COMPARISON OF DYNAMICALLY AND STATICALLY SCOPED LISP

Asya Campbell

June 1985
Report No. CSD-850024

Comparison of Dynamically and Statically Scoped LISP

Asya Campbell

Comprehensive paper submitted in partial satisfaction of the requirements for M.S. degree in Computer Science.

University of California, Los Angeles

June 1985

Introduction

This paper describes and explores the differences between lexically scoped and dynamically scoped dialects of LISP. Currently, LISP is used as a tool in Artificial Intelligence (AI) - which is one of the most advanced and futuristic fields of computer science. But actually, LISP is one of the oldest languages still in use.

Most of the key ideas of LISP were developed by John McCarthy in 1956 through 1958. And in 1958 through 19€2 the actual programming language was implemented and applied to the problems of artificial intelligence [McCa78]. Parts of LISP were based on theoretical principals of lambda calculus and other parts were added on for the convenience of the users.

Through the years, LISP has developed into many dialects (such as Franz Lisp, InterLisp, MacLisp, T LISP, and other AI languages based on LISP). Now we can not even talk about "standard LISP", because so many implementations have provided their own features.

In this paper I will review two major dialects: dynamically scoped Franz Lisp (based on McCarthy's original LISP, [McCa60], McCa65]) and statically scoped T LISP (based on Guy Steele & Gerald Sussman SCHEME, [S&S 75], [S&S78a]). I will show how the features of these two dialects are reflected in their semantics, and then contrast and compare these two dialects.

The rest of this section describes the general characteristics of LISP-like languages and also gives an introduction to the formal specifications of programming languages. In the next sections, I will present the syntax and the semantics of LISP and SCHEME. A lot of it is taken from the literature and class notes from 232B, Denota-

tional Semantics, taught by professor David Martin, Fall 1984, UCLA; but I also added and changed some of the equations to make the grammars of the two languages more comparable.

LISP is an expression-oriented, applicative order (i.e. in a function application the arguments are evaluated before the function is applied) language with parenthesized syntax known as Cambridge Polish (or prefix polish) notation. The language is designed primarily for symbolic data processing. Symbolic expressions are usually referred to as S-expressions. The most elementary type of S-expression is the atomic symbol (atom). An atom is a string of numerals and letters, starting with a letter.

Examples of atoms:

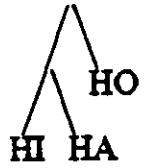
A

B

AB

W47Y

These symbols are called atomic because they are not split into individual characters and are taken as a whole. All S-expressions are built out of atomic symbols and the punctuation marks such as "(", ")", and ".". Internally, S-expressions are represented as LISP trees, which are like binary trees, except that all the information is attached to the leaves (exterior nodes). For example, the S-expression ((HI . HA) . HO) is represented by the following tree:



Next I want to review some basic functions of S-expressions. The first function is CONS. It takes two arguments, and it is used to build S-expressions from smaller S-expressions. So given arguments X and Y, CONS [X,Y] constructs a dotted pair whose left value is X and right value is Y.

For example,

CONS [A,B] = (A . B)

CONS [(A . B), C] = ((A . B) . C)

The next two functions do just the opposite of CONS. They are called CAR and CDR, and they produce subexpressions of a given expression. Both functions take one argument. If the argument is a dotted pair, then CAR returns the first part of the argument and CDR returns the rest of the argument. Both functions are undefined if the argument is atomic.

For example,

CAR [(A . B)] = A

CAR [((A . B) . C)] = (A . B)

CAR [(A)] = A

CAR [A] is undefined

CDR [(A . B)] = B

CDR [((A . B) . C)] = C

CDR [A] is undefined

It is quite common in LISP to chain together CAR and CDR operations. There is a way to abbreviate multiple CAR's and CDR's by forming function names that begin with C, have several A's and/or D's in the middle and end with R.

For example,

CADR [(A . (B . C))] = CAR [CDR [(A . (B . C))]] = CAR [(B . C)] = B

The two remaining basic functions are known as predicates. A predicate is a function whose value is either true or false. In LISP, true is represented by the atom T and false is represented by the atom NIL. The following two definitions are from [McCa65].

The predicate ATOM is true if its argument is an atomic symbol, and false if its argument is composite.

For example,

ATOM [APPLE] = T

ATOM [(A . B)] = NIL

ATOM [CAR [(A . C)]] = T

The predicate EQ is a test for equality on atomic symbols. It is undefined for non-atomic arguments.

For example,

EQ [A, A] = T

EQ [A, B] = NIL

EQ [(A . B), (A . B)] is undefined

EQ [A, (A . B)] is undefined

In real systems the value of undefined is usually represented as NIL, which is the same as false.

It is usually more convenient to write a list of expressions of indefinite length, such as (X Y Z), instead of using dot notation. Such notation is known as list notation. The separator for the elements of a list is blank. Each list (l_1 l_2 ... l_n) can be defined in

terms of dot notation as (l_1 . (l_2 . (... (l_n . NIL) ...))). The atomic symbol NIL serves as a terminator for lists. Accordingly, NIL is also used to represent an empty list ().

For example,

(Z) = (Z . NIL)

(X Y Z) = (X . (Y . (Z . NIL)))

((X Y) Z) = ((X . (Y . NIL)) . (Z . NIL))

((X)) = ((X . NIL) . NIL)

(Z . (X Y)) = (Z . (X . (Y . NIL)))

There is another function, NULL, which is quite often referred to as one of the elementary functions. This predicate is useful for deciding when a list is exhausted. NULL is true if and only if its argument is NIL.

All the elementary functions described above can be applied to S-expressions written in list notation, as well as to S-expressions written in dot notation.

In LISP, there do not exist type declarations. It is a "typeless" language. Also in most LISP dialects there is no distinction between program and data. So, it is possible to manipulate programs as data and to execute data as programs. For more information on LISP the reader is referred to [McCa65] and to [Pleb80], from which most of the above information is taken from.

To describe programming languages formally, people use syntax and semantics. Syntax deals with grammatical structure of programs in the language, while semantics is used to describe the meaning of programs in the language. There has been a lot of work done in syntax, and the rules for defining syntax of programming languages are pretty much defined and clear. In this paper, I will use a BNF (Backus Normal Form) grammar to define the syntaxes of SCHEME and LISP. Defining semantics is more difficult,

mainly due to the complexity of the subject. There are a lot of different approaches to solving this problem.

Each approach should fulfill the following criteria, according to U. Pleban:

i. it should be mathematically sound;

ii. it should be applicable to all programming languages and their features;

iii. it should be relevant to proving program correctness and equivalence;

iv. it should expose the structure of implementations.

There are three major approaches to formal specification of language semantics: operational, denotational, and axiomatic. Operational semantics usually specifies some formal system that models the interpretation of programs. The computational model is an abstract machine with transitions among states. A very good example of this approach is VDL (Vienna Definition Language), [L&W 66], [Wegn72]. Axiomatic semantics provides axioms and rules of inference, similar to the axioms and rules for logical calculus. Axiomatic semantics formulates assertions about a programming language. Denotational semantics takes a functional approach to the specification of the programming language semantics. A denotation is a function usually written in lambda calculus.

In this paper I have chosen the denotational approach as the basis for comparison of the dynamically scoped LISP and the statically scoped SCHEME. This approach is especially well suited for LISP-like languages because LISP is based on lambda calculus. For a description of denotational semantics, the reader should consult the standard references, such as [Tenn76] and [Gord79].

Syntax of LISP 1.0

The following list describes syntactic domains of LISP.

Syntactic Domains:

P : Prog Programs

E : Exp Expressions

F : Fn Functions

id : Id Identifiers

at : At Atoms

bf : BFn basic (elementary) functions

Syntax Equations:

$P \rightarrow F_1 \ldots F_n E \qquad n \geq 0$

The above equation (which I added, myself) defines a program to be any number of function definitions, followed by expression E.

$E \rightarrow$ at

| id

| $(F\ E_1 \ldots E_n) \qquad n \geq 0$

| (SETQ id E)

| $(COND\ (E_{11}\ E_{12}) \ldots (E_{n1}\ E_{n2})) \qquad n > 0$

An expression can be just an atom and then that atom is the value of the expression. It can also be an identifier, then the value of the expression is the S-expression that is bound to that identifier at the time when we evaluate the expression. An expression can also be a function, followed by a list of arguments, which are also expressions. Thus we allow the composition of functions.

An expression can also be of the form (SETQ id E), in which the value of E is assigned to variable id and returns the value of E. The last part of the above rule gives us the format of conditional expressions. Conditional expressions give us ability to branch. The meaning of the conditional expression is: if E_{11} is true, then the value of E_{12} is the value of the entire expression. If E_{11} is false, then if E_{21} is true the value of E_{22} is the value of the entire expression. And so on, searching for the E_{i1} which is true from left to right. If none of the E_{i1}s are true, then the value of the entire expression is undefined, [McCa65].

F -> bf
| id
| (LAMBDA (id_1 ... id_n) E) $\quad n \geq 0$
| (LABEL id F)

bf -> CAR | CDR | CONS | EQ | ATOM | NULL

A function can be a basic (elementary) function. It can be simply an identifier. In this case its meaning must be previously defined in the environment. A function can also be defined using lambda notation, thus establishing a correspondence between the arguments and the formal parameters used in the function. In order to be able to write recursive functions, we introduce the LABEL operator. In (LABEL id F), the name id is

assigned to function F so that a use of id in F can denote a recursive call, [McCa65].

Let Env denote the domain of environments. An environment r : Env keeps track of the bindings of identifiers. So, it is concerned with questions of scope. The environment is a mapping from identifiers to denotable values. Denotable values are things that can be written down in that language. In LISP denotable values are S-expressions and open functions. Open function is the result of function definition. It takes in run-time environment, return address (expression continuation), and parameters (S-expressions), and gives back a final answer. There is no need to introduce locations, since none of the basic functions can modify list structure, [M&P 80].

Continuations are a functional representation of the concept of flow of control. For example, continuation for a command denotes the effect of the rest of the computation, which is not always the same as this command's textual successors in the program. Expression continuations in LISP are mappings from the domain of S-expressions (result of the function) to the domain of final answers. Answer a : Ans is either an S-expression or an Error. A summary of semantic domains is listed below.

Semantic Domains:

Q : unspecified	Atomic Values (quotations)
e : S = Q + (S x S)	S-expressions
a : Ans = S + Error	Answers
k : Ec = S -> Ans	Expression continuations
d : Dv = S + OFun	Denotable values
r : Env = Id -> (Dv + {*unbound*})	Environments
f : OFun = Env -> Ec -> S* -> Ans	open functions

z : Fc = OFun -> Ans Function continuations

There are four semantic functions: **P**, **B**, **E**, and **F**, whose definition constitute the core of the standard semantics of LISP. They are listed below. **B** maps literal atoms to atomic values. Since the details of this mapping are irrelevant, it is left unspecified. **E** takes an expression, an environment, and an expression continuation and gives us a final answer. **F** takes a function, an environment, and a function continuation and gives us a final answer.

Semantic Functions:

P : Prog -> Ans

B : At -> Q (unspecified)

E : Exp -> Env -> Ec -> Ans

F : Fn -> Env -> Fc -> Ans

Semantic Equations:

(E1) **E**[at] (r) (k) = k (**B**[at])

All atoms are just handed to the function **B** and the result is fed into expression continuation k.

(E2) $\mathbf{E}$[id] (r) (k) = (r(id) = *unbound* -> UNBOUND-VAR,
(isS (r(id)) -> k (r(id)),
NON-S-EXPR
)
)

The value of the variable id is looked up in the environment r. If it is *unbound*, then an error message is produced, otherwise, if it is an S-expression then the value found is passed to the rest of the program, otherwise, an error message is produced.

(E3) $\mathbf{E}$[(F E_1 ... E_n)] (r) (k)

= $\mathbf{F}$[F] (r);
λf . $\mathbf{E}$[E_1] (r);
λe_1 . $\mathbf{E}$[E_2] (r);
.
.
.
$\lambda e_{(n-1)}$. $\mathbf{E}$[E_n] (r);
λe_n . f (r) (k) (e_1, ..., e_n)

In the above function evaluation, function F is evaluated in run-time environment r to get a function or a function representation, which then can be applied to the parameters e_1, ..., e_n.

(E4) $\mathbf{E}[(\text{COND } (E_{11}\ E_{12}) \ldots (E_{n1}\ E_{n2}))]\ (r)\ (k)$

$= \mathbf{E}[E_{11}]\ (r);$
cond $(\mathbf{E}[E_{12}]\ (r)\ (k),$
$\mathbf{E}[E_{21}]\ (r);$
cond $(\mathbf{E}[E_{22}]\ (r)\ (k),$

.

.

$\mathbf{E}[E_{n1}]\ (r);$
cond $(\mathbf{E}[E_{n2}]\ (r)\ (k),$
UNDEF-COND
)
)
)

where
cond: (Ans x Ans) -> S -> Ans

cond (a_1, a_2) (e) = a_2 if e has the value NIL and a_1 otherwise

If E_{11} evaluates to non-NIL value, then the value of the COND form is the value of E_{12}; otherwise, we check the value of E_{21} and so on.

(E5) $\mathbf{E}[(\text{SETQ id E})]\ (r)\ (k)$

$= \mathbf{E}[E]\ (r);$
$\lambda e.k\ (e/r(id))$

Value of E is assigned to variable id and then it is passed to expression continuation k.

(F1) **F**[CAR] (r) (z) = z (car)

where
car: OFun

car (r) (k) (e) = (e = <e_1, e_2> -> k (e_1),
BAD-CAR-ARG
)

F[CDR] (r) (z) = z (cdr)

where
cdr: OFun

cdr (r) (k) (e) = (e = <e_1, e_2> -> k (e_2),
BAD-CDR-ARG
)

F[CONS] (r) (z) = z (cons)

where
cons: OFun

cons (r) (k) (e_1, e_2) = (isS (e_1) and
isS (e_2) -> k (<e_1, e_2>),
BAD-CONS-ARG
)

F[EQ] (r) (z) = z (eq)

where
eq: OFun

eq (r) (k) (e_1, e_2) = (isAtom (e_1) and
isAtom (e_2) -> (e_1 = e_2 -> T,
NIL
),
BAD-EQ-ARG
)

```
F[ATOM] (r) (z) = z (atom)

where
    atom: OFun

    atom (r) (k) (e) = (isAtom (e) -> T,
                                    (isS (e) -> NIL,
                                                BAD-ATOM-ARG
                                    )
                        )
```

```
F[NULL] (r) (z) = z (null)

where
    null: OFun

    null (r) (k) (e) = (isNull (e) -> T,
                                      NIL
                        )
```

The function CAR produces a subexpression of a given expression e. If e is a dotted pair, then CAR sends the first part of the argument to continuation k, otherwise, an error message is produced. The rest of the basic functions are similar.

```
(F2)  F[id] (r) (z) =  (r(id) = unbound -> UNBOUND-FUN,
                                    (isOFun (r(id)) -> z (r(id)),
                                                       NON-FUN
                                    )
                        )
```

The value of the identifier is looked up in the environment. If it is *unbound*, an error message is produced. Otherwise, if it is an open function then its value is passed to the function continuation, else another error message is produced.

(F3) $\mathbf{F}[(\text{LAMBDA } (id_1 \ldots id_n)\ E]\ (r)\ (z) = z\ (f)$

where
$f = \lambda r'\ .\lambda k.\lambda(e_1, \ldots, e_n).$
$\mathbf{E}[E]\ (r'\ [e_1, \ldots, e_n/id_1, \ldots, id_n]\ (k)$

When we see a LAMBDA expression, we construct its denotation, f, and it is passed to the function continuation z. f is defined so that when the LAMBDA expression is applied to the arguments, the LAMBDA variables $id_1, \ldots, id_n$ (formal parameters) are bound to actual parameters $e_1, \ldots, e_n$. The body E is evaluated with these bindings, r' is the call-time environment.

(F4) $\mathbf{F}[(\text{LABEL id F})]\ (r)\ (z)$
$= \mathbf{F}[F]\ (r)\ (\lambda f.z\ (\lambda r'\ .f(r'\ [f/id])))$

LABELs were introduced in order to be able to write recursive functions. The call-time environment r' is modified by binding of f to name id; f is result of $\mathbf{F}[F]$.

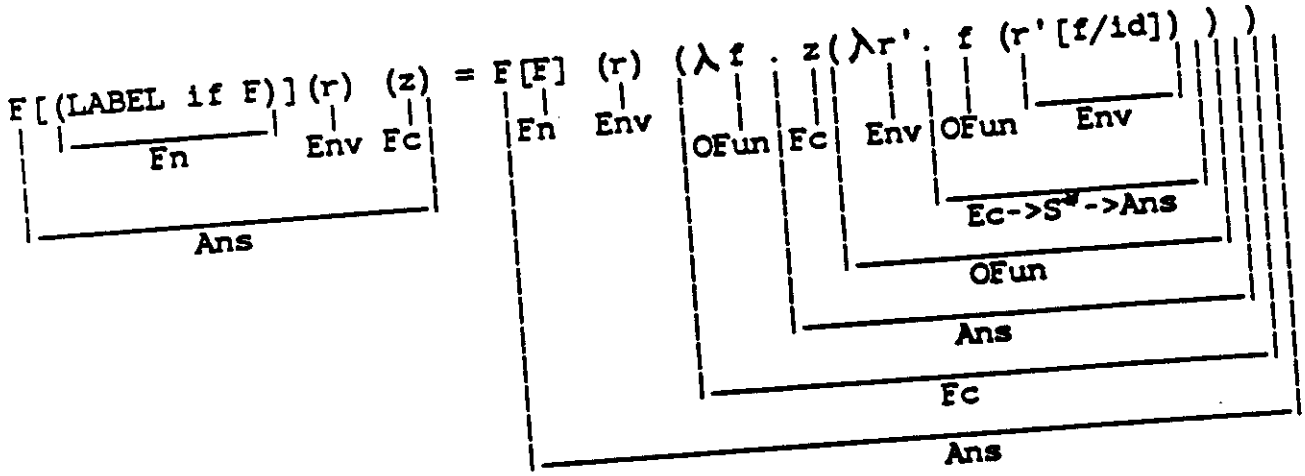
Most of the above information is from [McCa60], [McCa65], and [Gor 75].

As an example of checking the types, consider the type checking for the equation (F4). I will do it from inside out. On the left hand side (LABEL id F) is of type Fn, r is Env, and z is Fc. The result of that is of type Ans, since Fn -> Env -> Fc -> Ans. On the right hand side, it is a little more involved:

$r'\ [f/id]$ is of type Env. f is OFun. Thus, $f(r'\ [f/id])$ is Ec -> S* -> Ans. $\lambda r'$ is of type Env. Thus, $\lambda r'\ .f(r'\ [f/id])$ is OFun. z is Fc, and since Fc = OFun -> Ans, we get Ans. λf is OFun. OFun -> Ans gives us Fc. Therefore, $(\lambda f.z\ (\lambda r'\ .f(r'\ [f/id])))$ is of type Fc. F is of type Fn, r is of type Env. Thus, the whole right hand side gives us

Ans (Fn -> Env -> Fc -> Ans).

The following diagram illustrates the above type checking.



We can check types like that for every equation, but because it is time consuming and not very entertaining, I will not bore the reader with that.

I could not find anywhere in the literature the equation for the LISP program. So, based on my syntax equation above, I defined the following equation for the semantics of LISP 1.0.

$$(P)\quad \mathbf{P}[(\text{LABEL } id_1\ F_1) \ldots (\text{LABEL } id_n\ F_n)\ E] \qquad n \geq 0$$

$$= \mathbf{F}[F_1]\ (r_0);\ \lambda f_1.\ \mathbf{F}[F_2]\ (r_0);$$

$$\vdots$$

$$\lambda f_{(n-1)}.\ \mathbf{F}[F_n]\ (r_0);\ \lambda f_n.(\mathbf{E}[E]\ (r_0[r'])\ (k_0))$$

where, $r' = f_1 \ldots f_n/id_1 \ldots id_n$

This equation defines n functions in the initial environment r_0, and then evaluates an expression E in that environment, modified by the definition of the functions, and using the initial expression continuation k_0.

Initial Environment

r_0 = CAR, CDR, CONS, .../car, cdr, cons, ...

CAR, CDR, CONS, etc. are identifiers; car, cdr, cons, etc. are OFun.

Initial Continuation

$k_0 = \lambda s.s$

The answer of the program is the value of E, which is an S-expression.

Syntax of SCHEME

In [S&S 75], [S&S78a] and related papers, Steele and Sussman describe a dialect of LISP called SCHEME. The major difference between SCHEME and LISP is that SCHEME is lexically scoped. It has considerable significance for the ease with which SCHEME can be compiled. In fact, Steele has written a compiler for SCHEME, called RABBIT, [Stee78].

The following list represents the syntactic domains of SCHEME. Most of it is from [S&S 75], [S&S 78a], [Pleb79], and [Pleb80].

Syntactic Domains:

P : Prog	Programs
D : Def	Global function definitions
F : Fn	Named abstractions
A : Abs	Unnamed abstractions
E : Exp	Expressions
S : S-exp	S-expressions
id : Id	Identifiers
at : At	Atoms
num: Num	Numbers (integer and reals)

Syntax Equations:

P -> D E

| E

D -> (DEFINE (F_1 ... F_n)) $n > 0$

F -> (id A)

A -> (LAMBDA (id_1 ... id_n) E) $n \geq 0$

To define (possibly) mutually recursive functions globally we use DEFINE. Each of the named abstractions F_i is of the form (id A), where id is just an identifier and A is a lambda expression defined above.

E -> T
| NIL
| (QUOTE S)
| num
| id
| A
| (ASET! id E)
| (E_0 E_1 ... E_n) $n \geq 0$
| (IF E_1 E_2)
| (IF E_1 E_2 E_3)
| (LABELS (F_1 ... F_n) E) $n > 0$
| (CATCH id E)
| (THROW id E)

S -> at

| num

| (S_1 . S_2)

An expression is either T or NIL, which are literal atoms, or (QUOTE S), which is a constant - S-expression. It can also be an identifier, like in LISP, or a number. Having number in SCHEME but not in LISP is not very important, since the domain of atoms in LISP is not specified, so we can say that a number is an atom. An expression can also be of a form (ASET! id E), where the value of E is assigned to variable id. Therefore, ASET! behaves like SETQ in LISP. An expression can also be an A, unnamed abstraction, which is just a lambda expression of the form (LAMBDA (id_1 ... id_n) E). In contrast to LISP, a lambda expression in SCHEME always evaluates to a closure, i.e. it is associated with an environment containing bindings for its free variables, thus achieving lexical scoping, [Pleb80].

Both IF statments in SCHEME are just simplified versions of the LISP conditional COND. LABELS is like LABEL in LISP except that it is easier to define mutually recursive functions using LABELS. The mutually recursive functions are defined only for the evaluation of the body of the LABELS form.

Semantics of SCHEME

The basic values in SCHEME, as in LISP, are atoms: integers, reals, and literal atoms. Domain of integers is denoted by N, domain of reals by R, and domain of literal atoms (quotations) by Q.

Let e denote a member of the domain of expressible values Ev, which are also storable values. It can be a basic value, a closed function, an expression continuation, or it can have a CAR-part and a CDR-part, which are both in Ev. Having closed functions here is necessary, since functions are first-class objects in SCHEME, which means that they may be assigned to a variable, passed to a function, produced as a result by a function, and used as atomic constituents of data structures.

Expression continuations model SCHEME escape objects, which are also first-class objects. Answer a : Ans is simply an expressible value or an error. In SCHEME, identifiers can only denote locations, thus denotable value d : Dv is identical to location l : Loc, which is left unspecified.

Environments r : Env, map identifiers to their denotations or {*unbound*}. Stores s : Store, map locations to their contents (the storable values) or {*unused*}. The run-time information is kept in the store. The compile-time information is kept in the environment.

SCHEME functions are mapped to closed functions f, member of the domain CFun. Such a function takes a list of evaluated arguments from expressible values Ev and expression continuation Ec, and produces a command continuation. Closed functions in SCHEME, in contrast to the open functions in LISP, do not take environment as their parameter. SCHEME is statically scoped, so you do not need a run-time en-

vironment as input to the function. You also do not need a compile-time environment for closed functions in SCHEME because it can be figured out at compile time.

There are also three domains of continuations. They are command continuations c : Cc, expression continuations k : Ec, and definition continuations u : Dc.

The above discussion is summarized in the following list of domain equations, [Pleb80].

Semantic Domains:

Q : unspecified	Atoms (quotations)
N	Integers
R	Reals
B = Q + N + R	Basic Values
e : Ev = B + Cfun + Ec + (Ev x Ev)	Expressible values
l : Loc	Locations (unspecified)
r : Env = Id -> (Dv + {*unbound*})	Environments
s : Store = Loc -> (Sv + {*unused*})	Stores
e : Sv = Ev	Storable values
d : Dv = Loc	Denotable values
a : Ans = Ev + Error	Answers
f : CFun = Ev* -> Ec -> Cc	Closed functions
c : Cc = Store -> Ans	Command continuations
k : Ec = Ev -> Cc	Expression continuations
u : Dc = Env -> Cc	Definition continuations

The following list of seven semantic interpretation functions constitutes the core of standard semantics of SCHEME. Those semantic functions are: **P**, **D**, **F**, **A**, **E**, **S**, **B**. In SCHEME, the Env for **E** and **F** is run-time (call-time) environment. We have to distinguish between call-time and compile-time (defining) environments. The run-time information is kept in the store. The compile-time information is kept in the environment. We use locations to keep track of the run-time information. That is why we need locations in SCHEME.

Semantic Functions:

P : Prog -> Ans

D : Def -> Env -> Dc -> Cc

F : Fn -> Env -> Dc -> Cc

A : Abs -> Env -> CFun

E : Exp -> Env -> Ec -> Cc

S : Sexp -> Ec -> Cc

B : (At + Num) -> B (unspecified)

Semantic Equations:

(S1) **S**[at] (k) = k (**B**[at])

(S2) **S**[num] (k) = k (**B**[num])

All atoms and numbers are just handed to the function **B** and the result is fed into expression continuation k.

(S3) $\mathbf{S}[S_1 . S_2]$ (k)

$= \mathbf{S}[S_1];$
$\lambda e_1.\mathbf{S}[S_2];$
$\lambda e_2.\mathbf{alloc}^*(1);$
$\lambda l.\lambda s.k(<e_1, e_2>)\ (s[<e_1, e_2>/l])$

Strictly speaking, l is a vector of 1 (one) location. An S-expression with CAR-part and a CDR-part causes a location to be allocated and an operation cons to be performed.

(E1) $\mathbf{E}[T]\ (r)\ (k) = \mathbf{S}[T]\ (k)$

(E2) $\mathbf{E}[NIL]\ (r)\ (k) = \mathbf{S}[NIL]\ (k)$

(E3) $\mathbf{E}[(QUOTE\ S)]\ (r)\ (k) = \mathbf{S}[S]\ (k)$

(E4) $\mathbf{E}[num]\ (r)\ (k) = \mathbf{S}[num]\ (k)$

All the above trivial cases are just handed to the function **S**.

(E5) $\mathbf{E}[id]\ (r)\ (k) = (r(id) = \textit{unbound}$ -> UNBOUND-VAR,
contents (r(id)) (k)
)

The value of the variable id is looked up in the environment r. If it is *unbound*, then error message is produced, otherwise, the utility function **contents** is called. (See below for the explanation of all the utility functions.)

(E6) $\mathbf{E}[\![A]\!]\ (r)\ (k) = k\ (\mathbf{A}[\![A]\!]\ (r))$

The above just passes the result of $\mathbf{A}[\![A]\!]$ in the environment r to the expression continuation k.

(E7) $\mathbf{E}[\![(\text{ASET! id E})]\!]\ (r)\ (k)$

$$= (r(id) = \textit{unbound} \rightarrow \mathbf{err}(\text{UNBOUND-VAR}),\ \mathbf{E}[\![E]\!]\ (r);\ \lambda e.\lambda s.k\ (e)\ (s[e/r(id)]))$$

ASET! behaves exactly like SETQ in LISP, except that we first check whether variable id is bound to a location.

(E8) $\mathbf{E}[\![(E_0\ E_1\ \ldots\ E_n)]\!]\ (r)\ (k)$

$$= \mathbf{E}[\![E_0]\!]\ (r);\ CFun?;\ \lambda f.\mathbf{E}[\![E_1]\!]\ (r);\ \lambda e_1.\mathbf{E}[\![E_2]\!]\ (r);\ \ldots\ \lambda e_{(n-1)}.\mathbf{E}[\![E_n]\!]\ (r);\ \lambda e_n.f(\langle e_1, \ldots, e_n\rangle)\ (k)$$

In (E8) E_0 is evaluated once, and if it evaluates to a function, then the result is applied to the arguments $E_1, \ldots, E_n$, which are evaluated from left to right.

(E9) $\mathbf{E}[(\text{IF } E_1\ E_2)] = \mathbf{E}[(\text{IF } E_1\ E_2\ \text{NIL})]$

(E10) $\mathbf{E}[(\text{IF } E_1\ E_2\ E_3)]\ (r)\ (k)$

$= \mathbf{E}[E_1]\ (r);$
$\quad \text{cond}(\mathbf{E}[E_2]\ (r)\ (k), \mathbf{E}[E_3]\ (r)\ (k))$

where
cond: (Cc x Cc) -> Ev -> Cc

cond $(c_1, c_2)\ (e) = c_2$ if $e = \mathbf{B}[\text{NIL}]$ and c_1 otherwise

If E_1 evaluates to non-NIL, then the value of the IF form is the value of E_2; otherwise, it is the value of E_3. Actually, it is just a simplified version of general LISP COND form.

(E11) $\mathbf{E}[(\text{LABELS } ((id_1\ A_1) \ldots (id_n\ A_n))\ E)]\ (r)\ (k)$

$= \mathbf{alloc}^*\ (n);$ allocate n locations
$\quad \lambda(l_1, \ldots, l_n).$
$\quad\quad \mathbf{F}[(id_1\ A_1)]\ (r');$
$\quad\quad\quad \lambda r_1.\mathbf{F}[(id_2\ A_2)]\ (r');$
$\quad\quad\quad\quad \cdot$
$\quad\quad\quad\quad \cdot$
$\quad\quad\quad\quad \cdot$
$\quad\quad\quad\quad \lambda r_{(n-1)}.\mathbf{F}[(id_n\ A_n)]\ (r');$
$\quad\quad\quad\quad\quad \lambda r_n.\mathbf{E}[E]\ (r'')\ (k)$

where
$r' = r\ [l_1, \ldots, l_n/id_1, \ldots, id_n]$
$r'' = r\ [r_1]\ [r_2] \ldots [r_n]$

Evaluate E relative to r and s modified by the (possibly) mutually recursive (local) function definitions $(id_i\ A_i)$, $i = 1, 2, \ldots, n$. The identifiers (function names) $id_1, \ldots, id_n$ are bound to the values of the LAMBDA expressions $A_1, \ldots, A_n$ respectively. The body E of LABELS form is evaluated in an extended environment r'', where the local bindings for $id_1, \ldots, id_n$ are known, [Pleb80]. The environments r, r' and r'' are compile-time environments.

(E12) **E**[(CATCH id E)] (r) (k)

$$= \textbf{ialloc}^{*}\ (<k>);\ \lambda l.\textbf{E}[E]\ (r[l/id])\ (k)$$

The above is like a label definition in FORTRAN. The CATCH variable id is bound to a memory location which is initialized to the current expression continuation (the "escape object"), and the body E is evaluated with these new bindings. If during this evaluation (THROW id E) is encountered, then the rest of the evaluation of the CATCH body E is discontinued, and the value of the body E of the THROW is returned as the value of the CATCH.

(E13) **E**[(THROW id E)] (r) (k)

$$= (r(id) = \textit{unbound} \rightarrow \textbf{err}(\text{UNBOUND-VAR}),\ \textbf{contents}\ (r(id));\ Ec?;\ \lambda k'\ .\textbf{E}[E]\ (r)\ (k')\ )$$

The above is like a GOTO in FORTRAN. The first occurrence of k' is regarded as a value, and the second occurrence regarded as continuation. The identifier id must be bound to an "escape object", which is then applied to the value of the expression E. This effectively returns from the CATCH expression which caused the creation of the "escape object".

Actually, CATCH and THROW operators are found in some LISP 1.0 implementations, for example, Franz Lisp.

(A) **A**[(LAMBDA (id_1 ... id_n) E] (r)

$$= \lambda e^*.\lambda k.(\mathrm{lg}(e^*) = n \;\text{->}\; \mathbf{ialloc}^*(e^*);\ \lambda(l_1, \ldots, l_n).(\mathbf{E}[\![E]\!]\ (r')\ (k)\ ),\ \mathbf{err}(\text{ARG-MISMATCH}))$$

where $r' = r[l_1, \ldots, l_n/id_1, \ldots, id_n]$

When we see a LAMBDA expression, we construct a denotation so that the LAMBDA variables $id_1, \ldots, id_n$ are bound to new initialized locations $l_1, \ldots, l_n$. The body E is evaluated with these new bindings.

$$(F)\quad \mathbf{F}[\![(\mathrm{id}\ A)]\!]\ (r)\ (u) = \lambda s.(r(id) = \textit{unbound} \;\text{->}\; \text{UNBOUND-VAR},\ u(r(id)/id)\ (s[\mathbf{A}[\![A]\!]\ (r)/r(id)]))$$

The function value produced by $\mathbf{A}[\![A]\!]$ (r) is assigned to the location to which identifier id is bound. It updates the states, which are used in (E11) (although they cancel out of the equation).

(D) $\mathbf{D}[(\text{DEFINE } ((\text{id}_1\ A_1) \ldots (\text{id}_n\ A_n)))]\ (r)\ (u)$

$= \mathbf{alloc}^*\ (n);$ allocate n locations
$\lambda(l_1, \ldots, l_n).$
$\mathbf{F}[(\text{id}_1\ A_1)]\ (r');$
$\lambda r_1.\mathbf{F}[(\text{id}_2\ A_2)]\ (r');$
.
.
.
$\lambda r_{(n-1)}.\mathbf{F}[(\text{id}_n\ A_n)]\ (r');$
$\lambda r_n.u(r' ')$

where
$r' = r\ [l_1, \ldots, l_n/\text{id}_1, \ldots, \text{id}_n]$
$r' ' = r_1[r_2] \ldots [r_n]$

The above equation handles the (global) definition of n functions ($\text{id}_i\ A_i$), i = 1, 2, ..., n (which may be mutually recursive) in the environment and store. The equation is very similar to the one for LABELS (E11). The little environments $r_1, \ldots, r_n$ are combined and passed to the original definition continuation u. The environment r is compile-time environment (actually it is initial environment).

(P1) $\mathbf{P}[D\ E] = \mathbf{D}[D]\ (r_0)\ (\lambda r' .\mathbf{E}[E]\ (r_0[r'])\ (k_0))\ (s_0)$

(P2) $\mathbf{P}[E] = \mathbf{E}[E]\ (r_0)\ (k_0)\ (s_0)$

The above two equations define a SCHEME program by first defining all the functions in the initial environment r_0, and then evaluating E in the environment r_0 modified by r' , which contains all the definitions of the above functions, with the initial expression continuation k_0 and the initial store s_0.

Initial Environment

$r_0 = l_{CAR}, l_{CDR}, l_{CONS}, .../CAR, CDR, CONS, ...$

Initial Continuation

$k_0 = \lambda e.\lambda s.e$ take final value, ignore final store

Initial Store

$s_0 = f_{CAR}, f_{CDR}, f_{CONS}, .../l_{CAR}, l_{CDR}, l_{CONS}, ...$

Where $f_{CAR}, f_{CDR}, f_{CONS}, ...$: CFun are SCHEME primitive function definitions in preallocated locations $l_{CAR}, l_{CDR}, l_{CONS}, ...$

Utility Semantic Functions

Function **alloc*** allocates $n \geq 0$ distinct unused locations in s.

alloc* : N -> (Loc* -> Cc) -> Cc Allocate storage

alloc* (n) (t) (s)

= t(<l_1, ..., l_n>) (s), $n \geq 0$
where,
$s(l_1) = ... = s(l_n) =$ ***unused***
= STORAGE-FULL, otherwise

Function **ialloc*** allocates initialized storage.

ialloc* : Ev^* -> (Loc^* -> Cc) -> Ec

ialloc* (e^*) (t)

$$= \textbf{alloc}^* \; (lg(e^*)); \; \lambda l^*.\lambda s.\; t(l^*)\;(s[e^*/l^*])$$

where,
$e^* = \langle e_1, \ldots, e_n \rangle$
$l^* = \langle l_1, \ldots, l_n \rangle$

Function **contents** checks if location l is in use, and if it is, passed to the expression continuation k; otherwise, an error is produce

contents : Loc -> Ec -> Cc

contents (l) (k) (s) = (s(l) = *unused* -> UNINIT-LOC, k (s(l)) (s)

)

SCHEME versus LISP

SCHEME and LISP 1.0 were described above in structural denotational semantics, i.e. semantics in which the meanings of syntactic constructs are given in terms of the meanings of their parts, [M&P 80]. The implementation of SCHEME (T Lisp) follows that denotational semantics. But most of the LISP 1.0 implementations (such as Franz Lisp and MacLisp) have added a lot of features, like availability to the programmer of EVAL and APPLY, and evaluation of the function position E_0 in an application (E_0 E_1 ... E_n) many times. That is why the structural definition of modern LISP languages is not possible.

We have presented the syntax and semantics of standard subsets of SCHEME and LISP. Besides relatively minor differences between the two, such as the usage of IF and LABELS instead of COND and LABEL, SCHEME has very different semantic flavor from LISP. The following is a list of the most important differences between the two:

1. Free variables in LAMBDA expressions are lexically bound in SCHEME and are dynamically bound in LISP. This makes the construction of a compiler for SCHEME easier than for LISP. It also makes it easier to debug programs in SCHEME than in LISP, because in SCHEME all the references to a local variable binding are textually apparent in the program. Thus, the connection between binding and reference is unchanged at run-time, [S&S78b].

2. In some LISP systems, there is SET operator in addition to SETQ operator. (SET E_0 E_1) is like a (SETQ id E) except that instead of id we can have an arbitrary S-expression E_0, which is evaluated. In SCHEME there is no analog to SET operator, only to SETQ. In SCHEME there is a distinction between atoms which

are used as a constants, and atoms which are used as variables. As Steele and Sussman mention in [S&S78a], "although the case of a general evaluated expression for the variable name causes no real semantic difficulty, it can be confusing to the reader. Moreover, in two years we [Steele and Sussman] have not found a use for it. Therefore, we have replaced ASET with ASET!." Another reason for leaving it out of the definition of SCHEME, is so that SCHEME will have a structural denotational semantic definition.

3. In SCHEME there is a sharp distinction between S-expressions, which define data, and LAMBDA expressions, which define functions. Thus, user-defined functions do denote functions in the mathematical sense. This is the most important semantic difference between SCHEME and LISP. In LISP, program and data are not distinguished. This is easy to do in dynamically scoped LISP because global variables do not need to be bound until run-time anyway. LAMBDA expressions in LISP generally do not denote functions (except when an explicit word FUNCTION is used), but are just function representations in the form of S-expressions and therefore, they can be used and manipulated as data. In SCHEME, because it is statically scoped, every LAMBDA expression evaluates to a function, i.e. it is closed in the definition environment. The only operation which is defined for a function in SCHEME is invocation. But also, functions in SCHEME are treated as first-class objects, i.e. they may be passed as arguments to functions, returned as a value from functions, and they may be atomic parts of data structures.

Most of the above is from [M&P 80], [Pleb79], and [Pleb80].

In this paper I described the similarities and differences between statically scoped SCHEME and dynamically scoped LISP. Both languages were described using denotational semantics. Compiling SCHEME into efficient object code is possible because it is statically scoped. That also makes debugging easier in SCHEME than in LISP. But there are some applications which require the flexibility of interpretive execution and the on-the-fly creation of executable code it allows, [M&P 80]. This reason and also heavy usage of LISP by AI community makes it very unlikely that SCHEME will replace LISP in the very near future.

Finally, I have included in the appendicies a working program written in Franz Lisp and T LISP to demonstrate the differences in the output of that program due to the differences in scoping of two dialects. I also included a contour model to show the differences between dynamic and static bindings. There is also a copy of the same program, written in a block structured language, like pseudo-pascal.

Appendix I.

This is an example of running a program in Franz Lisp, which is dynamically scoped.

```
(de dynamic ()
  (setq globalx 0)
  (proc2 )
)

(de proc2 ()
  ((lambda (globalx) (proc1)) 3)
)

(de proc1 ()
  globalx
)
```

```
Franz Lisp, Opus 38.50
-> (load 'dynamic)
[load dynamic.l]
t
-> (dynamic)
3
-> globalx
3
```

Appendix II.

This is an example of running the same program in T LISP, which is statically scoped.

```
(herald static)

(define (static )
  (set globalx 0)
  (proc2 )
)

(define (proc2 )
  ((lambda (globalx) (proc1)) 3)
)

(define (proc1 )
  globalx
)
```

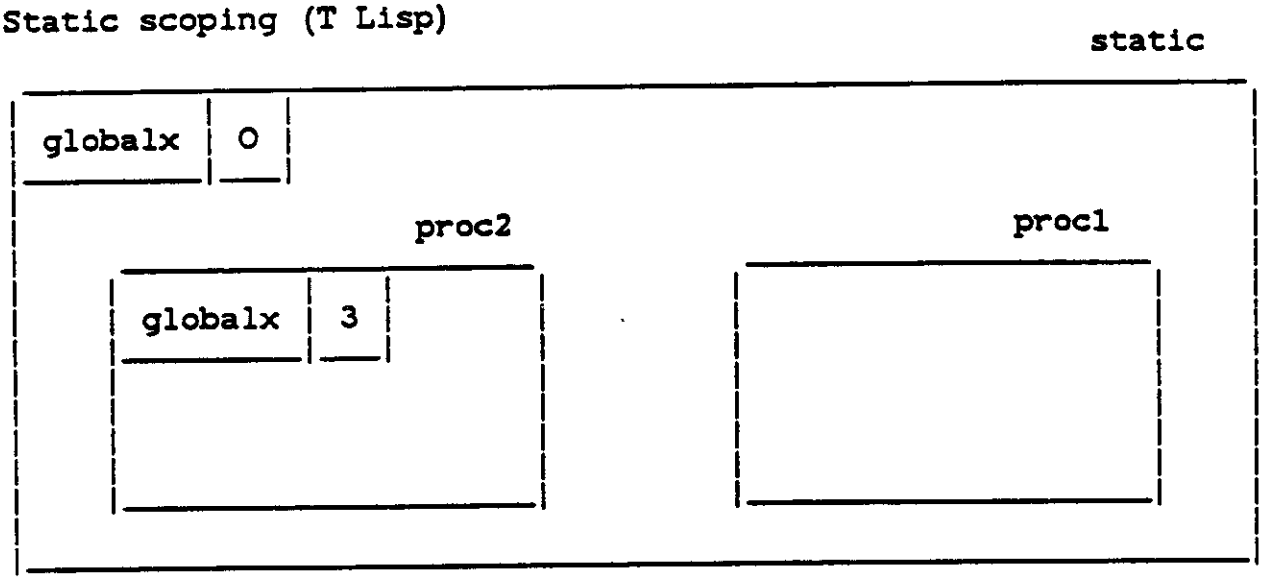
```
T 2.8 (132)  VAX11/UNIX Copyright (C) 1984 Yale University
> (load 'static)

;Loading "static.t" into *SCRATCH-ENV*
STATIC PROC2 PROC1 #{Procedure 2 PROC1}
> (static)
[Binding GLOBALX] 0
> globalx
0
```

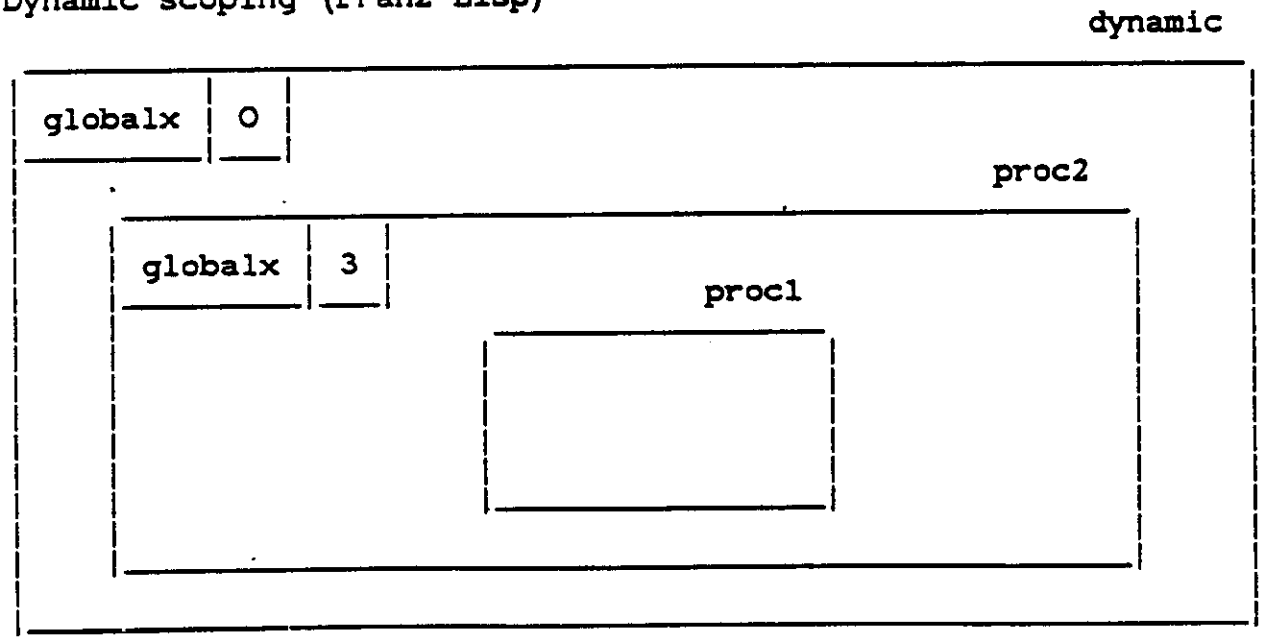
Appendix III.

Contour model of the same program shows the difference between static and dynamic bindings. Value of the variable, globalx, is printed out from procedure proc1.

Static scoping (T Lisp)



Dynamic scoping (Franz Lisp)



Appendix IV.

This is example of the same program written in a block structured language.

```
program block ;

var
  globalx : integer;

{************  PROC1  ************}

  procedure proc1;

  begin
     writeln (globalx)
  end;

{************  PROC2  ************}

  procedure proc2;

  var
    globalx : integer;

  begin
     globalx := 3;
     proc1
  end;

{************  MAIN  ************}

begin
   globalx := 0;
   proc2
end.
```

REFERENCES

[Gord75] Gordon, Michael J. C., Operational Reasoning and Denotational Semantics, Memo AIM-264, Artificial Intelligence Laboratory, Stanford University, CA, August 1975.

[Gord79] Gordon, Michael J. C., The Denotational Description of Programming Languages, Spring-Verlag, New York, 1979.

[L&W 66] Lucas, P. & Walk, K., On the Formal Description of PL/1, Annual Review in Automatic Programming, vol. 6, part 3, Pergamon Press, 1966.

[McCa60] McCarthy, John, Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, CACM, vol. 3, no. 4, pp. 184-195.

[McCa65] McCarthy, John, Paul W. Abrahams, David J. Edwards, Timothy P. Hunt & Michael I. Levin, LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, Massachusetts, 1965.

[McCa78] McCarthy, John, History of LISP, ACM SIGPLAN Notices, History of Programming Languages Conference, vol. 13, no. 8, August 1978, pp. 217-223.

[McCa80] McCarthy, John, LISP - Notes on its Past and Future, Conference Record of the 1980 LISP Conference, Standford University, CA, August 1980.

[McDe80] McDermott, Drew, An Efficient Allocation Scheme in a Interpreter for a Lexically-scoped LISP, Conference Record of the 1980 LISP Conference, Standford University, CA, August 1980, pp. 154-162.

[M&P 80] Muchnik, Steven S. & Pleban, Uwe F., A Semantic Comparison of LISP and SCHEME, Conference Record of the 1980 LISP Conference, Standford University, CA, August 1980, pp. 56-64.

[Pleb79] Pleban, Uwe F., The Standard Semantics of a Subset of SCHEME, a Dialect of LISP, Technical Report TR-79-3, Department of Computer Science, University of Kansas, Lawrence, Kansas, July 1979.

[Pleb80] Pleban, Uwe F., A Denotational Approach to Flow Analysis and Optimization of SCHEME, a Dialect of LISP, Ph.D. Dissertation, University of Kansas, Lawrence, Kansas, 1980.

[Stee77] Steele, Guy Lewis Jr., Debunking the "Expensive Procedure Call " Myth or, Procedure Call Implementation Considered Harmful or, Lambda: The Ultimate GOTO, Memo 443, Artificial Intelligence Laboratory, M.I.T., October 1977.

[Stee78] Steele, Guy Lewis Jr., RABBIT: A Compiler for SCHEME, Technical Report AI-TR-474, Artificial Intelligence Laboratory, M.I.T., January 1978.

[S&S 75] Steele, Guy Lewis Jr. & Gerald Jay Sussman, SCHEME, An Interpreter for Extended Lambda Calculus, Memo 349, Artificial Intelligence Laboratory, M.I.T., December 1975.

[S&S78a] Steele, Guy Lewis Jr. & Gerald Jay Sussman, The Revised Report on SCHEME, a Dialect of LISP, Memo 452, Artificial Intelligence Laboratory, M.I.T., January 1978.

[S&S78b] Steele, Guy Lewis Jr. & Gerald Jay Sussman, The Art of the Interpreter or, The Modularity Complex (Parts Zero, One, and Two), Memo 453, Artificial Intelligence Laboratory, M.I.T., May 1978.

[S&S 80] Steele, Guy Lewis Jr. & Gerald Jay Sussman, The Dream of a Lifetime: A Lazy Variable Extent Mechanism, Conference Record of the 1980 LISP Conference, Standford University, CA, August 1980, pp. 163-172.

[Tenn76] Tennent, R. D., The Denotational Semantics of Programming Languages, Communications of the ACM vol. 19, no. 8, August 1976, pp. 437-453.

[Wegn72] Wegner, Peter, The Vienna Definition Language, Computing Surveys, vol. 4, no. 1, March 1972.