a shade of gray.

Unequal intensities of the primaries produce other colors. For example, red and blue in the absence of green produce magenta, while red and green produce yellow, and blue and green produce cyan. All other colors are produced by combinations of the primaries in various intensities.

The advantage of the RGB color-specification system is that it corresponds directly to the hardware that produces the color. The RGB system has several disadvantages, however.

One disadvantage is that most persons learn



Color Names

```
AntiqueWhite1   LavenderBlush3   OliveDrab4      VioletRed2       cornsilk3          gray18  gray64        ivory2                 navajo white       sea green     yellow4
AntiqueWhite2   LavenderBlush4   OrangeRed2      VioletRed3       cornsilk4          gray19  gray65        ivory3                 navy               seashell
AntiqueWhite3   LemonChiffon2    OrangeRed3      VioletRed4       cyan               gray2   gray66        ivory4                 old lace           seashell2
AntiqueWhite4   LemonChiffon3    OrangeRed4      alice blue       cyan2              gray20  gray67        khaki                  olive drab         seashell3
CadetBlue1      LemonChiffon4    PaleGreen1      antique white    cyan3              gray21  gray68        khaki1                 orange             seashell4
CadetBlue2      LightBlue1       PaleGreen2      aquamarine       cyan4              gray22  gray69        khaki2                 orange red         sienna
CadetBlue3      LightBlue2       PaleGreen3      aquamarine2      dark goldenrod     gray23  gray7         khaki3                 orange2            sienna1
CadetBlue4      LightBlue3       PaleGreen4      aquamarine4      dark green         gray24  gray70        khaki4                 orange3            sienna2
DarkGoldenrod1  LightBlue4       PaleTurquoise1  azure            dark khaki         gray25  gray71        lavender               orange4            sienna3
DarkGoldenrod2  LightCyan2       PaleTurquoise2  azure2           dark olive green   gray26  gray72        lavender blush         orchid             sienna4
DarkGoldenrod3  LightCyan3       PaleTurquoise3  azure3           dark orange        gray27  gray73        lawn green             orchid1            sky blue
DarkGoldenrod4  LightCyan4       PaleTurquoise4  azure4           dark orchid        gray28  gray74        lemon chiffon          orchid2            slate blue
DarkOliveGreen1 LightGoldenrod1  PaleVioletRed1  beige            dark salmon        gray29  gray75        light blue             orchid3            slate gray
DarkOliveGreen2 LightGoldenrod2  PaleVioletRed2  bisque           dark sea green     gray3   gray76        light coral            orchid4            snow
DarkOliveGreen3 LightGoldenrod3  PaleVioletRed3  bisque2          dark slate blue    gray30  gray77        light cyan             pale goldenrod     snow2
DarkOliveGreen4 LightGoldenrod4  PaleVioletRed4  bisque3          dark slate gray    gray31  gray78        light goldenrod        pale green         snow3
DarkOrange1     LightPink1       PeachPuff2      bisque4          dark turquoise     gray32  gray79        light goldenrod yellow pale turquoise     snow4
DarkOrange2     LightPink2       PeachPuff3      black            dark violet        gray33  gray8         light grey             pale violet red    spring green
DarkOrange3     LightPink3       PeachPuff4      blanched almond  deep pink          gray34  gray80        light pink             papaya whip        steel blue
DarkOrange4     LightPink4       RosyBrown1      blue             deep sky blue      gray35  gray81        light salmon           peach puff         tan
DarkOrchid1     LightSalmon2     RosyBrown2      blue violet      dim gray           gray36  gray82        light sea green        peru               tan1
DarkOrchid2     LightSalmon3     RosyBrown3      blue2            dodger blue        gray37  gray83        light sky blue         pink               tan2
DarkOrchid3     LightSalmon4     RosyBrown4      blue4            firebrick          gray38  gray84        light slate blue       pink1              tan4
DarkOrchid4     LightSkyBlue1    RoyalBlue1      brown            firebrick1         gray39  gray85        light slate gray       pink2              thistle
DarkSeaGreen1   LightSkyBlue2    RoyalBlue2      brown1           firebrick2         gray4   gray86        light steel blue       pink3              thistle1
DarkSeaGreen2   LightSkyBlue3    RoyalBlue3      brown2           firebrick3         gray40  gray87        light yellow           pink4              thistle2
DarkSeaGreen3   LightSkyBlue4    RoyalBlue4      brown3           firebrick4         gray42  gray88        lime green             plum               thistle3
DarkSeaGreen4   LightSteelBlue1  SeaGreen1       brown4           floral white       gray43  gray89        linen                  plum1              thistle4
DarkSlateGray1  LightSteelBlue2  SeaGreen2       burlywood        forest green       gray44  gray9         magenta                plum2              tomato
DarkSlateGray2  LightSteelBlue3  SeaGreen3       burlywood1       gainsboro          gray45  gray90        magenta2               plum3              tomato2
DarkSlateGray3  LightSteelBlue4  SkyBlue1        burlywood2       ghost white        gray46  gray91        magenta3               plum4              tomato3
DarkSlateGray4  LightYellow2     SkyBlue2        burlywood3       gold               gray47  gray92        magenta4               powder blue        tomato4
DeepPink2       LightYellow3     SkyBlue3        burlywood4       gold2              gray48  gray93        maroon                 purple             turquoise
DeepPink3       LightYellow4     SkyBlue4        cadet blue       gold3              gray49  gray94        maroon1                purple1            turquoise1
DeepPink4       MediumOrchid1    SlateBlue1      chartreuse       gold4              gray5   gray95        maroon2                purple2            turquoise2
DeepSkyBlue2    MediumOrchid2    SlateBlue2      chartreuse2      goldenrod          gray50  gray97        maroon3                purple3            turquoise3
DeepSkyBlue3    MediumOrchid3    SlateBlue3      chartreuse3      goldenrod1         gray51  gray98        maroon4                purple4            turquoise4
DeepSkyBlue4    MediumOrchid4    SlateBlue4      chartreuse4      goldenrod2         gray52  gray99        medium aquamarine      red                violet
DodgerBlue2     MediumPurple1    SlateGray1      chocolate        goldenrod3         gray53  green         medium blue            red2               violet red
DodgerBlue3     MediumPurple2    SlateGray2      chocolate1       goldenrod4         gray54  green yellow  medium orchid          red3               wheat
DodgerBlue4     MediumPurple3    SlateGray3      chocolate2       gray               gray55  green2        medium purple          red4               wheat1
HotPink1        MediumPurple4    SlateGray4      chocolate3       gray1              gray56  green3        medium sea green       rosy brown         wheat2
HotPink2        MistyRose2       SpringGreen2    coral            gray10             gray57  green4        medium slate blue      royal blue         wheat3
HotPink3        MistyRose3       SpringGreen3    coral1           gray11             gray58  honeydew      medium spring green    saddle brown       wheat4
HotPink4        MistyRose4       SpringGreen4    coral2           gray12             gray59  honeydew2     medium turquoise       salmon             white
IndianRed1      NavajoWhite2     SteelBlue1      coral3           gray13             gray6   honeydew3     medium violet red      salmon1            white smoke
IndianRed2      NavajoWhite3     SteelBlue2      coral4           gray14             gray60  honeydew4     midnight blue          salmon2            yellow
IndianRed3      NavajoWhite4     SteelBlue3      cornflower blue  gray15             gray61  hot pink      mint cream             salmon3            yellow green
IndianRed4      OliveDrab1       SteelBlue4      cornsilk         gray16             gray62  indian red    misty rose             salmon4            yellow2
LavenderBlush2  OliveDrab2       VioletRed1      cornsilk2        gray17             gray63  ivory         moccasin               sandy brown        yellow3
```

Figure 2. X Color Names

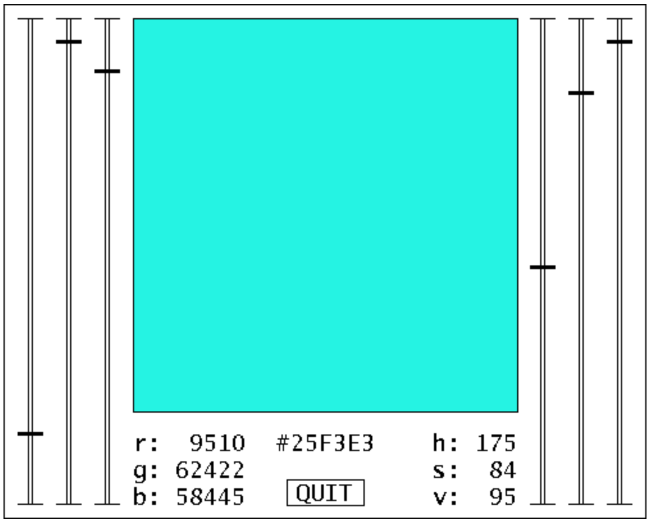
colors with a subtractive model in which a combination of pigments produces a darker color, not a lighter one. Persons used to thinking in terms of subtractive colors usually are surprised to learn that additive primaries red and green produce yellow. Another problem with the RGB system is that it isn't particularly intuitive. Unless you have experience with the RGB system, it may not be obvious to you how to get an orange color by combining red, green, and blue light. And how would you get teal blue?

In Icon, the intensities of red, green, and blue can be specified in several ways. Strings of the form "#*rgb*", "#*rrggbb*", "#*rrrgggbbb*", and "#*rrrrggggbbbb*" specify intensities in which *r*, *g* and *b* are hexadecimal digits. The more digits used, the more precisely a color can be specified. The specification "#DDD" might suffice for a light gray, but to get pink, something like "#FFC0CB" is needed.

Intensities also can be specified by comma-separated decimal values in the range from 0 to 65535. For example, "32000,0,0" specifies a dark red (less than half the maximum intensity for red, and no other primary).

As these ranges suggest, Icon maintains 16 bits of information for color intensities. The human visual system doesn't approach this degree of precision in discriminating among colors, so small variations in numerical specifications usually are unnoticeable.

There are other color models that are more intuitive and that can be translated to RGB. One model that works well for most artistically trained persons is the hue-saturation-value (HSV) model. Hue distinguishes among different colors, such as red, cyan, and purple. Hue usually is measured in degrees around a color circle; red is at 0°, green at 120°, and blue at 240°. Saturation is a measure of how far a color is from a gray of equal intensity. Bright red is highly saturated, while pink is comparatively unsaturated. Value measures the perceived intensity of a color. Saturation and value are measured on a scale of 0 to 100.



```
r:  9510  #25F3E3   h: 175
g: 62422            s:  84
b: 58445            v:  95
```

Figure 3. hsvpick

The program hsvpick provides an interactive method for selecting colors by experimentation. It has sliders with which you can adjust the RGB or HSV values. As you drag a slider, the color in the square changes accordingly. See Figure 3.

One problem with using window-specific color names in an Icon program is that they aren't portable. For this reason, programs that need to be portable among different window systems should use Icon color names or numerical specifications. The function XColorValue(s) produces the comma-separated decimal form for the color s. Once a numerical specification is determined for a name that is supported on one platform, this numerical specification then can be used in place of the name to make the use of the color more portable.

Next Time

We'll continue with color in the next issue of the Analyst, describing how to use colors and discussing some of the problems involved with color.

References

1. *Colour; Why the World Isn't Grey,* Hazel Rosotti, Princeton University Press, 1983.

2. *Color, Light, Sight, Sense*, Moritz Zwimpfer, Schiffer Publishing Ltd., 1988.

3. *The Color Compendium*, Augustine Hope and Margaret Walch, Van Nostrand Reinhold, 1990.

4. *Color in the 21st Century*, Helene W. Eckstein, Watson-Guptill, 1991.

5. *Principles of Color Design*, Wucius Wong, Van Nostrand Reinhold, 1987.

6. *Theory and Use of Color*, Luigina De Grandis, Abrahams, Inc., 1986.

Programmer-Defined Control Operations

Computational Repertoires

Every programming language has a repertoire of built-in operations that provides the languages's basic computational capabilities. Repertoires vary in their functionality and may be cast in different ways, but the underlying idea always is the same. Almost every programming language has some mechanism for extending its built-in repertoire so that programmer can compose larger computational units. Usually such extensions can be used in the same way as the built-in operations are used. Icon, for example, has a repertoire of built-in functions and a way for declaring procedures. (These are Icon's terms; some other programming languages use different terms.) The use of functions and procedures in Icon is indistinguishable.

There's a tension between the built-in repertoire and the capabilities that are left to programmer-defined extensions. Operations may be included in the built-in repertoire for several reasons. Some operations are so fundamental that they can't be defined but must be built in. Basic input and output are examples of this. Some operations that could be defined in terms of others are built in for efficiency. Other operations are built in for the convenience of programmers. Still other built-in operations contribute to the "character" of the language — what it emphasizes as important.

A large computational repertoire in a programming language provides many capabilities that are readily available and don't have to be defined. But a large computational repertoire imposes a burden on a language: the amount that must be mastered by programmers, the size of the implementation, its maintenance, and its documentation.

Individuals have different opinions about computational repertoires. Different kinds of operations are viewed as important or unimportant, depending on individual needs. Philosophical views differ from "minimalist" to "the more the better". Not long ago, there was a lively discussion in the Icon news group about whether entab() and detab() should have been built in to Icon or left to procedures in the program library.

There also are syntactic issues. Icon uses operator notation for a relatively small set of operations that are used frequently or that correspond to conventional mathematical notation. The rest of the repertoire is cast in a functional syntax, which has the virtue of being open-ended and readily extendable. But again there are tensions. Operators are concise and quickly keyboarded. Operator precedence and associativity rules reduce the need for parentheses. But if there are too many operators, it's easy to get them confused and there is more likelihood that expressions will group in unintended ways because of precedence and associativity.

All of these issues must be dealt with by programming language designers. The result almost always is a compromise, and sometimes an unhappy one. And it's easy enough to make unfortunate decisions. Why, for example, should Icon's rarely used operation for producing a refreshed copy of a co-expression be dignified by an operator instead of being a function?

Extensibility in Programming Languages

The basic issue of extensibility in programming languages has been of varying interest to programming language designers [1]. Extensibility of the computational repertoire generally is taken for granted and work has focused on issues like the extensibility of types and even control structures. Some rather bizarre ideas have been suggested, such as the ability to change the syntax of a programming language dynamically during program execution.

Icon's capabilities for extending its built-in type repertoire are limited to record declarations, which add new type names that can be checked during program execution. Icon does not, however, provide any way for connecting record types to type-specific operations or extending the built-in repertoire to these types.

Control Structures

Although some proposals have been made for providing ways to extend the built-in repertoire of control structures in programming languages, little has been done about it. By control structures we mean constructions such as loops (while-do) and expression selection (if-then-else). In Icon, a control structure is any expression that allows departure from the ordinary flow of control. Most but not all control structures in Icon are distinguished by reserved words. Alternation, repeated alternation, limitation, and string scanning are control structures that are represented with special characters.

Control structures are very fundamental to imperative programming languages. While it's easy enough to add to the computational repertoire of a programming language, either by increasing its vocabulary of built-in functions or by defining new ones, control structures are, almost by definition, idiosyncratic and problematical. Since control structures are so basic, their performance is important and hence they are "hardwired" into the implementation for efficiency, making extensibility much harder.

Control structures differ from the computational repertoire of a programming language in another way. While there are needs for all kinds of computational capabilities, a few control structures usually suffice. Granted, individuals who become used to the control structures in one programming language expect them in another. For example, we occasionally get requests for a do-while control structure to complement Icon's while-do. Nevertheless, the concern about control structures is insignificant compared to the concern about the computational repertoire. This doesn't necessarily mean that Icon could not benefit from new control structures, even novel ones. Instead, the problem is more that it's hard to conceive of novel control structures that would be useful. Increasing vocabulary is far easier than developing new concepts.

The issue of control structures actually is more interesting in Icon than in most imperative programming languages. Because of generators and expressions that can fail, there's a larger range of possible control structures in Icon than in imperative programming languages that have a "flat" from of expression evaluation that always produces one result. Icon's iteration (every-do), alternation, repeated alternation, and limitation control structures would be meaningless in a programming language like Pascal of C. What possible meaning could *expr1* | *expr2* have in such a language? In some sense, control structures are more interesting in Icon than in most other imperative programming languages.

Generators, in the uniform and general way that they are supported in Icon, were novel when Icon was designed. New control structures for dealing with them had to be invented. Iteration and alternation are fairly obvious. It's not clear, however, that Icon has all the useful control structures related to generators that it might, or that the ones it has are cast in the best way. A historical note in this regard may help to put this matter in perspective. Repeated alternation and limitation were added to Icon after it was a mature and established language. They were motivated more by considerations of the abstract properties of sequences of results than by any evident need. Indeed, they originally were viewed as questionable curiosities. Yet once they became familiar in use, they substantially increased the ease of some kinds of programming .

Other control structures related to generators have been suggested for Icon [2]. Perhaps they would be useful, but it's hard to know when they can't be used. And adding control structures to the implementation of Icon is harder than adding them to most other programming languages

Limitations of Goal-Directed Evaluation

There is one problem with expression evaluation in Icon that a new control structure could solve, at least in principle: the inability to get the results from generators in parallel. Because of goal-directed evaluation, suspended generators are resumed in a last-in, first-out fashion. That is, the last generator to suspend always is the first one to be resumed if an alternative value is needed. There is no way to get alternative values for other suspended generators until all the ones for the last suspended generator are produced. Thus, in

```
   every write(!&lcase, "      ", !&ucase)
```

the lines written are

```
      a        A
      a        B
      a        C
           …
```

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

RBBS: (602) 621-2283

FTP: cs.arizona.edu (cd /icon)

```
        a       Z
        b       A
        b       B
           ...
```

not

```
        a       A
        b       B
        c       C
           ...
        z       Z
```

If you want the latter output, you can, of course, obtain it by indexing in a loop, as in:

```
    every i := 1 to *&lcase do
      write(&lcase[i], "      ", &ucase[i])
```

Although it's possible to work around this aspect of expression evaluation, it can't be done with parallel generation. Yet generators are certainly the most powerful feature of Icon, and it is frustrating that they cannot be used in such situations.

Co-Expressions

This problem, and related ones, motivated the introduction of co-expressions in Icon. A co-expression "captures" an expression without evaluating it and produces a data object that can be activated as needed to produce the results of the expression. For the example above, co-expressions can be used as follows:

```
  lower := create !&lcase
  upper := create !&ucase

  while write(@lower, "      ", @upper)
```

which produces

```
        a       A
        b       B
        c       C
           ...
        z       Z
```

The loop terminates when there is not another value for !&lcase and @lower fails.

A procedure can be used to encapsulate this operation:

```
  procedure parawrite(C1, C2)

    while write(@C1, "     ", @c2)

  end
```

For the example above, this procedure would be called as

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

In the general case, the loop terminates when either co-expression runs out of results.

With co-expressions, many things can be done with generators that otherwise are not possible. (But not everything; there's no general way, for example, to produce the results of a generator out of sequence.)

Although co-expressions are very powerful, they are somewhat awkward to use. Procedures such as parawrite() help, but calling them is awk-

The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
Editors

The Icon Analyst is published six times a year. A one-year subscription is $25 in the United States, Canada, and Mexico and $35 elsewhere. To subscribe, contact

Icon Project
Department of Computer Science
Gould-Simpson Building
The University of Arizona
Tucson, Arizona 85721
U.S.A.

voice: (602) 621-8448

fax: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

...uunet!arizona!icon-project

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona

ward because of the need to supply co-expressions as arguments.

Programmer-Defined Control Operations

Programmer-defined control operations are designed to overcome this problem by providing a way to extend the built-in repertoire of control structures in Icon. It should not surprise you that programmer-defined control operations use co-expressions. They don't go as far as allowing a special syntax for a new control structure, but they do make it possible to write procedures that control program flow in many ways that aren't otherwise possible.

Programmer-defined control operations also serve a pedagogical purpose. Just as you can learn a lot about a built-in function by writing it as a procedure [3], you can learn a lot about Icon's built-in control structures by writing them as programmer-defined control operations. For example, if you're still a bit perplexed by the difference between while-do and every-do, writing them as programmer-defined control operations most likely will give you that "aha!" experience of real insight.

Writing programmer-defined control operations requires some knowledge of co-expressions and, like most unfamiliar areas of programming, a little experience coupled with a shift in thinking is necessary to make effective use of them. We'll describe the mechanism and then present some examples on which you can build.

Programmer-defined control operations depend on an alternative form of procedure invocation in which braces rather than parentheses are used, as in

```
parawrite{expr1, expr2}
```

When a procedure is called in this way, the arguments given are not evaluated. Instead a list of co-expressions for the expressions is passed to the procedure. Thus,

```
parawrite{!&lcase, !&ucase}
```

is equivalent to

```
parawrite([create !&lcase, create !&ucase])
```

Consequently, the user of parawrite{} does not have to provide the creates; the use of co-expressions is hidden. Because co-expressions are provided, the arguments are not evaluated before the procedure is called. The use of the list allows an arbitrary number of arguments to be passed. The use of a list is a historical consequence of the fact that the programmer-defined control operation form of procedure call predated the facility for defining procedures with an arbitrary number of arguments.

The procedure for writing the results of generators in parallel now can be cast as:

```
procedure parawrite(args)
  while write(@args[1], "      ", @args[2])
end
```

That's all there is to the mechanism — just a little "syntactic sugar". The rest comes from writing procedures to activate co-expressions in various ways.

Examples

We'll start by modeling some of the existing control structures of Icon. Alternation is a "natural", since it shows how the order in which the results of generators can be controlled:

```
procedure Alt(args)
  local x

  while x := @args[1] do suspend x
  while x := @args[2] do suspend x

end
```

In other words, alternation produces all the results from its first expression followed by all the results from the second. We can write this procedure in a more concise way, but we'll save that for later.

Earlier in this article, we mentioned modeling every-do and while-do. Here they are:

```
procedure Every(args)

  repeat {
    @args[1] | fail
    @^args[2]
    }

  fail

end

procedure While(args)

  repeat {
    @^args[1] | fail
    @^args[2]
    }

  fail

end
```

The control clause in Every{} succeeds as long as @args[1] produces a result. The do clause deserves note: ^args[2] produces a refreshed version of args[2] so that every time it is activated, it produces the first result of the expression. That is, the expression in the do clause is *re-evaluated* each time through the every-do loop and it does not matter whether it succeeds or fails. If we didn't refresh the co-expression, we'd get a form of parallel evaluation in which the next result of args[2] was generated for every iteration of the loop.

The difference between every-do and while-do is shown in the refreshing of the expression in the control clause of while-do — the control expression is re-evaluated each time though the loop instead of being resumed. Note that the bodies of Every{} and While{} differ in only one character.

As an example of adding a control operation for a control structure that Icon does not have, consider the C for statement:

```
for (expr1; expr2; expr3) statement
```

When a for statement is executed, *expr1* is evaluated and its result is ignored (*expr1* typically is used for initialization). Next, *expr2* is evaluated in a loop. If its value is zero, the execution of the for statement is complete. If its value is not zero, *statement* is evaluated (since it is a statement, it has no value in C). Next *expr3* is evaluated and its result is ignored (*expr3* typically is used to increment a counter or pointer). All of the expressions can be omitted; we'll skip that aspect of for in our example.

Here's a programmer-defined control operation that implements C's for statement:

```
procedure For(args)
   @args[1]
   repeat {
      if @^args[2] then @^args[4] else fail
      @^args[3]
      }
   fail
end
```

Note that the co-expressions for the expressions that may be evaluated more than once are refreshed before activating them. Since Icon has no statements, the statement in the C for loop is treated as an expression but its value is ignored. Finally, For{} fails when *expr2* fails. Since it's an expression in Icon, and the built-in looping control structures in Icon fail, this seems like the best choice.

Next Time

Perhaps now you can see some of the possibilities for programmer-defined control operations. We'll continue this subject in the next issue of the Analyst with more examples. We'll also point out some of the limitations of programmer-defined control operations.

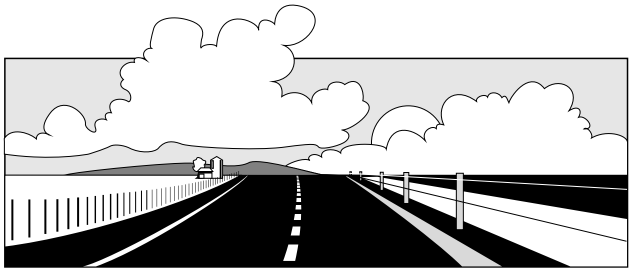
References

1. Proceedings of the International Symposium on Extensible Languages, Grenoble, France, *SIGPLAN Notices*, Vol. 6, No. 12, December, 1971.

2. *Control Mechanisms for Generators in Icon*, Stephen B. Wampler, doctoral dissertation, Department of Computer Science, The University of Arizona, 1981.

3. "Modeling Icon Functions", Icon Analyst 11, pp. 5-7.

4. "Modeling String Scanning", Icon Analyst 6, pp. 1-2.



What's Coming Up

In the next issue of the Analyst, we'll have follow-up articles on color and programmer-defined control operations.

In Issue 7 of the Analyst, we described variant translators, C-based processors that can be used to translate Icon programs into various formats. In the next issue we'll go one step further with meta-variant translators, which allow much of the translation to be specified in Icon instead of C.