SCHEME AS AN INTERACTIVE GRAPHICS PROGRAMMING ENVIRONMENT

by

Peehong Chen
Department of Electrical Engineering & Computer Science
University of California
Berkeley, CA 94720

and

L. David Sabbagh
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 166

SCHEME AS AN INTERACTIVE GRAPHICS PROGRAMMING ENVIRONMENT

by

Peehong Chen, University of California
and
L. David Sabbagh, Indiana University

March, 1985

Scheme as an Interactive Graphics Programming Environment

Peehong Chen

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

L. David Sabbagh

Computer Science Department
Indiana University
Bloomington, IN 47405

Abstract

This paper describes several interesting aspects of Scheme Graphics (SG), an interactive graphics system built on top of Scheme. Most other interactive graphics environments do not offer much programming power by themselves. What is special about SG is that a very expressive language interpreter is integrated into the system making it a highly interactive environment for graphics programming. Scheme is an applicative order, lexically-scoped dialect of Lisp. In Scheme, functions and continuations are first-class citizens. With such, it is relatively easy to simulate modern programming concepts like abstract data types and message passing. We pay special attention to the implications of these language features in a graphics domain. In particular, we discuss issues such as data representation, functional programming, object-oriented programming, and exception handling, all in the context of SG.

1 Instruction

Scheme Graphics (SG), as the name implies, is an interactive graphics programming environment based on Scheme. Scheme is a programming language that runs with a compiler as well as an interpreter. To create an interactive environment for graphics applications, we built a subset of the ACM Core system [GSC79], a set of linear algebra operators, and some miscellaneous functions in Scheme. These graphics-related primitives and Scheme itself are collectively called SG [Chen83a]. Application programs of SG can also access a rich repertoire of Scheme, Franz Lisp, or even C functions in the environment. [1]

Most other interactive graphics environments do not offer much programming power by themselves. They are "interactive" at the *command* level instead of exposing a complete *language* to the world, thus the interactive programming capability in those systems is somewhat restricted. If the user wants to do something beyond what the command set can handle, he has to leave the environment temporarily, prepare those things in a programming language, compile them, and then return to where he left. The user does not lose anything for programming *per se*, but the low degree of interactiveness may be a nuisance. In Unigrafix [Séquin83], a Unix-based interactive graphics environment, for instance, some shell level commands are available for its users. Although the command interpreter of a Unix shell supports some naive programmability, the user still needs help from the C compiler outside the graphics environment to manipulate complex operations.

What is special about SG is that the power of a very expressive language is integrated into the system making it a highly interactive environment for graphics programming. Scheme is a dialect of Lisp. Its major departure from Lisp includes lexical scoping and treating functions and continuations as first-class objects. In this paper, rather than going into the implementation details of SG (concerned readers may refer to [Chen83b]), we try to describe the implications of these language features which make SG "expressive" in the context of computer graphics. We present mostly Scheme code in this paper and hope the reader will not be disturbed by parentheses. It is assumed that Pascal (or any language in that family) counterparts of some examples, although not presented, are understood implicitly.

Section 2 gives a brief overview of the programming language Scheme, introducing primarily those features that are of interest to SG programming. In the section following we discuss how graphics objects are represented in SG. The next two sections examine object manipulations: Section 4 shows a functional style programming which SG inherits directly from Scheme's applicative order evaluation; Section 5 focuses on SG's prototyping capability whereby graphics applications in an object-oriented style similar that of Smalltalk [Goldberg83]

[1] The Scheme used in our work has an *import* facility that allows the user to call any functions written in Franz Lisp or C within the Scheme environment.

```
<expression> ::= <constant>
               | <identifier>
               | (if <expression> <expression> <expression>)
               | (lambda (<identifiers>) <expression>) ; Lambda expression
               | (change! <identifier> <expression>)   ; Assignment
               | <application>                         ; Invocation
               | <syntactic-extension>
<identifiers> ::= <empty> | <identifier> <identifiers>
<application> ::= (<expressions>)
<expressions> ::= <empty> | <expression> <expressions>
<syntactic-extension> ::= (<keyword> <objects>)
<objects> ::= .....      ; Definition is omitted here
.....                    ; More omitted
```

Figure 1: A simplified Scheme syntax

can be modeled. Finally in Section 6 we demonstrate the power of continuations for exception handling which often is desirable in an interactive system.

2 Overview of Scheme

The programming language Scheme was designed and first implemented at MIT in 1975 by Gerald J. Sussman and Guy L. Steele, Jr. as part of an effort to understand the actor model of computation. It is based on the lambda calculus described by Alonzo Church [Church41] and serves as "a simple concrete experimental domain for certain issues of programming semantics and style." The revised report on Scheme was published by Steele and Sussman in 1978 [Steele78]. In 1980 they had their first Scheme VLSI chip implemented and at the same time a full report on their work was also released [Steele80]. SG is implemented in Scheme-311 [Fessenden83] which is written in Franz Lisp [Foderaro80] running under Berkeley Unix on VAX 11/780.

Scheme may be characterized as an applicative order, block structured, lexically scoped, tail recursive dialect of Lisp. In its semantic structure it is as closely akin to Algol 60 as to early Lisps. Its syntactic structure is simple, as shown in Figure 1.

There are quite a few system-provided primitives, functions, and macros. [2] The standard Lisp list access primitives **car** and **cdr** return the first element and a list excluding the first element, respectively. The macro **list** takes any number of arguments and returns a list containing them. The function **append**

[2]The difference between functions and macros is that the expression bound to a function name is evaluated when defined and that to a macro is only a symbolic (or syntactic) binding (not evaluated). When executed a macro needs to *expand* itself.

takes two lists and returns a list with the two appended. The macro `define` does the binding between an identifier and an expression which can potentially be recursive.

Scheme is block-structured. In the simplest case, the system has a facility called `block` (or `progn`) that evaluates a sequence of expressions in the given order and returns the result of the last. Furthermore, the macro `let` plays the role of structuring programs, whose syntax is:

```
(let ([id1      exp1]
      [id2      exp2]
      .....        ; More local bindings
      )
      body)
```

The semantic is that `let` creates a block (i.e. `body` in our case) with variable `id1` bound to `exp1`, `id2` to `exp2`, ... *etc.* Since Scheme is lexically-scoped, the variables are local to `body`. There is a variant of `let` called `letrec` with an identical syntax. The difference is that if an `id` is to be bound to a lambda expression, `letrec` allows it to be recursive whereas `let` does not.

The effect of normal order evaluation can be obtained when necessary by using *thunks*, functions of no arguments. Thus

```
(define th (lambda () exp))
```

binds `th` to `exp` unevaluated (i.e. *freezing*). Evaluation (or *thawing*) of `exp` may be enforced by invoking `th`, as in

```
(th) ; The invocation needs no arguments for a thunk.
```

Scheme treats *functions* (or *lambda expressions*) as first-class citizens. In other words, functions in Scheme are just ordinary Lisp objects (e.g. atoms, lists) in the sense that they can be passed to functions as arguments or they can be returned as values of functions. The difference is that they can be applied with arguments whereas non-functional objects can not. This property does not exist in many Lisp systems. For example, a function invocation that still returns a function is illegal in Franz Lisp but is common in Scheme. [3] In fact, it is this fundamental departure from conventional Lisp that makes modern programming language concepts such as *data abstraction*, *message passing*, *exception handling*, *etc.* possible in Scheme. What they mean to a graphics environment will be discussed later.

An interesting observation is that these important concepts are not built-in features of Scheme. The language kernel is compact and efficient. To prototype

[3] In the thunk example above, exp being a lambda expression is not allowed in Franz but is a basic technique to model data abstraction in Scheme (see Section 5).

other systems, the user simply makes the concepts concreted on a by-need basis. SG is a typical example that takes advantage of this powerful incrementality.

3 Data Representation

SG represents graphics objects in a data structure that is most natural to any Lisp systems — *lists*. The basic data types in a graphics environment are *vectors* and *matrices*. Representing vectors as lists is straightforward. For instance, suppose `vector` is aliased to the Lisp `list` operation, then in SG,

```
(define v (vector 1 2 3 4))
```

binds the identifier v to the vector `[1 2 3 4]`. Furthermore, notice that a matrix is just a list of its component row vectors. Assuming `matrix` to be another alias for `list`, to declare a 2×2 matrix $m = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ will be:

```
(define m (matrix (vector 1 2) (vector 3 4))).
```

The same technique can be applied to represent less trivial geometric objects. Again, let `vertex`, `face`, and `object` all be aliases for `list`. Let uc be a unit cube sitting on the first octant of a left-handed 3D space with the origin being one of its corners. A simple way to represent uc is to declare it as an object with six faces each of which is a list of four vertices specified in a predefined fashion (e.g. clockwise). The following definition specifies the front face of uc:

```
(define fFRONT
   (face (vertex 0 0 0) (vertex 0 0 1) (vertex 1 0 1) (vertex 1 0 0)))
```

The other five faces (i.e. f_{BACK}, f_{TOP}, f_{BOTTOM}, f_{LEFT}, and f_{RIGHT}) can be specified in the same manner. So uc itself is just an object consisting of all these faces, that is,

```
(define uc (object fFRONT fBACK fTOP fBOTTOM fLEFT fRIGHT))
```

For viewing transformations, it is often necessary to extend 3D vectors with a homogeneous coordinate. In SG, it is easy to define the function `homogenize` that does the job:

```
(define homogenize (v)
   (append v (vector 1)))
```

Later when the homogeneous coordinate is no longer necessary, we can just strip off the trailing 1. No vector of any fixed size ever needs to be predeclared. Obviously this flexibility comes from a dynamic feature of Lisp family languages — *weak typing*. Imagine what the corresponding effort will be to represent graphics objects and to do transformations in a stongly-typed language environment like Pascal!

4 Functional Programming

SG programming is mostly done in a functional style. Functional programming makes use of the properties of mathematical functions. A mathematical function is a relation that maps a set (the *domain*) to a set (the *range*) in a way that for each member in the domain, there is a unique image in the range. Thus a function can be defined by specifying the following: (1) the domain, (2) the range, and (3) the rules of mappings between the two. To *invoke* a function, we apply specific elements in the domain to yield a value in the range. For instance, the lambda expression [Church41]

$\lambda x.\lambda y.xy$

defines a curried two-argument function of multiplication such that the application (i.e. invocation)

$((\lambda x.\lambda y.xy)2)3$

yields the result 6.

Due to the fact that languages are implemented in computers which have certain physical constraints, not all properties of mathematical functions can be realized in functional languages. For example, parameters are names of memory cells where their values are stored in programming languages instead of just the representations of values like in mathematical functions. However, since geometry and algebra are the core of computer graphics, it seems reasonable to claim that a functional approach in those categories is a more natural reflection of their properties than using most imperative style programming languages.

Scheme is not pure functional, at least not as pure as McCarthy's Lisp [McCarthy60] or Backus' FP [Backus78]. Pure functional languages should be side effect free. Scheme, like most modern Lisp systems, has `change!` (i.e. `setq`) that does nothing but side effects. As we saw earlier, `let` (or `letrec`) also causes some local side effects to its body. These non-functional features are added to the language in order to achieve more flexibility and cleaner structures. Aside from that, most programming styles are still functional in modern Lisp systems, including SG.

Figure 2 depicts a version of 3D cross product which typifies the functional style programming in SG. It takes two vectors v1 and v2, returns v1×v2. In `cross` a local routine `det` is defined in the `let` clause to calculate determinants. The function `comp` takes a vector v and an index k and returns the kth element of v. The function `1+` is a unit incrementor and the function `mod` is the modulus operator.

Suppose v1 is bound to $[a_1\ \ b_1\ \ c_1]$ and v2 is bound to $[a_2\ \ b_2\ \ c_2]$. `(det 1)`

```
(define cross
   (lambda (v1 v2)
      (let ([det (lambda (k)
                    (- (* (comp v1 (1+ (mod      k 3)))
                          (comp v2 (1+ (mod (1+ k) 3))))
                       (* (comp v1 (1+ (mod (1+ k) 3)))
                          (comp v2 (1+ (mod      k 3)))))))])
         (if (and v1 v2)
             (let ([d1 (length v1)]  [d2 (length v2)])
                  (if (= d1 d2)
                      (if (= d1 3)
                          (vector (det 1) (det 2) (det 3))
                          (writeln '|Arguments not 3D|))
                      (writeln '|Arguments unequal in dimension|)))
             (writeln '|Arguments contain zero vector|)))))
```

Figure 2: Definition of 3D cross product in SG

will return the determinant of $\begin{pmatrix} b_1 & c_1 \\ b_2 & c_2 \end{pmatrix}$. Similarly **(det 2)** will return the determinant of $\begin{pmatrix} c_1 & a_1 \\ c_2 & a_2 \end{pmatrix}$, and **(det 3)**, that of $\begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix}$. If something is wrong with the arguments, an error message will be printed by **writeln** which returns the empty list **nil**.

The returned value is essential to functional programming. Everything other than **nil** (which logically means *false*) is regarded as *true* in Scheme. Continuing the previous example, suppose we want to do some transformation to the cross product **w** (if it exists). In SG it is conceivable to have the following piece of code:

```
(let ([w     (cross v1 v2)])
     (if w
         (transform w .....)      ; If w is non-nil, transform
         .....)                   ; If w is nil, do something else
     )
.....                             ; Rest of the code
```

Recall that **cross** returns **nil** if **v1** and **v2** cannot be crossed.

The language C has a similar flat view on numerical and logical values. Scheme is more elaborate, non-nil values include all symbols: numerical constants, strings, lists, *etc*. In a less functional language like Pascal, the code to handle our example will be comparatively awkward.

```
(define VECTOR
   (lambda ()
      (let ([v (vector)]
            [init-lock  nil])
           (letrec
              ([self (lambda (msg)
                        (case msg
                           [init  (lambda (u)
                                     (if init-lock
                                         self
                                         (block
                                            (change! v u)
                                            (change! init-lock t)
                                            self)))]
                           [cont  v]
                           [hom   (((VECTOR) 'init) (homogenize v)))]
                           [cross (lambda (U)
                                     (((VECTOR) 'init) (cross v (U 'cont)))]
                           [draw  (block
                                     (line_rel_3 (car v) (cadr v) (caddr v))
                                     self)]
                         )))])
              self))))
```

Figure 3: A simplified version of the vector object.

5 Object-Oriented Programming

An even more advanced program structure, namely the object-oriented system, can be modeled in SG. Two programming language concepts are closely coupled with such a structure. The first concept is *data abstraction* which hides information from unauthorized access [Morris73]. The second concept, *message passing*, serves as the mechanism to access the abstracted data. In essence, we are talking about a data entity that (1) encapsulates information and (2) defines the related data manipulation routines marked with entry names. The outside world cannot have access to the information except through the designated entries.

The structure of Smalltalk *objects*, *a la* Simula-67 [Dahl70] *classes*, [4] is a good application of this data abstraction idea. In fact, in Smalltalk there is nothing but objects. Communication between objects is done via message passing. Thus to draw the vector [1 2] is to send a message (asking for drawing)

[4]Simula *classes* also influenced a number of programming languages designed in the 70's. For instance, *modules* in Mesa [Mitchell79] and Modula-2 [Wirth83], and *packages* in Ada [ANSI83] are all descendents of *classes*.

to an object of type Vector currently encapsulating the information [1 2] (in Smalltalk this object is an *instance* of the *class* Vector.)

SG is able to prototype an object-oriented system similar to Smalltalk without much difficulty. To implement objects requires the use of thunks discussed earlier. Figure 3 defines an encapsulated version of the vector data type called VECTOR. [5] It freezes the evaluation of a `let` clause which first reserves a private memory `v` (the invocation of `vector` with no arguments gives one cons cell: `nil`) and returns a lambda expression (i.e. `self`). This `self` is defined by `letrec` because it is recursive. The body of `self` is a `case` expression that evaluates a string (the message) and takes the appropriate action. In other words, VECTOR is a class (i.e. data type) that reserves for its instances some *instance variables* (i.e. private memory) and defines an interface with the rest of the system. This interface consists of some message entries and corresponding *methods* that describe how to carry out the requested actions on the encapsulated information.

Let V be an instance of VECTOR (an instance is created by thawing VECTOR), that is,

```
(define V (VECTOR)).
```

Then `((`V` 'init) `u`)` initializes the vector once and for all. No one will ever be able to modify the encapsulated vector after the first initialization (due to the guard `init-lock`). Peeking V's content can be done by saying `(`V` 'content)`. Algebraic and geometric operations involving V itself and possibly others are done in a similar fashion. For instance, `(`V` 'hom)` gives a homogenized version of V, and `((`V` 'cross) `U`)` returns the cross product of V and U, each an instance of VECTOR. To draw the vector (relative to the current cursor position), `(`V` 'draw)` plots a line by calling the standard SG 3D primitive `line_rel_3`.

Apparently Figure 3 is just a simplified version of the actual data type. More methods can be added to VECTOR as long as they are consistent with the structure. To make things even more complicated, an access list can be implemented within the scope of the `let` clause in Figure 3 so that the vector becomes accessible only to a controlled class of alien objects.

The object-oriented program structure is compatible with the basic data representation and functional style programming of SG. The encapsulated data are represented the same way as was discussed in Section 3 and the entry routines are coded along the line of what was advocated in Section 4. This compatibility and SG's overall flexibility are due to some Scheme features: (1) Lexical scoping, which makes data encapsulation possible. A dynamic scoping like that of Lisp may cause unexpected identifier clash and therefore guarantees no effective information hiding. (2) Functions as first-class entities, which underscores the realization of message passing. In an environment where functions are not

[5] Here the convention is that data types or identifiers bound to certain data types are in upper case if they denote *objects*, and in lower case if they are not.

treated equally with other parametric objects (as in Pascal) or the thunk construct (i.e. expression freezing and thawing) is not allowed (as in Lisp) system prototyping techniques examplified by Figure 3 would be either impossible or very cumbersome.

6 Exception Handling

Some graphics applications manage system-user interaction of their own. Whatever form (mouse-driven or the old-fashioned keyboeard-input, command line interpreter) the user interface may be, an *undo* mechanism is usually available. This mechanism falls as a special case into the general *exception handling* paradigm although the latter is a much more complicated problem than the former.

The basic technique for handling undo is to push previous operations onto a stack and pop them when normal progression of user input pauses upon receiving the undo request. SG takes a different approach. Scheme has a built-in control structure called *continuation* that can potentially be used to deal with a variety of exceptions including the undo mechanism. A continuation is a function of one argument that refers to the rest of the computation. In Scheme whenever an expression is evaluated there is a continuation wanting its result. Normally these ubiquitous continuations are hidden behind the scenes. However, Scheme provides the programmer with a function `call-with-current-continuation` (abbreviated `call/cc`) to access the current continuation. Its standard use follows:

```
<computation-before-call/cc>
(call/cc (lambda (K) <body>))
<rest-of-the-computation>
```

where `K`, the current continuation, is a one-argument function that, when passed a value within the scope of `<body>`, causes the value to be immedaitely returned as the result of `call/cc`. The normal execution is aborted and the system restores its state to the one right before `K` was created and picks up "the rest of the computation" from the expression which lexically follows `call/cc`.

To clarify the idea, suppose a graphics editor takes sequences of characters as commands in an unbuffered mode (i.e. the system accepts each input character as soon as it is typed, no carriage return necessary). Figure 4 shows a skeleton of the editor called `top-level`. It defines a local routine `edit` that, when invoked, reads a single character first (by `readc`), then calls the current continuation `K`. Depending on what the input character is, it triggers the corresponding subroutine with `K` as the argument. For example, if `chr` is the letter `n`, it means the user is requesting a node (e.g. a circle) to be drawn. The corresponding subroutine `node` is called and possibly more input characters are expected to follow in order to complete the node drawing command. At any time during

```
(define top-level
   (letrec ([edit (lambda ()
                     (let ([chr (readc)])
                        (block
                           (call/cc
                              (lambda (K)
                                 (case chr
                                    [n      (node K)]   ; Requesting a node
                                    [e      (edge K)]   ; Requesting an edge
                                    [t      (text K)]   ; Requesting some text
                                    .....               ; Other options
                                 )))
                           (edit))))])
         (edit)))
```

Figure 4: The top level of a simple graphics editor

the session of command input, the characters typed (and thus the system state changed) so far can be undone by entering a special character, say *control-K*. Upon receiving *control-K* in node's scope `(K t)` will restore the system back to the state before `call/cc` was executed. The program then picks up the rest of the computation. In our case it is the recursive invocation of `edit`.

The undo example is a simple application of the continuation facility. In the graphics domain exceptions may happen in many different situations. For instance, matrices involved in 3D transformations may not be compatible for multiplications, control polygon specifications for spline drawings may not have enough end conditions, ... *etc*. The technique described above can be used to handle these exceptions in principle.

7 Conclusions

We have shown how a programming language like Scheme may affect the style and structure of graphics applications. In SG graphics entities are represented and manipulated in a way that resembles their geometric and algebraic counterparts. Furthermore, an object-oriented graphics environment can be modeled with moderate effort. Prototypes of other systems such as turtle geometry [Abelson81] and functional geometry [Hendersen82] have also shown similar compactness.

There is no significant difference between the graphics capability of SG and that of other systems. The strength of SG, however, is that it offers a great deal of flexibility for graphics-related "interactive" programming. Based on the graphics primitives and language properties of SG, user applications can be tuned to meet different requirements gracefully. This feature is generally not available in other interactive graphics environments.

With the advent of personal workstations that are equipped with high resolution bit-mapped displays and linked to high quality laser printers, the notion of programming has changed drastically in recent years. Two overall lessons emerge from the SG experience. The first lesson is that a modern programming environment must incorporate some graphics capability to parallel these hardware breakthroughs. The necessary ingredients for a successful interactive environment described in [Sandewall78] is readily available in Scheme. SG proves that graphics to Lisp systems is a plus. It opens many doors to new research in user interface and graphics programming.

Our second lesson is that graphics applications, though language independent, can be realized with less overhead if the underlying programming language is carefully chosen. To be robust, the graphics environment exposed to the user should incorporate some programming flexibility which normally can be imported from the underlying language.

8 Acknowledgements

We would like to thank the people who were involved in the design and implementation of Scheme Graphics. Final integration and the user manual of SG were done by Wen-Ying Chi. Linear algebra package was hacked by George Springer. Eric Ost implemented functional gemeotry. Thanks are also due to Will Clinger, Dan Friedman, and Mitch Wand, the implementors of Scheme-311.

References

[Abelson81] Harold Abelson and Andrea A. diSessa, *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*, MIT Press, Cambridge, Massachusetts, 1981.

[Backus78] John Backus, "Can programming be librated from the von Neumann style? a functional style and its algebra of programs," *Comm. ACM*, Vol. 21, No. 8 (August 1978), pp. 613-641.

[Chen83a] Peehong Chen, W. Y. Chi, E. M. Ost, L. D. Sabbagh, and G. Springer, "Scheme Graphics Reference Manual," technical report #145, Computer Science Department, Indiana University, August 1983.

[Chen83b] Peehong Chen and L. D. Sabbagh, "A Functional Approach to Geometric Applications in Computer Graphics," technical report #146, Computer Science Department, Indiana University, August 1983.

[Church41] Alonzo Church, "The calculi of lambda conversion," *Annals of Mathematics Studies*, No. 6, Princeton University Press, 1941.

[Dahl70] Ole-Johan Dahl, B. Myhrhaug, and K. Nygaard, "Simula 67 Common Based Language," Publication N. S-22, Oslo: Norwegian Computing Center, October 1970.

[ANSI83] American National Standards Institute, *Reference manual for the Ada Programming Language*, ANSI/MILSTD 1815A, February 1983.

[Fessenden83] Carol Fessenden, W. D. Clinger, D. P. Friedman, and C. T. Haynes, "Scheme 311 Version 4 Reference Manual," technical report #137, Department of Computer Science, Indiana University, February 1983.

[Foderaro80] John K. Foderaro, *The FRANZ LISP Manual*, Computer Science Division, University of California, Berkeley, 1980.

[Goldberg83] Adele Goldberg and Dave Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[GSC79] ACM Graphics Standards Committee, "Status report of the Graphics Standards Committee," *Computer graphics*, Vol. 13, No. 3, August 1979.

[Hendersen82] Peter Hendersen, "Functional geometry," *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, ACM Order No. 552820, 1982.

[McCarthy60] John McCarthy, "Recursive functions of symbolic expressions and their computation by machine," *Comm. ACM*, Vol. 3, No. 4 (April 1960), pp. 185-195.

[Michell79] James G. Mitchell, W. Maybury, and Richard Sweet, "Mesa Language Manual," Version 5.0, Xerox PARC CSL-79-3, April 1979.

[Morris73] James H. Morris Jr., "Protection in programming languages," *Comm. ACM*, Vol. 16, No. 1 (January 1973), pp. 15-21.

[Sandewall78] Erik Sandewall, "Programming in an interactive environment: the "Lisp" experience," *Computing Surveys*, Vol. 10, No. 1 (March 1978), pp. 35-71.

[Séquin83] Carlo H. Séquin, Mark Segal, and Paul Wensley, "UNIGRAFIX 2.0 User's Manual and Tutorial," technical report UCB/CSD 83/161, Computer Science Division, University of California, Berkeley, December 1983.

[Steele78] Guy L. Steele and G. J. Sussman, "The Revised Report on SCHEME, a Dialect of LISP," MIT Artificial Intelligence Laboratory Memo #452, Cambridge, MA, January 1978.

[Steele80] Guy L. Steele and G. J. Sussman, "Design of a LISP-based microprocessor," *Comm. ACM*, Vol 23, No. 11 (Nov. 1980), pp. 628-645.

[Wirth83] Niklaus Wirth, *Programming in Modula-2*, 2nd edition, Springer-Verlag, 1983.