Nils M Holm

Scheme 9 from Empty Space

Preface

This textbook contains the complete and heavily commented source code to an interpreter for a broad subset of R5RS Scheme. It covers many essential properties of the language, including:

– automatic memory management
– lexical closures
– tail recursion (tail call elimination)
– syntax transformation (macros)
– arbitrary precision (bignum) arithmetics

The book is not exactly for beginners, because the better part of it consists of code. Although the code is very straight-forward, some familiarity with both C and Scheme is required to follow it.

The code appears in the same order as in the actual source file, so some forward references occur. Expect to read the code at least twice unless you jump to forward references while reading it. If you intend to do so: the index contains the page number of each function, procedure, constant, and variable declared anywhere in the program.

Because the text contains the *whole* code of the interpreter, there will be advanced parts and easy parts. Feel free to skip parts that you consider trivial. Parts that are complex can often be identified by the larger amount of annotations and figures that surround them.

You might even want to download a copy of the source code from the **Scheme 9 from Empty Space** section at `http://t3x.org` and study the actual code and the book at the same time.

The text uses different fonts for different types of code. In annotations, `C code` prints in a typewriter font, **`Scheme code`** in boldface typewriter font, and **abstract variables** use a boldface font. Listings of Scheme and C code are both rendered in a light typewriter font. Furthermore the following notations are used:

`form1 ===> form2`	denotes the transformation of **form1** to **form2**
`type1 --> type2`	denotes a procedure that maps **type1** to **type2**
`(p t1 ...) --> type`	denotes a procedure **p** of types **t1**... --> **type**
`f => v`	denotes the evaluation of a form **f** to a value **v** (“**f** evaluates to **v**”)

Annotations (which are renderd with the same font as this text) always precede the code that they describe – mostly C functions or Scheme procedures. There are also some inline annotations which are ordinary C or Scheme comments.

Feel free to use, modify and redistribute the Scheme 9 from Empty Space code, as long as you keep intact all copyright notices in the code and documentation. When distributing modified copies, please mark them as such.

Nils M Holm, July 2007

Contents

C Part 7

Miscellanea 7
Memory Management 14
The Reader 22
The Printer 31
Interlude 35
Evaluator: Variable Lookup 39
Evaluator: Special Form Handling 41
Interlude: Bignum Arithmetics 52
Evaluator: Primitive Handlers 62
Evaluator: Evaluator 81
Read Eval Print Loop 99
Initialization 100

Scheme Part 107

Library Procedures 107
Syntax Expander 115
Quasiquote Expander 119
Library Syntax 120

List of Figures 121

Index 123

C Part

Scheme 9 from Empty Space (S9fES) is an interpreter for a broad subset of R5RS Scheme. It is written in ANSI C (C89) and Scheme. The S9fES code strives to be comprehensible rather than fast. It is aimed at people who want to study the implementation of Scheme (in a language other than Scheme).

Miscellanea

```
/*
 * Scheme 9 from Empty Space
 * Copyright (C) 2007 Nils M Holm <nmh@t3x.org>
 */
```

S9fES is a tree-walking interpreter using deep binding and hashed environments. It employs a constant-space mark and sweep garbage collector with in-situ string and vector pool compaction. Memory pools grow on demand. The interpreter uses bignum integer arithmetics exclusively.

The complete C part of the source code is contained in the file `s9.c`. When this file is compiled with the `-DNO_SIGNALS` option, POSIX signal handling is disabled. In this case, sending `SIGINT` or `SIGQUIT` to the interpreter will terminate it. With signals enabled, sending `SIGINT` will return to the read-eval-print loop (REPL), and `SIGQUIT` will shut down the interpreter.

```
/*
 * Use -DNO_SIGNALS to disable POSIX signal handlers.
 */
```

When `Debug_GC` is set to 1, additional information will print during garbage collections (GCs).

```
int     Debug_GC = 0;

#define VERSION "2007-06-29"

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <limits.h>
#ifdef NO_SIGNALS
 #define signal(sig, fn)
#endif
```

`TEXT_LEN` is the maximal length of a symbol name, string literal or integer literal. `MAX_PORTS` is the maximal number of ports (files) that can be open at the same time. `SEGMENT_LEN` is the size (in nodes) [1] by which the memory pools grow when free memory runs low. `HASH_THRESHOLD` is the minimal number of variables that a rib must contain to be hashed.

```
#define TEXT_LEN        1024
#define MAX_PORTS       32
```

1 A “node” is a rather abstract unit, which will be explained later. Its actual size depends on the host platform.

```
#define SEGMENT_LEN     32768
#define HASH_THRESHOLD  64
```

MEMORY_LIMIT_KN is the maximal amount of nodes that the interpreter will allocate before bailing out with a fatal error notice. Setting this limit to 0 will disable the limit.

```
/* Hard memory limit in K-nodes, 0 = none */
#define MEMORY_LIMIT_KN 1024
```

Bignum arithmetics are performed in terms of a unit called an “integer segment”. The size of an integer segment equals the size of a C `int` on the underlying platform.

INT_SEG_LIMIT is the largest number that can be stored in an integer segment plus one. It is equal to the largest power or 10 that can be stored in an int.DIGITS_PER_WORD is the number of decimal digits that can be stored in an integer segment. The following code tries to guess the size of an integer segment and sets the above constants accordingly.

```
#if INT_MAX >= 1000000000000000000              /* 64-bit */
 #define DIGITS_PER_WORD        18
 #define INT_SEG_LIMIT          1000000000000000000
#elif INT_MAX >= 1000000000                     /* 32-bit */
 #define DIGITS_PER_WORD        9
 #define INT_SEG_LIMIT          1000000000
#elif INT_MAX >= 10000                          /* 16-bit */
 #define DIGITS_PER_WORD        4
 #define INT_SEG_LIMIT          10000
#endif
```

The following flags drive the garbage collector. They are stored in the “tag” field of a node. (Except for the port flags, which are stored in port structures.)

AFLAG signals that the given node is the root of an atomic object. Everything that is not a cons cell is an atom. The “car” field of an atomic node holds the type of the atom, and its “cdr” field points to the value of the atom. MFLAG (mark flag) and SFLAG (state flag) are used together to form the three states of the garbage collecting automaton. VFLAG (vector flag) is similar to AFLAG, but the cdr field of a vector node points to vector space rather than node space. UFLAG indicates that a port is in use and cannot be closed by the collector. LFLAG locks the UFLAG of a port.

```
/* GC flags */
#define AFLAG   0x01    /* Atom, Car = type, CDR = next */
#define MFLAG   0x02    /* Mark */
#define SFLAG   0x04    /* State */
#define VFLAG   0x08    /* Vector, Car = type, CDR = content */
#define UFLAG   0x10    /* Port: used flag */
#define LFLAG   0x20    /* Port: locked (do not close) */
```

These are the states of the evaluator. They will be explained in detail when we encounter them later in the text.

```
enum EVAL_STATES {
        MATOM,  /* Processing atom */
        MARGS,  /* Processing argument list */
        MBETA,  /* Beta-reducing */
        MIFPR,  /* Processing predicate of IF */
        MSETV,  /* Processing value of SET! */
        MDEFN,  /* Processing value of DEFINE */
        MDSYN,  /* Processing value of DEFINE-SYNTAX */
        MBEGN,  /* Processing BEGIN */
        MCONJ,  /* Processing arguments of AND */
        MDISJ,  /* Processing arguments of OR */
        MCOND   /* Processing clauses of COND */
};
```

“Special values” are represented by negative integers, so they cannot be offsets into the node array. The special values `NIL`, `TRUE`, `FALSE`, and `ENDOFFILE` represent the Scheme atoms **`()`**, **`#T`**, **`#F`**, and **`#<eof>`**.

The `UNDEFINED` value is used to indicate that a symbol is not bound. When a symbol is bound to `UNDEFINED`, the interpreter considers it to be not bound at all. The `UNSPECIFIC` value is used to express the results of operations without a specific value, such as **`(if #f #f)`**.

The values `DOT` and `RPAREN` represent the lexemes “`.`” and “`)`”. `NOEXPR` is a special value that indicates an unknown or uninteresting source when reporting an error.

```
#define special_value_p(x)        ((x) < 0)
#define NIL               (-1)
#define TRUE              (-2)
#define FALSE             (-3)
#define ENDOFFILE         (-4)
#define UNDEFINED         (-5)
#define UNSPECIFIC        (-6)
#define DOT               (-7)
#define RPAREN            (-8)
#define NOEXPR            (-9)
```

Global variables:

```
int     Pool_size = 0,          /* Current size of the node pool */
        Vpool_size = 0;         /* Current size of the vector pool */
```

The `Car`, `Cdr`, and `Tag` arrays form the “node pool”. Each node **n** consists of `Car[n]`, `Cdr[n]`, and `Tag[n]`. `Car` and `Cdr` contain offsets of other members of the node pool. So in fact S9fES uses indices rather than pointers, but the values of `Car` and `Cdr` will frequently be refered to as pointers, because they point to other nodes of the pool. `Tag` contains the GC flags of a node.

The size of a node is $2\times$`sizeof(int)` + `sizeof(char)`.

```
int     *Car = NULL,            /* Car parts of nodes */
        *Cdr = NULL;            /* Cdr parts of nodes */
char    *Tag = NULL;            /* Tag fields of nodes */
```

`Vectors` holds the values of vector objects (Scheme strings and vectors).

```
int     *Vectors = NULL;        /* Vector pool */
int     Free_vecs = 0;          /* Pointer to free vector space */
```

`Stack`, `Tmp`, and all other symbols that are initialized with `NIL` refer to objects in the node pool.

```
int     Stack = NIL,            /* Evaluation stack */
        Stack_bottom = NIL;     /* Bottom of Stack */
int     State_stack = NIL;      /* State stack of the evaluator */
int     Tmp_car = NIL,          /* Safe locations during allocation */
        Tmp_cdr = NIL,
        Tmp = NIL;
int     Free_list = NIL;        /* Pointer to free node */
int     Symbols = NIL;          /* Global symbols table */
int     Program = NIL;          /* Program being executed */
int     Environment = NIL;      /* Global environment */
int     Acc = NIL;              /* Accumulator (used by the evaluator) */
```

`Ports` and `Port_flags` form the port pool. Each port **n** refers to the `FILE* Ports[n]` and the associated flags `Port_flags[n]`.

```
FILE    *Ports[MAX_PORTS];
char    Port_flags[MAX_PORTS];
int     Input_port = 0,         /* Current input port */
        Output_port = 1;        /* Current output port */
int     Level = 0;              /* Parenthesis nesting level */
int     Error_flag = 0;
int     Load_level = 0;         /* Number of nested LOADs */
int     Displaying = 0;         /* Distinguish DISPLAY from WRITE */
int     Quiet_mode = 0;         /* Be quiet (-q option) */
```

These variables refer to internal nodes like type tags (e.g. `S_char` or `S_procedure`), but also to keywords that need to be looked up by the interpreter internally (like `S_else` or `S_unquote`).

```
int     S_char, S_else, S_input_port, S_integer, S_latest,
        S_output_port, S_primitive, S_procedure, S_quasiquote,
        S_quote, S_string, S_symbol, S_syntax, S_unquote,
        S_unquote_splicing, S_vector;
```

These variables refer to symbols that are Scheme keywords.

```
int     S_and, S_begin, S_cond, S_define, S_define_syntax, S_if,
        S_lambda, S_let, S_letrec, S_or, S_quote, S_set_b,
        S_syntax_rules;
```

`GC_root` holds the roots of the active part of the node pool. The active parts of the node pool are shaped like trees, and the GC roots are the roots of these trees. Each of the trees rooted at `GC_root` will be considered active in garbage collections. All nodes that are not refered to by `GC_root` (directly or indirectly) will be reclaimed and added to the free list. `NULL` terminates `GC_root`.

```
int     *GC_root[] = { &Program, &Symbols, &Environment, &Tmp, &Tmp_car,
                       &Tmp_cdr, &Stack, &Stack_bottom, &State_stack,
                       &Acc, NULL };
```

```
#define nl()            pr("\n")
#define reject(c)       ungetc(c, Ports[Input_port])
```

These macros are used to extract values from atoms:

Macro	**Type**	**Return value**
`string()`	`node -> char*`	string text (C string)
`string_len()`	`node -> int`	string length (*including* NUL)
`vector()`	`node -> node*`	array holding vector members
`vector_len()`	`node -> int`	number of vector members
`vector_size()`	`int -> int`	size of vector in `ints`
`port_no()`	`node -> port`	offset into `Ports[]`
`char_value()`	`node -> int`	character

Most of these macros accept a node and return a property of that node. The `string`... macros may only be applied to string nodes, the `vector`... macros only to vector nodes, etc.

`vector_size()` accepts a size in bytes and returns the number of vector cells (`ints`) required to store an object of the given size.

```
#define string(n)         ((char *) &Vectors[Cdr[n]])
#define string_len(n)     (Vectors[Cdr[n] - 1])
#define vector_size(k)    (((k) + sizeof(int)-1) / sizeof(int) + 2)
#define vector(n)         (&Vectors[Cdr[n]])
#define vector_len(n)     (vector_size(string_len(n)) - 2)
#define port_no(n)        (cadr(n))
#define char_value(n)     (cadr(n))

#define caar(x)           (Car[Car[x]])
#define cadr(x)           (Car[Cdr[x]])
#define cdar(x)           (Cdr[Car[x]])
#define cddr(x)           (Cdr[Cdr[x]])
#define caaar(x)          (Car[Car[Car[x]]])
#define caadr(x)          (Car[Car[Cdr[x]]])
#define cadar(x)          (Car[Cdr[Car[x]]])
#define caddr(x)          (Car[Cdr[Cdr[x]]])
#define cdadr(x)          (Cdr[Car[Cdr[x]]])
#define cddar(x)          (Cdr[Cdr[Car[x]]])
#define cdddr(x)          (Cdr[Cdr[Cdr[x]]])
#define caaddr(x)         (Car[Car[Cdr[Cdr[x]]]])
#define caddar(x)         (Car[Cdr[Cdr[Car[x]]]])
#define cadadr(x)         (Car[Cdr[Car[Cdr[x]]]])
#define cadddr(x)         (Car[Cdr[Cdr[Cdr[x]]]])
#define cdddar(x)         (Cdr[Cdr[Cdr[Car[x]]]])
#define cddddr(x)         (Cdr[Cdr[Cdr[Cdr[x]]]])

#define null_p(n)         ((n) == NIL)
#define eof_p(n)          ((n) == ENDOFFILE)
#define undefined_p(n)    ((n) == UNDEFINED)
#define unspecific_p(n) ((n) == UNSPECIFIC)
```

The following macros test the type of a given node.

```
#define boolean_p(n)    ((n) == TRUE || (n) == FALSE)

#define integer_p(n) \
        (!special_value_p(n) && (Tag[n] & AFLAG) && Car[n] == S_integer)
#define primitive_p(n) \
        (!special_value_p(n) && (Tag[n] & AFLAG) && Car[n] == S_primitive)
#define procedure_p(n) \
        (!special_value_p(n) && (Tag[n] & AFLAG) && Car[n] == S_procedure)
#define special_p(n)    ((n) == S_and    || \
                         (n) == S_begin  || \
                         (n) == S_cond   || \
                         (n) == S_define || \
                         (n) == S_define_syntax || \
                         (n) == S_if     || \
                         (n) == S_lambda || \
                         (n) == S_let    || \
                         (n) == S_letrec || \
                         (n) == S_or     || \
                         (n) == S_quote  || \
                         (n) == S_syntax_rules  || \
                         (n) == S_set_b)
#define char_p(n) \
        (!special_value_p(n) && (Tag[n] & AFLAG) && Car[n] == S_char)
#define syntax_p(n) \
        (!special_value_p(n) && (Tag[n] & AFLAG) && Car[n] == S_syntax)
#define input_port_p(n) \
        (!special_value_p(n) && (Tag[n] & AFLAG) && Car[n] == S_input_port)
#define output_port_p(n) \
        (!special_value_p(n) && (Tag[n] & AFLAG) && Car[n] == S_output_port)

#define symbol_p(n) \
        (!special_value_p(n) && (Tag[n] & VFLAG) && Car[n] == S_symbol)
#define vector_p(n) \
        (!special_value_p(n) && (Tag[n] & VFLAG) && Car[n] == S_vector)
#define string_p(n) \
        (!special_value_p(n) && (Tag[n] & VFLAG) && Car[n] == S_string)
```

The following types need no quotation in Scheme. The undefined and unspecific value are auto-quoting for technical reasons that will become clear when discussing the evaluator. Actually, we are cheating a bit here, because R5RS vectors are not really auto-quoting, but assuming that they are simplifies the implementation.

```
#define auto_quoting_p(n) (boolean_p(n)    || \
                           char_p(n)       || \
                           eof_p(n)        || \
                           integer_p(n)    || \
                           string_p(n)     || \
                           undefined_p(n)  || \
                           unspecific_p(n) || \
                           vector_p(n))
```

The "rib" is a structure that holds parts of a procedure application during evaluation. It is a four-element list with the following members:

Member	**Content**
`rib_args`	List of arguments yet to be processed
`rib_append`	The last element of `rib_result`
`rib_result`	The values of already processed arguments
`rib_source`	The complete source expression

`rib_result` shares its last member with `rib_append`, so `rib_append` can be used to append members to `rib_result` efficiently (that is, in O(1) time rather than O(n) time).

```
#define rib_args(x)      (Car[x])
#define rib_append(x)    (cadr(x))
#define rib_result(x)    (caddr(x))
#define rib_source(x)    (cadddr(x))
```

The `atom_p()` predicate [2] tests whether a given node is the root of an atomic Scheme object.

```
#define atom_p(n) \
        (special_value_p(n) || (Tag[n] & AFLAG) || (Tag[n] & VFLAG))

#define pair_p(x) (!atom_p(x))

int error(char *msg, int expr);
```

All interpreter output goes through the `pr()` function.

```
void pr(char *s) {
        if (Ports[Output_port] == NULL)
                error("output port is not open", NOEXPR);
        else
                fwrite(s, 1, strlen(s), Ports[Output_port]);
}

void print(int n);  /* print a node */
```

The `error()` function set `Error_flag` and writes an error message to the default output stream (`stdout`). When the interpreter runs in quiet mode, it terminates the interpreter process. When the `expr` argument is not equal to `NOEXPR`, it will print after the error message. `fatal()` is like `error()`, but accepts no source expression and terminates the interpreter in any case.

```
int error(char *msg, int expr) {
        int     oport;

        if (Error_flag) return UNSPECIFIC;
        oport = Output_port;
        Output_port = 1;
        Error_flag = 1;
```

2 Following LISP tradition, the trailing "p" indicates a predicate.

```
        printf("error: %s", msg);
        if (expr != NOEXPR) {
                pr(": ");
                print(expr);
        }
        nl();
        Output_port = oport;
        if (Quiet_mode) exit(1);
        return UNSPECIFIC;
}

int fatal(char *msg) {
        printf("fatal ");
        error(msg, NOEXPR);
        exit(2);
}
```

Memory Management

S9fES uses a Deutsch/Schorr/Waite mark and sweep garbage collector that runs in constant space. A mark and sweep collector employs two phases. The first phase marks all nodes that can be accessed by the program that is currently executing. The second phase sweeps through the node pool and moves all unmarked nodes to the free list. A constant space collector is a collector that uses the same amount of storage, no matter what structures it is traversing. Most notably, it does not use the stack to recurse into trees.

When the memory allocator runs low on free nodes, it allocates a new segment of memory and adds it to the pool. This is what the `new_segment()` function does. Initially, the pool has a size of zero, so this function also allocates the initial pool with a size of one segment.

```
void new_segment(void) {
        Car = realloc(Car, sizeof(int) * (Pool_size + SEGMENT_LEN));
        Cdr = realloc(Cdr, sizeof(int) * (Pool_size + SEGMENT_LEN));
        Tag = realloc(Tag, Pool_size + SEGMENT_LEN);
        Vectors = realloc(Vectors, sizeof(int) * (Vpool_size + SEGMENT_LEN));
        if (    Car == NULL || Cdr == NULL || Tag == NULL ||
                Vectors == NULL
        ) {
                fatal("out of memory");
        }
        memset(&Car[Pool_size], 0, SEGMENT_LEN * sizeof(int));
        memset(&Cdr[Pool_size], 0, SEGMENT_LEN * sizeof(int));
        memset(&Tag[Pool_size], 0, SEGMENT_LEN);
        memset(&Vectors[Pool_size], 0, SEGMENT_LEN * sizeof(int));
        Pool_size += SEGMENT_LEN;
        Vpool_size += SEGMENT_LEN;
}

void mark(int n);  /* mark phase */
```

The `mark_vector()` function marks the members of a vector. The argument `n` is a node and `type` is the type of that node.

Actually, the S9fES garbage collector does not *fully* run in constant space. It does recurse to mark the members of vectors. This may be considered a bug, but not a critical one, because deeply nested vectors are rare (or at least not as common as deeply nested lists).

```
void mark_vector(int n, int type) {
        int     *p, k;

        p = &Vectors[Cdr[n] - 2];
        *p = n;
        if (type == S_vector) {
                k = vector_len(n);
                p = vector(n);
                while (k) {
                        mark(*p);
                        p++;
                        k--;
                }
        }
}
```

The `mark()` function implements a finite state machine (FSM) that traverses a tree rooted at the node `n`. The function marks all nodes, vectors, and ports that it encounters during traversal as "life" objects, i.e. objects that may not be recycled. The FSM uses three states (0,1,2) that are formed using the collector flags `MFLAG` and `SFLAG`. `MFLAG` is a state flag and the "mark" flag – which is used to tag live nodes – at the same time. The following figures illustrate the states of the root node during the traversal of a tree of three nodes. Marked nodes are rendered with a grey background.

State 0: Node **N** is unvisited. The parent points to NIL, both flags are cleared.

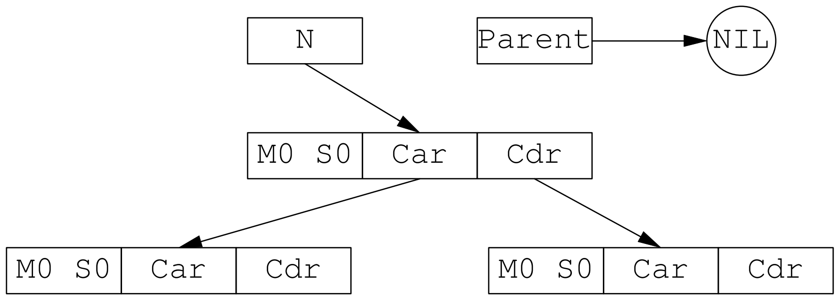


Fig. 1 – Garbage collection, State 0

State 1: **N** now points to the car child of the root node, the parent pointer points to the root node, and the parent of the parent is stored in the car part of the root node. Both flags are set. The node is now marked.

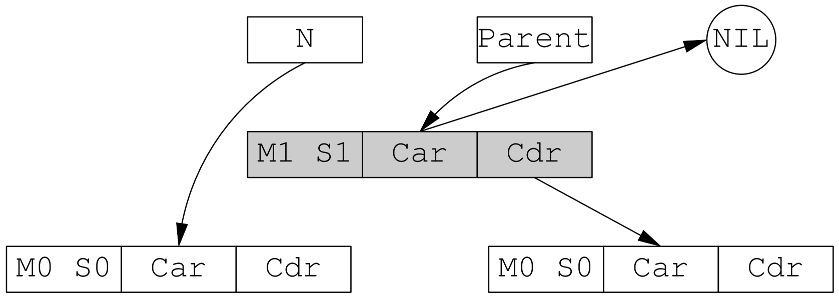


Fig. 2 – Garbage collection, State 1

State 2: When the car child is completed, the car pointer of the root is restored, the grandparent moves to the cdr part of the root node, and **N** moves to the cdr child. The **S** flag is cleared, and the root node is now completely traversed.

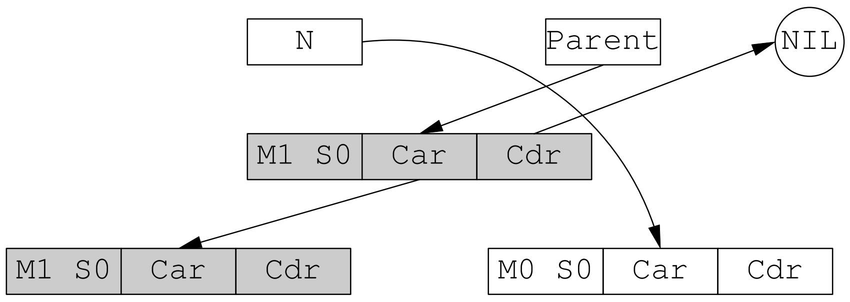


Fig. 3 – Garbage collection, State 2

State 2: When the FSM returns from the cdr child, it finds the root node in state 2. To return to the root, it restores the cdr pointer of the root node and the parent. **N** moves up to the root node. Because the parent is now NIL, the traversal is completed.

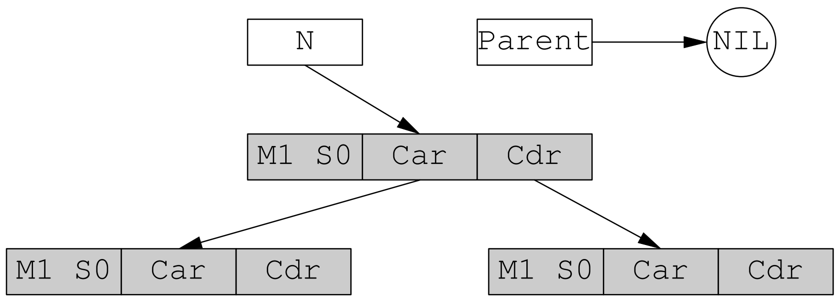


Fig. 4 – Garbage collection, finished

When the FSM hits an atom or an already marked node during its traversal, it returns to the parent node immediately. Because nodes get marked *before* their descendants are traversed, the FSM can traverse cyclic structures without entering an infinite loop.

```
/*
 * Mark nodes which can be accessed through N.
 * Using the Deutsch/Schorr/Waite (aka pointer reversal) algorithm.
 * S0: M==0 S==0 unvisited, process CAR
 * S1: M==1 S==1 CAR visited, process CDR
 * S2: M==1 S==0 completely visited, return to parent
 */
void mark(int n) {
        int     p, parent;

        parent = NIL;   /* Initially, there is no parent node */
        while (1) {
                if (special_value_p(n) || Tag[n] & MFLAG) {
                        if (parent == NIL) break;
                        if (Tag[parent] & SFLAG) {      /* S1 --> S2 */
                                p = Cdr[parent];
                                Cdr[parent] = Car[parent];
                                Car[parent] = n;
                                Tag[parent] &= ~SFLAG;
                                Tag[parent] |=  MFLAG;
                                n = p;
                        }
                        else {                          /* S2 --> done */
```

```
                                p = parent;
                                parent = Cdr[p];
                                Cdr[p] = n;
                                n = p;
                        }
                }
                else {
                        if (Tag[n] & VFLAG) {           /* S0 --> done */
                                Tag[n] |= MFLAG;
                                /* Tag[n] &= ~SFLAG; */
                                mark_vector(n, Car[n]);
                        }
                        else if (Tag[n] & AFLAG) {      /* S0 --> S2 */
                                if (input_port_p(n) || output_port_p(n))
                                        Port_flags[port_no(n)] |= UFLAG;
                                p = Cdr[n];
                                Cdr[n] = parent;
                                /*Tag[n] &= ~SFLAG;*/
                                parent = n;
                                n = p;
                                Tag[parent] |= MFLAG;
                        }
                        else {                          /* S0 --> S1 */
                                p = Car[n];
                                Car[n] = parent;
                                Tag[n] |= MFLAG;
                                parent = n;
                                n = p;
                                Tag[parent] |= SFLAG;
                        }
                }
        }
}
```

The `unmark_vectors()` function unmarks all strings and vectors in the vector pool. The following disgram outlines the structure of the vector pool (the grey parts are meta information):

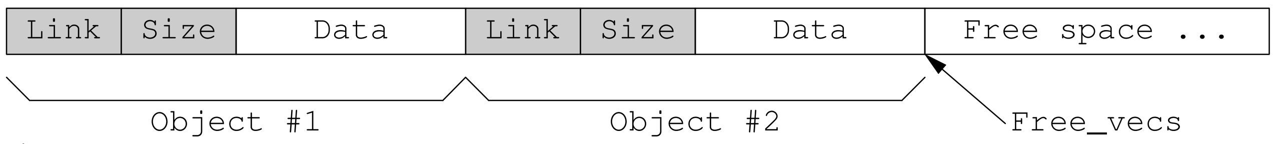


Fig. 5 – Vector pool structure

The “link” field of an object points back to the string or vector node that refers to the data of that object. The “size” field specifies the size of the object in the data area in bytes. The mark and link fields have a size of one `int` and the size of each data field is a multiple of `sizeof(int)` (that is, data areas are allocated in `int`s).

Because the vector pool does not have an `MFLAG`, the `unmark_vectors()` function clears the link fields of the objects by setting them to NIL.

```
/* Mark all vectors unused */
void unmark_vectors(void) {
        int     p, k, link;

        p = 0;
        while (p < Free_vecs) {
                link = p;
                k = Vectors[p+1];
                p += vector_size(k);
                Vectors[link] = NIL;
        }
}
```

The `gc()` function is invoked whenever the interpreter runs low on free nodes. It first unmarks all I/O ports that are not locked and then marks all objects that can be accessed through `GC_root`. Thereafter it rebuilds the free list by clearing it and then adding all nodes that were not tagged during the mark step. In the same loop, nodes that were marked are unmarked again. Finally, the function closes all unused ports.

Note that `gc()` does *not* garbage collect the vector pool. This is done by a separate collector.

```
/* Mark and Sweep GC. */
int gc(void) {
        int     i, k;

        if (Debug_GC) pr("GC called: ");
        for (i=0; i<MAX_PORTS; i++)
                if (Port_flags[i] & LFLAG)
                        Port_flags[i] |= UFLAG;
                else
                        Port_flags[i] &= ~UFLAG;
        for (i=0; GC_root[i] != NULL; i++) mark(GC_root[i][0]);
        k = 0;
        Free_list = NIL;
        for (i=0; i<Pool_size; i++) {
                if (!(Tag[i] & MFLAG)) {
                        Cdr[i] = Free_list;
                        Free_list = i;
                        k = k+1;
                }
                else {
                        Tag[i] &= ~MFLAG;
                }
        }
        for (i=0; i<MAX_PORTS; i++) {
                if (!(Port_flags[i] & UFLAG) && Ports[i] != NULL) {
                        fclose(Ports[i]);
                        Ports[i] = NULL;
                }
        }
        if (Debug_GC) printf("%d nodes reclaimed\n", k);
        return k;
}
```

`alloc3()` is the principal memory allocator of S9fES. It allocates a fresh node and fills its car, cdr, and tag fields with the parameters passed to it. The fresh node is taken from the head of the free list and then removed from the list. When the free list is empty, a garbage collection is performed. When the size of the free list is below 10% of the size of the node pool after a collection, `alloc3()` allocates a new segment.

The nodes passed to `alloc3()` are protected from being recycled by storing their values in the `Tmp_car` and `Tmp_cdr` fields, which are part of `GC_root`.

Note that it is a *very* bad idea to wait for the free list to run completely out of nodes before adding a new segment, because garbage collections become more and more frequent as available memory shrinks. Allowing this to happen adds a big memory management overhead and degrades performance severely.

```
/* Allocate a fresh node and initialize with PCAR,PCDR,PTAG. */
int alloc3(int pcar, int pcdr, int ptag) {
        int     n, k;

        if (Free_list == NIL) {
                if (ptag == 0) Tmp_car = pcar;
                Tmp_cdr = pcdr;
                k = gc();
                /*
                 * Performance increases dramatically if we
                 * do not wait for the pool to run dry.
                 */
                if (k < Pool_size / 10) {
                        if (    MEMORY_LIMIT_KN &&
                                Pool_size + SEGMENT_LEN > MEMORY_LIMIT_KN*1024
                        ) {
                                if (Free_list == NIL)
                                        fatal("alloc3(): hit memory limit");
                        }
                        else {
                                new_segment();
                                gc();
                                if (Debug_GC)
                                        printf("Alloc3: new segment,"
                                               " nodes = %d\n",
                                                        Pool_size);
                        }
                }
                Tmp_car = Tmp_cdr = NIL;
        }
        n = Free_list;
        Free_list = Cdr[Free_list];
        Car[n] = pcar;
        Cdr[n] = pcdr;
        Tag[n] = ptag;
        return n;
}
```

This is a short cut for allocating cons nodes:

```
#define alloc(pcar, pcdr) alloc3((pcar), (pcdr), 0)
```

The `gcv()` function performs vector garbage collection and compaction. It first unmarks all string and vector objects and then performs an ''ordinary'' garbage collection in order to mark life vectors. After that it compacts the vector pool by moving all life objects to the beginning of the pool. Because vector objects – unlike nodes – have variables sizes, memory fragmentation could occur without compaction.

The following figure shows a fragmented vector pool (before running `gcv()`). Life objects are rendered in grey.

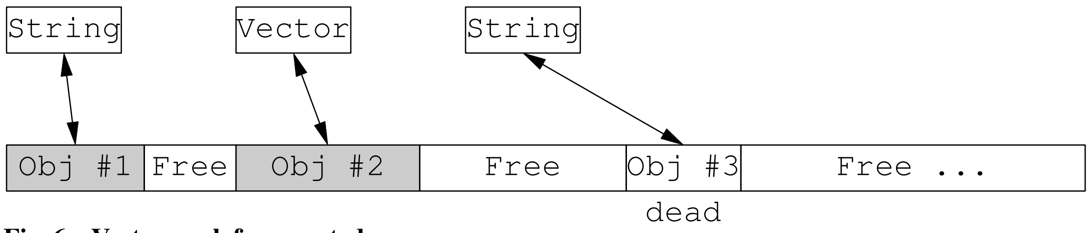


Fig. 6 – Vector pool, fragmented

The compacting garbage collector moves the life objects to one side of the pool, so that one contiguous area of free space is created at the other end. When an object is moved, the reference in the node that stores its data in the vector object is updated to reflect the new address of the vector object.

The following figure shows the same vector pool as above, but after compaction:

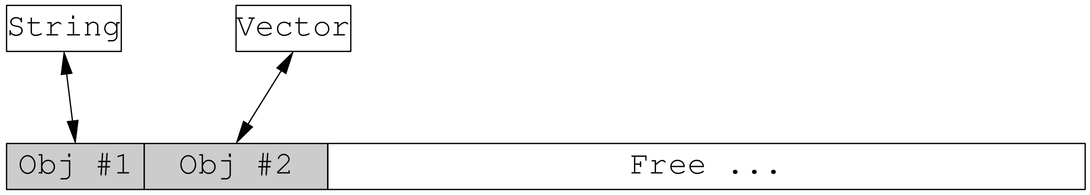


Fig. 7 – Vector pool, compacted

```
/* In situ vector pool garbage collection and compaction */
int gcv(void) {
        int     v, k, to, from;

        unmark_vectors();
        gc();           /* re-mark life vectors */
        if (Debug_GC) printf("GCV called: ");
        to = from = 0;
        while (from < Free_vecs) {
                v = Vectors[from+1];
                k = vector_size(v);
                if (Vectors[from] != NIL) {
                        if (to != from) {
                                memmove(&Vectors[to], &Vectors[from],
                                        k * sizeof(int));
                                Cdr[Vectors[to]] = to + 2;
                        }
                        to += k;
                }
                from += k;
        }
        k = Free_vecs - to;
        if (Debug_GC) printf("%d cells reclaimed\n", k);
        Free_vecs = to;
        return k;
}
```

The `allocv()` function is similar to `alloc()` but allocates a string or a vector node. Because memory compaction is an expensive operation, `allocv()` maintains a rather large vector pool: about two times the size of the maximal amount of memory required so far. Like `alloc()`, `allocv()` can trigger the allocation of an extra segment.

```
/* Allocate vector from pool */
int allocv(int type, int size) {
        int     v, n, wsize, k;

        wsize = vector_size(size);
        if (Free_vecs + wsize >= Vpool_size) {
                k = gcv();
                if (    Free_vecs + wsize >=
                        Vpool_size - Vpool_size / 2
                ) {
                        if (    MEMORY_LIMIT_KN &&
                                Pool_size + SEGMENT_LEN > MEMORY_LIMIT_KN*1024
                        ) {
                                if (Free_list == NIL)
                                        fatal("allocv(): hit memory limit");
                        }
                        else {
                                new_segment();
                                gcv();
                                if (Debug_GC)
                                        printf("Allocv: new segment,"
                                                " nodes = %d\n",
                                                        Pool_size);
                        }
                }
        }
        v = Free_vecs;
        Free_vecs += wsize;
        n = alloc3(type, v+2, VFLAG);
        Vectors[v] = n;
        Vectors[v+1] = size;
        return n;
}
```

The `save()` and `unsave()` functions maintain the global stack.

```
#define save(n) (Stack = alloc((n), Stack))

/* Pop K nodes off the Stack, return last one. */
int unsave(int k) {
        int     n = NIL; /*LINT*/

        while (k) {
                if (Stack == NIL) fatal("unsave(): stack underflow");
                n = Car[Stack];
                Stack = Cdr[Stack];
                k = k-1;
        }
        return n;
}
```

The Reader

The “reader” is the part of the Scheme interpreter that implements the **`read`** procedure, which translates the external (textual) representation of a program into an internal representation. The internal representation used by S9fES is a tree of nodes.

`find_symbol()` finds a symbol in the global symbol table and returns it. When no symbol with the given name exists, it returns NIL.

```
int find_symbol(char *s) {
        int     y;

        y = Symbols;
        while (y != NIL) {
                if (!strcmp(string(Car[y]), s))
                        return Car[y];
                y = Cdr[y];
        }
        return NIL;
}
```

`make_symbol()` creates a new symbol with the name `s`. The parameter `k` specifies the length of the symbol name.

```
int make_symbol(char *s, int k) {
        int     n;

        n = allocv(S_symbol, k+1);
        strcpy(string(n), s);
        return n;
}
```

`add_symbol()` creates a unique symbol. When a symbol with the given name altead... exists, it returns it, otherwise it creates a new one.

```
int add_symbol(char *s) {
        int     y;

        y = find_symbol(s);
        if (y != NIL) return y;
        Symbols = alloc(NIL, Symbols);
        Car[Symbols] = make_symbol(s, strlen(s));
        return Car[Symbols];
}
```

All input of the interpreter goes through the `read_c()` macro.

```
#define read_c() getc(Ports[Input_port])

#define read_c_ci() tolower(read_c())

int read_form(void);  /* read a Scheme form */
```

The `read_list()` function reads the members of a list plus the closing parenthesis. It is called by `read_form()` when an opening parenthesis is encountered in input and it calls `read_form()` to read each member of a list.

`read_list()` reports an error if it encounters a syntactically incorrect improper list (with no element before the dot or multiple elements after the dot) or when the input stream is exhausted before a closing parenthesis is found.

```
int read_list(void) {
        int     n,      /* Node read */
                m,      /* List */
                a,      /* Used to append nodes to m */
                c;      /* Member counter */
        char    *badpair;

        badpair = "bad pair";
        Level = Level+1;
        m = alloc(NIL, NIL);    /* root */
        save(m);
        a = NIL;
        c = 0;
        while (1) {
                if (Error_flag) {
                        unsave(1);
                        return NIL;
                }
                n = read_form();
                if (n == ENDOFFILE)  {
                        if (Load_level) {
                                unsave(1);
                                return ENDOFFILE;
                        }
                        error("missing ')'", NOEXPR);
                }
                if (n == DOT) {
                        if (c < 1) {
                                error(badpair, NOEXPR);
                                continue;
                        }
                        n = read_form();
                        Cdr[a] = n;
                        if (n == RPAREN || read_form() != RPAREN) {
                                error(badpair, NOEXPR);
                                continue;
                        }
                        unsave(1);
                        Level = Level-1;
                        return m;
                }
                if (n == RPAREN) break;
                if (a == NIL)
                        a = m;          /* First member: insert at root */
                else
                        a = Cdr[a];     /* Following members: append */
                Car[a] = n;
                Cdr[a] = alloc(NIL, NIL); /* Alloc space for next member */
                c = c+1;
        }
        Level = Level-1;
        if (a != NIL) Cdr[a] = NIL;     /* Remove trailing empty node */
        unsave(1);
        return c? m: NIL;
}
```

Create a quotation or quasiquotation (the forms **`(quote n)`**, **`(quasiquote n)`**, **`(unquote n)`**, and **`(unquote-splicing n)`**).

```
int quote(int n, int quotation) {
        int     q;

        q = alloc(n, NIL);
        return alloc(quotation, q);
}
```

Check whether the string `s` represents a number. A number is a non-empty sequence of decimal digits with an optional leading plus (+) or minus (−) character.

```
int str_numeric_p(char *s) {
        int     i;

        i = 0;
        if (s[0] == '+' || s[0] == '-') i = 1;
        if (!s[i]) return 0;
        while (s[i]) {
                if (!isdigit(s[i])) return 0;
                i = i+1;
        }
        return 1;
}
```

The `string_to_bignum()` function converts the string `s` to a bignum integer. The internal form of a bignum integer is illustrated in the following figure.

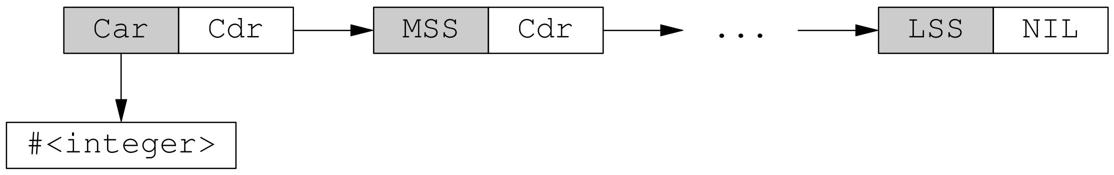


Fig. 8 – Bignum integer structure

Atomic nodes are rendered with a grey car part. The first node of a bignum integer has the type tag **`#<integer>`** as its car part and a list of integer segments as its cdr part. Each member of the list of integer segments is atomic and stores the segment itself in the car part and a pointer to the next node in its cdr part. The first member of the list holds the *most significant segment* (MSS) which includes the sign of the bignum integer. There may be any number of integer segments between the MSS and the LSS (*least significant segment*). When a bignum integer fits in one integer segment, the MSS node has a cdr part of NIL

Note that `mark()` does not follow the car part of the first node of a bignum integer, because it is atomic. This does not matter, though, because the **`#<integer>`** tag is in the symbol table which is part of `GC_root`, so the type tag is protected from GCs anyway.

```
int string_to_bignum(char *s) {
        int     k, j, n, v, sign;

        sign = 1;
        if (s[0] == '-') {
                s++;
                sign = -1;
        }
        else if (s[0] == '+') {
                s++;
        }
        k = strlen(s);
        n = NIL;
        while (k) {
                j = k <= DIGITS_PER_WORD? k: DIGITS_PER_WORD;
                v = atol(&s[k-j]);
                s[k-j] = 0;
                k -= j;
                if (k == 0) v *= sign;
                n = alloc3(v, n, AFLAG);
        }
        return alloc3(S_integer, n, AFLAG);
}
```

The `make_char()` function creates the internal representation of a character literal. The internal representation of all atomic types is similar:

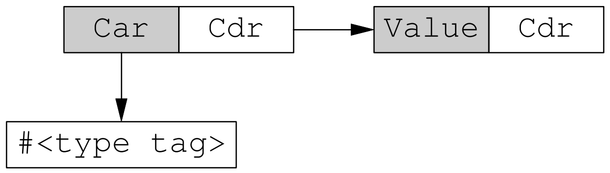


Fig. 9 – Atomic type structure

Some types have multi-node values (like bignum integers), others have a single-node value. Characters have a **`#<char>`** type tag and store the character itself in the value field. The cdr part of the value node is NIL.

```
/* Create a character literal. */
int make_char(int x) {
        int     n;

        n = alloc3(x, NIL, AFLAG);
        return alloc3(S_char, n, AFLAG);
}
```

The `character()` function reads a character literal and returns its internal representation. It also translates multi-character names like **`#\space`** and **`#\newline`**.

```
/* Read a character literal. */
int character(void) {
        char    buf[10];
        int     i, c;

        for (i=0; i<9; i++) {
                c = read_c();
                if (i > 0 && !isalpha(c)) break;
                buf[i] = c;
        }
        reject(c);
        buf[i] = 0;
        if (i == 0) c = ' ';
        else if (i == 1) c = buf[0];
        else if (!strcmp(buf, "space")) c = ' ';
        else if (!strcmp(buf, "newline")) c = '\n';
        else if (!strcmp(buf, "linefeed")) c = '\n';
        else {
                error("bad # syntax", NOEXPR);
                c = 0;
        }
        return make_char(c);
}
```

Create the internal representation of a string. Strings have a **`#<string>`** type tag and store a pointer to the vector pool in their value fields.

```
/* Create a string; K = length */
int make_string(char *s, int k) {
        int     n;

        n = allocv(S_string, k+1);
        strcpy(string(n), s);
        return n;
}
```

The `string_literal()` function parses a string literal and returns its internal representation. It recognizes the espace sequences `\\` and `\"` and flags illegal uses of the backslash.

```
/* Read a string literal. */
int string_literal(void) {
        char    s[TEXT_LEN+1];
        int     c, i, n, q;
        int     inv;

        i = 0;
        q = 0;
        c = read_c();
        inv = 0;
        while (q || c != '"') {
                if (Error_flag) break;
                if (i >= TEXT_LEN-2) {
                        error("string literal too long", NOEXPR);
                        i = i-1;
                }
                if (q && c != '"' && c != '\\') {
                        s[i++] = '\\';
                        inv = 1;
                }
                s[i] = c;
                q = !q && c == '\\';
                if (!q) i = i+1;
                c = read_c();
        }
        s[i] = 0;
        n = make_string(s, i);
        if (inv) error("invalid escape sequence in string", n);
        return n;
}
```

Read an “unreadable” external representation and report an error.

```
/* Report unreadable object */
int unreadable(void) {
        int     c, i;
        char    buf[TEXT_LEN];
        int     d;

        strcpy(buf, "#<");
        i = 2;
        while (1) {
                c = read_c_ci();
                if (c == '>' || c == '\n') break;
                if (i < TEXT_LEN-2) buf[i++] = c;
        }
        buf[i++] = '>';
        buf[i] = 0;
        d = Displaying;
        Displaying = 1;
        error("unreadable object", make_string(buf, i));
        Displaying = d;
        return NIL;
}
```

These characters delimit tokens.

```
#define separator(c) \
        ((c) == ' '  || (c) == '\t' || (c) == '\n' || \
         (c) == '\r' || (c) == '('  || (c) == ')'  || \
         (c) == ';'  || (c) == '#'  || (c) == '\'' || \
         (c) == '`'  || (c) == ','  || (c) == '"'  || \
         (c) == EOF)
```

The `symbol_or_number()` function reads a token (text up to a delimiter) and check whether the text of that token represents a valid integer number. If it does, it returns a bignum integer, otherwise it returns a symbol. Symbols are created if they do not already exist.

```
int symbol_or_number(int c) {
        char    s[TEXT_LEN];
        int     i;

        i = 0;
        while (!separator(c)) {
                if (i >= TEXT_LEN-2) {
                        error("symbol too long", NOEXPR);
                        i = i-1;
                }
                s[i] = c;
                i = i+1;
                c = read_c_ci();
        }
        s[i] = 0;
        reject(c);
        if (str_numeric_p(s)) return string_to_bignum(s);
        return add_symbol(s);
}
```

Skip over nested block comments (**#| ... |#**) and return the first character following the comment. Block comments are not part of R5RS, but they are cheap to implement, so here we go.

```
int nested_comment(void) {
        int     p, c, k;
        k = 1;
        p = 0;
        c = read_c();
        while (k) {
                if (c == EOF) fatal("end of input in nested comment");
                if (p == '#' && c == '|') { k++; c = 0; }
                if (p == '|' && c == '#') { k--; c = 0; }
                p = c;
                c = read_c();
        }
        return c;
}
```

Convert the list `m` into a vector and return it. The `msg` parameter is an error message that will print when `m` is an improper list. The message is passed to `list_to_vector()` rather than hardwiring it, because the function is used for different purposes which require different error messages.

```
static int list_to_vector(int m, char *msg) {
        int     n;
        int     vec, k;
        int     *p;

        k = 0;
        for (n = m; n != NIL; n = Cdr[n]) {
                if (atom_p(n)) return error(msg, m);
                k++;
        }
        vec = allocv(S_vector, k*sizeof(int));
        p = vector(vec);
        for (n = m; n != NIL; n = Cdr[n]) {
                *p = Car[n];
                p++;
        }
        return vec;
}
```

Read a vector literal and return it.

```
static int read_vector(void) {
        int     n;

        n = read_list();
        save(n);
        n = list_to_vector(n, "bad vector syntax");
        unsave(1);
        return n;
}
```

The `read_form()` function implements most of the functionality of the Scheme procedure **`read`**. It reads one Scheme form from the current input port and returns its internal representation. Reading from a port other than the current input port is implemented in the `pp_read()` function which follows later.

```
int read_form(void) {
        int     c, c2;

        c = read_c_ci();
        while (1) {     /* Skip spaces and comments */
                while (c == ' ' || c == '\t' || c == '\n' || c == '\r') {
                        if (Error_flag) return NIL;
                        c = read_c_ci();
                }
                if (c == '#') {
                        c = read_c_ci();
                        if (c == '|') {
                                c = nested_comment();
                                continue;
                        }
                        if (c == ';') {
                                read_form();
                                c = read_c_ci();
                                continue;
                        }
                        if (c != '!') {
                                reject(c);
                                c = '#';
                                break;
                        }
                }
                else if (c != ';')
                        break;
                while (c != '\n' && c != EOF) c = read_c_ci();
        }
        if (c == EOF) return ENDOFFILE;
        if (c == '(') {
                return read_list();
        }
        else if (c == '\'') {
                return quote(read_form(), S_quote);
        }
        else if (c == '`') {
                return quote(read_form(), S_quasiquote);
        }
        else if (c == ',') {
                c = read_c_ci();
                if (c == '@') {
                        return quote(read_form(), S_unquote_splicing);
                }
                else {
                        reject(c);
                        return quote(read_form(), S_unquote);
                }
        }
        else if (c == '#') {
                c = read_c_ci();
                if (c == 'f') return FALSE;
                if (c == 't') return TRUE;
                if (c == '\\') return character();
                if (c == '(') return read_vector();
                if (c == '<') return unreadable();
                return error("bad # syntax", NOEXPR);
        }
        else if (c == '"') {
                return string_literal();
        }
        else if (c == ')') {
                if (!Level) return error("unexpected ')'", NOEXPR);
                return RPAREN;
        }
        else if (c == '.') {
                c2 = read_c_ci();
                reject(c2);
                if (separator(c2)) {
                        if (!Level) return error("unexpected '.'", NOEXPR);
                        return DOT;
                }
                return symbol_or_number(c);
        }
        else {
                return symbol_or_number(c);
        }
}

int xread(void) {
        if (Ports[Input_port] == NULL)
                return error("input port is not open", NOEXPR);
        Level = 0;
        return read_form();
}
```

The Printer

The "printer" is the part of the Scheme interpreter that implements the **`write`** procedure, which translates the internal (node) representation of a program into an external, readable representation.

The `print` function, that basically implements **`write`**, employs a lot of helper functions which follow below. These helper functions test the argument passed to them for a specific type. They return zero, if an object of *another* type was passed to them. When the type matches, they print the external representation of the object passed to them and return one.

```
/* Print bignum integer. */
int print_integer(int n) {
        int     first;
        char    buf[DIGITS_PER_WORD+2];

        if (Car[n] != S_integer) return 0;
        n = Cdr[n];
        first = 1;
        while (1) {
                if (n == NIL) break;
                if (first)
                        sprintf(buf, %d, Car[n]);
                else
                        sprintf(buf, %0*d, DIGITS_PER_WORD, Car[n]);
                pr(buf);
                n = Cdr[n];
                first = 0;
        }
        return -1;
}

void print(int n);

/* Print expressions of the form (QUOTE X) as ’X. */
int print_quoted(int n) {
        if (    Car[n] == S_quote &&
                Cdr[n] != NIL &&
                cddr(n) == NIL
        ) {
                pr("’");
                print(cadr(n));
                return 1;
        }
        return 0;
}

int print_procedure(int n) {
        if (Car[n] == S_procedure) {
                pr("#<procedure ");
                print(cadr(n));
        /*      pr(" ");                        */
        /*      print(caddr(n));                */
        /*      pr(" ");                        */
        /*      print(cdddr(n));                */
                pr(">");
                return -1;
        }
        return 0;
}

int print_char(int n) {
        char    b[2];
        int     c;

        if (Car[n] != S_char) return 0;
        if (!Displaying) pr("#\\");
        c = cadr(n);
        if (!Displaying && c == ’ ’) {
                pr("space");
        }
        else if (!Displaying && c == '\n') {
                pr("newline");
        }
        else {
                b[1] = 0;
                b[0] = c;
                pr(b);
        }
        return -1;
}

int print_string(int n) {
        char    b[2];
        int     k;
        char    *s;

        if (Car[n] != S_string) return 0;
        if (!Displaying) pr("\"");
        s = string(n);
        k = string_len(n);
        b[1] = 0;
        while (k) {
                b[0] = *s++;
                if (!Displaying)
                        if (b[0] == '"' || b[0] == '\\')
                                pr("\\");
                pr(b);
                k = k-1;
        }
        if (!Displaying) pr("\"");
        return -1;
}

int print_symbol(int n) {
        char    b[2];
        int     k;
        char    *s;

        if (Car[n] != S_symbol) return 0;
        s = string(n);
        k = string_len(n);
        b[1] = 0;
        while (k) {
                b[0] = *s++;
                pr(b);
                k = k-1;
        }
        return -1;
}
```

```
int print_primitive(int n) {
        if (Car[n] != S_primitive) return 0;
        pr("#<primitive ");
        print(cddr(n));
        pr(">");
        return -1;
}
```

R5RS Scheme cannot print syntax transformers, but S9fES can. This may be considered a bug.

```
int print_syntax(int n) {
        if (Car[n] != S_syntax) return 0;
        pr("#<syntax>");
        return -1;
}

int print_vector(int n) {
        int     *p;
        int     k;

        if (Car[n] != S_vector) return 0;
        pr("#(");
        p = vector(n);
        k = vector_len(n);
        while (k--) {
                print(*p++);
                if (k) pr(" ");
        }
        pr(")");
        return -1;
}

int print_port(int n) {
        char    buf[100];

        if (Car[n] != S_input_port && Car[n] != S_output_port)
                return 0;
        sprintf(buf, "#<%s-port %d>",
                Car[n] == S_input_port? "input": "output",
                cadr(n));
        pr(buf);
        return -1;
}

void print(int n) {
        if (Ports[Output_port] == NULL) {
                error("output port is not open", NOEXPR);
                return;
        }
        else if (n == NIL) {
                pr("()");
        }
        else if (n == ENDOFFILE) {
                pr("#<eof>");
        }
        else if (n == FALSE) {
                pr("#f");
        }
        else if (n == TRUE) {
                pr("#t");
        }
        else if (n == UNDEFINED) {
                pr("#<undefined>");
        }
        else if (n == UNSPECIFIC) {
                pr("#<unspecific>");
        }
        else {
                if (print_char(n)) return;
                if (print_procedure(n)) return;
                if (print_integer(n)) return;
                if (print_primitive(n)) return;
                if (print_quoted(n)) return;
                if (print_string(n)) return;
                if (print_symbol(n)) return;
                if (print_syntax(n)) return;
                if (print_vector(n)) return;
                if (print_port(n)) return;
                pr("(");
                while (n != NIL) {
                        print(Car[n]);
                        n = Cdr[n];
                        if (n != NIL && atom_p(n)) {
                                pr(" . ");
                                print(n);
                                n = NIL;
                        }
                        if (n != NIL) pr(" ");
                }
                pr(")");
        }
}
```

Interlude

This is a collection of utility functions.

```
int length(int n) {
        int     k = 0;

        while (n != NIL) {
                k++;
                n = Cdr[n];
        }
        return k;
}
```

Append b to a destructively (b replaces the trailing NIL of a).

```
int appendb(int a, int b) {
        int     p, last = NIL;

        if (a == NIL) return b;
        p = a;
        while (p != NIL) {
                if (atom_p(p)) fatal("append!: improper list");
                last = p;
                p = Cdr[p];
        }
        Cdr[last] = b;
        return a;
}
```

Create a fresh list that consists of the same objects as the original list. Only the nodes that form the "spine" of the list are allocated freshly (the upper and lower rows of the below figure are spines). Both the original list and the fresh list share the same members:

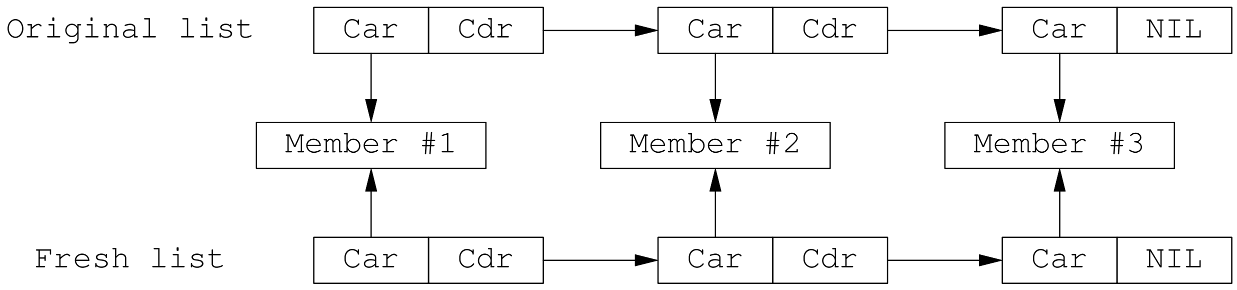


Fig. 10 – Shared list

```
int flat_copy(int n, int *lastp) {
        int     a, m, last;

        if (n == NIL) {
                lastp[0] = NIL;
                return NIL;
        }
        m = alloc(NIL, NIL);
        save(m);
        a = m;
        last = m;
        while (n != NIL) {
                Car[a] = Car[n];
                last = a;
                n = Cdr[n];
                if (n != NIL) {
                        Cdr[a] = alloc(NIL, NIL);
                        a = Cdr[a];
                }
        }
        unsave(1);
        lastp[0] = last;
        return m;
}
```

Check whether n is a valid argument list (either NIL or a single symbol or a (proper or improper) list of symbols).

```
int argument_list_p(int n) {
        if (n == NIL || symbol_p(n)) return 1;
        if (atom_p(n)) return 0;
        while (!atom_p(n)) {
                if (!symbol_p(Car[n])) return 0;
                n = Cdr[n];
        }
        return n == NIL || symbol_p(n);
}

int list_of_symbols_p(int n) {
        return !symbol_p(n) && argument_list_p(n);
}
```

The `rehash` function rebuilds the hash table of the environment rib `e`. A rib is an association list with a vector consed to it. The empty list **()** may replace the vector to indicate that an environment rib is not hashed. The following figure illustrates the layout of a hashed rib.

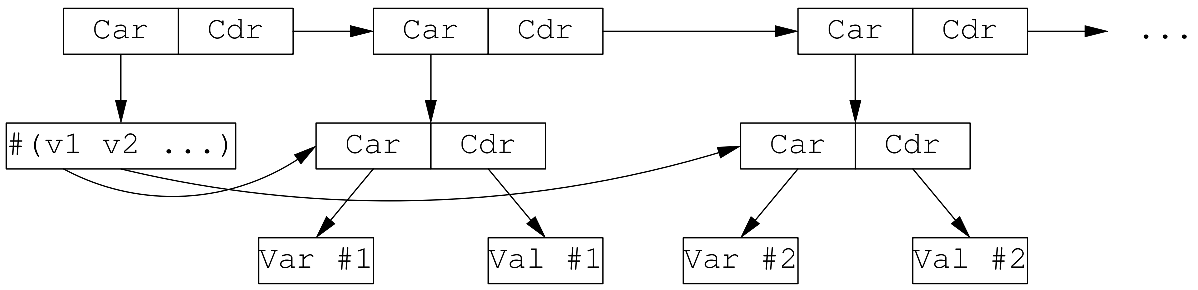


Fig. 11 – Hashed environment rib

The above rib is equivalent to the Scheme form

`(#((var1 . val1) (var2 . val2)) (var1 . val1) (var2 . val2))`

The vector in the first slot of the list shares its members with the following slots of the list. A value can be looked up in the rib by computing the hash value **h** of a variable name and then checking whether the **h**'th element of the hash table has the same variable as its key. If so, the variable is found in O(1) time. When the keys do not match, a hash collision occurred and the variable has to be looked up using linear search.

No hash tables are computed when the length of the rib is below `HASH_THRESHOLD`. This is done because computing hash values can be more expensive than just doing a linear search in small ribs.

Hash tables are computed only when a program is running. This is a hack to improve the startup time of the interpreter. Without this hack, the hash table would be rebuild each time a symbol is added to the global environment. With this hack, it is done only once after building the entire global environment.

The hash function used in `rehash` is simple and fast. It initializes the hash value **h** with zero. For each character of the variable name, it shifts **h** to the left by eight bits and adds the value of the character. The hash value is the resulting sum modulo the length of the rib.

```
void rehash(int e) {
        int             i, p, *v;
        unsigned int    h, k = length(e)-1;
        char            *s;

        if (Program == NIL || k < HASH_THRESHOLD) return;
        Car[e] = allocv(S_vector, k * sizeof(int));
        v = vector(Car[e]);
        for (i=0; i<k; i++) v[i] = NIL;
        p = Cdr[e];
        for (i=1; i<k; i++) {
                s = string(caar(p));
                h = 0;
                while (*s) h = (h<<8) + *s++;
                v[h%k] = Car[p];
                p = Cdr[p];
        }
}
```

Extend the environment rib `e` by adding an association of the variable `v` with the value `a` to it.

```
int extend(int v, int a, int e) {
        int     n;

        n = alloc(a, NIL);
        n = alloc(v, n);
        Cdr[e] = alloc(n, Cdr[e]);
        rehash(e);
        return e;
}
```

Create a new environment by adding a rib (a.k.a. "local environment") to the global environment. Return the new global environment. When the length of the rib is not smaller than `HASH_THRESHOLD`, attach a hash table to the rib.

```
int make_env(int rib, int env) {
        int     e;

        Tmp = env;
        rib = alloc(NIL, rib);
        e = alloc(rib, env);
        Tmp = NIL;
        if (length(rib) >= HASH_THRESHOLD) {
                save(e);
                rehash(rib);
                unsave(1);
        }
        return e;
}
```

Try to find the variable v in the local environment e using the attached hash table (if any). Return the association of the variable upon success and NIL in case of a hash collision or a non-existant hash table.

```
/* hash stats */
int coll = 0, hits = 0;

int try_hash(int v, int e) {
        int             *hv, p;
        unsigned int    h, k;
        char            *s;

        if (e == NIL || Car[e] == NIL) return NIL;
        hv = vector(Car[e]);
        k = vector_len(Car[e]);
        s = string(v);
        h = 0;
        while (*s) h = (h<<8) + *s++;
        p = hv[h%k];
        if (p != NIL && Car[p] == v) {
                hits++;
                return p;
        }
        coll++;
        return NIL;
}
```

Evaluator: Variable Lookup

Lookup a variable v in the environment env. The environment env is a list of environment ribs that will be searched in sequence.

```
int lookup(int v, int env) {
        int     e, n;

        while (env != NIL) {
                e = Car[env];
                n = try_hash(v, e);
                if (n != NIL) return n;
                while (e != NIL) {
                        if (v == caar(e)) return Car[e];
                        e = Cdr[e];
                }
                env = Cdr[env];
        }
        return NIL;
}
```

In fact the representation of environment ribs in figure 11 was slightly simplified, because values do not associate directly with values but with *boxes that contain values*. Here is a more accurate picture of an association:

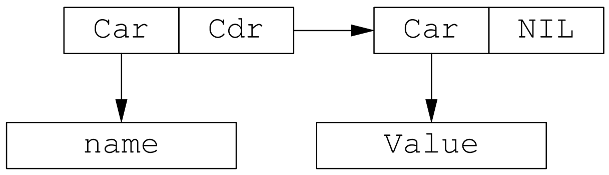


Fig. 12 – Association structure

Such an association is equivalent to the Scheme form

`((name . (value)))` or **`((name value))`**.

The additional list around the value is called the ''box'' of that value. This box serves as a location for storing values. This extra level of indirection is necessary to implement special forms like **`set!`** and **`letrec`** efficiently.

The `location_of()` function retrieves the box associated with a variable `v` in the environment `env`. If the variable is not bound to any box or the name of the variable is the name of a special form, an error is reported. Note that user-defined syntax transformers should be handled in the same way, but because this is only a small deviation from the standard, they are not.

```
int location_of(int v, int env) {
        int     n;

        n = lookup(v, env);
        if (n == NIL) {
                if (special_p(v))
                        error("bad syntax", v);
                else
                        error("symbol not bound", v);
                return NIL;
        }
        return Cdr[n];
}

int value_of(int v, int env) {
        int     n;

        n = location_of(v, env);
        return n == NIL? NIL: Car[n];
}
```

Evaluator: Special Form Handling

These helpers are called whenever a special form receives an incorrect number of arguments.

```
int too_few_args(int n) {
        return error("too few arguments", n);
}

int too_many_args(int n) {
        return error("too many arguments", n);
}
```

A “special form handler” is a function that handles the interpretation of a special form. “Special forms” are those forms that constitute the primitive syntax of Scheme. They look like applications of keywords like **`lambda`**, **`define`**, **`if`**, and **`set!`**.

Each special form handler receives three arguments: the special form `x` and two `int` pointers named `pc` and `ps`. It rewrites the form `x` in a way that is specific to the special form handler and returns it. The pointers `pc` and `ps` are used to control what the core of the evaluator does with the rewritten form. `Ps` is the new state of the evaluator and `pc` is the “continue” flag. Setting the `pc` flag signals the evaluator that the returned form is an expression rather than a value. In this case the evaluation of the form must continue. Hence the name of this flag.

Special form handlers are also responsible for checking the syntax of the forms passed to them.

The parameters `state` and `neutral` are specific to the `make_sequence()` handler. This functions handles the **`and`**, **`or`**, and **`begin`** special forms. All these forms are similar. They differ only by their neutral element and the new state of the evaluator. These parameters are passed to `make_sequence()` using the `state` and `neutral` arguments.

When the special form passed to `make_sequence()` has no arguments (like **`(or)`**), the handler returns the neutral element immediately. Neither `pc` nor `ps` are changed, because the neutral element is a value and not an expression. When the form has one argument (like **`(begin 'foo)`**), the handler returns that argument and sets the continue flag so that the argument (which is an expression) is evaluated, too.

When the form has multiple arguments, the handler saves the tail of the argument list on the stack for later evaluation, sets a new evaluator state, and sets the continue flag to 2. This tells the evaluator to keep the new state. It finally returns the first argument of the received form.

Here is an example: When `make_sequence()` receives **`(and a b c)`**, it pushes **`(b c)`** to the stack, sets the state to `MCONJ` and the continue flag to 2, and returns **`a`**. The new state `MCONJ` tells the evaluator to run the code for **`and`**.

```
/* Set up sequence for AND, BEGIN, OR. */
int make_sequence(int state, int neutral, int x, int *pc, int *ps) {
        if (Cdr[x] == NIL) {
                return neutral;
        }
        else if (cddr(x) == NIL) {
                *pc = 1;
                return cadr(x);
        }
        else {
                *pc = 2;
                *ps = state;
                save(Cdr[x]);
                return cadr(x);
        }
}

#define sf_and(x, pc, ps) \
        make_sequence(MCONJ, TRUE, x, pc, ps)

#define sf_begin(x, pc, ps) \
        make_sequence(MBEGN, UNSPECIFIC, x, pc, ps)
```

`Cond` handler. Syntax: **`(cond (pred body ...) ...)`**
Push the list of clauses passed to **`cond`**, change the state to `MCOND`, and return the predicate of the first clause. If there are no clauses, return `UNSPECIFIC` without changing the state.

```
int sf_cond(int x, int *pc, int *ps) {
        int     clauses, p;

        clauses = Cdr[x];
        p = clauses;
        while (p != NIL) {
                if (atom_p(p) || atom_p(Car[p]) || atom_p(cdar(p)))
                        return error("cond: bad syntax", p);
                p = Cdr[p];
        }
        if (clauses == NIL) return UNSPECIFIC;
        if (caar(clauses) == S_else && Cdr[clauses] == NIL) {
                p = alloc(TRUE, cdar(clauses));
                clauses = alloc(p, Cdr[clauses]);
        }
        save(clauses);
        *pc = 2;
        *ps = MCOND;
        return caar(clauses);
}
```

`If` handler. Syntax: **`(if pred consequent alternative)`**
Push the argument list, change the state to `MIFPR`, and return the predicate. If there is no alternative, create one that has an unspecific value.

```
int sf_if(int x, int *pc, int *ps) {
        int     m;
        m = Cdr[x];
        if (m == NIL || Cdr[m] == NIL)
                return too_few_args(x);
        if (cddr(m) != NIL && cdddr(m) != NIL)
                return too_many_args(x);
        if (cddr(m) == NIL)
                cddr(m) = alloc(UNSPECIFIC, NIL);
        save(m);
        *pc = 2;
        *ps = MIFPR;
        return Car[m];
}
```

Create an internal temporary variable for each member of the list `x`. Internal variable names consist of two leading # signs and a number, so they cannot be read by the Scheme reader. `make_temporaries()` returns a list of the generated variables. For example, for `x = `**`(a b c)`** it would return **`(##2 ##1 ##0)`**.

The `make_temporaries()` function is used to create temporary variables for **`letrec`**.

```
int make_temporaries(int x) {
        int     n, v, k = 0;
        char    buf[10];

        n = NIL;
        save(n);
        while (x != NIL) {
                sprintf(buf, "##%d", k);
                v = add_symbol(buf);
                n = alloc(v, n);
                Car[Stack] = n;
                x = Cdr[x];
                k++;
        }
        unsave(1);
        return n;
}
```

The `make_assignments()` function accepts two lists of variables and generates a **`begin`** form whose body binds each variable of the list `t` to the corresponding variable of the list `x`. For `x = `**`(a b c)`** and `t = `**`(##0 ##1 ##2)`**, the function would generate the form

```
(begin (set! c ##2)
       (set! b ##1)
       (set! a ##0))
```

```
int make_assignments(int x, int t) {
        int     n, asg;

        n = NIL;
        save(n);
        while (x != NIL) {
                asg = alloc(Car[t], NIL);
                asg = alloc(Car[x], asg);
                asg = alloc(S_set_b, asg);
                n = alloc(asg, n);
                Car[Stack] = n;
                x = Cdr[x];
                t = Cdr[t];
        }
        unsave(1);
        return alloc(S_begin, n);
}

int make_undefineds(int x) {
        int     n;

        n = NIL;
        while (x != NIL) {
                n = alloc(UNDEFINED, n);
                x = Cdr[x];
        }
        return n;
}
```

The `make_recursive_lambda()` function creates a form that binds recursive procedures using **`lambda`** and **`set!`**. It is used to rewrite applications of **`letrec`** and local **`define`**s.

Given a list of variables `v`, a list of (actual) arguments `a`, and a procedure body, the function creates the following form:

```
((lambda (v1 ...)
   ((lambda (t1 ...)
      (begin (set! v1 t1)
             ...
             body))
    a1 ...))
 #<undefined> ...)
```

To verify that this form is indeed equivalent to **`letrec`** is left as an exercise to the reader.

```
int make_recursive_lambda(int v, int a, int body) {
        int     t, n;

        t = make_temporaries(v);
        save(t);
        body = appendb(make_assignments(v, t), body);
        body = alloc(body, NIL);
        n = alloc(t, body);
        n = alloc(S_lambda, n);
        n = alloc(n, a);
        n = alloc(n, NIL);
        n = alloc(v, n);
        n = alloc(S_lambda, n);
        save(n);
        n = alloc(n, make_undefineds(v));
        unsave(2);
        return n;
}
```

The `extract_from_let()` function is used to extract variables or arguments from **`let`** and **`letrec`** expressions. The `caller` parameter indicates the type of expression that is being processed. `X` is the list of bindings of the expression. When `part` equals `VARS`, `extract_from_let()` extracts variables, and when it equals `ARGS`, the function extracts arguments.

Given the list of bindings `x = ` **`((a 1) (b 2) (c 3))`** and `part = VARS`, the function returns **`(c b a)`**. With `part = ARGS`, it returns **`(3 2 1)`**.

```
#define VARS 1
#define ARGS 2

/* Extract variables or arguments from LET/LETREC. */
int extract_from_let(int caller, int x, int part) {
        int     a, n;
        char    *err;

        err = caller == S_let? "let: bad syntax":
                                "letrec: bad syntax";
        a = NIL;
        while (x != NIL) {
                if (atom_p(x)) return error(err, x);
                n = Car[x];
                if (atom_p(n) || Cdr[n] == NIL || cddr(n) != NIL)
                        return error(err, x);
                a = alloc(part==VARS? caar(x): cadar(x), a);
                x = Cdr[x];
        }
        return a;
}
```

The `extract_from_defines()` function performes basically the same task as `extract_from_let()`, but instead of lists of bindings it processes a list of **`define`** forms. It is used to rewrite local **`define`**s as **`lambda`** expressions. The `restp` pointer is used to return the rest of the list `x`, which begins at the first non-**`define`**, back to the caller. Given the list

```
((define x 1)
 (define y 2)
 (+ x y))
```

`extract_from_defines()` would return **`((+ x y))`** through the `restp` parameter.

```
/* Extract variables or arguments from a set of DEFINEs. */
int extract_from_defines(int x, int part, int *restp) {
        int     a, n;
        a = NIL;
        while (x != NIL) {
                if (atom_p(x) || atom_p(Car[x]) || caar(x) != S_define)
                        break;
                n = Car[x];
                if (length(n) != 3)
                        return error("define: bad syntax", n);
                if (pair_p(cadr(n))) {
                        /* (define (proc vars) ...) */
                        if (part == VARS) {
                                a = alloc(caadr(n), a);
                        }
                        else {
                                a = alloc(NIL, a);
                                save(a);
                                Car[a] = alloc(cdadr(n), cddr(n));
                                Car[a] = alloc(S_lambda, Car[a]);
                                unsave(1);
                        }
                }
                else {
                        a = alloc(part==VARS? cadr(n): caddr(n), a);
                }
                x = Cdr[x];
        }
        *restp = x;
        return a;
}
```

The `resolve_local_defines()` function rewrites **`define`**s that occur at the beginning of the body of a **`lambda`** function. It replaces the local **`define`**s with applications of **`lambda`** and **`set!`** that resemble **`letrec`**. See `make_recursive_lambda()` for further details.

The transformation performed by `resolve_local_defines()` is basically the following (but it uses a combination of **`lambda`** and **`set!`** in the place of **`letrec`** to avoid an extra transformation):

```
(define (f)        ===>  (define (f)
  (define a1 v1)           (letrec ((a1 v1)
  ...                               ...
  (define aN vN)                    (aN vN))
  body)                      body))
```

```
/* Rewrite local defines using LAMBDA and SET! */
int resolve_local_defines(int x) {
        int     v, a, n, rest;

        a = extract_from_defines(x, ARGS, &rest);
        if (Error_flag) return NIL;
        save(a);
        v = extract_from_defines(x, VARS, &rest);
        save(v);
        if (rest == NIL) rest = alloc(UNSPECIFIC, NIL);
        save(rest);
        n = make_recursive_lambda(v, a, rest);
        unsave(3);
        return n;
}
```

`Lambda` handler. Syntax: **`(lambda vars body ...)`**
Create a new procedure from a lambda expression:

`(lambda vars body ...)  ===>  #<procedure vars>`

where **`#<procedure vars>`** is an atomic type with the following structure:

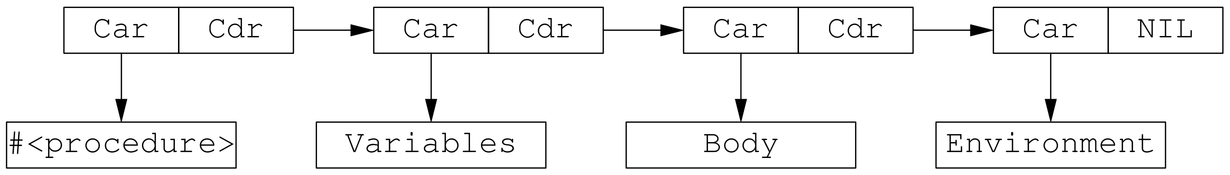


Fig. 13 – Procedure structure

The “variables” and “body” are the variables and the body of the lambda expression. “Environment” is the environment that was in effect at the time of the creation of the procedure. In Scheme speak it is called the “lexical environment” of the procedure.

When the body of the lambda function consists of multiple expressions, these expressions are combined in a single **`begin`**. When there are local **`define`**s at the beginning of the body, they are converted using `resolve_local_defines()`.

The `sf_lambda()` handler does not have a `pc` or `ps` parameter, because it always returns a value that does not require further evaluation.

```
int sf_lambda(int x) {
        int     n, k;

        k = length(x);
        if (k < 3) return too_few_args(x);
        if (!argument_list_p(cadr(x)))
                return error("bad argument list", cadr(x));
        if (pair_p(caddr(x)) && caaddr(x) == S_define)
                n = resolve_local_defines(cddr(x));
        else if (k > 3)
                n = alloc(S_begin, cddr(x));
        else
                n = caddr(x);
        n = alloc(n, Environment);
        n = alloc(cadr(x), n);
        return alloc3(S_procedure, n, AFLAG);
}
```

`Let` handler. Syntax: **`(let ((var arg) ...) body ...)`**
Rewrite a **`let`** expression `x` in terms of **`lambda`**:

```
(let ((v1 a1)    ===>  ((lambda (v1 ... vN)
      ...                  body)
      (vN aN))           a1 ... aN)
  body)
```

This special form handler does not change the state of the evaluator but does require re-evaluation of its return value. Hence it sets `pc`=1.

```
/* Transform LET to LAMBDA */
int sf_let(int x, int *pc) {
        int     v, a, b;
        int     n, e;

        if (length(x) < 3) too_few_args(x);
        e = cadr(x);
        a = extract_from_let(S_let, e, ARGS);
        if (Error_flag) return NIL;
        save(a);
        v = extract_from_let(S_let, e, VARS);
        b = cddr(x);
        n = alloc(v, b);
        n = alloc(S_lambda, n);
        n = alloc(n, a);
        unsave(1);
        *pc = 1;
        return n;
}
```

`Letrec` handler. Syntax: **`(letrec ((var arg) ...) body ...)`**
Rewrite the **`letrec`** expression `x` in terms of **`lambda`** and **`set!`**:

```
(letrec ((v1 a1)  ===>  ((lambda (v1 ... vN)
         ...               ((lambda (t1 ... tN)
         (vN aN))             (begin (set! v1 t1)
  body)                              ...
                                     (set! vN tN)
                                     body))
                            a1 ... aN))
                         #<undefined> ...)
```

Like `sf_let()`, `sf_letrec()` does not change the state of the evaluator, but requires its return value to be re-evaluated. Hence it sets `pc`=1.

```
/* Transform LETREC to LAMBDA and SET! */
int sf_letrec(int x, int *pc) {
        int     v, a;
        int     n, e;

        if (length(x) < 3) too_few_args(x);
        e = cadr(x);
        a = extract_from_let(S_letrec, e, ARGS);
        if (Error_flag) return NIL;
        save(a);
        v = extract_from_let(S_letrec, e, VARS);
        save(v);
        n = make_recursive_lambda(v, a, cddr(x));
        unsave(2);
        *pc = 1;
        return n;
}
```

`Quote` handler. Syntax: **`(quote form)`**
Return the argument of `x`. No re-evaluation required.

```
int sf_quote(int x) {
        int     k = length(x);

        if (k < 2) return too_few_args(x);
        if (k > 2) return too_many_args(x);
        return cadr(x);
}
```

See `make_sequence()` for details about the **`or`** form.

```
#define sf_or(x, pc, ps) \
        make_sequence(MDISJ, FALSE, x, pc, ps)
```

`Set!` handler. Syntax: **`(set! symbol expr)`**
Push the location bound to `symbol`, change the evaluator state to `MSETV`, and return `expr`.

```
int sf_set_b(int x, int *pc, int *ps) {
        int     n, k;

        k = length(x);
        if (k < 3) return too_few_args(x);
        if (k > 3) return too_many_args(x);
        if (!symbol_p(cadr(x)))
                return error("set!: symbol expected", cadr(x));
        n = location_of(cadr(x), Environment);
        if (Error_flag) return NIL;
        save(n);
        *pc = 2;
        *ps = MSETV;
        return caddr(x);
}
```

Find the variable `v` in the environment rib `e`.

```
int find_local_variable(int v, int e) {
        while (e != NIL) {
                if (v == caar(e)) return Car[e];
                e = Cdr[e];
        }
        return NIL;
}
```

`Define` handler. Syntax: **`(define name expr)`**
or **`(define (name var ...) body ...)`**
The handler first rewrites procedure definitions of the form

`(define (name var ...) body ...)`

to

`(define name (lambda (var ...) body ...))`

Then it attempts to locate a local variable with the given name. When one exists, it will be used to bind the value. Otherwise a new variable is created in the innermost context. Finally, `sf_define()` saves the expression argument on the stack and changes the evaluator state. When **`define`** is used to create a new procedure, the state is changed to `MDEFN` and else it is changed to `MSETV`. The handler returns the expression argument for further evaluation.

The difference between `MSETV` and `MDEFN` is that `MDEFN` causes the procedure that will be bound to the given name to employ *dynamic scoping* rather than lexical scoping. Dynamic scoping allows global **`define`**s to be mutually recursive.

```
int sf_define(int x, int *pc, int *ps) {
        int     v, a, n, k;

        if (Car[State_stack] == MARGS)
                return error("define: bad local context", x);
        k = length(x);
        if (k < 3) return too_few_args(x);
        if (symbol_p(cadr(x)) && k > 3) return too_many_args(x);
        if (!argument_list_p(cadr(x)))
                return error("define: expected symbol or list, got", cadr(x));
        if (!symbol_p(cadr(x))) {
                a = cddr(x);
                a = alloc(cdadr(x), a);
                a = alloc(S_lambda, a);
                Tmp = a;
                n = caadr(x);
        }
        else {
                a = caddr(x);
                n = cadr(x);
        }
        v = find_local_variable(n, Car[Environment]);
        if (v == NIL) {
                Car[Environment] = extend(n, UNDEFINED, Car[Environment]);
                v = cadar(Environment);
        }
        save(Cdr[v]);
        *pc = 2;
        if (!atom_p(a) && Car[a] == S_lambda)
                *ps = MDEFN;    /* use dynamic scoping */
        else
```

```
                *ps = MSETV;
        Tmp = NIL;
        return a;
}
```

`Define-syntax` handler. Syntax: **`(define-syntax name syntax-transformer)`**
The `sf_define_synatx()` handler creates a symbol with the given name if it does not already exist. It pushes the location bound to that name, changes the evaluator state to `MDSYN`, and returns the `syntax-transformer` argument. Valid syntax-transformers are created by `sf_syntax_rules()` below.

```
int sf_define_syntax(int x, int *pc, int *ps) {
        int     a, n, v, k = length(x);

        if (k < 3) return too_few_args(x);
        if (k > 3) return too_many_args(x);
        if (!symbol_p(cadr(x)))
                return error("define-syntax: expected symbol, got", cadr(x));
        a = caddr(x);
        n = cadr(x);
        v = lookup(n, Environment);
        if (v == NIL) {
                Car[Environment] = extend(n, UNDEFINED, Car[Environment]);
                v = cadar(Environment);
        }
        save(Cdr[v]);
        *pc = 2;
        *ps = MDSYN;
        return a;
}
```

`Syntax-rules` handler.
Syntax: **`(syntax-rules (keyword ...) ((pattern template) ...))`**
This handler merely checks the syntax of the form that it receives. It returns a **`#<syntax>`** object which has the following structure:

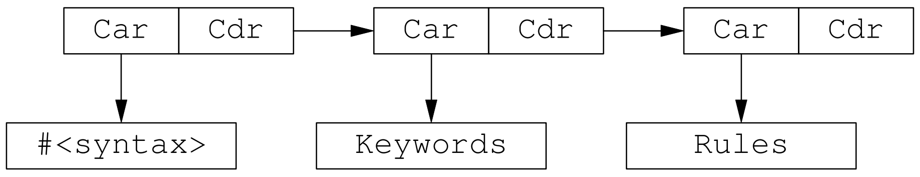


Fig. 14 – Syntax transformer structure

"Keywords" and "rules" are simply the arguments passed to **`syntax-rules`**. The handler merely creates a new object by attaching the **`#<syntax>`** type tag to the tail of the received form.

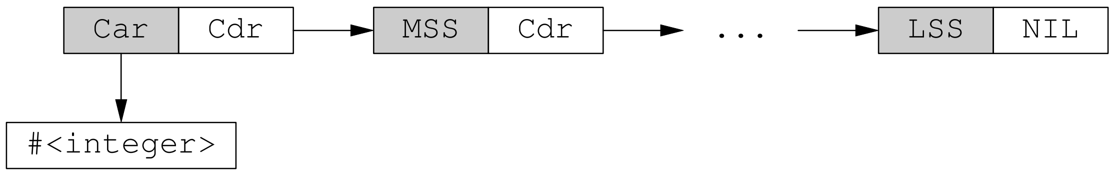
Because `sf_syntax_rules()` returns a value which does not require further evaluation, it does not change the `ps` or `pc` flags.

```
int sf_syntax_rules(int x) {
        int     m, cl, k = length(x);
        m = Cdr[x];
        if (k < 3) return too_few_args(x);
        if (!list_of_symbols_p(Car[m]))
                return error("syntax-rules: expected list of symbols, got",
                        Car[m]);
        cl = Cdr[m];
        while (cl != NIL) {
                if (atom_p(cl))
                        return error("syntax-rules: improper list of rules",
                                Cdr[m]);
                if (atom_p(Car[cl]) || atom_p(cdar(cl)))
                        return error("syntax-rules: bad clause", Car[cl]);
                cl = Cdr[cl];
        }
        return alloc3(S_syntax, m, AFLAG);
}
```

Interlude: Bignum Arithmetics

The bignum arithmetic functions of S9fES use the algorithms that most human beings would use on a sheet of paper. However, instead of using decimal numbers, it uses numbers with a base of `INT_SEG_LIMIT`. Only when multiplying and dividing numbers, it uses a hybrid method of base-10 and base-`INT_SEG_LIMIT` arithmetics in order to reduce the number of operations.

The bignum operators work on lists of integer segments internally. A list of integer segments is obtained by performing a **`cdr`** operation on a bignum integer. Remember that bignums have the following internal structure:



So removing the head of the list (which contains the **`#<integer>`** type tag) leaves a list of integer segments, which may be considered a representation of a base-`INT_SEG_LIMIT` number.

The `make_integer()` function creates a single-segment bignum integer with the given value `i`.

```
int make_integer(int i) {
        int     n;

        n = alloc3(i, NIL, AFLAG);
        return alloc3(S_integer, n, AFLAG);
}
```

`integer_value()` returns the value of a single-segment bignum integer `x`. If `x` has more than one integer segment, it reports an error.

```
int integer_value(char *src, int x) {
        char    msg[100];

        if (cddr(x) != NIL) {
                sprintf(msg, "%s: integer argument too big", src);
                error(msg, x);
                return 0;
        }
        return cadr(x);
}
```

Create a fresh bignum with the absolute value of `a`. The new bignum shares all but the first integer segment with the original bignum.

```
int bignum_abs(int a) {
        int     n;

        n = alloc3(abs(cadr(a)), cddr(a), AFLAG);
        return alloc3(S_integer, n, AFLAG);
}
```

Create a fresh bignum with the negative value of `a`. The new bignum shares all but the first integer segment with the original bignum.

```
int bignum_negate(int a) {
        int     n;

        n = alloc3(-cadr(a), cddr(a), AFLAG);
        return alloc3(S_integer, n, AFLAG);
}

#define bignum_negative_p(a) ((cadr(a)) < 0)

#define bignum_zero_p(a) ((cadr(a) == 0) && (cddr(a)) == NIL)
```

Create a fresh list containing the segments of the bignum integer in reverse order. The parameter `n` is a list of integer segments.

```
int reverse_segments(int n) {
        int     m;

        m = NIL;
        save(m);
        while (n != NIL) {
                m = alloc3(Car[n], m, AFLAG);
                Car[Stack] = m;
                n = Cdr[n];
        }
        unsave(1);
        return m;
}
```

```
int bignum_add(int a, int b);
int bignum_subtract(int a, int b);
```

The `_bignum_add()` function performs the addition of bignum integers. The function performs only additions of non-negative operands. Operations with one or more negative operands are rewritten as follows:

sum	**is rewritten to**
-a + -b	-(|a| + |b|)
-a + b	b - |a|
a + -b	a - |b|

`_bignum_add()` returns a fresh bignum integer whose value is the sum of the parameters `a` and `b`.

```
int _bignum_add(int a, int b) {
        int     fa, fb, carry, r;
        int     result;

        if (bignum_negative_p(a)) {
                if (bignum_negative_p(b)) {
                        /* -A+-B --> -(|A|+|B|) */
                        a = bignum_abs(a);
                        save(a);
                        b = bignum_abs(b);
                        save(b);
                        a = bignum_add(a, b);
                        unsave(2);
                        return bignum_negate(a);
                }
                else {
                        /* -A+B --> B-|A| */
                        a = bignum_abs(a);
                        save(a);
                        a = bignum_subtract(b, a);
                        unsave(1);
                        return a;
                }
        }
        else if (bignum_negative_p(b)) {
                /* A+-B --> A-|B| */
                b = bignum_abs(b);
                save(b);
                a = bignum_subtract(a, b);
                unsave(1);
                return a;
        }
        /* A+B */
        a = reverse_segments(Cdr[a]);
        save(a);
        b = reverse_segments(Cdr[b]);
        save(b);
        carry = 0;
        result = NIL;
        save(result);
        while (a != NIL || b != NIL || carry) {
                fa = a==NIL? 0: Car[a];
                fb = b==NIL? 0: Car[b];
                r = fa + fb + carry;
                carry = 0;
                if (r >= INT_SEG_LIMIT) {
                        r -= INT_SEG_LIMIT;
                        carry = 1;
                }
                result = alloc3(r, result, AFLAG);
                Car[Stack] = result;
                if (a != NIL) a = Cdr[a];
                if (b != NIL) b = Cdr[b];
        }
        unsave(3);
        return alloc3(S_integer, result, AFLAG);
}
```

`bignum_add()` protects ist arguments before passing them to `_bignum_add()` which performs the add operation. This is done because `_bignum_add()` has multiple `return` points, and the operands would have to be unprotected at each of them.

```
int bignum_add(int a, int b) {
        Tmp = b;
        save(a);
        save(b);
        Tmp = NIL;
        a = _bignum_add(a, b);
        unsave(2);
        return a;
}
```

Return one if the bignum `a` has a smaller value than the bignum `b` and else return zero.

```
int bignum_less_p(int a, int b) {
        int     ka, kb, neg_a, neg_b;

        neg_a = bignum_negative_p(a);
        neg_b = bignum_negative_p(b);
        if (neg_a && !neg_b) return 1;
        if (!neg_a && neg_b) return 0;
        ka = length(a);
        kb = length(b);
        if (ka < kb) return neg_a? 0: 1;
        if (ka > kb) return neg_a? 1: 0;
        Tmp = b;
        a = bignum_abs(a);
        save(a);
        b = bignum_abs(b);
        unsave(1);
        Tmp = NIL;
        a = Cdr[a];
        b = Cdr[b];
        while (a != NIL) {
                if (Car[a] < Car[b]) return neg_a? 0: 1;
                if (Car[a] > Car[b]) return neg_a? 1: 0;
                a = Cdr[a];
                b = Cdr[b];
        }
        return 0;
}
```

Return one if the bignum a has the same value as the bignum b and else return zero.

```
int bignum_equal_p(int a, int b) {
        a = Cdr[a];
        b = Cdr[b];
        while (a != NIL && b != NIL) {
                if (Car[a] != Car[b]) return 0;
                a = Cdr[a];
                b = Cdr[b];
        }
        return a == NIL && b == NIL;
}
```

The _bignum_subtract() function performs the subtraction of bignum integers. It computes only differences between non-negative operands. Operations with one or more negative operands are rewritten as follows:

difference	**is rewritten to**
−a − −b	|b| − |a|
−a − b	−(|a| + b)
a − −b	a + |b|

_bignum_subtract() returns a fresh bignum integer whose value is the difference between the parameters a and b.

```
int _bignum_subtract(int a, int b) {
        int     fa, fb, borrow, r;
        int     result;

        if (bignum_negative_p(a)) {
                if (bignum_negative_p(b)) {
                        /* -A--B --> -A+|B| --> |B|-|A| */
                        a = bignum_abs(a);
                        save(a);
                        b = bignum_abs(b);
                        save(b);
                        a = bignum_subtract(b, a);
                        unsave(2);
                        return a;
                }
                else {
                        /* -A-B --> -(|A|+B) */
                        a = bignum_abs(a);
                        save(a);
                        a = bignum_add(a, b);
                        unsave(1);
                        return bignum_negate(a);
                }
        }
        else if (bignum_negative_p(b)) {
                /* A--B --> A+|B| */
                b = bignum_abs(b);
                save(b);
                a = bignum_add(a, b);
                unsave(1);
                return a;
        }
        /* A-B, A<B --> -(B-A) */
        if (bignum_less_p(a, b))
                return bignum_negate(bignum_subtract(b, a));
        /* A-B, A>=B */
        a = reverse_segments(Cdr[a]);
        save(a);
        b = reverse_segments(Cdr[b]);
        save(b);
        borrow = 0;
        result = NIL;
        save(result);
        while (a != NIL || b != NIL || borrow) {
                fa = a==NIL? 0: Car[a];
                fb = b==NIL? 0: Car[b];
                r = fa - fb - borrow;
                borrow = 0;
                if (r < 0) {
                        r += INT_SEG_LIMIT;
                        borrow = 1;
                }
                result = alloc3(r, result, AFLAG);
                Car[Stack] = result;
                if (a != NIL) a = Cdr[a];
                if (b != NIL) b = Cdr[b];
        }
        unsave(3);
        while (Car[result] == 0 && Cdr[result] != NIL)
                result = Cdr[result];
        return alloc3(S_integer, result, AFLAG);
}
```

`bignum_subtract()` protects its arguments before passing them to `_bignum_subtract()` which performs the subtract operation. This is done because `_bignum_subtract()` has multiple `return` points, and the operands would have to be unprotected at each of them.

```
int bignum_subtract(int a, int b) {
        Tmp = b;
        save(a);
        save(b);
        Tmp = NIL;
        a = _bignum_subtract(a, b);
        unsave(2);
        return a;
}
```

Create a new bignum with the digits of `a` shifted to the left by one decimal digit. The rightmost digit of the fresh bignum is filled with `fill`.

```
int bignum_shift_left(int a, int fill) {
        int     r, carry, c;
        int     result;

        a = reverse_segments(Cdr[a]);
        save(a);
        carry = fill;
        result = NIL;
        save(result);
        while (a != NIL) {
                if (Car[a] >= INT_SEG_LIMIT/10) {
                        c = Car[a] / (INT_SEG_LIMIT/10);
                        r = Car[a] % (INT_SEG_LIMIT/10) * 10;
                        r += carry;
                        carry = c;
                }
                else {
                        r = Car[a] * 10 + carry;
                        carry = 0;
                }
                result = alloc3(r, result, AFLAG);
                Car[Stack] = result;
                a = Cdr[a];
        }
        if (carry) result = alloc3(carry, result, AFLAG);
        unsave(2);
        return alloc3(S_integer, result, AFLAG);
}
```

`bignum_shift_right()` creates a new bignum with the digits of `a` shifted to the right by one decimal digit. It returns a pair containing the new bignum integer and the rightmost digit of the original value (the digit that "fell off" when shifting to the right). For example, `a = 12345` would give **`(1234 . 5)`**.

```
/* Result: (a/10 . a%10) */
int bignum_shift_right(int a) {
        int     r, carry, c;
        int     result;

        a = Cdr[a];
        save(a);
        carry = 0;
        result = NIL;
        save(result);
        while (a != NIL) {
                c = Car[a] % 10;
                r = Car[a] / 10;
                r += carry * (INT_SEG_LIMIT/10);
                carry = c;
                result = alloc3(r, result, AFLAG);
                Car[Stack] = result;
                a = Cdr[a];
        }
        result = reverse_segments(result);
        if (Car[result] == 0 && Cdr[result] != NIL) result = Cdr[result];
        result = alloc3(S_integer, result, AFLAG);
        Car[Stack] = result;
        carry = make_integer(carry);
        unsave(2);
        return alloc(result, carry);
}
```

Create a fresh bignum integer whose value is the product of the bignums `a` and `b`.

```
int bignum_multiply(int a, int b) {
        int     neg, result, r, i;

        neg = bignum_negative_p(a) != bignum_negative_p(b);
        a = bignum_abs(a);
        save(a);
        b = bignum_abs(b);
        save(b);
        result = make_integer(0);
        save(result);
        while (!bignum_zero_p(a)) {
                r = bignum_shift_right(a);
                i = caddr(r);
                a = Car[r];
                caddr(Stack) = a;
                while (i) {
                        result = bignum_add(result, b);
                        Car[Stack] = result;
                        i--;
                }
                b = bignum_shift_left(b, 0);
                cadr(Stack) = b;
        }
        if (neg) result = bignum_negate(result);
        unsave(3);
        return result;
}
```

The `bignum_equalize()` procedure prepares two bignums for division. It multiplies the divisor by 10 (by shifting it to the left) until one more multiplication would make it greater than the divident. It returns a pair containing the scaled divisor and the number by which the divisor was multiplied. For example, for a divsor `a = 123` and a divident `b = 12345`, the function would return **`(12300 . 100)`**.

```
int bignum_equalize(int a, int b) {
        int     r, f, r0, f0;

        r0 = a;
        save(r0);
        f0 = make_integer(1);
        save(f0);
        r = r0;
        save(r);
        f = f0;
        save(f);
        while (bignum_less_p(r, b)) {
                cadddr(Stack) = r0 = r;
                caddr(Stack) = f0 = f;
                r = bignum_shift_left(r, 0);
                cadr(Stack) = r;
                f = bignum_shift_left(f, 0);
                Car[Stack] = f;
        }
        unsave(4);
        return alloc(r0, f0);
}
```

Create a pair of bignum integers whose values are the quotient and the disivion remainder of the bignums `a` and `b`. Yes, this function is complex and does a lot of stack acrobatics. Sorry about that.

```
/* Result: (a/b . a%b) */
int _bignum_divide(int a, int b) {
        int     neg, neg_a, result, f;
        int     i, c, c0;

        neg_a = bignum_negative_p(a);
        neg = neg_a != bignum_negative_p(b);
        a = bignum_abs(a);
        save(a);
        b = bignum_abs(b);
        save(b);
        if (bignum_less_p(a, b)) {
                if (neg_a) a = bignum_negate(a);
                unsave(2);
                return alloc(make_integer(0), a);
        }
        b = bignum_equalize(b, a);
        cadr(Stack) = b; /* cadddddr */
        Car[Stack] = a; /* caddddr */
        c = NIL;
        save(c);        /* cadddr */
        c0 = NIL;
        save(c0);       /* caddr */
        f = Cdr[b];
        b = Car[b];
        cadddr(Stack) = b;
        save(f);        /* cadr */
        result = make_integer(0);
        save(result);   /* Car */
        while (!bignum_zero_p(f)) {
                c = make_integer(0);
                cadddr(Stack) = c;
                caddr(Stack) = c0 = c;
                i = 0;
                while (!bignum_less_p(a, c)) {
                        caddr(Stack) = c0 = c;
                        c = bignum_add(c, b);
                        cadddr(Stack) = c;
                        i++;
                }
                result = bignum_shift_left(result, i-1);
                Car[Stack] = result;
                a = bignum_subtract(a, c0);
                Car[cddddr(Stack)] = a;
                f = Car[bignum_shift_right(f)];
                cadr(Stack) = f;
                b = Car[bignum_shift_right(b)];
                cadr(cddddr(Stack)) = b;
        }
        if (neg) result = bignum_negate(result);
        Car[Stack] = result;
        if (neg_a) a = bignum_negate(a);
        unsave(6);
        return alloc(result, a);
}

int bignum_divide(int x, int a, int b) {
        if (bignum_zero_p(b))
                return error("divide by zero", x);
        Tmp = b;
        save(a);
        save(b);
        Tmp = NIL;
        a = _bignum_divide(a, b);
        unsave(2);
        return a;
}
```

Evaluator: Primitive Handlers

A primitive procedure of S9fES is a Scheme procedure that is implemented in C.

This section discusses the primitive procedures of S9fES. Each primitive handler receives an expression and returns its value. The arguments to primitives already are evaluated at this point. The arguments also have been type-checked when they are passed to the handlers, so the handler can assume the correct argument types and number of arguments.

(apply proc [expr ...] list) --> form
In fact, this procedure does not perform a function application, but merely rewrites
(apply proc [expr ...] list) to **(proc [expr ...] . list)** and returns it. The evaluator then re-evaluates the returned expression. So, strictly speaking, **apply** is not an ordinary primitive procedure, but a hybrid (half procedure, half special form handler).

```
int pp_apply(int x) {
        int     m, p, q, last;
        char    *err = "apply: improper argument list";

        m = Cdr[x];
        p = Cdr[m];
        last = p;
        while (p != NIL) {
                if (atom_p(p)) return error(err, x);
                last = p;
                p = Cdr[p];
        }
        p = Car[last];
        while (p != NIL) {
                if (atom_p(p)) return error(err, Car[last]);
                p = Cdr[p];
        }
        if (cddr(m) == NIL) {
                p = cadr(m);
        }
        else {
                p = flat_copy(Cdr[m], &q);
                q = p;
                while (cddr(q) != NIL) q = Cdr[q];
                Cdr[q] = Car[last];
        }
        return alloc(Car[m], p);
}
```

(boolean? expr) --> boolean

```
int pp_boolean_p(int x) {
        return boolean_p(cadr(x))? TRUE: FALSE;
}
```

(car expr) --> form

```
int pp_car(int x) {
        return caadr(x);
}
```

(cdr expr) --> form

```
int pp_cdr(int x) {
        return cdadr(x);
}
```

(char? expr) --> boolean

```
int pp_char_p(int x) {
        return char_p(cadr(x))? TRUE: FALSE;
}
```

(char->integer char) --> integer

```
int pp_char_to_integer(int x) {
        return make_integer(cadadr(x));
}
```

(char-alphabetic? char) --> boolean

```
int pp_char_alphabetic_p(int x) {
        return isalpha(char_value(cadr(x)))? TRUE: FALSE;
}
```

These helpers are used to compare characters in the below `char_predicate()` function.

```
#define L(c) tolower(c)
int char_ci_le(int c1, int c2) { return L(c1) <= L(c2); }
int char_ci_lt(int c1, int c2) { return L(c1) <  L(c2); }
int char_ci_eq(int c1, int c2) { return L(c1) == L(c2); }
int char_ci_ge(int c1, int c2) { return L(c1) >= L(c2); }
int char_ci_gt(int c1, int c2) { return L(c1) >  L(c2); }

int char_le(int c1, int c2) { return c1 <= c2; }
int char_lt(int c1, int c2) { return c1 <  c2; }
int char_eq(int c1, int c2) { return c1 == c2; }
int char_ge(int c1, int c2) { return c1 >= c2; }
int char_gt(int c1, int c2) { return c1 >  c2; }
```

The `char_predicate()` function handles the char comparison predicates. The `name` parameter holds the name of the predicate (for error reporting), `p` is a pointer to one of the above comparison functions, and `x` is the application of the predicate.

Because `char_predicate()` handles variadic procedure calls, it checks its arguments locally.

```
int char_predicate(char *name, int (*p)(int c1, int c2), int x) {
        char    msg[100];
        x = Cdr[x];
        while (Cdr[x] != NIL) {
                if (!char_p(cadr(x))) {
                        sprintf(msg, "%s: expected char, got", name);
                        return error(msg, cadr(x));
                }
                if (!p(char_value(Car[x]), char_value(cadr(x))))
                        return FALSE;
                x = Cdr[x];
        }
        return TRUE;
}
```

(char-ci<=? char1 char2 [char3 ...]) --> boolean
(char-ci<? char1 char2 [char3 ...]) --> boolean
(char-ci=? char1 char2 [char3 ...]) --> boolean
(char-ci>=? char1 char2 [char3 ...]) --> boolean
(char-ci>? char1 char2 [char3 ...]) --> boolean

```
#define R return
int pp_char_ci_le_p(int x) { R char_predicate("char-ci<=?", char_ci_le, x); }
int pp_char_ci_lt_p(int x) { R char_predicate("char-ci<?",  char_ci_lt, x); }
int pp_char_ci_eq_p(int x) { R char_predicate("char-ci=?",  char_ci_eq, x); }
int pp_char_ci_ge_p(int x) { R char_predicate("char-ci>=?", char_ci_ge, x); }
int pp_char_ci_gt_p(int x) { R char_predicate("char-ci>?",  char_ci_gt, x); }
```

(char<=? char1 char2 [char3 ...]) --> boolean
(char<? char1 char2 [char3 ...]) --> boolean
(char=? char1 char2 [char3 ...]) --> boolean
(char>=? char1 char2 [char3 ...]) --> boolean
(char>? char1 char2 [char3 ...]) --> boolean

```
int pp_char_le_p(int x) { R char_predicate("char<=?", char_le, x); }
int pp_char_lt_p(int x) { R char_predicate("char<?",  char_lt, x); }
int pp_char_eq_p(int x) { R char_predicate("char=?",  char_eq, x); }
int pp_char_ge_p(int x) { R char_predicate("char>=?", char_ge, x); }
int pp_char_gt_p(int x) { R char_predicate("char>?",  char_gt, x); }
```

(char-downcase char) --> char

```
int pp_char_downcase(int x) {
        return make_char(tolower(char_value(cadr(x))));
}
```

(char-lower-case? char) --> boolean

```
int pp_char_lower_case_p(int x) {
        return islower(char_value(cadr(x)))? TRUE: FALSE;
}
```

(char-numeric? char) --> boolean

```
int pp_char_numeric_p(int x) {
        return isdigit(char_value(cadr(x)))? TRUE: FALSE;
}
```

(char-upcase char) --> char

```
int pp_char_upcase(int x) {
        return make_char(toupper(char_value(cadr(x))));
}
```

(char-upper-case? char) --> boolean

```
int pp_char_upper_case_p(int x) {
        return isupper(char_value(cadr(x)))? TRUE: FALSE;
}
```

(char-whitespace? char) --> boolean

```
int pp_char_whitespace_p(int x) {
        int     c = char_value(cadr(x));
        return (c == ' '  || c == '\t' || c == '\n' ||
                c == '\r' || c == '\f')? TRUE: FALSE;
}
```

Close the given port and release the associated port structure.

```
void close_port(int port) {
        if (port < 0 || port >= MAX_PORTS) return;
        if (Ports[port] == NULL) return;
        if (fclose(Ports[port]))
                fatal("close_port(): fclose() failed");
        Ports[port] = NULL;
        Port_flags[port] = 0;
}
```

(close-input-port input-port) --> unspecific

```
int pp_close_input_port(int x) {
        if (port_no(cadr(x)) < 2)
                return error("please do not close the standard input port",
                                NOEXPR);
        close_port(port_no(cadr(x)));
        return UNSPECIFIC;
}
```

(close-output-port output-port) --> unspecific

```
int pp_close_output_port(int x) {
        if (port_no(cadr(x)) < 2)
                return error("please do not close the standard output port",
                                NOEXPR);
        close_port(port_no(cadr(x)));
        return UNSPECIFIC;
}
```

(cons expr1 expr2) --> pair

```
int pp_cons(int x) {
        return alloc(cadr(x), caddr(x));
}
```

Create a port object. The `port_no` parameter is an offset into the port array (`Ports` and `Port_flags`) and `type` is either `S_input_port` or `S_output_port`.

```
int make_port(int port_no, int type) {
        int     n;

        n = alloc3(port_no, NIL, AFLAG);
        return alloc3(type, n, AFLAG);
}
```

(current-input-port) --> input-port

```
int pp_current_input_port(int x) {
        return make_port(Input_port, S_input_port);
}
```

(current-output-port) --> output-port

```
int pp_current_output_port(int x) {
        return make_port(Output_port, S_output_port);
}

int pp_write(int x);
```

(display expr [output-port]) --> unspecific
This primitive delegates its arguments to `pp_write()`. The `Displaying` flag controls whether output will be pretty-printed or converted to external representation.

```
int pp_display(int x) {
        Displaying = 1;
        pp_write(x);
        Displaying = 0;
        return UNSPECIFIC;
}
```

(eof-object? expr) --> boolean

```
int pp_eof_object_p(int x) {
        return cadr(x) == ENDOFFILE? TRUE: FALSE;
}
```

(eq? expr1 expr2) --> boolean

```
int pp_eq_p(int x) {
        return cadr(x) == caddr(x)? TRUE: FALSE;
}
```

(= integer1 integer3 [integer3 ...]) --> boolean

```
int pp_equal(int x) {
        x = Cdr[x];
        while (Cdr[x] != NIL) {
                if (!integer_p(cadr(x)))
                        return error("=: expected integer, got", cadr(x));
                if (!bignum_equal_p(Car[x], cadr(x))) return FALSE;
                x = Cdr[x];
        }
        return TRUE;
}
```

(> integer1 integer3 [integer3 ...]) --> boolean

```
int pp_greater(int x) {
        x = Cdr[x];
        while (Cdr[x] != NIL) {
                if (!integer_p(cadr(x)))
                        return error(">: expected integer, got", cadr(x));
                if (!bignum_less_p(cadr(x), Car[x])) return FALSE;
                x = Cdr[x];
        }
        return TRUE;
}
```

(>= integer1 integer3 [integer3 ...]) --> boolean

```
int pp_greater_equal(int x) {
        x = Cdr[x];
        while (Cdr[x] != NIL) {
                if (!integer_p(cadr(x)))
                        return error(">=: expected integer, got", cadr(x));
                if (bignum_less_p(Car[x], cadr(x))) return FALSE;
                x = Cdr[x];
        }
        return TRUE;
}
```

(input-port? expr) --> boolean

```
int pp_input_port_p(int x) {
        return input_port_p(cadr(x))? TRUE: FALSE;
}
```

(integer->char integer) --> char

```
int pp_integer_to_char(int x) {
        int     n;

        n = integer_value("integer->char", cadr(x));
        if (n < 0 || n > 127)
                return error("integer->char: argument value out of range",
                                cadr(x));
        return make_char(n);
}
```

(integer? expr) --> boolean

```
int pp_integer_p(int x) {
        return integer_p(cadr(x))? TRUE: FALSE;
}
```

(< integer1 integer3 [integer3 ...]) --> boolean

```
int pp_less(int x) {
        x = Cdr[x];
        while (Cdr[x] != NIL) {
                if (!integer_p(cadr(x)))
                        return error("<: expected integer, got", cadr(x));
                if (!bignum_less_p(Car[x], cadr(x))) return FALSE;
                x = Cdr[x];
        }
        return TRUE;
}
```

(<= integer1 integer3 [integer3 ...]) --> boolean

```
int pp_less_equal(int x) {
        x = Cdr[x];
        while (Cdr[x] != NIL) {
                if (!integer_p(cadr(x)))
                        return error("<=: expected integer, got", cadr(x));
                if (bignum_less_p(cadr(x), Car[x])) return FALSE;
                x = Cdr[x];
        }
        return TRUE;
}
```

(list->string list) --> string
Because **list->string** accepts only proper lists of chars, it does some additional type checking locally.

```
int pp_list_to_string(int x) {
        int     n, p, k = length(cadr(x));
        char    *s;

        n = make_string("", k);
        s = string(n);
        p = cadr(x);
        while (p != NIL) {
                if (atom_p(p))
                        return error("list->string: improper list", p);
                if (!char_p(Car[p]))
```

```
                        return error("list->string: expected list of char, "
                                "got list containing",
                                Car[p]);
                *s++ = cadar(p);
                p = Cdr[p];
        }
        *s = 0;
        return n;
}
```

(list->vector list) --> vector

```
int pp_list_to_vector(int x) {
        return list_to_vector(cadr(x), "improper list in list->vector");
}
```

The `open_port()` function attempts to allocate a port structure. When it succeeds, it attempts to open the file with the given path in the given mode. The `mode` is one of the `fopen()` modes `"r"` and `"w"`. When `open_port()` manages to allocate a port and associate it with the given file, it returns the index of the allocated port. In case of an error it returns -1.

`open_port()` attempts to free a port structure by running a garbage collection when no port can be allocated intially.

```
int open_port(char *path, char *mode) {
        int     i, tries;

        for (tries=0; tries<2; tries++) {
                for (i=0; i<MAX_PORTS; i++) {
                        if (Ports[i] == NULL) {
                                Ports[i] = fopen(path, mode);
                                if (Ports[i] == NULL)
                                        return -1;
                                else
                                        return i;
                        }
                }
                if (tries == 0) gc();
        }
        return -1;
}

int eval(int x);  /* Evaluator */
```

The `load()` function opens a new input port, locks it, and evaluates expressions read from that port until input is exhausted or an error occurred. When it finishes reading, it restores the input port that was in effect before.

The `file` argument specifies the file to be associated with the new input port. `load()` returns -1, if the file could not be opened and zero upon success. Note that the return code only indicates whether the specified file could be opened and does not say anything about the correctness of expressions in that file.

```
int load(char *file) {
        int     n;
        int     new_port, old_port;

        new_port = open_port(file, "r");
        if (new_port == -1) return -1;
        Port_flags[new_port] |= LFLAG;
        old_port = Input_port;
        Input_port = new_port;
        while (!Error_flag) {
                n = xread();
                if (n == ENDOFFILE) break;
                if (!Error_flag) n = eval(n);
        }
        close_port(new_port);
        Input_port = old_port;
        return 0;
}
```

(load string) --> unspecific

```
int pp_load(int x) {
        if (load(string(cadr(x))) < 0)
                return error("load: cannot open file", cadr(x));
        return UNSPECIFIC;
}
```

(make-string integer [char]) --> string

```
int pp_make_string(int x) {
        int     n, c, k;
        char    *s;

        k = integer_value("make-string", cadr(x));
        n = make_string("", k);
        s = string(n);
        c = cddr(x) == NIL? ' ': char_value(caddr(x));
        memset(s, c, k);
        s[k] = 0;
        return n;
}
```

(make-vector integer [expr]) --> vector

```
int pp_make_vector(int x) {
        int     n, i, m, k;
        int     *v;

        k = integer_value("make-vector", cadr(x));
        n = allocv(S_vector, k * sizeof(int));
        v = vector(n);
        m = cddr(x) == NIL? FALSE: caddr(x);
        for (i=0; i<k; i++) v[i] = m;
        return n;
}
```

(- integer1 [integer2 ...]) --> integer

```
int pp_minus(int x) {
        int     a;

        x = Cdr[x];
        if (Cdr[x] == NIL) return bignum_negate(Car[x]);
        a = Car[x];
        x = Cdr[x];
        save(a);
        while (x != NIL) {
                if (!integer_p(Car[x]))
                        return error("-: expected integer, got", Car[x]);
                a = bignum_subtract(a, Car[x]);
                Car[Stack] = a;
                x = Cdr[x];
        }
        unsave(1);
        return a;
}
```

(open-input-file string) --> input-port

```
int pp_open_input_file(int x) {
        int     n, p;

        p = open_port(string(cadr(x)), "r");
        if (p < 0) return error("could not open input file", cadr(x));
        Port_flags[p] |= LFLAG;
        n = make_port(p, S_input_port);
        Port_flags[p] &= ~LFLAG;
        return n;
}
```

(open-output-file string) --> output-port

```
int pp_open_output_file(int x) {
        int     n, p;

        p = open_port(string(cadr(x)), "w");
        if (p < 0) return error("could not open output file", cadr(x));
        Port_flags[p] |= LFLAG;
        n = make_port(p, S_output_port);
        Port_flags[p] &= ~LFLAG;
        return n;
}
```

(output-port? expr) --> boolean

```
int pp_output_port_p(int x) {
        return output_port_p(cadr(x))? TRUE: FALSE;
}
```

(pair? expr) --> boolean

```
int pp_pair_p(int x) {
        return atom_p(cadr(x))? FALSE: TRUE;
}
```

(+ [integer ...]) --> integer

```
int pp_plus(int x) {
        int     a;

        x = Cdr[x];
        if (Cdr[x] == NIL) return Car[x];
        a = make_integer(0);
        save(a);
        while (x != NIL) {
                if (!integer_p(Car[x]))
                        return error("+: expected integer, got", Car[x]);
                a = bignum_add(a, Car[x]);
                Car[Stack] = a;
                x = Cdr[x];
        }
        unsave(1);
        return a;
}
```

(procedure? expr) --> boolean

```
int pp_procedure_p(int x) {
        return (procedure_p(cadr(x)) || primitive_p(cadr(x)))?
                TRUE: FALSE;
}
```

(quotient integer1 integer2) --> integer

```
int pp_quotient(int x) {
        return Car[bignum_divide(x, cadr(x), caddr(x))];
}
```

(read [input-port]) --> form

```
int pp_read(int x) {
        int     n, new_port, old_port;

        new_port = Cdr[x] == NIL? Input_port: port_no(cadr(x));
        if (new_port < 0 || new_port >= MAX_PORTS)
                return error("bad input port", cadr(x));
        old_port = Input_port;
        Input_port = new_port;
        n = xread();
        Input_port = old_port;
        return n;
}
```

The `read_char()` function implements both the **`read-char`** and **`peek-char`** procedures. It reads a character from the given input port or from the current input port if no port is specified. When `unget` is non-zero, it puts the character back into the input stream after reading it. `read_char()` returns the character read.

```
int read_char(int x, int unget) {
        int     c, new_port, old_port;

        new_port = Cdr[x] == NIL? Input_port: port_no(cadr(x));
        if (new_port < 0 || new_port >= MAX_PORTS)
                return error("bad input port", cadr(x));
        if (Ports[new_port] == NULL)
                return error("input port is not open", NOEXPR);
        old_port = Input_port;
        Input_port = new_port;
        c = read_c();
        if (unget) reject(c);
        Input_port = old_port;
        return c == EOF? ENDOFFILE: make_char(c);
}
```

`(peek-char [input-port]) --> char`

```
int pp_peek_char(int x) {
        return read_char(x, 1);
}
```

`(read-char [input-port]) --> char`

```
int pp_read_char(int x) {
        return read_char(x, 0);
}
```

`(remainder integer1 integer2) --> integer`

```
int pp_remainder(int x) {
        return Cdr[bignum_divide(x, cadr(x), caddr(x))];
}
```

`(set-car! pair expr) --> unspecific`

```
int pp_set_car_b(int x) {
        caadr(x) = caddr(x);
        return UNSPECIFIC;
}
```

`(set-cdr! pair expr) --> unspecific`

```
int pp_set_cdr_b(int x) {
        cdadr(x) = caddr(x);
        return UNSPECIFIC;
}
```

(set-input-port! port) --> unspecific

This procedure is not part of R5RS. It sets the current input port to the given port.

```
int pp_set_input_port_b(int x) {
        Input_port = port_no(cadr(x));
        return UNSPECIFIC;
}
```

(set-output-port! port) --> unspecific

This procedure is not part of R5RS. It sets the current output port to the given port.

```
int pp_set_output_port_b(int x) {
        Output_port = port_no(cadr(x));
        return UNSPECIFIC;
}
```

(string->list string) --> list

```
int pp_string_to_list(int x) {
        char    *s;
        int     n, a, k, i;

        s = string(cadr(x));
        k = string_len(cadr(x));
        n = NIL;
        a = NIL;
        for (i=0; i<k-1; i++) {
                if (n == NIL) {
                        n = a = alloc(make_char(s[i]), NIL);
                        save(n);
                }
                else {
                        Cdr[a] = alloc(make_char(s[i]), NIL);
                        a = Cdr[a];
                }
        }
        if (n != NIL) unsave(1);
        return n;
}
```

(string->symbol string) --> symbol

```
int pp_string_to_symbol(int x) {
        return add_symbol(string(cadr(x)));
}
```

(string-append [string1 ...]) --> string

Because `string-append` is a variadic procedure, it checks the types of its arguments locally.

```
int pp_string_append(int x) {
        int     p, k, n;
        char    *s;
        k = 0;
        for (p = Cdr[x]; p != NIL; p = Cdr[p]) {
                if (!string_p(Car[p]))
                        return error("string-append: expected string, got",
                                        Car[p]);
                k += string_len(Car[p])-1;
        }
        n = make_string("", k);
        s = string(n);
        k = 0;
        for (p = Cdr[x]; p != NIL; p = Cdr[p]) {
                strcpy(&s[k], string(Car[p]));
                k += string_len(Car[p])-1;
        }
        return n;
}
```

(string-copy string) --> string

```
int pp_string_copy(int x) {
        return make_string(string(cadr(x)), string_len(cadr(x))-1);
}
```

(string-fill! string char) --> unspecific

```
int pp_string_fill_b(int x) {
        int     c = char_value(caddr(x)),
                i, k = string_len(cadr(x))-1;
        char    *s = string(cadr(x));

        for (i=0; i<k; i++) s[i] = c;
        return UNSPECIFIC;
}
```

(substring string integer1 integer2) --> string

```
int pp_substring(int x) {
        int     k = string_len(cadr(x))-1;
        int     p0 = integer_value("substring", caddr(x));
        int     pn = integer_value("substring", cadddr(x));
        char    *src = string(cadr(x));
        char    *dst;
        int     n;

        if (p0 < 0 || p0 > k || pn < 0 || pn > k || pn < p0) {
                n = alloc(cadddr(x), NIL);
                return error("substring: bad range",
                                alloc(caddr(x), n));
        }
        n = make_string("", pn-p0);
        dst = string(n);
        if (pn-p0 != 0) memcpy(dst, &src[p0], pn-p0);
        dst[pn-p0] = 0;
        return n;
}
```

(string-length string) --> integer

```
int pp_string_length(int x) {
        return make_integer(string_len(cadr(x))-1);
}
```

(string-ref string integer) --> char

```
int pp_string_ref(int x) {
        int     p, k = string_len(cadr(x))-1;

        p = integer_value("string-ref", caddr(x));
        if (p < 0 || p >= k)
                return error("string-ref: index out of range",
                                caddr(x));
        return make_char(string(cadr(x))[p]);
}
```

(string-set! string integer char) --> unspecific

```
int pp_string_set_b(int x) {
        int     p, k = string_len(cadr(x))-1;

        p = integer_value("string-set!", caddr(x));
        if (p < 0 || p >= k)
                return error("string-set!: index out of range",
                                caddr(x));
        string(cadr(x))[p] = char_value(cadddr(x));
        return UNSPECIFIC;
}
```

A portable, case-insensitive version of `strcmp()`.

```
int strcmp_ci(char *s1, char *s2) {
        int     c1, c2;

        for (;;) {
                c1 = tolower(*s1++);
                c2 = tolower(*s2++);
                if (!c1 || !c2 || c1 != c2) break;
        }
        return c1<c2? -1: c1>c2? 1: 0;
}
```

These helpers are used to compare characters in the below `string_predicate()` function.

```
int string_ci_le(char *s1, char *s2) { return strcmp_ci(s1, s2) <= 0; }
int string_ci_lt(char *s1, char *s2) { return strcmp_ci(s1, s2) <  0; }
int string_ci_eq(char *s1, char *s2) { return strcmp_ci(s1, s2) == 0; }
int string_ci_ge(char *s1, char *s2) { return strcmp_ci(s1, s2) >= 0; }
int string_ci_gt(char *s1, char *s2) { return strcmp_ci(s1, s2) >  0; }
int string_le(char *s1, char *s2) { return strcmp(s1, s2) <= 0; }
int string_lt(char *s1, char *s2) { return strcmp(s1, s2) <  0; }
int string_eq(char *s1, char *s2) { return strcmp(s1, s2) == 0; }
int string_ge(char *s1, char *s2) { return strcmp(s1, s2) >= 0; }
int string_gt(char *s1, char *s2) { return strcmp(s1, s2) >  0; }
```

The `string_predicate()` function handles the string comparison predicates. The `name` parameter holds the name of the predicate (for error reporting), `p` is a pointer to one of the above comparison functions, and `x` is the application of the predicate.

Because `string_predicate()` handles variadic procedure calls, it checks its arguments locally.

```
int string_predicate(char *name, int (*p)(char *s1, char *s2), int x) {
        char    msg[100];

        x = Cdr[x];
        while (Cdr[x] != NIL) {
                if (!string_p(cadr(x))) {
                        sprintf(msg, "%s: expected string, got", name);
                        return error(msg, cadr(x));
                }
                if (!p(string(Car[x]), string(cadr(x))))
                        return FALSE;
                x = Cdr[x];
        }
        return TRUE;
}
```

(string-ci<=? string1 string2 [string3 ...]) --> boolean
(string-ci<? string1 string2 [string3 ...]) --> boolean
(string-ci=? string1 string2 [string3 ...]) --> boolean
(string-ci>=? string1 string2 [string3 ...]) --> boolean
(string-ci>? string1 string2 [string3 ...]) --> boolean

```
#define SP return string_predicate
int pp_string_ci_le_p(int x) { SP("string-ci<=?", string_ci_le, x); }
int pp_string_ci_lt_p(int x) { SP("string-ci<?",  string_ci_lt, x); }
int pp_string_ci_eq_p(int x) { SP("string-ci=?",  string_ci_eq, x); }
int pp_string_ci_ge_p(int x) { SP("string-ci>=?", string_ci_ge, x); }
int pp_string_ci_gt_p(int x) { SP("string-ci>?",  string_ci_gt, x); }
```

(string<=? string1 string2 [string3 ...]) --> boolean
(string<? string1 string2 [string3 ...]) --> boolean
(string=? string1 string2 [string3 ...]) --> boolean
(string>=? string1 string2 [string3 ...]) --> boolean
(string>? string1 string2 [string3 ...]) --> boolean

```
int pp_string_le_p(int x) { SP("string<=?", string_le, x); }
int pp_string_lt_p(int x) { SP("string<?",  string_lt, x); }
int pp_string_eq_p(int x) { SP("string=?",  string_eq, x); }
int pp_string_ge_p(int x) { SP("string>=?", string_ge, x); }
int pp_string_gt_p(int x) { SP("string>?",  string_gt, x); }
```

`(string? expr) --> boolean`

```
int pp_string_p(int x) {
        return string_p(cadr(x))? TRUE: FALSE;
}
```

`(symbol->string symbol) --> string`

```
int pp_symbol_to_string(int x) {
        char    *s = string(cadr(x));

        return make_string(s, strlen(s));
}
```

`(symbol? expr) --> boolean`

```
int pp_symbol_p(int x) {
        return symbol_p(cadr(x))? TRUE: FALSE;
}
```

`(syntax->list symbol) --> list`

The **`syntax->list`** procedure is not part of R5RS. If the given symbol names a syntax transformer, it returns the syntax rules of that transformer. If the symbol does not refer to a syntax transformer, it returns **`#f`**. The syntax rules of a transformer are equal to the argument list of the corresponding **`syntax-rules`** form.

```
int pp_syntax_to_list(int x) {
        int     n;

        n = cadr(x);
        if (symbol_p(n)) n = lookup(n, Environment);
        if (n == NIL) return FALSE;
        n = cadr(n);
        if (!syntax_p(n)) return FALSE;
        return Cdr[n];
}
```

`(* [integer ...]) --> integer`

```
int pp_times(int x) {
        int     a;

        x = Cdr[x];
        if (Cdr[x] == NIL) return Car[x];
        a = make_integer(1);
        save(a);
        while (x != NIL) {
                if (!integer_p(Car[x]))
                        return error("+: expected integer, got", Car[x]);
                a = bignum_multiply(a, Car[x]);
                Car[Stack] = a;
                x = Cdr[x];
        }
        unsave(1);
        return a;
}
```

If an application of **unquote** and **unquote-splicing** should make it to this point, they are used outside of a **quasiquote** context. This is an error.

```
int pp_unquote(int x) {
        return error("unquote: not in quasiquote context", NOEXPR);
}

int pp_unquote_splicing(int x) {
        return error("unquote-splicing: not in quasiquote context", NOEXPR);
}
```

(vector->list vector) --> list

```
int pp_vector_to_list(int x) {
        int     *v, n, a, k, i;

        v = vector(cadr(x));
        k = vector_len(cadr(x));
        n = NIL;
        a = NIL;
        for (i=0; i<k; i++) {
                if (n == NIL) {
                        n = a = alloc(v[i], NIL);
                        save(n);
                }
                else {
                        Cdr[a] = alloc(v[i], NIL);
                        a = Cdr[a];
                }
        }
        if (n != NIL) unsave(1);
        return n;
}
```

(vector-fill! vector expr) --> unspecific

```
int pp_vector_fill_b(int x) {
        int     fill = caddr(x),
                i, k = vector_len(cadr(x)),
                *v = vector(cadr(x));

        for (i=0; i<k; i++) v[i] = fill;
        return UNSPECIFIC;
}
```

(vector-length vector) --> integer

```
int pp_vector_length(int x) {
        return make_integer(vector_len(cadr(x)));
}
```

(vector-ref vector integer) --> form

```
int pp_vector_ref(int x) {
        int     p, k = vector_len(cadr(x));

        p = integer_value("vector-ref", caddr(x));
        if (p < 0 || p >= k)
                return error("vector-ref: index out of range",
                                caddr(x));
        return vector(cadr(x))[p];
}
```

(vector-set! vector integer expr) --> unspecific

```
int pp_vector_set_b(int x) {
        int     p, k = vector_len(cadr(x));

        p = integer_value("vector-set!", caddr(x));
        if (p < 0 || p >= k)
                return error("vector-set!: index out of range",
                                caddr(x));
        vector(cadr(x))[p] = cadddr(x);
        return UNSPECIFIC;
}
```

(vector? expr) --> boolean

```
int pp_vector_p(int x) {
        return vector_p(cadr(x))? TRUE: FALSE;
}
```

(write expr [output-port]) --> unspecific

```
int pp_write(int x) {
        int     new_port, old_port;

        new_port = cddr(x) == NIL? Output_port: port_no(caddr(x));
        if (new_port < 0 || new_port >= MAX_PORTS)
                return error("bad output port", caddr(x));
        old_port = Output_port;
        Output_port = new_port;
        print(cadr(x));
        Output_port = old_port;
        return UNSPECIFIC;
}
```

(write-char char [output-port]) --> unspecific

```
int pp_write_char(int x) {
        return pp_display(x);
}
```

(wrong string [expr]) --> bottom
This procedure is not part of R5RS. It writes the message given in the string argument to the standard output port and aborts program execution. If an additional argument is specified, its external representation is appended to the message. **Wrong** sets `Error_flag` and hence does not have a return value.

```
int pp_wrong(int x) {
        return error(string(cadr(x)), length(x) > 2? caddr(x): NOEXPR);
}
```

Evaluator: Evaluator

This section describes the very core of the S9fES interpreter: a finite state machine that evaluates Scheme expressions.

The following `enum` lists the Scheme types for the type checker.

```
enum TYPES {
        T_NONE,
        T_BOOLEAN,
        T_CHAR,
        T_INPUT_PORT,
        T_INTEGER,
        T_OUTPUT_PORT,
        T_PAIR,
        T_PAIR_OR_NIL,
        T_PROCEDURE,
        T_STRING,
        T_SYMBOL,
        T_VECTOR
};
```

This structure is used to type check and apply a primitive procedure. See the `primitive()` function shown later in this section for details.

```
struct Primitive_procedure {
        int     (*handler)(int expr);
        int     min_args;
        int     max_args;       /* -1 = variadic */
        int     arg_types[3];
};
```

These are symblic names for the all primitives known to the interpreter.

```
enum PRIMITIVES {
        PP_APPLY, PP_BOOLEAN_P, PP_CAR, PP_CDR, PP_CHAR_P,
        PP_CHAR_TO_INTEGER, PP_CHAR_ALPHABETIC_P, PP_CHAR_CI_LE_P,
        PP_CHAR_CI_LT_P, PP_CHAR_CI_EQ_P, PP_CHAR_CI_GE_P,
        PP_CHAR_CI_GT_P, PP_CHAR_DOWNCASE, PP_CHAR_LOWER_CASE_P,
        PP_CHAR_NUMERIC_P, PP_CHAR_UPCASE, PP_CHAR_UPPER_CASE_P,
        PP_CHAR_WHITESPACE_P, PP_CHAR_LE_P, PP_CHAR_LT_P, PP_CHAR_EQ_P,
        PP_CHAR_GE_P, PP_CHAR_GT_P, PP_CLOSE_INPUT_PORT,
        PP_CLOSE_OUTPUT_PORT, PP_CONS, PP_CURRENT_INPUT_PORT,
        PP_CURRENT_OUTPUT_PORT, PP_DISPLAY, PP_EOF_OBJECT_P, PP_EQ_P,
        PP_EQUAL, PP_GREATER, PP_GREATER_EQUAL, PP_INPUT_PORT_P,
        PP_INTEGER_P, PP_INTEGER_TO_CHAR, PP_LESS, PP_LESS_EQUAL,
        PP_LIST_TO_STRING, PP_LIST_TO_VECTOR, PP_LOAD, PP_MAKE_STRING,
        PP_MAKE_VECTOR, PP_MINUS, PP_OPEN_INPUT_FILE, PP_OPEN_OUTPUT_FILE,
        PP_OUTPUT_PORT_P, PP_PAIR_P, PP_PEEK_CHAR, PP_PLUS, PP_PROCEDURE_P,
        PP_QUOTIENT, PP_READ, PP_READ_CHAR, PP_REMAINDER, PP_SET_CAR_B,
        PP_SET_CDR_B, PP_SET_INPUT_PORT_B, PP_SET_OUTPUT_PORT_B,
        PP_STRING_TO_LIST, PP_STRING_TO_SYMBOL, PP_STRING_APPEND,
        PP_STRING_COPY, PP_STRING_FILL_B, PP_STRING_LENGTH, PP_STRING_REF,
        PP_STRING_SET_B, PP_STRING_CI_LE_P, PP_STRING_CI_LT_P,
        PP_STRING_CI_EQ_P, PP_STRING_CI_GE_P, PP_STRING_CI_GT_P,
        PP_STRING_LE_P, PP_STRING_LT_P, PP_STRING_EQ_P, PP_STRING_GE_P,
        PP_STRING_GT_P, PP_STRING_P, PP_SUBSTRING, PP_SYMBOL_P,
        PP_SYMBOL_TO_STRING, PP_SYNTAX_TO_LIST, PP_TIMES, PP_UNQUOTE,
        PP_UNQUOTE_SPLICING, PP_VECTOR_FILL_B, PP_VECTOR_LENGTH,
        PP_VECTOR_SET_B, PP_VECTOR_REF, PP_VECTOR_TO_LIST, PP_VECTOR_P,
        PP_WRITE, PP_WRITE_CHAR, PP_WRONG
};
```

The following array lists the handler and type information for each primitive procedure of S9fES.

The above enum must reflect the order of the array below! The above symbolic names are used to address individual members of the below array. Changing the order of one of them breaks this association, so special care must be taken when inserting new primitives.

```
struct Primitive_procedure Primitives[] = {
 { pp_apply,              2, -1, { T_PROCEDURE,  T_NONE,       T_NONE } },
 { pp_boolean_p,          1,  1, { T_NONE,       T_NONE,       T_NONE } },
 { pp_car,                1,  1, { T_PAIR,       T_NONE,       T_NONE } },
 { pp_cdr,                1,  1, { T_PAIR,       T_NONE,       T_NONE } },
 { pp_char_p,             1,  1, { T_NONE,       T_NONE,       T_NONE } },
 { pp_char_to_integer,    1,  1, { T_CHAR,       T_NONE,       T_NONE } },
 { pp_char_alphabetic_p,  1,  1, { T_CHAR,       T_NONE,       T_NONE } },
 { pp_char_ci_le_p,       2, -1, { T_CHAR,       T_CHAR,       T_NONE } },
 { pp_char_ci_lt_p,       2, -1, { T_CHAR,       T_CHAR,       T_NONE } },
 { pp_char_ci_eq_p,       2, -1, { T_CHAR,       T_CHAR,       T_NONE } },
 { pp_char_ci_ge_p,       2, -1, { T_CHAR,       T_CHAR,       T_NONE } },
 { pp_char_ci_gt_p,       2, -1, { T_CHAR,       T_CHAR,       T_NONE } },
 { pp_char_downcase,      1,  1, { T_CHAR,       T_NONE,       T_NONE } },
 { pp_char_lower_case_p,  1,  1, { T_CHAR,       T_NONE,       T_NONE } },
 { pp_char_numeric_p,     1,  1, { T_CHAR,       T_NONE,       T_NONE } },
 { pp_char_upcase,        1,  1, { T_CHAR,       T_NONE,       T_NONE } },
 { pp_char_upper_case_p,  1,  1, { T_CHAR,       T_NONE,       T_NONE } },
 { pp_char_whitespace_p,  1,  1, { T_CHAR,       T_NONE,       T_NONE } },
 { pp_char_le_p,            2, -1, { T_CHAR,        T_CHAR,        T_NONE } },
 { pp_char_lt_p,            2, -1, { T_CHAR,        T_CHAR,        T_NONE } },
 { pp_char_eq_p,            2, -1, { T_CHAR,        T_CHAR,        T_NONE } },
 { pp_char_ge_p,            2, -1, { T_CHAR,        T_CHAR,        T_NONE } },
 { pp_char_gt_p,            2, -1, { T_CHAR,        T_CHAR,        T_NONE } },
 { pp_close_input_port,     1,  1, { T_INPUT_PORT,  T_NONE,        T_NONE } },
 { pp_close_output_port,    1,  1, { T_OUTPUT_PORT,T_NONE,         T_NONE } },
 { pp_cons,                 2,  2, { T_NONE,        T_NONE,        T_NONE } },
 { pp_current_input_port,   0,  0, { T_NONE,        T_NONE,        T_NONE } },
 { pp_current_output_port,  0,  0, { T_NONE,        T_NONE,        T_NONE } },
 { pp_display,              1,  2, { T_NONE,        T_OUTPUT_PORT,T_NONE } },
 { pp_eof_object_p,         1,  1, { T_NONE,        T_NONE,        T_NONE } },
 { pp_eq_p,                 2,  2, { T_NONE,        T_NONE,        T_NONE } },
 { pp_equal,                2, -1, { T_INTEGER,     T_INTEGER,     T_NONE } },
 { pp_greater,              2, -1, { T_INTEGER,     T_INTEGER,     T_NONE } },
 { pp_greater_equal,        2, -1, { T_INTEGER,     T_INTEGER,     T_NONE } },
 { pp_input_port_p,         1,  1, { T_NONE,        T_NONE,        T_NONE } },
 { pp_integer_p,            1,  1, { T_NONE,        T_NONE,        T_NONE } },
 { pp_integer_to_char,      1,  1, { T_INTEGER,     T_NONE,        T_NONE } },
 { pp_less,                 2, -1, { T_INTEGER,     T_INTEGER,     T_NONE } },
 { pp_less_equal,           2, -1, { T_INTEGER,     T_INTEGER,     T_NONE } },
 { pp_list_to_string,       1,  1, { T_PAIR_OR_NIL,T_NONE,         T_NONE } },
 { pp_list_to_vector,       1,  1, { T_PAIR_OR_NIL,T_NONE,         T_NONE } },
 { pp_load,                 1,  1, { T_STRING,      T_NONE,        T_NONE } },
 { pp_make_string,          1,  2, { T_INTEGER,     T_NONE,        T_NONE } },
 { pp_make_vector,          1,  2, { T_INTEGER,     T_NONE,        T_NONE } },
 { pp_minus,                1, -1, { T_INTEGER,     T_NONE,        T_NONE } },
 { pp_open_input_file,      1,  1, { T_STRING,      T_NONE,        T_NONE } },
 { pp_open_output_file,     1,  1, { T_STRING,      T_NONE,        T_NONE } },
 { pp_output_port_p,        1,  1, { T_NONE,        T_NONE,        T_NONE } },
 { pp_pair_p,               1,  1, { T_NONE,        T_NONE,        T_NONE } },
 { pp_peek_char,            0,  1, { T_INPUT_PORT,  T_NONE,        T_NONE } },
 { pp_plus,                 0, -1, { T_NONE,        T_NONE,        T_NONE } },
 { pp_procedure_p,          1,  1, { T_NONE,        T_NONE,        T_NONE } },
 { pp_quotient,             2,  2, { T_INTEGER,     T_INTEGER,     T_NONE } },
 { pp_read,                 0,  1, { T_INPUT_PORT,  T_NONE,        T_NONE } },
 { pp_read_char,            0,  1, { T_INPUT_PORT,  T_NONE,        T_NONE } },
 { pp_remainder,            2,  2, { T_INTEGER,     T_INTEGER,     T_NONE } },
 { pp_set_car_b,            2,  2, { T_PAIR,        T_NONE,        T_NONE } },
 { pp_set_cdr_b,            2,  2, { T_PAIR,        T_NONE,        T_NONE } },
 { pp_set_input_port_b,     1,  1, { T_INPUT_PORT,  T_NONE,        T_NONE } },
 { pp_set_output_port_b,    1,  1, { T_OUTPUT_PORT,T_NONE,         T_NONE } },
 { pp_string_to_list,       1,  1, { T_STRING,      T_NONE,        T_NONE } },
 { pp_string_to_symbol,     1,  1, { T_STRING,      T_NONE,        T_NONE } },
 { pp_string_append,        0, -1, { T_STRING,      T_NONE,        T_NONE } },
 { pp_string_copy,          1,  1, { T_STRING,      T_NONE,        T_NONE } },
 { pp_string_fill_b,        2,  2, { T_STRING,      T_CHAR,        T_NONE } },
 { pp_string_length,        1,  1, { T_STRING,      T_NONE,        T_NONE } },
 { pp_string_ref,           2,  2, { T_STRING,      T_INTEGER,     T_NONE } },
 { pp_string_set_b,         3,  3, { T_STRING,      T_INTEGER,     T_CHAR } },
 { pp_string_ci_le_p,       2, -1, { T_STRING,      T_STRING,      T_NONE } },
 { pp_string_ci_lt_p,       2, -1, { T_STRING,      T_STRING,      T_NONE } },
 { pp_string_ci_eq_p,       2, -1, { T_STRING,     T_STRING,     T_NONE } },
 { pp_string_ci_ge_p,       2, -1, { T_STRING,     T_STRING,     T_NONE } },
 { pp_string_ci_gt_p,       2, -1, { T_STRING,     T_STRING,     T_NONE } },
 { pp_string_le_p,          2, -1, { T_STRING,     T_STRING,     T_NONE } },
 { pp_string_lt_p,          2, -1, { T_STRING,     T_STRING,     T_NONE } },
 { pp_string_eq_p,          2, -1, { T_STRING,     T_STRING,     T_NONE } },
 { pp_string_ge_p,          2, -1, { T_STRING,     T_STRING,     T_NONE } },
 { pp_string_gt_p,          2, -1, { T_STRING,     T_STRING,     T_NONE } },
 { pp_string_p,             1,  1, { T_NONE,       T_NONE,       T_NONE } },
 { pp_substring,            3,  3, { T_STRING,     T_INTEGER,    T_INTEGER } },
 { pp_symbol_p,             1,  1, { T_NONE,       T_NONE,       T_NONE } },
 { pp_symbol_to_string,     1,  1, { T_SYMBOL,     T_NONE,       T_NONE } },
 { pp_syntax_to_list,       1,  1, { T_NONE,       T_NONE,       T_NONE } },
 { pp_times,                0, -1, { T_INTEGER,    T_NONE,       T_NONE } },
 { pp_unquote,              1,  1, { T_NONE,       T_NONE,       T_NONE } },
 { pp_unquote_splicing,     1,  1, { T_NONE,       T_NONE,       T_NONE } },
 { pp_vector_fill_b,        2,  2, { T_VECTOR,     T_NONE,       T_NONE } },
 { pp_vector_length,        1,  1, { T_VECTOR,     T_NONE,       T_NONE } },
 { pp_vector_set_b,         3,  3, { T_VECTOR,     T_INTEGER,    T_NONE } },
 { pp_vector_ref,           2,  2, { T_VECTOR,     T_INTEGER,    T_NONE } },
 { pp_vector_to_list,       1,  1, { T_VECTOR,     T_NONE,       T_NONE } },
 { pp_vector_p,             1,  1, { T_NONE,       T_NONE,       T_NONE } },
 { pp_write,                1,  2, { T_NONE,       T_OUTPUT_PORT,T_NONE } },
 { pp_write_char,           1,  2, { T_CHAR,       T_OUTPUT_PORT,T_NONE } },
 { pp_wrong,                1,  2, { T_STRING,     T_NONE,       T_NONE } },
};
```

Report a type error. `Who` is the procedure that received an unexpected type, `what` is a string stating the type that the procedure expected, and `got` is the offending object as received by the procedure.

```
int expected(int who, char *what, int got) {
        char    msg[100];

        sprintf(msg, "%s: expected %s, got",
                string(cddr(who)), what);
        return error(msg, got);
}
```

The `primitive()` function performs a call to a primitive procedure. It first checks the number of arguments and the types of the arguments. If a procedure expects more than three arguments, only the first three types are checked. When a procedure is variadic, only the non-variadic part of the arguments is checked. In both cases, the primitive procedure must check the remaining arguments locally.

The `T_NONE` constant is used to indicate that an argument may have any type.

```
int primitive(int x) {
        int     id, n, k, na, i, a;

        id = cadar(x);
        k = length(x);
        if (k-1 < Primitives[id].min_args)
                return too_few_args(x);
```

```
if (k-1 > Primitives[id].max_args && Primitives[id].max_args >= 0)
        return too_many_args(x);
a = Cdr[x];
na = Primitives[id].max_args < 0? Primitives[id].min_args:
                                Primitives[id].max_args;
if (na > k-1) na = k-1;
for (i=1; i<=na; i++) {
        switch (Primitives[id].arg_types[i-1]) {
        case T_NONE:
                break;
        case T_BOOLEAN:
                if (!boolean_p(Car[a]))
                        return expected(Car[x], "boolean", Car[a]);
                break;
        case T_CHAR:
                if (!char_p(Car[a]))
                        return expected(Car[x], "char", Car[a]);
                break;
        case T_INPUT_PORT:
                if (!input_port_p(Car[a]))
                        return expected(Car[x], "input-port", Car[a]);
                break;
        case T_INTEGER:
                if (!integer_p(Car[a]))
                        return expected(Car[x], "integer", Car[a]);
                break;
        case T_OUTPUT_PORT:
                if (!output_port_p(Car[a]))
                        return expected(Car[x], "output-port", Car[a]);
                break;
        case T_PAIR:
                if (atom_p(Car[a]))
                        return expected(Car[x], "pair", Car[a]);
                break;
        case T_PAIR_OR_NIL:
                if (Car[a] != NIL && atom_p(Car[a]))
                        return expected(Car[x], "pair or ()", Car[a]);
                break;
        case T_PROCEDURE:
                if (    !procedure_p(Car[a]) &&
                        !primitive_p(Car[a])
                )
                        return expected(Car[x], "procedure", Car[a]);
                break;
        case T_STRING:
                if (!string_p(Car[a]))
                        return expected(Car[x], "string", Car[a]);
                break;
        case T_SYMBOL:
                if (!symbol_p(Car[a]))
                        return expected(Car[x], "symbol", Car[a]);
                break;
                case T_VECTOR:
                        if (!vector_p(Car[a]))
                                return expected(Car[x], "vector", Car[a]);
                        break;
                }
                a = Cdr[a];
        }
        n = (*Primitives[id].handler)(x);
        return n;
}
```

`make_application()` creates an application of the specified procedure to a datum. `proc_name` names the procedure to be called. If no symbol with the given name exists or the symbol is not bound, return **`()`**. When the symbol is bound, return

`(#<procedure> (quote #f))`

where **`#<procedure>`** is the value bound to the symbol. **`#F`** is just a placeholder that will be replaced with an actual argument before the application is submitted for evaluation.

```
/* Return (#<procedure> (quote #f)) or () */
int make_application(char *proc_name) {
        int     p_sym, p, app;

        p_sym = find_symbol(proc_name);
        if (p_sym == NIL) return NIL;
        p = lookup(p_sym, Environment);
        if (p == NIL) return NIL;
        p = cadr(p);
        app = alloc(FALSE, NIL);
        app = alloc(S_quote, app);
        app = alloc(app, NIL);
        app = alloc(p, app);
        return app;
}
```

The `has_property_p()` function traverses the tree of nodes `x` and checks whether an object of that tree has a given property. If so it returns one and else zero. The property to be matched is expressed by the function `p`. `has_property_p()` is used to search expressions for applications of syntax transformers and quasiquotation.

```
int has_property_p(int (*p)(int x), int x) {
        if (atom_p(x)) return 0;
        if (Car[x] == S_quote) return 0;
        if (p(x)) return 1;
        while (!atom_p(x)) {
                if (has_property_p(p, Car[x])) return 1;
                x = Cdr[x];
        }
        return 0;
}
```

Check whether `x` is a syntax transformer.

```
int syntactic_symbol_p(int x) {
        int     y;

        if (symbol_p(Car[x])) {
                y = lookup(Car[x], Environment);
                if (y != NIL && syntax_p(cadr(y))) return 1;
        }
        return 0;
}
```

Check whether `x` is an application of quasiquotation.

```
int quasiquotation_p(int x) {
        return Car[x] == S_quasiquote;
}
```

Check whether `x` uses user-defined syntax.

```
int uses_transformer_p(int x) {
        return has_property_p(syntactic_symbol_p, x);
}
```

Check whether `x` uses quasiquotation.

```
int uses_quasiquote_p(int x) {
        return has_property_p(quasiquotation_p, x);
}

int _eval(int x);  /* Evaluate x */
```

Quasiquotation is handled by rewriting quasiquote templates to applications of **`list`**, **`quote`**, and **`append`**. While this technique does not cover full quasiquotation as described in R5RS, it is simple and easy to implement. The only parts not covered by this approach are dotted pairs and improper lists, so these forms may not occur in Scheme 9's quasiquote templates.

The following rules are used to rewrite applications of **`quasiquote`**:

```
    `x   ===>  'x
   `,x   ===>  x
  `(x)   ===>  (list 'x)
 `(,x)   ===>  (list x)
`(,@x)   ===>  (append x)
```

Performing these transformations in C code is too cumbersome, so they are delegated to a Scheme program. This works in the following way: When S9fES starts, it loads the Scheme part of the interpreter, which contains a procedure named **`expand-quasiquote`**, which performs above transformations. To apply it to a quasiquote expression, the interpreter creates a procedure application (using `make_application()`), replaces the **`#f`** placeholder in it with the form to transform, and submits the resulting expression to itself.

The `expand_qq()` function searches an expression for applications of **`quasiquote`** and transforms them as described above. `X` is the expression to search and `app` is an application of **`expand-quasiquote`** to a placeholder.

```
int expand_qq(int x, int app) {
        int     n, a;

        if (Error_flag) return x;
        if (atom_p(x)) return x;
        if (Car[x] == S_quote) return x;
        if (Car[x] == S_quasiquote) {
                cadadr(app) = x;
                return _eval(app);
        }
        n = a = NIL;
        save(n);
        while (!atom_p(x)) {
                if (n == NIL) {
                        n = alloc(expand_qq(Car[x], app), NIL);
                        Car[Stack] = n;
                        a = n;
                }
                else {
                        Cdr[a] = alloc(expand_qq(Car[x], app), NIL);
                        a = Cdr[a];
                }
                x = Cdr[x];
        }
        Cdr[a] = x;
        unsave(1);
        return n;
}
```

The `expand_quasiquote()` function runs `expand_qq()`, but only if the expression `x` actually does contain applications of **`quasiquote`**. It also delivers the application of **`expand-quasiquote`** to `expand_qq()`. This function is a performance hack.

```
int expand_quasiquote(int x) {
        int     app;

        if (Error_flag) return x;
        if (atom_p(x)) return x;
        if (!uses_quasiquote_p(x)) return x;
        app = make_application("expand-quasiquote");
        if (app == NIL) return x;
        save(app);
        x = expand_qq(x, app);
        unsave(1);
        return x;
}
```

"Syntax expansion" is the process that rewrites user-defined syntax as primitive syntax using a set of syntax rules. Like quasiquote expansion, syntax expansion is delegated to a Scheme procedure.

The `expand_all_syntax()` function searches an expression for applications of user-defined syntax transformers and expands them. `x` is the expression to search and `app` is an application of the **`expand-syntax`** procedure, which performs syntax expansion. `expand_all_syntax()` does *not* search applications of **`define-syntax`**, so syntax transformers can be redefined.

```
int expand_all_syntax(int x, int app) {
        int     y, n, a;

        if (Error_flag) return x;
        if (atom_p(x)) return x;
        if (Car[x] == S_quote || Car[x] == S_define_syntax) return x;
        if (symbol_p(Car[x])) {
                y = lookup(Car[x], Environment);
                if (y != NIL && syntax_p(cadr(y))) {
                        cadadr(app) = x;
                        return _eval(app);
                }
        }
        n = a = NIL;
        save(n);
        while (!atom_p(x)) {
                if (n == NIL) {
                        n = alloc(expand_all_syntax(Car[x], app), NIL);
                        Car[Stack] = n;
                        a = n;
                }
                else {
                        Cdr[a] = alloc(expand_all_syntax(Car[x], app), NIL);
                        a = Cdr[a];
                }
                x = Cdr[x];
        }
        Cdr[a] = x;
        unsave(1);
        return n;
}
```

The `expand_syntax()` function runs `expand_all_syntax()`, but only if the expression `x` actually does contain applications of user-defined syntax. Like `expand_quasiquote()`, this function is a performance hack.

```
int expand_syntax(int x) {
        int     app;

        if (Error_flag) return x;
        if (atom_p(x)) return x;
        if (Car[x] == S_quote || Car[x] == S_define_syntax) return x;
        if (!uses_transformer_p(x)) return x;
        app = make_application("expand-syntax");
        if (app == NIL) return x;
        save(app);
        x = expand_all_syntax(x, app);
        unsave(1);
        return x;
}
```

These functions are used to save and restore the evaluator state.

```
#define save_state(v) (State_stack = alloc3((v), State_stack, AFLAG))

static int restore_state(void) {
        int     v;

        if (State_stack == NIL) fatal("restore_state(): stack underflow");
        v = Car[State_stack];
        State_stack = Cdr[State_stack];
        return v;
}
```

The `bind_arguments()` function makes the environment stored in a procedure the current environment and extends that environment by adding a new environment rib. The new rib holds the bindings of the variables of the procedure to some actual arguments passed to that procedure. The parameter `n` is (a node representing) the application of a procedure **p** to a list of arguments **a**:

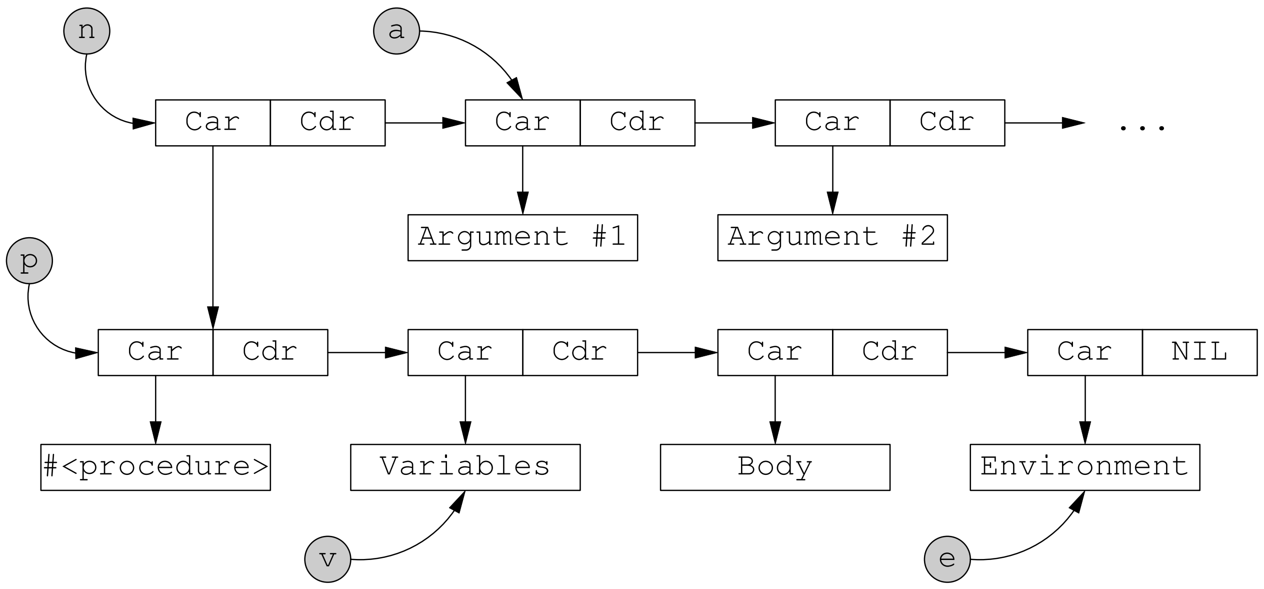


Fig. 15 – Procedure application

The new environment becomes the context of the following operations. The previous context is saved on the global stack. It will be restored when the procedure **p** returns.

```
int bind_arguments(int n, int name) {
        int     p,  /* procedure */
                v,  /* variables of p */
                a,  /* actual arguments */
                e;  /* environment of p */
        int     rib;

        save(Environment);
        p = Car[n];
        v = cadr(p);
        e = cdddr(p);
        a = Cdr[n];
        if (e != NIL) Environment = e;
        rib = NIL;
        save(rib);
        while (!atom_p(v)) {
                if (atom_p(a)) return too_few_args(n);
                Tmp = alloc(Car[a], NIL);
                Tmp = alloc(Car[v], Tmp);
                rib = alloc(Tmp, rib);
                Car[Stack] = rib;
                v = Cdr[v];
                a = Cdr[a];
        }
        if (symbol_p(v)) {
                Tmp = alloc(a, NIL);
                Tmp = alloc(v, Tmp);
                rib = alloc(Tmp, rib);
                Car[Stack] = rib;
        }
        else if (a != NIL) {
                return too_many_args(n);
        }
        unsave(1);
        Environment = make_env(rib, Environment);
        return UNSPECIFIC;
}
```

The `tail_call()` function implements tail call elimination. It is invoked whenever a (non-primitive) Scheme procedure is called. The function examines the state of the caller on the evaluator's state stack. When the caller's state is `MBETA`, the call in progress is a tail call. In this case, the caller's context is removed from the global stack. This is an outline of the global stack and state stack at the time of a procedure call:

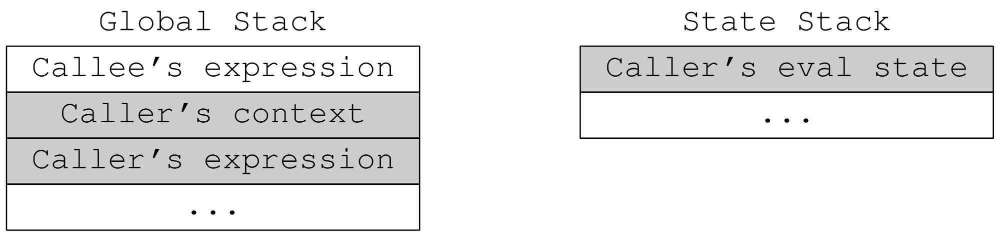


Fig. 16 – Procedure call context

If the caller's evaluator state (on top of the state stack) is `MBETA`, all elements that are rendered in grey are removed from the stacks.

```
void tail_call(void) {
        if (State_stack == NIL || Car[State_stack] != MBETA) return;
        Tmp = unsave(1);
        Environment = Car[Stack];
        unsave(2);
        restore_state();
        save(Tmp);
        Tmp = NIL;
}
```

Evaluate a special form by passing the form to the appropriate handler.

```
int apply_special(int x, int *pc, int *ps) {
        int     sf;

        sf = Car[x];
        if (sf == S_and) return sf_and(x, pc, ps);
        else if (sf == S_begin) return sf_begin(x, pc, ps);
        else if (sf == S_cond) return sf_cond(x, pc, ps);
        else if (sf == S_define) return sf_define(x, pc, ps);
        else if (sf == S_define_syntax) return sf_define_syntax(x, pc, ps);
        else if (sf == S_if) return sf_if(x, pc, ps);
        else if (sf == S_lambda) return sf_lambda(x);
        else if (sf == S_let) return sf_let(x, pc);
        else if (sf == S_letrec) return sf_letrec(x, pc);
        else if (sf == S_quote) return sf_quote(x);
        else if (sf == S_or) return sf_or(x, pc, ps);
        else if (sf == S_set_b) return sf_set_b(x, pc, ps);
        else if (sf == S_syntax_rules) return sf_syntax_rules(x);
        else fatal("internal: unknown special form");
        return UNSPECIFIC;
}
```

Make all variables of a procedure dynamically scoped by removing the lexical environment of that procedure.

```
void make_dynamic(int x) {
        if (procedure_p(x))
                cdddr(x) = NIL; /* clear lexical env. */
}
```

The `_eval()` function implements a finite state machine (FSM) that evaluates a S9fES program by traversing its internal representation. Because S9fES uses trees of nodes as its internal representation, such a FSM is also known as a “tree-walking interpreter”.

The tree-walker of the `_eval()` function does not recurse at the C level, but maintains some state on the global stack and the state stack in order to navigate the tree structure of its input program. When the program runs in constant space, so does `_eval()`.

Because I do not want to fragment the code of `_eval()` with inline annotations, here comes a summary of its mode of operation.

Before the main loop (`while(!Error_flag)`) is entered, the function saves some objects on the global stack. One of these is the "stack bottom" which refers to the first element on the stack that has *not* been pushed during evaluation itself. It is used to check whether the stack runs empty during evaluation. When this happens (and the "continue" flag is clear), evaluation ends. The stack bottom is also used to remove objects pushed by `_eval()` in case of an abnormal termination. When the main loop terminates, `_eval()` restores the state saved before. Hence `_eval()` can be called recursively. This is needed for expanding quasiquotation and syntax transformers as well as for the implementation of **`load`**.

The evaluator uses four internal data objects to maintain its state:

(1) The "continue" flag `c` tells the evaluator that the result of an operation has to be evaluated once again. This happens when a special form handler rewrites an expression or after extracting the body of a procedure during beta reduction. For example, the evaluator rewrites **`((lambda (x) (- x)) 5)`** to **`(- x)`** after binding **`x`** to 5. The resulting expression **`(- x)`** is then re-submitted for evaluation by setting `c`. When the `c` flag is set to two rather than to one, it tells the evaluator that the state indicator `s` was changed and should not be restored in the current cycle.

(2) The "evaluator state" `s` indicates which task the evaluator currently performs. Whenever the evaluator descends into a sublist (the arguments of a procedure application), it pushes the value of `s` to the state stack and then changes it to `MARGS`. When it finishes processing the sublist, it restores the value of `s`. These evaluator states exist:

`MATOM`	Evaluating an atom. This is the original state
`MARGS`	Evaluating the arguments of a procedure call.
`MBETA`	Evaluating the body of a procedure.
`MIFPR`	Evaluating the predicate of an **`if`** form.
`MSETV`	Evaluating the value of a **`set!`** or **`define`** form.
`MDEFN`	Evaluating a function definition using **`define`**.
`MDSYN`	Evaluating a syntax transformer (second argument of **`define-syntax`**).
`MBEGN`	Evaluating an expression of **`begin`** (but not the last one).
`MCONJ`	Evaluating an expression of **`and`** (but not the last one).
`MDISJ`	Evaluating an expression of **`or`** (but not the last one).
`MCOND`	Evaluating a predicate of a clause of **`cond`**.

(3) The `cbn` flag indicates whether an expression that is submitted for re-evaluation shall be applied to its arguments using call by name. When `cbn` is set, symbols will be passed through and only the first argument of procedure applications (the procedure itself) will be evaluated. This is used to avoid duplicate evaluation in **`apply`**, **`and`**, and **`or`**.

(4) The `rib` structure is used to keep intermediate stages during the evaluation of a procedure application. Whenever the evaluator finds a procedure application, it sets up a "rib" structure (which is not to be mixed up with an "environment rib") and sets the state `s` to `MARGS`. The inner loop of the interpreter then uses the rib to evaluate the procedure and its actual arguments. As descibed in detail earlier in this text (pg. 13), the rib structure is a list of four members:

`(arguments append result source)`

When the rib is set up, it holds the arguments yet to be processed, a box where new values are to be appended, the resulting argument list (containing the values of expressions) and the original source expression. The following sequence outlines the state of the rib during the evaluation of the expression **`(* (+ 1 2) (- 5 3))`** (the source expression is omitted because it is constant):

Rib values	**Evaluating**
`(((+ 1 2) (- 5 3)) (()) (()))`	**`*`**
`(((- 5 3)) (()) (#<primitive *> ()))`	**`(+ 1 2)`**
`(() (()) (#<primitive *> 3 ()))`	**`(- 5 3)`**
`(() (()) (#<primitive *> 3 2))`	

When the inner evaluator loop is entered with `s=MARGS` and a rib with an empty argument part, the result part of that rib holds a procedure application that is ready to be passed to a handler. The handler can be deduced from the type of the first member of the result part.

Finally, here is the code of `_eval()`:

```
int _eval(int x) {
        int     m2,     /* Root of result list */
                a,      /* Used to append to result */
                rib;    /* Temp storage for args */
        int     cbn,    /* Use call-by-name in next iteration */
                s,      /* Current state */
                c;      /* Continuation */
        int     name;   /* Name of procedure to apply */

        save(x);
        save(State_stack);
        save(Stack_bottom);
        Stack_bottom = Stack;
        s = MATOM;
        c = 0;
        cbn = 0;
        while (!Error_flag) {
                if (x == NIL) {                 /* () -> () */
                        /* should catch unquoted () */
                        Acc = x;
                        cbn = 0;
                }
                else if (auto_quoting_p(x) ||
                        procedure_p(x) ||
                        primitive_p(x)
                ) {
                        Acc = x;
                        cbn = 0;
                }
                else if (symbol_p(x)) {         /* Symbol -> Value */
                        if (cbn) {
                                Acc = x;
                                cbn = 0;
                        }
                        else {
                                Acc = value_of(x, Environment);
                                if (Acc == UNDEFINED)
                                        error("symbol not bound", x);
                                if (Error_flag) break;
                        }
                }
                else {                          /* (...) -> Value */
                        /*
                         * This block is used to DESCEND into lists.
                         * The following structure is saved on the
                         * Stack: RIB = (args append result source)
                         * The current s is saved on the State_stack.
                         */
                        Acc = x;
                        x = Car[x];
                        save_state(s);
                        /* Check call-by-name built-ins and flag */
                        if (special_p(x) || cbn) {
                                cbn = 0;
                                rib = alloc(Acc, NIL);  /* source */
                                rib = alloc(Acc, rib);  /* result */
                                rib = alloc(NIL, rib);  /* append */
                                rib = alloc(NIL, rib);  /* args */
                                x = NIL;
                        }
                        else {
                                Tmp = alloc(NIL, NIL);
                                rib = alloc(Acc, NIL);  /* source */
                                rib = alloc(Tmp, rib);  /* result */
                                rib = alloc(Tmp, rib);  /* append */
                                rib = alloc(Cdr[Acc], rib); /* args */
                                Tmp = NIL;
                                x = Car[Acc];
                        }
                        save(rib);
                        s = MARGS;
                        continue;
                }
                /*
                 * The following loop is used to ASCEND back to the
                 * root of a list, thereby performing BETA REDUCTION.
                 */
                while (1) if (s == MBETA) {
                        /* Finish BETA reduction */
                        Environment = unsave(1);
                        unsave(1);      /* source expression */
                        s = restore_state();
                }
                else if (s == MARGS) {  /* Append to list, reduce */
```

```
rib = Car[Stack];
x = rib_args(rib);
a = rib_append(rib);
m2 = rib_result(rib);
/* Append new member */
if (a != NIL) Car[a] = Acc;
if (x == NIL) {         /* End of list */
        Acc = m2;
        /* Remember name of caller */
        name = Car[rib_source(Car[Stack])];
        /* Save result (new source expression) */
        Car[Stack] = Acc;
        if (primitive_p(Car[Acc])) {
                if (cadar(Acc) == PP_APPLY)
                        c = cbn = 1;
                Acc = x = primitive(Acc);
        }
        else if (special_p(Car[Acc])) {
                Acc = x = apply_special(Acc, &c, &s);
        }
        else if (procedure_p(Car[Acc])) {
                name = symbol_p(name)? name: NIL;
                tail_call();
                bind_arguments(Acc, name);
                x = caddar(Acc);
                c = 2;
                s = MBETA;
        }
        else {
                error("application of non-procedure",
                        name);
                x = NIL;
        }
        if (c != 2) {
                unsave(1); /* source expression */
                s = restore_state();
        }
        /* Leave the ASCENDING loop and descend */
        /* once more into N. */
        if (c) break;
}
else if (atom_p(x)) {
        error("improper list in application", x);
        x = NIL;
        break;
}
else {          /* N =/= NIL: Append to list */
        /* Create space for next argument */
        Cdr[a] = alloc(NIL, NIL);
        rib_append(rib) = Cdr[a];
        rib_args(rib) = Cdr[x];
        x = Car[x];     /* Evaluate next member */
```

```
                        break;
                }
        }
        else if (s == MIFPR) {
                x = unsave(1);
                unsave(1);      /* source expression */
                s = restore_state();
                if (Acc != FALSE)
                        x = cadr(x);
                else
                        x = caddr(x);
                c = 1;
                break;
        }
        else if (s == MCONJ || s == MDISJ) {
                Car[Stack] = cdar(Stack);
                if (    (Acc == FALSE && s == MCONJ) ||
                        (Acc != FALSE && s == MDISJ) ||
                        Car[Stack] == NIL
                ) {
                        unsave(2);      /* state, source expr */
                        s = restore_state();
                        x = Acc;
                        cbn = 1;
                }
                else if (cdar(Stack) == NIL) {
                        x = caar(Stack);
                        unsave(2);      /* state, source expr */
                        s = restore_state();
                }
                else {
                        x = caar(Stack);
                }
                c = 1;
                break;
        }
        else if (s == MCOND) {
                if (Acc != FALSE) {
                        x = cdar(Car[Stack]);
                        if (length(x) > 1)
                                Acc = x = alloc(S_begin, x);
                        else
                                x = Car[x];
                        unsave(2);      /* state, source expr */
                        s = restore_state();
                }
                else if (cdar(Stack) == NIL) {
                        unsave(2);      /* state, source expr */
                        s = restore_state();
                        x = UNSPECIFIC;
                }
                else {
```

```
                                Car[Stack] = cdar(Stack);
                                x = caaar(Stack);
                                if (x == S_else && cdar(Stack) == NIL)
                                        x = TRUE;
                        }
                        c = 1;
                        break;
                }
                else if (s == MBEGN) {
                        Car[Stack] = cdar(Stack);
                        if (cdar(Stack) == NIL) {
                                x = caar(Stack);
                                unsave(2);      /* state, source expr*/
                                s = restore_state();
                        }
                        else {
                                x = caar(Stack);
                        }
                        c = 1;
                        break;
                }
                else if (s == MSETV || s == MDEFN || s == MDSYN) {
                        if (s == MDEFN) make_dynamic(Acc);
                        if (s == MDSYN && !syntax_p(Acc)) {
                                error("define-syntax: expected syntax, got",
                                        Acc);
                                break;
                        }
                        x = unsave(1);
                        unsave(1);      /* source expression */
                        s = restore_state();
                        Car[x] = Acc;
                        Acc = x = UNSPECIFIC;
                        c = 0;
                        break;
                }
                else { /* s == MATOM */
                        break;
                }
                if (c) {        /* Continue evaluation if requested */
                        c = 0;
                        continue;
                }
                if (Stack == Stack_bottom) break;
        }
        while (Stack != Stack_bottom) unsave(1);
        Stack_bottom = unsave(1);
        State_stack = unsave(1);
        unsave(1);
        return Acc;             /* Return the evaluated expr */
}
```

In fact, `_eval()` does only half of the job. The other half is done in `eval()` below.

```
int eval(int x) {
        save(x);
        x = expand_quasiquote(x);
        Car[Stack] = x;
        x = expand_syntax(x);
        x = _eval(x);
        unsave(1);
        return x;
}
```

Read Eval Print Loop

The "read eval print loop" – or REPL – is the interactive interface of a Scheme environment. It typically issues a prompt, reads an expression, evaluates it, prints it and loops. Hence its name.

The `clear_local_envs()` function throws away any environment ribs that may be left over after an aborted evaluation.

```
void clear_local_envs(void) {
        while (Cdr[Environment] != NIL)
                Environment = Cdr[Environment];
}
```

These are signal handlers. The `SIGINT` handler just reports an error. Doing so will set `Error_flag` which aborts evaluation and returns to the REPL.

```
#ifndef NO_SIGNALS
void keyboard_interrupt(int sig) {
        error("interrupted", NOEXPR);
        signal(SIGINT, keyboard_interrupt);
}

void keyboard_quit(int sig) {
        fatal("received quit signal, exiting");
}
#endif
```

`repl()` implements the REPL. No surprise. The `sane_env` variable binds to a protected box that keeps a copy of a usable environment. Whenever an evaluation is aborted, the global environment is restored using `sane_env`. The REPL exits when the input stream is exhausted.

```
void repl(void) {
        int     n, sane_env;

        sane_env = alloc(NIL, NIL);
        save(sane_env);
        if (!Quiet_mode) {
                signal(SIGINT, keyboard_interrupt);
                signal(SIGQUIT, keyboard_quit);
        }
        while (1) {
                Error_flag = 0;
                Input_port = 0;
                Output_port = 1;
                clear_local_envs();
                Car[sane_env] = Environment;
                if (!Quiet_mode) pr("> ");
                Program = xread();
                if (Program == ENDOFFILE) break;
                if (!Error_flag) n = eval(Program);
                if (!Error_flag && n != UNSPECIFIC) {
                        print(n);
                        pr("\n");
                        Car[S_latest] = n;
                }
                if (Error_flag) Environment = Car[sane_env];
        }
        unsave(1);
        pr("\n");
}
```

Initialization

The `make_primitive()` function creates a primitive procedure object that is represented by the following node structure:

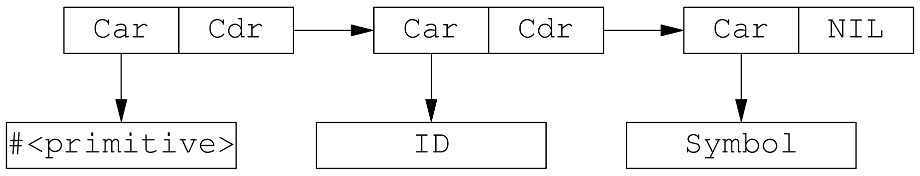


Fig. 17 – Primitive procedure structure

The atomic `ID` field contains a an offset into the `Primitives` array. The `Symbol` field refers to the symbol that the primitive is bound to, so that the external representation of primitives can be made a bit more informative, i.e. **`#<primitive car>`** instead of just **`#<primitive>`**.

```
int make_primitive(char *s, int id) {
        int     n;

        n = add_symbol(s);
        n = alloc3(id, n, AFLAG);
        return alloc3(S_primitive, n, AFLAG);
}
```

The `add_primitive()` function creates a new primitive and extends the global environment rib with it.

```
void add_primitive(char *s, int id) {
        int     v;
        v = add_symbol(s);
        Environment = extend(v, make_primitive(s, id), Environment);
}
```

Populate the initial global environment with some procedures. Also add the `**` variable whose value is the result of the most recently evaluated program. `**` is not part of R5RS.

```
int make_initial_env(void) {
        Environment = alloc(NIL, NIL);
        Environment = extend(add_symbol("**"), NIL, Environment);
        S_latest = cdadr(Environment);
        add_primitive("*", PP_TIMES);
        add_primitive("+", PP_PLUS);
        add_primitive("-", PP_MINUS);
        add_primitive("<", PP_LESS);
        add_primitive("<=", PP_LESS_EQUAL);
        add_primitive("=", PP_EQUAL);
        add_primitive(">", PP_GREATER);
        add_primitive(">=", PP_GREATER_EQUAL);
        add_primitive("apply", PP_APPLY);
        add_primitive("boolean?", PP_BOOLEAN_P);
        add_primitive("car", PP_CAR);
        add_primitive("cdr", PP_CDR);
        add_primitive("char->integer", PP_CHAR_TO_INTEGER);
        add_primitive("char-alphabetic?", PP_CHAR_ALPHABETIC_P);
        add_primitive("char-ci<=?", PP_CHAR_CI_LE_P);
        add_primitive("char-ci<?", PP_CHAR_CI_LT_P);
        add_primitive("char-ci=?", PP_CHAR_CI_EQ_P);
        add_primitive("char-ci>=?", PP_CHAR_CI_GE_P);
        add_primitive("char-ci>?", PP_CHAR_CI_GT_P);
        add_primitive("char-downcase", PP_CHAR_DOWNCASE);
        add_primitive("char-lower-case?", PP_CHAR_LOWER_CASE_P);
        add_primitive("char-numeric?", PP_CHAR_NUMERIC_P);
        add_primitive("char-upcase", PP_CHAR_UPCASE);
        add_primitive("char-upper-case?", PP_CHAR_UPPER_CASE_P);
        add_primitive("char-whitespace?", PP_CHAR_WHITESPACE_P);
        add_primitive("char<=?", PP_CHAR_LE_P);
        add_primitive("char<?", PP_CHAR_LT_P);
        add_primitive("char=?", PP_CHAR_EQ_P);
        add_primitive("char>=?", PP_CHAR_GE_P);
        add_primitive("char>?", PP_CHAR_GT_P);
        add_primitive("char?", PP_CHAR_P);
        add_primitive("close-input-port", PP_CLOSE_INPUT_PORT);
        add_primitive("close-output-port", PP_CLOSE_OUTPUT_PORT);
        add_primitive("cons", PP_CONS);
        add_primitive("current-input-port", PP_CURRENT_INPUT_PORT);
        add_primitive("current-output-port", PP_CURRENT_OUTPUT_PORT);
        add_primitive("display", PP_DISPLAY);
        add_primitive("eq?", PP_EQ_P);
        add_primitive("eof-object?", PP_EOF_OBJECT_P);
        add_primitive("input-port?", PP_INPUT_PORT_P);
        add_primitive("integer->char", PP_INTEGER_TO_CHAR);
        add_primitive("integer?", PP_INTEGER_P);
        add_primitive("list->string", PP_LIST_TO_STRING);
        add_primitive("list->vector", PP_LIST_TO_VECTOR);
        add_primitive("load", PP_LOAD);
        add_primitive("make-string", PP_MAKE_STRING);
        add_primitive("make-vector", PP_MAKE_VECTOR);
        add_primitive("open-input-file", PP_OPEN_INPUT_FILE);
        add_primitive("open-output-file", PP_OPEN_OUTPUT_FILE);
        add_primitive("output-port?", PP_OUTPUT_PORT_P);
        add_primitive("pair?", PP_PAIR_P);
        add_primitive("peek-char", PP_PEEK_CHAR);
        add_primitive("procedure?", PP_PROCEDURE_P);
        add_primitive("quotient", PP_QUOTIENT);
        add_primitive("read", PP_READ);
        add_primitive("read-char", PP_READ_CHAR);
        add_primitive("remainder", PP_REMAINDER);
        add_primitive("set-car!", PP_SET_CAR_B);
        add_primitive("set-cdr!", PP_SET_CDR_B);
        add_primitive("set-input-port!", PP_SET_INPUT_PORT_B);
        add_primitive("set-output-port!", PP_SET_OUTPUT_PORT_B);
        add_primitive("string->list", PP_STRING_TO_LIST);
        add_primitive("string->symbol", PP_STRING_TO_SYMBOL);
        add_primitive("string-append", PP_STRING_APPEND);
        add_primitive("string-copy", PP_STRING_COPY);
        add_primitive("string-fill!", PP_STRING_FILL_B);
        add_primitive("string-length", PP_STRING_LENGTH);
        add_primitive("string-ref", PP_STRING_REF);
        add_primitive("string-set!", PP_STRING_SET_B);
        add_primitive("string-ci<=?", PP_STRING_CI_LE_P);
        add_primitive("string-ci<?", PP_STRING_CI_LT_P);
        add_primitive("string-ci=?", PP_STRING_CI_EQ_P);
        add_primitive("string-ci>=?", PP_STRING_CI_GE_P);
        add_primitive("string-ci>?", PP_STRING_CI_GT_P);
        add_primitive("string<=?", PP_STRING_LE_P);
        add_primitive("string<?", PP_STRING_LT_P);
        add_primitive("string=?", PP_STRING_EQ_P);
        add_primitive("string>=?", PP_STRING_GE_P);
        add_primitive("string>?", PP_STRING_GT_P);
        add_primitive("string?", PP_STRING_P);
        add_primitive("substring", PP_SUBSTRING);
        add_primitive("symbol->string", PP_SYMBOL_TO_STRING);
        add_primitive("symbol?", PP_SYMBOL_P);
        add_primitive("syntax->list", PP_SYNTAX_TO_LIST);
        add_primitive("unquote", PP_UNQUOTE);
        add_primitive("unquote-splicing", PP_UNQUOTE_SPLICING);
        add_primitive("vector->list", PP_VECTOR_TO_LIST);
        add_primitive("vector-fill!", PP_VECTOR_FILL_B);
        add_primitive("vector-length", PP_VECTOR_LENGTH);
        add_primitive("vector-ref", PP_VECTOR_REF);
        add_primitive("vector-set!", PP_VECTOR_SET_B);
        add_primitive("vector?", PP_VECTOR_P);
        add_primitive("write", PP_WRITE);
        add_primitive("write-char", PP_WRITE_CHAR);
        add_primitive("wrong", PP_WRONG);
        Environment = alloc(Environment, NIL);
        return Environment;
}
```

Initialize the interpreter: release the ports, lock the standard input/output ports, setup the node pools, create the type tags and Scheme keywords.

```
void init(void) {
        int     i;

        for (i=2; i<MAX_PORTS; i++) Ports[i] = NULL;
        Ports[0] = stdin;
        Ports[1] = stdout;
        Port_flags[0] = LFLAG;
        Port_flags[1] = LFLAG;
        Input_port = 0;
        Output_port = 1;
        new_segment();
        gc();
        S_char = add_symbol("#<char>");
        S_input_port = add_symbol("#<input-port>");
        S_integer = add_symbol("#<integer>");
        S_output_port = add_symbol("#<output-port>");
        S_primitive = add_symbol("#<primitive>");
        S_procedure = add_symbol("#<procedure>");
        S_string = add_symbol("#<string>");
        S_symbol = add_symbol("#<symbol>");
        S_syntax = add_symbol("#<syntax>");
        S_vector = add_symbol("#<vector>");
        S_else = add_symbol("else");
        S_and = add_symbol("and");
        S_begin = add_symbol("begin");
        S_cond = add_symbol("cond");
        S_define = add_symbol("define");
        S_define_syntax = add_symbol("define-syntax");
        S_if = add_symbol("if");
        S_lambda = add_symbol("lambda");
        S_let = add_symbol("let");
        S_letrec = add_symbol("letrec");
        S_quote = add_symbol("quote");
        S_quasiquote = add_symbol("quasiquote");
        S_unquote = add_symbol("unquote");
        S_unquote_splicing = add_symbol("unquote-splicing");
        S_or = add_symbol("or");
        S_set_b = add_symbol("set!");
        S_syntax_rules = add_symbol("syntax-rules");
        Environment = make_initial_env();
        Program = TRUE;
        rehash(Car[Environment]);
}
```

The `load_library()` function loads the Scheme part of the interpreter. It attempts to open the file `s9.scm` in a set of directories specified in the `S9FES_LIBRARY_PATH` environment variable. If that variable is undefined, its value defaults to `.:~/.s9fes:/usr/local/share/s9fes`. Path names are separated by colons (`:`). A tilde (~) at the beginning of a path is replaced with the value of the environment variable `HOME`. `load_library()` returns when a library could be **`load`**ed successfully. When no library could be loaded, a fatal error is reported.

```
void load_library(void) {
        char    *path, buf[100], *p;
        char    libpath[256];
        char    *home;

        path = getenv("S9FES_LIBRARY_PATH");
        home = getenv("HOME");
        if (path == NULL)
                path = strcpy(buf, ".:~/.s9fes:/usr/local/share/s9fes");
        p = strtok(path, ":");
        while (p != NULL) {
                if (p[0] == '~') {
                        if (strlen(p) + strlen(home) > 240)
                                fatal("path too long in S9FES_LIBRARY_PATH");
                        sprintf(libpath, "%s%s/s9.scm", home, &p[1]);
                }
                else {
                        if (strlen(p) > 248)
                                fatal("path too long in S9FES_LIBRARY_PATH");
                        sprintf(libpath, "%s/s9.scm", p);
                }
                if (load(libpath) == 0) {
                        /* printf("Library: %s\n", libpath); */
                        return;
                }
                p = strtok(NULL, ":");
        }
        fatal("could not load library: \"s9.scm\"");
}
```

Load initialization commands from `~/.s9fes/rc` if that file exists.

```
void load_rc(void) {
        char    rcpath[256];
        char    *home;

        home = getenv("HOME");
        if (home == NULL) return;
        if (strlen(home) + 12 >= 256) fatal("path too long in HOME");
        sprintf(rcpath, "%s/.s9fes/rc", home);
        load(rcpath);
}
```

```
void usage(void) {
        printf("Usage: s9 [-q] [-v] [-f program]\n");
        exit(1);
}
```

Ready to lift off...

```
int main(int argc, char **argv) {
        init();
        argv++;
        load_library();
        while (*argv != NULL) {
                if (**argv != '-') break;
                (*argv)++;
                while (**argv) {
                        switch (**argv)  {
                        case 'q':
                                Quiet_mode = 1;
                                (*argv)++;
                                break;
                        case 'f':
                                if (argv[1] == NULL) usage();
                                load_rc();
                                if (load(argv[1]))
                                        error("program file not found",
                                                NOEXPR);
                                exit(Error_flag? 1: 0);
                                break;
                        case 'v':
                                printf("Version: %s\n", VERSION);
                                exit(1);
                                break;
                        default:
                                usage();
                                break;
                        }
                }
                argv++;
        }
        if (!Quiet_mode) printf("Scheme 9 from Empty Space"
                                " (C) 2007 Nils M Holm\n");
        load_rc();
        repl();
        return 0;
}
```

Scheme Part

```
;;
;; Scheme 9 from Empty Space
;; Copyright (C) 2007 Nils M Holm <nmh@t3x.org>
;;
```

Library Procedures

Some standard Scheme procedures are implemented in Scheme rather in C for a variety of reasons: because they are too complex, too trivial, or because they depend on other procedures that are implemented in Scheme. Most of the procedures that follow here are R5RS compliant. Exceptions are labeled as such.

```
(define (not x) (eq? #f x))

(define number? integer?)

(define (port? x)
  (or (input-port? x)
      (output-port? x)))

(define (eqv? a b)
  (cond
    ((number? a)
      (and (number? b) (= a b)))
    ((char? a)
      (and (char? b) (char=? a b)))
    (else (eq? a b))))

(define (equal? a b)
  (cond
    ((eq? a b) #t)
    ((string? a)
      (and (string? b) (string=? a b)))
    ((and (pair? a) (pair? b))
      (and (equal? (car a) (car b))
           (equal? (cdr a) (cdr b))))
    ((and (vector? a) (vector? b))
      (equal? (vector->list a) (vector->list b)))
    (else (eqv? a b))))

(define (null? x)
  (eq? '() x))

(define (list? x)
  (or (null? x)
      (and (pair? x)
           (list? (cdr x)))))

(define (caaaar x) (car (car (car (car x)))))
(define (caaadr x) (car (car (car (cdr x)))))
(define (caadar x) (car (car (cdr (car x)))))
```

```
(define (caaddr x) (car (car (cdr (cdr x)))))
(define (cadaar x) (car (cdr (car (car x)))))
(define (cadadr x) (car (cdr (car (cdr x)))))
(define (caddar x) (car (cdr (cdr (car x)))))
(define (cadddr x) (car (cdr (cdr (cdr x)))))
(define (cdaaar x) (cdr (car (car (car x)))))
(define (cdaadr x) (cdr (car (car (cdr x)))))
(define (cdadar x) (cdr (car (cdr (car x)))))
(define (cdaddr x) (cdr (car (cdr (cdr x)))))
(define (cddaar x) (cdr (cdr (car (car x)))))
(define (cddadr x) (cdr (cdr (car (cdr x)))))
(define (cdddar x) (cdr (cdr (cdr (car x)))))
(define (cddddr x) (cdr (cdr (cdr (cdr x)))))

(define (caaar x) (car (car (car x))))
(define (caadr x) (car (car (cdr x))))
(define (cadar x) (car (cdr (car x))))
(define (caddr x) (car (cdr (cdr x))))
(define (cdaar x) (cdr (car (car x))))
(define (cdadr x) (cdr (car (cdr x))))
(define (cddar x) (cdr (cdr (car x))))
(define (cdddr x) (cdr (cdr (cdr x))))

(define (caar x) (car (car x)))
(define (cadr x) (car (cdr x)))
(define (cdar x) (cdr (car x)))
(define (cddr x) (cdr (cdr x)))

(define (assoc x a)
  (cond ((null? a) #f)
    ((equal? (caar a) x) (car a))
    (else (assoc x (cdr a)))))

(define (assq x a)
  (cond ((null? a) #f)
    ((eq? (caar a) x) (car a))
    (else (assq x (cdr a)))))

(define (assv x a)
  (cond ((null? a) #f)
    ((eqv? (caar a) x) (car a))
    (else (assv x (cdr a)))))
```

`Map-car` is not a standard Scheme procedure, but it is useful for implementing some other higher order functions such as **`map`**, **`fold-left`**, etc. **`Map-car`** is like **`map`** with a single list argument:

```
(map f '(x y z))  ===>  (list (f x) (f y) (f z))
```

```
(define (map-car f a)
  (letrec
    ((mapc
       (lambda (a r)
         (cond ((null? a) (reverse r))
           (else (mapc (cdr a)
                       (cons (f (car a)) r)))))))
    (mapc a '()))))
```

`Fold-left` is not a standard Scheme procedure, either, but it is very useful when implementing variadic functions. The **`let`** in the value of the below **`define`** makes sure that **`map-car`** is included in the lexical environment of **`fold-left`**. This method prevents later re-definitions of **`map-car`** from altering **`fold-left`**. This pattern will be used in all functions using **`map-car`** or **`fold-left`**.

The purpose of **`fold-left`** is to fold lists into values:

```
(fold-left R 0 '(1 2 3)) ===> (R (R (R 0 1) 2) 3)
(fold-left R 0 '(1 2 3) '(a b c)) ===> (R (R (R 0 1 a) 2 b) 3 c)
```
...

```
(define fold-left
  (let ((map-car map-car))
    (lambda (f b . a*)
      (letrec
        ((carof
           (lambda (a)
             (map-car car a)))
         (cdrof
           (lambda (a)
             (map-car cdr a)))
         (fold
           (lambda (a* r)
             (cond ((null? (car a*)) r)
               (else (fold (cdrof a*)
                       (apply f r (carof a*))))))))
        (cond
          ((null? a*)
            (wrong "fold-left: too few arguments"))
          ((null? (car a*)) b)
          (else (fold a* b)))))))
```

`fold-right` is included for symmetry with **`fold-left`**. The difference to **`fold-left`** is that this function folds its lists to the right:

```
 (fold-left cons 0 '(1 2 3))  ===>  (((0 . 1) . 2) . 3)
(fold-right cons 0 '(1 2 3))  ===>  (1 . (2 . (3 . 0)))
```

```
(define fold-right
  (let ((map-car map-car))
    (lambda (f b . a*)
      (letrec
        ((carof
           (lambda (a)
             (map-car car a)))
         (cdrof
           (lambda (a)
             (map-car cdr a)))
         (foldr
           (lambda (a* r)
             (cond ((null? (car a*)) r)
               (else (foldr (cdrof a*)
                       (apply f (append (carof a*) (list r)))))))))
        (cond
          ((null? a*)
            (wrong "fold-right: too few arguments"))
          ((null? (car a*)) b)
          (else (foldr (map reverse a*) b)))))))


(define (reverse a)
  (letrec
    ((reverse2
       (lambda (a b)
         (cond ((null? a) b)
           (else (reverse2 (cdr a)
                   (cons (car a) b)))))))
    (reverse2 a '())))

(define append
  (let ((fold-left fold-left))
    (lambda a
      (letrec
        ((append2
           (lambda (a b)
             (cond ((null? a) b)
               (else (append2 (cdr a) (cons (car a) b))))))
         (append-wrapper
           (lambda (a b)
             (cond ((null? b) a)
               (else (append2 (reverse a) b))))))
        (fold-left append-wrapper '() a)))))

(define (length x)
  (letrec
    ((length2
       (lambda (x r)
         (cond ((null? x) r)
           (else (length2 (cdr x) (+ r 1)))))))
    (length2 x 0)))

(define (list . x) x)

(define (list-ref x n)
  (car (list-tail x n)))

(define (list-tail x n)
  (cond ((zero? n) x)
    ((null? x) (wrong "list-tail: index out of range"))
    (else (list-tail (cdr x) (- n 1)))))
```

```
(define (member x a)
  (cond ((null? a) #f)
    ((equal? (car a) x) a)
    (else (member x (cdr a)))))

(define (memq x a)
  (cond ((null? a) #f)
    ((eq? (car a) x) a)
    (else (memq x (cdr a)))))

(define (memv x a)
  (cond ((null? a) #f)
    ((eqv? (car a) x) a)
    (else (memv x (cdr a)))))

(define map
  (let ((map-car map-car))
    (lambda (f . a*)
      (letrec
        ((carof
           (lambda (a)
             (map-car car a)))
         (cdrof
           (lambda (a)
             (map-car cdr a)))
         (map2
           (lambda (a* r)
             (cond ((null? (car a*)) (reverse r))
               (else (map2 (cdrof a*)
                     (cons (apply f (carof a*)) r)))))))
        (cond
          ((null? a*)
            (wrong "map: too few arguments"))
          (else (map2 a* '())))))))

(define (for-each f . a*)
  (cond
    ((null? a*)
      (wrong "for-each: too few arguments"))
    (else (apply map f a*)
          (if #f #f))))

(define (abs x) (if (< x 0) (- x) x))

(define (even? x) (zero? (remainder x 2)))

(define (expt x y)
  (letrec
    ((square
       (lambda (x)
         (* x x)))
     (expt2
       (lambda (x y)
         (cond
           ((zero? y) 1)
           ((even? y)
             (square (expt2 x (quotient y 2))))
           (else (* x (square (expt2 x (quotient y 2)))))))))
    (if (negative? y)
        (wrong "expt: negative exponent" y)
        (expt2 x y))))

(define gcd
  (let ((fold-left fold-left))
    (lambda a
      (letrec
        ((gcd2
           (lambda (a b)
             (cond ((zero? b) a)
               ((zero? a) b)
               ((< a b) (gcd2 a (remainder b a)))
               (else (gcd2 b (remainder a b)))))))
        (fold-left gcd2 0 (map (lambda (x) (abs x))
                               a))))))

(define lcm
  (let ((fold-left fold-left))
    (lambda a
      (letrec
        ((lcm2
           (lambda (a b)
             (let ((cd (gcd a b)))
               (* cd (* (quotient a cd)
                        (quotient b cd)))))))
        (fold-left lcm2 1 (map (lambda (x) (abs x))
                          a))))))

(define max
  (let ((fold-left fold-left))
    (lambda (a . b)
      (fold-left (lambda (a b)
                   (if (> a b) a b))
                 a b))))

(define min
  (let ((fold-left fold-left))
    (lambda (a . b)
      (fold-left (lambda (a b)
                   (if (< a b) a b))
                 a b))))

(define (modulo a b)
  (let ((rem (remainder a b)))
    (cond ((zero? rem) 0)
      ((eq? (negative? a) (negative? b)) rem)
      (else (+ b rem)))))

(define (negative? x) (< x 0))

(define (odd? x) (not (even? x)))

(define (positive? x) (> x 0))

(define (sqrt square)
  (letrec
    ((sqrt2 (lambda (x last)
       (cond
         ((= last x) x)
         ((= last (+ 1 x))
           (if (> (* x x) square) (- x 1) x))
         (else (sqrt2 (quotient
                        (+ x (quotient square x))
                        2)
                      x))))))
    (if (negative? square)
        (wrong "sqrt: negative argument" square)
        (sqrt2 square 0))))

(define (zero? x) (= 0 x))

(define (number->string n . radix)
  (letrec
    ((digits
       (list->vector (string->list "0123456789abcdef")))
     (conv
       (lambda (n rdx res)
         (cond ((zero? n) res)
           (else (conv (quotient n rdx) rdx
                   (cons (vector-ref digits (remainder n rdx))
                         res))))))
     (get-radix
       (lambda ()
         (cond ((null? radix) 10)
           ((< 1 (car radix) 17) (car radix))
           (else (wrong "bad radix in number->string" radix))))))
    (let ((r (get-radix)))
      (cond
        ((zero? n) "0")
        ((negative? n)
          (list->string
            (cons #\- (conv (abs n) r '()))))
        (else (list->string (conv n r '())))))))

(define (string . x) (list->string x))
```

```
(define (string->number str . radix)
  (letrec
    ((digits
       (string->list "0123456789abcdef"))
     (value-of-digit
       (lambda (x)
         (letrec
           ((v (lambda (x d n)
             (cond ((null? d) 17)
               ((char=? (car d) x) n)
               (else (v x (cdr d) (+ n 1)))))))
           (v (char-downcase x) digits 0))))
     (conv3
       (lambda (lst res rdx)
         (cond ((null? lst) res)
           (else (let ((dval (value-of-digit (car lst))))
                   (and (< dval rdx)
                        (conv3 (cdr lst)
                          (+ (value-of-digit (car lst))
                             (* res rdx))
                          rdx))))))))
     (conv
       (lambda (lst rdx)
         (if (null? lst) #f (conv3 lst 0 rdx))))
     (sconv
       (lambda (lst rdx)
         (cond
           ((null? lst) #f)
           ((char=? (car lst) #\+)
             (conv (cdr lst) rdx))
           ((char=? (car lst) #\-)
             (let ((r (conv (cdr lst) rdx)))
               (if r (- r) #f)))
           (else (conv lst rdx)))))
     (get-radix
       (lambda ()
         (cond ((null? radix) 10)
           ((< 1 (car radix) 17) (car radix))
           (else (wrong "bad radix in string->number" radix))))))
    (sconv (string->list str) (get-radix))))

(define (vector . x) (list->vector x))

(define (newline . port)
  (apply display #\newline port))

(define (call-with-input-file file proc)
  (proc (open-input-file file)))

(define (call-with-output-file file proc)
  (proc (open-output-file file)))
```

```
(define with-input-from-file
  (let ((set-input-port! set-input-port!))
    (lambda (file thunk)
      (let ((outer-port (current-input-port))
            (new-port (open-input-file file)))
        (set-input-port! new-port)
        (let ((input (thunk)))
          (close-input-port new-port)
          (set-input-port! outer-port)
          input)))))

(define with-output-to-file
  (let ((set-output-port! set-output-port!))
    (lambda (file thunk)
      (let ((outer-port (current-output-port))
            (new-port (open-output-file file)))
        (set-output-port! new-port)
        (thunk)
        (close-output-port new-port)
        (set-output-port! outer-port)))))
```

Syntax Expander

The **expand-syntax** procedure is itself not part of R5RS, but it performs (an extended subset of) the syntax transformation described in R5RS. Given a syntax transformer, it expands an application of that syntax transformer to a form that is free of user-defined syntax. For example,

```
(define-syntax pairs
  (syntax-rules ()
    ((_ x ...)
       (list '(x x) ...))))

(expand-syntax '(pairs 1 2 3)) => (list '(1 1) '(2 2) '(3 3))
```

Expand-syntax defines its helper functions in a **letrec** to avoid name space pollution.

```
(define (expand-syntax form)
  (letrec
```

Extend environment `env`, `x` = name, `v` = value.

```
    ((ext-env
       (lambda (x v env)
         (cons (cons x v) env)))
```

Match-ellipsis matches an ellipsis (`...`) in a pattern. **Form** is the form to be matched against the ellipsis, **pattern** is the rest of the pattern (the part that follows the ellipsis), **literals** is the list of literals that is passed through to **match**, and **env** is the environment created to far.

This function employs a longest match approach, so given the arguments

```
form     = (x x x k k k)
pattern  = (k) ; this is the part that follows the ellipsis
literals = (k)
env      = ()
```

it will return **`((... . (x x x k k)))`**. Note that this is slightly more than R5RS specifies, although many implementations include this extension. R5RS requires that **`pattern`** be empty. Hence it does not allow patterns of the form **`(x ... y)`**, but only **`(x ...)`**.

```
(match-ellipsis
  (lambda (form pattern literals env)
    (letrec
      ((try-match (lambda (head tail)
        (let ((v (match tail pattern literals env)))
          (cond (v (ext-env '... (reverse head) v))
            ((null? head) #f)
            (else (try-match (cdr head)
                             (cons (car head) tail))))))))
      (try-match (reverse form) ()))))
```

`Match` matches a pattern against a form. When the form matches the pattern, it returns a list containing the associations between variables in the pattern and matched subforms. When the pattern does not match, it returns **`#f`**. For example,

```
(match '(k 2 3) '(k 2 v) '(k) '()) => ((v . 3))
```

An environment is returned because the form matches the pattern. The pattern variable **`v`** is included in the environment, because it matches the 3 in the form. **`K`** is not included in the environment, because it is a literal (a.k.a. keyword).

```
(match
  (lambda (form pattern literals env)
    (letrec
      ((_match (lambda (form pattern env)
        (cond
          ((memq pattern literals)
            (if (eq? form pattern) env #f))
          ((and (pair? pattern) (eq? (car pattern) '...))
            (match-ellipsis form (cdr pattern) literals env))
          ((symbol? pattern)
            (ext-env pattern form env))
          ((and (pair? pattern) (pair? form))
            (let ((e (_match (car form) (car pattern) env)))
              (and e (_match (cdr form) (cdr pattern) e))))
          (else (if (equal? form pattern) env #f))))))
      (_match form pattern env))))
```

The **`find-rule`** procedure attempts to finds a rule whose pattern matches a given form. **`Rules`** is a list containing all syntax rules of the syntax transformer being applied. **`Name`** is the name of the syntax transformer. When a matching pattern is found return a list containing the pattern, the template associated with that pattern, and the environment created by matching the form. When no pattern matches, the syntax of **`form`** is wrong and an error is reported.

```
(find-rule
  (lambda (form rules name literals)
    (cond
      ((null? rules)
        (wrong "bad syntax" name))
      (else (let ((e (match form (caar rules) literals '())))
              (if e (list (caar rules) (cadar rules) e)
                    (find-rule form (cdr rules) name literals)))))))
```

`Map-improper` is like **`map-car`** but accepts improper lists, too.

```
(map-improper
  (lambda (f a)
    (letrec
      ((map-i
         (lambda (a r)
           (cond ((null? a) (reverse r))
             ((not (pair? a)) (append (reverse r) (f a)))
             (else (map-i (cdr a) (cons (f (car a)) r)))))))
      (map-i a '()))))
```

`Subst-ellipsis` creates a list of forms. Each form of the list is created by replacing the variable **`var`** with one of its values in the template **`tmpl`**. All possible values of **`var`** are contained in **`val*`**. **`Subst-ellipsis`** also substitutes values for all other variables contained in **`env`**. For example,

```
(subst-ellipsis 'v '(k v) '(1 2 3) '((k . -)))
=> ((- 1) (- 2) (- 3))
```

```
(subst-ellipsis
  (lambda (var tmpl val* env)
    (map (lambda (v)
           (tmpl->form #f tmpl (cons (cons var v) env)))
         val*)))
```

Substitute names of **`env`** by values of **`env`** in **`form`**. The **`pattern`** is used to find the names of variables when substituting ellipses. **`Pattern`** may be set to **`#f`** to indicate that no variable is available for ellipsis substitution. In this case, ellipses will be treated like ordinary variables.

```
(tmpl->form
  (lambda (pattern form env)
    (cond
      ((not (pair? form))
        (let ((v (assv form env)))
          (if v (cdr v) form)))
      ((and (pair? form)
            (pair? (cdr form))
            (eq? (cadr form) '...))
        (let ((var (if (pair? pattern) (car pattern) pattern)))
          (let ((v-ell (assq '... env))
                (v-var (assq var env)))
            (if v-ell
                (if v-var
                    (append (subst-ellipsis
                              var
                              (car form)
                              (if v-var
                                  (cons (cdr v-var) (cdr v-ell))
                                  (cdr v-ell))
                              env)
                            (cddr form))
                    (append (list (tmpl->form #f (car form) env))
                            (cdr v-ell)
                            (cddr form)))
                (wrong "unmatched ... in syntax-rules")))))
      ((pair? form)
        (cons (tmpl->form (if (pair? pattern)
                              (car pattern)
                              #f)
                          (car form)
                          env)
              (tmpl->form (if (pair? pattern)
                              (cdr pattern)
                              #f)
                          (cdr form)
                          env)))
      (else form))))
```

Syntax-transform the given form. Pass the resulting form back to **expand-all**, so that recursive applications of syntax transformers will be expanded as well.

```
     (transform
       (lambda (form)
         (let ((syn (syntax->list (car form))))
           (if (not syn)
               (wrong "not a syntax transformer" (car form))
               (let ((name (car form))
                     (literals (car syn))
                     (rules (cdr syn)))
                 (let ((pat/tmpl/env (find-rule form rules name literals)))
                   (expand-all
                     (apply tmpl->form pat/tmpl/env))))))))
```

Expand all applications of syntax transformers in the given form.

```
     (expand-all
       (lambda (form)
         (cond
           ((not (pair? form)) form)
           ((eq? (car form) 'quote) form)
           ((syntax->list (car form))
             (transform form))
           (else (map-improper expand-all form))))))

    (expand-all form)))
```

Quasiquote Expander

The **`expand-quasiquote`** procedure rewrites quasiquote templates to applications of **`list`**, **`quote`**, and **`append`**. Quasiquoted atoms are transformed to quoted objects. When a quasiquoted list passed to **`expand-quasiquote`** does not contain any applications of **`unquote-splicing`**, it is converted to an application of **`list`** where only **`unquote`**d object are not quoted:

```
`(a b ,c)  ===>  (list 'a 'b c)
```

When a quasiquoted list does contain application of **`unquote-splicing`**, **`list`** is replaced with **`append`** and each member of the list is wrapped up in an application of **`list`**, except for those that are spliced:

```
`(a ,b ,@c)  ===>  (append (list 'a) (list b) c)
```

Nested list are expanded recursively.

```
(define (expand-quasiquote form)
  (letrec
    ((does-splicing?
       (lambda (form)
         (cond
           ((not (pair? form)) #f)
           ((and (pair? (car form))
                 (eq? 'unquote-splicing (caar form)))
             #t)
           (else (does-splicing? (cdr form))))))
     (qq-list
       (lambda (form)
         (if (does-splicing? form)
             (cons 'append
                   (map (lambda (x)
                          (if (and (pair? x)
                                   (eq? 'unquote-splicing (car x)))
                              (cadr x)
                              (list 'list (expand-qq x))))
                        form))
             (cons 'list (map expand-qq form)))))
     (expand-qq
       (lambda (form)
         (cond
           ((vector? form)
             (list 'list->vector (qq-list (vector->list form))))
           ((not (pair? form)) (list 'quote form))
           ((eq? 'quasiquote (car form)) (expand-qq (cadr form)))
           ((eq? 'unquote (car form)) (cadr form))
           (else (qq-list form))))))
    (expand-qq (cadr form))))
```

Library Syntax

Now that syntax expansion is up and running, here are some standard Scheme syntax transformers.

```
(define-syntax case
  (syntax-rules (else)
    ((_ key (else expr . rest))
       (begin expr . rest))
    ((_ key (data expr . rest))
       (if (memv key 'data)
           (begin expr . rest)
           (if #f #f)))
    ((_ key (data1 expr1 . rest1) more-cases ...)
       (if (memv key 'data1)
           (begin expr1 . rest1)
           (case key more-cases ...)))))

(define-syntax let*
  (syntax-rules ()
    ((_ () x . rest)
       (let () x . rest))
    ((_ ((n v)) x . rest)
       (let ((n v)) x . rest))
    ((_ ((n1 v1) (n2 v2) ...) x . rest)
       (let ((n1 v1))
         (let* ((n2 v2) ...) x . rest)))))

(define-syntax delay
  (syntax-rules ()
    ((_ form)
      (let ((%%r #f))
        (lambda ()
          (cond (%%r (car %%r))
            (else (set! %%r (cons form '()))
                  (car %%r))))))))
```

`Force` is not really syntax, but connected to **`delay`** and hence included here.

```
(define (force x) (x))
```

That's it. A rather broad subset of Scheme in about 110 pages of source code. I hope you enjoyed the tour.

List of Figures

Fig. 1 – Garbage collection, State 0 15
Fig. 2 – Garbage collection, State 1 15
Fig. 3 – Garbage collection, State 2 16
Fig. 4 – Garbage collection, finished 16
Fig. 5 – Vector pool structure 17
Fig. 6 – Vector pool, fragmented 20
Fig. 7 – Vector pool, compacted 20
Fig. 8 – Bignum integer structure 25
Fig. 9 – Atomic type structure 26
Fig. 10 – Shared list 36
Fig. 11 – Hashed environment rib 37
Fig. 12 – Association structure 40
Fig. 13 – Procedure structure 47
Fig. 14 – Syntax transformer structure 51
Fig. 15 – Procedure application 90
Fig. 16 – Procedure call context 91
Fig. 17 – Primitive procedure structure 100

Index

#

43
#|...|# 28
* procedure 78
** 101
+ procedure 72
− procedure 71
< procedure 68
<= procedure 68
= procedure 67
> procedure 67
>= procedure 67
~ 104

A

AFLAG 8
ARGS 45
Acc 10
abs procedure 111
absolute value (bignum) 53
add_primitive() 100
add_symbol() 22
addition (bignum) 54
alloc() 20
alloc3() 19
allocv() 21
and form 41,93
append procedure 110
append, destructive 36
appendb() 36
apply procedure 62,93
apply_special() 92
argument list (syntax) 37
argument_list_p() 37
arguments 90
arithmetics 52
assoc procedure 108
assq procedure 108
assv procedure 108
atomp_p() 13
auto quotation 12
auto-quoting_p() 12

B

begin form 41f,47
bignum integer 25
bignum_abs() 53
_bignum_add() 54
bignum_add() 55
_bignum_divide() 60
bignum_divide() 61
bignum_equal_p() 56
bignum_equalize() 60
bignum_less() 55
bignum_multiply() 59
bignum_negate() 53
bignum_negative_p() 53
bignum_shift_left() 58
bignum_shift_right() 58
_bignum_subtract() 56
bignum_subtract() 57
bignum_zero_p() 53
bind_arguments() 90
body (lambda) 47
boolean? procedure 62
boolean_p() 12
box 40

C

Car 9
Cdr 9
caar()...cddddr() 11
caar...cddddr procedures 107
call-by-name flag 93
call-with-input-file procedure 114
call-with-output-file procedure 114
car procedure 63
case form 120
cdr procedure 63
char->integer procedure 63
char-alphabetic? procedure 63
char-ci...? procedures 64
char-downcase procedure 64

char-lower-case? procedure 64
char-numeric? procedure 65
char-upcase procedure 65
char-upper-case? procedure 65
char-whitespace? procedure 65
char...? procedures 64
char? procedure 63
char_ci_lt()... 63
char_lt()... 63
char_p() 12
char_predicate() 63
char_value() 11
character() 26
clear_local_envs() 99
close-input-port procedure 65
close-output-port procedure 65
close_port() 65
comment 28
compaction 20
compilation flags 7
cond form 42
cons procedure 66
continue flag 41,93
current-input-port procedure 66
current-output-port procedure 66
D
DIGITS_PER_WORD 8
DOT 9
Displaying 10,66
define form (local) 45
define form 50
define-syntax form 51
delay form 120
display procedure 66
division (bignum) 60
dynamic scoping 92
E
ENDOFFILE 9
EVAL_STATES enumeration 8
Environment 10
Error_flag 10,13
ellipsis 115
environment (macro) 116
environment rib 37,40,90
eof-object? procedure 66
eof_p() 11
eq? procedure 66
equal? procedure 107
equivalence (bignum) 56
eqv? procedure 107
error() 13
escape sequence 27
_eval() 92
eval() 99
evaluator state 8,41,90,93
even? procedure 111
expand-quasiquote procedure 119
expand-syntax procedure 115
expand_all_syntax() 89
expand_qq() 88
expand_quasiquote() 88
expand_syntax() 89
expected() 84
expt procedure 111
extend() 38
external representation 22
extract_from_defines() 45
extract_from_let() 45
F
FALSE 9
Free_list 10
Free_vecs 10
fatal() 13
find_local_variable() 49
find_symbol() 22
flat_copy() 36
fold-left procedure 109
fold-right procedure 109
for-each procedure 111
force procedure 120
fragmentation 20
G
GC_root 10,18
garbage collector (vectors) 20
garbage collector flags 8
garbage collector 14

gc() 18
gcd procedure 112
gcv() 20
global variables 9
H
HASH_THRESHOLD 7,37
has_property_p() 86
hash function 38
hashed environment 37
I
I/O port 65f,69
INT_SEG_LIMIT 8,52
Input_port 10
if form 42
init() 103
initialization 103
input-port? procedure 67
input_port_p() 12
integer segment 8,52
integer->char procedure 67
integer? procedure 68
integer_p() 12
integer_value() 52
internal representation 22
K
keyboard_interrupt() 99
keyboard_quit() 99
keyword 51
L
LFLAG 8
LSS 25
Level 10
Load_level 10
lambda form 44,47f
lcm procedure 112
least significant segment 25
length procedure 110
length() 35
less-than (bignum) 55
let form 45,47
let* form 120
letrec form 40,43f,48
lexical environment 47
library 104,107
link field (vectors) 17
list procedure 110
list->string procedure 68
list->vector procedure 69
list-ref procedure 110
list-tail procedure 110
list? procedure 107
list_of_symbols() 37
list_to_vector() 29
load procedure 70,93
load() 69
load_library() 104
load_rc() 104
local environment 38,47,90
location_of() 40
longest match 116
lookup() 39
M
MARGS 8,93
MATOM 8
MAX_PORTS 7
MBEGN 8
MBETA 8,91
MCOND 8,42
MCONJ 8,41
MDEFN 8,50
MDISJ 8
MDSYN 8,51
MEMORY_LIMIT_KN 8
MFLAG 8,15
MIFPR 8,42
MSETV 8,50
MSS 25
main() 105
make-string procedure 70
make-vector procedure 70
make_application() 86
make_assignments() 43
make_char() 26
make_dynamic() 92
make_env() 38
make_initial_env() 101

make_integer() 52
make_port() 66
make_primitive() 100
make_recursive_lambda() 44
make_sequence() 41
make_string() 26
make_symbol() 22
make_temporaries() 43
make_undefineds() 44
map procedure 111
map-car procedure 108
mark and sweep 14
mark flag 15
mark() 15
mark_vector() 15
max procedure 112
member procedure 111
memq procedure 111
memv procedure 111
min procedure 112
modulo procedure 112
most significant segment 25
multiplication (bignum) 59

N

NIL 9
NOEXPR 9
NO_SIGNALS 7
negative value (bignum) 53
negative? procedure 113
nested_comment() 28
new_segment() 14
newline procedure 114
nl() 11
node macros 11
node pool 9
node structure 9
not procedure 107
null? procedure 107
null_p() 11
number->string procedure 113
number? procedure 107

O

Output_port 10
odd? procedure 113
open-input-file procedure 71
open-output-file procedure 71
open_port() 69
or form 41,49,93
output-port? procedure 71
output_port_p() 12

P

PRIMITIVES enumeration 81
Pool_size 9
Port_flags 10
Ports 10
Primitive_procedure structure 81
Primitives 82
Program 10
pair? procedure 72
pair_() 13
pattern 51,116
pc variable 41
peek-char procedure 73
pointer reversal 15
port? procedure 107
port_no() 11
positive? procedure 113
pp_apply() 62
pp_boolean_p() 62
pp_car() 63
pp_cdr() 63
pp_char_alphabetic_p() 63
pp_char_ci_eq_p() 64
pp_char_ci_ge_p() 64
pp_char_ci_gt_p() 64
pp_char_ci_le_p() 64
pp_char_ci_lt_p() 64
pp_char_downcase() 64
pp_char_eq_p() 64
pp_char_ge_p() 64
pp_char_gt_p() 64
pp_char_le_p() 64
pp_char_lower_case_p() 64
pp_char_lt_p() 64
pp_char_numeric_p() 65
pp_char_p() 63

pp_char_to_integer() 63
pp_char_upcase() 65
pp_char_upper_case_p() 65
pp_char_whitespace_p() 65
pp_close_input_port() 65
pp_close_output_port() 65
pp_cons() 66
pp_current_input_port() 66
pp_current_output_port() 66
pp_display() 66
pp_eof_object_p() 66
pp_eq_p() 66
pp_equal() 67
pp_greater() 67
pp_greater_equal() 67
pp_input_port_p() 67
pp_integer_p() 68
pp_integer_to_char() 67
pp_less() 68
pp_less_equal() 68
pp_list_to_string() 68
pp_list_to_vector() 69
pp_load() 70
pp_make_string() 70
pp_make_vector() 70
pp_minus() 71
pp_open_input_file() 71
pp_open_output_file() 71
pp_output_port_p() 71
pp_pair_p() 72
pp_peek_char() 73
pp_plus() 72
pp_procedure_p() 72
pp_quotient() 72
pp_read() 72
pp_read_char() 73
pp_remainder() 73
pp_set_car_b() 73
pp_set_cdr_b() 73
pp_set_input-port_b() 74
pp_set_output-port_b() 74
pp_string_append() 74
pp_string_copy() 75
pp_string_fill_b() 75
pp_string_length() 76
pp_string_p() 78
pp_string_ref() 76
pp_string_set_b() 76
pp_string_to_list() 74
pp_string_to_symbol() 74
pp_substring() 75
pp_symbol_p() 78
pp_symbol_to_string() 78
pp_syntax_to_list() 78
pp_times() 78
pp_unquote() 79
pp_unquote_splicing() 79
pp_vector_fill_b() 79
pp_vector_length() 80
pp_vector_p() 80
pp_vector_ref() 80
pp_vector_set_b() 80
pp_vector_to_list() 79
pp_write() 80
pp_write_char() 81
pp_wrong() 81
pr() 13
predicate 13
primitive procedure 62,81,100
primitive() 84
primitive_p() 12
print() 34
print_char() 32
print_integer() 31
print_port() 34
print_primitive() 33
print_procedure() 32
print_quoted() 32
print_string() 33
print_symbol() 33
print_syntax() 34
print_vector() 34
printer 31
procedure application 62,84
procedure definition 50
procedure? procedure 72

procedure_p() 12
ps variable 41
Q
Quiet_mode 10
quasiquotation 87,119
quasiquotation_p() 87
quasiquote form 119
quotation 24,49,79,87,119
quote form 49,119
quote() 24
quotient procedure 72
R
REPL 99
RPAREN 9
read eval print loop 99
read procedure 22,29,72
read_vector() 29
read-char procedure 73
read_c() 23
read_c_ci() 23
read_char() 73
read_form() 29
read_list() 23
reader 22
recursive interpretation 87
rehash() 37
reject() 11
remainder procedure 73
repl() 99
resolve_local_defines() 46
restore_state() 90
reverse procedure 110
reverse_segments() 53
rib (environment) 37
rib structure 13,93
rib_append 13
rib_args 13
rib_result 13
rib_source 13
S
S9FES_LIBRARY_PATH 104
SEGMENT_LEN 7
SFLAG 8,15
SIGINT 7,99
SIGQUIT 7
S_and...S_syntax-rules 10
S_char...S_vector 10
Stack 10
Stack_bottom 10
Stats_stack 10
Symbols 10
save() 22
save_state() 90
segment (memory) 14,19
separator() 28
set! form 40,43f,48f
set-car! procedure 73
set-cdr! procedure 73
set-input-port! procedure 74
set-output-port! procedure 74
sf_and() 42
sf_begin() 42
sf_cond() 42
sf_define() 50
sf_define_syntax() 51
sf_if() 42
sf_lambda() 47
sf_let() 47
sf_letrec() 48
sf_quote() 49
sf_set_b() 49
sf_syntax_rules() 51
shift left (bignum) 58
shift right (bignum) 58
size field (vectors) 17
size of a node 9
special form handler 41
special form 41,92
special value 9
special_p() 12
spine (list) 36
sqrt procedure 113
stack bottom 93
str_numeric_p() 24
strcmp_ci() 76
string procedure 113

string() 11
string->list procedure 74
string->number procedure 114
string->symbol procedure 74
string-append procedure 74
string-ci<=? procedure 77
string-ci<? procedure 77
string-ci=? procedure 77
string-ci>=? procedure 77
string-ci>? procedure 77
string-copy procedure 75
string-fill! procedure 75
string-length procedure 76
string-ref procedure 76
string-set! procedure 76
string<=? procedure 77
string<? procedure 77
string=? procedure 77
string>=? procedure 77
string>? procedure 77
string? procedure 78
string_ci_le_p()... 77
string_le_p()... 77
string_len() 11
string_literal() 27
string_p() 12
string_predicate() 76
substring procedure 75
subtraction (bignum) 56
symbol 28
symbol->string procedure 78
symbol? procedure 78
symbol_or_number() 28
symbol_p() 12
syntactic_symbol_p() 87
syntax expansion 89
syntax rules 116
syntax transformation 115
syntax transformer 40,51
T
TEXT_LEN 7
TRUE 9
TYPES enumeration 81
Tag 9
Tmp_car 10,19
Tmp_cdr 10,19
syntax->list procedure 78
syntax-rules form 51
syntax_p() 12
tag field 8
tail call elimination 91
tail_call() 91
template 51,117
token separators 28
token 28
too_few_args() 41
too_many_args() 41
tree walking 92
try_hash() 39
type error 84
typing 81
U
UFLAG 8
UNDEFINED 9
UNSPECIFIC 9
undefined_p() 11
unmark_vectors() 17
unquote form 79,119
unquote-splicing form 79,119
unreadable form 27
unsave() 22
unspecific_p() 11
usage() 104
uses_quasiquotation_p() 87
uses_transformer_p() 87
V
VARS 45
VFLAG 8
Vector 10
Vpool_size 9
value_of() 40
variable 40
variables (lambda) 47
variadic procedure 63,77
vector pool compaction 20
vector pool 9

vector procedure 114
vector() 11
vector->list procedure 79
vector-fill! procedure 79
vector-length procedure 80
vector-ref procedure 80
vector-set! procedure 80
vector? procedure 80
vector_len() 11
vector_p() 12
vector_size() 11
with-input-from-file procedure 115
with-output-to-file procedure 115
write procedure 31,80
write-char procedure 81
wrong procedure 81
X
xread() 31
Z
zero? procedure 113