THE SKI MANUAL

by

Gray Clossman

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT No. 110

THE SKI MANUAL

GRAY CLOSSMAN

JULY, 1981

This material is based upon work supported by the National Science Foundation under Grant MCS80-09201.

page 1. Title page. Contents by section name.

The SKI Manual

by Gray Clossman

This report is divided into the following sections:

A introduction.

B contents and naming conventions.

C short description of the Scheme language and parallel processes.

D specification of the Ski language.

E definitions and pretty-printing them.

F editing a definition.

G tracing procedures.

H interactive help and error messages.

I classes and objects.

J editing a file.

K simulating parallel processing.

L the SKI compiler.

M the stack language interpreter.

N customized definition types.

O the SKI source files.

P glossary.

A. INTRODUCTION

SKI is an implementation of the programming language Scheme [Steele and Sussman 78] and is based directly on the IUSCHEME implementation by Mitchell Wand [Wand 80a]. The Ski (see note on capitalization conventions in section B, "CONTENTS AND NAMING CONVENTIONS") language includes, in addition to the kernel forms of Scheme, the data abstraction constructs CLASSES and OBJECTS and kernel forms for expressing parallel computations. The SKI run-time system is an interactive program and consists of an editor, a compiler, and a "microcode-like" interpreter which executes compiled Ski programs and simulates parallel execution of parallel computations as in [Wand 80b]. SKI runs in conjunction with UCI-LISP [Bobrow et al] and uses the ILISP s-expression oriented editor, the ILISP break package, and a file editor. SKI has grown out of our work on the artificial intelligence project SEEK-WHENCE (SW) [Hofstadter et al 79] for which we are implementing a program architecture that uses data-driven parallel processes as in the HEARSAY II program [REDDY et al 76]. This report is a manual and tutorial for the SKI implementation of Scheme. We assume that the reader is, at the very least, familiar with some implementation of Lisp. We strongly recommend that the reader keep handy for reference:

1) documentation of the Scheme language, for example, [Steele and Sussman 78],

2) some documentation of ILISP, for example, [Bobrow et al] or [Meehan 79], and

3) "Continuation-Based Multiprocessing", [Wand 80b].

Anyone who wants to modify SKI should have all of the above and also the manual [Wand 80a] for SKI's predecessor, IUSCHEME.

Type "RU SKI[1,GAC]" to run SKI at SAIL. This text is SKI.MAN[1,GAC].

overview of SKI

SKI runs on top of the UCI-LISP system. SKI was implemented on the version of ILISP documented in [Bobrow et al] on a DEC-10 at Indiana University under the TOPS-10 operating system and at Stanford Artificial Intelligence Lab under the WAITS operating system. Questions, observed bugs, suggestions, extensions, or improvements are invited. Correspondance to:

Gray Clossman
Computer Science Dept.
101 Lindley Hall,
Indiana University
Bloomington, Indiana 47401

SKI resembles LISP programming systems in many ways. A programmer interactively writes, tests, and edits a system of functions to compose a Ski program. All of the capabilities of the underlying ILISP system are available to a SKI programmer, and any ILISP program may be run from SKI. We have avoided redefining Ilisp primitives when implementing SKI. The underlying Ilisp system is undisturbed. We hope that any program that runs in ILISP will run in SKI.

SKI consists of:

1) a READ-EVAL-PRINT LOOP which reads input from a terminal or a file and prints values to a terminal or file.

2) a COMPILER. Scheme expressions are compiled into an assembly language for a virtual stack machine. This assembly language is list-structured and is called the STACK LANGUAGE.

3) a STACK LANGUAGE INTERPRETER which executes the stack language. The interpreter is the virtual stack machine and is written in Ilisp.

4) an S-EXPRESSION-ORIENTED EDITOR. SKI uses the ILISP STRUCTURE EDITOR [Bobrow et al] [Meehan 79] to edit Scheme programs and file definitions.

5) a FILE EDITOR. A file is represented in SKI by a FILE DEFINITION: a list of the names of functions, procedures, data, and comments on a file. You edit a file by editing its file definition.

The file editor and extensions to the ILISP editor are separable from SKI and are appropriate for use in ILISP systems. The source language for the editor extensions, a versatile pretty-printer, and the file editor is on the file FILHAN.IL [1,GAC] at SAIL.

Debugging facilities for Ski programs include:

1) COMPILE-TIME ERROR CHECKING.

2) the ILISP BREAK PACKAGE [Bobrow et al] [Meehan 79]. When an error condition occurs, the SKI system enters a BREAK, i.e., the computation is frozen and control is thrown to the break package. In a break, a programmer may examine variable bindings and function definitions, edit them, and continue the computation after corrections are made.

3) the functions ERR, ERROR, and ERRSET. These functions let the SKI programmer detect and generate errors in order to direct the flow of control in a Ski program.

4) the functions TRACE and UNTRACE, compatible with the ILISP trace package, for tracing Scheme procedures.

background

In the summer of 1980, we began the implementation of the SEEK-WHENCE project. The architecture of the SW program requires a variable number of processes that run in parallel, communicate both by messages to each other and through a global data structure, can execute arbitrary expressions indivisibly, and can be scheduled by one or more schedulers which are also processes. We had to choose a programming language in which to implement SW. The language had to be easy to write, debug, and extend as needed, and had to have support software (file handling, editing, etc.) for convenient programming. For many reasons -- familiarity, availability, expressive power, extensibility -- Lisp seemed right. UCI-LISP was familiar to us, and provides outstanding editing and debugging facilities. We were aware that Mitchell Wand at Indiana University had implemented a dialect of Scheme, called IUSCHEME, which included the data abstraction constructs CLASSES and OBJECTS (similar to those of SIMULA [Dahl et al 70] and Smalltalk [Shoch]). Wand had written an operating system in IUSCHEME using the Scheme primitive CATCH for process saving [Wand 80b]. IUSCHEME had many of the qualities we were looking for. It had data abstraction; it was written in Lisp1.6 and so was modifiable and extensible; it supported parallelism using timing interrupts to simulate parallel execution of processes; it implemented Scheme's tail recursion; and it ran acceptably fast -- half as fast as interpreted Lisp1.6. We decided that IUSCHEME looked promising for SW. SKI, which very roughly stands for "Scheme at Indiana University", took shape in the fall of 1980.

B. CONTENTS AND NAMING CONVENTIONS.

page
1. Title page. Contents by section name.
2. Section A. Overview. Major subprograms.
3. Debugging. Background.
4. Section B. Contents.
5. Capitalization conventions. Naming conventions in file editor.
6. Section C. Lexical vs. dynamic. Example contrasting Lisp and Scheme.
7. Values. Procedures are values. Closures. Evaluation of expressions.
8. Magic words. Tail recursion. Continuations.
9. Continuations used to simulate parallel processing.
10. Section D. Programs. Applicative order. Kernel forms. LAMBDA.
11. IF. QUOTE. LABELS. CLASS. Side effects. DEFINE.
12. ASETQ. Dynamic binding. FLUIDBIND. FLUID. STATIC.
13. LISP. CATCH. Syntactic extensions. BLOCK. LET. DO. ITERATE.
14. TEST. COND. OR. AND. AMAPCAR.
15. User-provided extensions. DSYNM. Primitive functions. PROCP. ENCLOSE.
16. Section E. Definition. Definer. DEFINE. SKI top level. Pretty-print.
17. PP. PPX. Defs. Vals. DEFSETQ.
18. Sample Ski program: the 3n+1 problem. TNPO.
19. Example of tail recursion. Sample run-time error.
20. Recompiling Ski procedures.
21. Section F. Editing. EDIT. EDITX. Sample editing session.
22. Sample editing session, continued.
23. Section G. Tracing. TRACE. UNTRACE. Sample traced procedure.
24. Sample traced procedure, continued.
25. Section H. HELP. ?. Error-handling. Compiler modes.
26. STREAMLINE. UNSTREAMLINE. SLEUTH. UNSLEUTH. Error messages.
27. Illegal opcode. Lookup error. Wrong number of arguments.
28. Compiler, Unbound, Lisp, Closure, and Message errors. In a break.
29. Exiting from a break. The read loop. The function SKI. Nested SKIs.
30. Section I. Class. Object. CLASS. Message. SELF.
31. More 3n+1. Data abstraction. Records with named fields.
32. Sample CLASS expression. Data protection. Calling objects.
33. Sending messages to objects. SEND. Top-down vs. bottom-up.
34. Abstract data structure. TNPO. Sample call to OR.
35. Sample program with classes and objects, continued.
36. Sample program with classes and objects, continued.
37. Sample program with classes and objects, continued.
38. Sample program with classes and objects, continued.
39. Sample program with classes and objects, continued.
40. Sample program with classes and objects, continued.
41. Sample program with classes and objects, continued.
42. Sample program with classes and objects, continued.
43. Section J. File editor. File names. Sample file definition.
44. What files can be edited. Reading files. GETFILE, GETFILEQ, GETDEFS.
45. GETDEFSQ. TY.
46. INVENTORY. CONTENTS. Writing files. FILE.
47. Create file. Modify file. Get definition for existing file.
48. Restore file definition for file. Sample of file editing.
49. EDIT. RENAMEFILE. DRENAMEFILE. DELETEFILE.
50. Section K. Simulating parallelism. RACE. Contestant. ENTER. EMPTYP. RUN.
51. Error in a contestant. DROPOUT. ENTER. #TIME. #ENABLED. PREEMPT.
52. Custom parallelism. Time slices. Sample race. TNPO.
53. Sample race, continued. Stream of values.
54. Section L. SKI compiler. "Compiling out" identifiers. Precedence.
55. Compile-time error-checking. NUMBER-OF-ARGS. Declare subr or lsubr.
56. No forward declaration problem. Definition time. Compile time.
57. Section M. Flags. #ENABLED. #TIME. #STATS.
58. Section N. Custom definitions. DEFINERS. GRINPROPS. -PPRINT.
59. -FORMAT, -UNFORMAT. Formatting for pretty-printing.
60. Section O. Compile source files. Space allocation. Source code names.
61. Section P. Glossary. "break" through "expr".
62. Glossary, continued. "file definition" through "SKI".
63. Glossary, continued. "Ski" through "VAL".
64. References.

capitalization conventions

This report describes the integration of two programs -- Mitchell Wand's implementation of Scheme and the UCI-LISP implementation of Lisp. It talks about several very similar languages -- Scheme, Ski, Lisp1.6, and Ilisp. The naming conventions used in this report will often be important. The name of a program or system will usually be written in all capitals (e.g., ILISP, SKI), while the name of a language supported by or interpreted by a program will be written with its first letter capitalized (e.g., Ilisp, Ski). Names of individual Ilisp or Scheme functions and variables will be printed uppercase (e.g., COND, LABELS, INVENTORY). Definition types -- named by the property names (e.g., EXPR, MACRO, DEF) under which they appear on the Ilisp property lists of identifiers -- will be printed lowercase (e.g., expr, macro, def). Section P of this manual is a glossary of terms. The naming conventions used for functions and variable names in the SKI source files are explained in section O, "THE SKI SOURCE FILES".

naming conventions in the file editor

Most of the functions implementing the file editor come in pairs of a macro and a subr (e.g., INVENTORY - INVENTORYX, GETFILE - GETFILELX). The macro of the pair usually provides the more convenient way to call the function from the top level. It typically will not require its arguments to be quoted, and may accept a variable number of arguments. (But see individual function specifications in section J, "EDITING A FILE".) The subrs of the pairs do evaluate their arguments (as subrs always do), and if you program your own extensions to the file editor, feel free to use them as subroutines. The subr of each pair is named by suffixing "X" or "LX" to the name of the macro. In general, the suffix "X" indicates a subr that takes single file names or variable names as arguments. For example, RENAMEFILEX takes two arguments, the names of two files. But the suffix "LX" indicates a subr that takes at least one list of things as an argument. For example, GETFILELX takes one argument, a list of file names.

Some global Ilisp variables used by the file editor are:

GRINPROPS — the list of property names of definition types that may be pretty-printed, edited, and stored on files.

DEFINERS — the list of definers -- functions that store definitions on property lists of identifiers -- and the property names that correspond to them.

TRACEDFNS — the list of names of functions that are currently traced.

SCREENSIZE — the number of printed lines that fit on the user's terminal screen. See explanation of TY in section J, "EDITING A FILE".

ERRVALEXP
ERRVALFLAG — special variables used by the file editor.

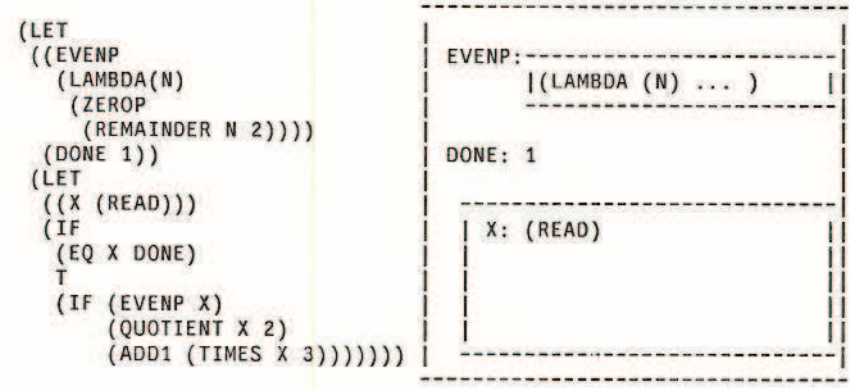
C. SHORT DESCRIPTION OF THE SCHEME LANGUAGE AND PARALLEL PROCESSES

Scheme is a lexically scoped dialect of Lisp. Scheme, Algol, and Pascal are all examples of lexically scoped languages. Each is a block-structured language where identifiers declared within a block are local to that block, i.e., references to an identifier are valid only in expressions written in the block in which that identifier is declared. Scheme uses lambda binding to declare identifiers. So the scope of an identifier -- the parts of a program in which references to that identifier have meaning -- can be determined by examining the source-language text of the program. The identifier's scope does not depend on the program's run-time flow of control. Flow of control in a given program will change from one execution of the program to the next because it is data-dependent, but the source-language text of that program will not change. Roughly speaking, then, a lexically scoped language can be compiled. Based solely on the source text of a Scheme program, a compiler can determine whether an identifier reference is valid, and if so, exactly where the value associated with that identifier will be at run time. Since this determination can be made on the basis of a static form of a program -- its source text -- "statically scoped" is synonymous with "lexically scoped". Identifiers in a lexically scoped language are said to be "statically bound" and are called "static identifiers" or "static variables". Identifiers in a dynamically scoped language are said to be "fluidly bound" and are called "fluid identifiers" or "fluid variables". SKI provides both static and fluid environments, but in Scheme programs, any identifier that is not marked "fluid", e.g., (FLUID X), is treated as a static identifier. In the following example Scheme program, the Scheme magic word "LET" is an alternate syntax for lambda binding. The LET expression here binds the variables EVENP and DONE. In general:

"(LET ((<var1> <exp1>) ... (<varn> <expn>)) <body>)"

is equivalent to

"((LAMBDA (<var1> ... <varn>) <body>) <exp1> ... <expn>)".



The Scheme program above reads a number, x. The program returns the value T if x is equal to DONE; if x is even, it returns x divided by two; but if x is odd, it multiplies x by three and adds one. The program contains three blocks: the outermost block where the identifiers EVENP and DONE are bound, a block in which N is bound, and a block in which X is bound. The identifiers EVENP and DONE are meaningful everywhere in the program, because the block in which they are bound contains every other block in the program. Note the uses of EVENP and DONE in the bottom inner block. The identifier N is not within the lexical scope of the bottom LET expression and so cannot be referenced in the bottom inner block. Similarly, X cannot be mentioned in the top inner block. The following equivalent Lisp program DOES refer to X in its function EVENP.

```
((LAMBDA(EVENP DONE)
  ((LAMBDA(X)
    (COND ((EQ X DONE) T)
          ((EVENP) (QUOTIENT X 2))
          (T (ADD1 (TIMES X 3)))))
   (READ)))
 (QUOTE (LAMBDA NIL (ZEROP (REMAINDER X 2))))
 1))
```

The reference to X in the Lisp program depends on the run-time flow of control of the program. The Lisp function EVENP here MUST be called from some other expression in which X is bound to a number or an error will result. The validity of the reference to X in the Lisp program cannot be determined from the text of the program. Because of this, there are some properties of the program that cannot be proved, and a compiler for Lisp cannot replace the identifier X in this Lisp expression by a specification of the location where its value will be found at run time. References of this type are illegal in Scheme. The SKI compiler replaces each identifier in a Scheme program by a pair of small integers that code for the location of its value at run time.

A Scheme identifier may have as its value EITHER a datum (constant, atom, or list) OR a procedure. (PROCEDURE is a data type in Scheme, like ATOM, LIST, NUMBER, etc. The predicate PROCP returns T if its argument has a procedural value, just as NUMBERP returns T if its argument has a numeric value. Procedures may be passed as arguments to other procedures and returned as the values of other procedures.)

A VALUE is either a
DATUM, which is a constant, an atom, a list, or a string;
or a
PROCEDURE, which is a closure, a continuation, or an object.

In Scheme, a procedure is always the VALUE of some identifier or expression. This contrasts with ILISP and most LISP systems, where an identifier may simultaneously have a value and one or more functional properties. Scheme procedures are closed functions (called "closures" -- closures, objects, and continuations are all closed functions in Ski and may all be called "closures".) Because all functions are closed, there is no funarg problem in Scheme as there is with Lisp. All Scheme functions are automatically full upward and downward funargs. Procedures in Ski, including primitives (e.g., CAR, CDR, CONS), user-defined Scheme procedures, and Lisp functions (any Ilisp function may be called from SKI), are the values of variables. When one wants to know the value of a variable in SKI, one types that variable to the SKI top-level read-eval-print loop. SKI then prints on the terminal screen the value of the variable, if it has one. So if one types "CAR", then the value of the variable CAR -- the procedural value that computes the function CAR -- is printed on the screen.

What does a procedural value -- a closure -- look like? By definition in [Steele and Sussman 78], the run-time representation of a procedure (a closure) is implementation-dependent, while Scheme programs (which, of course, use procedures) are implementation-independent. For example, one way to generate a closure is to evaluate a lambda expression. The syntax for lambda expressions is part of the Scheme language; that syntax is implementation-independent. But a procedural value -- a closure -- generated at run time from a lambda expression, has an implementation- dependent representation. Therefore, PROCEDURE is a protected data type; procedural values should not be explored or decomposed with, say, CAR or CDR or PRINT. When a procedural value is printed on the terminal screen, it is shown as the atom "**CLOSURE**" (or "**OBJECT**" or "**CONTINUATION**"). If you type "CAR" to the top-level, it prints "**CLOSURE**", which is all you get to see of the representation of the closure. But if you type "(CAR (QUOTE (A B C)))", it works: it returns "A".

Like Lisp programs, Scheme programs are composed entirely of expressions. (Expressions, unlike GOTOs and DO loops in FORTRAN for example, return values when executed.) All expression are in prefix functional notation, and the CAR of an expression is called the "function" of that expression and is said to be in the "function position". When a Scheme expression is evaluated, each element of the

expression is evaluated once; the element in the function position must evaluate to a procedural value. The other elements of the expression are the arguments to this procedure and should evaluate to appropriate values. For example, in the expression, "(F (ADD1 X) (CAR Y) (GRAPPLE Z))", the identifier "F" must evaluate to a closure that is a functions of three arguments, and "(ADD1 X)", "(CAR Y)", and "(GRAPPLE Z)" are the arguments to that function.

The only Scheme expressions whose functions do not necessarily evaluate to closures are those with Scheme "magic words" in their function positions. Magic words are reserved words (key words). The SKI compiler has special routines to generate stack language for magic words. Magic words are used where one would typically use fexprs in Lisp, e.g., IF, ASETQ, and BLOCK, which correspond roughly to the Ilisp functions COND, SETQ, and PROGN. A SKI user may define additional magic words.

In many implementations of Lisp, the environment in which a function is called is always retained until the conclusion of the evaluation of that function call. This is because it is very difficult to analyze and anticipate the flow of control and, in particular, the fluid environment, in a dynamically scoped language like Lisp. However, since Scheme is a lexically scoped language, it is possible for a compiler for Scheme to determine when environments will and will not be needed at run time to complete the evaluation of any pending function calls. The data structures containing unneeded environments can be garbage-collected at the earliest possible moment. Because of this, iterative algorithms may be written in Scheme as recursive procedures without risk of run-time recursion stack overflows. Such procedures are called "tail-recursive" procedures and are useful for algorithms for which one would use a DO loop in Fortran, or a WHILE or FOR statement in Algol or Pascal.

A Scheme procedural value -- a closure -- carries with it ALL of the information (non-local variable bindings) necessary to compute a result EXCEPT for its actual arguments. A closure is represented in the SKI implementation as a 3-tuple:

1) the reserved flag-word "**CLOSURE**" (or "**OBJECT**" or "**CONTINUATION**" depending on the type of closure)

2) the body of the procedure

3) the static environment in which values of identifiers (including references to other functions, since they, too, are the values of identifiers) are to be found.

There are several kinds of expressions that evaluate to closures at run time:

1) lambda expressions
2) primitive function names (e.g., ADD1, CAR, CONS)
3) user-defined Scheme procedure names
4) Lisp function names
5) CATCH variables (which evaluate to continuations)
6) CLASS expressions (which evaluate to objects)

Any of these expressions evaluates to a Scheme procedural value and may be written in the function position (CAR position) of a Scheme expression. Arbitrary Scheme expressions that return as their values closures that are ultimately generated by one of the above types of expressions may be used as functions. Magic words are also written in the function positions of expressions. No other type of expression may be used as a function in a Scheme expression.

One way that closures can be created is with CATCH expressions. A catch expression has the syntax: (CATCH <var> <body>), where <var> is a variable, and <body> is an arbitrary Scheme expression. Within the lexical scope of a CATCH expression, <var> is automatically bound to a special kind of closure called a CONTINUATION or CONTINUATION OBJECT. A continuation is a procedural value and may be saved and invoked later. A continuation is usually saved by assigning it to some variable. A continuation may be thought of as a partly evaluated expression which may be restarted to resume evaluation at a later time. So, one may think of

a continuation as a process that has been suspended and is "asleep", but which may be "awakened" later to continue its computation. With continuations, then, one can simulate parallel processing. Processes, represented as Scheme procedures, may be packaged into continuations, and these continuations may be run little by little, one after another, round and round. The time slice that each process gets (the amount of computing time a process is allowed to run before it is put back to sleep) may be set initially by the SKI user and changed dynamically during a computation if desired.

D. SPECIFICATION OF THE SKI LANGUAGE

The material on the following seven pages is freely adapted from "The Revised Report on SCHEME, A Dialect of Lisp", [Steele and Sussman 78]. We have modified the excerpts here to be stylistically consistent with the rest of this report. Explanations of differences between Steele and Sussman's SCHEME and SKI have been inserted into the text.

A. THE REPRESENTATION OF SCHEME PROCEDURES AS S-EXPRESSIONS

SCHEME programs are represented as LISP s-expressions. The evaluator interprets these s-expressions in a specified way. This specification constitutes the definition of the language.

Atoms that are not atomic symbols (identifiers) evaluate to themselves. Typical examples of such atoms are numbers and strings (character arrays). Symbols are treated as identifiers (variables). They may be lexically bound by lambda expressions. There is a global environment containing values for (some) free variables. Many of the variables in this global environment initially have as their values primitive operations such as, for example, CAR, CONS, and *PLUS. SCHEME differs from most LISP systems in that the atom CAR is not itself an operation (in the sense of being an invocable object, e.g., a valid first argument to APPLY), but only has one as a value when considered as an identifier.

Non-atomic forms are divided by the evaluator into two classes: combinations and "magic forms". Magic forms are recognized by the presence of a "magic word" (a reserved word) in the CAR position of the form. All non-atomic forms that are not magic forms are considered to be combinations. The system has a small initial set of magic words; there is also a mechanism for creating new ones.

When a combination is to be evaluated, it is considered to be a list of subforms. These subforms are all evaluated. The first value must be a procedure; it is applied to the other values to get the value of the combination. There are four important points here:

(1) The procedure position (also called the "function position") is always evaluated just like any other position. (This is why the primitive operators are the VALUES of global identifiers.)

(2) The procedure is never "re-evaluated"; if the first subform fails to evaluate to an applicable procedure, it is an error. Thus, unlike most LISP systems, SCHEME always evaluates the first subform of a combination exactly once.

(3) The argument forms (and procedure form) may in principle be evaluated in any order. This is unlike the usual LISP left-to-right order.

(4) The arguments are all completely evaluated before the procedure is applied; that is, Scheme, like much of Lisp, is an applicative-order language.

B. CATALOGUE OF MAGIC FORMS

B.1 KERNEL MAGIC FORMS

The magic forms in this section are all part of the kernel of Scheme, and so must exist in any SCHEME implementation.

(LAMBDA (<identifier list>) <body>)

Lambda expressions evaluate to procedures (to closures). Unlike most LISP systems, SCHEME does not consider a lambda expression (an s-expression whose CAR is the atom LAMBDA) to be a procedure. A lambda expression only EVALUATES to a procedure. A lambda expression should be thought of as a partial DESCRIPTION of a procedure; a procedure and a description of it are conceptually distinct objects. A lambda expression must be closed (associated with an environment) to produce a procedure. Evaluation of a lambda expression performs such a closure operation.