The Icon Analyst

In-Depth Coverage of the Icon Programming Language

October 1997
Number 44

In this issue …

New Analyst Features 1
Program Visualization in 3D..............1
Graphics Corner...................................8
From the Wizards 9
Function Tracing................................11
A String-Manipulation Problem.......12
Tricky Business 15
What's Coming Up 16

New Analyst Features

If you've been reading the Analyst for a while, you know it has several features or "columns" that appear irregularly. At present these are:

Programming Tips
From the Library
From the Wizards

Starting with this issue, we're adding two more:

Graphics Corner
Tricky Business

As with the others, we'll let the contents of these speak for themselves.

We're also adding a **Links** section to some articles to list URLs for Web pages that contain additional relevant material. These are indicated by notation of the form {1}, which refers to the first numbered item in the **Links** section at the end of the article.

We'll also put hot links for these on the Analyst Web pages that contain additional material for subscribers.

Program Visualization in 3D

Recently we started to explore the possibilities of visualizing program behavior in three dimensions.

Our goals are modest — static three-dimensional models that depict the history or state of some aspect of program execution. We don't mean an image that portrays a three-dimensional scene but rather an actual three-dimensional model that can be explored.

Although animation, in which the model changes with time, is possible, the resources it requires in terms of hardware, software, experience, and production time are far beyond our means.

Going from two-dimensional animated visualizations to three-dimensional static ones is like entering another world. The concepts, techniques, and results are very different. Little carries over from one to the other. Instead of a rapidly changing "projection" of program activity, three-dimensional visualization allows the user to explore a "scene" composed of shapes corresponding to program events arranged in various ways. Such scenes can be examined from different viewpoints. With a "browser", the user can fly through the world, examine individual objects at close range, back off to get an overall view from a distance, and so on.

Creating 3D Models

There are many programs for producing 3D models. They range from freeware products with modest capabilities to very expensive commercial applications that are in the domain of professionals. Most of these programs are designed to produce images of a scene, not models in which a user can navigate easily.

For program visualization, we need a language into which program activity can be rendered. There are only a few such *scene description languages*. Most are designed for special purposes,

are complex, and are implemented only for a specific platform. For our objectives, we need a scene description language that is easy to program and generally available on commonly used platforms.

The place to look for cross-platform capabilities these days is the Web. For three-dimensional scenes, this leads to VRML, the "Virtual Reality Modeling Language". See the side-bar **About VRML** at the right.

Forget the "virtual reality" part — although VRML is intended ultimately to bring interactive, realistic, and animated "worlds" to the Web, it's a long way from that. Fortunately, that's not our interest in visualization. The important thing is that VRML has all (or almost all) the capabilities we need for an initial exploration of three-dimensional program visualizations that can be viewed on many platforms.

Our goal, then is to produce programs that monitor program behavior and output VRML scenes.

We've explored several possibilities. In this article, we'll describe our first results in three-dimensional visualization of storage allocation.

Visualizing Storage Allocation

Storage allocation is, as before, a good place to start: It is familiar, it is varied enough to be interesting, and it is important. Allocation events are rendered as shapes in the scene, with the types of allocation coded by color (there are few enough types that it is easy to distinguish them in this way) and to translate the amount of allocation to the size of a shape.

Since we're working in three dimensions, objects need to be positioned in meaningful ways that take advantage of the available space. This leads us to the concept of a path in space that in some way corresponds to the passage of time. Time, however, is linear and we need to map it into locations in three-dimensional space. Figures 1 and 2 on the next page, which are from programs in the suite used for dynamic analysis [1], are examples of one approach we've taken.

The pole in the center serves as a "trailhead" that indicates the start of the path. The paths wind outward in a six-sided "spiral". Each allocation is represented by a cylinder whose height is proportional to the amount of allocation (all base radii are the same). There is a fixed distance between each successive allocation. (We don't have a clock of sufficiently high resolution to translate the time between successive allocations into distance.) Paths are drawn on ground plane to make it easier to follow the "time line".

About VRML

VRML was motivated by the desire to bring 3D scenes and "the virtual reality experience" to the Web. VRML stands for Virtual Reality Modeling Language. It was originally dubbed Virtual Reality Markup language, in analogy to HTML, but since VRML is not a markup language, the name was quickly changed. VRML is pronounced *vermel* or *vermul* by those in the know.

VRML is a 3D scene description language. VRML files are called "worlds" and bear the suffix .wrl (sometimes pronounced *dot world* or maybe *dot wurld*). VRML files are text files.

One important aspect of VRML for the purpose of program visualization is that it is a *language*. See the side-bar **VRML as a Language** on page 9. This means that visualizations for different programs can be created by an application.

Another important aspect of VRML is its portability across all popular platforms. See the side-bar **VRML Browsers** on page 6.

Because VRML is designed for use on the Web, it is relatively unsophisticated: Few platforms are fast enough and have connections with sufficiently high bandwidth for downloading and navigating sophisticated 3D scenes in real time. VRML's capabilities nicely match the needs of program visualization: VRML provides various shapes, colors, layout capabilities, and so on.

The success of VRML for its intended purpose is a matter of some debate. Not surprisingly, persons with a vested interest in VRML sing its praises. Others are less sanguine. VRML is, however, in widespread use in the scientific community for publishing 3D information. There are thousands of VRML models of molecules, mathematical shapes, and so on on the Web.

See also the side-bar on **Versions of HTML** on page 5.

Navigation allows inspection from different viewpoints. Figure 3 on the next page shows a view from overhead. Figure 4 shows a view looking across the top of a world in which there are many list allocations of the same size, and Figure 5 shows a ground-level view from among the objects.

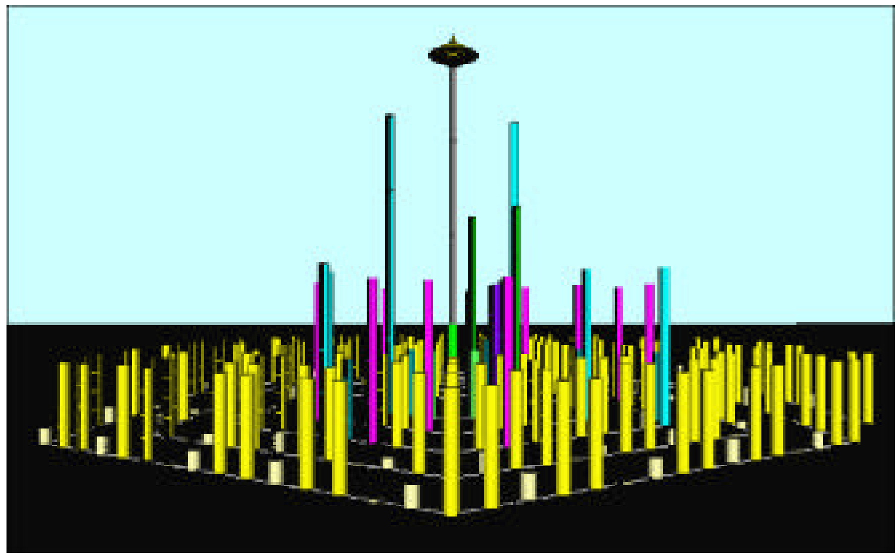


Figure 1. Allocation in rsg

Observations

We were somewhat surprised to discover that static three-dimensional visualizations of storage allocation provided insight that we didn't get from two-dimensional animations.

Part of the reason for this seems to lie in the ability to explore a static world. The three-dimensional representation of allocation also gives a clearer understanding of the sizes of allocations and their relationships to each other.

For example, although we've seen many two-dimensional animations of storage allocation in iiencode, we did not appreciate its true nature until we saw a three-dimensional scene (see Figure 2). All allocations are for small strings and substring trapped variables — allocation "debris" that results from string subscripting [2].

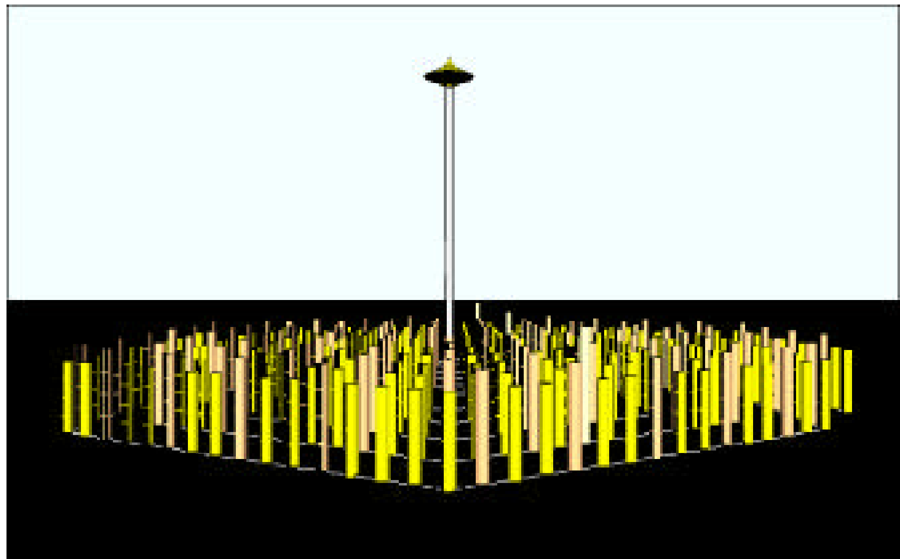


Figure 2. Allocation in iiencode

Of course, looking at grayscale images as shown in this article conveys little of the colored three-dimensional scene. For this, visit the Web page for this issue of the *Analyst* (see the box at the bottom of page 14).

Shapes

Using the same footprint for all objects and scaling the heights to account for sizes gives a correct impression of magnitudes and allows placement of objects along a path without "collisions".

We've generally found that cylinders and boxes work best. See Figure 6 on the next page for an example of a world that uses boxes instead of cylinders.

The other primitive VRML shapes are spheres and cones. Scaling them in only one dimension does not produce visually attractive results. It's possible to create almost any shape in VRML — for example, chess pieces — but the use of primitive shapes greatly reduces the complexity of VRML worlds and the load they place on browsers. It *is* tempting, though … .

Paths

The question of paths is an interesting one. Just a straight line is not particularly useful, since it does not take advantage of the other dimensions

and typically produces a world with a "bounding box" whose aspect ratio is not handled gracefully by VRML browsers.

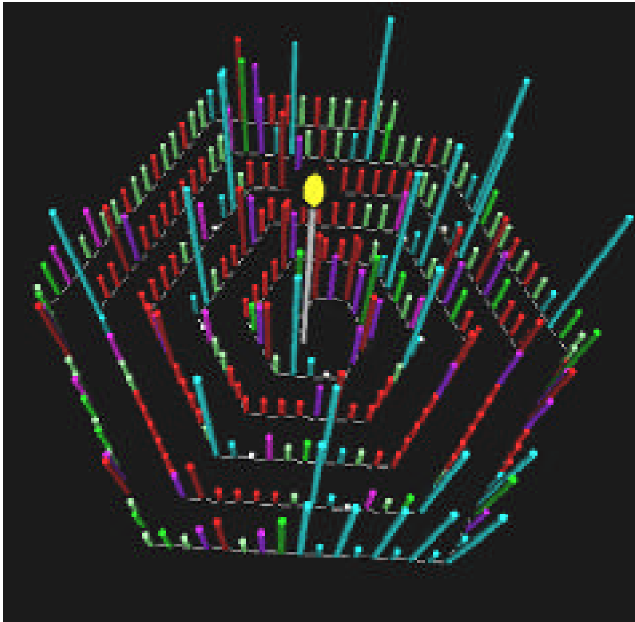


Figure 3. A View from Above

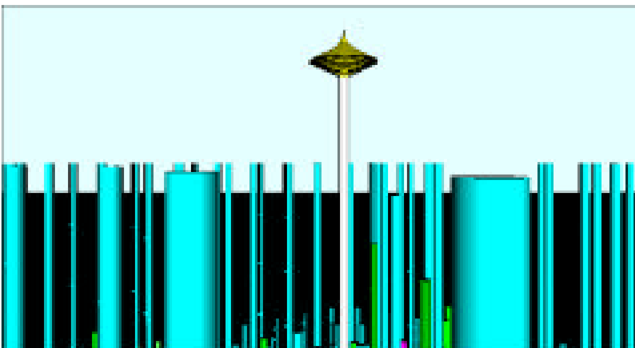


Figure 4. A "Tree-Top" View

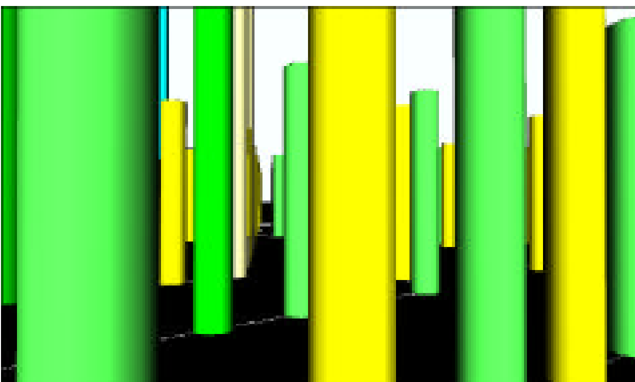


Figure 5. "Among the Stalks"

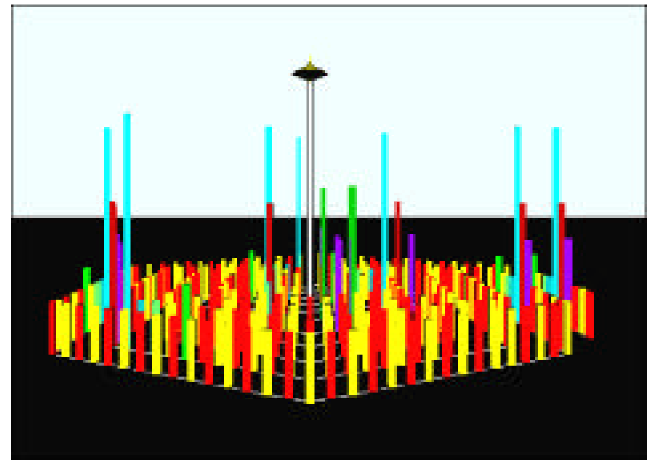


Figure 6. A World with Box Shapes

In theory, a path needs to have an essentially unlimited length, since a point on a path is needed for every allocation and some programs perform many thousands of allocations. In practice, there is a limit to the number of objects in a world before browsers bog down — or crash. We've not had success with more than 500 objects (which is considerably more than the ones shown in this article).

If a path is to be useful as a navigational tool

- It should have hundreds of points.
- It should not cross itself or even closely approach itself.
- It should use at least two dimensions in a space-efficient manner.
- It should be easy to follow visually.
- Successive points should be approximately equidistant.
- It should be visually pleasing.
- If the path is not confined to two dimensions, its projection on the base plane usually should satisfy the preceeding requirements.

We've tried many approaches to constructing paths, including Turtle Graphics [3] and points on mathematical curves and surfaces. This is an interesting subject in and of itself, and we plan to have an article on constructing paths in an upcoming issue of the Analyst.

We've generally found that two-dimensional paths confined to a base plane are most satisfactory. Some three-dimensional paths are, however, quite interesting. See Figure 7, which shows storage allocation as "beads" on a helical path.

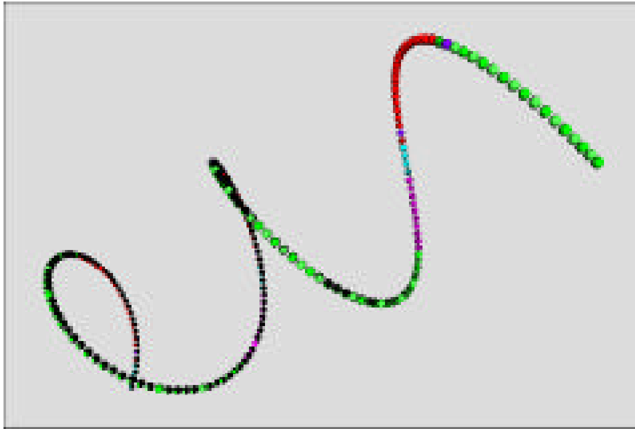


Figure 7. A Strandof Beads

In this world, the sizes of allocations are not represented; all beads are the same size. It's rather complicated to lay out spheres of different sizes in a meaningful and attractive manner on such a path, and we've focused our efforts on other things.

We've also experimented with multiple paths in the same world. Figure 8 shows a world in which different kinds of allocation are in separate "enclaves".

Creating Allocation Worlds

It is conceptually simple to convert allocation events into objects on a path. There are two inputs in parallel: allocation events and points on a path.

Here's a sketch of the process:

```
while get allocation event do {
  get path point
  place shape at point
  }
```

We found it useful for experimentation and program development to put the allocation events

Versions of VRML

There are two official versions of VRML: 1.0 and 2.0. Unfortunately, both VRML 1.0 and 2.0 carry the .wrl suffix. It's easy to tell them apart, however; the first line of a VRML file is a mandatory comment that clearly identifies the version. Although VRML 1.0 is officially designated as outdated, it's in widespread use and is likely to continue to be.

The versions differ considerably in capabilities. Version 1.0 provides only static, silent worlds. Version 2.0 improves some basic aspects of 1.0 and adds sound and movies as well as limited animation and user interaction. At present, the support software for VRML 1.0 is better than for 2.0, although that is changing with time.

Since VRML 1.0 is simpler and the related software is more robust, it is the preferred version for publishing static scientific models.

We've used VRML 1.0 for the worlds shown in this article.

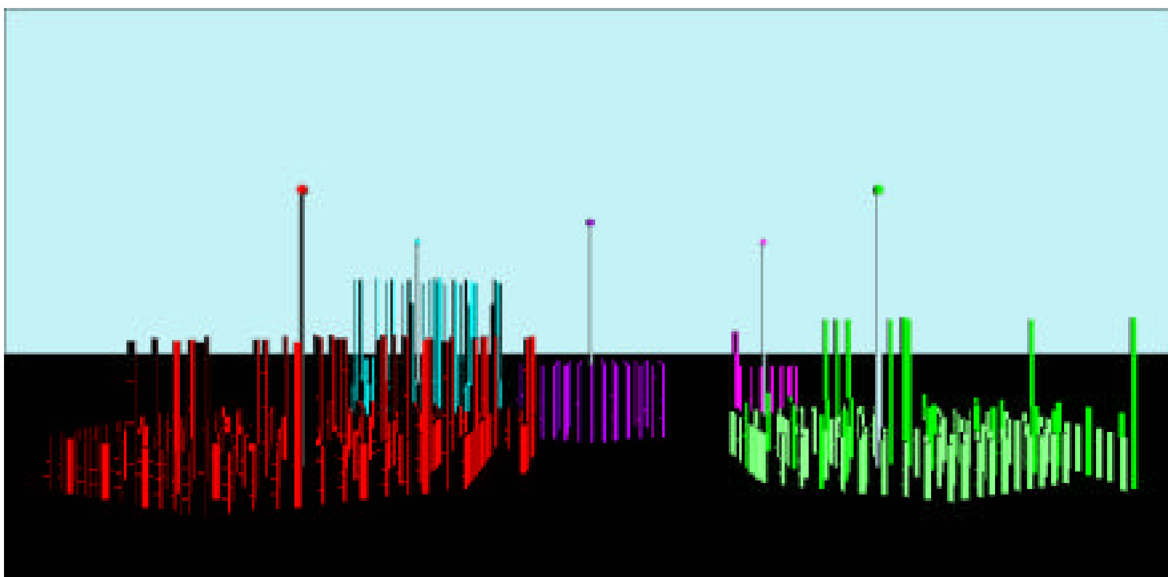


Figure 8. Multiple Paths

and path points in lists prior to constructing a VRML file. This also allows reuse of previously constructed lists.

We represented events and points by records:

```
record Event(size, type)
record Point(x, y, z)
```

and put the events and points in lists events and path. Then the translation of events into objects on the path is

```
while event := get(events) do {
  point := get(path)
  put(world, object(event, point))
  }
```

where object() is a procedure that constructs an appropriate object and world is a list of the objects from which a VRML file eventually is produced.

Here's a sketch of object():

```
procedure object(event, point)
  return Separator([
    Material(color(event.type)),
    Translate(location(point)),
    Transform(scale(event.size),
    Cylinder()
    ])
```

Separator() is a record that represents a VRML separator node that isolates its subnodes so that they do not affect the rest of the world. Material() establishes the color for subsequent nodes. Translate moves the origin to the location specified by point and Transform() scales the following cylinder, which is what actually appears in the world.

We won't get further into the details of VRML here. What needs to be done to construct the worlds we've shown is relatively straightforward, but there are a lot of details and producing optimal VRML code adds complexity.

You'll find more information about writing programs to construct VRML worlds in the **From the Wizards** article that starts on page 9.

Exploring VRML

If you're interested in learning more about VRML, there are two principal resources: books and the Web.

At present there are more than 30 books on VRML and more are in the works. (To put this in perspective, there are more that 240 books on HTML.) The VRML books range from surveys of worlds on the Web for novices to books on how to program in VRML. There is even a "For Dummies" book [4], which is better than most books on VRML we've seen and probably a good place to start. The book situation is complicated by the two versions of VRML. See the side-bar on page 5. Some books cover one version, some cover both, and some don't make the distinction clear. The quality of

VRML Browsers

Programs for viewing and navigating VRML worlds are called browsers. They come in two forms: stand-alone applications and plug-ins for Web Browsers.

Plug-ins, which allow loading and navigating in VRML worlds directly from a Web browser, are better developed and more readily available than stand-alone browsers — at least on most platforms. Both Netscape and Internet Explorer bundle VRML plug-ins with their latest releases for the most popular platforms.

Silicon Graphics was instrumental in the design and development of VRML (their Open Inventor format was the basis for VRML 1.0), so it's not surprising that the most impressive VRML software exists for SGI platforms. VRML software for Windows, again for obvious reasons, is the most extensive and widely available.

Except on powerful platforms, navigating a static VRML 1.0 world of even modest complexity may be slow and tedious. It also takes some experience to become comfortable with navigation in three dimensions.

It's easy to get lost and disoriented when navigating. Most browsers provide a way to reset the original view, which is determined by a "camera" in the world, or in some cases, the whim of the browser.

Regrettably but predictably, different VRML browsers use different models of navigation. To complicate matters, few if any of the browsers currently available implement all the features of VRML completely and correctly, and many browsers are seriously buggy. In additional, the same world may look very different when viewed with different browsers.

Unless you're in the trade, it's probably best to pick one browser and stick with it.

some books is adversely affected by the rush to get into print in a highly hyped area.

While we haven't found a book on VRML that we can recommend without reservations, we found the books listed in References 5 - 7 to be useful.

Book superstores typically have only a few of the available books on VRML on their shelves where you can browse before buying. On the Web, the best sources for technical books in our opinion are Amazon {1} and CLBooks {2}, which recently acquired Computer Literacy Bookstores.

The amount of VRML material on the Web is overwhelming. If you use one of the popular search engines and just ask for "VRML", you're likely to get anywhere from hundreds of thousands to millions of hits, depending on the search engine. The place to start is the VRML Repository {3}, from which you can get to almost any VRML resource if you try hard enough. If you have a special interest, a refined search may get you there faster.

If you want to explore VRML worlds, there are many sites on the Web. We particularly like the worlds at Links 4 through 8.

Conclusions

We've barely scratched the surface of what's possible in three-dimensional program visualization using only VRML. We have a few experiments underway, including detailed views of the contents of structures and three-dimensional visualization of control flow.

On Our Web Site

We've put a lot of material related to this article on the Web page for this issue of the *Analyst*. In addition to color images for the figures, we've included VRML worlds that you can download or browse on-line. We've also included some images and worlds for which there wasn't space in this article, as well as some of our more interesting mistakes. Have fun!

Icon on the Web

Information about Icon is available on the World Wide Web at

http://www.cs.arizona.edu/icon/

References

1. "Dynamic Analysis of Icon Programs", *Icon Analyst* 29, pp. 10-12.

2. *The Implementation of the Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Princeton University Press, Princeton, New Jersey, 1986, pp. 71-73.

3. "Turtle Graphics", *The Icon Analyst* 24, pp. 6-10.

4. *VRML and 3D on the Web for Dummies*, David Kay and Douglas Muder, IDG books Worldwide, 1996.

5. *Special Edition; Using VRML*, Stephen Matusuba and Bernie Roehl, Que, 1996.

6. *The VRML Sourcebook*, Andea L. Ames, David R. Nadeau, and John H. Moreland, John Wiley & Sons, Inc., 1996.

7. *VRML 2.0 Sourcebook*, Andea L. Ames, David R. Nadeau, and John H. Moreland, John Wiley & Sons, Inc., 1997.

Links

Note: These links and others are on the Web page for this issue of the *Analyst*. Please realize that Web sites come and go and also move. We believe, however, that the URLs listed below are reasonably stable.

1. Amazon Books: http://www.amazon.com/

2. Computer Literacy: http://www.clbooks.com/

3. The VRML Repository: http://www.sdsc.edu/vrml/repository.html

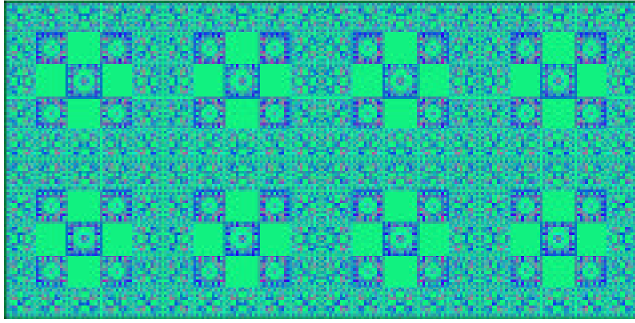
4. Molecular Models: http://www.nyu.edu/pages/mathmol/library/

5. Mathematical Knots: http://www.cs.ubc.ca/nest/imager/contributions/scharein/KnotPlot.html

6. Mathematical Objects: http://www.geom.umn.edu/software/weboogl/zoo/

7. Crystal Structures: http://193.49.43.3/dif/3D_gallery.html

8. Polyhedra: http://www.li.net/~george/virtual-polyhedra/vp.html



Graphics Corner

Repeat patterns, also called allover patterns, result from replicating a motif to cover the plane (or at least part of it).

Repeat patterns date from antiquity. They add visual interest to otherwise monotonous surfaces and generally are less expensive to produce than other types of decoration. Repeat patterns are prevalent in manufactured goods, especially fabrics and wallpaper. Repeat patterns now are popular as backgrounds for Web pages. They also are used as "textures" that are wrapped around shapes in three-dimensional modeling.

Producing attractive motifs for repeat patterns is an art. Here we're just concerned with replicating a motif (image) to create a repeat pattern.

If replication is done so that there are no gaps or overlaps, the result is referred to as *tiling the plane*. The image used for tiling is called the *generating tile*, or GT.

There are only three regular convex polygons that can be arranged to tile the plane: the equilateral triangle, the square, and the regular hexagon, which is shown in Figure 1.

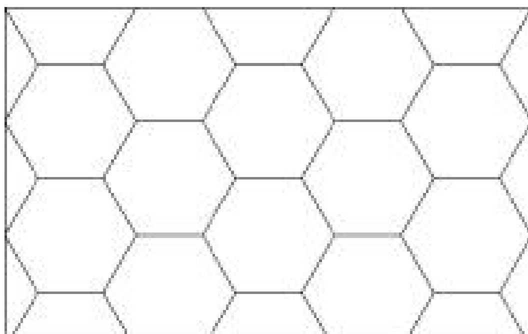


Figure 1. Hexagonal Tiling

If the regularity constraint is removed, any triangle or quadrilateral can be arranged to tile the plane. Any convex pentagon having a pair of parallel sides can tile the plane. There are three distinct kinds of irregular hexagons that can tile the plan. Finally, no convex polygon with more than six sides can tile the plane. See References 1 and 2 for more information about tiling.

From a programming standpoint, rectangular tilings are by far the easiest and fastest to produce. Furthermore, many other tilings can be converted to rectangular tilings. For example, a rectangular GT (RGT) for a hexagonal tiling is shown by the marquee in Figure 2.

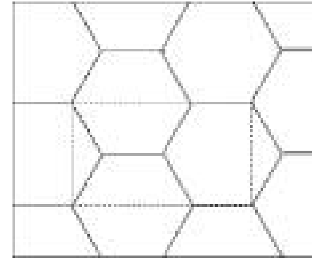


Figure 2. A Rectangular Tile for Hexagons

Although the aspect ratio for the RGT for regular hexagons is irrational (3:1), images are composed of discrete pixels and it's easy to drag a selection rectangle (called a marquee) in an image manipulation application such as PhotoShop to select an appropriate rectangular region that produces an acceptable tiling.

In Icon, the function CopyArea() provides an easy way to replicate an RGT over a larger area. CopyArea() also has the virtue of being fast because it is a memory-to-memory operation.

Here's a procedure that tiles a window with a rectangular area from another window:

```
procedure tile(win1, win2, x1, y1, w1, h1)
   local w, h, wmax, hmax

   /win1 := &window
   /win2 := &window
   /x1 := 0
   /y1 := 0
   /w1 := WAttrib(win1, "width")
   /h1 := WAttrib(win1, "height")
   wmax := WAttrib(win2, "width")
   hmax := WAttrib(win2, "height")

   if (w1 | h1) = 0 then fail

   CopyArea(win1, win2, x1, y1, w1, h1)

   w := abs(w1)
   h := abs(h1)
```

```
    while w < wmax do {            # copy and double
      CopyArea(win2, win2, 0, 0, w, h, w, 0)
      w  := 2
      }

    while h < hmax do {            # copy and double
      CopyArea(win2, win2, 0, 0, w, h, 0, h)
      h  := 2
      }

    return

  end
```

This procedure first copies the specified rectangle to the target window and then copies the already tiled portion of the target window onto itself, doubling the dimensions each time.

This may seem like overkill since CopyArea() is fast, but consider tiling with a very small rectangle, say 2×2 pixels, onto a large target window, say 1000×1000 pixels. Without the optimization, 500^2 = 250,000 copies are required. With the optimization, only 19 copies are required: a reduction factor of more than 13,100. Even if CopyArea() took no time at all, the loop to compute its parameters and call it does take time. Granted, the example above is extreme, but even for more typical situations, the difference in speed is quite noticeable.

Library Resources

In addition to the procedure tile(), the Icon program library includes two programs to assist in tiling: tiler, which tiles image strings [3], and imgpaper, which displays tiled versions of a sequence of images.

More to Come

We'll have more to say about repeat patterns in future **Graphics Corners**, including symmetries for tiling and methods for creating repeat patterns.

References

1. *Tilings and Patterns*, Branko Grünbaum and G. C. Shephard, W. H. Freeman and Company, 1987.

2. *Mathematics; The Science of Patterns*, Keith Devlin, Scientific American Library, 1997.

3. *Graphics Facilities for the Icon Programming Language; Version 9.3*, Gregg M. Townsend, Ralph E. Griswold, and Clinton L. Jeffery, IPD281, 1996.

From the Wizards

Record Fields

The article **Program Visualization in 3D** that starts on page 1 describes the use of VRML for program visualization. To do this, it is necessary to write programs that produce VRML files. See the side-bar **VRML as a Language** on the next page.

We wanted to be able to cast programs to construct VRML files in a way that captured the concepts of VRML and corresponded as closely as possible to VRML syntax. VRML worlds consists of objects called nodes. The natural program representation is to use record declarations for each node type. The fields follow naturally. For example,

```
record Cylinder(height, radius)
```

is a declaration for cylinders, so that

```
pillar := Cylinder(20.5, 2)
```

creates a cylinder 20.5 units in height and 2 units in radius.

Since some VRML field names are long (an example is textureCoordIndex), it was tempting to abbreviate the field names in the record declarations. Programs, however, might contain explicit field name references, as in

```
pillar.height := 30
```

so it seemed wise to use the same names VRML does; at least one can go to the VRML documentation to recall a field name.

When it comes time to output the VRML code for a cylinder, the code might be:

```
write("Cylinder {
write("   height ", Cylinder.height)
write("   radius", Cylinder.radius)
write("   }")
```

VRML as a Language

VRML is a language, but it is not a programming language in the ordinary sense. VRML does not have any computational component or any control structures. As a language, it consists only of a repertoire of statements.

A VRML file is composed of *nodes* that represent shapes, geometric transformations, colors, textures, lights, cameras, and so forth. Nodes have fields that describe their properties. The names of nodes always begin with an uppercase letter and the names of fields always begin with a lowercase letter.

Here, for example, is a node for a cylindrical shape:

```
Cylinder {
  height 20.5
  radius 2.0
  }
```

VRML worlds are dimensionless. Sizes are simply "units" — a unit could represent a micron or a light-year. By convention, however, a VRML unit usually is taken to be a meter. This facilitates the combination of worlds. (A VRML file can contain references (URLs) to other VRML files.) The cylinder above therefore would be interpreted as being 20.5 meters high and 2.0 meters in radius.

Some kinds of nodes have fields that contain (point to) other nodes. There is a single root node that contains all subsequent nodes; a VRML world is a tree structure.

Since there are many node types, the code to write out the VRML file might look something like this:

```
case type(node) of {
      …
  "Cylinder": {
    write("Cylinder {
    write("   height ", Cylinder.height)
    write("   radius", Cylinder.radius)
    write("   }")
    }
      …
"Sphere": {
    write("Sphere {
    write("   radius", Spherer.radius)
    write("   }")
    }
      …
```

and so on for each node type.

Writing this code quickly becomes tiresome — there are 36 node types in VRML 1.0 and many more in VRML 2.0.

Fortunately, there's a much shorter way to cast the code to write a VRML file. Recall from the article on records in a recent *Analyst* [1] that it is possible to get the names of the fields of a record from the record itself. A somewhat different version of the procedure given there is what's needed to handle the many VRML node types in a generic way:

```
procedure field(R, i)

    name(R[i]) ? {
      tab(upto('.') + 1)
      return tab(0)
      }

end
```

Thus, the code for writing VRML files can be written this simply:

```
write(type(node), " {")
every i := 1 to  node do
  write("   ", field(x, i), " ", x[i])
write("   }")
```

Since the record names are chosen to coincide with the VRML node names, type(node) takes care of all of them.

Of course, it's not really this simple. Since a node may contain pointers to other nodes, recursion is needed. To get a readable file, care needs to be given to layout and the indentation of levels. There also are a few node types that require special handling. But all-in-all, the method above saves an enormous amount of coding.

Reference

1. "Records", *Icon Analyst* 41, pp. 7-10.

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

ftp.cs.arizona.edu (cd /icon)

Function Tracing

Icon itself does not provide a mechanism for tracing built-in functions, although ProIcon for the Macintosh does [1].

It is possible, however, to get the effect of function tracing by overloading functions with procedures of the same name and then tracing procedures.

At first glance, it may seem as if overloading a function with a procedure of the same name would make the function inaccessible. As described in an earlier Analyst article [2], however, there is a way: proc(s, 0) returns the built-in function named s, even if that function has been overloaded.

Suppose, for example, that the function pop() is overloaded by a procedure to perform the same computation as the function. The procedure might look like this:

```
procedure pop(L)

   f := proc("pop", 0)

   return f(L)

end
```

The procedure fails if f() fails, since return *expr* fails if *expr* fails.

It's not necessary to get the function for every call of such a procedure; that can be done once with a static variable and an initial clause:

```
procedure pop(L)
   static f

   initial f := proc("pop", 0)

   return f(L)

end
```

Two things are needed for a procedure structure that is applicable to all functions: handling multiple arguments and generators.

Multiple arguments can be handled by list invocation and a declaration for a variable number of arguments. An example is this procedure to overload write():

```
procedure write(args[])
   static f

   initial f := proc("write", 0)

   return f ! args

end
```

Suspension is easy. Here's a procedure to overload seq():

```
procedure seq(args[])
   static f

   initial f := proc("seq", 0)

   suspend f ! args

end
```

Furthermore, it's not necessary to make special cases for functions that aren't generators; suspend works just as well as return for functions that are not generators. There is a subtlety, however:

```
suspend expr
```

the suspend expression itself fails if *expr* fails, but this does not cause the procedure call to fail. It's flowing off the end of the procedure following the failure of suspend that causes failure in this case.

So a general model for overloading a function fnc() with a procedure is

```
procedure fnc(args[])
   static f

   initial f := proc("fnc", 0)

   suspend f ! args

end
```

Providing overloading procedures for all functions is a daunting task — Version 9.3 of Icon has 140 functions. There is help, however, in the Icon program library.

The program gftrace produces a file of procedure declarations of the form shown above for all functions. Run it and save the output in a file named, say, ftrace.icn. You then can create a library module for it with

```
icont –c ftrace
```

and link it in a program for which you want procedure tracing using

```
link ftrace
```

You'll need to enable tracing, of course.

Since an overloading procedure has a single list argument, this is reflected in the format of trace messages. An example of function tracing is shown in Figure 1 on the next page.

The file gftrace produces does a few extra things, including providing a way to trace proc() itself.

```
lst.icn     :  7   | list(list_1 = [])
ftrace.icn  :125   | list suspended list_3 = []
lst.icn     :  9   | read(list_4 = [])
ftrace.icn  :142   | read suspended "This file illust..."
lst.icn     :  9   | put(list_5 = [list_3 = [],"This file illust..."])
ftrace.icn  :141   | put suspended list_3 = ["This file illust..."]
lst.icn     :  9   | read(list_6 = [])
ftrace.icn  :142   | read suspended "better"
lst.icn     :  9   | put(list_7 = [list_3(1),"better"])
ftrace.icn  :141   | put suspended list_3 = ["This file illust...","better"]
lst.icn     :  9   | read(list_8 = [])
ftrace.icn  :142   | read suspended "possible."
lst.icn     :  9   | put(list_9 = [list_3(2),"possible."])
ftrace.icn  :141   | put suspended list_3 = ["This file illust...","better","possible."]
lst.icn     :  9   | read(list_10 = [])
ftrace.icn  :142   | read failed
lst.icn     : 11   | pull(list_11 = [list_3(3)])
ftrace.icn  :139   | pull suspended "possible."
lst.icn     : 11   | write(list_12 = ["possible."])
ftrace.icn  :171   | write suspended "possible."
lst.icn     : 11   | pull(list_13 = [list_3(2)])
ftrace.icn  :139   | pull suspended "better"
lst.icn     : 11   | write(list_14 = ["to give a better..."])
ftrace.icn  :171   | write suspended "better"
lst.icn     : 11   | pull(list_15 = [list_3(1)])
ftrace.icn  :139   | pull suspended "This file illust..."
lst.icn     : 11   | write(list_16 = ["This file illust..."])
ftrace.icn  :171   | write suspended "This file illust..."
lst.icn     : 11   | pull(list_17 = [list_3 = []])
ftrace.icn  :139   | pull failed
lst.icn     : 13   main failed
```

Figure 1. Function Tracing Output

If you don't want to trace all functions, the Icon program library provides an alternative method: the procedure module iftrace allows tracing of specified functions. iftrace in turn depends on a file similar to the one produced by gftrace.

This sounds confusing, but if you look at the files involved, it should be easy to figure out how to use them. The programming techniques used in iftrace are worth studying in their own right.

References

1. *The ProIcon Programming Language for the Apple Macintosh Computers; Version 2.0*, The Bright Forest Company, Tucson, Arizona, 1990.

2. "Records", Icon Analyst 41, pp. 7-10.

A String Manipulation Problem

About the time we were completing the last issue of the Analyst, Steve Wampler posed a problem to the Icon users group: Write a procedure that returns the longest common initial substring of two strings. The idea, of course, was to produce the fastest solution.

By coincidence, we'd just formulated a solution for a special case of this problem as part of a procedure for testing numbers for versumness [1]. If we'd had more time, we would have waited for other solutions and adapted the best for our needs. But we didn't, and we went with the one we'd already done.

Steve received several solutions to the problem and crafted some of his own. There is considerable diversity among the solutions, but they split into two categories according to method: those that use string scanning and those that don't. Some solutions are "optimistic" in the sense they start looking

for the longest possible match, while others take the opposite approach.

We converted the procedures to expressions to eliminate the overhead of procedure calls and used empg from the Icon program library[2-3] for timings.

The first 11 solutions that follow come from Steve's collection. We added the last four, which we'll explain later.

Solution 1:

```
every i := seq() do
  if not (s1[i] == s2[i]) then break s1[1:i]}
```

Solution 2:

```
every i := seq() do
  if not match(s1[i],s2,i) then break s1[1:i]
```

Solution 3:

```
s1[1+:(p :=   s1 to 0 by –1)] == s2[1+:p]
```

Solution 4:

```
s1[1+:(p := (many(s1      s2, s1)|0) to 0 by –1)] ==
  s2[1+:p]
```

Solution 5:

```
i := 1

while s1[i] == s2[i] do
  i +:= 1

s1[1+:i–1]
```

Solution 6:

```
i := 1
j := 0

while match(s1[j+1 +:i], s2, j+1) do {
  j +:= i
  i   := 2
  }

while j +:= 1 & s1[j] == s2[j]

s1[1+:j–1]
```

Solution 7:

```
s1[1+:lcph(s1, s2, 0)]
```

where lcph() is

```
procedure lcph(s1, s2 ,k)
  local i, j

  i := 1
  j := 0

  while match(s1[(k + j + 1)+:i], s2, k + j + 1) do {
    j +:= i
    i   := 2
    }

   if j = 0 then return 0
     else return j + lcph(s1, s2, k + j)

end
```

Solution 8:

```
s1 ? {
  every ch := !s2 do {
    =ch | break
    tab(1)
    }
  }
```

Solution 9:

```
s1 ? {
  tab(match(s2[1:  s2+1 to 1 by –1]))
  }
```

Solution 10:

```
s2 ? {
  =s1[1+:  s1 to 0 by –1]
  }
```

Solution 11:

```
s2 ? {
  =s1[1+:  (if   s1 >   s2 then s2 else s1) to 0 by –1]
  }
```

Solution 12:

```
s1 ? {
  =s2[1:(  s2 + 1) to 1 by –1]
  }
```

Solution 13:

```
if   s2 >   s1 then s1 :=: s2
s1 ? =s2[1:(  s2 + 1) to 1 by –1]
```

Solution 14:

```
s1 ? {
  =s2[1:(((  s1 >   s2) |   s1) + 1) to 1 by –1]
  }
```

Solution 15:

```
i := ((  s1 >   s2) |   s1

s1 ? =s2[1:(i + 1) to 1 by –1]
```

Not surprisingly, the timings varied considerably and depended, of course, on the two strings

used. There is no “typical” data for this problem. The strings might be relatively short and used for “command completion” or very long as in the case of versum numbers. Here are the strings we used for testing:

	s1	s2
1	""	""
2	"aaaa"	"aaaa"
3	"aaaaaaaaaaaaaaaaaaaaa"	"aaaaaaaaaaaaaaaaaaaa"
4	"aaaaaaaaaaaaaaaaaaaaa"	"aaaaaaaaaaaaaaaaaaab"
5	"aaaaaaaaaabaaaaaaaaaa"	"baaaaaaaaaaaaaaaaaaa"
6	"aaaaaaaaaaaaaaaaaaaaa"	"baaaaaaaaaaaaaaaaaaa"
7	"a"	"aaaaaaaaaaaaaaaaaaaa"
8	"aaaaaaaaaaaaaaaaaaaa"	"a"

The results are shown in Figure 1, ranked from best to worst total time for all eight tests. The best times for each test are marked with asterisks, but note that in some cases the times are very close together. For what it’s worth, the figures are in milliseconds on a 233 Mhz DEC Alpha.

When we started this article, we wondered how the method used in the last issue of the Analyst would stack up (and noted that one of the solutions submitted to Steve, *Solution 11*, is very similar to it). Our method was designed for the case where the two strings are the same length, and it behaves poorly when s2 is significantly longer than s1. We crafted another solution to swap s1 and s2 when this is the case, and we then added two more solutions, hoping to improve performance slightly.

We haven’t identified the authors of the solutions, but the one from the last Analyst is *Solution 12*. *Solutions 13*, *14*, and *15* resulted from changes to it.

Several things about the timings deserve note:

• All the solutions that use string scanning are faster than the solutions that do not.

• *Solution 7,* which uses recursion to effect doubling of the substrings examined, suffers from the procedure overhead and initialization involved.

• None of the four solutions with the best total timings is the fastest in any single test.

If you come up with a method that is significantly different from the ones listed, send it to us and we’ll include it in a future issue of the Analyst.

References

1. “Factors of Versum Numbers”, Icon Analyst 41, pp. 9-14.

2. “The Anatomy of a Program — Timing Icon Expressions”, Icon Analyst 18, pp. 8-11.

3. “The Anatomy of a Program — Timing Icon Expressions”, Icon Analyst 19, pp. 6-9.

solution	*test1*	*test2*	*test3*	*test4*	*test5*	*test6*	*test7*	*test8*	*total*
14	0.0334	0.0351	0.0351	0.0425	0.1201	0.1951	0.0334	0.0334	0.4947
13	0.0351	0.0325	0.0351	0.0449	0.1234	0.1967	0.0341	0.0351	0.5018
15	0.0367	0.0341	0.0358	0.0467	0.1234	0.1967	0.0367	0.0366	0.5101
11	0.0334	0.0325	0.0351	0.0458	0.1551	0.2617	0.0334	0.0334	0.5970
12	0.0284*	0.0284	0.0301*	0.0375*	0.1166	0.1875	0.1749	0.0283*	0.6034
10	0.0284*	0.0283*	0.0401	0.0525	0.1601	0.2684	0.0284*	0.2517	0.6062
8	0.0417	0.0625	0.2434	0.2417	0.1408	0.0291*	0.0401	0.0308	0.7993
9	0.0383	0.0375	0.0401	0.0534	0.1608	0.2701	0.2517	0.0383	0.8519
4	0.0601	0.0584	0.0834	0.1041	0.0784*	0.5317	0.0575	0.3883	0.9736
3	0.0308	0.0317	0.0501	0.0751	0.2817	0.4958	0.0317	0.3549	0.9969
6	0.0917	0.1251	0.2684	0.2551	0.2151	0.0617	0.0717	0.0884	1.0888
2	0.0601	0.0967	0.3817	0.3634	0.2101	0.0325	0.0417	0.0551	1.1862
1	0.0601	0.0951	0.3801	0.3675	0.2101	0.0325	0.0417	0.0467	1.1871
5	0.0601	0.0975	0.4017	0.3901	0.2184	0.0301	0.0399	0.0451	1.2378
7	0.1901	0.2301	0.5284	0.4525	0.3534	0.0533	0.1151	0.1417	1.9229

Figure 1. Timing Results

Tricky Business

Preprocessor Definitions

Icon's preprocessor allows defined names to be associated with strings so that when the name appears subsequently, the corresponding string is substituted in its place.

A typical use of defined names is to provide symbolic constants as in

```
$define Height 500
```

The use of defined names is not limited to constants. Any string can be given a name with the exception that quoted literals must be complete. In addition, whitespace before and after the string, including trailing comments, is discarded.

A potentially valuable use of preprocessor definitions is to associate names with frequently used expressions. Furthermore, well-chosen names can convey meaning in a more direct way than the expressions themselves.

We frequently are annoyed when writing scanning expressions by the number of characters it takes to express a simple operation (and are reminded that SNOBOL4 [1] handles at least this aspect of string analysis in a more parsimonious way).

An example is the following scanning expression, which breaks out comma-terminated fields of a record:

```
item ? {
   name := tab(upto(','))
   move(1)
   ssn := tab(upto(','))
   move(1)
   position := tab(upto(','))
   move(1)
   salary := tab(upto(','))
   move(1)
   hire_date := tab(upto(','))
   }
```

Two expressions, representing simple concepts, are used repeatedly. The number of repetitions might be more or less, depending on the data.

Although it's possible to write this as a loop without repeating the expressions, assigning the values to the variables then is a problem.

The expressions above contain two scanning concepts:

tab(upto(','))	get the string up to the separator
move(1)	skip the separator

We can associate names with these expressions as follows:

```
$define Field         tab(upto(','))
$define Separator     move(1)
```

Here we've used names that describe the structure of the string to be matched, not how the match is accomplished.

Now we can rewrite the scanning expression as

```
item ? {
   name := Field
   Separator
   ssn := Field
   Separator
   position := Field
   Separator
   salary := Field
   Separator
   hire_date := Field
   }
```

This is not only requires less writing than the previous formulation, but in our opinion, it also is easier to understand.

Additional Material

Additional material related to this issue of the Analyst, including color images and Web links, is available on the Web. The URL is

http://www.cs.arizona.edu/icon/www/analyst/iasub/ia44/ia44sub.html

A similar programming technique can be used in many other situations. As with most other programming techniques, there's a certain "art" in doing it well.

We used **Tricky Business** for this programming technique because it is all too easy to abuse, leading to error-prone and incomprehensible programs. As a truly awful example, consider these definitions:

```
$define i  if
$define t  then
$define e  else
$define w  while
$define d  do
$define r  return
```

and a program containing something like this:

```
i j > 0 t r e w read() d j+:= 1
```

Using Icon's preprocessor intelligently and with restraint can, however, make programming easier and produce more elegant results. We'll have another article on uses of the preprocessor in a future issue of the Analyst.

Reference

1. *The SNOBOL4 Programming Language*, second edition, Ralph E. Griswold, James F. Poage, and Ivan P. Polonsky, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1971.

What's Coming Up

We have several articles in the works. For the next issue of the Analyst, we expect to have a program anatomy entitled "Numerical Carpets". We'll also have more on tiling and possibly another article on using Icon's preprocessor.

The last article in the series on debugging will describe itweak, an Icon debugger. We also expect to have the second article on versum factors.

As always, what actually appears in an Analyst depends on how things go and what fits together to make the pages come out right.

The I con Analyst

Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend
Editors

The I con 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
The University of Arizona
P.O. Box 210077
Tucson, Arizona 85721-0077
U.S.A.

voice: (520) 621-6613

fax: (520) 621-4246

Electronic mail may be sent to:

icon–project@cs.arizona.edu

THE UNIVERSITY OF
ARIZONA ®
TUCSON ARIZONA

and

Bright Forest Publishers
Tucson Arizona

© 1997 by Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend

All rights reserved.