Nils M Holm

Scheme 9 from Empty Space

Contents

The Base System

Real Number Extension

Syntax Transformation

Appendix

What's New in this Edition?

First of all, the third edition reflects the current version of the S9fES code (2014-07-21). In the last years, lots of small changes have been incorporated into the code base, including their descriptions in prose. These were mostly small things, the basic structure of the interpreter has not changed much in the past four years. The only major modification was an adaption of the image file format to systems that use address space layout randomization (ASLR) or position independent code (PIC).

Then there were a few typos, markup glitches, and sentences that could use clarification. So the new edition is mostly a cleaned-up, slightly extended, and updated version of the 2nd edition.

The only thing that is new is a very short chapter demonstrating that `syntax-rules` actually can count by bending the rules a bit. The previous edition claimed that this was impossible (which it *is* from a practical point of view), but I thought it might be fun to include a counter-proof in the text.

All in all, I think that the 3rd edition is a better book than its predecessors. Mostly because the prose has been streamlined once again, so I think it is an even easier read than the previous versions. The topic is hard enough, so at least the prose should be fun and easy to comprehend.

I hope you will enjoy the new edition!

Nils M Holm, June 2014

A Guided Tour through the Code

Scheme 9 from Empty Space (S9fES) is an interpreter for R4RS Scheme. It is itself written in ANSI C (C89) and Scheme. The S9fES code strives to be clear and comprehensible. It is portable without being inefficient or incomplete. If you want to study the implementation of a real-world Scheme system, including the nitty gritty bits, such as continuations, macros, tail-call elimination, bignum arithmetics, floating point arithmetics, constant-space garbage collection with compaction, etc., this text should be a good starting point.

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 files, so some forward references will occur. Expect to read the text 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 *entire* code of the interpreter, there will be advanced parts and easy parts. Feel free to skip parts that you consider to be trivial. Parts that are complex can often be identified by the 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 T3X.ORG [http://t3x.org] (if it still exists!) and study the actual code and the book at the same time.

If you are reading a colored copy of this text, the following colors are used to highlight individual parts of the code sections. The following is a sample Scheme code section:

```
; Sample                What it is
; -----------------     ---------------------
  #|explanation|#     a comment
  (())                  parentheses
  foo-frobnicate        a user-defined symbol
  "hello, world!"       a data literal
  string-ci=?           a Scheme procedure
  define                Scheme syntax
  reverse!              a S9fES extension
```

And here comes a sample C section:

```
/* Sample                What it is                */
/* -----------------     ------------------------- */
   /* explanation */     a comment
   {};                   punctuation
   foo_frobnicate        a user-defined symbol
   "hello, world!"       a data literal
   fopen                 an ANSI C library function
   void                  C syntax
   __nuxi__              a language extension/macro
```

However, if you are watching this show in black and white, that's fine, too. The colors are not really necessary to understand what's going on!

In any case the text uses different fonts for different types of code. In annotations,

```
Scheme code and C code
```

are rendered in a typewriter font and may be placed in boxes or enclosed by vertical brackets. *Parameters* and procedure *arguments* (variables) print in italics. 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 rendered in 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.

In case you wonder why some parts of box diagrams have a gray background: unless the related prose indicates otherwise, this means that the corresponding object is an atom.

Feel free to use, modify and redistribute the Scheme 9 from Empty Space code, it is in the public domain (which means: do whatever you please with it).

And now: *enjoy the tour!*

Nils M Holm, October 2010

Declarations (Header)

```
/*
 * Scheme 9 from Empty Space
 * By Nils M Holm, 2007-2014
 * Placed in the Public Domain
 */
```

Prelude

This section uses nested `#ifdef` instead of `#if` in order to support the Plan 9 C language [http://plan9.bell-labs.com/sys/doc/compiler.html], which intentionally omits `#if`.

Because we want to support both Un*x and Plan 9, we will have to figure out where we are compiling.

```
/*
 * Ugly prelude to figure out if
 * we are compiling on a Un*x system.
 */

#ifdef __NetBSD__
 #ifndef unix
  #define unix
 #endif
#endif

#ifdef __unix
 #ifndef unix
  #define unix
 #endif
#endif

#ifdef __linux
 #ifndef unix
  #define unix
 #endif
#endif

#ifndef unix
 #ifndef plan9
  #error "Either 'unix' or 'plan9' must be #defined."
 #endif
#endif
```

Next, we set up some OS-dependent stuff, mostly to make the OS-specific procedures in the Un*x extension work on Linux. See your favorite Un*x/POSIX/Plan 9 programming literature for details.

```
#ifdef unix
 #ifndef _BSD_SOURCE
  #define _BSD_SOURCE
 #endif
 #ifndef __FreeBSD__
  #ifndef __NetBSD__
```

```
   #ifndef _POSIX_SOURCE
    #define _POSIX_SOURCE
    #define _POSIX_C_SOURCE 200112L
   #endif
   #ifndef _XOPEN_SOURCE
    #define _XOPEN_SOURCE 500
   #endif
  #endif
 #endif
#endif

#ifdef plan9
 #include <u.h>
 #include <libc.h>
 #include <stdio.h>
 #include <ctype.h>
 #define NO_SIGNALS
 #define signal(sig, fn)
 #define exit(x) exits((x)? "error": NULL)
 #define ptrdiff_t int
#endif

#ifdef unix
 #include <stdlib.h>
 #include <stddef.h>
 #include <stdio.h>
 #include <string.h>
 #include <ctype.h>
 #ifdef NO_SIGNALS
  #define signal(sig, fn)
 #else
  #include <signal.h>
  #ifndef SIGQUIT
   /* MinGW does not define SIGQUIT */
   #define SIGQUIT SIGINT
  #endif
 #endif
#endif
```

The company who brought you the unspeakable horror has to do its own thing once again, so

```
/*
 * Tell later MSC compilers to let us use the standard CLIB API.
 * Blake McBride < b l a k e @ m c b r i d e . n a m e >
 */

#ifdef _MSC_VER
 #if _MSC_VER > 1200
  #ifndef _CRT_SECURE_NO_DEPRECATE
   #define _CRT_SECURE_NO_DEPRECATE
  #endif
 #endif
 #ifndef _POSIX_
  #define _POSIX_
 #endif
#endif
```

Configuration

DEFAULT_LIBRARY_PATH will be searched by the interpreter for heap images, library packages, help files, etc. It can be overridden with a command line flag.

```
#ifndef DEFAULT_LIBRARY_PATH
 #define DEFAULT_LIBRARY_PATH \
                "."                             \
                ":lib"                          \
                ":ext"                          \
                ":contrib"                      \
                ":~/.s9fes"                     \
                ":/usr/local/share/s9fes"
#endif
```

INITIAL_SEGMENT_SIZE is the initial number of "nodes" in the cons pool and "cells" in the vector pool. The terms will be explained in the following.

TOKEN_LENGTH is the maximum length of various objects, such as string and character literals, symbols, path names, etc. MAX_PORTS is the maximum number of input and output ports that can be kept open simultaneously. MAX_IO_DEPTH is the maximum *depth* (number of nested objects) of a list or vector to read or print. MAX_CALL_TRACE is the maximum number of recently called procedures to remember for error reporting.

HASH_THRESHOLD is the size of the smallest lexical environment that will be hashed. This value is rather arbitrary. It has been determined through experimentation.

```
#ifndef INITIAL_SEGMENT_SIZE
 #define INITIAL_SEGMENT_SIZE   32768
#endif

#define TOKEN_LENGTH            1024
#define MAX_PORTS               32
#define MAX_IO_DEPTH            65536   /* Reduce on 16-bit systems! */
#define HASH_THRESHOLD          5
#define MAX_CALL_TRACE          100
```

The DEFAULT_LIMIT_KN constant specifies the maximum number of nodes that will *ever* be allocated by the interpreter. The rather peculiar default value is based on the way in which segment sizes grow. Each time a new memory segment is added, the segment size grows by a factor of 3/2, so the allocation sequence reaches this pool size when growing the pool for the 12th time.

```
/* Default memory limit in K-nodes, 0 = none */
#define DEFAULT_LIMIT_KN        12392
```

A cell is an atomic storage location that is large enough to hold a C pointer. It will be used to store conses, bignum segments, and other interesting stuff. The ptrdiff_t type appears to be a natural choice in ANSI C environments.

```
/* A "cell" must be large enough to hold a pointer */
#define cell    ptrdiff_t
```

Really don't use the following!

```
/* 64-bit emulation on 32-bit system; DO NOT USE! */
#ifdef EMULATE_64
```

```
 #undef BITS_PER_WORD_32
 #define BITS_PER_WORD_64
 #undef cell
 #define cell           long long
 #define labs(x)        llabs(x)
 #define atol(x)        atoll(x)
#endif
```

Always use BITS_PER_WORD_32 on 32-bit platforms. BITS_PER_WORD_64 improves numeric performance on 64-bit systems. The BITS_PER_WORD_16 option is mostly a lie. While it theoretically is possible to compile S9fES on 16-bit systems, it will not allow you to run a lot of interesting programs.

```
/* Pick one ... */
/* #define BITS_PER_WORD_64 */
/* #define BITS_PER_WORD_32 */
/* #define BITS_PER_WORD_16 */

/* ... or assume a reasonable default */
#ifndef BITS_PER_WORD_16
 #ifndef BITS_PER_WORD_32
  #ifndef BITS_PER_WORD_64
   #define BITS_PER_WORD_32
  #endif
 #endif
#endif
```

An "integer segment" is an atomic computational unit that is used in bignum arithmetics. This will be explained in detail in the section dealing with bignums. The larger an integer segment is, the faster the numeric operations will be. INT_SEG_LIMIT specifies the smallest value that *cannot* be represented by an integer segment. DIGITS_PER_WORD specifies the number of decimal digits per segment. Note that integer arithmetics is basically base-10^n internally, where *n* depends on the segment size. MANTISSA_SEGMENTS is the number of integer segments used to represent the mantissa of a real number. MANTISSA_SIZE is the number of digits in a mantissa.

```
/*
 * N-bit arithmetics require sizeof(cell) >= N/8.
 * When MANTISSA_SIZE (below) gets more than 60 places, you
 * will have to supply a better value for PI in "s9-real.scm".
 */

#ifdef BITS_PER_WORD_64
 #define DIGITS_PER_WORD        18
 #ifdef EMULATE_64
  #define INT_SEG_LIMIT         1000000000000000000LL
 #else
  #define INT_SEG_LIMIT         1000000000000000000L
 #endif
 #define MANTISSA_SEGMENTS      1
#else
 #ifdef BITS_PER_WORD_32
  #define DIGITS_PER_WORD       9
  #define INT_SEG_LIMIT         1000000000L
  #define MANTISSA_SEGMENTS     2
 #else
  #ifdef BITS_PER_WORD_16
   #define DIGITS_PER_WORD      4
   #define INT_SEG_LIMIT        10000
```

```
   #define MANTISSA_SEGMENTS    3
  #else
   #error "BITS_PER_WORD_* undefined (this should not happen)"
  #endif
 #endif
#endif

/* Mantissa sizes differ among systems */
#define MANTISSA_SIZE   (MANTISSA_SEGMENTS * DIGITS_PER_WORD)
```

Internals

Most of the _TAG constants control the garbage collector (GC). ATOM_TAG marks a node (see below) "atomic" in the sense of Scheme: it is not a cons cell, but an object that cannot be decomposed at Scheme-level. From the GC's point of view, this means that its "car" field is *not* a reference to another node and hence must not be followed. A node with the ATOM_TAG set is called an "atom".

The MARK_TAG marks a node "used" during GC. The STATE_TAG together with MARK_TAG is used to store the two-bit state of a node during garbage collection (also GC).

VECTOR_TAG marks a vector object (vector or string), which references space in the separate vector pool. PORT_TAG indicates an I/O port, which may be closed by the GC when it is no longer used. USED_TAG marks an I/O port as "used" -- this may be thought of as a "marked" tag for ports. When an I/O port has the LOCK_TAG set, it may not be closed, even if it is not used.

Note: USED_TAG and LOCK_TAG are stored in Port_flags[] rather than in Tag.

CONST_TAG is the only tag that is not used by the GC. It indicates that the corresponding object is immutable. It is set, for instance, in string literals, vector literals, and *all* nodes of quoted lists.

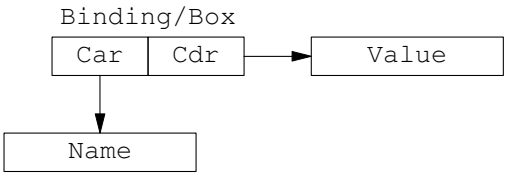
```
/*
 * Node tags
 */

#define ATOM_TAG        0x01    /* Atom, Car = type, CDR = next */
#define MARK_TAG        0x02    /* Mark */
#define STATE_TAG       0x04    /* State */
#define VECTOR_TAG      0x08    /* Vector, Car = type, CDR = content */
#define PORT_TAG        0x10    /* Atom is an I/O port (with ATOM_TAG) */
#define USED_TAG        0x20    /* Port: used flag */
#define LOCK_TAG        0x40    /* Port: locked (do not close) */
#define CONST_TAG       0x80    /* Node is immutable */
```

EVAL_STATES is an enumeration of the states that the evaluator may cycle through while reducing an expression to its normal form. It will be explained in detail in the evaluator section.

```
/*
 * Evaluator states
 */

enum EVAL_STATES {
        EV_ATOM,        /* Evaluating atom */
        EV_ARGS,        /* Evaluating argument list */
        EV_BETA,        /* Evaluating procedure body */
        EV_IF_PRED,     /* Evaluating predicate of IF */
```

```
        EV_SET_VAL,     /* Evaluating value of SET! and DEFINE */
        EV_MACRO,       /* Evaluating value of DEFINE-SYNTAX */
        EV_BEGIN,       /* Evaluating expressions of BEGIN */
        EV_AND,         /* Evaluating arguments of AND */
        EV_OR,          /* Evaluating arguments of OR */
        EV_COND         /* Evaluating clauses of COND */
};
```

A "binding" is a structure of the form shown in fig.1.



[Fig.1: Binding structure]

The car part of the binding is a symbol naming the variable implemented by the binding. The cdr part contains the value of the variable.

Note that the Scheme storage model says that a variable is created by binding a symbol to a *storage location* rather than a value. This actually holds for the above, because we use the binding itself as a storage "box" and the `set-cdr!` operation to alter the value inside of that box.

The `binding_box()` and `binding_value()` macros access the storage location of a binding and the value stored in it. `box_value()` is used to access the value when given a box. It is the same as `binding_value()` due to the above optimization. `make_binding()` creates a new binding.

`Car` is used to access the name of a binding, because we routinely use `caar` and friends to process environments (lists of bindings), which breaks any abstraction of the name field anyway.

```
/*
 * Binding structure
 */

#define make_binding(v, a)      (cons((v), (a)))
#define binding_box(x)          (x)
#define binding_value(x)        (cdr(x))
#define box_value(x)            (cdr(x))
```

"Special objects" are represented by negative values. Although they are atomic, they are no atoms (you may think of them as "virtual particles"). Atoms are nodes and nodes are identified by a (positive) offset into the node pool. So their *negative* value is sufficient to mark special objects as such.

`NIL` is `()`, `TRUE` is `#t`, `FALSE` is `#f`, `END_OF_FILE` is the "EOF object" (as identified by the `eof-object?` procedure), `UNDEFINED` is an undefined value (used internally), `UNSPECIFIC` is an unspecific value as returned by `set!`, etc, `NAN` is an invalid numeric result ("not a number"), `DOT` marks the dot of a dotted list internally in the reader, `RPAREN` marks a closing parenthesis and `RBRACK` a closing bracket. `NOEXPR` denotes an unspecific source when reporting an error.

```
/*
 * Special objects
 */
```

```
#define special_value_p(x)      ((x) < 0)
#define NIL                     (-1)
#define TRUE                    (-2)
#define FALSE                   (-3)
#define END_OF_FILE             (-4)
#define UNDEFINED               (-5)
#define UNSPECIFIC              (-6)
#define NAN                     (-7)
#define DOT                     (-8)
#define RPAREN                  (-9)
#define RBRACK                  (-10)
#define NOEXPR                  (-11)
```

Types are also special objects. They appear in the car parts of atoms to indicate the types of Scheme objects. Each of the following constants resembles the corresponding Scheme data type, except for those:

`T_NONE` indicates "no specific type" in type checking. `T_PAIR_OR_NIL` matches either a pair or `()`.

`T_PRIMITIVE` is used to mark primitive procedures and `T_CONTINUATION` marks continuations internally. To the type checker, these constants are synonyms of `T_PROCEDURE`.

`T_SYNTAX` marks macros.

```
/*
 * Types
 */

#define T_NONE                  (-20)
#define T_BOOLEAN               (-21)
#define T_CHAR                  (-22)
#define T_INPUT_PORT            (-23)
#define T_INTEGER               (-24)
#define T_OUTPUT_PORT           (-25)
#define T_PAIR                  (-26)
#define T_PAIR_OR_NIL           (-27)
#define T_PRIMITIVE             (-28)
#define T_PROCEDURE             (-29)
#define T_REAL                  (-30)
#define T_STRING                (-31)
#define T_SYMBOL                (-32)
#define T_SYNTAX                (-33)
#define T_VECTOR                (-34)
#define T_CONTINUATION          (-35)

/*
 * Short cuts for primitive procedure definitions
 * Yes, ___ violates the C standard, but it's too tempting
 */

#define BOL T_BOOLEAN
#define CHR T_CHAR
#define INP T_INPUT_PORT
#define INT T_INTEGER
#define LST T_PAIR_OR_NIL
#define OUP T_OUTPUT_PORT
#define PAI T_PAIR
```

```
#define PRC T_PROCEDURE
#define REA T_REAL
#define STR T_STRING
#define SYM T_SYMBOL
#define VEC T_VECTOR
#define ___ T_NONE
```

The `Primitive_procedure` structure holds information about a primitive procedure, i.e. a Scheme procedure that is implemented in C. `name` holds the name of the procedure, (e.g. "cdr"), `handler` is a pointer to a C function that maps a tree of nodes to a tree of nodes, `min_args` and `max_args` contain the minimum and maximum number of arguments that may be passed to a procedure. `max_args`=-1 indicates that a procedure has no upper argument limit; the procedure is "variadic". The `arg_types[]` array holds the types (see above) of the first three arguments of the primitive procedure. Types of subsequent arguments must be checked by the procedure itself.

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

#define PRIM    struct Primitive_procedure
```

`PRIM_SEG_SIZE` is the number of primitive procedure slots to be allocated at once when adding primitives. This number should be sufficient to hold all core primitives, to avoid unnecessary re-allocation of the `Primitives` array during initialization.

```
#define PRIM_SEG_SIZE   256
```

Global Variables

These are the global variables used by the interpreter. `EXTERN` is pre-defined to expand to nothing when included by the main program. When included by extensions (which must not define `EXTERN`), the below `#define` will make the globals "extern".

```
/*
 * Globals
 */

#ifndef EXTERN
 #define EXTERN extern
#endif
```

`Cons_segment_size` and `Vec_segment_size` specify the sizes of the segments that have just been added to the corresponding pools. Each time a new segment is added, these values grow by a factor of 3/2. `Cons_pool_size` and `Vec_pool_size` hold the current total sizes of the pools.

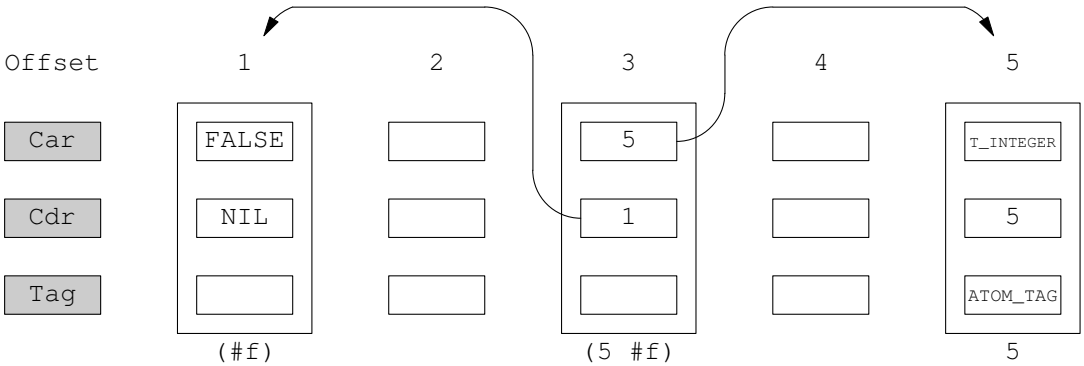
```
EXTERN int      Cons_segment_size,
                Vec_segment_size;
EXTERN int      Cons_pool_size,
                Vec_pool_size;
```

The next three variables implement both the node pool and the node data structure. Each node has

three fields named "car", "cdr", and "tag". When a node implements a "cons cell" (or just "cons"), car and cdr point to the car and cdr part of that cons cell, respectively.

The tag field holds various meta information about the node, which is mostly connected to typing and garbage collection. Atomic nodes (integers, strings, vectors, procedures, etc) have the `ATOM_TAG` bit set in the tag field. Conses have the `ATOM_TAG` cleared.

Each node is identified by its offset into the `Car`, `Cdr`, and `Tag` arrays, which together implement the "cons pool" (outlined in figure 2).



[Fig.2: Node pool structure]

A node references another node by keeping the offset of that node in its car or cdr field. In the above diagram, node 3 references node 1 via its cdr field and node 5 via its car field. Node 1 is `(#f)`, node Node 5 is the integer `5`, so node 3 is `(5 #f)`. Only node 5 has the atom tag set, because an integer is an atom. (`#F` is a special value.)

```
EXTERN cell     *Car,
                *Cdr;
EXTERN char     *Tag;
```

`Vectors` is the vector pool. Strings and vectors will be allocated here.

```
EXTERN cell     *Vectors;
```

`Free_list` is the offset of the node on the top of the list of free nodes. Free nodes are chained together via their cdr fields. `Free_vecs` points to the free region that is maintained at the end of the vector pool.

```
EXTERN cell     Free_list;
EXTERN cell     Free_vecs;
```

`Primitives` is an array holding one `PRIM` struct for each primitive procedure in the entire system. When adding primitives to the system, `PRIM` structs will be copied from local arrays to this array, thereby assigning a unique index to each primitive procedure. This method allows to identify primitives by their offset in the array.

Rationale: This approach was chosen, because storing pointers to `PRIM` structures directly in the primitive procedure nodes caused trouble on computers using address space layout randomization (ASLR) or position-independent code (PIC, PIE) when primitive nodes were written to an image file. When the image file was read back into memory by a different instance of the interpreter, the addresses of the structs could have changed, causing the interpreter to crash.

In earlier versions of S9, this was prevented by making the address of the `Primitives` array part of the magic header (causing the interpreter to fail at image load time), but the new approach is more user-friendly.

```
EXTERN PRIM     *Primitives;
```

`Last_prim` is the slot number (index) of the next primitive procedure to be added to `Primitives`. Hence it is also equal to the number of primitives currently in the system.

`Max_prims` is the current size (number of slots) of `Primitives`. It will grow on demand.

```
EXTERN int      Last_prim, Max_prims;
```

The following code contains the various registers of the virtual Scheme machine. `Stack` is the global stack for storing intermediate results, call frames, outer environments, etc. Because the evaluator is reentrant, the bottom of the stack is memorized in `Stack_bottom`. `State_stack` is used to save the interpreter state when evaluating nested expressions.

`Tmp`, `Tmp_car`, and `Tmp_cdr` protect objects from the garbage collector while allocating nodes.

`Symbols` is a list of all symbols that are known to the interpreter. `Program` is the program that is currently evaluating. `Environment` is the environment that is currently in effect. `Acc` is the accumulator -- a register that holds the result of a (partial) evaluation.

`Apply_magic` and `Callcc_magic` hold the indices of the `apply` and `call/cc` procedures in the `Primitives` array. This is done because these procedures will have to be treated in a special way by the evaluator.

`New` is part of a hack to work around the sub-optimal order of assignment evaluation of C. Because `car(x)` may be evaluated *before or after* `cons(a,b)` in

```
car(x) = cons(a,b);
```

the allocator (`cons()`) may relocate the pool, so `car(x)` will write to an undefined memory region. This is why we have to do it this way:

```
New = cons(a,b);
car(x) = New;    /**sigh**/
```

```
EXTERN cell     Stack,
                Stack_bottom;
EXTERN cell     State_stack;
EXTERN cell     Tmp_car,
                Tmp_cdr,
                Tmp;
EXTERN cell     Symbols;
EXTERN cell     Program;
EXTERN cell     Environment;
EXTERN cell     Acc;
EXTERN cell     Apply_magic, Callcc_magic;
EXTERN cell     New;
```

`Level` is the number of nested parentheses during `read`, `Load_level` the number of nested `load`s, and `Displaying` indicates whether we are `display`ing or `write`ing an object. The code of `display` and `write` is almost identical, so we use this flag to run the same function in different modes.

```
EXTERN int      Level;
EXTERN int      Load_level;
EXTERN int      Displaying;
```

These variables deal with error reporting. `Called_procedures[]` holds references to the most recently called procedures. It works as a ring buffer. `Proc_ptr` points to the next free slot in the buffer. `Proc_max` is the maximum number of references to store in the buffer (this value may not be larger than `MAX_CALL_TRACE`).

`File_list` is a list of file names referenced by nested `load`.

`Line_no` is the current input line number in the file connected to the current input port. `Opening_line` is the line number on which an outermost opening parenthesis is placed. It is used for error reporting.

`Printer_count` and `Printer_limit` limit the number of characters written in error messages. When `Printer_count` becomes greater than `Printer_limit`, the printer will emit `"..."` and stop printing.

```
EXTERN cell     Called_procedures[MAX_CALL_TRACE];
EXTERN int      Proc_ptr, Proc_max;
EXTERN cell     File_list;
EXTERN int      Line_no;
EXTERN int      Opening_line;
EXTERN int      Printer_count, Printer_limit;
```

`Trace_list` is a list of procedure names (symbols) that are currently being traced (see `trace` ^ [page 133]).

```
EXTERN cell     Trace_list;
```

`Ports[]` is an array holding all I/O ports that may be used by S9fES. Opening a port allocates a slot from this array. Unused ports are set to `NULL`. Each `Port` is associated with a corresponding slot in `Port_flags[]`, which holds the state of the port. (See `USED_TAG`, `LOCK_TAG` ^ [page 19].)

`Input_port` and `Output_port` are the ports that are currently being read and written respectively (they are delivered in Scheme programs by `(current-input-port)` and `(current-output-port)`). `Error_port` is connected to `stderr`. It is used to report errors in batch mode (when `Quiet_mode`=1).

```
EXTERN FILE     *Ports[MAX_PORTS];
EXTERN char     Port_flags[MAX_PORTS];
EXTERN int      Input_port,
                Output_port,
                Error_port;
```

`Command_line` is a copy of the arguments passed to a Scheme program via `argv[]`. `Memory_limit_kn` is the memory limit of the interpreter in kilo nodes.

When `Quiet_mode` is set to one, the interpreter will not print a banner or issue a prompt and it will exit after catching an error. Error messages will go to stderr instead of stdout.

```
EXTERN char     **Command_line;
EXTERN long     Memory_limit_kn;
EXTERN int      Quiet_mode;
```

`Error_flag` will be set as soon as an error is caught by `error()`. Because this flag may be set by the `SIGINT` handler, it has to be declared "volatile".

```
EXTERN volatile int     Error_flag;
```

The evaluator uses the following variables to identify some frequently-used symbols in *O(1)* time.

```
/* Short cuts for accessing predefined symbols */
EXTERN cell     S_arrow, S_else, S_extensions, S_latest,
                S_library_path, S_loading, S_quasiquote,
                S_quote, S_unquote, S_unquote_splicing;
EXTERN cell     S_and, S_begin, S_cond, S_define,
                S_define_syntax, S_if, S_lambda, S_or,
                S_set_b;
```

Macros

Input/Output

These are just some shortcuts. `nl()` emits a newline sequence, `reject()` puts a character back to an input port, `read_c()` reads a character from an input port, and `read_c_ci()` reads a character and transforms it to lower case. *All* interpreter input goes through `read_c()`.

```
/*
 * I/O
 */

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

Node Decomposition

These macros decompose the various atomic node types. Note that atoms are only atomic at the Scheme level. Behind the scenes we have to access their individual (sub-atomic, as it were) parts. These operations involve some ugly, low-level array wrestling, so they are abstracted as macros.

`string()` returns a pointer to the characters of a string atom. `string_len()` returns the number of characters in a string atom (*including* the trailing `NUL` character!). Note that a string may contain NULs at any position, so we extract the length from the vector pool.

`symbol_name()` extracts the name of a symbol and `symbol_len()` the length of the name. These are in fact the same as the corresponding `string_` macros.

`vector()` extracts a pointer to the elements stored in a vector. The elements are stored in an array of `cell`s. `vector_link()` and `vector_index()` extract values of internal fields of a vector (see below). `vector_len()` computes the actual length of a vector as seen by Scheme. `vector_size()` converts the size of a vector in bytes into its size in cells (including meta information).

`port_no()` extracts the port number of a port, i.e, its offset into the `Ports[]` and `Port_flags[]` arrays.

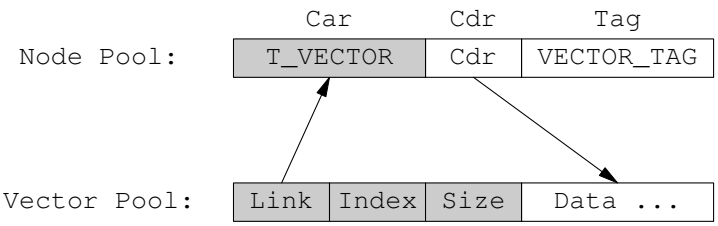
`char_value()` returns the character stored in a Scheme "char".

```
/*
 * Access to fields of atoms
 */

#define string(n)        ((char *) &Vectors[Cdr[n]])
#define string_len(n)    (Vectors[Cdr[n] - 1])
#define symbol_name(n)   (string(n))
#define symbol_len(n)    (string_len(n))
#define vector(n)        (&Vectors[Cdr[n]])
#define vector_link(n)   (Vectors[Cdr[n] - 3])
#define vector_index(n)  (Vectors[Cdr[n] - 2])
#define vector_size(k)   (((k) + sizeof(cell)-1) / sizeof(cell) + 3)
#define vector_len(n)    (vector_size(string_len(n)) - 3)
#define port_no(n)       (cadr(n))
#define char_value(n)    (cadr(n))
```

Vector Representation

A Scheme vector is represented internally as an array of cells with an additional header. The following constants are used to access the header fields.

`RAW_VECTOR_LINK` is the "link field" of a vector. It points back to the node that points to the (data field) of the array in the vector pool (see figure 3).



[Fig.3: Vector back link]

The vector back link is required by the garbage collector. When the vector pool is compacted, vector data may move, so the cdr field of the node representing a vector may have to be updated in order to reflect the new location of the vector data.

`RAW_VECTOR_INDEX` is also used in garbage collection. In the mark phase it stores the offset of the vector slot that is currently being marked. It is required for constant-space marking. (See the Memory Management section [page 38] for details.)

`RAW_VECTOR_SIZE` holds the size of a vector *in bytes*. This is done because strings are also stored in vectors and computing the length of a string is done more frequently than computing the length of a vector.

Note that for a string, the `RAW_VECTOR_SIZE` may be smaller than the actual size of the vector containing the string. This is because vector sizes are always a multiple of the size of a cell, but string lengths are not limited in such a way. The `vector_size()` macro converts a `RAW_VECTOR_SIZE` into an actual vector size in cells.

RAW_VECTOR_DATA, finally, holds the first vector element. Subsequent vector elements follow immediately.

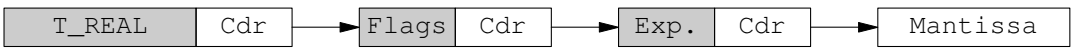
```
/*
 * Internal vector representation
 */

#define RAW_VECTOR_LINK         0
#define RAW_VECTOR_INDEX        1
#define RAW_VECTOR_SIZE         2
#define RAW_VECTOR_DATA         3
```

Real Number Representation

A S9fES real number (floating point number) consists of four or more nodes holding its

- type
- flags
- exponent
- (variable-length) mantissa

All of its nodes have the atom flag set, because their car fields carry values rather than references to other nodes (see figure 4).



[Fig.4: Real number representation]

The mantissa is a *fixed-length* sequence of bignum integer segments, which are explained in detail in the section about bignum arithmetics.

The x_real_flags(), x_real_exponent(), and x_real_mantissa() macros extract the individual parts of a real number. There is only one flag, REAL_NEGATIVE, which -- when set -- indicates that the number is negative. The x_real_negative_flag() macro tests this flag.

```
/*
 * Flags and structure of real numbers
 */

#define x_real_flags(x)          (cadr(x))
#define x_real_exponent(x)       (caddr(x))
#define x_real_mantissa(x)       (cdddr(x))

#define REAL_NEGATIVE   0x01

#define x_real_negative_flag(x)  (x_real_flags(x) & REAL_NEGATIVE)
```

Nested Lists

These are the usual abbreviations for nested car and cdr. We define only those that are actually used in the code.

```
/*
 * Nested lists
 */

#define car(x)          (Car[x])
#define cdr(x)          (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 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 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 cddadr(x)       (Cdr[Cdr[Car[Cdr[x]]]])
#define cdddar(x)       (Cdr[Cdr[Cdr[Car[x]]]])
#define cddddr(x)       (Cdr[Cdr[Cdr[Cdr[x]]]])
```

Type Predicates

Most of the following type predicates test for the corresponding Scheme data types. `eof_p()` tests for the EOF object, `undefined_p()` tests for an undefined value, which is used, for instance, as the value of undefined symbols. `unspecific_p()` tests for the (sic!) unspecific value (as returned by `set!` and friends).

`constant_p()` checks whether an object is immutable. `primitive_p()` and `continuation_p()` are used internally only. At the Scheme level, the types they test for are procedures.

`special_p()` checks whether a given symbol introduces a special form (e.g.: `quote`, `if`, `define`). `syntax_p()` tests whether an object is a macro.

`atom_p()` returns truth, if its argument is an atom (has either the `ATOM_TAG` or `VECTOR_TAG` set). It also returns truth for special objects (like `#t` or the EOF), which are also atomic from the Scheme point of view.

`auto_quoting_p()` tests whether an object is self-quoting. We say that all atoms are self-quoting, which is a lie, but simplifies things a lot. Its only evil side effect is that is makes `()` and unquoted vectors valid Scheme expressions, which they are not.

```
/*
 * Type predicates
 */

#define eof_p(n)         ((n) == END_OF_FILE)
#define undefined_p(n)   ((n) == UNDEFINED)
#define unspecific_p(n)  ((n) == UNSPECIFIC)
```

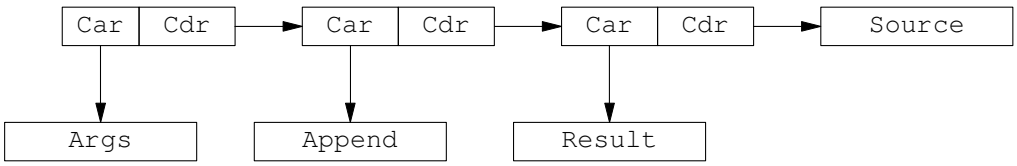
```
#define boolean_p(n)    ((n) == TRUE || (n) == FALSE)

#define constant_p(n)   (!special_value_p(n) && (Tag[n] & CONST_TAG))

#define integer_p(n) \
        (!special_value_p(n) && (Tag[n] & ATOM_TAG) && Car[n] == T_INTEGER)
#define number_p(n) \
        (!special_value_p(n) && (Tag[n] & ATOM_TAG) && \
                (Car[n] == T_REAL || Car[n] == T_INTEGER))
#define primitive_p(n) \
        (!special_value_p(n) && (Tag[n] & ATOM_TAG) && Car[n] == T_PRIMITIVE)
#define procedure_p(n) \
        (!special_value_p(n) && (Tag[n] & ATOM_TAG) && Car[n] == T_PROCEDURE)
#define continuation_p(n) \
        (!special_value_p(n) && (Tag[n] & ATOM_TAG) && \
                Car[n] == T_CONTINUATION)
#define real_p(n) \
        (!special_value_p(n) && (Tag[n] & ATOM_TAG) && Car[n] == T_REAL)
#define special_p(n)    ((n) == S_quote   || \
                         (n) == S_begin   || \
                         (n) == S_if      || \
                         (n) == S_cond    || \
                         (n) == S_and     || \
                         (n) == S_or      || \
                         (n) == S_lambda  || \
                         (n) == S_set_b   || \
                         (n) == S_define  || \
                         (n) == S_define_syntax)
#define char_p(n) \
        (!special_value_p(n) && (Tag[n] & ATOM_TAG) && Car[n] == T_CHAR)
#define syntax_p(n) \
        (!special_value_p(n) && (Tag[n] & ATOM_TAG) && Car[n] == T_SYNTAX)
#define input_port_p(n) \
        (!special_value_p(n) && (Tag[n] & ATOM_TAG) && (Tag[n] & PORT_TAG) \
         && Car[n] == T_INPUT_PORT)
#define output_port_p(n) \
        (!special_value_p(n) && (Tag[n] & ATOM_TAG) && (Tag[n] & PORT_TAG) \
         && Car[n] == T_OUTPUT_PORT)

#define symbol_p(n) \
        (!special_value_p(n) && (Tag[n] & VECTOR_TAG) && Car[n] == T_SYMBOL)
#define vector_p(n) \
        (!special_value_p(n) && (Tag[n] & VECTOR_TAG) && Car[n] == T_VECTOR)
#define string_p(n) \
        (!special_value_p(n) && (Tag[n] & VECTOR_TAG) && Car[n] == T_STRING)

#define atom_p(n) \
        (special_value_p(n) || (Tag[n] & ATOM_TAG) || (Tag[n] & VECTOR_TAG))

#define auto_quoting_p(n) atom_p(n)

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

The Rib Cage

A "rib" is an argument list in the process of creation. At any point in the evaluation of a program, the

evaluator state contains a set of ribs (due to nested procedure application), which may be thought of as a "rib cage".

The S9fES rib structure is outlined in figure 5.



args arguments to be evaluated.
append slot where evaluated arguments will be appended
result evaluated arguments
source the complete source expression

[Fig.5: Rib structure]

The *source* field is stored in the *cdr* part of the last cons in order to minimize consing during evaluation.

The `rib_args()`, `rib_append()`, `rib_result()`, and `rib_source()` macros access the individual parts of a rib. The *append* field of a rib basically caches the last element of the *result* list so that evaluated arguments can be appended in *O(1)* time. (See also: append pointer [page 50])

Note: Because adding elements to a rib is a destructive operation, this optimization breaks `call/cc` with multiple arguments in the same `lambda`. E.g., the following code will break the evaluator:

```
(let ((k (call/cc (lambda (k) k)))
      (x #t))
  ...)
```

This happens because adding the value `#t` to the rib will **mutate** the binding previously captured by `call/cc`, so invoking *k* later will re-establish the wrong state.

Rationale: The author of S9fES thinks that overall performance is more important that general `call/cc`. If you think otherwise, this is where to fix it.

```
/*
 * Rib structure
 */

#define rib_args(x)     (car(x))
#define rib_append(x)   (cadr(x))
#define rib_result(x)   (caddr(x))
#define rib_source(x)   (cdddr(x))
```

Allocators

Allocators have to be fast, so we implement them as macros. These macros are wrappers around the `cons3()` procedure, which is the principal node allocator.

`cons()` allocates a new cons cell. `new_atom()` allocates a new atom. `save()` is used to save objects on the global stack. `save_state()` saves the evaluator state on the state stack. It uses `cons3()` with `ATOM_TAG`, because the object it saves is not a node, but a C `int`.

```
/*
 * Allocators
 */

#define cons(pa, pd)            cons3((pa), (pd), 0)

#define new_atom(pa, pd)        cons3((pa), (pd), ATOM_TAG)

#define save(n)         (Stack = cons((n), Stack))
#define save_state(v)   (State_stack = cons3((v), State_stack, ATOM_TAG))
```

Arithmetics Primitives

These arithmetic functions are so low-level that they are best written as macros. `x_bignum_negative_p()`, `x_bignum_positive_p()` and `x_bignum_zero_p()` test whether a bignum integer is negative, positive, or zero, respectively.

`x_real_negative_p()`, `x_real_positive_p()`, and `real_zero_p()` are the corresponding real number operations. All of these macros expect the correct type, e.g. you cannot pass an integer to `x_real_zero_p()`.

`x_real_negate()` creates a fresh real number with the inverse prefix of the one passed to it.

```
/*
 * Bignum arithmetics
 */

#define x_bignum_negative_p(a)  ((cadr(a)) < 0)
#define x_bignum_zero_p(a)      ((cadr(a) == 0) && (cddr(a)) == NIL)
#define x_bignum_positive_p(a) \
                (!x_bignum_negative_p(a) && !x_bignum_zero_p(a))

/*
 * Real-number arithmetics
 */

#define x_real_zero_p(x) \
        (car(x_real_mantissa(x)) == 0 && cdr(x_real_mantissa(x)) == NIL)

#define x_real_negative_p(x) \
        (x_real_negative_flag(x) && !x_real_zero_p(x))

#define x_real_positive_p(x) \
        (!x_real_negative_flag(x) && !x_real_zero_p(x))

#define x_real_negate(a) \
        make_real(x_real_flags(a) & REAL_NEGATIVE?      \
                        x_real_flags(a) & ~REAL_NEGATIVE: \
                        x_real_flags(a) |  REAL_NEGATIVE, \
                x_real_exponent(a), x_real_mantissa(a))
```

Prototypes of Public Functions

The below functions are used by some S9fES extensions. Hence their prototypes are listed here.

```
/*
 * Prototypes
 */

void    add_primitives(char *name, PRIM *p);
cell    symbol_ref(char *s);
cell    cons3(cell pcar, cell pcdr, int ptag);
int     new_port(void);
char    *copy_string(char *s);
cell    error(char *msg, cell expr);
void    fatal(char *msg);
cell    integer_value(char *src, cell x);
int     length(cell x);
cell    make_char(int c);
cell    make_integer(cell i);
cell    make_port(int portno, cell type);
cell    make_string(char *s, int k);
cell    unsave(int k);
```

Interpreter Core (C Part)

```
/*
 * Scheme 9 from Empty Space
 * By Nils M Holm, 2007-2014
 * Placed in the Public Domain
 */
```

Miscellanea

First, some compile-time options. If you define `NO_SIGNALS`, the POSIX signal handlers will not be compiled in. In this case, sending the S9 process a `SIGINT` or `SIGQUIT` signal will terminate the process. When the option is undefined, `SIGINT` will return to the read-eval-print loop (REPL) and `SIGQUIT` will shut down the interpreter.

Note that defining `NO_` options is normally a bad idea due to double negation ("not defining `NO_SIGNALS` enables signals"), but in this case actually enabling the option will probably be a rare occasion and hence the positive case should be the default.

`BITS_PER_WORD_64` will make use of 64-bit integers on 64-bit systems. Not setting this option on 64-bit systems will only hurt performance, but will still work. `REALNUM` enables floating point arithmetics. We will not bother with floating point hardware, but implement reals on top of bignum integers.

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

```
 * Use -DBITS_PER_WORD_64 on 64-bit systems.
 * Use -DREALNUM to enable real number support
 *      (also add "s9-real.scm" to the heap image).
 */
```

VERSION is the version tag as displayed by the -v command line option. It is also used in image headers to make sure that each image can only be loaded by the interpreter version that created it.

We define EXTERN here to create the globals in the Header section [page 15]. EXTENSIONS is a list of initializer calls for extensions. When no extensions have been passed to us on the command line, we use an empty list of initializers.

```
#define VERSION "2014-07-21"

#define EXTERN
 #include "s9.h"
#undef EXTERN

#ifndef EXTENSIONS
 #define EXTENSIONS
#endif
```

Garbage collection is silent by default. The GC_root[] array contains the starting points for the mark phase of the garbage collector, i.e. all nodes pointed to by these variables will be marked "used".

```
int     Verbose_GC = 0;

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

Counting

The counter structure is used for counting things during evaluation. We cannot use bignum arithmetics here, because doing so might invoke the counter (which would then invoke bignums, which ... you get the idea).

Run_stats enables counting in general while Cons_stats counts the allocation of "cons" cells. We count interesting things such as reductions steps, cons cells allocated, total storage allocated, and garbage collections performed.

Storage is measured in the rather abstract unit of "nodes", which is explained in the Header section [page 22].

```
/*
 * Counting
 */

int     Run_stats, Cons_stats;

struct counter {
        int     n, n1k, n1m, n1g, n1t;
};
```

```
struct counter  Reductions,
                Conses,
                Nodes,
                Collections;
```

The `reset_counter()` function sets a counter to zero, and `count()` increments the given counter. When a counter overflows, we abort the computation in progress.

```
void reset_counter(struct counter *c) {
        c->n = 0;
        c->n1k = 0;
        c->n1m = 0;
        c->n1g = 0;
        c->n1t = 0;
}

void count(struct counter *c) {
        char    msg[] = "statistics counter overflow";
        c->n++;
        if (c->n >= 1000) {
                c->n -= 1000;
                c->n1k++;
                if (c->n1k >= 1000) {
                        c->n1k -= 1000;
                        c->n1m++;
                        if (c->n1m >= 1000) {
                                c->n1m -= 1000;
                                c->n1g++;
                                if (c->n1g >= 1000) {
                                        c->n1t -= 1000;
                                        c->n1t++;
                                        if (c->n1t >= 1000) {
                                                error(msg, NOEXPR);
                                        }
                                }
                        }
                }
        }
}
```

The `counter_to_list()` function turns a counter into a list of five integers resembling the trillions, billions, millions, thousands, and ones of its value:

```
(trillions billions millions thousands ones)
```

Turning this representation into a bignum is left to some Scheme code.

```
cell counter_to_list(struct counter *c) {
        cell    n, m;

        n = make_integer(c->n);
        n = cons(n, NIL);
        save(n);
        m = make_integer(c->n1k);
        n = cons(m, n);
        car(Stack) = n;
        m = make_integer(c->n1m);
        n = cons(m, n);
        car(Stack) = n;
```

```
        m = make_integer(c->n1g);
        n = cons(m, n);
        car(Stack) = n;
        m = make_integer(c->n1t);
        n = cons(m, n);
        unsave(1);
        return n;
}
```

Output

These are raw output functions. *All* Scheme output will be sent through this interface.The `pr_raw()` function emits the first *k* characters of the string *s*. The `Printer_limit` and `Printer_count` variables are used to limit the number of characters printed in error messages. When writing a newline to the default output port, we auto-flush the output to generate line-by-line output on terminals. (In case the default port is connected to a `FILE` that is not line-buffered.)

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

void flush(void) {
        fflush(Ports[Output_port]);
}

void pr_raw(char *s, int k) {
        if (Printer_limit && Printer_count > Printer_limit) {
                if (Printer_limit > 0)
                        fwrite("...", 1, 3, Ports[Output_port]);
                Printer_limit = -1;
                return;
        }
        fwrite(s, 1, k, Ports[Output_port]);
        if (Output_port == 1 && s[k-1] == '\n')
                flush();
        Printer_count += k;
}

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

Error Handling

These routines deal with error conditions. The `bye()` function is the principal exit point of the interpreter process. It resets the TTY to a sane state in case curses support is compiled in. `reset_tty()` resets the TTY state when the `CURSES_RESET` compile-time option is defined.

```
/*
 * Error Handling
 */
```

```
void reset_tty(void) {
#ifdef CURSES_RESET
        cell pp_curs_endwin(cell);
        pp_curs_endwin(NIL);
#endif
}

void bye(int n) {
        reset_tty();
        exit(n);
}
```

The `print_error_form()` function is used to print the form or object that caused an error. It limits the number of characters to print in order to avoid error messages spanning multiple lines (or pages, or even infinite output). `print_call_trace()` prints the names of the most recently called procedures. `Proc_max` is the maximum number of procedures to print.

```
void print_form(cell n);

void print_error_form(cell n) {
        Printer_limit = 50;
        Printer_count = 0;
        print_form(n);
        Printer_limit = 0;
}

void print_calltrace(void) {
        int     i, j;

        for (i=0; i<Proc_max; i++)
                if (Called_procedures[i] != NIL)
                        break;
        if (i == Proc_max)
                return;
        pr("call trace:");
        i = Proc_ptr;
        for (j=0; j<Proc_max; j++) {
                if (i >= Proc_max)
                        i = 0;
                if (Called_procedures[i] != NIL) {
                        pr(" ");
                        print_form(Called_procedures[i]);
                }
                i++;
        }
        nl();
}
```

The `error()` routine is called whenever the interpreter catches an error. It prints the message *msg* and the optional Scheme object *expr*. `Error_flag` is used to signal the interpreter that an error has occurred. When it is set, `error()` will not report any subsequent errors until the flag is reset, so only one error is reported per evaluation. When the interpreter works in quiet mode (`Quiet_mode` is set), `error()` will terminate the interpreter. When the `NOEXPR` constant is passed to `error()` in the *expr* argument, it will not print any Scheme object after the message. The constant indicates that the error occurred in no (particular) expression.

```
void reset_tty(void);
```

```
cell error(char *msg, cell expr) {
        int     oport;
        char    buf[100];

        if (Error_flag)
                return UNSPECIFIC;
        oport = Output_port;
        Output_port = Quiet_mode? 2: 1;
        Error_flag = 1;
        pr("error: ");
        if (box_value(S_loading) == TRUE) {
                if (File_list != NIL) {
                        print_form(car(File_list));
                        pr(": ");
                }
                sprintf(buf, "%d: ", Line_no);
                pr(buf);
        }
        pr(msg);
        if (expr != NOEXPR) {
                pr(": ");
                Error_flag = 0;
                print_error_form(expr);
                Error_flag = 1;
        }
        nl();
        print_calltrace();
        Output_port = oport;
        if (Quiet_mode)
                bye(1);
        return UNSPECIFIC;
}
```

`fatal()` is called whenever the interpreter reaches a state that does not allow it to continue operation. It then prints an error message (even if `Error_flag` is already set) and exits.

```
void fatal(char *msg) {
        Output_port = Quiet_mode? 2: 1;
        pr("fatal ");
        Error_flag = 0;
        error(msg, NOEXPR);
        bye(2);
}
```

Memory Management

Before we can do anything useful, we need a garbage collector (GC). The GC, of course, needs a pool of free nodes and vector cells to operate on. So we allocate those pools first. The `new_cons_segment()` function adds a new "memory segment" to the node pool. It is called whenever the pool becomes too small. The initial pool size is zero, so the function also allocates the initial pool.

Each node consists of a `Car`, `Cdr`, and `Tag` element, so the total size of a node is

```
2 * sizeof(cell) + sizeof(char)
```

The size of a segment to be added to the pool is kept in the `Cons_segment_size` variable. The value of this variable grows by a factor of 3/2 after each allocation, so the pool grows exponentially. At any point its current size is the sum over

$$\texttt{INITIAL\_SEGMENT\_SIZE} * 1.5^{(i-1)}$$

where *i* ranges from 1 to the number of prior `new_cons_segment()` calls.

```
/*
 * Memory Management
 */

void new_cons_segment(void) {
        Car = realloc(Car, sizeof(cell) * (Cons_pool_size+Cons_segment_size));
        Cdr = realloc(Cdr, sizeof(cell) * (Cons_pool_size+Cons_segment_size));
        Tag = realloc(Tag, Cons_pool_size + Cons_segment_size);
        if (Car == NULL || Cdr == NULL || Tag == NULL)
                fatal("new_cons_segment: out of physical memory");
        memset(&car(Cons_pool_size), 0, Cons_segment_size * sizeof(cell));
        memset(&cdr(Cons_pool_size), 0, Cons_segment_size * sizeof(cell));
        memset(&Tag[Cons_pool_size], 0, Cons_segment_size);
        Cons_pool_size += Cons_segment_size;
        Cons_segment_size = Cons_segment_size * 3 / 2;
}
```

`new_vec_segment()` allocates segments of the vector pool in the same way as `new_cons_segment()` allocates segments of the node pool.

```
void new_vec_segment(void) {
        Vectors = realloc(Vectors, sizeof(cell) *
                        (Vec_pool_size + Vec_segment_size));
        if (Vectors == NULL)
                fatal("out of physical memory");
        memset(&Vectors[Vec_pool_size], 0, Vec_segment_size * sizeof(cell));
        Vec_pool_size += Vec_segment_size;
        Vec_segment_size = Vec_segment_size * 3 / 2;
}
```

The S9fES garbage collector uses a constant-space mark-and-sweep method that is based on the Deutsch/Schorr/Waite (DSW) pointer reversal algorithm. A constant-space GC never allocates any additional storage outside of the tree traversed by it. It stores return information for getting back to previously visited nodes in the nodes themselves by reversing pointers to child nodes to make them point to parent nodes. This will be explained in detail in the following. S9fES extends the DSW algorithm to handle vectors, too.

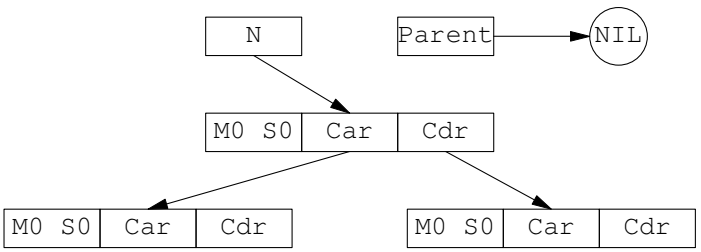
A mark-and-sweep collector works in two phases, the "mark" phase and the "sweep" phase. The mark phase traverses trees of nodes that start at specific points and marks all nodes of those trees as "used". S9fES keeps these starting points in the `GC_root[]` array. Typical starting points include, for instance, the symbol table, the current environment, and the stack.

In the sweep phase, the collector recycles all nodes that were *not* marked in the mark phase by adding them to the free list. Nodes that cannot be reached from the GC roots can never be reached by any program code, either, so the collector has proven that they can be recycled safely.

The mark phase of the S9fES collector also marks accessible I/O ports as used and its sweep phase closes ports that are no longer accessible.

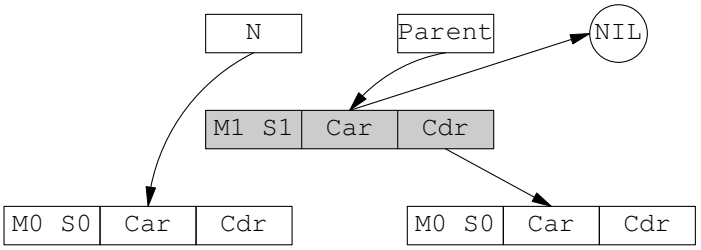
The `mark()` function implements the mark phase of the DSW GC. It consists of a finite state machine (FSM) that traverses a tree rooted at the node *n* in depth-first order, thereby marking all of its nodes (and associated I/O ports and vectors) as used.

The FSM uses three states (0,1,2) which are represented by the `STATE_TAG` and `MARK_TAG` bits of the `Tag` fields of the traversed nodes. When entering an unvisited node, both flags will be zero and when leaving a node, the `MARK_TAG` will be set, thereby indicating that the node is still in use. When visiting a node with the `MARK_TAG` bit set, the GC will simply leave it alone and return to the parent. The following figures will illustrate how the FSM uses the state of each node to traverse a tree in constant space. The variable *parent* is used to keep track of the parent of the current node.



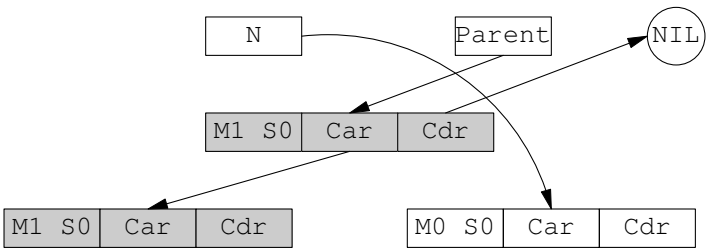
[Fig.6: GC state 0]

In GC state 0 (fig.6), the tree of three nodes is completely unvisited. All nodes are marked unused (*M*=0) and there is no parent, because *n* points to the root of the tree.



[Fig.7: GC state 1]

In state 1 (fig.7), the root of the tree has been marked (indicated by gray background). In addition its *S* flag has been set to indicate that the traversal of that node is not yet complete. *N* now points to the car child of the old *n* and *parent* points to the parent of (the new) *n*. The car field of the old *n* points back to the parent of *parent*, which is NIL at this point. The car field of the currently visited node is used for temporary storage. Its value will be restored from *n* later.



[Fig.8: GC state 2]

In state 2 (fig.8), the original value of the car field of the root node has been restored from *n*. *N* now points to the cdr child of the root node, the cdr field of the root node points to the parent of *parent*, and *parent* points to the parent of *n*. The *S* flag of the root node is now cleared to indicate that it was completely visited. This is not entirely true, though, because the FSM is still processing its cdr part.