# A Scheme for Interactive Graphics

Brian Beckman

Computer Graphics Laboratory
Information Systems Engineering Section

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
brian@granite.jpl.nasa.gov

17 July 1990

# TABLE OF CONTENTS

# SUMMARY

**The Scheme programming language is an excellent basis for a graphics shell, or interpreted scripting language. Basing a shell on an existing language such as Scheme allows programmers to take full advantage of interactive programming while avoiding the distractions of language design and maintenance. Basing a shell on a high-level language such as Scheme allows the shell to be pressed into service for tasks far beyond the capabilities normally expected of a shell.**

**This paper presents a Scheme-based graphics shell called *GLeme* and several example shell programs: *objectize*, a graphical object caching system for high-performance rendering; the *receptionist*, a callback kernel for user interface code; *dfgraph*, a data flow executive; and *LINKS*, an embedded "power-user" language. These developments are all part of a scientific visualization software project.**

**General arguments are presented that a sufficiently flexible shell is a universal modeling and communication protocol that can cement together a whole suite of graphics programming tools.**

# INTRODUCTION

## GRAPHICS TODAY

More and more, computer graphics is becoming a mandatory aspect of interactive software. Color graphics hardware is so affordable that it is ubiquitous even among personal computers. Bit-mapped displays and networked graphics software are the norm for workstations. Standard graphics interfaces such as PostScript[1] and X Windows[2] are making it easier than ever for software developers to take advantage of that hardware.

Perhaps the most widespread and systematic use of graphics in interactive software is for user interfaces. *W*indows, *i*cons, *m*enus, *p*op-ups, and *s*croll bars (or WIMPS, for short) are all familiar examples of graphical controls common across applications. Many applications also use domain-specific graphics. For example, business applications produce charts and graphs; programming and computer-aided software engineering CASE tools produce structure and flow diagrams; and scientific packages produce plots and other visualizations.

## TOOLS FOR GRAPHICS SOFTWARE

Programmers must nowadays face the concern of graphics in addition to the familliar computational and structural concerns of software. Often, the graphics aspects of an application are much more complex than the other aspects. Appropriate tools can simplify the design and implementation of graphics software and help programmers to separate graphics from the other concerns.

Two kinds of graphics tools are necessary: interactive graphical editors (IGEs) and language tools. IGEs include such tools as modelers and paint-draw programs. Language tools include most graphics programming interfaces, such as PostScript, GKS, CORE[3], QuickDraw[4], and X Windows (see Mallgren[5] for formal approaches to graphics specification languages). These languages are usually embedded in a standard, general-purpose *host* language such as C or Fortran. For the purposes of this paper, language tools are further subclassified according to the nature of the host language. Graphics languages embedded in compiled languages have different uses than graphics languages embedded in interpreted languages. Figure 1 illustrates the taxonomy of these tool types.

IGEs—or “show what you mean” tools—are needed because they are much better for layout tasks than are language tools. Language—or “say what you mean”—tools are needed because they are much better for specifying repetitive, conditional, or mathematically patterned graphics than are IGEs (try drawing a regular seven-sided polygon, or better yet, a *k*-sided polygon, with a typical paint or draw program; this task is much easier to specify in a programming language). Language tools are also needed for automated tasks that run without human attention (IGEs are inherently interactive) and for integrating graphics subtasks into application programs.

Compiled language tools are needed because applications written in compiled languages must be able to specify graphics. Interpreted language tools are needed because both human users and application programs need the ability to issue graphics command streams to be rendered immediately by an independent, general-purpose process.

Table 1 summarizes the relative strengths and weaknesses of these tool types for several commonplace graphics programming tasks.

## OUTLINE OF THE PAPER

This paper is about interpreted graphics language tools, or graphics *shells*. The rest of this Introduction presents opinions and desiderata generalized from the specific experiences presented in the later sections of the paper. The following topics are discussed in general terms in the Introduction: (1) the role of shells in a well-equipped graphics software development environment, (2) the principles for engineering a shell, and (3) the applications of shells. The rest of the paper presents experience engineering a shell that evolved from an *ad hoc* scripting language into a comprehensive and coherent graphics programming environment. Examples of ambitious applications of that shell are presented in the last section of the paper.

## COOPERATION NEEDED

Graphics software is best produced in an environment in which IGEs cooperate with language tools. Each kind of tool has its domain of maximum efficacy, and using the wrong tool for a task can result in significant waste of time. Unless tools cooperate, however, one may be forced to use the wrong tool merely because it creates output in a required format.

Cooperation means that each kind of tool can work with the output format of the other kind. For example, a programmer might perform sketch-and-adjust tasks using a draw program, with which "debugging" a graphic is easy. Perhaps the programmer wishes to include a few mathematical shapes, such as regular polygons. With tools that cooperate, the draw program reads those in procedural code text format and integrates them with hand-drawn graphics. Then the programmer might *generate* procedural code for drawing the finished graphic from its *binary representations* maintained internally by the draw program. The procedural code can be invoked in application programs.

If a graphic needs revision, one might begin with the binary representation, edit it with the draw program, and regenerate the procedural representation. Alternatively, one might *convert* or *compile* an existing procedural text into a binary representation for direct manipulation with the draw program.

Normally, graphics application programming environments have both language tools and IGEs, but they are not compatible. They put out completely different formats, and no way is provided to translate between the formats. Figure 2 illustrates an environment in which the IGEs are isolated from the procedural tools. Figure 3 shows the cooperative environment advocated in this paper.

## INTERPRETERS NEEDED

Given the need for procedural graphics, why is an *interpreted* procedural language needed? Are not procedural representations in compiled languages, such as C and Fortran, enough? The answer is no, and the reason is that the interactivity and modularity of an interpreter is needed.

Given any sufficiently complex system, graphical or otherwise, a programmer needs to be able to perform certain tasks, namely

1. to issue commands and verify their effects immediately without having to write and manage a compiled program, a time-consuming and distracting affair involving multiple forms of the program (e.g. source, object files, and executables) and multiple tools (e.g. editors, compilers, and linkers);

2. to combine commands using the familiar programming idioms of sequencing, abstraction, selection, iteration, and recursion;

3. to view and edit data structures interactively (e.g. directories, for operating systems; display lists[3], for graphics);

4. to store, transmit, and invoke command streams in files (*scripts*);

5. to arrange concurrent or distributed processes so that streams generated in one process are interpreted by another (this allows the separation of the concerns of *graphics specification* and *graphics rendering and displaying*; the former is the province of the application program, and the latter is the province of the graphics shell interpreter).

These needs are the minimal functional requirements for every shell. One may summarize them by saying that a shell must be *interactive* (requirement 1), *programmable* (requirements 2 and 3), and *modular* (requirements 4 and 5).

## THE SHELL AS THE CENTRAL TOOL

Requirements 1, 4, and 5 can all be met in one stroke if a shell is independent of the source of its input, treating the keyboard, files, and other processes equally, as mere command streams. In UNIX™ parlance, such a shell is a *filter*.[6] It is assumed in the rest of this paper that all shells are filters in a UNIX or UNIX-like environment.

Figure 4 illustrates the concept of a graphics shell in a C programming environment. Since the shell is a filter, it is shown as equally accessible to programmers as to programs. It is shown layered on top of a C interface to the graphics hardware. The C interface is accessible both directly to programs and indirectly to programmers through a layer of C programming tools.

The universal, direct accessibility of the shell is its strength as a communication medium. Figure 5 shows a shell accepting input from concurrent and distributed processes, application programs, stored files, and the keyboard. Having a common language, the shell language, unifies the entire collection of tools. Since that language is textual, any command stream in that figure can be tapped, monitored, and verified by a programmer. Streams can also be passed through various transformation filters, which are easy to write since the protocol is text-based.

It may also be desirable to embed the shell itself in an application program, allowing the program to input and interpret graphics specifications. In general, it is a good idea for a subroutine "hook" to exist for *every* filter program, because if a facility is useful in an interactive modality it is likely to be useful when called from a program.

## THE META-SHELL CONCEPT

A clean separation can be made between the special-purpose commands and the generic programming support of a shell. The requirements listed above are not specific to the graphics commands of a shell. Rather, they address the generic programming support. Those requirements apply to any interactive tool providing access to *any* system, whether an operating system, a graphics system, a communication system, etc. Thus, the requirements can be thought of as the functional requirements for a general-purpose core language around which special-purpose shells may be built—a *meta-shell*. Figure 6 illustrates this two-layer architecture. This figure is an elaboration of figure 4 and shows the internal structure of the shell.

## ENGINEERING A SHELL

Graphics application programming environments are *not* usually supplied "out of the box" with graphics shells, so programmers must create them. A typical approach is to craft a simple command-sequence interpreter from scratch (see Kernighan and Pike[6] and the section "Evolution of a Graphics Shell" below for examples of this approach).

A command-sequence interpreter allows a programmer to enter graphics commands from the keyboard, files, and other processes, and to see the graphics rendered immediately on a display device. So requirements 1, 4, and 5 are satisfied at the outset. Such an interpreter is, in fact, sufficient for many applications. As it is put to more and more demanding use, however, one finds that more and more features, such as arithmetic, variables, functions, conditionals, loops, type definitions, etc., are desirable. In other words, one "discovers" meta-shell requirements 2 and 3.

Usually, one incrementally adds these constructs to the existing command language. This is quite a reasonable approach, given that requirements 2 and 3 were not foreseen, since it allows one to take advantage of existing code and averts a full-fledged language design task. However, the process of grafting in new constructs requires bending and stressing the syntax and semantics of existing constructs to make the new ones fit. Before long, one has a severely weakened infrastructure. In fact, many operating system shells have the obvious syntactic patchworks that betray such a software engineering history. Most shells are so weak or so ugly as programming languages that they are not used in practice for much beyond sequencing despite all the effort that has gone into extending them.

## IN LIEU OF INVENTING A NEW LANGUAGE

It is much better to anticipate the programming features that a shell will ultimately need. By choosing them carefully, an entire range of users can be supported, from those needing only command sequencing to those doing very advanced programming. The effort of supporting those programming features can be minimized if the shell is based on an existing, well-documented, extensible, interactive language. Programmers need not spend time and energy reinventing ordinary programming constructs.

## SCHEME FOR A META-SHELL

Scheme[7, 8, 9, 10, 11] is one of the simplest, cleanest, and most expressive programming languages in existence today (see Steele's foreword to Springer and Friedman[11] for support of this opinion). To quote Scheme's defining document:

> Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and [only a] few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and message passing styles, find convenient expression in Scheme.[9]

From the functional point of view, Scheme seems to be the ideal choice for a meta-shell. Those users who need nothing beyond sequencing have almost no syntax to learn. Those in need of high-level programming support can find the necessary building blocks. Being based on the lambda calculus, Scheme is closer to the theoretical foundations of programming than other interactive languages such as BASIC and Forth.[12, 13] Recursion and procedural abstraction are supported in an exceptionally direct way.

However, certain practical requirements, in addition to the functional requirements already articulated, must be met before any language can be chosen for a meta-shell. To be practical, a meta-shell must (1) be easy to implement and maintain, (2) be portable and reusable, (3) have acceptable time and space performance, and (4) be easy to extend or customize.

It so happens that a fair number of implementations of Scheme satisfying both these practical requirements and the aforementioned functional requirements are readily available either in the public domain or with such liberal licence terms that the development of research software is not hindered. The next main section of this paper relates experiences using one of these implementations to build a graphics shell for the Silicon Graphics Personal Iris™ GL™ graphics library. This shell, dubbed *GLeme* , comprises 327 constants and 374 procedures devoted to graphics and layered on top of Scheme. The implementation of the graphics layer was accomplished by the author in a few days' time.

### Using C and GLeme Together

To build GLeme in the first place, C code was added to the core Scheme interpreter to provide graphics primitives. Due to the flexibility and quality of the Scheme implementation, adding new primitives proved to be easy. Likewise, application-specific commands can be easily added to GLeme, interfacing it to application-specific C libraries. In this way, computationally demanding algorithms (e.g. rendering) can be coded in C, and structurally demanding algorithms (e.g. GUI management) can be coded in GLeme. This composite style allows each language to be used for the kind of programming it was designed for, giving the programmer maximum problem-solving leverage. The complementarity of C and GLeme is summarized in Table 2. This composite style is made attractive by the customizability and extensibility of the GLeme implementation.

## SOFTWARE ENGINEERING WITH A SHELL

Once one has recognized that a graphics shell can have all the capabilities of a full, high-level, interactive programming language, one's ambitions for it can be considerably expanded. The next few paragraphs explain how a high-level interactive language is useful for software engineering problems common to all complex software, whether they be graphics or otherwise.

### Modular software architecture

Most interactive graphics applications are very complex. The mere presence of a graphical user interface (GUI) forces otherwise unnecessary complexity on an application program. The GUI management aspects of an application can easily dominate the code. Many graphical systems require that all applications take event-driven form. This form "inverts" the traditional control structure of an application, putting the interface units in command of the functional units.

Modular programming, whether object-oriented or based on other paradigms, is the preferred technique for managing complexity. A high-level interactive language allows a programmer to "sketch" out an architecture incorporating all the GUI elements required in the finished application. The sketched architecture can be interactively tested simply by executing it in the shell.

### Experimental design

There are often many alternative ways to solve a particular software problem. Frequently, experimentation is the best way to decide among the alternatives. A high-level shell language can make experimentation very inexpensive by providing sufficient infrastructure that the programmer can try out ideas without having to invest time in building or acquiring that infrastructure. Such infrastructure includes automated memory management, runtime type and range checking, and easy ways to build ordinary data structures such as lists, stacks, and trees.

### Programmer education

Developers face a rather daunting learning problem with today's graphics systems. A comprehensive software package supporting merely two-dimensional (2D) user interface graphics can run to thousands of procedures, data types, and named constants (see the Macintosh[4] and X Window System[2] documentation for examples). Three-dimensional (3D) graphics systems have even larger interfaces. Programmers must spend significant time learning these interfaces, and managers must pay them to do it.

Furthermore, there are two levels to the learning problem. First, a programmer must incorporate the new interface into the "recognition vocabulary" deeply enough to recognize interface elements and look them up in the documentation. Second, a programmer must incorporate the new interface into the "expressive vocabulary", that is, deeply enough to know common idiom and to invent creative, efficient, and elegant applications.

The recognition level can be achieved quickly, often in a few weeks if there are tutorials and examples to study. The expressive level is much deeper, requiring extensive trial-and-error experience over several months. Essentially, learning a modern graphics interface is learning a new, and typically quite large, graphics language. An interactive shell is an invaluable tool for learning since it gives a user immediate feedback and makes exploration easy.

## Rapid prototyping

It is often the case that software users cannot clearly articulate their requirements. They say something like “I can’t tell you what I want, but I’ll know it when I see it.” Programmers must be able to produce conceptual demonstration programs—rapid prototypes—very quickly and cheaply. A prototype exhibits candidate functionality to the user, who then criticizes and suggests changes. The programmers produce another prototype, the users criticize, and the process iterates. A high-level, interactive language can be invaluable in this process since it allows faster, hence cheaper, programming.

## Program specification

Once the design of an application has been finalized, it must be communicated to implementation programmers. Expressing a design in an actual programming language allows a designer to test the design by executing it before committing it to an implementation. Thus, shell code can be used as executable design documentation.

# BACKGROUND KNOWLEDGE

The next two sections of this paper go into detailed descriptions of the engineering and the applications of GLeme. For deep understanding of this material, the reader must know Scheme or another Lisp dialect. Those readers without this background can achieve an overview-level understanding simply by knowing that Scheme is a prefix notation language. Expressions are either atoms, like numbers or symbols, or procedure applications. Procedure applications are denoted by nested parenthesized expressions with the operator in the first position followed by zero or more arguments, as in (*operator*□*operand*□). The operator and each of the operands are in turn expressions, and the same evaluation rules apply to operators and operands. Thus, the value of an expression can be a procedure, which is a first-class data type, equal in status to numbers, symbols, strings, etc. So, for example, the expression (+ 1 2) evaluates to 3, the symbol + evaluating to the *add* procedure, and the numerals 1 and 2 evaluating to themselves.

Most graphics expressions are *commands* since their primary purpose is to generate side effects on the display screen or graphics state. Their return values are often unimportant. For example, the following command sequence would cause the pen to move to the origin of a 3D space and then to draw a line to the point (10,□20,□30).

```
(move  0  0  0)
(draw 10 20 30)
```

The return values of these expressions are not used.

Detailed knowledge of the graphics library GL is not needed. Any GL entry points used in the examples are explained in context.

# EVOLUTION OF A GRAPHICS SHELL

One of the first things a graphics programmer does when faced with a new machine or a new interface to learn is to build an interactive shell for it. This allows the programmer to experiment with and test the hardware or interface and to begin designing libraries and applications. This section gives an example of a series of shells that evolved from a simple, line-oriented sequence interpreter to a full programming environment based on a reusable Scheme meta-shell.

## *PAD*

The staff of the CGL (the Computer Graphics Laboratory of the Jet Propulsion Laboratory) had built several shells from scratch over the years for a number of different graphics interfaces. One simple shell, *PAD*, now defunct, was a line-oriented interpreter accepting prefix expressions of the form

```
command operand1 operand2 ... <carriage-return>
```

with *command* representing one of the machine- or library-dependent graphics entry points, e.g. *draw-circle* or *set-color*, and the operands being numerals. Command symbols were looked up in a hash table for speed. Each symbol was associated with a library procedure that almost always had the same name in order to have direct correspondence with the interface reference documentation, thereby alleviating the need for lengthy shell documentation. When a symbol was read by the interpreter, its corresponding procedure was looked up in the hash table. After being found, a procedure was executed immediately with the given operands. Each command had its own operand specifications and semantics. For example, *draw-circle* took a center coordinate pair and a radius, all floating point numbers. *Set-color* took three integer red-green-blue components in the range 0-65535.

The user could initiate a keyboard session simply by typing *PAD* at the UNIX prompt. Since *PAD* was a filter, the user could also initiate a playback session, taking input from the file *script* by typing

```
PAD < script
```

in accordance with the normal UNIX syntax for input redirection. *Script* could contain an arbitrarily long sequence of *PAD* expressions specifying an arbitrarily complex rendering or animation. The user could also cause output from another process, say a modeler or animation previewer, to be interpreted and displayed in real time by typing

```
modeler | PAD
```

*PAD* was actually used in this way for graphical monitoring of an experimental operating system[14] and for visualizing a military simulation.[15]

## *PAD* IMPROVED

The next version of *PAD* included simple infix arithmetic, so that each operand could be either a numeral or an expression generated by the following BNF production:

```
expr :- (expr arith-op expr)
```

with *arith-op* being one of +, -, *, /, % (*remainder*), or ^ (*exponentiation*). This simple arithmetic grammar could be processed with a near-trivial hand-coded recursive descent parser.

The next increment in *PAD* functionality was a higher-order command, @, that, given a filename operand, took commands from the file, returning control to the calling context on end-of-file. Naturally, @ inclusions could be nested to arbitrary depth, so @ provided a primitive procedural abstraction capability, with procedures represented by script files. A script could have formal arguments named *#1*, *#2*, ..., *#9* for their positions in the argument list. The script would be invoked with *actual* arguments as follows:

```
@ script actual-arg1 actual-arg2 ...
```

In combination with the arithmetic capability, positional arguments allowed a script to specify a position-independent graphic. For example, a script named *box* took the coordinates of an origin, a box width, and a box height and might have been called like this:

```
@ box 10 10 50 50
```

to draw a box with the upper left corner at position (10,□10), with width 50 and height 50. The script was implemented as follows ("--" introduces a comment line):

```
-- box x-origin(#1) y-origin(#2) width(#3) height(#4)
move #1        #2
draw (#1 + #3) #2
draw (#1 + #3) (#2 + #4)
draw #1        (#2 + #4)
draw #1        #2
```

Several later versions of *PAD* were developed that added more and more capability. The evolution of *PAD* is somewhat similar to the evolution of higher-order calculators in chapter 8 of Kernighan and Pike.[6]

## CRITIQUE OF *PAD*

The original *PAD* interpreter had the advantage that its implementation was extremely simple, hence easy to debug and to port. The adapting of *PAD* to a new machine or graphics interface was partially automated. A generic version of the interpreter, called *PAD-boilerplate*, could be ported in less than a day.

But *PAD* also suffered from severe limitations. Its only data type was *number*, and it had no data structures, so it was inhospitable to interfaces requiring array or record arguments. Its only programming support was its primitive arithmetic and script abstraction. Its only control structure was the sequence. Recursion was not possible since there were no conditional statements.

The author realized that some of the features and goals of PAD were worth saving. Those were (1) reusability of the core language (the meta-shell concept); (2) easy adaptability to new interfaces; (3) simplicity and, therefore, portability; (4) interactivity; and (5) modularity—the ability transparently to take input from files, the keyboard, and processes. It was equally clear that the core programming language of *PAD* needed improvement. The needed features were (1) selection, iteration, and recursion (abstraction and sequencing already existed to some degree); (2) data structures including arrays, records, lists, and user-defined data types or classes (the data structuring facility needed to be at least as rich as that of the C language, the implementation language of *PAD* and also the host language for most graphics libraries of interest to the CGL); (3) a library of reusable, dynamic, linked, polymorphic data structures; and (4) support for modular programming, if not for data abstraction or object-oriented programming.

## A NEW SHELL NEEDED

The LinkWinds[16] project provided stimulus for another round of graphics shell development in the CGL. LinkWinds is an interactive exploratory data analysis and visualization system for scientific data. Both 2D and 3D images synthesized from scientific data are displayed in multiple windows. Image windows are called “visuals”. Control windows implementing *virtual scientific instruments* are linked to the visuals (hence the name “LinkWinds”). Controls allow the user interactively to vary parameters and to see the immediate effect on the visuals. The links among windows are dynamic and user-configurable. LinkWinds contains a configuration environment, which is a simple data flow visual programming tool. It has an intuitive interface; the novice user does not need knowledge of data flow theory, though the advanced user can take advantage of that knowledge.

LinkWinds was developed on Iris workstations with the GL graphics library, which has interfaces to C, Fortran, and Pascal. This library directly operates the high-performance 3D hardware of the workstation. It is documented in manuals supplied with workstations.[17,18,19]

LinkWinds required many sophisticated software constructs, and the demands of LinkWinds provided impetus for the work in this paper. An embedded data flow executive was needed to manage the dynamically linked visuals and controls. A GUI system was needed not only for WIMPS, but also to support design and management of on-screen virtual scientific instruments. An embedded scripting language was needed to support sophisticated users who wanted to bypass the interactive-only GUI and program LinkWinds directly.

## GLEME

For some time, a customizable C/C++ interpreter was considered both for a rapid prototyping tool and for the embedded scripting language of LinkWinds. Significant experimentation was done to estimate the resources needed to develop one. Such an interpreter would have given LinkWinds developers the advantage of being able to prototype in the language of the final implementation of LinkWinds. However, the practical experience of the CGL was that C and C++ programmers spend a great deal of time programming such things as stacks, lists, trees, and variable-length strings. A language that supported these “ordinary data structures” in a higher-level way was more desirable.

It was realized that some dialect of Lisp would serve. The prospect of using Lisp was somewhat alarming to some programmers on the team because of the general reputation of Lisp for being large, slow, complex, and difficult to learn. However, the author was fortunate to discover the Scheme dialect which is expressly designed to be small, fast, simple, and easy to learn. The author also found a high-quality Scheme interpreter with good performance. The next step was somehow to combine Scheme and the large, C-callable GL library.

GLeme was the result. It is (nearly) standard Scheme plus a complete, Scheme-callable interface to GL. It is based on the XScheme interpreter-compiler written by David Betz.[20] Several XScheme enhancements and a dynamic loader for C routines were generously provided to the author by Eric Raible.[21]

XScheme is written in C. It implements a substantial subset of Scheme (exact-inexact numbers and “bignums” are not implemented). It is easily ported, containing very little machine-dependent and operating system-dependent code. It is also easily *customized* with new routines written in C to be called from Scheme and *extended* with new routines written in Scheme itself. Most GLeme graphics primitives were implemented in C, though some have a little Scheme “helper” code. The C “glue” between Scheme and GL is simple and follows a repetitive pattern.

## DETAILS OF GLEME

The following description of GLeme shows how a large graphics library can be quickly embedded in Scheme, given only an embedding in C. This description not only explains how GL is bound to Scheme, it also explains a few GL functions and idioms relevant for later examples.

It was thought desirable that a programmer should be able to use GLeme almost immediately, without first acquiring a great deal of new knowledge. Thus, a programmer can run GLeme and immediately and interactively execute graphics commands. The following example shows a simple command sequence one might type at the keyboard just to get started with GLeme. These commands draw a green rectangle on a red field in a 300 by 300 window.

```
(prefposition 100 400 100 400)  ;; set window pos. attributes
(winopen "foo")                 ;; open new window and make current
(color red)                     ;; set graphics state variable
(clear)                         ;; clear to current color
(color green)                   ;; set graphics state variable
(rectf 100 100 200 200)         ;; filled rectangle in current color
(winclose (winget))             ;; close current window
```

Note that this example requires no profound knowledge of Lisp programming or idiom. Thus, since such elementary sequences are adequate for many purposes, users may get started slowly, learning only what they need at each session and working slowly up to advanced usages.

To keep GLeme easy to learn, it was desirable that it not require lengthy documentation. So, the GLeme interface to GL is kept formally close to the C interface, which is thoroughly documented. A few rules sufficed for a programmer, knowing only the C interface to a GL function, to deduce how to call the function from GLeme. The following paragraphs give these rules.

## RULE 1.

**For every GL library C function there is a GLeme version with the same name.** For example, corresponding to the C function *backbuffer*, which is called like this:

```
backbuffer (TRUE); —or— backbuffer (FALSE);
```

there is the GLeme command *backbuffer*, called like this:

```
(backbuffer #t) —or— (backbuffer #f)
```

## RULE 2.

**If a C function takes numerical arguments, then the GLeme version takes numerical arguments.** For example, the C function *draw* takes three *Coords* as arguments, as in:

```
draw (10., 20., 30.);
```

Likewise, the GLeme command *draw* takes three numbers, like this

```
(draw 10 20 30)
```

Notice that GLeme automatically promotes the actual integer arguments to the required floating point types before calling GL. Conversely, the following is an error in GLeme:

```
(drawi 10.1 20.2 30.3)
```

because it specifically calls the GL function *drawi*, which requires integer arguments, and yet passes explicit floating point arguments. Although GLeme automatically *promotes* numerical data types from integer to floating point, as in the *draw* example, it does not *demote* types, as in the *drawi* example, because it might destroy information by so doing.

## RULE 3.

**If a C function takes a Boolean argument (zero/non-zero), then the GLeme version takes *#t* or *#f*.**

## RULE 4.

**If a C function takes an array (or, more precisely, a pointer to the first element of the array [see ch. 5 of Kernighan and Ritchie[22]]), then the GLeme version takes a Scheme vector.** For example:

```
C:  float a_poly[][3] = {2.,  1.,  0.,     1.,  3.,  0.,
                         2.,  5.,  0.,     4.,  5.,  0.,
                         5.,  3.,  0.,     4.,  1.,  0. } ;
    polf (6, a_poly) ;  /* draw filled polygon */

GLeme:
    (polf 6 #(#(2 1 0) #(1 3 0)
              #(2 5 0) #(4 5 0)
              #(5 3 0) #(4 1 0)))
```

Again, note the automatic promotion of the actual argument types from integer to floating point. Also note that the temporary variable *a_poly*, necessary in the C version, is not necessary in the GLeme version. One can just write the vector argument in-line. Of course, one might also use a variable just as in the C version.

Since the length of a Scheme vector is automatically recoverable from the vector, the argument 6 is not strictly necessary in the GLeme version. However, it is retained so that the GLeme version of *polf* has the closest possible correspondence with the documented C version. The same holds true for other GL functions. Arguments that may not be strictly necessary in GLeme are retained for formal correspondence with C.

## RULE 5.

There is a major exception to rule 4 in regards to the GL matrix stack, which stores 4×4 homogeneous coordinate transformation matrices for viewing, projection, and modeling. This stack is manipulated by GL entry points such as *pushmatrix* and *popmatrix*. **A GLeme graphics matrix is a list of lists, not a vector of vectors.** This is so that graphics matrices can be manipulated with an elegant library derived from that in chapter 3 of Abelson and Sussman.[7]

## RULE 6.

**The C interface definition files *gl.h* and *device.h* have GLeme equivalents, defining over 300 symbolic constants, that are automatically loaded at startup.** This means that one may write

`(qdevice LEFTMOUSE)` —or— `(color RED)`

just as in C.

## RULE 7.

**If a C function outputs multiple values, the corresponding GLeme function returns a list with the corresponding values.** A C function can only return a single value. So, if multiple values must be put out by a function, all but at most one must be loaded into addresses supplied explicitly in the argument list.[22] So, for example, the C function *getorigin*, which returns the global coordinates of a the local coordinate frame of a window, must be called like this (*&* is the C address-of or referencing operator):

```
long x, y;
getorigin (&x, &y) ;
```

A Scheme procedure, on the other hand, can return an aggregate data structure, such as a list. The GLeme version of *getorigin* should be called like this

```
(getorigin)
```

which returns—or evaluates to—the list (*x y*).

One would usually use *getorigin* in a *let* or *let** expression, which defines local variables in a basic block. The following is idiomatic:

```
(let ((x (car  (getorigin)))        ;; let x be (car  (getorigin))
      (y (cadr (getorigin))))       ;; let y be (cadr (getorigin))
 ;; other code referring to x and y ;;
```

The procedure *car* returns the first element of its list argument; the procedure *cadr* returns the second element of its list argument; and *getorigin* evaluates to list (*x y*).

One might encounter the improbable happenstance that the user moves the window in the instant between the two calls of *getorigin*, resulting in inconsistent values of *x* and *y*. This problem can be solved using a *let** expression, which is a "sequentialized" *let*, to assign the list returned by *getorigin* to a temporary variable, which is then separated into atoms with *car* and *cadr*:

```
(let* ((temp (getorigin))
       (x    (car temp))
       (y    (cadr temp)))
 ;; other code referring to x and y ;;
```

Of course, GLeme allows a functional style, so that one may write (*car* (*getorigin*)) for *x* and (*cadr* (*getorigin*)) for *y* wherever those values are needed, with the proviso that if the user moves the window, the results of *getorigin* change dynamically.

If a C function both returns a value *and* loads addresses given in the argument list, then the equivalent GLeme version again returns a list. The first element of the returned list is the value returned directly by the C function. The rest of the returned list contains the values that the C version loads in the addresses in its argument list. For example, consider the GL function *qread.* This function reads a device number and a corresponding device value out of the hardware queue. It is used, for example, to get mouse positions, button clicks, keyboard hits, and hardware dial positions. In C, *qread* must be called as follows:

```
long theDevice;
short theValue;
theDevice = qread (&theValue) ;
```

The equivalent GLeme code might be:

```
(let* ((q         (qread))  ;; qread returns (device value)
       (theDevice (car q))
       (theValue  (cadr q)))
  ;; ... more code
```

In this case, one must use *let** to ensure that *qread* is called only once since it has the side effect of deleting an element from the event queue hardware. *Getorigin* has no side effects, so it can be used in a functional style, whereas *qread* cannot.

## VERIFYING GLEME

The Iris manuals present many tutorial C programs. Versions of these same programs coded in GLeme were used to test the implementation. The testing criterion was that in all cases a GLeme program behave identically to the corresponding C program, except for performance (see the section on performance below).

Most of the tutorial programs are very straightforward translations from C to GLeme, but some are more adventurous excursions into Scheme idiom. Frequently, the biggest difference between a Scheme program and the equivalent C program involves iteration. C encourages index loops over arrays, while Scheme encourages recursion over lists. To make it possible for Scheme programs to resemble their C counterparts more closely, a Scheme *for* loop is provided. This is not standard Scheme, but rather a *syntax extension* or macro (see Dybvig[8] and Kohlbecker[23] for the theory and implementation of syntax extensions). Briefly, there are ten forms of the *for* loop, which are shown in table 3 with their C equivalents. Naturally, these Scheme *for* loops may be arbitrarily nested. They are implemented as simple "named *let*" constructs (see Appendix 1 for the implementation of *for* and Dybvig[8] or the Scheme Report[9] for description of the named *let*).

As one becomes more accustomed to Scheme programming, it is worthwhile to abandon C-like index loops over arrays in favor of the more idiomatic recursions and *map*, *for-each*, and *apply* constructs over lists. The reason is that the latter are much faster, at least in XScheme.

## PERFORMANCE OF GLEME

Space and time performance is critical in computer graphics. Space efficiency is required in scientific visualization, since data sets and graphical display lists can be extremely large. Speed is required since users want to manipulate views interactively to explore objects.

Some informal performance experiments were done with GLeme, comparing the time-to-completion of example programs from the GL manuals written in C with their GLeme equivalents. Brevity prevents going into details, but suffice it to say the performance is comparable for programs that set up a model to render and then call GL to perform the rendering in hardware. This is to be expected since, in such programs, the computationally demanding parts are done in hardware. Interactive programs that take parameters in a loop from the mouse or dial box, say to allow the user to move an object or the point of view, also tend to have comparable performance. Both C code and GLeme code can respond faster than the user can move the mouse. Where GLeme falls short of C is in code with critical loops in software. GLeme code is typically three to five times slower than the equivalent C code and also sometimes exhibits a "heartbeat" phenomenon due to periodic garbage collection. This was taken to be a surprisingly good showing on the part of GLeme, actually. Nevertheless, with the *composite style* of programming detailed below, performance-critical code can be coded in C, and excellent performance can be achieved under all circumstances.

It is argued in this paper that if an interactive shell has sufficient programming power, it can be used for more ambitious tasks than normally are ascribed to scripting tools. In some of those roles, such as rapid prototyping and program specification, performance of the shell is not a primary issue (within reason). However, if the shell is used in a real application, one must be careful to employ it in such a way as to preserve performance. This is especially important when using Lisp dialects, some implementations of which are fat and slow.

# EXAMPLE APPLICATIONS

In this section, some of the applications of GLeme in the LinkWinds project are presented. The purpose of these examples is to show the kinds of ambitious tasks that can be fearlessly approached with a strong shell language.

The first example is the simplest. It also is one of the most sophisticated, demonstrating an advanced Scheme programming concept—transforming one procedure into another one. This example is a display list caching mechanism called *objectize* that transparently replaces an application drawing procedure with a hardware call that performs the rendering at maximum speed.

The next example is the largest and most ambitious. It is a callback kernel named *receptionist* for managing mouse, keyboard, and other device events. This kernel allows GUI code to be structured as a collection of event-handling modules that communicate with the hardware through the receptionist and amongst one another according to a separate, user-defined protocol. This example relies on a straightforward object-oriented programming simulator, *methods*, which is presented in Appendix 2.

The next example is a data flow executive called *dfgraph* that also uses *methods*. This executive allows data flow programs to be constructed and executed dynamically, say through a graphical editor.

Finally, LinkWinds was given its own, user-level scripting language, *LINKS*, by embedding GLeme in LinkWinds. LINKS allows the expert user to bypass the graphical user interface and to program desired operations directly (the arguments in the Introduction that both interactive tools and programming tools are needed by programmers apply equally to the user level). Embedded scripting languages appear in many established applications such as emacs, AutoCAD™, and HyperCard™.

## *OBJECTIZE*

This example implements a caching mechanism for graphical procedures. This mechanism allows application code to take advantage of the high-performance display list hardware of the Iris and yet keep a natural structure. A natural structure is one in which graphical objects are represented by procedures that draw them.

*Objectize* is a second-order procedure: it takes a drawing *thunk*, or procedure of no arguments, and returns a new thunk. The new thunk creates a *GL object* in the display list hardware the first time it is called and causes maximum-performance rendering of the object at the hardware rate by invoking the GL object on the first and subsequent calls. In the GL context, *object* is a technical term referring to a graphical hardware display list element and is not to be confused with any other kind of object.

*Objectize* works by creating a hidden internal variable, *obj*, which is initialized to nil, and returning a new, anonymous thunk, created by a (*lambda* () ...) construct, that does the following. If *obj* is nil, which is true only on the first call, it creates a unique object identity number by calling the GL entry point *genobj* and assigns this number to *obj*. It then appends this identity number to a global list for later “garbage collection”. It commands the hardware to accumulate drawing instructions in a display list by calling the GL entry point *makeobj* and the user-supplied drawing thunk. Then, it commands the hardware to stop accumulating instructions through the GL entry point *closeobj*. Finally, whether the object was just created or not, the object is rendered by calling the GL entry point *callobj*.

Subsequent calls of the "objectized" drawing thunk justify the overhead of creating the hardware object since rendering is done at maximum speed. The garbage collection helper procedure, *gc-objects*, removes all objectized thunks from the hardware by calling the GL entry point *delobj* for every element of the global object list and resetting that list to nil. This procedure must be explicitly called by client code. A call of any prior objectized thunk after garbage collection causes a hardware error.

```
(set! //oblist// '())

(define (objectize draw-thunk)
  (let ( (obj '()) )
    (lambda ()
      (if (null? obj)
        (begin
          (set! obj (genobj))
          (set! //oblist// (cons obj //oblist//))
          (makeobj obj)
            (draw-thunk)
          (closeobj)))
      (callobj obj))))

(define (gc-objects)
  (for-each delobj //oblist//)
  (set! //oblist// '()))
```

For an example of using *objectize*, suppose one has the following drawing procedure that draws a green-filled rectangle:

```
(lambda () (color GREEN) (rectf 100 100 200 200))
```

If not using *objectize*, one would normally assign this procedure to a global variable:

```
(define my-rect (lambda () (color GREEN) (rectf 100 100 200 200)))
```

and then call the procedure wherever it is needed, as follows:

```
(my-rect)
```

Using *objectize*, the definition of *my-rect* is only slightly different:

```
(define my-rect (objectize
                 (lambda () (color GREEN) (rectf 100 100 200 200))))
```

and the calling syntax is identical. The objectizing is thus said to be transparent; the calling code cannot determine whether drawing procedures have been objectized.

## CLASS *RECEPTIONIST*

The receptionist is an intelligent machine interface for handling unsolicited input events in interactive programs. The events are generated by interactive devices, such as a mouse or the keyboard. The receptionist is a callback kernel that allows the application to be written as a modular collection of event-handling units. These *handlers* can be guarded by *filter* predicates that exclude unwanted events (this use of the term *filter* is not to be confused with the UNIX filter—it refers here to a predicate procedure). The receptionist keeps a dynamic database of filter-handler pairs called *workers* and passes control to them when appropriate events occur. The receptionist is modeled after a human receptionist in an office building who greets all visitors and dispatches them to appropriate workers.

## The Event Loop Style Discouraged

The receptionist is designed to be used with systems organized around a central hardware event queue that stores data from all input devices until solicited by an application program. Examples of such systems include the Macintosh™, X Windows, and GL, so they are quite common. The receptionist is layered on top of the event queue management software of the system.

Such systems encourage an *event-loop* style of programming in which the application revolves around an infinite polling loop of the following form:

```
forever
{
    get next event from queue
    switch (event kind)
    case mouse:
        if mouse in menu
            do menu event
        else if mouse in window
            do window event
        else if ...
    case keyboard:
        if character completes a token
            interpret token
        else if character is a command code
            perform command
        else if ...
    case ...
}
```

Such a form suffers from a number of weaknesses. The event loop, which is a central, monolithic structure, tends to grow in size and complexity as the application evolves. There is a temptation to insert new code in-line in the event loop. Loops running to several hundred lines are not uncommon in practice. The loop can become very difficult to maintain due to the webs of nonlocal dependencies that evolve. Furthermore, the event loops of many applications have many elements in common.

## A Modular Style Encouraged

The receptionist is an attempt to achieve an abstraction of these common elements. With it, application code takes the form of a sequence of module definitions, followed by registration of the modules with the receptionist and then passing of control to it.

```
define worker 1 filter
define worker 1 handler

define worker 2 filter
define worker 2 handler
...
define worker n filter
define worker n handler

main
  register worker 1 with receptionist
  register worker 2 with receptionist
  ...
  register worker n with receptionist
  pass control to receptionist
```

With this style, the interdependencies amongst the workers can be more easily managed. Interworker communication is decoupled from communication between workers and the event queue. The communication protocol amongst workers should probably be expressed in an object-oriented style. However, any style—communication by global variables, flags, semaphores, etc.—is compatible with the receptionist, which only manages the event queue.

## Workers and Events

A worker is actually a triple

```
(device-number event-filter event-handler)
```

that is abbreviated in the code below to

```
(dn ef eh)
```

*Device-number* is a system-dependent code denoting a physical or logical device. Device number 0 is reserved for *null events*. Handlers for null events are background tasks that execute when the hardware event queue is empty.

*Event-filter* is a Boolean function of one argument returning true when the corresponding *event-handler* should be called. The argument to the filter is the event found by the receptionist. An *event* is a pair

```
(device-number device-data)
```

abbreviated to

```
(dn dd)
```

Filters are used to refine the definition of an event at the application level, for example by checking whether a mouse hit was inside a certain on-screen object such as a button or slider.

The *event-handler* is a function of one argument—an event—that processes a detected, or "filtered", event. The event-handler performs whatever application-dependent actions are associated with the event.

## Operation

When control is passed to it, the receptionist takes an event from the system event queue, checks all registered workers with *device-number* matching that of the event, and calls all handlers whose filters evaluate to true.

## Implementation

The following code implements class *receptionist* under the conventions of the *methods* package documented in Appendix 2. The code here is somewhat simplified from that actually in use. For example, several methods for removing workers from the database are not shown.

### Breaking Out

The first line of code defines a global variable denoting a continuation used to break out of the event-dispatching loop of the receptionist. A *continuation* is a Scheme procedure representing an execution context. It may loosely be thought of as a stack frame or as a place to *go-to* (every real language seems to have some sort of *go-to*, and the continuation is the Scheme version). This *breakout continuation* is initialized to nil and is set to a desired continuation by later code.

```
(set! **standard-breakout-continuation** '())
```

### The Worker Database

The next line begins the definition of the constructor procedure for instances of class *receptionist*. The procedure definition occupies several pages. The first expression in that procedure is a *let** expression establishing the instance variables of a receptionist.

```
(define (new-receptionist . initial-workers)

  (let* ( (maxsize 16)
          (size     1)
          (tbl     (make-vector maxsize))
          (dummy   (vector-set! tbl 0 '(0)))
          (supers '()) )
```

The worker database is organized as a binary search table keyed by device numbers for speedy search. The table is implemented as a Scheme vector of initial length 16. The elements of this vector are lists of the form

```
(device-number (ef1 eh1) (ef2 eh2) ... (efn ehn))
```

that is, a list whose *car* is a device number and whose *cdr* is a (possibly empty) list of filter/handler pairs. Such a list is called an *event service record* and represents all the workers for the device.

The table is further initialized so that the first slot contains an empty list of filter/handler pairs attached to device number zero. Element zero of the table always contains the event service record for the null event, that is, the list (0 (*ef* *eh*) ...). This condition is the *global invariant* of the *receptionist* class.

## Binary Search for Event Service Records

The binary search algorithm is standard; it has three local variables, *min*, *mid*, and *max*, which are table indices representing the state of the search. These variables obey three invariants: $min \geq 0$, $max < size$, $min \leq mid \leq max$. The search key is a device number. Two similar internal search procedures are given. *Index-search* returns the index of the table slot containing the matching event service record given a search key, and *search* returns the event service record in that slot. *Index-search* is used only by internal procedures when workers are being added to the database. Search is available externally to receptionist client code. The search loops of both are implemented with the named *let* construct.

```
(define (index-search key)
  (let iter ( (min 0) (max (-1+ size)) )
    (let* ( (mid  (quotient (+ min max) 2))
            (v    (vector-ref tbl mid))
            (test (car v)) )
      (cond
       ( (= key test)  mid                 )
       ( (>= min max)  '()                 )
       ( (< key test)  (iter min (-1+ mid)) )
       ( else          (iter (1+ mid) max)  )))))

(define (search key)
  (let iter ( (min 0) (max (-1+ size)) )
    (let* ( (mid  (quotient (+ min max) 2))
            (v    (vector-ref tbl mid))
            (test (car v)) )
      (cond
       ( (= key test)  v                   )
       ( (>= min max)  '()                 )
       ( (< key test)  (iter min (-1+ mid)) )
       ( else          (iter (1+ mid) max)  )))))
```

The following internal procedure checks the global invariant:

```
(define (check) ;;; check the invariant
  (if (not (= (car (vector-ref tbl 0)) 0))
          (error
            "Receptionist must always have a null worker")))
```

## The *install* Method

The next three procedures implement the *install* method of the *receptionist* class. This method takes a worker triple as its argument. If there already exists an element in the search table whose search key equals the device-number of the worker, then the filter and handler of the worker are appended to the event service record attached to the key. Otherwise, a new event service record is created from the worker data and is sifted into the table in a position corresponding to the device number, which becomes the search key for the new record. The table is always maintained in order of the search keys, as required by the binary search algorithm.

The *install* method takes care of two more details. If the new record causes a space overflow in the table, new space twice as large as the old space is allocated, and the table data, including the new record, is copied to the new space. Also, before exiting, the method registers the device number with the hardware by calling the *qdevice* GL entry point. This call sets the hardware event mask, allowing the device to generate and enqueue events.

It can be seen that the *install* method can be time-consuming. This is appropriate as it will not be called as frequently as the *search* method. The receptionist algorithms are optimized for fast lookup.

```
(define (swap i j)
  (let ( (t (vector-ref tbl i)) )
    (vector-set! tbl i (vector-ref tbl j))
    (vector-set! tbl j t)))

(define (sift-in worker)
  ;;; precondition -- maxsize > size
  (vector-set! tbl size (list (car worker) (cdr worker)))
  (let iter ( (i (-1+ size)) )
    (cond
     ( (= i 0)  (check) )
     ( (< (car (vector-ref tbl (1+ i)))
          (car (vector-ref tbl i)))
       (swap i (1+ i)) (iter (-1+ i)) )
     ( else (check) )))
  (set! size (1+ size)))

(define (install worker)
  (let ( (lookup (search (car worker))) )
    (cond

     ( lookup
       (vector-set!
        tbl
        (index-search (car worker))
        (cons (car lookup)
              (cons (cdr worker) (cdr lookup)))) )

     ( (= size maxsize)
       (set! maxsize (* maxsize 2))
       (let ( (temp (make-vector maxsize)) )
         (for (i from 0 to size)
              (vector-set! temp i (vector-ref tbl i)))
         (set! tbl temp))
       (sift-in worker) )

     ( else (sift-in worker) )))

  (qdevice (car worker))
  self)
```

### The *print* Method

The next two procedures are used to display the contents of the search table.

```
(define (show sym val)
  (display sym) (display " = ") (display val) (newline))

(define (print)
  (show 'size size)
  (show 'maxsize maxsize)
  (display "#(") (newline)
  (for (i from 0 to size)
       (display " ")
       (display (vector-ref tbl i))
       (newline))
  (display ")")
  (newline))
```

### The Basic Loop

The next two procedures implement the basic event-dispatching loop of the receptionist. *Iter-worker* is a tail-recursive procedure that checks every filter in a list of filter/handler pairs, passing the filter a given event and executing the corresponding handler if the filter evaluates true.

```
(define (iter-worker w data)
  (cond
    ( (null? w) )
    ( ((caar w) data) ((cadar w) data) ;; handler if filter
        (iter-worker (cdr w) data) )
    ( else
        (iter-worker (cdr w) data) ) ))
```

The next procedure, *loop*, is only slightly complicated by the presence of expressions to manage continuations. When *loop* is called, a desired break-out continuation is passed to it. This continuation is "where to go-to" when an event triggering loop termination occurs. This event is defined by a *quit-worker*. A default quit-worker is given below.

After calling *qreset*, a GL entry point that clears the hardware queue, *loop* sets the global variable ***standard-breakout-continuation*** to the given continuation so it can be referred to by a quit worker. *Loop* then enters an infinite *while* loop.

The first action in the infinite loop is to call the GL entry point *qtest*, which non-destructively reads the head of the hardware event queue. *Qtest* returns 0 if the hardware queue is empty; otherwise it returns the device number of the first event in the queue. The return value of *qtest* is passed to the table-searching procedure *search.* The return value of *search* is assigned to the local variable *esr*, which stands for event service record.

There are three possibilities: *esr* is null; *esr* corresponds to device 0, which generates null events; or *esr* corresponds to a real hardware event. If *esr* is null, then no worker for the device in the queue is known by the receptionist, and the event is discarded from the queue by a call of *qread*, the GL entry point that *destructively* reads the head of the queue. If the device number is zero, then all null event filters and handlers are called *via iter-worker* and passed the dummy event (0□0). If these tests fail, then an actual event is read from the queue, and all filter/handler pairs are called with that event *via iter-worker*. It is necessary explicitly to test the device against zero since *qread* will wait on an empty queue.

```
(define (loop breakout-continuation)
  (qreset)
  (set! **standard-breakout-continuation**
        breakout-continuation)
  (while #t
    (let ( (esr (search (qtest))) )
      (cond
        ( (null? esr) (qread) ) ;;; trash the event
        ( (= 0 (car esr)) (iter-worker (cdr esr) '(0 0)) )
        ( else (iter-worker (cdr esr) (qread)) )))))
```

### The *enter-loop* Method

The previous procedures, *iter-worker* and *loop*, are helper routines for the *enter-loop* method.

```
(define (enter-loop)
  (call-with-current-continuation loop))
```

*Call-with-current-continuation* is a standard Scheme procedure that calls its argument, which must be a procedure of one argument, and passes it the current continuation. When the quit worker is activated, control will jump to a point just after this line of code, which, in turn, returns in a normal manner to the client procedure that invoked *enter-loop*.

### *Self*

The following procedure is a message dispatcher, a required bit of *methods* infrastructure, and is the return value of the receptionist constructor procedure:

```
(define (self msg)
  (cond
   ( (eq? msg 'install)      install          )
   ( (eq? msg 'retrieve)     search           )
   ( (eq? msg 'print)        print            )
   ( (eq? msg 'table)        (lambda () tbl)  )
   ( (eq? msg 'enter-loop)   enter-loop       )
   ( (search-supertypes supers msg)           )
   ( else  (make-error-method "Receptionist" msg) )))

(for-each install initial-workers)

self))
```

### Standard Filters and Workers

The following procedures are part of the receptionist package and implement certain frequently used filters:

```
(define (always data) #t)
(define (never data)  #f)
(define (button-down data) (= (cadr data) 1))
(define (button-up   data) (= (cadr data) 0))
```

The following worker defines a standard quit worker that breaks out of the infinite loop whenever the escape key is pressed. Note that the filter of this worker is *always* and the handler calls the escape procedure that the receptionist has previously loaded in the global variable ***standard-breakout-continuation*** as a side-effect of the *enter-loop* method.

```
(define (standard-quitter data)
  (**standard-breakout-continuation** #t))

(define standard-quit-worker
  (list ESCKEY always standard-quitter))
```

The procedure that follows, *do-until-event*, is a hard-looping construct often useful in handlers. For example, one might want to begin an operation when the mouse button goes down and to perform the operation continuously until the mouse button is released. *Do-until-event* takes an event, *data*, and a procedure of no arguments, *thunk*, to call in a hard loop until an event matching *data* is found in the hardware queue. All nonmatching events are discarded. The code is straightforward, and, in the interest of brevity, its analysis is left to the reader.

```
(define (do-until-event data thunk)

  (let ( (dev (car data))
         (val (cadr data)) )

    (qdevice dev)
    (let iter ( (q (qtest)) )
      (cond
        ( (= q 0) (thunk) (iter (qtest)) ) ;; nothing in queue
        ( else (let ( (qr (qread)) )    ;; Read queue contents
                 (cond
                   ( (and (= q dev) (= val (cadr qr))) 'done )
                   ( else (thunk) (iter (qtest)) ))) )))))
```

## Summary of the Receptionist

Though there is a fair amount of code in the *receptionist* class, it is straightforward and easy to read. It is also notable that this high-level architectural device is implemented in a shell language. This is possible only because the language has the exceptionally powerful programming features of Scheme.

The receptionist is used to implement graphical user interfaces, and the code is fast enough in practice as to be indistinguishable from an implementation of an equivalent facility in C, which is many times larger in lines of code. Many examples of use of the receptionist are available from the author.

Note also that to port the receptionist to another system one need merely emulate the four GL entry points *qdevice*, *qreset*, *qtest*, and *qread*.

## DATA FLOW

The next few paragraphs show a simple data-driven data flow executive called *dfgraph* along with a primitive data flow node class and a brief test transcript. This example employs class inheritance. A longer example implementing a quadratic equation solver is available from the author.

## A Little Data Flow Language

The complete data flow package, the implementation of which is too long to be presented here, may be viewed as a little data flow language embedded in Scheme (the term "little language" has been coined by Brian Kernighan.[24] Dfgraph, like *methods*, does not use the GL extensions of GLeme; that is, it is implemented in "vanilla" Scheme.

This language has the following structure. A data flow *graph* is a list of *nodes*. A node is either a *primitive node* or a graph. Thus, a graph may contain nested subgraphs. Every node has zero or more *input arcs* and zero or more *output arcs*. A node operates by consuming tokens from its input arcs, performing an operation on the tokens, and appending tokens to its output arcs.

It is natural to imagine generating expressions in this little language from a graphical editor. Even the definition of the language evokes strong graphical imagery. One can imagine drawing a data flow program on the screen with a mouse and having the code generated and executed by *dfgraph*. Such has not been implemented yet.

Primitive nodes differ from graphs in how their internal operations are implemented. The operation of a primitive node is represented by a procedure of $K$ arguments that returns a list of $L$ output values, where $K$ is the number of input arcs and $L$ is the number of output arcs of the node. The operation of a graph is represented by a number of *internal* nodes and arcs that together perform a composite action. The *external* input arcs and output arcs of a graph are those with only one end connected to an internal node.

## Data Flow Arcs Are *Queues*

Arcs are first-in, first-out (FIFO) queues, instances of the *queue* class presented in Appendix 2. They may be represented graphically as arrows pointing from one node to another. An arc is an output arc of the node at its tail and an input arc of the node at its head. The node at the tail of an arc appends tokens to the tail of the queue, and the node at the head of an arc consumes tokens from the head of the queue. A node is *ready to fire* when all of its input arcs are nonempty (this is data-driven data flow). When a node fires, exactly one token is taken from each input arc and exactly one token is pushed into each output arc.

## Node Methods

Every node has methods for the following messages:
*ready?* — return *#t* if and only if the node is ready to fire;
*fire!* — fire the node, modifying the input and output arcs as a side effect;
*print* — print a list of the front tokens in each input arc;
*clear!* — clear all input and output arcs.

Table 4 contains a list of node classes that have been implemented as part of the development of *dfgraph*.

## Class *Pnode*

The following procedure is the constructor for primitive nodes, or *pnodes*. This class is used as an ancestor class by specific primitive node classes such as *add-node* and *sub-node*.

```
(define (new-pnode proc input-queues output-queues)
```

*Proc* is a procedure of *K* inputs that returns a list of *L* outputs, where *K* is the number of input arcs and *L* is the number of output arcs. *Input-queues* is a list of *K* instances of the queue class representing the input arcs, and *output-queues* is a list of *L* queues representing the output arcs. The variables *proc*, *input-queues*, and *output-queues* are all in the local environment of *new-pnode*, so they function as instance variables.

The next line of code defines the conventional instance variable *supers*, required by *mehotds* and bound initially to nil.

```
  (let* ( (supers '()) ) ;;; a root class; no ancestors
```

### The *ready?* Method

The next two internal procedures implement the *ready?* method of the node by returning *#t* if and only if all input queues are non-empty.

```
    (define (all-qs-non-empty? qlist)
      (cond
       (  (null? qlist)  #t  )
       (  else  (and (not (send (car qlist) 'empty?))
                     (all-qs-non-empty? (cdr qlist)))  )))

    (define (ready-to-fire?)
      (all-qs-non-empty? input-queues))
```

### The *fire!* Method

The next internal procedure implements the *fire!* method by applying *proc* to a list assembled by removing a single item from the front of each input queue by using the *front!* method of the queue class, and by pushing a single token into each output queue using the *append!* method of the queue class. Note how short the code is, the brevity being achieved through Scheme's *map* and *apply* primitives, which take care of all iteration over the input and output queues.

```
(define (fire-node!)
  (let* (  (input-list
             (map (lambda (q) (send q 'front!)) input-queues))
            (output-list (apply proc input-list))  )

    (define (iter output-items output-qs)
      (cond
       ( (null? output-qs)    self )
       ( (null? output-items) self )
       ( else  (let ( (o (car output-items))
                      (q (car output-qs)) )
                 (send q 'append! o)
                 (iter (cdr output-items)
                       (cdr output-qs))) )))

    (iter output-list output-queues)))
```

### The *print* and *clear!* Methods

The following are the *print* and *clear!* methods of a pnode.

```
(define (print)
  (define (front q)
    (if (send q 'empty?) "Empty" (send q 'front)))
  (display
   (map front input-queues))
  (newline)
  self)

(define (clear!)
  (map (lambda (q) (send q 'clear!)) input-queues)
  (map (lambda (q) (send q 'clear!)) output-queues)
  self)
```

### *Self*

Next is the method-dispatching procedure that represents instances of the class *pnode*. As with all *methods* classes, this procedure is returned by the constructor.

```
(define (self msg)
  (cond
   ( (eq? msg 'ready?)  ready-to-fire? )
   ( (eq? msg 'fire!)   fire-node!     )
   ( (eq? msg 'print)   print          )
   ( (eq? msg 'clear!)  clear!         )
   ( (search-supertypes supers msg)    )
   ( else  (make-error-method "Data-flow-node" msg) )))

self))
```

## Class *Dfgraph*

The next procedure constructs a data flow graph given a list of nodes, a list of input arcs, and a list of output arcs. The given arcs must connect at exactly one end to nodes in the given list, though this is not checked by the procedure.

```
(define (new-dfgraph graph input-queues output-queues)
```

The argument *graph* is a list of nodes that may be primitive nodes or other graphs; *input-queues* is a list of *K* queues for input arcs; and *output-queues* is a list of *L* queues for output arcs, just as with the *pnode* class. These arguments become instance variables when the constructor is called.

*Dfgraph* is a root class with no ancestors.

```
(let* ( (supers '()) )
```

### The *ready?* Method

The *ready?* method is identical to that for class *pnode*.

```
(define (all-qs-non-empty? qlist)
  (cond
   ( (null? qlist) #t )
   ( else (and (not (send (car qlist) 'empty?))
               (all-qs-non-empty? (cdr qlist))) )))

(define (ready-to-fire?)
  (all-qs-non-empty? input-queues))
```

### Helpers for *print* and *clear!*

The following *print* and *clear!* method helpers simply recurse down the list *graph*, sending the *print* and *clear!* messages to each element of the list. If the element is a pnode, it actually performs the print or clear operation of the pnode. If the element is a nested subgraph, it recursively passes the messages to elements of the subgraph until all pnodes have been reached.

```
(define (print graph)
  (cond
   ( (null? graph) self )
   ( else (send (car graph) 'print)
          (print (cdr graph)))))

(define (clear! graph)
  (cond
   ( (null? graph) self )
   ( else (send (car graph) 'clear!)
          (clear! (cdr graph)))))
```

### *Self*, *fire!*, *print*, and *clear!*

Unlike the *pnode* class, *self*, returns the *fire!*, *print*, and *clear!* methods as inline *lambda* expressions. The *fire!* method is especially notable since it calls the external procedure *execute-data-flow*, explained next.

```
(define (self msg)
  (cond
   ( (eq? msg 'ready?) ready-to-fire?                          )
   ( (eq? msg 'fire!)  (lambda () (execute-data-flow graph)) )
   ( (eq? msg 'print)  (lambda () (print graph))             )
   ( (eq? msg 'clear!) (lambda () (clear! graph)             )
   ( (search-supertypes supers msg)                          )
   ( else  (make-error-method "DFGraph" msg)               )))

self))
```

## Execute Data Flow

The following procedure is the data flow executive. It takes a graph, which is a list of nodes, as input. It loops over the list until no nodes are ready, sending *fire!* messages to each ready node. The nodes, which are connected by shared arcs, must form an acyclic graph to guarantee termination. This condition is not checked. Note that if any elements of the graph are themselves graphs, their *fire!* methods internally call *execute-data-flow*. This large-scale recursion allows abstracted, reusable node classes that perform complex tasks to be defined.

```
(define (execute-data-flow graph)

  (define (one-pass graph)
    (cond
     ( (null? graph) #f ) ;;; so far, nothing is ready.

     ( (send (car graph) 'ready?)
           (send (car graph) 'fire!)
           (or #t (one-pass (cdr graph))) )

        ;;; at least one node was ready, so OR in #t.

     ( else (one-pass (cdr graph)) )))

  (define (iter) ;;; loop until nothing is ready.
    (cond
     ( (not (one-pass graph)) 'done )
     ( else  (iter) )))

  (iter))
```

## Class *Add-node*

The following pnode example is the *add-node*. An add-node adds the tokens in its two input arcs, producing one output token on its single output arc. Since it is sensible to say "every add-node is a pnode", it is sensible for class *add-node* to inherit attributes and methods from class *pnode*. The "is-a" relation between classes is a criterion for inheritance in the *methods* package.

```
(define (new-add-node left-input right-input output)

    (let* ((supers (list
                     (new-pnode
                      (lambda (a b) (list (+ a b)))
                      (list left-input right-input)
                      (list output)))))

    (define (self msg) ;;; has no local methods --
      (cond            ;;; all messages delegated
       ( (search-supertypes supers msg) )
       ( else  (make-error-method "Add-node" msg)  )))

    self))
```

## Testing Data Flow

The following is a transcript of a test of add-nodes and *execute-data-flow*:

```
(define arc1 (new-queue 10 20 30 40))

(define arc2 (new-queue 5 5 5))

(define o-arc (new-queue))

(define an-add-node (new-add-node arc1 arc2 o-arc))

(define (print-all)
  (send arc1 'print)
  (send arc2 'print)
  (send o-arc 'print))

(print-all)
```

This call of *print-all* results in the following output, showing the contents of the first input arc, the second input queue, and the output queue:

```
(10 20 30 40)
(5 5 5)
()
```

Next, *execute-data-flow* is called on a data flow graph that is a list of one node. This procedure returns the symbol *'done* when no nodes are ready to fire.

```
(execute-data-flow (list an-add-node)) ;returns 'done

(print-all)
```

which prints

```
(40)
()
(15 25 35)
```

One can see from the output of *print-all* that data have flowed out of the input queues and into the output queue. One input token is left behind because it had no positionally corresponding token in the other input queue.

## THE EMBEDDED LINKS LANGUAGE

The LINKS language is to LinkWinds as HyperTalk is to HyperCard or mocklisp is to emacs. It is an embedded, extensible, textual language in which the LinkWinds user can execute commands; define new commands; define variables and complex data structures; do sequencing, iteration, selection, abstraction, recursion — in short, *program.* A programming facility is important because not everything the user wants to do can be easily expressed with operations in a GUI environment. At the very least, the user might want to record a set of mouse clicks and other GUI actions to be edited, copied, renamed, played back, etc.

The user can drive LinkWinds in two ways: (1) graphically, with mouse clicks, keyboard hits, dial twiddles, and actions on other devices; (2) textually, with commands typed into a LINKS listener window. There does not yet exist a complete definition of the LinkWinds GUI, which would include a list of graphical actions and equivalent LINKS commands. Therefore, this section focuses on the basic GLeme infrastructure used for LINKS.

Commands are of two types: LINKS primitives and composite commands. Every LINKS primitive has a direct graphical equivalent. Composite commands are made up of LINKS primitives and other composite commands packaged in function definitions, loops, conditionals, and other programming constructs.

Naturally, composite commands do not have direct graphical equivalents. However, since a composite command always reduces to a sequence of primitive commands when executed, it is possible to animate the GUI, causing it automatically to play back the sequence of primitives generated by any composite command. The reverse is also possible: if a LINKS listener window is open, every graphical action can cause the echoing of the equivalent LINKS primitive command in the interpreter window.

The GUI and LINKS interfaces to LinkWinds are completely equivalent. It is therefore possible to arrange the GUI so that the LINKS listener window and the displayed state of the GUI are kept consistent at all times. This allows the user both to perform actions either by typing expressions in the listener window or by manipulating the GUI display, and to switch back and forth between the two modalities at will.

A special, embeddable version of GLeme serves as the core for LINKS, just as Scheme is the core for GLeme. This special, embeddable version differs from GLeme only in its top-level control strategy. GLeme operates, like all interactive Lisp systems, in an infinite *read-eval-print*, or REP, loop. LINKS, on the other hand, has a REP *hook*, which does one read-eval-print sequence when called by a C program. Thus, LinkWinds is the master program in control of LINKS.

# CONCLUSION

Scheme is an easily customized interactive programming language. Well-debugged implementations of the core language can be acquired in the public domain. Its utility as a base language for graphics shells lies in the ease with which it can be integrated with a graphics interface and in its expressive power for solving problems in interactive computer graphics. This power is outstanding, as shown in several examples of ambitious modules presented in this paper.

A shell with the programming power of Scheme is indispensable for the engineering of complex graphics software and can also play a crucial role in a tightly coupled environment of cooperating software development tools. The next challenge is to specify and build such an environment.

# ACKNOWLEDGEMENTS

This work was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

I am grateful to Robert Tausworthe, Leo Blume, and Kurt Kandt for their comments.

# REFERENCES


**1.** *The PostScript Page Description Language*, Addison-Wesley, Reading, MA, 1987.

**2.** A. Nye, *Xlib Programming Manual*, Vol. 1, The X Window System Series, O'Reilly & Associates, Sebastopol, CA, 1988.

**3.** J.D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Systems Programming Series, Addison-Wesley, Reading, MA, 1982.

**4.** *Inside Macintosh*, Vol. I through V, Addison-Wesley, Reading, MA, 1985.

**5.** W.R. Mallgren, *Formal Specification of Interactive Graphics Programming Languages*, The MIT Press, Cambridge, MA, 1983, An ACM Distinguished Dissertation.

**6.** B.W. Kernighan and R. Pike, *The UNIX ™ Programming Environment*, Prentice-Hall Software Series, Prentice-Hall, Englewood Cliffs, NJ, 1984.

**7.** H. Abelson, G.J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Electrical Engineering and Computer Science Series, MIT Press, Cambridge, MA, 1985.

**8.** R.K. Dybvig, *The Scheme Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

**9.** J. Rees and William Clinger, 'Revised3 Report on the Algorithmic Language Scheme' *ACM SIGPLAN Notices*, **21**, (12)(December 1986).

**10.** J.D. Smith, *An Introduction to Scheme*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

**11.** G. Springer and D.P. Friedman, *Scheme and the Art of Programming*, The MIT Press, McGraw-Hill, 1989.

**12.** F.S. Beckman, *The Mathematical Foundations of Programming*, Addison-Wesley Systems Programming Series, Addison-Wesley, Reading, MA, 1982.

**13.** G. Revesz, *Lambda-Calculus Combinators and Functional Programming*, Vol. 4, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, 1988.

**14.** D. Jefferson and B. Beckman, 'Distributed Simulation and the Time Warp Operating System' *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, ACM, November, 1987.

**15.** F. Wieland, L. Hawley, A. Feinberg, M. DiLoreto, L. Blume, J. Ruffles, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. Jefferson, ‘Distributed Combat Simulation and Time Warp: the Model and its Performance’ *Concurrency Practice and Experience*, **1**, (1)(1989).

**16.** A.S. Jacobson *LinkWinds: the Linked Windows Interactive Data System*. Tech. Rept. Jet Propulsion Laboratory, California Institute of Technology, in preparation, 1990.

**17.** *GT Graphics Library User's Guide, version 1.0.*, Silicon Graphics Inc., Mountain View, CA.

**18.** *4Sight User's Guide, version 2.0.*, Silicon Graphics Inc., Mountain View, CA.

**19.** *Graphics Library Reference Manual, C Edition, version 1.0.*, Silicon Graphics Inc., Mountain View, CA.

**20.** D.M. Betz, *XSCHEME: An Object-oriented Scheme*, P.O. Box 144, Peterborough, NH 03458, January, 1989, the manual and software is available electronically from a variety of public Internet archives.

**21.** E. Raible *private communication*. 1989, NASA Ames, Moffett Field, CA.

**22.** B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

**23.** E.E. Kohlbecker, Jr. *Syntactic Extensions in the Programming Language Lisp*, Ph.D. dissertation, Indiana University, August 1986.

**24.** B.W. Kernighan, *Little Languages*, invited talk given at the Jet Propulsion Laboratory, 1988.

**25.** B. Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988.

**26.** K. Normark *Simulation of Object-oriented Concepts and Mechanisms in Scheme*. Tech. Rept. R 90-01University of Aalborg, Institute for Electronic Systems, Department of Mathematics and Computer Science, Strandvejen 19, DK 9000 Aalborg, Denmark, email to normark@iesd.auc.dk, January, 1990.

# TABLES

**Table 1**
**Strengths and weaknesses of various types of tools**

| | Object layout and editing | Repetitive, recursive, or conditional tasks | Simulation and animation | Displaying from files, processes, etc. | Software design and testing |
|---|---|---|---|---|---|
| Compiled languages | extremely weak | strong | extremely strong | weak | weak |
| Interpreted shells | weak | strong | weak | extremely strong | strong |
| IGE tools | extremely strong | extremely weak | weak | weak | weak |

**Table 3**
**Versions of the *for* iteration macros**

| No. | Scheme version | C equivalent |
|---|---|---|
| 1 | `(for (i from 0 to 10)`<br>`  stmt ...)` | `for (i=0; i < 10; i++)`<br>`{ stmt ... }` |
| 2 | `(for (i from 0 thru 10)`<br>`  stmt ...)` | `for (i=0; i <= 10; i++)`<br>`{ stmt ... }` |
| 3 | `(for (i from 10 down to 0)`<br>`  stmt ...)` | `for (i=10; i > 0; --i)`<br>`{ stmt ... }` |
| 4 | `(for (i from 10 down thru 0)`<br>`  stmt ...)` | `for (i=10; i >= 0; --i)`<br>`{ stmt ... }` |
| 5 | `(for (i from 0 to 10 by 2)`<br>`  stmt ...)` | `for (i=0; i < 10; i += 2)`<br>`{ stmt ... }` |
| 6 | `(for (i from 0 thru 10 by 2)`<br>`  stmt ...)` | `for (i=0; i <= 10; i += 2)`<br>`{ stmt ... }` |
| 7 | `(for (i from 10 down to 0 by 2)`<br>`  stmt ...)` | `for (i=10; i > 0; i -= 2)`<br>`{ stmt ... }` |
| 8 | `(for (i from 10 down thru 0 by 2)`<br>`  stmt ...)` | `for (i=10; i >= 0; i -= 2)`<br>`{ stmt ... }` |
| 9 | `(for ( (id val) ...)`<br>`     ( test ...)`<br>`     ( update ...)`<br>`  stmt ...)` | `for (id = val, ... ;`<br>`     test && ...   ;`<br>`     update, ...   )`<br>`{ stmt ... }` |
| 10 | `(for (var in list-of-values)`<br>`  expressions written in terms`<br>`  of var ...)` | `no equivalent` |

**Table 2**
**Strengths and weaknesses of C and GLeme**

| | Structurally complex code | Computationally demanding code |
|---|---|---|
| C | weak | strong |
| GLeme | strong | weak |

**Table 4**
**Procedures in the data flow executive package**

| presented in this paper: | |
|---|---|
| Procedure | Explanation |
| *new-pnode* | create a primitive node |
| *new-dfgraph* | create a data flow graph |
| *execute-data-flow* | execute a data flow graph until done |
| *new-add-node* | create an adder node |
| in the extended example (available from the author): | |
| Procedure | Explanation |
| *new-mul-node* | create a multiplier |
| *new-div-node* | create a divider |
| *new-negate-node* | create a unary minus operator |
| *new-sub-node* | create a subtracter |
| *new-dup-node* | create a duplicator |
| *new-sqr-node* | create a squaring node |
| *new-const-node* | create a constant producer |
| *new-sqrt-node* | create a square root-taking node |
| *new-disc-node* | create a discriminant calculator |
| *new-dbl-node* | create a doubler |
| *new-quadratic-solver* | create a real root finder for quadratic equations |

**Table 5**
**Documentation of the *queue* class**

| Procedure | Explanation |
|---|---|
| `(send my-q 'append! item)` | puts *item* at the tail of *my-q* , returning *self* (the ! suffix in the name is a convention meaning that the method changes the state of the object). |
| `(send my-q 'empty?)` | Boolean: #t iff *my-q* is empty |
| `(send my-q 'remove!)` | removes head of *my-q* , returning *self* |
| `(send my-q 'clear!)` | clears *my-q* of all elements, returning *self* |
| `(send my-q 'front)` | returns head of *my-q* |
| `(send my-q 'front!)` | removes head of *my-q* and returns it |
| `(send my-q 'print)` | prints the contents of *my-q* |
| `(send my-q 'list)` | returns a list of the contents of *my-q* |
| `(send my-q 'append-list! Ls)` | appends all elements of *Ls* to the tail *my-q* , returning *self* |

# FIGURES

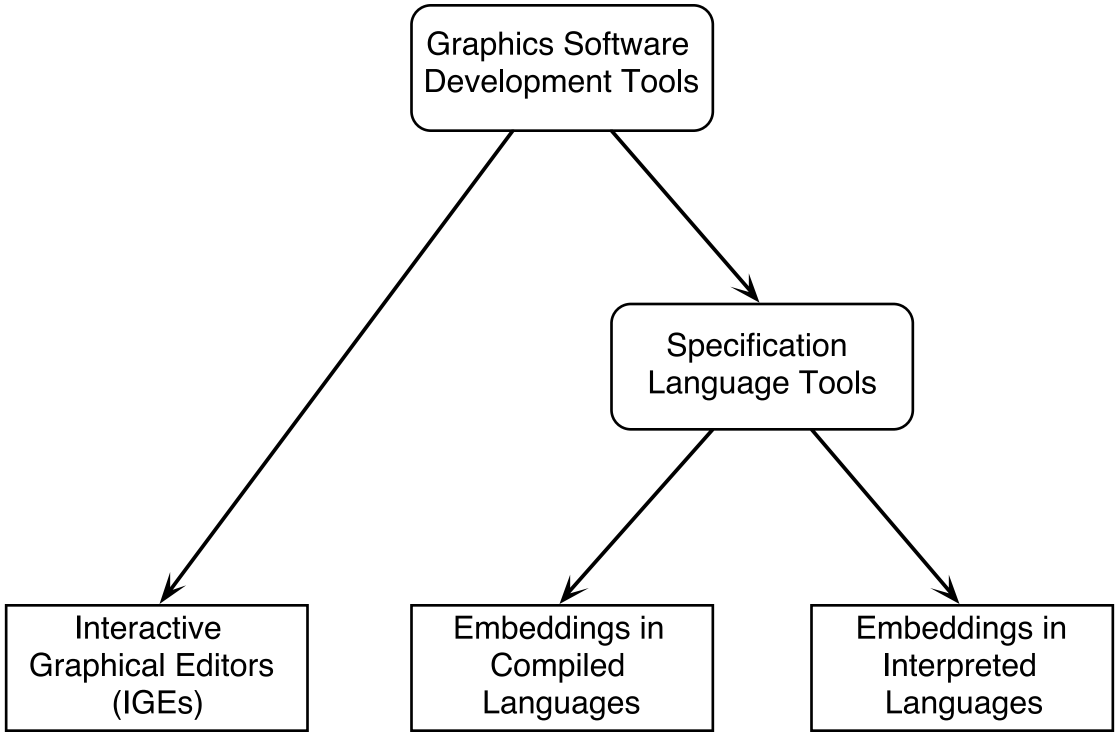


**Figure 1**
***Kinds of graphics software development tools***

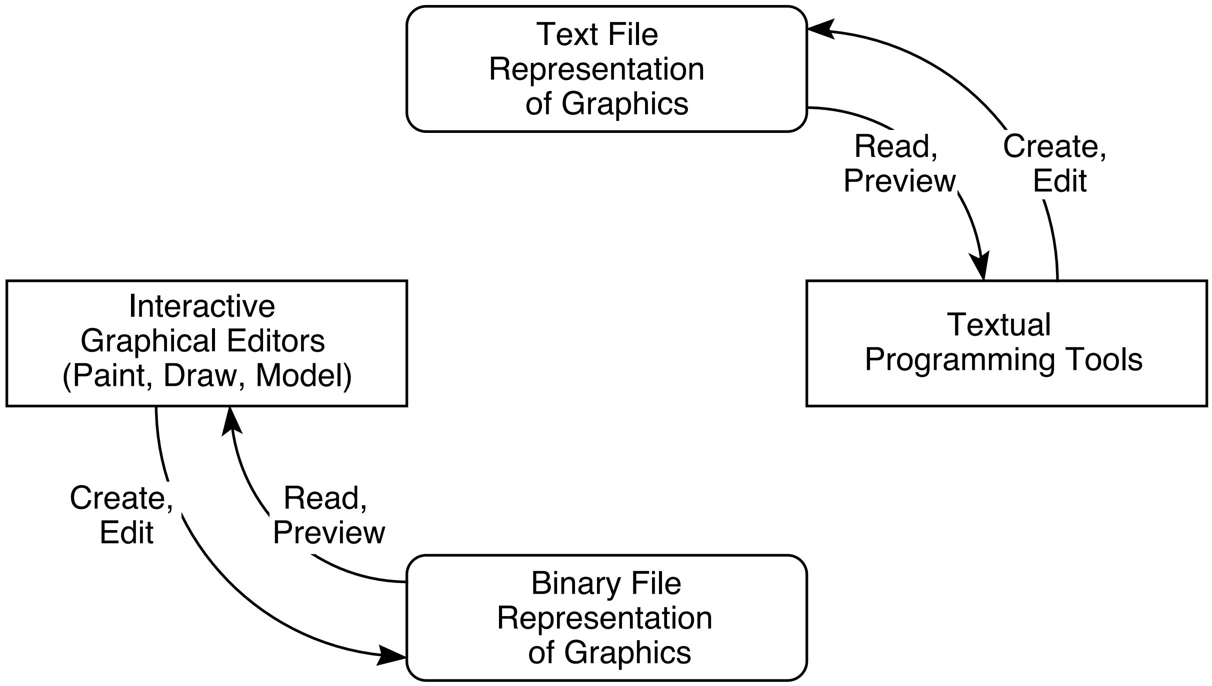


**Figure 2**
***Typical graphics tool suite***

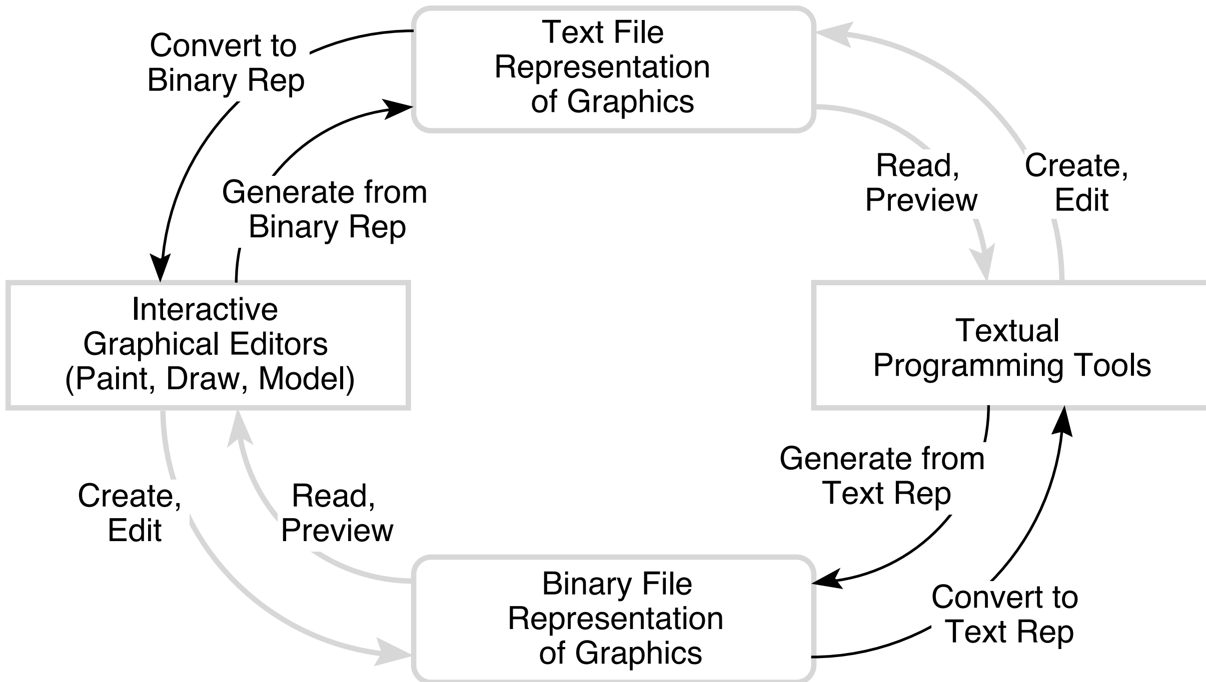


**Figure 3**
***Complete graphics tool suite***

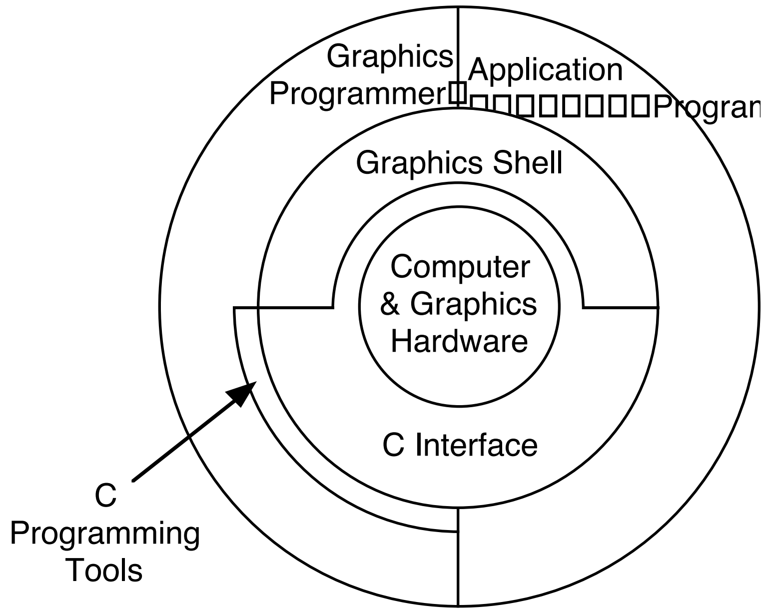


**Figure 4**
***Concept of a graphics shell***

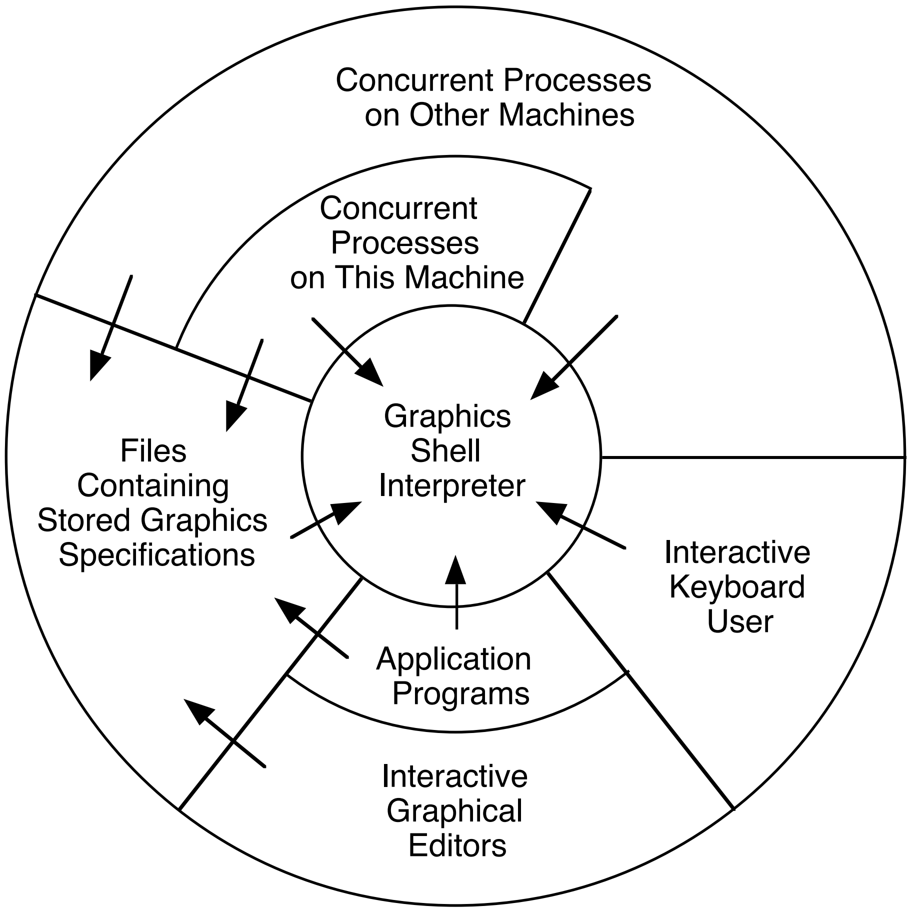


**Figure 5**
***Central role of a shell***

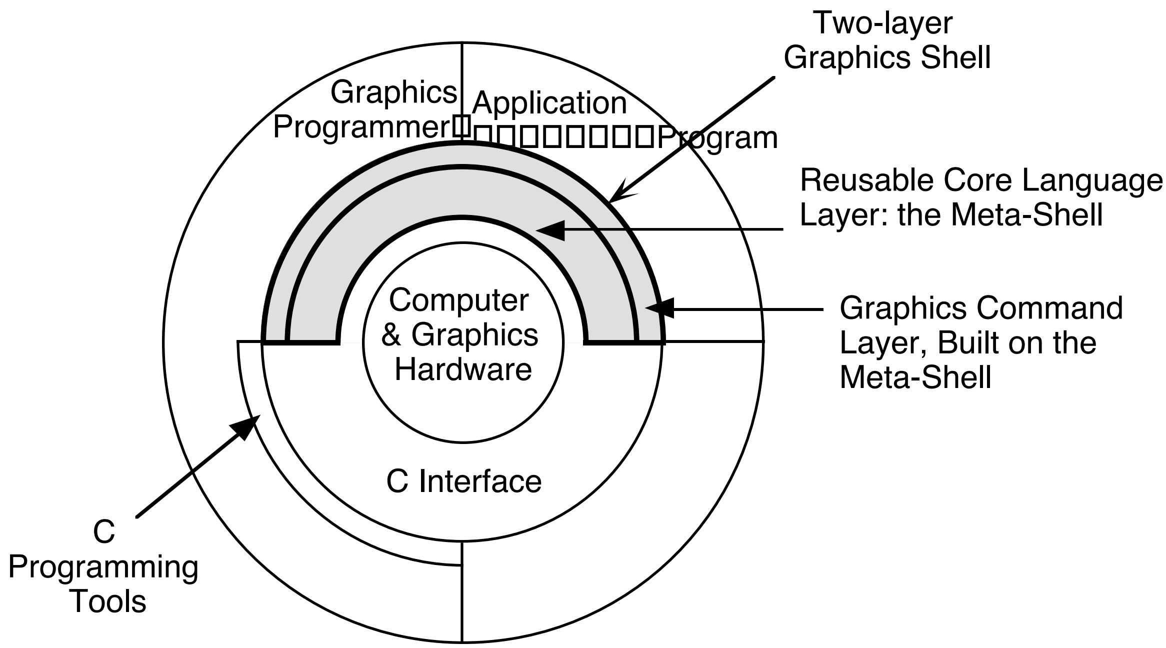


**Figure 6**
***The meta-shell concept***

# APPENDIX 1: *FOR* LOOP MACRO IMPLEMENTATION

```
(extend-syntax (for from to thru down by)

  ( (for ( (id val) ...)
         ( test ...)
         ( update ...)
         stmt ...)
    (let iter ((id val) ...)
      (cond
        ( (and #t test ...) stmt ... (iter update ...) )
        ( else #t ))) )

  ( (for ( id from n to m )
         stmt ...)
    (let iter ((id n))
      (cond
       ( (>= id m) #t )
       ( else stmt ... (iter (1+ id))))) )

  ( (for ( id from n thru m )
         stmt ...)
    (let iter ((id n))
      (cond
       ( (> id m) #t )
       ( else stmt ... (iter (1+ id))))) )

  ( (for ( id from n down to m )
         stmt ...)
    (let iter ((id n))
      (cond
       ( (<= id m) #t )
       ( else stmt ... (iter (-1+ id))))) )

  ( (for ( id from n down thru m )
         stmt ...)
    (let iter ((id n))
      (cond
       ( (< id m) #t )
       ( else stmt ... (iter (-1+ id))))) )

  ( (for ( id from n to m by k )
         stmt ...)
    (let iter ((id n))
      (cond
       ( (>= id m) #t )
       ( else stmt ... (iter (+ id k))))) )

  ( (for ( id from n thru m by k )
         stmt ...)
```

```
  (let iter ((id n))
    (cond
     ( (> id m) #t )
     ( else stmt ... (iter (+ id k))))) )

( (for ( id from n down to m by k )
       stmt ...)
  (let iter ((id n))
    (cond
     ( (<= id m) #t )
     ( else stmt ... (iter (- id k))))) )

( (for ( id from n down thru m by k )
       stmt ...)
  (let iter ((id n))
    (cond
     ( (< id m) #t )
     ( else stmt ... (iter (- id k))))) )

( (for ( var in lyst ) stmt ...)
  (for-each (lambda (var) stmt ...) lyst) ) )
```

# APPENDIX 2: THE *METHODS* OOP SHELL

## METHODS OOP SHELL

*Methods* is a very straightforward object-oriented programming simulator with multiple inheritance and error handling (see Meyer[25] for the basic terminology of object-oriented programming). It is modeled after the message passing technique in chapter 3 of Abelson and Sussman.[7] It is in “pure” Scheme, without using GL capabilities. It is similar to the simulations presented in Normark’s report[26], but was independently developed.

Like Abelson and Sussman’s examples, *methods* requires the programmer manually to follow a certain discipline. Enforcing the programming discipline would require syntax extensions, an approach deemed undesirable due to its inflexibility to change. The programming discipline uses only basic Scheme, but requires the programmer to follow a pattern in defining and using classes.

## Overview

Every class, or object type, in *methods* is represented by its constructor procedure. When called, a constructor returns an instance of the class represented by a message-dispatching procedure. The message-dispatching procedure is usually named *self* so that an object can conveniently send messages to itself (*self* is, however, an internal name not known outside the constructor). *Self* returns *method* procedures that client code applies to actual arguments. Thus, a class is a third-order procedure, returning a procedure that returns procedures. *Self* cooperates in multiple inheritance by walking the inheritance hierarchy upwards until it finds a class that can understand a message. If no class understanding the message is found, an error method is returned.

The present version of methods does not support code sharing, so every instance of a class has its own private copies of the method code. This limitation was retained because it simplifies the programming discipline, and its impact on performance was not troubling.

## Error Processing

There are two modes of error processing. In *normal mode*, an error is reported only by the first receiver of a message. In *debug mode*, an inheritance traceback is given whereby every class in an inheritance hierarchy reports when it fails to recognize a message. The following global variable represents that mode:

```
(define **method-mode** 'normal-method-mode)
```

The user can set these modes as follows:

```
(define (set-debug-method-mode)
  (set! **method-mode** 'debug-method-mode))

(define (set-normal-method-mode)
  (set! **method-mode** 'normal-method-mode))

(define (reset-debug-method-mode)  ;;; synonym for set-normal
  (set! **method-mode** 'normal-method-mode))
```

and test them with the following procedure:

```
(define (test-debug-method-mode)
  (eq? **method-mode** 'debug-method-mode))
```

Some helper procedures are needed. When an object's class cannot recognize a message, and none of its ancestor classes can recognize it, the object returns an error method. The following global variables are bound to the error method:

```
(define **method-error-class-name** "No class name.")

(define **method-error-message** 'no-message)

(define (error-method . junk-args)
  (display **method-error-class-name**)
  (display ": unknown message: '")
  (display **method-error-message**)
  (newline)
  '())

(define (make-error-method class-name msg)
  (set! **method-error-class-name** class-name)
  (set! **method-error-message** msg)
  error-method)
```

## Multiple Inheritance

Inheritance is implemented by a recursive procedure, *search-supertypes*. When an object is instantiated, local instances of all its supertypes must also be instantiated as part of the programming discipline. Also as part of the discipline, *self*, the message-dispatching procedure, must call *search-supertypes*, passing it the list of local supertype instances as a last resort when a message is not locally recognized.

*Search-supertypes* must cooperate in error handling in a mode-dependent way.

```
(define (search-supertypes supers msg)
  (define method '())
  (if (test-debug-method-mode)
      (begin
       (display "Searching...")
       (newline)))
  (cond
   ( (null? supers) '() )
   ( (begin
      (set! method ((car supers) msg))
      (eq? method error-method))
        (if (test-debug-method-mode)
            (error-method))
        (search-supertypes (cdr supers) msg) )
   ( else method )))
```

*Search-supertypes* works as follows: a local variable, *method*, is defined. Its value is not important to begin with. If debugging is on, a line is printed telling the user that the inheritance hierarchy is being searched. Then, the list of supertypes, *supers*, is examined. If the list is empty (*null?*), nil is returned, which means that no ancestor types understand the message. The nil return value signals the calling procedure to return *error-method*. If *supers* is not empty, the message is passed to the first supertype in *supers*, i.e. to (*car supers*). The method returned is assigned to the local variable *method*.

If this method is the one and only global *error-method*, then the supertype, and, recursively, all its supertypes, did not recognize the message. If debugging is on, the returned *error-method* is executed, contributing to the inheritance traceback being printed. Finally, the value of a recursive call of *search-supertypes* on the rest of the list of supertypes, (*cdr supers*), is returned.

If the method returned by the first supertype is *not* the one and only global *error-method*, then the supertype *did* understand the message after all somewhere in the hierarchy, and *search-supertypes* returns the method.

In summary, *search-supertypes* passes the message to the ancestor classes, in pre-order, returning the first non-error method found.

## Overriding Methods

*Search-supertypes* automatically supports multiple inheritance due to its recursive structure. The list of supertypes is searched in order from front to back. The first match of a message results in the successful finding of a method and the termination of *search-supertypes*. Thus, the order of supertypes in the list is significant when more than one supertype can understand a given message. The earlier members of the list shadow the later ones, preventing their methods from being found. In some object-oriented programming systems, one refers to the “overriding” of methods. The shadowing in *methods* is its form of method overriding, and it is under explicit control of the programmer who sets the order in every list of supertypes.

## Invoking Methods

There are two ways to invoke a method in an object. The first way is simply to pass an object a message, getting back a method. The method can either be called in-line or be cached in a variable and invoked later. Such caching has performance advantages if the method is to be called in a loop. Invoking the method in-line, which is perhaps the more convenient construction, results in expressions like the following:

```
((object 'message) arg1 arg2)
```

where the inner parentheses denote immediate invocation of the method procedure returned by the expression (*object 'message*), which is, in turn, an invocation of the message-dispatching procedure.

This is fairly readable and a fine idiom, but it has its limitations. Suppose that ((*object 'message*) *arg1 arg2*) evaluates to another object, to which one would like to send the message *'m2* with the arguments *arg3* and *arg4*. Such a situation, expanded in-line, results in this:

```
((((object 'message) arg1 arg2) 'm2) arg3 arg4)
```

The number of leading parentheses is syntactically ugly. The following message *sending* procedure eliminates the need for leading parentheses:

```
(define (send object msg . args)
  (apply (object msg) args))
```

The earlier examples of message-passing expressions can now be much more cleanly written, as follows:

```
(send object 'message arg1 arg2)
(send (send object 'message arg1 arg2) 'm2 arg3 arg4)
```

## Class *Queue*

The following code is an extended example of a class implemented in *methods*. It implements the queue abstract data type (ADT), adapted from Abelson and Sussman, pg 208 ff. This data type is used to implement arcs in the data flow executive presented above. It is somewhat shortened from the version actually in use, for the sake of brevity. Detailed explanation follows the code, which is shown as an uninterrupted block so the reader may see the overall shape of a program in the *methods* discipline. This block of code is the constructor procedure, *new-queue*, for objects of type *queue*.

```
(define (new-queue . initial-list)

  (let (  (q (cons '() '())) (supers '())  )

    (define (head) (car q))
    (define (tail) (cdr q))
    (define (set-head! item) (set-car! q item))
    (define (set-tail! item) (set-cdr! q item))
    (define (empty-queue?) (null? (head)))

    (define (front)  ;;; Return front element
      (if (empty-queue?)
          (error "FRONT called on empty queue")
          (car (head))))

    (define (front!)  ;;; Remove front element and return it.
      (let ((f (front)))
        (remove-queue!) f))

    (define (append-queue! item)
      (let ((elt (cons item '())))  ; could be (list item)
        (cond
         ( (empty-queue?) (set-head! elt) (set-tail! elt) )
         ( else     (set-cdr! (tail) elt) (set-tail! elt) ))
       self))

    (define (append-list! lyst)
      (for-each append-queue! lyst) self)

    (define (remove-queue!)
      (cond
       ( (empty-queue?)
         (error "REMOVE called on empty queue")  )
       ( else
         (set-head! (cdr (head)))  self)))

    (define (clear-queue!)
      (set! q (cons '() '())) self)

    (define (print) (display (head)) (newline))

    (define (self msg)
      (cond
       ( (eq? msg 'append!)      append-queue!      )
       ( (eq? msg 'empty?)       empty-queue?       )
       ( (eq? msg 'remove!)      remove-queue!      )
       ( (eq? msg 'clear!)       clear-queue!       )
       ( (eq? msg 'front)        front              )
       ( (eq? msg 'front!)       front!             )
       ( (eq? msg 'print)        print              )
       ( (eq? msg 'list)         (lambda () (head)) )
       ( (eq? msg 'append-list!) append-list!       )
       ( (search-supertypes supers msg)             )   ;; line c1
       ( else (make-error-method "Queue" msg)       ))) ;; line c2

    (append-list! initial-list)  ;;; returns ``self'' ))
```

To create a queue instance containing some initial elements, say 1, 2, and 3, write

```
(define my-queue (new-queue 1 2 3))
```

The number of initial elements is arbitrary.

The body of the constructor is a *let* expression defining a lexical environment. Every queue has a private instance variable, *q*, bound to a pair of pointers to the head and tail of a list of the queue elements. It also has an instance variable, *initial-list*, bound to the list of initial queue elements passed as arguments to *new-queue*. The final instance variable is the conventional one, *supers*, naming the inheritance hierarchy. This implementation of queue has no ancestor classes, so *supers* is nil.

Inside the *let* block is a collection of method and helper procedures. The reader is referred to Abelson and Sussman for detailed explanations of these procedures. Of primary interest here is the illustration of the *methods* programming discipline.

The next to last item in the *let* block is the message-dispatching procedure, *self*. This procedure doubles as reference list-style documentation of the messages and methods of a queue. The calling syntax of these methods is shown in Table 5.

Many of a queue's methods return *self* so that *send* expressions may be conveniently nested, as in

```
(send (send my-q 'append item) 'print)
```

The last two lines of *self*, labeled *c1* and *c2*, are the conventional lines that implement inheritance and error checking. Similar lines must appear in every class using to the *methods* discipline.