TECHNICAL REPORT NO. 149

C – SCHEME REFERENCE MANUAL

by

R. Kent Dybvig

September 1983

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

C-SCHEME REFERENCE MANUAL

R. Kent Dybvig

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405

TECHNICAL REPORT NO. 149

C-SCHEME REFERENCE MANUAL

by

R. Kent Dybvig

September, 1983

Research reported herein was supported in part by the National Science Foundation under grant numbers MCS 79-04183 and MCS 83-04567.

C-Scheme Reference Manual

Abstract

This manual describes a new version of Scheme called *C*-Scheme. Like Scheme, *C*-Scheme is lexically scoped, supports full function closures and continuations, and is implemented in such a way that tail-recursions perform iteration. *C*-Scheme *curries* all function definitions and applications, providing a mechanism for partial application. *C*-Scheme provides a *timer interrupt* facility which can be used with continuations to implement multitasking.

The *C*-Scheme implementation consists of a powerful preprocessor, a parser and an interpreter for the parser output. It is intended to be usable as the core of a production-quality Scheme system.

C-Scheme is coded in the *C* programming language and runs on a Vax 11/780 minicomputer under the UNIX† timesharing system.

---

† **UNIX is a Trademark of Bell Laboratories.**

CONTENTS

Preface

This thesis describes *C*-Scheme, a new implementation of Scheme for the Vax11-780 computer running 4.1 bsd Unix. It is intended to serve as the reference manual for *C*-Scheme.

C-Scheme is a programming system with a complete lexical scanner for input expressions; a macro preprocessor, parser and evaluator for these expressions; an output facility for *C*-Scheme data; and an allocator for dynamic storage allocation. *C*-Scheme features full continuations, lexical scoping, full function closures, *curried* function definition, and timer interrupts.

Scheme was originally written as a dialect of Lisp, and retains the syntax and many of the list processing functions of Lisp. Like Lisp, Scheme's primary program construct is function application. Lisp and Scheme both support automatic allocation and deallocation of data structures. Programs in both languages are represented with data structures supported by the languages, making programming tools such as debuggers and program editors easy to implement. Scheme is smaller, cleaner and easier to learn than most of the current Lisp dialects, making it an excellent tool for teaching and research. Scheme is lexically scoped and supports full function closures (that is, functions can be passed to or returned from other functions). Lisp is usually dynamically scoped and usually does not support full function closures. Also, Scheme supports *continuations*, a general non-local exit mechanism. Full continuations allow jumps back into code which may have already finished execution, and make possible the implementation of coroutines and other interesting control structures.

Because Scheme is so similar to Lisp, anyone not already familiar with Lisp will find it useful to read *The Little Lisper* [Friedman 1974] or the *Lisp 1.5 Primer* [Weissman 1968].

Scheme is similar in many respects to Pascal and other block structured languages, because of lexical scoping and Scheme's block structuring macros *let* and *letrec*. However, Pascal programs tend to use less functions and more assignment statements. Also, Pascal compilers support allocation of data objects but usually only support a primitive deallocation mechanism.

C-Scheme has several features not found in other Scheme systems. The most important feature is complete *currying* of function definitions and applications. *Currying* allows the programmer to view any function as taking its arguments one at a time. Conversely, any function of one argument which returns a function may be viewed as a function of more than one argument. It is often useful to create intermediate functions by applying a general function to less arguments than it expects. One nice result of *currying* is that the function *apply*, which applies a function to a list of arguments, may be written in *C*-Scheme without the use of *eval*.

Another feature of *C*-Scheme not found in most other Scheme systems is a *timer interrupt* facility. A single function, appropriately called *timer*, controls an internal clock. The user may program the timer for an arbitrary time interval. When the timer expires, the user-specified interrupt routine is invoked. The interrupt routine may use continuations to save the current computation and to start another; this provides enough mechanism to write a multitasking scheduler.

C-Scheme also allows the keywords for syntactic extensions (macros), special forms and functions to overlap; the syntactic extension is performed first. The special form *lambda* is extended to allow a name which is visible only within the body of the *lambda*. When the closure for the *lambda* expression is built the name will be bound to the closure. This permits self-contained and concise recursive function definitions. Unlike some recent implementations of Scheme, *C*-Scheme supports full continuations.

C-Scheme has a preprocessor which condenses the code into a small *kernel* language. The preprocessor expands syntactic extensions and curries function definitions and applications. The output of the preprocessor is called the *kernel language*. A parser further reduces the language into threaded code. The output code actually contains pointers to evaluation functions so that very little work has to be done by the interpreter at run time.

C-Scheme has a small set of special forms, only *quote*, *lambda*, *if*, *prog2* and *change!*. *Prog2* actually generates no code and thus incurs no overhead at run time. Most of the language constructs are implemented with syntactic extensions or functions.

C-Scheme's allocator/collector was designed with recent technology and allows objects of arbitrary size to be created. Vectors and strings can be handled more efficiently than in systems capable of allocating only fixed-sized blocks.

Certain types of function applications are optimized. Most notably, a function application with a lambda expression in the function position is in-line coded and incurs only the overhead of adding values to the environment. A closure is not created for the lambda expression. This optimization is important since the frequently used macros *let*, *let** and *letrec* translate into this type of application.

This thesis gives many example *C*-Scheme functions, including:

- a *queue* function showing how data may be abstracted with *C*-Scheme;
- a *stack* function which is a module with local data and several entries;
- four different factorial functions which highlight currying and tail recursion;
- a stream constructor macro which implements lazy cons;
- an elementary scheduler which shows how continuations and timer interrupts may be used to implement multitasking.

This work has been supported by members of the computer science departments of both Indiana University and The University of North Carolina at Chapel Hill. In particular, I would like to acknowledge the guidance and support of Dan Friedman and Mitch Wand. I would also like to thank Rick Snodgrass and Bruce Smith for suggestions and comments on the final drafts. Also, I appreciate the patience of Don Stanat and Gyula Magó and the use of UNC resources for the completion of my thesis. Special thanks go to my wife, Sue, for proofreading each draft and helping me battle the text formatter.

1. Introduction

This is the reference manual for a new version of Scheme called *C*-Scheme. The *C*-Scheme system is not built on top of an existing Lisp system as many previous Scheme systems have been. Rather, it is intended to be the core of a production-quality system.

C-Scheme is coded in the C programming language and runs on a Vax 11/780 minicomputer under the UNIX† (4.1 BSD) timesharing system.

Section 2 of this manual gives the information necessary to begin using *C*-Scheme, including a sample session. Section 3 gives some examples of *C*-Scheme programming. Section 4 describes the data objects available to the *C*-Scheme programmer. *C*-Scheme identifiers are described in section 5. Control structures such as function definition, conditionals and sequencing are given in section 6. Sections 7 through 13 define *C*-Scheme functions for manipulating data. Section 14 describes the reader. The implementation is discussed in sections 15, 16 and 17. References and an index of forms conclude the manual in sections 18 and 19. The remainder of this introduction gives a brief outline of the features of *C*-Scheme.

1.1 *C*-Scheme

C-Scheme is an applicative-order, lexically scoped programming language based on Alonzo Church's lambda calculus [Church 1941]. Scheme was introduced by Guy Steele and Gerald Sussman [Sussman & Steele 1975], [Steele & Sussman 1978]. Steele and Sussman call it a dialect of Lisp, since it uses Lisp syntax, data structures, and many Lisp functions. However, the language framework more closely resembles Algol, with lexically scoped identifiers and block structure. Lisp syntax aids the development of programming tools since programs are represented as data structures in the language. For example, *C*-Scheme has a macro-preprocessor which is itself written in *C*-Scheme.

C-Scheme's primary construct is function application. Unlike Lisp and Algol, *C*-Scheme supports full function *closures*. A function is *closed* with the lexical *environment* it is created in so

† UNIX is a Trademark of Bell Laboratories.

that identifier bindings are passed along with the function. The resulting *closure* can be used as an argument to other functions or returned as a value.

C-Scheme is implemented in such a way that a function call from the tail of an expression behaves as if it were a jump. This means that tail-recursive functions execute without net growth of the interpreter stack and thus may be used to perform iteration. Although most compilers perform this optimization, interpreters usually do not.

One of the nicest features of *C*-Scheme, as with other Schemes, is the ability to request the *continuation* of any expression being evaluated. Intuitively, a continuation is the total information needed to complete the execution of the expression from a given point. In this implementation, the continuation consists of the control stack and current lexical bindings. Once requested, the continuation can be invoked just like a normal function closure. Invocation of a continuation has the effect of returning the computation to the point where the continuation was requested. One obvious use of this mechanism is for non-local exits or error trapping. However, it is more powerful than a simple non-local GOTO, since the state can be reinstated even after the computation has completed. As long as the continuation exists, its control stack and environment remain intact, and the computation can be restarted any number of times. This can be used for unusual looping constructs, coroutines and backtracking searches.

Unlike other Scheme systems, *C*-Scheme supports *currying* of function parameters [Rosser 1982], [Stoy 1977]. Any function of n arguments is represented as a function of 1 argument whose value will be a function of n-1 arguments ($n > 1$). Because of this, function arguments may be passed in one at a time, yielding interesting intermediate functions. (For example, if + normally takes two integers and returns their sum, applying + to only one integer, say 10, would produce a new function which adds *10* to its argument.)

Timer interrupts are another feature of *C*-Scheme not included in most Scheme systems. The timer can be enabled by using an enable function with two arguments: the number of discrete intervals (called *ticks*) to wait and an interrupt service routine. When the specified number of ticks has elapsed, control is passed to the interrupt service routine. If the routine returns without doing anything, control proceeds as if the interrupt did not happen. Multitasking may be performed with this timer using continuations, as shown by Wand [Wand 1980].

C-Scheme does not support the fluid binding of [Steele & Sussman 1975].

Several other Scheme systems are currently in existence. The ⟦Scheme 311⟧ Reference Manual [Fessenden, et.al. 1983] describes a Scheme system used for several computer science courses and for research at Indiana University. Computer Science students at MIT use Scheme for one of their first programming courses. A somewhat different flavor of Scheme intended for production use, T, is being developed at Yale [Rees & Adams 1982].

1.2 Syntax

The BNF grammar below gives the syntax for *C*-Scheme expressions. Expressions followed by an asterisk may be repeated zero or more times.

```
<exp>                        ::=    <constant>
                                  | <identifier>
                                  | <syntactic extension>
                                  | <special form>
                                  | <combination>
<constant>                   ::=    <integer> | nil | <string> | <vector>
<identifier>                 ::=    <symbol>
<syntactic extension>        ::=    (<macro keyword> <object>*)
<macro keyword>              ::=    <symbol>
<special form>               ::=    (quote <object>)
                                  | (lambda (<symbol>*) <exp>)
                                  | (if <exp> <exp> <exp>)
                                  | (prog2 <exp> <exp>)
                                  | (change! <symbol> <exp>)
<combination>                ::=    (<exp> <exp>*)
```

Nil is a special null pointer, and is used to mark the ends of lists and represents the boolean value false in conditional expressions. <object> is any *C*-Scheme data type, e.g. symbols, numbers, strings and lists. The data types are discussed in §4. Macro keywords are symbols with the ***macro*** property on their property list.

C-Scheme identifiers are not automatically changed to either upper or lower case. This means that two identifiers which differ only in case are different identifiers. Currently, all identifiers recognized by the *C*-Scheme interpreter are lower case. Global identifiers (except function symbols) and property list keywords (see §4) are normally enclosed in two pairs of asterisks, as with ***macro*** above, to prevent clash with user identifiers. Special forms, macros and functions which cause side-effects are normally followed by an exclamation point, as with *change!*.

Expressions written as lists (special forms, syntactic extensions and combinations) are referred to as *forms*. The BNF grammar for forms is actually ambiguous. It is possible for some forms to be parsed as more than one kind of form. Syntactic extensions are checked for first, followed by special forms. If the expression is neither, it is assumed to be a combination.

Special forms make up the core of *C*-Scheme. There are only five special forms: *quote*, *lambda*, *if*, *prog2* and *change!*. Programs use *quote* to introduce literal data, *Lambda* to define functions, *if* for conditional execution, *prog2* for sequencing execution, and *change!* to change identifier bindings.

Syntactic extensions are created by defining symbols as preprocessor macros. Macros can improve the readability of code by abbreviating commonly used structures. Most of the control structures described in this manual are implemented as macros. Macros are written entirely in *C*-Scheme, and are expanded by the *C*-Scheme-coded preprocessor (see §17).

Combinations, or function applications, are the most common forms in *C*-Scheme programs. A combination is a form consisting of a function expression and zero or more argument expressions. *C*-Scheme executes a combination by first executing the function and argument expressions, then applying the function to the arguments. The expressions may be executed in any order, so *C*-Scheme programs should not depend upon the order of evaluation.

Combinations are automatically *curried* before evaluation begins. Applying a function to zero arguments is the same as applying it to *nil*. The application of a function to more than one argument is equivalent to applying it to one at a time (associating to the left). Function definitions (using *lambda* -- see §6) are transformed similarly. In each of the following examples the left form is equivalent to the right.

```
(f)                          (f nil)
(f x)                        (f x)
(f x y)                      ((f x) y)
(f x y z)                    (((f x) y) z)

(lambda () body)             (lambda nil body)
(lambda (x) body)            (lambda x body)
(lambda (x y) body)          (lambda x (lambda y body))
```

The functions included in the basic *C*-Scheme system are either *primitive* (coded in C), or *library* (coded in *C*-Scheme). Both print as ** *closure* **. It should not matter to any computation whether a function is *primitive* or *library*. Either type of function may be invoked directly, or passed as data to be invoked later. This is not true of the keywords to special forms or syntactic extensions. These really are syntax and only make sense when written directly. Attempting to pass a macro or special form keyword will cause an error unless the keyword has a function definition as well.

1.3 Evaluation of *C*-Scheme Expressions

Execution of a *C*-Scheme expression takes three steps: preprocessing, parsing and evaluating. The preprocessor expands macros, curries functions, and attempts to evaluate time-independent expressions (expressions which evaluate to a constant in any context). The parser resolves local variable references and compiles special forms and combinations into a more compact, partially threaded stack machine language. Finally, the evaluator is a virtual stack machine for this language.

The parser and evaluator together form the interpreter *kernel* and are coded entirely in C. The preprocessor is coded in *C*-Scheme.

2. Using *C*-Scheme

C-Scheme is an interactive system. To start *C*-Scheme simply type "scheme". The *C*-Scheme toplevel will print an asterisk (*) on the screen to let you know it is ready for input. When you have completed a *C*-Scheme session, invoke the function *exit*. *Exit* takes no arguments.

2.1 *C*-Scheme Toplevel

The user interface to *C*-Scheme is a top level *read-execute-print* loop. It loops forever, reading an expression, evaluating the expression and printing the result.

The looping is actually performed by the *C*-Scheme kernel, and the *read-execute-print* is performed by a *C*-Scheme-coded function. The *C*-Scheme kernel assumes that the global value of the identifier ***toplevel*** is a function of zero or one arguments. The kernel loop applies ***toplevel*** to *nil* and throws away the result.

A default ***toplevel*** is provided. It is reasonable to change ***toplevel*** to customize it to personal taste or needs, but care must be taken. Once **toplevel** is trashed, there is no recovery!

The default ***toplevel*** prompts the user with an asterisk (*), reads one expression, prints a newline, executes the expression and prints the result with another newline. If all goes well (no errors), the global value of the identifier ***last-input*** is set to the input expression, and ***last-output*** is set to the result of the execution. ***last-input*** and ***last-output*** may be used in the next input expression, usually in the event that the user forgot to save them. In the following short sample session, the percent sign (%) is the operating system prompt, the asterisk (*) is the *C*-Scheme prompt and the user input appears only on lines with a prompt:

```
% scheme
Scheme version 1.0

*(plus 2 3 4)

9
*(times **last-output** 10)

90
*(lambda (x) (times x x))

** closure **
*(define square **last-output**)

square
*(square 3)

9
*(exit)
%
```

2.2 Errors and Interrupting Computation

C-Scheme does not yet have a break package. When an error occurs, a descriptive message is printed to the standard error file and control returns to the toplevel.

If a *C*-Scheme program appears to be in an infinite computation, the *break* key may be hit. This causes the message "interrupted" to print on the standard error file and control to return to the top level. (The interrupt may be caught by a user-defined keyboard interrupt service routine. See §6.)

2.3 Loading *C*-Scheme Code

C-Scheme functions can be placed on a file and loaded into the *C*-Scheme system with the *load* function. *Load* takes a single string argument which must be the pathname of an existing file, e.g. "*lib/scheme/myfuns.s*". Each *C*-Scheme expression in the file is executed. *Loads* may be nested, of course.

3. Sample *C*-Scheme Functions

This section gives a few simple *C*-Scheme functions, highlighting some of *C*-Scheme's features. More examples are spread throughout the text. These examples assume familiarity with Scheme or Lisp. Refer to later sections of this manual for the definition of any unfamiliar construct or function.

3.1 Four Factorial Functions

The factorial function is used to illustrated various *C*-Scheme programming styles.

```
(define fact-1
    (lambda (x)
        (if (0? x) 1 (times x (fact-1 (1- x))))))

(define fact-2
    (lambda f (x)
        (if (0? x) 1 (times x (f (1- x))))))

(define fact-3
    (lambda f (a x)
        (if (0? x) a (f (times a x) (1- x)))))

(define fact-4
    ((lambda f (a x)
         (if (0? x) a (f (times a x) (1- x))))
        1))
```

Fact-1 is the basic recursive factorial function. It is a function of one argument. If the argument is *0* the value is *1*. Otherwise the value is the argument times the value of calling *fact-1* with the argument minus *1*.

Fact-2 is only slightly different from *fact-1*. It is a also a function of one argument with essentially the same definition. The only difference is that it does not rely on the value of the symbol it is bound to remaining constant; it is self-contained. If the function is moved to another identifier it will still work even if the original identifier's value changes.

Fact-3 is a tail recursive version. This can be useful since tail-recursive functions behave like loops and will not cause the interpreter's control stack to grow. Many recursive functions can be written with tail-recursion. However, it is not always desirable to do so, since the tail-recursive version is usually not as clear and the recursion may not go deep enough to matter.

Fact-3 uses an accumulator *a* to collect the value, multiplying *a* by *x* each time through the loop. When *x* reaches *0*, *a* is returned. *Fact-3* must be called with *1* as its first argument in order to initialize the accumulator properly. This is normally done with the use of another function which other routines call.

Fact-4 is similar to *fact-3* except that it takes advantage of *currying*. Before the function is ever bound to the identifier *fact-4* it is applied to a single argument, *1*, initializing the accumulator.

Currying is used for *partial application* of functions to arguments. Dwyer and Dybvig suggest a more general form, *bind*, which allows any of the arguments to be bound, not just the first [Dwyer & Dybvig 1981]. Georgeff describes many of the advantages of partial application and argues that the implementation need not be less efficient than a system without partial application [Georgeff 1982].

```
(fact-1 3)                           =>   6
(define fact-1-save fact-1)          =>   fact-1-save
(define fact-1 (lambda (x) 0))       =>   fact-1
(fact-1-save 3)                      =>   0

(fact-2 3)                           =>   6
(define fact-2-save fact-2)          =>   fact-2-save
(define fact-2 (lambda (x) 0))       =>   fact-2
(fact-2-save 3)                      =>   6

(fact-3 1 10)                        =>   3628800
(fact-3 2 10)                        =>   7257600

(fact-4 10)                          =>   3628800
```

3.2 Queue: An Abstract Data Type

Since *C*-Scheme is lexically scoped, abstract data types may be constructed without any additions to the language. The following is an example of a *queue* data type, with operations *type, empty, put and get*. No one outside the queue object can access the queue's data or change the functionality of the operations on queues.

```
(define queue
   (lambda ()
      (let
         ((head nil) (tail nil))
         (lambda (request)
             (case request
                 (type 'queue)
                 (empty (null head))
                 (put
                    (lambda (v)
                       (progn
                          (if (null head)
                             (change! head (change! tail (cons v nil)))
                             (change! tail
                                (cdr (rplacd tail (cons v nil)))))
                          v)))
                 (get
                    (if (null head)
                       (error "queue get: queue is empty" nil)
                       (let ((v (car head)))
                          (progn
                             (if (null (change! head (cdr head)))
                                (change! tail nil))
                             v))))
                 (otherwise (error "queue: invalid request" request))))
      )
   )
)
```

```
(define q (queue))              =>   q
(q 'type)                       =>   queue
(q 'empty)                      =>   t
(q 'put 3)                      =>   3
(q 'empty)                      =>   nil
(q 'get)                        =>   3
(define putq (q 'put))          =>   putq
(putq '(a b c))                 =>   (a b c)
(putq 8)                        =>   8
(putq "hi")                     =>   "hi"
(q 'get)                        =>   (a b c)
(q 'get)                        =>   8
(q 'get)                        =>   "hi"
(q 'empty)                      =>   t
```

3.3 Suspending Cons

Friedman and Wise have suggested that *cons* should not evaluate its arguments [Friedman & Wise 1976], [Wise 1982]. That is, the *car* and *cdr* fields are *suspended* when *cons* is invoked. Evaluation occurs only when the *car* or *cdr* is accessed. This example gives a version of *cons*, called *scons* (for *stream cons*) which suspends its second argument (the *cdr*). This allows infinite structures to be described with *C*-Scheme.

Scons is written as a macro. It leaves the first argument untouched and suspends its second, using the *freeze* macro to package the argument as a function of zero arguments. *(freeze x)* is equivalent to *(lambda () x)*.

A new *cdr* function, *scdr*, is needed which checks to see if the end of the list is a closure. If it is, it uses *thaw* to invoke the closure, replaces the *cdr* of the list with the value and returns the value.

```
(macro scons
    (lambda (m)
        '(cons ,(cadr m) (freeze ,(caddr m)))))

(define scdr
    (lambda (x)
        (progn (if (procp (cdr x)) (rplacd! x (thaw (cdr x))))
            (cdr x))))

(scons 3 4)                    =>    (3 . ** closure **)
(cdr (scons 3 4))              =>    ** closure **
(scdr (scons 3 4))             =>    4
```

The function *stream-access* takes a stream *s* and an integer *n* and returns the *n*th element of *s*.

```
(define stream-access
    (lambda (s n)
        (if (0? n) (car s) (stream-access (scdr s) (1- n)))))
```

Here *scons* is used to create a stream of Fibonnaci numbers:

```
(define fiblist
    (scons 1
        ((lambda fibgen (fib-2 fib-1)
             (let ((x (plus fib-2 fib-1))) (scons x (fibgen fib-1 x))))
            0 1)))

fiblist                        =>    (1 . ** closure **)
(stream-access fiblist 5)      =>    8
fiblist                        =>    (1 1 2 3 5 8 . ** closure **)
(stream-access fiblist 20)     =>    10946
```

3.4 Apply-all and Mapcar

Apply-all is a function which takes two lists and applies each of the elements of the first to the corresponding element in the second. The two lists should be of equal length.

Apply-all builds up a list, *l*, by adding the result of each successive apply to the end of the list. When the function list becomes empty, *l* is returned.

```
(define apply-all
    ((lambda loop (l funs args)
         (if (null funs)
             l
             (loop (append l (list ((car funs) (car args))))
                  (cdr funs)
                  (cdr args))))
       nil)))

(apply-all (car cdr 1+ 1-) '((a b) (c d) 10 20))  =>  (a (d) 11 19)
```

The *C*-Scheme function *mapcar* is somewhat similar. It applies a single function to the elements of a list:

```
(mapcar 1+ '(1 2 3 4 5))          =>   (2 3 4 5 6)
```

Mapcar suffers from the limitation that the function it applies must be a single argument function (or the result would be a list of closures). *Apply-all* can be used with *mapcar* to produce a macro, *mapcars*, which accepts multiple argument lists. A macro must be a function of one argument. It will be passed the list structure representing the entire invoking expression (for example, if *moo* is a macro in the expression *(moo x y)* then the argument passed to *moo* by the preprocessor would be *(moo x y)*). The expression returned by the macro will be used in place of the invoking expression.

The *mapcars* macro translates input in the form *(mapcars f l1 l2 ...)* into the corresponding calls to *mapcar* and repeated calls to *apply-all*. For example:

```
(mapcars (lambda (x y z) (list x y z)) '(a b c) '(d e f) '(g h i))
```

is equivalent to

```
(apply-all (apply-all (mapcar cons '(a b c)) '(d e f)) '(g h i))
```

```
(macro mapcars
    (let
        ((help
            (lambda help (x l)
                (if (null l)
                    x
                    (help `(apply-all ,x ,(car l)) (cdr l))))))
        (lambda (m)
            (help `(mapcar ,(cadr m) ,(caddr m)) (cdddr m)))))
```

Mapcars uses *mapcar* with the function and the first argument list. *Apply-all* is used to apply this list to the second argument list, again to apply this result to the next list and so on.

```
(mapcars 1+ '(1 2 3))                    =>   (2 3 4)
(mapcars cons '(a b c) '(1 2 3))         =>   ((a . 1) (b . 2) (c . 3))
```

4. Data Types

C-Scheme contains a small set of built-in data types. This section describes each of the data types, with examples where appropriate. The exact syntax accepted by the reader is given in §14.

All of the data types except for *cons cells* are considered to be *atoms*, even though some of the other data types may not truly be atomic (vectors, for example). This fact is often used in list manipulation routines to determine the boundaries of the list structure. *C*-Scheme provides the predicate *atom* of one argument as well as explicit predicates for each of the data types.

4.1 Inums

Inums are positive and negative integers, consisting of a sequence of digits optionally preceded by a plus or minus sign, e.g. *982374*, *-723*, *+1*. *Inums* are directly coded into pointers, so they take up no storage space. Because they are coded as pointers they have limited magnitude, approximately 2^{30}. Currently, *C*-Scheme supports no other numeric types.

4.2 Symbols

Symbols in *C*-Scheme are character sequences not containing left-parens, right-parens, or other characters interpreted specially by the reader. Only sequences which cannot be interpreted by the reader as a number are symbols, e.g. *plus*, *hi-there*, *1+*, *this_is_a_long_symbol*. Symbols are used as identifiers or data objects in *C*-Scheme programs (see §5). Every symbol has an associated *property list*, consisting of the symbol's *global value* followed by a list of alternating keys and values which can be accessed using the functions *get* and *put* (see §10). The symbol's global value can only be changed by the *change!* special form; it is used by the evaluator and cannot otherwise be accessed.

4.3 Nil

Nil, also written *()*, represents the null pointer. It usually marks the end of a list. It also represents the boolean value *false* in conditional expressions. *Nil*'s value is always itself; its global value cannot be changed and *nil* cannot be rebound locally.

4.4 Cons cells

Cons cells are the building blocks of lists, trees, and other data structures. A cons-cell is an ordered pair of *C*-Scheme objects, the *car* and the *cdr*, printed as *(car . cdr)*. A list consists of a sequence of cons-cells linked by the *cdr* field, with the last *cdr* field pointing to *nil*, e.g. *(element1 . (element2 . (element3 . nil)))*, which can be (and normally is) abbreviated by *(element1 element2 element3)*. The *cars* of each of the cons-cells point to the elements of the list.

4.5 Vectors

Vectors are included in *C*-Scheme for efficiency in the access of fairly static, large structures. Normally there is no need for arrays in *C*-Scheme programs, cons-cells serve quite well. This version supports only the special class of one-dimensional arrays, or vectors. Each element in the array is a pointer to a *C*-Scheme object. Vectors are created by the primitive *vec* and vector elements may be accessed or altered with special array accessing primitives *getv* and *setv*.

The reader builds vectors automatically out of a list of expressions delimited with brackets, e.g. *[1 2 3 4 5]*, *[(the first vector element is a list) [the second is itself a vector]]*.

4.6 Function Closures

Closures, or function objects, are valid data types in *C*-Scheme. Since *C*-Scheme is lexically scoped, the variable bindings in existence when a function is defined must be retained as long as the function exists. These bindings are kept in a data structure called an *environment*. When a function is defined, it is *closed* in the current environment, yielding a *function closure*, or simply a *closure*.

While closures print as ** *closure* ** they cannot be entered directly; only the special form *lambda* can create function closures.

4.7 Strings

Strings, or character vectors, are provided. They are read and printed with surrounding double quotes, e.g. *"hi there. I am a string"*. They are typically used as arguments to some of the system primitives and in printing messages to enable blanks and other special characters without escaping. There are currently no *C*-Scheme primitives for creating or pulling apart strings.

4.8 Files

File pointers are *C*-Scheme objects created by the system calls *infile* and *outfile*. They are necessary for all input and output. The three files *stdin*, *stdout* and *stderr* are created by the system and globally bound to symbols of the same names.

While file pointers print as ** *file pointer* **, there is no way to enter one directly.

5. Identifiers

C-Scheme *identifiers* are symbols which are either bound lexically or globally to a *C*-Scheme object. An identifier may be globally bound by the special form *change!* or the macro *define*. Local (lexical) bindings are created by *lambda* expressions and by the macros *let*, *let** and *letrec*. A local identifier's value may be changed with *change!* or *define*, although this is almost never needed in *C*-Scheme code.

5.1 Scope

C-Scheme identifiers are *lexically scoped* as in Algol 60. That is, the set of identifiers accessible by an expression depends only upon the context it is created in. An expression can only reference identifiers bound in lexically surrounding text.

Since *C*-Scheme is lexically scoped, it is always possible to determine the local identifiers accessible to an expression by analyzing the surrounding text. This is not true in most Lisp systems since Lisp is normally dynamically scoped. In a dynamically scoped system the set of identifiers accessible within a function depends upon the context of the expression at run time (the flow of control). With lexical scoping an identifier can be seen only by subexpressions of the expression which defines the identifier. In *C*-Scheme, only the body of a lambda expression can use its formal parameters. The *C*-Scheme parser takes advantage of the lexical scoping to generate efficient code for accessing local identifiers.

5.2 Extent

The *extent* of an identifier is the time during which there is some active code which might reference the identifier. In traditional Lisp and in Algol 60 the extent of an identifier is limited to the life of the declaring code. This is not true for *C*-Scheme. Identifiers in *C*-Scheme have *indefinite extent*. The reason is that function closures are *first-class data objects* [Stoy 1977]. When a function is defined, it is *closed* with the current environment and this environment is restored whenever the function is applied. Since the function closure might exist in the system indefinitely, so might the bindings of the identifiers it uses.

5.3 Data Hiding

This combination of lexical binding and indefinite extent can be used to hide data and create abstract data types. A function or set of functions can be defined within an environment containing local identifiers not accessible to any other functions (using *let*). These functions can be made accessible while the data remains hidden.

Example:

```
(let ((stack ()))
    (progn
        (define empty
            (lambda () (null stack)))
        (define push
            (lambda (x)
                (progn (change! stack (cons x stack))
                    x)))
        (define top
            (lambda () (car stack)))
        (define pop
            (lambda ()
                (let ((x (car stack)))
                    (progn (change! stack (cdr stack))
                        x))))))

(empty)                          =>   t
(push 'a)                        =>   a
(top)                            =>   a
(push 'b)                        =>   b
(list (pop) (pop))               =>   (b a)
```

6. Control Structures

This section describes the *C*-Scheme special forms *quote*, *lambda*, *if*, *prog2* and *change!*. Other forms which help structure *C*-Scheme programs are given, mostly macros.

The following format for describing *C*-Scheme forms is used in this section and throughout the remainder of the manual:

form [class]

Alias: alias1 alias2 ...
Returns: value returned
Errors: what makes it produce error messages

Description and examples.

Form gives the syntax of the expression. The form always begins with a keyword or function name. For special forms and macros, the syntax can be fairly complex, involving combinations of *C*-Scheme expressions. For functions, the syntax always consists of the function name followed by zero or more arguments. The name given function arguments is significant; *obj* means any *C*-Scheme object is allowed, *symbol* means only a *symbol* is allowed, etc.

Class is one of *special* (for special forms), *macro*, *primitive* or *library* (*C*-Scheme-coded functions). It is possible for one identifier to have more than one syntax or class. For example, *lambda* is described as both a macro and as a special form. The preprocessor notices when the result of a macro has the same *macro keyword* as the invoking expression, and thus prevents infinite recursion. This is further explained in §17. The brackets around *class* are merely to separate it from *form*.

Aliases are alternate symbols usable in place of the keyword or function name (see the *alias* primitive).

6.1 Quote

(quote *object***)** [special]

Returns: object with no interpretation

This is useful for creating list or identifier data. As noted in the reader section, *'exp* is the same as *(quote exp)*.

```
(quote 3)              =>   3
(quote a)              =>   a
'a                     =>   a
'(a b c)               =>   (a b c)
''(hi there)           =>   (quote (hi there))
'(lambda (x) 3)        =>   (lambda  (x) 3)
```

6.2 Function definition

(lambda *(id1 id2 ...) exp***)** [special]

(lambda *id0 (id1 id2 ...) exp***)** [macro]

Returns: function closure

Lambda closes *id1 id2 ...* with *exp* in the current environment. When the closure is applied, the arguments are added to the closed environment as bindings for *id1 id2 ...* and *exp* is evaluated in the extended environment.

The preprocessor performs the translation of this form into the *curried* version. The *curried* version is *not* accepted by the preprocessor. Since function definitions are *curried*, there is no need to apply this closure to all of its arguments at once. It is often worthwhile to apply a closure to one argument at a time, yielding an intermediate function of less arguments.

If *id0* is specified, *id0* is bound to the resulting closure itself within the closed environment (but not outside the closed environment); this allows terse definitions of recursive functions.

The case where the list of formals *(id1 id2 ...)* is empty is handled by creating a closure with *nil* as a parameter. It is still a function of one argument, but the argument is ignored, since *nil* is constant in all environments, and any local binding is ineffective. Thus, *exp* is conceptually *frozen* in the current environment until the closure is subsequently applied. It may be applied to no arguments, e.g. *(foo)*, which is translated by the preprocessor to *(foo nil)*. See *freeze* and *thaw* below.

```
(lambda (x) x)                                  =>   ** closure **
((lambda (x) x) 'anything)                      =>   anything

((lambda (x) (* x x)) 15)                       =>   225

(lambda (x y)
    (times x (plus y y)))                       =>   ** closure **

((lambda (x y)
     (times x (plus y y)))
    8)                                          =>   ** closure **

(((lambda (x y)
      (times x (plus y y)))
     8)
    2)                                          =>   32

(lambda fact (x)
      (if (0? x)
          1
          (* x (fact (1- x)))))                 =>   ** closure **

((lambda fact (x)
        (if (0? x)
            1
            (* x (fact (1- x)))))
    4)                                          =>   24

((lambda (f)
     (f 3 4))
    cons)                                       =>   (3 . 4)

(lambda () 1234)                                =>   ** closure **
((lambda () 1234))                              =>   1234

((lambda (x)
     (lambda () x))
    1234)                                       =>   ** closure **
(((lambda (x)
      (lambda () x))
     1234))                                     =>   1234
```

(freeze *exp*) [macro]

Returns: *exp frozen* in the current environment

Equivalent to *(lambda () exp)*. This is useful in conjunction with *thaw* to implement call-by-name.

(thaw *exp*) [macro]

Returns: result of invoking *exp* with no arguments
Errors: *exp* not a closure

```
(thaw (freeze (+ 2 3)))            =>   5

(define thunk
    ((lambda (x)
         (freeze x))
        'hithere))                 =>   ** closure **
(thaw thunk)                       =>   hithere
```

(iterate *id0* ((*id1 exp1*) (*id2 exp2*) ...) *exp*) [macro]

Same as:

```
((lambda id0 (id1 id2 ...) exp)
    exp1 exp2 ...)
```

6.3 Conditionals

(if *test-exp then-exp else-exp*) [special]

(if *test-exp then-exp*) [macro]

Returns: value of *then-exp* or *else-exp*, depending upon value of *test-exp*. If *else-exp* not specified, it defaults to *nil*

First *test-exp* is evaluated. If the result is non-*nil*, *then-exp* is evaluated and returned. Otherwise, *else-exp* is evaluated and returned. *Else-exp* is generally only left out when the value is not needed, i.e. *then-exp* performs a side-effect such as a read or print.

```
(if t 'then 'else)               =>   'then
(if nil 'then 'else)             =>   'else
(if (cdr '(a b c)) 'then 'else)  =>   'then
(if (cdr '(a)) (princ 'yes))     =>   nil     (nothing printed)
```

(**cond** (*test-exp1 exp1*) (*test-exp2 exp2*) ...) [macro]

(**cond** (*test-exp1 exp1*) (*test-exp2 exp2*) ... (*final-exp*))

Returns: the value of *expi* corresponding to the first non-*nil test-expi*, or *nil*.

The tests are evaluated sequentially starting with *test-exp1*. If any is non-*nil*, the corresponding *expi* is evaluated and returned as the result of the *cond*. If none of the tests are non-*nil*, *nil* is returned.

In the second form, if none of the tests are true, the value of *final-exp* is returned, i.e. *final-exp* is the "otherwise" clause.

```
(define equal
   (lambda (x y)
      (cond ((eq x y) t)
         ((atom x) nil)
         ((equal (car x) (car y))
            (equal (cdr x) (cdr y))))))
(equal 'a 'b)                                  =>   nil
(equal 'a 'a)                                  =>   t
(equal '(a b c) '(a b d))                      =>   nil
(equal '(1 (2)) '(1 (2)))                      =>   t
```

(**case** *tag-exp* (*id1 exp1*) (*id2 exp2* ...) [macro]

(**case** *tag-exp* (*id1 exp1*) (*id2 exp2*) ... (**otherwise** *final-exp*))

Returns: value of expression with *id eq* to value of *tag-exp*. If no *id* matches the value of *tag-exp*, *nil* is returned (or the value of *final-exp*, if the **otherwise** clause exists).

Tag-exp is evaluated first and bound to the identifier tag in the same environment as the *expi*.

```
(mapcar
     (lambda (x)
         (case x
              (a 'A)
              (b 'B)
              (c 'C)
              (otherwise 'F)))
     '(b c d a b+))                      =>   (B C F A F)
```

(**or** *exp1 exp2 ...*) [macro]

Returns: *t* if any of the expressions evaluates to a non-*nil* value, *nil* otherwise

The expressions are evaluated in sequence; once one of the expressions yields a non-*nil* value no more of the expressions are evaluated.

```
(or nil nil nil)          =>   nil
(or nil nil 'a)           =>   t
(or 'a nil)               =>   t

(define x t)              =>   x
(or x (change! x 0))      =>   t
x                         =>   t
```

(**and** *exp1 exp2 ...*) [macro]

Returns: *nil* if any of the expressions evaluates to a *nil* value, *t* otherwise

The expressions are evaluated in sequence; once one of the expressions yields a *nil* value no more of the expressions are evaluated.

```
(and t t t)               =>   t
(and '(hi) nil 'a)        =>   nil
(and "there" nil)         =>   nil

(define x nil)            =>   x
(and x (change! x 0))     =>   nil
x                         =>   nil
```

(**test** *exp1 exp2 exp3*) [macro]

Returns: if *exp1* is non-*nil* then *exp2* is evaluated and applied to the result of *exp1*, otherwise *exp3* is is evaluated and returned.
Errors: *exp1* evaluates to true and *exp2* does not evaluate to a closure.

Test can be used with predicates which return useful non-*nil* values.

```
(test (memq 'a '(1 2 3 a b c)) cadr nil)          =>   b
(test (cdr '(a b c)) (lambda (x) x) 'empty)      =>   (b c)
```

6.4 Sequencing

(prog2 *exp1 exp2*) [special]

Returns: value of *exp2*

Exp1 is evaluated and its value ignored, then *exp2* is evaluated and its value returned. Note that *exp1* and *exp2* are guaranteed to be executed in sequence. Prog2 is used primarily by the *progn* macro.

(progn *exp1 exp2* ...) [macro]

Alias: block
Returns: value of the last *exp*

Equivalent to (prog2 exp1 (prog2 exp2 (prog2 ...))).

The expressions are evaluated in sequence starting with *exp1*. All of the values except for the last are thrown away; the last value is returned. *Progn* is normally used to sequence side-effects, especially i/o.

```
(progn)                        =>   nil
(progn 1)                      =>   1
(progn (read) (read))          =>   reads two objects,
                                        returning the second.
```

6.5 Identifier Assignment

(change! *id exp*) [special]

Alias: change, setq
Returns: value of *exp*
Errors: *id* is *nil* or *t* (unless *t* is bound locally)

Exp is evaluated and *id* is bound to its value. If there is no local binding of *id*, the *global value* of *id* is set.

```
(change! x '(a b c d e))          =>   (a b c d e)
(length x)                        =>   5
(change! x (cdr x))               =>   (b c d e)
x                                 =>   (b c d e)

((lambda (x)
     (progn (change! x 3)
         (1+ x)))
    'ignored)                     =>   4
```

(define *id exp*) [macro]

Returns: *id*

Define is the same as *change!* except that it returns *id* instead of the value of *exp*. It is generally used to create global variables, especially function definitions.

```
(define x 3)                 =>   x
(+ x 10)                     =>   13

(define identity
    (lambda (x) x))          =>   identity
(identity [a b c d])         =>   [a b c d]
```

(let ((*id1 exp1*) (*id2 exp2*) ...) *exp*) [macro]

Returns: value of *exp* in the current environment augmented by the bindings of the *idi*s to the *expi*s

The *expi*s are evaluated (in any order) and bound to the corresponding *idi*s.

```
(let ((a '(a b c)) (b '(1 2 3)))
    (append a b))                              =>   (a b c 1 2 3)

(define c...r
    (let
        ((a car)
         (d cdr))
        (lambda (x)
            (lambda recurse (l)
                (if (null l)
                    x
                    ((if (eq (car l) 'a)
                         a
                         d)
                        (recurse (cdr l))
                    ))))))                     =>   c...r

(c...r '(1 (2 3) 4) '(a d a d))                =>   3
```

(let* ((*id1 exp1*) (*id2 exp2*) ...) *exp*) [macro]

Returns: value of *exp* in the current environment augmented by the bindings of the *idi*s to the *expi*s

The *expi*s are evaluated from left to right and bound to the corresponding *idi*s. Each *expi* is evaluated in an environment which contains the bindings of the previous identifiers.

```
(let* ((x '(a b c d)) (y (length x)))
    (cons y x))                                 =>   (4 a b c d)
```

(letrec ((*id1 exp1*) (*id2 exp2*) ...) *exp*) [macro]

Alias: labels
Returns: value of *exp* in environment augmented by the bindings of the *idi*s to the *expi*s

The *expi*s are evaluated (in any order) and bound to the corresponding *idi*s. If any of the *expi*s return closures these closures are bound in an environment which includes the new bindings. This allows the definition of mutually recursive functions.

```
(letrec ((a '(a b c)) (b '(1 2 3)))
    (append a b))                                  =>    (a b c 1 2 3)

(letrec
    ((x 3)
     (f (lambda (y)
          (if (0? y) 'f (g (- x y)))))
     (g (lambda (z)
          (if (0? z) 'g (f (+ z 1))))))
    (f 1))                                         =>    g
```

6.6 Continuations

Continuations are special closures which carry with them sufficient information to continue the computation from a given point. A particular continuation is invoked with the value of each *C*-Scheme expression that is evaluated. For example, during evaluation of (1+ (car x)) there will be a continuation waiting for the value of *(car x)* which will add *1* to it and return (via another continuation).

C-Scheme allows the user to request the continuation at any point with the function *call-with-current-continuation*, or *call/cc*.

(**call-with-current-continuation** *closure*) [primitive]

Alias: call/cc
Returns: the value of applying *closure* to the current continuation

Within the body of *closure*, the formal parameter will be bound to the continuation of the *call/cc* application. At any time this continuation (which acts like a normal closure) may be invoked with a single argument *x*. This will return *x* to the original caller of *call/cc*. If the continuation is not invoked in the body of *closure*, control passes back normally to the caller.

So far, *call/cc* looks like a label for non-local exits. While this is the most common use for *call/cc*, things can get much more complex. The continuation may be passed back, or set to an identifier visible outside the *call/cc*. Even after the *call/cc* returns normally the continuation may be invoked successfully. This causes control to return back to the *call/cc* as if the invocation had occurred within *closure*. Thus, control may be thrown both up and down the expression being evaluated. Coroutines and backtracking searches may be implemented with *call/cc*, and other more unusual control structures.

See the macros *catch* and *throw* for alternate syntax.

A multitasking scheduler using continuations with timer interrupts is shown in the next subsection, after the explanation of timer interrupts.

See [Wand 1980], [Fessenden et.al. 1983] and [Steele & Sussman 1978] for some interesting uses of *call/cc* (as *catch*).

```
(call/cc
    (lambda (c)
        (cons 'a (cons 'b 'c))))    =>   (a b . c)
(call/cc
    (lambda (c)
        (cons 'a (c 3))))           =>   3

(change! c
     (call/cc (lambda (c) c)))      =>   ** closure **
(c c)                               =>   ** closure **
(c 3)                               =>   3
c                                   =>   3
```

(**catch** *id exp*) [macro]

Returns: value of exp, or *val* if *id* is applied to *val*

Equivalent to *(call/cc (lambda (id) exp))*. See the primitive function *call/cc*.

(**throw** *exp1 exp2*) [macro]

Returns: does not return to caller
Errors: *exp1* does not evaluate to a continuation (closure)

Equivalent to *(exp1 exp2)*.

Throw merely invokes *exp1* with *exp2* and is normally used to emphasize that the expression being invoked is a continuation.

```
(define read-til-eof
    (lambda (fp)
        (catch return
            ((lambda loop (l)
                (let ((x (read fp 'eof)))
                    (if (eq x 'eof)
                        (throw return l)
                        (loop (cons x l)))))
             nil))))
```

This function will read from file fp, building up a first-in-first-out list of the expressions read. When an eof is found, the list is returned. (This function could be written without the *catch*.)

6.7 Timer Interrupts

C-Scheme currently allows the user to handle three types of interrupts: timer interrupts set by user control, keyboard interrupts and a garbage collection interrupt. The control of collector and keyboard interrupts is discussed in §13.

C-Scheme allows the user to set a count-down timer which invokes an interrupt routine after a specified number of discrete time intervals, called *ticks*. Each *tick* is a constant number of virtual machine cycles (see §16). The interrupt routine may be any function of zero arguments, i.e. a *thunk*. The continuation of the interrupt routine is the state of the current computation, so the computation will continue if the routine returns normally. Of course, the routine is free to save the computation with *call/cc*, or to pass control to a continuation formed earlier, or to generate an error. A multitasking scheduler may be built using continuations with timer interrupts (see the example below).

The function *timer* is used to control the timer. It takes two arguments: the number of *ticks* and the *interrupt service routine*. *Timer* returns a *cons cell*. The *car* is the number of ticks remaining from the previous call to *timer* (or zero if timer was disabled). The *cdr* is the interrupt service routine set by the last call to *timer* (or *nil* if the timer was disabled).

The timer may be disabled by passing 0 ticks and a null interrupt service routine, i.e. *(timer 0 nil)*. Also, any *C*-Scheme error causes the timer to be disabled.

It is possible to write functions to temporarily disable the timer or check how much time remains, using only *timer*. These functions are left as an exercise for the reader.

(timer *n closure*) [primitive]

Returns: cons cell containing old timer and closure (ticks . closure)

The following example is a simple multitasking scheduler:

```
(letrec
    ((pq (queue))
     (timer-handler
         (lambda ()
             (catch 1 (progn (pq 'put 1) (dispatch)))))
     (dispatch
         (lambda ()
             (if (pq 'empty)
                 (error "dispatch: process queue is empty" nil)
                 (progn (timer 10 timer-handler)
                     (throw (pq 'get) nil))))))
    (define process
        (lambda (thunk)
            (test (catch 1 1)
                (lambda (p)
                    (let ((t (timer 0 nil)))
                        (progn (pq 'put p)
                            (if (0? (car t))
                                (dispatch)
                                (timer (car t) (cdr t))))))
                (progn (thaw thunk) (timer 0 nil) (dispatch))))))
```

This elementary scheduler allows multiple processes to be served in round-robin fashion. A process, actually a continuation, is created from a *thunk* by the function *process*. Inactive processes reside on a process queue, *pq* (an instance of the *queue* abstract datatype from §3). The first process created is dispatched immediately. When a process is dispatched the timer is set. Subsequently, processes are dispatched only when the timer expires or when the active process terminates. When a timer interrupt occurs the old active process goes to the rear of the queue. New processes are also placed at the rear of the queue; the code which spawned them remains active.

The trickiest part of the code is the *test* expression in *process*. The return value from *catch* looks like it is always a continuation. Indeed, when *process* is called to create the process, the *catch* does return the continuation; this continuation becomes the new process. That is the non-*nil* branch of the test. The test is actually executed one other time, when the process is first dispatched. In *dispatch*, the continuation is given *nil*, and this becomes the return value of the *catch*, the *nil* branch of the test. In this manner, the code determines whether the process is being created or executed.

Pq, *timer-handler* and *dispatch* are hidden within the *letrec*. The only way to access the scheduler is with *process*, preventing unauthorized access to the scheduler. Also, all accesses to *pq* happen with the timer disabled. Otherwise a lock-out would be needed to prevent concurrent access.

The following process spawns two processes, which in turn spawn two processes each, which in turn ...

```
(process
    ((lambda rabbit (n)
         (freeze
             (let ((next (rabbit (1+ n))))
                 (progn (princ n)
                     (newline)
                     (process next)
                     (process next)))))
     0))
```

6.8 Macro Definition

(macro *id exp*) [macro]

Returns: *id*
Errors: *exp* does not evaluate to a closure

First *exp* is evaluated. Its value, which must be a closure, is placed on *id*'s property list under the key ***macro***. This effects preprocessing of any subsequent form which has *id* as its first element. When such a form is encountered, the closure is applied to the entire form, and the value of this application is used in place of the form. The result is sent back to the preprocessor (and more macros may be expanded within it). For more information and examples refer to §17.

```
(macro list
    (lambda (m)
        (if (null (cddr m))
            nil
            '(cons ,(cadr m)
                (list . ,(cddr m))))))        =>   list
(list 1 2 (+ 1 2))                            =>   (1 2 3)
```

7. Predicates

Predicates return *nil* for false and normally return the atom *t* for true, although some predicates return useful non-*nil* values.

Numeric predicates are described in §9.

(eq *obj1 obj2*) [primitive]

Alias: eq?
Returns: *t* if *obj1* and *obj2* are physically the same pointer

Eq compares the pointers *obj1* and *obj2*, NOT the objects to which they point. *Eq* is normally used to compare symbols. Since symbols are placed in a symbol table, two symbols which are typed in the same will occupy the same address (and be *eq*). *Inums*, which are encoded as pointers, may be tested with *eq*. This is not good practice, since other numbers which may have the same value may not be *eq*. Test for numeric equality with the primitive function "=".

```
(eq 'a 'b)                          =>   nil
(eq 'a 'a)                          =>   t
(eq '(a b c) '(d e f))              =>   nil
(eq '(a b c) '(a b c))              =>   nil

(let* ((x '(a b c)) (y x))
     (list (eq x '(a b c))
         (eq x x)
         (eq x y)))                 =>   (nil t t)
```

(**equal** *obj1 obj2*) [library]

Alias: equal?
Returns: *t* if *obj1* is *eq* to *obj2*, or if *obj1* and *obj2* are numbers and are =, or if *exp1* and *exp2* are equivalent list objects, i.e. their *cars* and *cdrs* are equal.

All non-numeric atoms which are not *eq* will always be non-*equal*. *Equal* really should test for equivalent string and vector objects as well.

```
(equal 'a 'a)                   =>   t
(equal 'a 'b)                   =>   nil
(equal 4 4)                     =>   t
(equal '(a b) '(a c))           =>   nil
(equal '(a (b)) '(a (b)))       =>   t
```

(null *obj***)** [primitive]

Alias: not, null?
Returns: *t* if *obj* is *nil*, *nil* otherwise

```
(null 'a)                 =>  nil
(null '(a b c))           =>  nil
(null nil)                =>  t
(null (cdr '(a)))         =>  t
```

(atom *obj***)** [primitive]

Alias: atom?
Returns: *nil* if *obj* is a *cons cell*, *t* otherwise.

```
(atom 'a)                 =>  t
(atom nil)                =>  t
(atom 3)                  =>  t
(atom '(a b c))           =>  nil
(atom (lambda (x) x))     =>  t
(atom [a b c d e f])      =>  t
```

(consp *obj***)** [primitive]

Alias: cons?
Returns: *t* if *obj* is a *cons cell*, *nil* otherwise

Logical opposite of *atom*.

```
(consp '(a b c))          =>  t
(consp "abc")             =>  nil
(consp nil)               =>  nil
```

(listp *obj***)** [primitive]

Alias: list?
Returns: *t* if *obj* is a *cons cell* or *nil*, *nil* otherwise

```
(listp '(a b c))          =>  t
(listp "abc")             =>  nil
(listp nil)               =>  t
```

(symbolp *obj***) [primitive]

Alias: symbol?
Returns: *t* if *obj* is a *symbol*, *nil* otherwise

```
(symbolp 'abc)          =>  t
(symbolp "abc")         =>  nil
(symbolp nil)           =>  t
(symbolp '(a b c))      =>  nil
```

(numberp *obj***) [primitive]

Alias: number?
Returns: *t* if *obj* is a *number*, *nil* otherwise

```
(numberp 3)             =>  t
(numberp "abc")         =>  nil
(numberp nil)           =>  nil
(numberp '(a b c))      =>  nil
```

(stringp *obj***) [primitive]

Alias: string?
Returns: *t* if *obj* is a *string*, *nil* otherwise

```
(stringp 3)             =>  nil
(stringp "abc")         =>  t
(stringp nil)           =>  nil
(stringp [a b c])       =>  nil
```

(vectorp *obj***) [primitive]

Alias: vector?
Returns: *t* if *obj* is a *vector*, *nil* otherwise

```
(vectorp 3)             =>  nil
(vectorp "abc")         =>  nil
(vectorp nil)           =>  nil
(vectorp [a b c])       =>  t
```

(closurep *obj*) [primitive]

Alias: closure? procp proc?
Returns: *t* if *obj* is a *closure*, *nil* otherwise

Both primitive functions and functions defined with *lambda* are *closures*, so *closurep* returns *t* for both.

```
(closurep (lambda (x) (+ x 3)))      =>   t
(closurep sub1)                      =>   t
(closurep '(a b c))                  =>   nil
(closurep 'x)                        =>   nil
```

(filep *obj*) [primitive]

Alias: file?
Returns: *t* if *obj* is a file pointer, *nil* otherwise

```
(filep (infile " /lib/funs.s"))    =>   t
(filep stdin)                      =>   t
(filep '(a b c))                   =>   nil
(filep 'x)                         =>   nil
```

(constantp *obj*) [library]

Returns: *t* if *obj* is a constant, *nil* otherwise

An object is constant if it is an atom and not a symbol, or if it is a list whose car is *quote*:

```
(define constantp
    (lambda (x)
        (if (atom x)
            (not (symbolp x))
            (eq (car x) 'quote))))

(constantp nil)                   =>   t
(constantp 'a)                    =>   nil
(constantp ''a)                   =>   t
(constantp "hi there")            =>   t
(constantp (plus 3 4))            =>   t
(constantp '(a b c))              =>   nil
```

8. List Manipulation

(cons *obj1 obj2*) [primitive]

Returns: a *cons cell* with car *obj1* and cdr *obj2*

Cons is the list *cons*tructor function. See the macro *list* for a more concise way of building lists of several elements.

```
(cons 'a 'b)                      =>   (a . b)
(cons 'a 'nil)                    =>   (a)
(cons 'a '(b c))                  =>   (a b c)
(cons '(a b c) '(p q))            =>   ((abc) p q)
(cons () ())                      =>   (nil)
(cons "hi"  [t h e r e])          =>   ("hi" . [t h e r e])
```

(car *list*) [primitive]

Returns: the first element of *list* (the car)

If *list* is *nil*, *car* returns *nil*. This is especially handy in macro definitions, often leading to more robust code with less error checking required.

```
(car '(a . b))              =>   a
(car '(a b c))              =>   a
(car (cons 'x 'y))          =>   x
(car (car nil))             =>   nil
```

(cdr *list*) [primitive]

Returns: all but the first element of *list* (the cdr)

The *cdr* of *nil* is *nil*. See the note under *car*.

```
(cdr '(a . b))              =>   b
(cdr '(a b c))              =>   (b c)
(cdr (cons 'x 'y))          =>   y
(cdr nil)                   =>   nil
```

(**c....r** *list*) [primitive]

Any combination of up to four *a*s and *d*s may be substituted for the dots, e.g. *cadr*, *cdaddr*, *cdddar*. Working from right to left in the string of *a*s and *d*s, *d* specifies to take the *cdr* and *a* specifies to take the *car*. For example, (cadr x) is equivalent to (car (cdr x)).

```
(cadr '(a b c))              =>  b
(cddr '(a b c))              =>  (c)
(caddr '(a b c))             =>  c
(caar '((a b) (c d)))        =>  a
(cdar '((a b) (c d)))        =>  (b)
(cdddr '(a b c))             =>  nil
```

(list *exp1 exp2* ...) [macro]

Returns: list of the values of *exp1*, *exp2* ...

The expressions may be evaluated in any order.

```
(list 1 2 3)             =>  (1 2 3)
(list 'a "hi" '(b))      =>  (a "hi" (b))
(list)                   =>  ()
```

(rplaca! *list obj*) [primitive]

Alias: rplaca
Returns: modified *list*

Replaces the *car* of *list* with *obj*.

```
(rplaca! '(a b c) 'X)        =>  (X b c)
(rplaca! '(x y) '(a b))      =>  ((a b) y)
```

(rplacd! *list obj*) [primitive]

Alias: rplacd
Returns: modified *list*
Errors: *list* not a *cons cell*

Replaces the *cdr* of *list* with *obj*.

```
(rplacd! '(a b c) 'X)        =>  (a . X)
(rplacd! '(x y) '(a b))      =>  (x a b)
```

(length *obj***)** [primitive]

Returns: the length of *obj*
Errors: *obj* is not a list, vector or string

```
(length '(a b c))          =>   3
(length nil)               =>   0
(length "abcdefg")         =>   7
(length [a b c d e])       =>   5
(length [])                =>   0
```

(append *list obj***)** [primitive]

Returns: a list consisting of all the elements of *list* followed by all of the elements of *obj*.

A copy is made of *list* and its last *cdr* is made to be *obj*.

```
(append '(a b c) '(1 2 3))     =>   (a b c 1 2 3)
(append '(a b c) 3)            =>   (a b c . 3)
(append '(a b c . d) 3)        =>   (a b c . 3)

(define x '(a b c))            =>   x
(append x '(1 2 3))            =>   (a b c 1 2 3)
x                              =>   (a b c)
```

(nconc *list obj***)** [primitive]

Returns: a list consisting of all the elements of *list* followed by all of the elements of *obj*.

Nconc performs a destructive *append*, and is more efficient than *append* when *list* can be clobbered.

```
(nconc '(a b c) '(1 2 3))      =>   (a b c 1 2 3)
(nconc '(a b c) 3)             =>   (a b c . 3)
(nconc '(a b c . d) 3)         =>   (a b c . 3)

(define x '(a b c))            =>   x
(nconc x '(1 2 3))             =>   (a b c 1 2 3)
x                              =>   (a b c 1 2 3)
```

(memq *obj list***)** [primitive]

Returns: the first sublist of *list* whose *car* is *eq* to *obj*, or *nil*

```
(memq 'a '(x y z))             =>   nil
(memq 'b '(a b c))             =>   (b c)
(memq 'x '(a x b x c x))       =>   (x b x c x)
(memq '(a b) '((a) (a b)))     =>   nil
```

(member *obj list*) [library]

Returns: the first sublist of *list* whose *car* is *equal* to *obj*, or *nil*

```
(member 'a '(x y z))                  =>   nil
(member 'b '(a b c))                  =>   (b c)
(member '(a b) '((a) (a b)))          =>   ((a b))
```

(remq *obj list*) [primitive]

Returns: a copy of *list* with all subexpressions *eq* to *obj* removed.

```
(remq 'a '(x y z))              =>   (x y z)
(remq 'b '(a b c))              =>   (a c)
(remq 'x '(a x b x c x))        =>   (a b c)
(remq '(a b) '((a) (a b)))      =>   ((a) (a b))
```

(remove *obj list*) [library]

Returns: a copy of *list* with all subexpressions *equal* to *obj* removed.

```
(remove 'a '(x y z))                  =>   (x y z)
(remove 'b '(a b c))                  =>   (a c)
(remove '(a b) '((a) (a b)))          =>   ((a))
```

(reverse *list*) [primitive]

Returns: a new list with the elements of *list* in reverse order

```
(reverse '(a b c d e))          =>   (e d c b a)
(reverse '(x (r s) q))          =>   (q (r s) x)
(reverse nil)                   =>   nil
```

(mapc *closure list*) [library]

Returns: *nil*

Closure is applied to the elements of *list* in sequence from left to right. *Mapc* is often used to print out a list of items.

```
(mapc
    (lambda (x)
        (progn (princ x)
            (newline)))
    '(hi there to you ))          =>   nil
                                         prints hi, there, to and you,
                                              each on its own line.
```

(**mapcar** *closure list*) [library]

Returns: list of the results of applying *closure* to each element of *list*

```
(mapcar -- '(1 2 3 4 5))        =>   (-1 -2 -3 -4 -5)
(mapcar cadr '((1 2) (3 4)))   =>   (2 4)
(mapcar (lambda (x) nil)
     '(a b c d e))              =>   (nil nil nil nil nil)
```

9. Numeric Computations

(0? *n*) [primitive]

Alias: zero?, zerop
Returns: *t* if *n* is 0, otherwise *nil*
Errors: *n* not a number

```
(0? (1- 1))          =>   t
(0? (+ 1 2))         =>   nil
```

(= *n1* *n2*) [primitive]

Returns: *t* if *n1* equals *n2*, otherwise *nil*
Errors: *n1* or *n2* not a number

This primitive is guaranteed to work for all numbers. The primitive eq works only for *inums* (which are encoded as addresses), so it is a good practice to use "=".

```
(= 4 (1- 5))         =>   t
(= 4 (1+ 5))         =>   nil
```

(< *n1* *n2*) [primitive]

Alias: less?, lessp
Returns: *t* if *n1* is less than *n2*, otherwise *nil*
Errors: *n1* or *n2* not a number

```
(< 23 98)            =>   t
(< 98 23)            =>   nil
(< -10 -10)          =>   nil
```

(> *n1* *n2*) [primitive]

Alias: greater?, greaterp
Returns: *t* if *n1* is greater than *n2*, otherwise *nil*
Errors: *n1* or *n2* not a number

```
(> 51 -16)           =>   t
(> -16 51)           =>   nil
(> 12 12)            =>   nil
```

(<= *n1 n2*) [primitive]

Returns: *t* if *n1* is less than or equal to *n2*, otherwise *nil*
Errors: *n1* or *n2* not a number

```
(<= -12 -15)        =>   nil
(<= -15 -12)        =>   t
(<= 9 9)            =>   t
```

(>= *n1 n2*) [primitive]

Returns: *t* if *n1* is greater than or equal to *n2*, otherwise *nil*
Errors: *n1* or *n2* not a number

```
(>= -15 30)         =>   nil
(>= 30 -15)         =>   t
(>= 0 0)            =>   t
```

(1- *n*) [primitive]

Alias: sub1
Returns: *n* - 1
Errors: *n* not a number

```
(1- 23)             =>   22
(1- (+ 2 3))        =>   4
```

(1+ *n*) [primitive]

Alias: add1
Returns: *n* + 1
Errors: *n* not a number

```
(1+ 23)             =>   24
(1+ -1)             =>   0
```

(+ *n1 n2*) [primitive]

Returns: sum of *n1* and *n2*
Errors: *n1* or *n2* not a number

See the macro *plus*, which can take more than two arguments.

```
(+ 4 12)            =>   16
(+ 15 -10)          =>   5
(define 5+ (+ 5))   =>   ** closure **
(5+ 50)             =>   55
```

(**plus** *n1 n2* ...) [macro]

Returns: the sum of *n1*, *n2* ...
Errors: non-numeric argument

The *n*s are evaluated in an unspecified order and the sum of the results is returned. At least one argument is required.

```
(plus (* 3 4) 9 (/ 8 2))      =>   25
(plus 9)                      =>   9
```

(- *n1 n2*) [primitive]

Alias: difference
Returns: *n2* subtracted from *n1*
Errors: *n1* or *n2* not a number

```
(- 50 2)          =>   48
(- -20 -30)       =>   10
```

(-- *n*) [primitive]

Alias: minus
Returns: *n* negated, same as (- 0 *n*)
Errors: *n* not a number

```
(-- 30)           =>   -30
(-- -10)          =>   10
(-- 0)            =>   0
```

(* *n1 n2*) [primitive]

Returns: product of *n1* and *n2*
Errors: *n1* or *n2* not a number

See the macro *times*, which can take more than 2 arguments.

```
(* 3 15)          =>   45
(* 3 (-- 10))     =>   -30
```

(**times** *n1 n2* ...) [macro]

Returns: the product of *n1*, *n2* ...
Errors: non-numeric argument

The *n*s are evaluated in an unspecified order and the product of the results is returned. At least one argument is required.

```
(times 5 6 (plus 4 1))      =>    150
(times 101)                 =>    101
```

(/ *n1 n2*) [primitive]

Alias: quotient
Returns: *n1* divided by *n2*
Errors: *n1* or *n2* not a number, *n2* = 0

Integer division.

```
(/ 25 5)      =>    5
(/ 17 3)      =>    5
```

(% *n1 n2*) [primitive]

Alias: mod, remainder
Returns: *n1* mod *n2*
Errors: *n1* or *n2* not a number, *n2* = 0

```
(% 25 5)      =>    0
(% 17 3)      =>    2
```

10. Property Lists and Aliases

There are several functions for manipulating the property list of a symbol. The format of a property list is:

```
(global-value key1 value1 key2 value2 ...)
```

The value of any property may be asked for at any time with the function *get*, new properties may be added (or existing properties changed) with *put*, or the entire property list may be obtained with *plist*. *Put* and *get* use *eq* when searching for a property, so symbols are normally used as keys.

There is no direct way to set the property list of a symbol (*put* only alters the contents). However, the function *alias* will cause the property list of one symbol to point to the same property list as another symbol. This has the effect of making the symbols act alike but print differently.

Many identifiers are *alias*ed already, as noted in some of the form descriptions in this and other sections.

Since local bindings are not stored on the property list of symbols, *alias* does not affect lexically scoped identifiers in any way.

The preprocessor looks at the properties **prep** and **macro**, but pays no attention to the name of a symbol. The only case where the symbols might act differently is during a call directly to *eval*, since the parser does look at the name to determine what are special forms. *Execute* preprocesses its argument before evaluating it, which is normally necessary anyway.

Note that once *aliased*, the symbols share a property list. Therefore, a change to one means a change to the other. This insures that the symbols are truly alike; a macro defined for one will work for the other, and any change of global binding will effect both.

(plist *symbol*) [primitive]

Returns: the property list of *symbol*

(get *symbol obj*) [primitive]

Returns: the value corresponding to key *obj* in *symbol*'s property or nil if key *obj* is not found

(put *symbol obj1 obj2*) [primitive]

Returns: *obj2*

Places value *obj2* under key *obj1* on the property list of *symbol*.

(alias *symbol1 symbol2*) [primitive]

Returns: the property list of *symbol2*

Changes *symbol1* so that it shares *symbol2*'s property list. Any properties *symbol1* might have had are lost, along with its global value.

```
(plist 'x)                       =>    (**unbound**)
(plist 'y)                       =>    (**unbound**)
(define x 3)                     =>    3
(plist 'x)                       =>    (3)
(get 'x 'prop)                   =>    nil
(put 'x 'prop 'value)            =>    value
(get 'x 'prop)                   =>    value
(alias 'x 'y)                    =>    (**unbound**)
(define x (lambda () "hi"))      =>    x
(y)                              =>    "hi"
```

11. Vector Manipulation

(**vec** *n obj*) [primitive]

Returns: a new vector of length *n* (indexed from zero) filled with *exp*
Errors: *n* not a number

There is no restriction on the size of vectors, unless a vector allocation would cause *C*-Scheme's address space (as determined by the operating system) to be exceeded.

```
(vec 4 nil)          =>   [nil nil nil nil]
(vec 10 'a)          =>   [a a a a a a a a a a]
(vec 0 nil)          =>   []
```

(**putv** *vector n obj*) [primitive]

Returns: *obj*
Errors: index *n* out of range

Sets the *n*th element of *vector* to *obj*.

```
(define v (vec 10 0))     =>   v
(putv v 0 1)              =>   1
v                         =>   [1 0 0 0 0 0 0 0 0 0]
```

(**getv** *vector n*) [primitive]

Returns: *n*th element of *vector*
Errors: index *n* out of range

```
(getv [a b c d e] 3)      =>   d
(define v [1 2 3 4])      =>   v
(putv v 0 0)              =>   0
(getv v 0)                =>   0
v                         =>   [0 2 3 4]
```

12. Input/Output Primitives

A few simple input/output primitives are provided in this version of *C*-Scheme. ***Read***, ***princ*** and ***newline*** require file pointers which are created by ***infile*** and ***outfile***. Three file pointers, ***stdin***, ***stdout***, and ***stderr***, are created by the system and bound to the obvious identifiers.

The macros *read*, *princ* and *newline* allow omiting the file arguments if input is to come from *stdin* or output to go to *stdout*.

(read *file obj*) [primitive]

(read *file*) [macro]

(read) [macro]

> Returns: the next expression read from *file*. If an eof is encountered, *obj* is returned.

If *file* is not specified, it defaults to *stdin*. If *obj* is not specified, it defaults to *nil*.

A *read* from *stdin* causes the *stdout* output buffer to be flushed, in order to facilitate interactive reading and printing.

(princ *obj file***)** [primitive]

(princ *obj***)** [macro]

> Returns: *obj*

Obj is printed on *file* with no carriage control. If *file* is not specified, it defaults to *stdout*.

```
(princ (cons 'a 'b))     prints   (a . b)
```

(newline *file***)** [primitive]

(newline) [macro]

> Returns: *nil*

A newline character (or newline characters) is written to *file*. and the output buffer associated with *file* is flushed. If *file1* is not specified it defaults to *stdout*.

(**infile** *filename*) [primitive]

Returns: a file pointer
Errors: file cannot be opened for reading

Filename must be a string. File *filename* is opened for reading.

```
(infile "input.s")        =>   ** file pointer **
(infile "lib/foo.s")      =>   ** file pointer **
```

(**outfile** *filename*) [primitive]

Returns: a file pointer
Errors: file cannot be opened for writing

Filename must be a string. File *filename* is opened for writing.

```
(outfile "output.s")      =>   ** file pointer **
```

(**close** *file*) [primitive]

Returns: *nil*
Errors: *file not open, or cannot be closed*

Closes file.

13. *C*-Scheme System Interface

(load *filename*) [library]

Returns: *nil*
Errors: file cannot be opened for reading

All *C*-Scheme code on file *filename* is executed as if entered from the terminal. *Load* may be used for preprocessed code saved with *save* (below).

```
(load "myfile.s")
```

(save *filename1 filename2*) [library]

Returns: *t*
Errors: file *filename1* cannot be opened for reading or file *filename2* cannot be opened for writing

Filename1 and *filename2* must be strings. *Save* prints the preprocessed version of file *filename1* onto file *filename2*. Currently, constant expressions are not evaluated by the preprocessor during a *save*, so this function is of questionable use. *Save*d files may be reloaded using *load*.

(execute *obj*) [library]

Returns: the result of preprocessing and evaluating *obj*

```
(define execute
    (lambda (x)
        (eval (prep x))))
```

```
(execute '(plus 3 4))      =>    7
(define x 3)               =>    x
(define y 'x)              =>    y
y                          =>    x
(execute y)                =>    3
```

(**prep** *obj*) [library]

Returns: the preprocessed form of *obj*

Preprocessing includes macro expansion and currying. *Prep* is often used to test out macro definitions, although its results are sometimes hard to read since it expands all macros and performs currying.

```
(prep 'a)                        =>   a
(prep '(car a))                  =>   (car a)
(prep '(lambda (x y) x))         =>   (lambda x (lambda y x))
(prep '(cons x y))               =>   ((cons x) y)
(prep '(list 1 2 3))             =>   ((cons 1) ((cons 2) (cons 3 nil)))
```

(eval *obj*) [primitive]

Returns: the result of parsing and evaluating *obj*

Obj is not preprocessed by *eval*. *Execute* should normally be used as it preprocesses *obj* first.

```
(eval '((+ 3) 4))              =>   7
(eval (prep '(plus 3 4)))      =>   7
```

(**apply** *closure list*) [library]

Returns: the result of applying *closure* to the argument list *list*

Apply takes *closure* and applies it to the first element of *list*, applies this result to the second element, etc., taking account of *currying*. If *list* is *nil*, *apply* simply applies *closure* to *nil*.

The definition of *apply* in *C*-Scheme does not use eval!:

```
(define apply
    (lambda (f args)
        (if (null args)
            (f nil)
            (iterate loop ((f f) (args args))
                (if args
                    (loop (f (car args)) (cdr args))
                    f))))))

(apply + (list 1 2))              =>   3
(apply (lambda (x) x) '(a))       =>   a
```

(keyboard-interrupt *closure***)** [primitive]

- Returns: closure

Changes the keyboard interrupt service routine to *closure*. The next keyboard interrupt (caused with the BREAK key) is handled by *closure*. That is, when an interrupt occurs *closure* is invoked with *nil*. As with timer interrupts (see §6), if *closure* returns normally the flow of control is not altered. This routine can be set up to provide a mechanism for terminating loops, exiting the system, etc.

Note that the routine must be explicitly reset with a new call to *keyboard-interrupt* after each interrupt.

(collect *n closure***)** [primitive]

Returns: closure

Causes the garbage collector to be invoked after *n* segments have been allocated (segments are currently 4096 words long). *Closure* is then invoked in the manner of timer or keyboard interrupts. *Collect* must be explicitly reinvoked each time interrupt occurs. No garbage collection occurs if *collect* is not called. *Collect* is invoked initially by the system so user programs will almost never need to use *collect*.

The system performs this call during initialization:

```
(define **collect_segments** 100)

(collect **collect_segments**
    (lambda f ()
        (collect **collect_segments** f)))
```

This allows the user to change the number of segments allocated between collections by changing the value of **collect-segments**.

(segments) [primitive]

Returns: the number of segments currently in use

Useful for customizing collection.

(exit) [primitive]

Returns: doesn't return

Exit closes all files and exits to the operating system.

(error *string obj***)** [primitive]

Returns: does not return to caller

Calls the internal *C*-Scheme error handler with *string* as the diagnostic message and *obj* as the "offending expression." The *C*-Scheme error handler prints *string* on the standard error *stderr* and if *obj* is not *nil* prints it as well. Control is returned to the *C*-Scheme toplevel.

```
(define max
   (lambda (x y)
      (cond
         ((not (number? x))
              (error "max: argument 1 not a number" x))
         ((not (number? y))
              (error "max: argument 2 not a number" y))
         ((>= x y) x)
         (y))))
```

14. The *C*-Scheme Reader

This section describes the *C*-Scheme reader by giving the scanning algorithm. Currently, no mechanism exists for changing or extending the reader.

The top level of the reader first skips all white space (blanks, tabs and newlines). The next character it sees determines the structure it will build:

((left paren) begins the reading of a list. The reader is called recursively until a right paren is found; the expressions read become the elements of a list. If no expressions are read before a right paren is found, the symbol *nil* is returned. If a dot (.) is read after one or more expressions have been read, exactly one expression must follow before a right paren. This expression is taken as the last *cdr* in the list being built. If no dot is read, the last *cdr* is taken to be nil.

" (double quote) begins the reading of a string. The sequence of characters following the double quote and before the next double quote is made into a string.

[(left bracket) begins the reading of a vector. The reader is called recursively until a right bracket is found; the expressions read become the elements of a vector. If no expressions are read before a right bracket is found, an empty vector is formed.

' (single forward quote) begins the reading of a quote special form. The reader is called recursively to obtain one expression and a list of the symbol *quote* and the expression is returned. For example, *'hithere* is translated to *(quote hithere)*.

` (back quote) begins the reading of a back-quote expression. Back-quote expressions are particularly helpful. Back-quote is used in concert with commas and at-signs to facilitate the construction of complex but regular data structures from templates. Back-quote expressions are especially useful in writing preprocessor macros (see the examples in the preprocessor section). Rules for back-quote expressions follow, but the examples at the end of this section will probably be easier to understand.

`exp, where *exp* contains no subexpressions starting with a comma, is equivalent to *'exp*.

',*exp* is equivalent to *exp*.

',@*exp* is an error

'*exp*, where *exp* contains subexpressions starting with commas, is equivalent to '*exp* except that when it is evaluated:

- each subexpression starting with a comma and no at-sign (@) is evaluated
- each subexpression starting with a comma followed by an at-sign is evaluated and spliced into the structure

If none of these special characters is seen, all characters up to the next white-space character, left or right paren, or left or right bracket are collected in a buffer. Any of these special characters may be forced into the buffer by preceding it with a back-slash (\), e.g. *par\(\)ens*, *hi\ there*. Two back-slashes (\\) will enter one back-slash into the buffer.

If the collected characters can be parsed as a number, a number is returned, otherwise a symbol is returned.

The syntax for numbers includes only integers consisting of the digits 0 through 9 optionally preceded by a plus (+) or minus (-) sign.

Examples:

```
"this is a string"           =>   "this is a string"
this_is_a_symbol             =>   this_is_a_symbol
this\ is\ also\ a\ symbol    =>   this is also a symbol
this_is_too123               =>   this_is_too123
89this_is_too!               =>   89this_is_too!
nil                          =>   nil
+1                           =>   1                  (a number)
1+                           =>   1+                 (a symbol)
-9372                        =>   -9372              (a symbol)
(a  b . c)                   =>   (a b . c)
(a b . (c))                  =>   (a b c)
(a b . (c d))                =>   (a b c d)
(a b . nil)                  =>   (a b)
(a b c)                      =>   (a b c)
()                           =>   nil
((a b) (c d) 4)              =>   ((a b) (c d) 4)
[ab cd (abc) 3]              =>   [ab cd (abc) 3]    (vector)
[]                           =>   []                 (empty vector)
'(a b c d e)'                =>   (quote (a b c d e))
```

```
`(a b c d e)                =>   (quote (a b c d e))
`(a b ,(+ 3 4) d e)         =>   (quote (a b 7 d e))
`(a b ,(list 1 2) d e)      =>   (quote (a b (1 2) d e))
`(a b ,@(list 1 2) d e)     =>   (quote (a b 1 2 d e))
`,(plus 3 4)                =>   (plus 3 4)
```

15. Allocator

The allocator performs the dynamic creation of *C*-Scheme data objects. Since there is no way to delete objects explicitly, a garbage collector works with the allocator to remove unreachable objects from the system.

The allocator uses a segmented heap layout, with each segment holding exactly one type of data, such as *cons cells* or *symbols*. The type of the data in a particular segment is coded into a segment table, one byte per segment. This allows for 256 different data types, more than enough for now (there are currently about 10 types).

More than one segment may hold data objects of the same type. If the allocator tries to allocate an object which will not fit in the current segment, it finds another. Objects can cross segment boundaries if the segments are contiguous. In fact, the allocator allows objects to occupy more than one segment by finding enough contiguous segments to hold the object. This means there is no limit on the size of objects (vectors, for example) save the limits imposed by the operating system on the maximum virtual memory size.

The garbage collector employs an iterative copying algorithm [Baker 1978]. When collection begins, all existing (non-empty) segments are marked as part of the old space. All reachable objects are reallocated by calling the allocator (which always marks segments it obtains as part of the new space) and a *forwarding address* is left in the old object. Any pointers to a copied object are updated using the forwarding addresses. When collection is complete the segments marked as part of the old space are marked empty, and thus become eligible for reallocation.

It is not a real-time collector, nor does it use any of the data compaction strategies such as *cdr-coding*. The real-time strategies involve pushing the collection forward slightly every time an object is allocated or accessed. For efficiency reasons this necessitates microcode support for many of the most common primitives (such as cons, car, cdr, etc.). The target machine (a Vax 11/780) does not allow microcode to be changed dynamically so microcode support is not possible.

16. Interpreter Kernel

This section describes the implementation of the *interpreter kernel*. Familiarity with the host language, *C*, is assumed and the code for the kernel is given at the end of this section.

The interpreter kernel consists of a parser and evaluator for a small subset of *C*-Scheme (hereafter called the *kernel language*). BNF for the kernel language is given in the table below. All features of *C*-Scheme not supported by the kernel language are provided by the preprocessor, macros and functions.

```
<exp>                  ::=   <constant>
                           | <identifier>
                           | <special form>
                           | <combination>
<constant>             ::=   <integer> | nil | <string> | <vector>
<identifier>           ::=   <symbol>
<special form>         ::=   (quote <object>)
                           | (if <exp> <exp> <exp>)
                           | (lambda <symbol> <exp>)
                           | (change! <symbol> <exp>)
                           | (prog2 <exp> <exp>)
<combination>          ::=   (<exp> <exp>)
```

The most significant difference from full *C*-Scheme is the kernel language's lack of multiple-argument functions. All function definitions and applications are curried by the preprocessor.

There are no syntactic extensions in the kernel language; macro expansion is done by the preprocessor.

The parser transforms programs in the *kernel language* into machine instructions, called *i-codes*, for a virtual machine which is supported on the real machine by *evaluator functions* and the evaluator's main loop.

16.1 Evaluator Functions

The evaluator consists of a main driver and a set of *evaluator functions*. These evaluator functions are the implementation of machine instructions for a simple virtual computer (referred to as the *kernel machine*).

The kernel machine is controlled by the evaluator's main loop. The primary activity of the main loop is to get the next machine instruction and call the evaluator function specified by the operator field of the instruction.

The evaluator functions operate on the kernel machine's four registers. These are the accumulator (*accum*), the current environment pointer (*curenv*), the control stack pointer (*cstack*) and the current instruction (*evalpc*).

The evaluator functions support the run-time evaluation necessary for special forms and function application. Primitives such as *car* and *cdr* are also evaluator functions.

The return value of an evaluator is either another instruction to execute or *nil*. Operands of instructions normally specify the instruction to perform next. If *nil* is returned, the evaluator "pops" the control stack (*cstack*) to obtain the next instruction.

The most trivial evaluator function, *Equote*, implements the *quote* special form. It consists of two lines of code:

```
Equote() {
      accum = DATA1(evalpc);
      return DATA2(evalpc);
}
```

This places the first operand of the current instruction in the accumulator and returns the second operand as the next instruction to execute (the second operand may of course be *nil*).

16.2 I-codes

The machine instructions are called *i-codes*. Each *i-code* has three fields: an operator and two operands. The operator is the physical address in memory of the evaluator function for the instruction, and is extracted from the instruction with the function *CODE*. The operands are *C*-Scheme pointers (represented by *C* ints). They are accessed with the functions *DATA1* and *DATA2*. Here is the *i-code* structure in *C* and the access functions (*C* macros):

```
typedef struct {
      int (*code)();
      int data1,data2;
} icode_object;

#define CODE(x) ((icode_object *)(x))->code
#define DATA1(x) ((icode_object *)(x))->data1
#define DATA2(x) ((icode_object *)(x))->data2
```

16.3 Parser

The goal of the parser is to reduce kernel language input to an *i-code* tree. The strategy loosely follows Steele's Scheme compiler [Steele 1977]. The parser uses recursive descent and generates its output in a single pass.

The parser performs several tasks as it generates code. The most important are:

- resolve local identifier references
- build a threaded structure using continuations
- optimize environment saving
- optimize certain types of combinations

Because of the lexical scoping of identifiers, the parser can determine exactly how far down in the run-time environment an identifier's value will be. At run-time all that is done is to *cdr* down *curenv* this distance to find the value.

For most *C*-Scheme expressions it is obvious what expression will be evaluated next (the *continuation*). The parser threads the code, by placing within each instruction the instruction to be evaluated next. For some expressions it is impossible to tell what will be next. For example, when parsing a lambda expression, the parser cannot determine what the continuation of the function body will be. In this case the next instruction is set to *nil* (at run time this will cause the next instruction to be taken from the control stack).

By monitoring environment usage during the parsing of an expression, the parser avoids saving the environment unnecessarily at run time. The parser does this with two parameters, *s* and *u*. *S* is a flag to the parser to say whether the environment must be saved if evaluation of the expression destroys it. *U* is a read/write parameter (in *C*, a pointer to an int). It is set by the parser if the expression being parsed uses the environment. Thus, if evaluation of an expression requires that two subexpressions be evaluated, the second is parsed first. If the return value of *u* is true, *s* is set in the parsing of the first.

The only way the current environment is destroyed is when a closure is applied to an argument. Then, the current environment is changed to the closed environment. The parser determines when this can happen and generates the appropriate code if the environment is needed.

In general the function expression of a combination may be quite complex. However, the parser recognizes certain forms in the function position and generates optimized code for these cases. If the function expression is a *lambda* form the parser does not force a closure to be made. Instead of creating a closure it generates code which will merely add the argument to the environment and execute the body of the *lambda*.

During constant propagation the preprocessor (see §17) may create combinations which have a closure or primitive in the function position (as opposed to a symbol or another form). If so, the parser uses the function as the continuation when it parses the argument. If the combination's continuation is non-*nil* it must either be saved on the control stack or within the function itself (possible only with primitives).

16.4 Code for the Interpreter Kernel

The code for the parser is listed first, followed by the code for the evaluation functions and the code for the evaluator's main loop. The interrupt handling code is included as well. The comments in the code should serve to clarify some of the parsing and evaluation strategies.

16.4.1 Parser

```
/*    parse(p)

the external interface to the parsing routines.  All it does is
call parse1 with p, a nil environment, a nil continuation, 0 for
the save flag, and the address of a cell which it ignores

*/
int parse(p) int p; {
        int ignore;

        return parse1(p,nil,nil,0,&ignore);
}

/*    parse1(p,e,c,s,u)
      p is the expression to parse
      e is a list of identifiers lexically visible to p, starting
           with the innermost visible.
      c is the instruction to execute next (continuation)
      s is the save flag: if true, save the environment
      u is a return parameter which must be set if any identifiers from
           the environment are accessed

      parse1 does a case statement on type.
      If the argument is anything but an identifier or a list, a
      Equote i-code is returned.
```

If the argument is an identifier, parse calls the help function lookup determine the location of the identifier in the environment. If lookup returns a negative number, the identifier was not found, so an Eid i-code is produced. Otherwise an Eaccess i-code is returned.

If the type is type_cons with a symbol in the car position, parse1 calls magic_look to determine if the symbol is the keyword for a magic (special) form. Magic_look scans the magic keyword table and returns the index of the symbol or -1 if the symbol is not a keyword. Addresses of the functions Pquote, Plambda, Pif, Pprog and Psetq are in a magic funs table, and if magic_look returns a non-negative value, this table is indexed by the magic_look value and the function found there is invoked.

If the car of the list is not a symbol or magic_look returns a negative value, parse calls Pappl.

```
*/

int parse1(p,e,c,s,u) register int p,e,c; int s,*u; {
        register int n;

        if (!p) return icode(Equote,p,c);

        switch (TYPE(p)) {
                default: error("parse: invalid argument type",nil);
                case type_icode: /* return p? */
                case type_clos:
                case type_vect:
                case type_str:
                case type_file:
                case type_inum:
                        return icode(Equote,p,c);
                case type_sym:
                        n = lookup(p,e);
                        if (n < 0) return icode(Esym,p,c);
                        *u = 1;
                        return icode(Eaccess,inum(n),c);
                case type_cons:
                        if (!CAR(p)) error("parse: invalid syntax",p);
                        n = magic_look(CAR(p));
                        return n >= 0 ?
                                magfuns[n](CDR(p),e,c,s,u) :
                                Pappl(CAR(p),CADR(p),e,c,s,u);
        }
}

static int lookup(sym,e) register int sym, e; {
        register int i = 0;

        while (e) {
                if (CAR(e) == sym)
                        return i;
                e = CDR(e);
                i++;
        }
        return -1;
}
```

```
int magic_look(x) register int x; {
        register int i;

        if (SYMBOLP(x))
                for(i=0; i <= magic_max; i++)
                        if (magsyms[i] == x) return i;

        return -1;
}

/*   Pquote(p,e,c,s,u)
     p is the cdr of the quote form

     Pquote just returns an Equote i-code with the quoted expression in
     data1 and the continuation in data2.
*/

static int Pquote(p,e,c,s,u) register int p; int e,c,s,*u; {

        return icode(Equote,CAR(p),c);
}

/*   Plambda(p,e,c,s,u)
     p is the cdr of the lambda form

     Plambda sets u to true (it will use the environment since it must
     close it with the body)
     The body is parsed in the environment with the parameter added on,
     a nil continuation since it is impossible to tell where it will
     end up, the save flag off since no-one will need the new
     environment, and u passed along for the ride, even though its fate
     is already determined.

     An Elambda i-code is returned with the parsed body in data1 and
     the continuation in data2.
*/

static int Plambda(p,e,c,s,u) register int p; int e,c,s,*u; {

        *u = 1;
        return icode(Elambda,parse1(CADR(p),cons(CAR(p),e),nil,0,u),c);
}
```

```
/*   Pif(p,e,c,s,u)
     p is the cdr of the if form

     Pif parses the then and else expressions with the same environment,
     continuation and save flags, and the address of a local variable
     used.

     Pif sets u if used is set.

     The test expression is parsed with the same environment, an Eif
     i-code (data1 = parsed then expression, data2 = parsed else
     expression) for the continuation, save flag if s or used, and u.

     Pif returns the parsed test expression.
*/

static int Pif(p,e,c,s,u) register int p,e; int c; register int s,*u; {
        register int p1,p2; int used = 0;

        p1 = parse1(CADR(p),e,c,s,&used);
        p2 = parse1(CADDR(p),e,c,s,&used);
        *u |= used;
        return parse1(CAR(p),e,icode(Eif,p1,p2),s|used,u);
}

/*   Pprog(p,e,c,s,u)
     p is the cdr of the prog2 form

     Pprog parses the second expression first, passing the address of a
     local variable used.

     If used is set, Pprog sets u.

     Pprog returns the first expression parsed with the parsed second
     expression as the continuation and the save flag or'd with used.

     Note that Pprog generates no i-codes.

     Also note that the parsing is done in reverse of the desired order
     of execution.
*/

static int Pprog(p,e,c,s,u) register int p,e; int c; register int s,*u; {
        register int p1; int used = 0;

        p1 = parse1(CADR(p),e,c,s,&used);
        *u |= used;
        return parse1(CAR(p),e,p1,s|used,u);
}
```

```
/*   Psetq(p,e,c,s,u)
     p is the cdr of the change! form

     Psetq returns value expression parsed with a continuation to
     either change the global value or the local value of the symbol,
     depending upon the return value of lookup.

     Psetq must require that the value expression save the environment
     when it will change the local value of the symbol.
*/

static int Psetq(p,e,c,s,u) register int p,e; int c,s; register int *u; {
        register int p1 = CAR(p); register int n = lookup(p1,e);

        if (n >= 0) {
                *u = 1;
                return parse1(CADR(p),e,icode(Esetq,inum(n),c),1,u);
        }
        else
                return parse1(CADR(p),e,icode(Edefine,p1,c),s,u);
}

/*   Pappl
     x is the function expression
     y is the argument expression

     e, c, s, u same as in parse1

     Pappl tries to be intelligent about certain combinations which
     occur frequently.  The first is when the function expression is a
     closure, the second when it is an icode (these are generated by
     the preprocessor when it is in **expand-constants** mode.
     The third common form is ((lambda id body) arg).

     If Pappl finds a closure it figures out which evaluation function
     to use and returns the parsed argument with a continuation that
     invokes the argument.  The evaluation function depends on whether
     the continuation is nil and whether the environment must be saved.
     If we are really lucky, the environment needn't be saved and the
     continuation is nil, so we use just the function itself as the
     continuation.

     If Pappl finds a primitive (represented as an i-code) in the car
     position the primitive is the argument's continuation.  All
     primitives reserve the data2 field for the continuation, so if
     Pappls continuation is not nil, a copy of the primitive with this
     continuation in data2 is made.

     For the third form, Pappl parses body in the environment with
     id on the front, creates an i-code which will put the accum on
     front of the curenv argument at run-time and passes this along as
     the continuation for the argument expression.  Of course it must
     worry about the environment and the continuation being saved or
     not.

     For any other application, both the function expression and
     argument expression are fully parsed, similarly to prog2 or if,
     and the evaluation function again depends upon whether the
     continuation is nil and whether the environment need be saved.
```

```
*/

int Pappl(x,y,e,c,s,u) register int x,y; int c,e,s,*u; {
    int used = 0; register int (*Efun)();

    switch (TYPE(x)) {
        case type_clos:
                Efun = c?(s?Ea2cs:Ea2c):(s?Ea2s:NULL);
                return parsel(y,e,Efun?icode(Efun,x,c):x,s,u);
        case type_icode:
                return parsel(y,e,c?icode(CODE(x),DATA1(x),c):x,s,u);
        case type_cons:
                if (CAR(x) == lambda_id) {
                        x = CDR(x);
                        if (c && s) c = icode(Erestore,c,NULL);
                        x = parsel(CADR(x),cons(CAR(x),e),c,s,&used);
                        *u |= used;
                        return parsel(y,e,icode(s?Ea1s:Ea1,x,NULL),used|s,u);
                }
        default:
                y = parsel(y,e,NULL,s,&used);
                *u |= used;
                Efun = c?(s?Ea3cs:Ea3c):(s?Ea3s:Ea3);
                return parsel(x,e,icode(Efun,y,c),used|s,u);
    }
}
```

16.4.2 Evaluator Functions

```
/*  Equote

    Equote sets accum to the quoted expression and returns the
    continuation.
*/

static int Equote() {
        accum = DATA1(evalpc);
        return DATA2(evalpc);
}

/*  Eaccess

    Eaccess calls the help function nth to retrieve the value of the
    identifier in curenv.
*/

static int Eaccess() {
        register int p = nth(curenv,INUM(DATA1(evalpc)));
        accum = CAR(p);
        return DATA2(evalpc);
}

static int nth(l,n) register int l, n; {
        while (n) l = CDR(l), n--;
        return l;
}
```

```
/*   Esym

     Esym sets accum to the value of the symbol, and causes an error if
     the symbol is unbound.
*/

static int Esym() {
        accum = VALUE(DATA1(evalpc));
        if (accum == unbound)
                error("undefined symbol",DATA1(evalpc));
        return DATA2(evalpc);
}

/*   Elambda

     Elambda makes a closure with curenv and retuns its continuation.
*/

static int Elambda() {
        accum = closure(DATA1(evalpc),curenv);
        return DATA2(evalpc);
}

/*   Eif

     Eif tests the accum.  If non nil returns the then-part as its the
     next instruction, otherwise the else-part
*/

static int Eif() {
        return accum ? DATA1(evalpc) : DATA2(evalpc);
}

/*   Esetq

     Esetq changes the local binding of a variable, calling nth
     for the cons cell where the value lies.
*/

static int Esetq() {
        register int p = nth(curenv,INUM(DATA1(evalpc)));
        CAR(p) = accum;
        return DATA2(evalpc);
}

/*   Edefine

     Edefine changes the global value of a symbol
*/

static int Edefine() {
        VALUE(DATA1(evalpc)) = accum;
        return DATA2(evalpc);
}
```

```
/*    Ea1, Ea1s, Erestore

      Ea1 and Ea1s place accum on the front of curenv and return the body
      of the lambda expression as the next instruction.  Erestore is
      needed to explicitly restore the environment since the evaluator loop
      does not look at cstack as long as the evaluator functions return
      non-nil instructions, which happens in this case.

      Ea2s pushes the current environment before returning
*/

int Ea1() {
        curenv = cons(accum,curenv);
        return DATA1(evalpc);
}

int Ea1s() {
        pushc(curenv);
        curenv = cons(accum,curenv);
        return DATA1(evalpc);
}

int Erestore() {
        popc(curenv);
        return DATA1(evalpc);
}

/*    Ea2c, Ea2s, Ea2cs

      Ea2c and Ea2cs push the continuation
      Ea2s and Ea2cs push curenv

      All return the closure to be evaluated
*/

int Ea2c() {
        pushc(DATA2(evalpc));
        return DATA1(evalpc);
}

int Ea2s() {
        pushc(curenv);
        return DATA1(evalpc);
}

int Ea2cs() {
        pushc(DATA2(evalpc));
        pushc(curenv);
        return DATA1(evalpc);
}
```

```
/*    Ea3, Ea3s, Ea3c, Ea3cs

      Ea3c and Ea3cs push the continuation
      Ea3s and Ea3cs push curenv

      The accum must be an i-code or closure, and is pushed onto the
           control stack
      The argument expression is returned as the next instruction.
*/

int Ea3() {
        register int ty;
        if (!accum || ((ty = TYPE(accum)) != type_clos && ty != type_icode))
                error("apply: invalid function",accum);
        pushc(accum);
        return DATA1(evalpc);
}

int Ea3s() {
        register int ty;
        if (!accum || ((ty = TYPE(accum)) != type_clos && ty != type_icode))
                error("apply: invalid function",accum);
        if (ty == type_clos) pushc(curenv);
        pushc(accum);
        return DATA1(evalpc);
}

int Ea3c() {
        register int ty;
        if (!accum || ((ty = TYPE(accum)) != type_clos && ty != type_icode))
                error("apply: invalid function",accum);
        pushc(DATA2(evalpc));
        pushc(accum);
        return DATA1(evalpc);
}

int Ea3cs() {
        register int ty;
        if (!accum || ((ty = TYPE(accum)) != type_clos && ty != type_icode))
                error("apply: invalid function",accum);
        pushc(DATA2(evalpc));
        if (ty == type_clos) pushc(curenv);
        pushc(accum);
        return DATA1(evalpc);
}
```

16.4.3 Evaluator Control Loop

```
/*    eval(p)

      eval takes an instruction as input, sets evalpc to this
      instruction and curenv to nil.  cstack and curenv may or may not
      have been set to nil, so eval can be called with a non empty
      continuation (for error throws, etc)

      Every 100 times through the loop eval checks such things as timer
      and keyboard interrupts and invokes the collector if necessary.
```

```
     When evalpc and cstack are empty, eval quits.

     Three types of things can be on the control stack: i-codes,
     closures and lists.  Anything else causes an error.
     i-code:
          call the evaluator function
     closure:
          tack the accum onto the front of the closed environment,
          making the new curenv.  The next instruction is the body.
     list:
          must be an environment saved earlier.  Restore it to curenv.
*/

eval(p) int p; {
        register int ticks = 1 /* set to happen first time */;

        evalpc = p;
        curenv = NULL;
        while (1) {
                if (!--ticks) {
                        ticks = ticks_per_period;
                        if (timer && !--timer) timer_handler();
                        if (gc_segments && segments_allocated >= gc_segments)
                                gc_handler();
                        if (signal_flag) signal_handler();
                }
                if (!evalpc) {
                        if (!cstack) return;
                        popc(evalpc);
                        continue;
                }
                switch(TYPE(evalpc)) {
                        case type_cons:
                                curenv = evalpc;
                                evalpc = nil;
                                break;
                        case type_clos:
                                curenv = cons(accum,ENV(evalpc));
                                evalpc = BODY(evalpc);
                                break;
                        case type_icode:
                                evalpc = CODE(evalpc)();
                                break;
                        default: error("invalid stack evalpc",evalpc);
                }
        }
}
```

```
/*   Einterrupt
     an evaluator function which restores the registers which are saved
     when an interrupt of any sort occurs
*/

static int Einterrupt() {
        accum = DATA1(evalpc);
        curenv = DATA2(evalpc);
        return NULL;
}

/*   interrupt(x)
     saves the registers and invokes x.  Einterrupt will clean up the
     interrupt if control ever gets back.
*/

static interrupt(x) int x; {
        if (evalpc != NULL) pushc(evalpc);
        pushc(icode(Einterrupt,accum,curenv));
        accum = NULL;
        evalpc = x;
        return;
}

/*   timer_handler
     interrupts with timer_closure (set up with a call to
     enable-interrupt)
*/

timer_handler() {
        interrupt(timer_closure);
        timer_closure = NULL;
        return;
}

/*   gc_handler
     collects and interrupts with gc_closure (set up by a call to
     collect)
*/

gc_handler() {
        gc();
        interrupt(gc_closure);
        gc_closure = NULL;
        return;
}
```

```
/*    sigint_handler
      this is the routine passed the the unix sigset function.  It sets
      signal_flag.  If signal_flag was already set, an error is
      generated with the assumption the interpreter wasn't getting back
      to the control loop.  Note that since the flag isn't reset in this
      case the signal will still happen after the error is caused.  This
      part still bothers me.
*/

static void sigint_handler() {
        if (signal_flag) error("",NULL);
        signal_flag = 1;
        return;
}

/*    signal_handler

      The real handler, invoked from the evaluator's control loop when
      signal_flag is set

      If signal_closure has not been set up (by a call to
      keyboard-interrupt), an error is generated, otherwise
      signal_closure is invoked as an interrupt routine.
*/

signal_handler() {
        signal_flag = 0;
        if (signal_closure == NULL) error("Interrupted.",NULL);
        interrupt(signal_closure);
        signal_closure = NULL;
        return;
}

/*    schsig_init
      sigset is documented in the Berkeley version 4 programmer's manual
*/

schsig_init() {
        signal_flag = 0;
        sigset(SIGINT,sigint_handler);

        timer = 0;

        gc_segments = 0;
}
```

17. Preprocessor

The *C*-Scheme preprocessor controls macro expansion, prepares special forms for the parser, curries function applications and definitions, and propagates constants. The output of the preprocessor is a program in the *kernel language* (§ 16).

17.1 Currying

The *C*-Scheme kernel only supports functions of one argument. A function of more than one argument is transformed by the preprocessor into a function which takes its arguments one at a time. A two-argument function transforms into a one-argument function which returns a function of one-argument. This transformation is called *Currying* [Church 1941], [Rosser 1982], [Stoy 1977].

Not only must function definitions be curried, but the applications must be expanded as well. Currying of function arguments associate to the left, that is (f a b) is equivalent to ((f a) b).

Currying is carried out as the last step of the preprocessor, so it is entirely transparent to the user (that is, it is as if the parser or evaluator was performing the currying).

Examples:

```
(prep '(lambda (x y) body))              =>    (lambda x (lambda y body))
(prep '(+ x y))                          =>    ((+ x) y)
(prep '((lambda (x y) body) 3 4))        =>    (((lambda x
                                                      (lambda y body))
                                                     3)
                                                    4)
(prep '((lambda (x y) (+ x y)) 3 4))     =>    (((lambda x
                                                      (lambda y (+ 3 4)))
                                                     3)
                                                    4)
```

The definition of a function of zero parameters is translated by the preprocessor into a form which the parser accepts as a function of one "invisible" argument. Rather than place a symbol in the argument position of the resulting lambda expression, the preprocessor places *nil*. Thus (lambda () body) translates to itself. The parser will accept the *nil* as the identifier, but will never find it as a local variable since *nil*'s value is constant.

The application of a function to zero arguments is translated to an application of the function to *nil*.

Examples:

```
(prep '(lambda () "hi there"))        =>    (lambda nil "hi there")
(prep '(f))                           =>    (f nil)
(prep '((lambda () "hi")))            =>    ((lambda nil "hi") nil)
```

17.2 Constant Propagation

The *C*-Scheme preprocessor attempts to evaluate any time-independent computation at preprocess time, rather than force it to be recomputed every single time the surrounding expression is evaluated.

The user can declare that the global value of any symbol is *constant*, by placing any non-*nil* value on the property list of the symbol under the property ***constant***. Initially, most of the primitive functions are declared *constant*. Together with any *C*-Scheme object which would normally evaluate to itself (strings, numbers, vectors, etc.) these *constant* symbols seed constant propagation.

In processing the *if* special form, if the *test-part* is found to be constant, its value is determined and the *if* reduces to either the *then-part* or the *else-part*.

In processing a *prog2* special form, if the first expression is constant, the *prog2* reduces to the second expression.

Change! special forms never propagate constants. The effect of a *change!* operation is obviously time-dependent.

Lambda expressions currently do not propagate constants. It is certainly possible to do for certain cases but the exceptions are abundant.

Combinations afford the most opportunity for constant propagation. If both the function expression and the argument expression are constant, the function is applied to the argument, yielding a new constant value (this value is *quoted* if necessary before it is returned). In a curried system this can happen quite often: *(+ 3 x)* will be translated to *((+ 3) x)* which will in turn be translated to *(** closure ** x)* where *** closure *** is the result of applying + to *3*.

The global value of the symbol ***expand-constants*** controls constant propagation. If this value is *nil*, the constants are not propagated (and the ***constant*** property of individual symbols is ignored). This can be useful for debugging.

Examples:

```
(prep '(+ 3 4))                      =>   7
(prep '(if (+ 3 4) 'yes 'no))        =>   'yes
(prep '(prog2 'a x))                 =>   x

(change! **expand-constants** nil)
(prep '(+ 3 4))                      =>   ((+ 3) 4)
(prep '(if (+ 3 4) 'yes 'no))        =>   (if ((+ 3) 4) 'yes 'no)
(prep '(prog2 'a x))                 =>   (prog2 'a x)
```

17.3 Macro Expansion

Macros provide the only means of extending *C*-Scheme's syntax. They are commonly used for:

- providing new control structures,
- abbreviating commonly used structures,
- writing "functions" with more than one argument,
- writing "functions" with optional arguments.

Macros improve the readability of code and serve an important part in the structuring *C*-Scheme code. For example, the macros *let*, *let** and *letrec* are invaluable shorthand for introducing local identifiers and mutual recursion. They are even more invaluable because they essentially give *C*-Scheme a block-structure, similar to Algol. *Case* and *cond* can drastically reduce the amount of if-then-elses in a program.

Macros are favored over additions to *C*-Scheme's set of *special forms*. Keeping the core of the language small allows the interpreter to be simple and fast.

A *syntactic extension*, or macro invocation, is a list with a macro keyword as its first element. A macro keyword is a symbol with a function closure on its property list under the property ***macro***. When the preprocessor sees such an expression, it invokes the function closure on the entire expression (thus, the first element of the argument to the macro function is always the function name itself).

The result returned from the function is sent back to the preprocessor for further processing. However, if the same macro keyword appears in the resulting expression as in the invocation, further macro expansion is disabled. This allows macros to be written with the same name as a special form or function (the *lambda* and *read* macros, for example).

Macros functions must be a function of one argument. They may be defined using the macro *macro*:

(macro *name function*)

This places *function* on *name*'s property list under the property ***macro***. Examples:

```
(macro freeze
    (lambda (m)
        '(lambda nil ,(cadr m))))

(prep '(freeze x))                          =>   (lambda () x)
(freeze "hi")                               =>   ** closure **
(thaw (freeze "hi"))                        =>   "hi"


(macro and
    (lambda (m)
        (if (cddr m)
            '(if ,(cadr m)
                 (and . ,(cddr m)) nil)
            (cadr m))))

(prep '(and a b c))                         =>   (if a (if b c nil) nil)
(and t t nil)                               =>   nil
(and 1 2 3)                                 =>   3


(macro let*
    (lambda (m)
        (let ((x (cadr m)))
            (if x
                '((lambda (,(caar x))
                       (let* ,(cdr x)
                           . ,(cddr m)))
                     ,(cadar x))
                (caddr m)))))

(prep '(let* ((x a) (y b) z))               =>   ((lambda x
                                                       ((lambda y z) b)
                                                      a)
(let* ((x 1) (y (1+ x))) (plus x y))        =>   3
```

17.4 Preprocessing Special Forms

There is a preprocess function for each of the *C*-Scheme special forms. When the preprocessor is loaded, it places the appropriate function on the property list of each special-form keyword under the property ***prep***. When a form is seen (after macro expansion) whose first element is a symbol with this property, the function is invoked with the entire form as its argument. The invocation of these ***prep*** functions is similar to macro expansion, except that these functions are invoked after the regular macro expansion and their result is never re-preprocessed.

The action of the ***prep*** functions for each of the keywords is straightforward:

quote: The form is returned unchanged. Quote is used to introduce data into the system so any interpretation by the preprocessor would be inappropriate.

lambda: First, *prep* is called on the body. If the argument list is initially empty, this is the definition of a zero argument function, and the argument list is changed to (list nil). The *nil* acts as a dummy argument which will not be seen by the parser as a local identifier, since the value of *nil* is always itself.

The argument list and the preprocessed body are passed to a help function, *prep-lambda*.

Prep-lambda simply returns the body when the argument list is empty, otherwise it recursively calls itself with the *cdr* of the argument list and the body. It returns a new lambda expression with the car of the argument list as the identifier and the result of the recursive call as the body.

if: For if, each of the subexpressions is preprocessed. If the test expression turns out to be constant, it is evaluated and one of the preprocessed *then-part* or *else-part* is returned. Otherwise a new *if* expression with the preprocessed expressions is constructed and returned.

prog2: Both of the subexpressions are preprocessed. If the first turns out to be a constant, the second is returned. Otherwise, a *prog2* is built out of the resulting expressions.

change!: *Change!* checks its first argument to make sure it is a symbol. Also, if the global value of ***expand-constants*** is true, the symbol is checked to make sure it does not have the ***constant*** property. In either case an error message is printed. Otherwise, the second expression is preprocessed and a new *change!* expression is formed from the symbol and the result.

Examples:

```
(prep '(quote anything))              =>   (quote anything)
(prep '(lambda () (change! x 3)))     =>   (lambda () (change! x 3))
(prep '(lambda (x y) (+ x y)))        =>   (lambda x (lambda y ((+ x) y)))
(prep '(if a b c))                    =>   (if a b c)
(prep '(if (+ x y) 'yes 'no))         =>   (if ((+ x) y) 'yes 'no)
(prep '(prog2 x y))                   =>   (prog2 x y)
(prep '(prog2 (change! x 3)
           (+ x x)))                  =>   (prog2 (change! x 3) ((+ x) x))
```

17.5 Preprocessor Definition in *C*-Scheme

```
(change! constantp
    (lambda (x)
        (if (atom x)
            (not (symbolp x))
            (eq (car x) 'quote))))

(put 'quote '**prep** (lambda (l) `(quote ,(cadr l))))

(put 'lambda '**prep**
    (lambda (l)
        (let ((e (prep (caddr l))))
            (iterate loop ((v (caadr l)) (vs (cdadr l)))
                `(lambda ,v ,(if vs (loop (car vs) (cdr vs)) e))))))

(put 'if '**prep**
    (lambda (l)
        (let ((x (prep (cadr l))))
            (if (and **expand-constants** (constantp x))
                (if x (prep (caddr l)) (prep (cadddr l)))
                `(if ,x ,(prep (caddr l))  ,(prep (cadddr l)))))))

(put 'prog2 '**prep**
    (lambda (l)
        (let ((x (prep (cadr l))))
            (if (and **expand-constants** (constantp x))
                (prep (caddr l))
                `(prog2 ,x ,(prep (caddr l)))))))

(put 'change! '**prep**
    (lambda (l)
        (let ((id (cadr l)))
            (cond
                ((not (symbolp id))
                     (error "prep: cannot change! non-symbol" id))
                ((and **expand-constants** (get id '**constant**))
        (error "prep: cannot change! id with **constant** property" id))
                (`(change! ,(cadr l) ,(prep (caddr l))))))))
```

```
(change! prep
    (let*
        ((constant (lambda (x) (if (constantp x) x '',x)))
         (build
             (lambda (x y)
                 (if (and **expand-constants** (constantp x) (constantp y))
                     (constant ((eval x) (eval y)))
                     `(,x ,y))))
         (prep_appl
            (lambda (l)
                 (iterate loop ((x (build (car l) (cadr l))) (l (cddr l)))
                     (if l (loop (build x (car l)) (cdr l)) x))))))
        ((lambda prep (ok-macro? e)
            (cond
                 ((and (symbolp e) **expand-constants** (get e '**constant**))
                     (constant (eval e)))
                 ((atom e) e)
                 ((not (symbolp (car e))) (prep_appl (mapcar (prep 't) e)))
                 ((let ((x (and ok-macro? (get (car e) '**macro**))))
                      (if x
                          (let ((after (x e)))
                              (prep
                                  (or (atom after)
                                      (not (eq (car e) (car after))))
                                  after))
                          (let ((x (get (car e) '**prep**)))
                              (if x
                                  (x e)
                                  (prep_appl (mapcar (prep 't) e)))))))))
            't)))
```

18. References

1. Baker, H.G. Jr. (1978) "List Processing in Real Time on a Serial Computer," *Communications of the ACM 21*, 4, pp. 280-294.

2. Church, A. (1941) "The Calculi of Lambda Conversion," *Annals of Mathematics Studies 6*, Princeton University Press.

3. Dwyer, R.A. and Dybvig, R.K. (1981) "A SCHEME for Distributed Processes," Indiana University Computer Science Department Technical Report 107.

4. Fessenden, C., Clinger, W., Friedman, D.P. and Haynes, C. (1983) "⟦Scheme 311⟧ Reference Manual" Indiana University Computer Science Department Technical Report 137.

5. Friedman, D.P. (1974) *The Little Lisper*, Science Research Associates, Inc.

6. Friedman, D.P. and Wise, D.S. (1976) "Cons Should Not Evaluate its Arguments," in *Automata, Languages and Programming*, eds. Michaelson, S., and Milner, R., Edinburgh University Press, pp. 257-284.

7. Georgeff, M.P. (1982) "A Scheme for Implementing Functional Values on a Stack Machine," *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 188-195.

8. Rees, J.A. and Adams, N.I. (1982) "T: A Dialect of Lisp, or LAMBDA: The Ultimate Software Tool, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 114-122.

9. Rosser, J.B. (1982) "Highlights of the History of The Lambda-Calculus" *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 216-224.

10. Steele, G.L. (1977) "Rabbit: A Compiler for Scheme (A Study in Compiler Optimization)," Massachusetts Institute of Technology Artificial Intelligence Memo 474, MIT AI Lab, Cambridge.

11. Steele, G.L. and Sussman, G.J. (1978) "The Revised Report on SCHEME," Massachusetts Institute of Technology Artificial Intelligence Memo 452.

12. Sussman, G.J. and Steele, G.L. (1975) "Scheme: an Interpreter for Extended Lambda Calculus," Massachusetts Institute of Technology Artificial Intelligence Memo 349.

13. Stoy, J.E. (1977) *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, pp. 38-77.

14. Wand, M. (1980) "Continuation-Based Multiprocessing," *Proceedings of the 1980 LISP Conference*, pp. 19-28.

15. Weissman, C. (1968) *Lisp 1.5 Primer*, Dickenson Publishing Co., Belmont, California.

16. Wise, D.S. (1982) "Interpreters for Functional Programming," in *Functional Programming and its Applications*, eds. Darlington, J. Henderson, P. Turner, D.A., Cambridge University Press, pp. 253-280.

19. Index of Forms