# Graphics Programming in Icon Version 9

Clinton L. Jeffery
*Division of MCSS, The University of Texas, San Antonio, TX 78249, U.S.A.*
*E-mail: jeffery@ringer.cs.utsa.edu*

Ralph E. Griswold and Gregg M. Townsend
*Department of Computer Science, The University of Arizona, Tucson, AZ 85721, U.S.A.*
*E-mail: {ralph|gmt}@cs.arizona.edu*

**Abstract**

Version 9 of the Icon programming language introduces support for graphics and user interface programming as an extension of the existing file-based input/output model, rather than introducing graphics as a disjoint facility. Simplicity, generality, and portability are the primary emphases. The result is a language in which common graphic effects are easy to write using ordinary procedural code. Complex techniques such as event-driven programming are optional and not forced on the programmer when they are not needed.

## Background and Objectives

Support for graphics and interactive programming usually consists of complex libraries that are difficult to learn by virtue of their girth. Libraries and toolkits found in many languages often make supported behavior trivial at the cost of making unsupported behavior difficult or impossible; many toolkits also emphasize the "user interface" aspect of graphical user interface programming while neglecting the goal of providing flexible graphics output capabilities. Interface construction tools help, but they too have limitations.

In the Icon programming language, we have added support for graphics and interface programming directly to the language. For the sake of those maintaining existing programs, as well as clarity and ease of learning, Icon's graphics facilities focus on simplicity and integration of older file-based and newer interactive input/output models.

Icon's graphics facilities consist of 47 functions and an extension of an existing type. Experience has shown that new users can write windowing applications immediately, yet the facilities are complete enough for development of sophisticated interfaces. Resulting programs run unmodified on X Window, and OS/2 Presentation Manager; MS Windows, Windows NT, and Macintosh support are pending. Ease of programming may be exemplified (if only simply) by Brad Myers' challenge issued prior to the CHI '92 conference: how hard is it to write a program in which the user uses a mouse to move a rectangle around on-screen?

Another important consideration in the language design was the integration of the graphics subsystem into the existing language. This issue addresses the difficulty with which the new input/output facilities are adopted in existing programs and by existing programmers. A question we might use to evaluate a design is: how hard is it to add mouse support to existing text-oriented applications?

An objective underlying these issues was to leave the programmer in control of the program. Most graphics facilities mandate *event-driven programming*. The basic tenet of event-driven programming is that at every instant the user, instead of the program, should be in charge of the interaction with the computer. When taken too far in language design, like most dogmas, the benefits of event-driven programming are achieved at the cost of an enforced program complexity that often is not necessary. In particular, it makes the job of adding graphics or the use of a mouse in text-oriented applications more difficult than is necessary. In Icon the programmer is free to decide when and how much the event-driven programming paradigm should be used. Programmers can write programs in the same way they always have. Adding interface features such as mouse input to a text-oriented application represents a minor enhancement rather than a redesign of program logic.

These three design criteria—simplification, integration, and control—determine the general characteristics of Icon's graphics facilities, and result in several beneficial side-effects. For example, the simplicity of the graphics facilities results in increased ease of implementation; there are fewer functions and features that must be implemented in order to make Icon run on a new window system.

## Overview of the Graphics Facilities

Icon's graphics facilities provide a new data type, `window`, and operations on that new data type. There are several fundamental ways in which this type and its operations simplify the large number of types and functions needed for graphics programming and window management in most systems. This section presents key features of Icon's graphics facilities. See [Jeff94] for a complete description.

### Windows as terminals

Type `window` is an extension of Icon's file data type. Windows may be substituted for files in the language's existing file input/output operations. In such usages, a window operates in a manner similar to a computer terminal. A window has a more substantial internal state than most terminals, including a text cursor position (analogous to conventional terminal cursor position), the current window contents, as well as various font, color, and graphics style attributes that affect the appearance of output.

In addition to the window-as-terminal programming model, every window simultaneously supports graphics operations on a two-dimensional array of pixels. There is no mode-switching between text and graphics. Graphics input/output does not affect the text model in any way, and vice versa, except that output in either model overwrites and obscures prior output in the same location within the window. Viewing windows as "graphics terminals" is consistent with the model actually provided by window systems such as MGR [Uhle88] and $8\frac{1}{2}$ [Pike91].

There is no concept of window exposure in Icon; window repainting is handled automatically by the language. Retained windows are essential in providing the programmer with the freedom to organize program control flow in a manner that is appropriate to the application instead of requiring that organization revolve around window system events.

**Encapsulation of complex features in windows**

A call to an Icon function typically results in many underlying graphic system calls. Similarly, the underlying system objects used during window input/output operations, such as network connections, graphics attributes, and graphics context information are all packaged together into a single source-level window value. The programmer may ignore them and can expect reasonable default behavior. This approach makes simple applications easy to develop and allows gradual increases in sophistication and functionality as an application matures.

Encapsulating multiple system objects and composing higher-level operations from multiple system calls are techniques typical of higher-level toolkits and languages. Icon can be viewed as an extreme case of the use of these techniques: Window system independence and ease of learning are achieved by implementing all window system operations in terms of normal language values such as strings and integers.

**Graphics**

The graphics facilities provide the kind of generality and flexibility found in the rest of the Icon language. Icon's graphics functions all take an optional window parameter followed by an arbitrary number of arguments, and automatically convert arguments to the appropriate type (such as integer pixel coordinates). Special types for graphical entities such as points and rectangles are not employed. Many arguments may be omitted and default to values appropriate for the graphics function in question. For example, a circle is obtained by drawing an arc at a specific location and with a specific width; the height of the arc defaults to the width and the starting angle and extent of the arc default to produce a complete circle. This kind of defaulting behavior provides a concise programming notation while minimizing the number of functions the programmer must learn. A further example is provided by the use of the default window, &window. This global variable provides a default window argument to the various graphics drawing functions. The net result is that in order to draw a circle, the Icon programmer typically need only write

```
DrawArc(x, y, diameter)
```

Because this simplification is achieved by the provision of default values, it does not reduce the capability of the function repertoire; it merely allows simple operations to be specified in a simple way and in a notation that is consistent with related but more complex operations.

**Window Attributes**

An Icon window consists of a hierarchy of underlying window system components. For manipulation of these components Icon abstracts a complex set of internal structures into a single set of fifty six *attributes* and associated *values*. Attributes and values are queried and assigned by means of the function WAttrib(). For example, the call

```
WAttrib("height")
```

returns the height of &window in pixels. The attribute may be changed by following it with an equal sign and a value:

```
WAttrib("height=300")
```

No language is perfect; the attribute model has proven to be effective but the functional notation provided by WAttrib() is cumbersome. There are various shorthand notations in Icon for commonly-used attributes and combinations of attributes; a record-style window.attribute notation would improve the attribute mechanism in a more general way. In any case, the basic simplicity of the attribute-value model provides a conceptual framework that is independent of the window system and is easily learned.

### Canvases and Contexts

Although graphics systems vary widely in their programming interfaces, the abstractions underlying various attributes fall into two categories: *canvas* attributes describe the physical appearance of the window, such as its size and position on the screen, while *context* attributes affect the behavior of various operations, such as the colors used in drawing.

The most important advantage of distinguishing canvas from context is to allow multiple contexts, each with varying attribute values, to be used on a given window without the necessity to save and restore relevant attributes. The function Clone() provides this capability in Icon; Clone(w) returns a window value whose operations apply to the same canvas as its window argument w, but whose context is a distinct copy of the argument's context. Collections of clones are routinely used to provide direct "color by number" mappings from application-domain values to window values with varying color or font settings.

### Color and Font Naming

Colors and fonts pose some of the most serious portability issues in the Icon graphics facilities, because the variation in color capabilities and font support among platforms is large. The problem of variation in color naming is addressed by a standard set of color names defined by the language. Names such as "light blue" are converted into numeric RGB values which are passed to the system. The color names are inspired by a system proposed earlier [Berk82], with certain additions.

Fonts pose an even greater problem than colors for portable applications. Icon can use whatever fonts are available on a given system, but the application writer must determine what fonts to use on the systems the program will run on, or write code that works with user-selected fonts. To provide source code portability for typical applications, Icon defines four portable font names corresponding to the best available fixed- and proportional-width, serif and sans-serif system fonts. The portable names are typewriter, mono, serif, and sans and application programmers can expect them to be available in many type sizes.

## Examples

Some simple examples illustrate various capabilities and the underlying philosopy adopted for graphics in Icon. It is beyond the scope of this paper to describe the Icon language itself, so the examples here rely on the reader's familiarity with C or Pascal. More exotic Icon-specific idioms are avoided; language constructs that are used and have no analogies in C or Pascal are explained. The major contentions supported by the examples are that a broad class of windowing applications need not be event-driven or based on callback procedures, and that interactive applications based on keystroke input need not be rewritten in order to incorporate mouse input. These examples are necessarily short; many more sophisticated examples are freely available in the Icon Program Library and can be obtained as described below in the section on Availability.

### *Xm*: a File Browser

*Xm* is a trivial windowing version of the UNIX more(1) command. *Xm* displays a text file (its first argument) in a window, one screenful at a time, and allows the user to scroll forward and backward through the document. This simple version of *xm* is less flexible than more in most ways (it is written, after all, in less than thirty lines of code), but it gains certain flexibility for free from the graphics system: The window can be resized at any time, and *xm* takes advantage of the new window size after the next keystroke. Figure 1 is a sample screen image from *xm*.

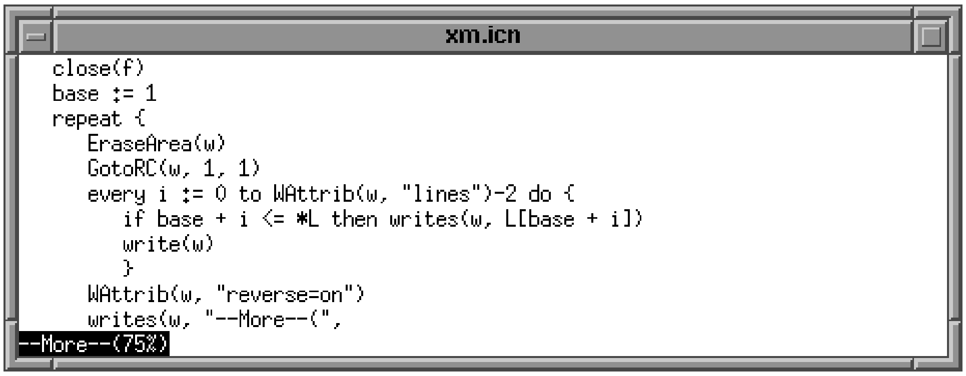


Figure 1: An xm window

*Xm* begins with the lines

```
procedure main(argv)
    if *argv = 0 then stop("usage: xm file")
```

The main procedure starts by checking the argument count (unary * is Icon's size operator) and halting with an error message if the program has been invoked with no filename. A more robust version of *xm* would handle this case in the proper UNIX fashion by reading from its standard input.

After its (scant) argument checking, *xm* opens the file to be read, followed by a window to display it in. File mode "g" denotes a graphics window rather than a conventional file.

```
    f := open(argv[1], "r") | stop("can't open ", argv[1])
    w := open(argv[1], "g") | stop("no window")
```

If either of these values cannot be obtained, *xm* gives up and prints an error message.

With an open file and window in hand, *xm* reads in the file. It reads in all the lines at once and places them in a list. The loop terminates when a call to read() fails on end-of-file. A more intelligent approach would be to read the file gradually as the user requests pages. This approach makes no difference for short files but is superior for very large files. Variable base stores the index of the top line on the screen and is initialized to 1.

```
L := [ ]
while put(L, read(f))
close(f)
base := 1
```

At this point, *xm* has done all of its preparatory work, and is ready to display the first page of the file on the screen. After displaying the page, it waits for the user to press a keystroke: a space bar indicates the next page should be displayed, the "b" key backs up and displays the preceding page, and the "q" key quits the application. The overall structure looks like:

```
repeat {
    # display the current page of text in the file
    # read and execute a one-keystroke command
    }
```

*Xm* writes out pages of text to the screen by indexing the list of lines L. The index of the first line that is displayed on the screen is remembered in variable base, and paging operations are performed as arithmetic on this variable.

```
EraseArea(w)
GotoRC(w, 1, 1)
every i := 0 to WAttrib(w, "rows") – 1 do {
    if base + i <= *L then writes(w, L[i + base])
    write(w)
    }
```

UNIX *more* writes a nice status line at the bottom of the screen in reverse video, indicating where the current page is within the file as a percentage of the total file size. Computing the percentage is done based on the last text line on the screen (base + WAttrib(w, "rows") – 1) rather than the first.

```
WAttrib(w, "reverse=on")
writes(w, "−−More−−(",
            ((100 > (base + WAttrib(w,"lines") – 2) * 100 / *L) | 100),
            "%)")
WAttrib(w, "reverse=off")
```

Keystrokes are read from the user using Icon's regular built-in function reads(). File functions such as reads() drop input activity that cannot be characterized in terms of characters. In order to enhance this application to support mouse input or explicit window resize handling, the call to reads() would be replaced by the more general function Event(), shown in the subsequent example programs.

```
case reads(w, 1) of {
    "q": break
    " ": base := (*L >= (base + WAttrib(w, "lines") – 1) | fail)
    "b": base := (0 < (base – WAttrib(w, "lines") + 1) | 0)
    }
```

*Xm* demonstrates that Icon demands little or no window system expertise of the programmer. Ordinary text applications can be ported to X with very few changes by adding a window argument to calls to functions such as read() and write(). After a program has been ported, it is simple to enhance it with features such as colors, fonts, and mouse handling.

### *Rfm*: dragging a rectangle on-screen

Brad Myers challenged a CHI '92 workshop on language support for graphical user interface programming with the question: "How hard is it to make a rectangle follow the mouse around on the screen?" Obviously a toolkit builder can hardwire such an operation as a library function, but this begs the question and at the same time engenders large libraries with difficult conceptual hurdles.

*Rfm* is a trivial Icon program in which a rectangle follows a mouse. A window is opened via file mode "g", for graphics; a loop reads user input and erases the rectangle and redraws it on each drag event. There are no deep abstractions nor is control flow implicit or inverted; aside from lexical information the main graphical concept employed is that of reversible pixel operations, described below.

```
procedure main()
    &window := open("hello", "g", "drawop=reverse")
    while 1 do
        if Event() === (&ldrag | &mdrag | &rdrag) then {
            # erase box at old position, then draw box at new position
            FillRectangle(\x, \y, 10, 10)
            FillRectangle(x := &x, y := &y, 10, 10)
            }
end
```

The attribute "drawop=reverse" is Icon's portable form of XOR-mode reversible pixel drawing. In a classical XOR-mode drawing, the foreground color is XOR-ed with the contents of drawn pixels; the advantage is that repeating the operation erases what was drawn, while the disadvantage is that on color displays, XOR-ing the foreground color with current pixel contents results in an output pixel whose color is undefined. Icon offers "drawop=xor", but also provides "drawop=reverse", a drawing operation where the XOR of the foreground and background is XOR-ed with drawn pixels, guaranteeing that pixels whose current contents are the foreground color become the background color, and vice-versa.

There are some Icon operators that may not be obvious in this example, but are not unintuitive once defined. Ampersands preceding identifiers such as &ldrag are not address-of operators, but rather indicate the use of an Icon keyword. &ldrag, &mdrag, and &rdrag are constant values for mouse events, while &x and &y are the coordinates of the mouse at the time of the event returned by Event(). The use of keywords, rather than returning a structure, is analogous to the common practice in compilers of returning attributes in global variables during lexical analysis.

The backslash operator used in \x and \y is a null-value test which in this program causes the first call to FillRectangle() not to be performed the first time through the loop. The vertical bar | is an *alternation* operator; when read aloud it is pronounced similar to an

OR. Icon's goal-directed expression evaluation mechanism allows alternative values (the constants associated with left, middle, and right drags in this case) to be produced as needed in an attempt to satisfy the surrounding expression (in this case, to make the equality test true). In C the best equivalent code might look like:

```
e = Event();
if (e == LDRAG || e == MDRAG || e == RDRAG) { /* ... */
```

The conditional expression in Icon is somewhat more concise.

**Fe: A Fisheye-View**

A fisheye view is a visual metaphor in which distortion is used to emphasize important portions of a view by allocating more screen space to them [Furn86]. Screen space is allocated to a given element proportional to its distance from one or more focus points, computed in a domain specific fashion. Fisheye views are useful in a variety of contexts, such as maps. *Fe* is a simple fisheye viewer for text files, such as might be used to emphasize the current line of execution during debugging. A sample view is presented in Figure 2.

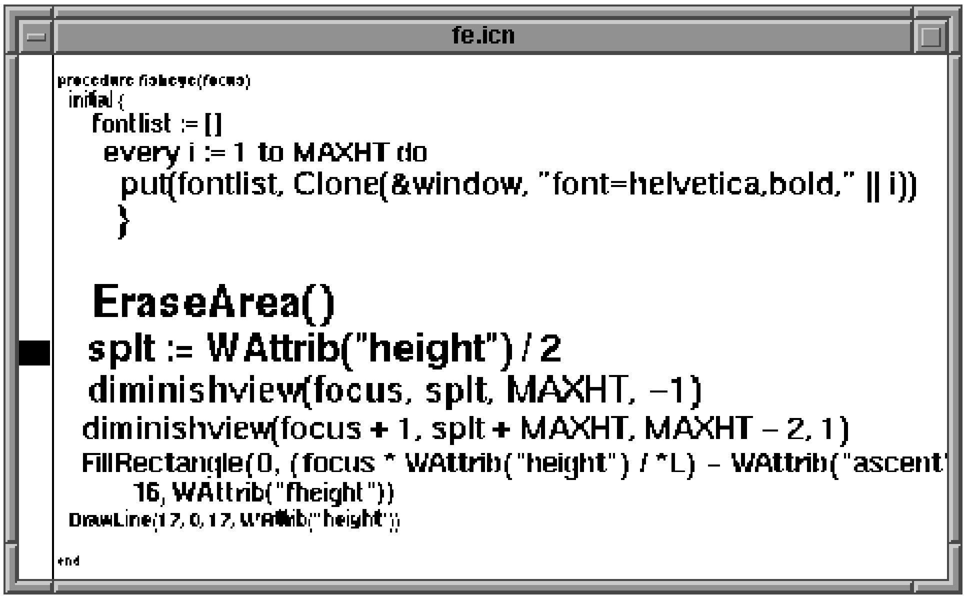


Figure 2: A fisheye view of some source code

The code for this example consists of a procedure main() and two subprocedures. The main procedure opens a window and reads an input file much as in the *xm* example presented earlier. The symbol MAXHT defines the pixel height of the largest text font, used to draw the line at the focus.

```
$define MAXHT 23
global fontlist, L
procedure main(args)
    fin := open(args[1]) | stop("no file")
    &window := open(args[1], "g") | stop("no window")
    flush(&output)
    L := [ ]
    while put(L, read(fin))
    focus := *L / 2
    fisheye(focus)
    # ... main() continues ...
```

*Fe*'s input handling consists of checks for mouse clicks in the scrollbar or "q" or the escape key to quit. Subprocedure fisheye() is called with the line number at which the focus (and the largest font size) should be directed.

```
    while 1 do {
        case Event() of {
            "q" | "\e": exit(0)
            &lpress | &ldrag: {
                if &x < 17 then {
                    focus := *L * &y / WAttrib("height")
                    fisheye(focus)
                    }
                }
            }
        }
end
```

The fisheye procedure creates a list of drawing contexts for the window the first time it is called, each with a different font size. It then clears the screen and calls the helping procedure diminishview() to draw the text. Finally, a scrollbar is drawn on the left side of the window indicating the position and proportion of the viewing area within the entire file.

```
procedure fisheye(focus)
    initial {
        fontlist := [ ]
        every i := 1 to MAXHT do
            put(fontlist, Clone(&window, "font=helvetica,bold," || i))
        }
    EraseArea()
    splt := WAttrib("height") / 2
    diminishview(focus, splt, MAXHT, -1)
    diminishview(focus + 1, splt + MAXHT, MAXHT - 2, 1)
    FillRectangle(0, (focus * WAttrib("height") / *L) - WAttrib("ascent"),
                  16, WAttrib("fheight"))
    DrawLine(17, 0, 17, WAttrib("height"))
end
```

The helping procedure diminishview() performs the actual text output, drawing each line in a successively smaller font size. It is called twice each time the screen is drawn, starting from the middle with the largest font and working towards the top or bottom of the screen as determined by the direction parameter.

```
procedure diminishview(focus, base, fontheight, direction)
    while 1 <= focus <= *L & 0 <= base <= WAttrib("height") do {
        GotoXY(20, base)
        writes(fontlist[fontheight], L[focus])
        base +:= fontheight * direction
        if 1 < focus < *L then {
            fontheight := max(fontheight – 2, 1)
            }
        focus +:= direction
        }
end
```

## Experience

Experience writing graphics programs with Icon has revealed some basic aspects of graphics programming that are relevant to language designers. Flexible argument handling discussed in the preceding section—sensible defaults, automatic type conversions, and variable-length argument lists—has proven useful in graphics as it has in Icon's other application domains.

Another language feature that is of great utility in graphics is control structure heterogeneity. Icon control structures, most notably the case expression, allow clauses of any data type, and different clauses may be of different types in the same control structure. In the domain of graphics programming, this allows simpler code than other systems usually require.

The case expression is the natural multi-way selection construct in many programming languages, but in processing window system input, it poses a problem. Input consists of different kinds of data; key presses are fundamentally different from mouse actions. A key press is naturally represented as a one-character string with the ASCII value of the key pressed, but such a representation is not so appropriate for mouse activity. Various forms of control and function keys present a similar representation problem.

Rather than employing a variant record or some other indirect encoding, Icon represents key presses as strings and mouse actions as integers. Related information such as the mouse position at the time of the input is delivered via associated keywords, a technique similar to that commonly used for lexical attributes in compilers. Since Icon's case expression allows values of any type in the case-selectors, no variant record is needed and little or no mental effort is involved in decoding input events or processing mixed key presses and mouse events.

## Conclusion

Icon uses a novel approach in the addition of graphics capabilities to a programming language with a conventional text-oriented input/output model. Icon succeeds in providing a smooth integration with pre-existing text operations and programming models. Experience with Icon has shown that simple windowing programs can be written in 10 to 100 lines, instead of the hundreds or thousands of lines required by many application program interfaces. Despite the limitations of a simplified programming model, Icon affords a concise notation with which to implement a wide range of graphic, window-based applications.

The approach used here to adding graphics to a programming language can be used for any high-level programming language. Some aspects of the design described here, such as the handling of colors and fonts, are directly applicable, while others fall into the category of making the extension fit naturally into the rest of language. In Icon, integration was facilitated by using capabilities such as generators and functions with an arbitrary number of arguments. Other languages offer different possibilities for the language designer.

## Availability

A large library of programs and procedures written in Icon already exists and provides a resource of reusable code for future applications. Icon and its library are in the public domain. Icon is available by anonymous FTP to cs.arizona.edu; cd /icon and get READ.ME for navigation instructions. Icon Project Document 255 [Jeff94] is a complete description of the graphics facilities described in this paper and a compressed PostScript copy can be obtained by FTP in the directory /icon/doc, file ipd255.ps.Z.

## Acknowledgements

Darren Merrill, Alex Jiang, and Barbara Wilkey contributed to the Presentation Manager, Macintosh, and MS Windows ports, respectively.

This work was supported in part by the National Science Foundation under Grants CCR-8713690 and CCR-8901573, a grant from the AT&T Research Foundation, and a UT San Antonio Faculty Research Award.

## References

[Berk82] Berk, T., Brownstein, L., and Kaufman, A. A New Color-Naming System for Graphics Languages. *IEEE Computer Graphics & Applications*, pages 37–44, May 1982.

[Furn86] Furnas, G. Generalized Fisheye Views. In *CHI '86 Proceedings*, pages 16–23, June 1986.

[Jeff94] Jeffery, C. L., Townsend, G. M., and Griswold, R. E. Graphics Facilities for the Icon Programming Language; Version 9.0. Technical Report IPD 255, Department of Computer Science, University of Arizona, July 1994.

[Pike91] Pike, R. 8 $\frac{1}{2}$, the Plan 9 Window System. In *USENIX Summer '91 Conference*, pages 257–265, June 1991.

[Uhle88] Uhler, S. A. MGR — C Language Application Interface. Technical report, Bell Communications Research, July 1988.

## Biographies

Clinton L. Jeffery is an Assistant Professor of Computer Science at the University of Texas at San Antonio. He received the Ph.D. degree in computer science in 1993 from The University of Arizona. His research interests include program execution monitoring, program visualization, and language support for graphics programming. He is a member of ACM, SIGPLAN, the IEEE Computer Society, and USENIX.

Ralph E. Griswold is a Regents' Professor of Computer Science at The University of Arizona. He received the Ph.D. degree in electrical engineering in 1962 from Stanford University. His research interests include non-numerical programming techniques, the design and implementation of high-level programming languages, and program visualization. He is the co-author of the SNOBOL, SL5, and Icon programming languages. He is a member of ACM, SIGPLAN, and the IEEE Computer Society.

Gregg M. Townsend is a Principal Research Specialist in the Department of Computer Science at the University of Arizona. He received the M.S. degree in Computer Science in 1984 from the University of Arizona. He contributed to the implementations of the Icon and SR programming languages; other interests include graphics, image processing, and software tools. He is a member of ACM, the IEEE Computer Society, and USENIX.