S9 CORE

A Toolkit for Implementing Dynamic Languages

Contents

Rationale

Dynamic languages typically require some basic infrastructure that is common in their implementations, including *garbage collection*, *primitive functions*, and *dynamic type checking*, but sometimes also features like *bignum arithmetics* and *heap images*. S9core offers all of the above, and some more, in a single object file that can be linked against a dynamic language implementation. It takes care of all the nitty gritty stuff and allows the implementor to focus on the design of the language itself.

Features

- Precise, constant-space, stop-the-world garbage collection with vector pool compaction (defragmentation) and finalization of I/O ports
- Non-copying GC, all nodes stay in their original locations
- Bignum (unbounded-precision) integer arithmetics
- Decimal-based, platform-independent real number arithmetics
- Persistent heap images
- Type-checked primitive functions
- Symbol identity
- Memory allocation on heap exclusively (no `malloc()` until the heap grows)
- A basis for implementing interpreters, runtime libraries, etc
- Statically or dynamically linked
- Available on Unix, Plan 9, and in C89/POSIX environments

Reference Manual

Setup and Namespace

A module that intends to use the S9core tool kit must include the S9core header using

```
#include <s9core.h>
```

As of Mk II, the tool kit has a separate name space which is implemented by beginning all symbol names with a **S9_** or **s9_** prefix. However, many symbols can be "imported" by adding

```
#include <s9import.h>
```

Doing so will create aliases of most definitions with the prefix removed, so you can write, for instance:

```
cons(a, cons(b, NIL))
```

instead of

```
s9_cons(a, s9_cons(b, S9_NIL))
```

There are some symbol names that will not have aliases – mostly tuneable parameters of **s9core.h**. Those names will print with their prefixes in this text. All other names will have their prefixes removed.

When a module wants to use S9core functions without importing them, the following rules apply:

A lower-case function or macro name is prefixed with **s9_**, e.g. **bignum_add** becomes **s9_bignum_add**.

A capitalized function or macro name has its first letter converted to lower case and an **S9_** prefix attached, e.g.: **Real_exponent** becomes **S9_real_exponent**.

An upper-case symbol gets an **S9_** prefix, e.g.: **NIL** becomes **S9_NIL**.

S9_VERSION

The **S9_VERSION** macro expands to a string holding the S9core version in "YYYYMMDD" (year, month, day) format.

C-Level Data Types

At C level, there are only two data types in S9core. Dynamic typing is implemented by adding type tags to objects on the heap.

cell
***

A "cell" is a reference to an object on the heap. All objects are addressed using cells. A cell is wide enough to hold a pointer on the host platform (typically a **ptrdiff_t**).

Example:

```
cell x, y;
```

PRIM (struct S9_primitive)
***

A **PRIM** is a structure containing information about a primitive procedure:

```
struct S9_primitive {
        char    *name;
        cell    (*handler)(cell expr);
        int     min_args;
        int     max_args;
        int     arg_types[3];
};
```

The **name** field names the primitive procedure. The **handler** is a pointer to a function from **cell** to **cell** implementing the primitive function. Because a **cell** may reference a list or vector, functions may in fact have any number of arguments (and, for that matter, return values).

The **min_args**, **max_args**, and **arg_types[]** fields define the type of the primitive function. **min_args** and **max_args** specify the expected argument count. When they are equal, the argument count is fixed. When **max_args** is less then zero, the function accepts any number of arguments that is greater or equal to **min_args**.

The `arg_types[]` array holds the type tags of the first three argument of the primitive. Functions with more than three arguments must check additional arguments internally. Unused argument slots can be set to `T_ANY` (any type accepted).

Example:

```
PRIM Prims[] = {
  { "cons", p_cons, 2, 2, { T_ANY,  T_ANY, T_ANY } },
  { "car",  p_car,  1, 1, { T_PAIR, T_ANY, T_ANY } },
  { "cdr",  p_cdr.  1, 1, { T_PAIR, T_ANY, T_ANY } },
  { NULL }
};
```

Where `p_cons`, `p_car`, and `p_cdr` are the functions implementing the corresponding primitives.

Calling Conventions

All S9core functions protect their parameters from the garbage collector, so it is safe, for example to write

```
make_real(1, 0, make_integer(x));
```

or

```
cell n = cons(c, NIL);
n = cons(b, n);
n = cons(a, n);
```

In the first case, the integer created by `make_integer()` will be protected in the application of `make_real()`. In the second example, the object c will be protected in the first call, and the list n will be protected in all subsequent applications of `cons()`. Note that the objects b and a are not protected during the first call and a is not protected during the second call, though.

Use `save()` and `unsave()` [pg 20] to protect objects temporarily.

Configuration

Initialization and Shutdown

```
void s9_init(cell **extroots);
```

The **s9_init()** function initializes the memory pools, connects the first three I/O ports to **stdin**, **stdout**, and **stderr**, and sets up the internal S9core structures. It must be called before any other S9core functions can be used.

The **extroots** parameter is a pointer to an array of addresses of **cell**s that will be protected from the garbage collector (the so-called "GC roots"). The last array member must be **NULL**. Because **cell**s can reference trees, lists, or vectors, larger structures may be protected from GC by including their handles in this array.

Example:

```
cell Environment;
cell *GC_roots[] = { &Environment, NULL };
...
s9_init(GC_roots);

void s9_fini(void);
```

The **s9_fini()** function shuts down S9core and releases all memory allocated by it. This function is normally never called, because clean-up is done by the operating system.

The only reason to call it is to prepare for the *re-initialization* of the toolkit, for example to recover from a failed image load (see **load_image()**).

Memory Allocation

```
S9_NODE_LIMIT
S9_VECTOR_LIMIT
```

The **S9_NODE_LIMIT** and **S9_VECTOR_LIMIT** constants specify the maximal sizes of the node pool and the vector pool, respectively. The "pools" are used to allocate objects at run time. Their sizes are measured in "nodes" for the node pool and **cell**s for the vector pool. Both sizes default to 14013×1024 (14,013K).

The size of a **cell** is the size of a pointer on the host platform. The size of a node is two **cell**s plus a **char**. So the total node memory limit using the default settings on a 64-bit host would be:

$$14013 \times 1024 \times (2 \times 8 + 1) \; = \; 243,938,304 \; bytes.$$

The default vector pool limit would be:

$$14013K \; cells \; = \; 114,794,496 \; bytes.$$

At run time, the S9core toolbox will *never* allocate more memory than the sum of the above (plus the small amount allocated to primitive functions at initialization time).

When S9core runs out of memory, it will print a message and terminate program execution. However, a program can request to handle memory allocation failure itself by passing a handler to the **mem_error_handler()** function (further explanations can be found below).

The amount allocated to S9core can be changed by the user. See the **set_node_limit()** and **set_vector_limit()** functions for details.

```
void mem_error_handler(void (*h)(int src));
```

When a function pointer is passed to **mem_error_handler()**, S9core will no longer terminate program execution when a node or vector allocation request fails. The request will *succeed* and the function passed to **mem_error_handler()** will be called. *The function is then required to handle the error as soon as possible,* for example by interrupting program execution and returning to the REPL, or by throwing an exception.

The integer argument passed to a memory error handler will identify the source of the error: 1 denotes the node allocator and 2 indicates the vector allocator.

Allocation requests can still succeed in case of a low memory condition, because S9core *never* allocates more than 50% of each pool. (This is done, because using more than half of a pool will result in *GC thrashing*, which would reduce performance dramatically.)

As soon as a memory error handler has been invoked, thrashing *will* start immediately. Program execution will slow down to a crawl and eventually the allocator will fail to recover from a low-memory condition and kill the process, *even with memory error handling enabled*.

The default handler (which just terminates program execution) can be reinstalled by passing **NULL** to **mem_error_handler()**.

```
void set_node_limit(int k);
void set_vector_limit(int k);
```

These functions modify the node pool and vector pool memory limits. The value passed to the function will become the new limit for the respective pool. The limits must be set up immediately after initialization and may not be altered once set. Limits are specified in *kilo* nodes, i.e. they will be multiplied by 1024 internally.

Setting either value to zero will disable the corresponding memory limit, i.e. S9core will grow the memory pools indefinitely until physical memory allocation fails. This may cause *massive swapping* in memory-heavy applications.

S9core memory pools both start with a size of 32768 units (**S9_INITIAL_SEGMENT_SIZE** constant) and grow exponentially to a base of $\frac{3}{2}$. With the default settings, the limit will be reached after growing either pool for 15 times.

Note that actual memory limits all have the form 32768×1.5^n, so a limit that is not constructed using the above formula will probably be smaller than expected. Reasonable memory limits (using the default segment size) are listed in figure 1.

As can be seen in the table, the minimal memory footprint of S9core is 416K bytes on 32-bit and 800K bytes on 64-bit systems. In order to obtain a smaller initial memory footprint, the **S9_INITIAL_SEGMENT_SIZE** constant has to be reduced and the table in figure 1 has to be recalculated.

Limit	64-bit memory	32-bit memory
32	800K	416K
48	1200K	625K
72	1800K	937K
108	2700K	1405K
162	4050K	2107K
243	6075K	3160K
364	9100K	4733K
546	14M	7089K
820	21M	11M
1,230	31M	16M
1,846	46M	24M
2,768	69M	36M
4,152	104M	54M
6,228	156M	81M
9,342	234M	121M
14,013	350M	182M
21,019	525M	273M
31,529	788M	410M
47,293	1182M	615M
70,939	1773M	922M
106,409	2660M	1383M
159,613	3990M	2075M
239,419	5985M	3112M
359,128	8978M	4669M
538,692	13G	7003M
808,038	20G	10G
1,212,057	30G	16G
1,818,085	45G	24G
2,727,127	68G	35G
4,090,690	102G	53G
6,136,034	153G	80G

Fig 1. Memory Limits

Arithmetics

```
S9_DITIGS_PER_CELL
S9_INT_SEG_LIMIT
```

S9_DIGITS_PER_CELL is the number of *decimal* digits that can be represented by a **cell** and **S9_INT_SEG_LIMIT** is the smallest integer that can *not* be represented by an "integer segment" (which has the size of one **cell**). The integer segment limit is equal to

$$10^{S9\_DITIGS\_PER\_CELL}$$

A **cell** is called an integer segment in S9core arithmetics, because numbers are represented by chains of **cell**s (segments).

The practical use of the **S9_INT_SEG_LIMIT** constant is that bignums that are smaller than this limit can be converted to (long) integers just be extracting their first segment.

These values are *not* tunable. **S9_DIGITS_PER_CELL** is 18 on 64-bit machines, 9 on 32- bit machines, and (theoretically) 4 on 16-bit machines.

S9_MANTISSA_SEGMENTS
S9_MANTISSA_SIZE

S9_MANTISSA_SEGMENTS his is the number of integer segments (see above) in the mantissae of real numbers. The default is one segment (18 digits of precision) on 64-bit hosts and two segments (also 18 digits) on 32-bit platforms. Each additional mantissa segment increases precision by **S9_DIGITS_PER_CELL** (see above), but also slows down real number computations.

This is a compile-time option and cannot be tweaked at run time.

S9_MANTISSA_SIZE is the number of decimal digits in a mantissa. It is used in the computation of various values, such as **Epsilon** [pg 27].

S9core Types

S9core data types are pretty LISP- or Scheme-centric, but most of them can be used in a variety of languages.

Each type may be associated with a predicate testing for the type, an allocator creating an object of the given type, and one or more accessors that extract values from the type. Predicates always

return 0 (false) or 1 (true). Type predicates succeed (return 1) if
the object passed to them is of the given type.

Special Values

Special values are constant, unique, can be compared with `==`,
and have no allocators.

Type: **NIL**
Predicate: **x == NIL**

NIL ("Not In List") denotes the end of a list, an empty list, or an
empty return value. For example, to create a list of the objects a,
b, and c, the following S9core code would be used:

```
cell list = cons(c, NIL);
list = cons(b, list);
list = cons(a, list);
```

See also: **T_LIST** [pg 14]

Type: **END_OF_FILE**
Predicate: **eof_p(x)**, **x == END_OF_FILE**

END_OF_FILE is an object that is reserved for indicating the end
of file when reading from an input source. The **eof_p()** predicate
returns truth only for the **END_OF_FILE** object.

Type: **UNDEFINED**
Predicate: **undefined_p(x)**, **x == UNDEFINED**

The **UNDEFINED** value is returned by a function to indicate that its
value for the given arguments is undefined. For example,

```
bignum_divide(One, Zero)
```

would return **UNDEFINED**.

Type: **UNSPECIFIC**
Predicate: **unspecific_p(x)**, **x == UNSPECIFIC**

The **UNSPECIFIC** value can be returned by functions to indicate
that their return value is of no importance and should be ignored.

```
USER_SPECIALS
special_p(x)
```

When more special values are needed, they should be assigned *decreasing values* starting at the value of the **USER_SPECIALS** constant. The predicate **special_p()** will return truth for all special values, including user-defined ones.

Example:

```
#define TOP       (USER_SPECIALS-0)
#define BOTTOM    (USER_SPECIALS-1)
```

Type: **VOID**
Predicate: **x == VOID**

VOID denotes the absence of a value. While **UNSPECIFIC** is typically *returned* by a function to indicate that its value is uninteresting, **VOID** may be *passed* to a function to indicate that the corresponding argument may be ignored.

Tagged Types

A "tagged" object is a compound data object (pair, tree) with a type tag in its first slot. Tagged objects typically carry some payload, such as an integer value, an I/O port, or a symbol name. The internal structure of a tagged object does not matter; it is created using an allocator function and its payload is accessed using one or multiple accessor functions.

Type: **T_ANY**

When used in a **PRIM** structure, this type tag matches any other type (i.e. the described primitive procedure will accept any type in its place).

Type: **T_BOOLEAN**
Allocator: **TRUE**, **FALSE**
Predicate: **boolean_p(x)**

The **TRUE** and **FALSE** objects denote logical truth and falsity.

Type: **T_CHAR**
Allocator: **make_char(int c)**
Predicate: **char_p(x)**
Accessor: **int char_value(x)**

T_CHAR objects store single characters. The **make_char()** function expects the character to store, and **char_value()** retrieves the character.

Example:

```
make_char('x')
```

Type: **T_INPUT_PORT**
Allocator: **make_port(int portno, T_INPUT_PORT)**
Predicate: **input_port_p(x)**
Accessor: **int port_no(x)**

The **make_port()** allocator boxes a port handle. The port handle must be obtained by one of the I/O routines [pg 31] before passing it to this function. **port_no()** returns the port handle stored in an **T_INPUT_PORT** (or **T_OUTPUT_PORT**) object.

Example:

```
cell p = open_input_port(path);
if (p >= 0) return make_port(p, T_INPUT_PORT);
```

Type: **T_INTEGER**
Allocator: **make_integer(cell segment)**
Predicate: **integer_p(x)**
Accessor: **cell bignum_to_int(x)**

The **make_integer()** function creates a single-segment bignum integer in the range from

$$-10^{S9\_DITIGS\_PER\_CELL} + 1 \;\; to \;\; 10^{S9\_DITIGS\_PER\_CELL} - 1$$

To create larger bignum integers, the **string_to_bignum()** function [pg 38] has to be used.

The **bignum_to_int()** accessor returns the value of a single-segment bignum integer or **UNDEFINED**, if the bignum has more than a single segment. There is no way to convert multi-segment bignums to a native C type.

Example:

```
cell x = make_integer(-12345);
int  i = bignum_to_int(x);
```

Type: `T_LIST`, `T_PAIR`
Allocator: `cons(cell car_val, cell cdr_val)`
Predicate: `pair_p(x)`
Accessor: `cell car(x)`, `cell cdr(x)`

The difference between the `T_PAIR` and `T_LIST` type tags is that `T_LIST` also includes `NIL`, which `T_PAIR` does not. Both type tags are used for primitive procedure type checking exclusively.

The `cons()` allocator returns an ordered pair of any two values. It is in fact an incarnation of the LISP function of the same name. The accessors `car()` and `cdr()` retrieve the first and second value from a pair, respectively.

`pair_p()` succeeds for pairs created by `cons()`. `T_LIST` corresponds to

`pair_p(x) || x == NIL`

Further accessors, like `caar()` and friends, are also available and will be explained later in this text. [pg 21]

Example:

```
cons(One, NIL);          /* list */
cell x = cons(One, Two); /* pair */
car(x);                  /* One  */
cdr(x);                  /* Two  */
```

Type: `T_OUTPUT_PORT`
Allocator: `make_port(int portno, T_OUTPUT_PORT)`
Predicate: `output_port_p(x)`
Accessor: `int port_no(x)`

See `T_INPUT_PORT`, above, for details.

Example:

```
make_port(port_no, T_OUTPUT_PORT);
```

Type: `T_PRIMITIVE`
Allocator: `make_primitive(PRIM *p)`
Predicate: `primitive_p(x)`
Accessor: `int prim_slot(x)`, `int prim_info(x)`

The **make_primitive()** function allocates a slot in an internal primitive function table, fills in the information in the given **PRIM** structure, and returns a primitive function object referencing that table entry. The **prim_info()** function retrieves the stored information (as a **PRIM ***).

The **prim_slot()** accessor returns the slot number allocated for a given primitive function object in the internal table. Table offsets can be used to identify individual primitive functions.

See the the discussion of the **PRIM** structure [pg 4] for an example of how to set up a primitive function. Given the table shown there, the following code would create the corresponding **T_PRIMITIVE** objects:

```
for (i=0; p[i].name; i++) {
        prim = make_primitive(&p[i]);
        ...
}
```

Type: **T_FUNCTION**
Allocator: n/a
Predicate: **function_p(x)**
Accessor: n/a

Function objects are deliberately underspecified. The user is required to define their own function object structure and accessors.

For example, a LISP function allocator might look like this:

```
cell make_function(cell args, cell body, cell env) {
        /* args and body should be GC-protected! */
        cell fun = cons(env, NIL);
        fun = cons(body, fun);
        fun = cons(args, fun);
        return new_atom(T_FUNCTION, fun);
}
```

Given the structure of this function object, the corresponding accessors would look like this:

```
#define fun_args(x) (cadr(x))
#define fun_body(x) (caddr(x))
```

```
#define fun_env(x)   (cadddr(x))
```

Type: **T_REAL**
Allocator: **make_real(int s, cell e, cell m)**
      Make_real(int f, cell e, cell m)
Predicate: **real_p(x)**
Accessor: **cell real_mantissa(x)**, **cell real_exponent(x)**,
      Real_flags(x)

A real number consists of three parts, a "mantissa" (the digits of the number), an exponent (the position of the decimal point), and a "flags" field, currently just containing the sign of the number.

The value of a real number is

$$sign \times mantissa \times 10^{exponent}$$

The **real_mantissa()** and **real_exponent()** functions extract the mantissa and exponent, respectively. When applied to a bignum integer, the mantissa will be the number itself and the exponent will always be 0.

Note that **real_mantissa** returns a bignum integer, but **real_exponent** returns an unboxed, **cell**-sized integer.

The **Real_flags()** accessor can only be applied to real numbers. It extracts the flags field.

The **make_real()** function is the principal real number allocator. It expects a sign s (-1 or 1), an exponent as single **cell**, and a mantissa in the form of a bignum integer. When the mantissa is too large, the function will return **UNDEFINED**.

Make_real() is a "quick and dirty" allocator. It expects a flags field in the place of a sign, a chain of integer segments instead of a bignum, and it does not perform any overflow checking. *Caution: This function can create an invalid real number!*

Examples:

```
cell m = make_integer(123);
cell r = make_real( 1,  0, m); /* 123 */
cell r = make_real( 1, 10, m); /* 1.23e+12 */
cell r = make_real(-1, -5, m); /* -0.00123 */
```

Type: **T_STRING**

Allocator: `make_string(char *s, int k)`
Predicate: `string_p(x)`
Accessor: `char *string(x)`, `int string_len(x)`

The `make_string()` function creates a string of the length k and initializes it with the content of s. When the length n of s is less than k, the last $k - n$ characters of the resulting string object will be undefined.

Strings are counted *and* NUL-terminated. The counted length of a given string is returned by the `string_len()` function, the C string length of x is `strlen(string(x))`.

The `string()` accessor returns a pointer to the `char` array holding the string.

Note: no string obtained by `string()` or `symbol_name()` may be passed to `make_string()` as an initialization string, because vector objects (including strings and symbols) may move during heap compaction. The proper way to copy a string is

```
int k = string_len(source);
cell dest = make_string("", k-1);
memcpy(string(dest), string(source), k);
```

Alternatively, the `copy_string()` function [pg 19] may be used.

Type: `T_SYMBOL`
Allocator: `make_symbol(char *s, int k)`,
            `symbol_ref(char *s)`
Predicate: `symbol_p(x)`
Accessor: `char *symbol_name(x)`, `int symbol_len(x)`

Typically, the `symbol_ref()` function is used to create *or reference* a symbol. A *symbol* is a unique string with an identity operation defined on it. I.e. referencing the same string twice using `symbol_ref` will return *the same symbol*. Hence symbols can be compared using the `==` operator.

The `make_symbol()` function creates an "uninterned" symbol, i.e. a symbol with no identity (which cannot be compared or referenced). In a typical implementation, this function will not be used.

See the **T_STRING** description for further details and caveats.

Example:

```
cell sym = symbol_ref("foo");
```

Type: **T_SYNTAX**
Allocator: n/a
Predicate: **syntax_p(x)**
Accessor: n/a

Like function objects, syntactic abstractions ("macros") are deliberately underspecified. Typically, the value of a **T_SYNTAX** object would be a **T_FUNCTION** object.

Type: **T_VECTOR**
Allocator: **make_vector(int k)**
Predicate: **vector_p(x)**
Accessor: **cell *vector(x)**, **int vector_len(x)**

The **make_vector()** function returns a vector of k elements (slots) with all slots set to **UNDEFINED**.

vector() returns a pointer to the slots of the given vector, **vector_len()** returns the number of slots.

Example:

```
cell v = make_vector(100);
save(v);
for (i=0; i<100; i++) {
        x = make_integer(i);
        vector(v)[i] = x;
}
unsave(1);
```

Note: the result of **vector()** may not be used on the left side of an assignment where the right side allocates any objects. When in doubt, first assign the value to a temporary variable and then the variable to the vector. For an explanation see **T_STRING**.

Type: **T_CONTINUATION**
Allocator: n/a
Predicate: **continuation_p(x)**
Accessor: n/a

A "continuation" object is used to store the value of a captured continuation (as in Scheme's `call/cc`). Its implementation is left to the user.

Additional Allocators

```
cell cons3(cell a, cell d, int t);
```

The `cons3()` function is the principal node allocator of S9core. It is like `cons()`, but has an additional parameter for the "tag" field. The tag field of a node assigns specific properties to a node. For example, it can turn a node into an "atom" [pg 19], a vector reference, or an I/O port reference. In fact, `cons()` is a wrapper around `cons3()` that supplies an empty (zero) tag field.

The most interesting user-level application of `cons3()` is maybe the option to mix in a **CONST_TAG** in order to create an immutable node. Note though, that immutability is not enforced by S9core itself, because it never alters any nodes. However, implementations using S9core can use the `constant_p()` predicate to check for immutability.

Also note that "atoms" are typically created by the `new_atom()` allocator, explained below.

```
cell copy_string(cell x);
```

This function creates an new string object with the same content as the given string object.

```
new_atom(x, d)
atom_p(x)
```

An *atom* is a node with its atom flag set. Unlike a "cons" node, as delivered by `cons()`, an atom has no reference to another node in its car field. Instead of a reference, it can carry any value in the car field, for example: the character of a character object, a bignum integer segment, or a type tag. The `new_atom()` function expects any value in the x parameter and a node reference in the d parameter.

Tagged S9core objects are composed of multiple atoms. For example, the following program would create a "character" object containing the character `'x'`:

```
cell n = new_atom('x', NIL);
n = new_atom(T_CHAR, n);
```

(Don't do this, though; use `make_char()` instead!)

The `atom_p()` function checks whether the given node is an atom. S9core atoms encompass all the special values (like `NIL`, `TRUE`, `END_OF_FILE`, etc), all nodes with the atom flag set (including all tagged types), and all vector objects (see below). In fact, only "conses" (as delivered by `cons()`) are considered to be non-atomic.

```
cell new_port(void);
```

The `new_port()` function returns a handle to a port, but does not assign any `FILE` to it. A file can be assigned by using the return value of `new_port()` as an index to the `Ports[]` array. A negative return value indicates failure (out of free ports).

Example:

```
int p = new_port();
if (p >= 0) {
        Ports[p] = fopen(file, "r");
}
```

```
cell new_vec(cell type, int size);
```

This function allocates a new *vector*. A vector object has a type tag in its car field and a reference into the vector pool in its cdr field, that is, neither of its fields reference any other node. The *type* parameter is the type tag to be installed in the new vector atom and *size* is the number *bytes* to allocate in the vector pool. The newly allocated segment of the vector pool will be left uninitialized.

Example:

```
new_vec(T_STRING, 100);
new_vec(T_VECTOR, 100 * sizeof(cell));
```

```
save(n)
```

```
cell unsave(int k);
```

save() saves an object on the internal S9core stack and **unsave(n)** removes n elements from the stack and returns the one last removed (i.e. the previously n^{th} element on the stack).

The S9core stack is mostly used to protect objects from being recycled by the GC.

Removing an element from an empty stack will cause a fatal error and terminate program execution.

Example:

```
cell a = cons(One, NIL);
save(a);
cell b = cons(Two, NIL); /* a is protected */
b = cons(b, NIL);        /* still protected */
a = unsave(1);
a = cons(a, b);
```

Additional Predicates

```
constant_p(x)
```

This predicate succeeds, if the object passed to it has its **CONST_TAG** set, i.e. if it should be considered to be immutable.

Example:

```
if (constant_p(x))
        /* error: x is constant */
```

```
number_p(x)
```

The **number_p()** predicate succeeds, if its argument is either a bignum integer or a real number.

Additional Accessors

```
caar(x) ... cddddr(x)
```

These are the usual LISP accessors for nested lists and trees. For instance,

`cadr(x)`

denotes the "car of the cdr of x". All names can be decoded by reading their "a"s and "d"s from the right to the left, where each "a" denotes a car accessor, and each "d" a cdr, e.g.

```
  cadadr of ((1 2) (8 9))
= cadar  of ((8 9))
= cadr   of (8 9)
= car    of (9)
=            9
```

`tag(x)`

The **`tag()`** accessor extracts the "tag" field of a node. It is mostly used in the implementation of type predicates, to find out whether a node has its **`S9_ATOM_TAG`** set. For instance:

```
#define T_DICTIONARY (USER_SPECIALS-1)
#define dictionary_p(n)               \
        (!special_p(n) &&             \
         (tag(n) & S9_ATOM_TAG) && \
         car(n) == T_DICTIONARY)
```

Primitive Procedures

A S9core primitive function consists of a **`PRIM`** entry [pg 4] describing the primitive, and a "handler" implementing it. Here is a **`PRIM`** structure describing the Scheme procedure **`list-tail`** which, given a list and an integer n, returns the tail starting at the nth element of the list.

```
  { "list-tail", p_list_tail, 2, 2,
    { T_LIST,   T_INTEGER, T_ANY } },
```

The corresponding handler, **`p_list_tail`**, looks like this:

```
cell pp_list_tail(cell x) {
    cell    p, n;

    n = bignum_to_int(cadr(x));
    if (n == UNDEFINED)
        return error("int argument too big");
```

```
    for (p = car(x); p != NIL; p = cdr(p), n--) {
        if (!pair_p(p))
            return error("not a proper list");
        if (n <= 0)
            break;
    }
    if (n != 0)
        return error("int argument too big");
    return p;
}
```

Like all primitive handlers, `p_list_tail` is a function from `cell` to `cell`, but the argument it receives is actually a `T_LIST` of arguments, so `car` accesses the first argument and `cadr` the second one.

The function first extracts the value of the integer argument and checks for overflow (multi-segment bignum). It then traverses the list argument, decrementing n until $n = 0$ or the end of the list is reached. After some final error checking, it returns the tail of the given list.

Primitive handlers usually do not have to type-check their arguments, because there is a function that can do that *before* dispatching the handler. See below.

```
char *typecheck(cell f, cell a);
```

The `typecheck()` function expects a primitive function object f and an argument list a. It checks the types of the arguments in a against the type tags in the `PRIM` structure of f. When all arguments match, it returns `NULL`.

When a type mismatch is found, the function returns a string explaining the nature of the type error in plain English. For example, passing a `T_LIST` and a `T_CHAR` to `list-tail` would return the message

```
list-tail: expected integer in argument #2
```

The program could then add a visual representation of the actual arguments that were about to be passed to the handler.

```
cell apply_prim(cell f, cell a);
```

The `apply_prim()` function extracts the handler from the primitive function object f, calls it with the parameter a, and delivers the return value of the handler.

`apply_prim()` itself does *not* protect its arguments. Doing so is in the responsibility of the implementation.

Symbol Management

```
cell find_symbol(char *s);
```

This function searches the internal symbol list for the given symbol. When the symbol is in the list ("interned"; see also `intern_symbol()`, below), then it returns a reference to it. Otherwise, it returns `NIL`.

```
cell intern_symbol(cell y);
```

This function adds the given symbol to an internal list of symbols. Symbols contained in that list are called "interned" symbols, and only those symbols can be checked for identity (i.e. compared with C's `==` operator).

The `intern_symbol()` function should only be used to intern "uninterned" symbols, i.e. symbols created by `make_symbol()`. Symbols creates by `symbol_ref()` are automatically interned.

Note: while uninterned symbols have their uses, almost all common use cases rely on interned symbols.

```
cell symbol_to_string(cell x);
cell string_to_symbol(cell x);
```

`symbol_to_string()` returns a string object containing the name of the given symbol. `string_to_symbol()` is the inverse operation; it returns a symbol with the name given as its string argument. It also makes sure that the new symbol is interned.

Bignum Arithmetics

Bignum arithmetics can never overflow, but performance will degrade linearly as numbers get bigger.

Zero, One, Two, Ten

These are constants for common values, so you do not have to allocate them using **make_integer()**.

cell bignum_abs(cell a);

This function returns the absolute value (magnitude) of its argument, i.e. the original value with a positive sign.

cell bignum_add(cell a, cell b);

bignum_add() adds two integers and returns their result.

cell bignum_divide(cell a, cell b);

bignum_divide() divides a by b and returns both the truncated integer quotient $trunc(a/b)$ and the truncated division remainder $a - trunc(a/b) \times b$ (where $trunc$ removes any non-zero fractional digits from its argument).

The result is delivered as a cons node with the quotient in the car and the remainder in the cdr field. For example, given

```
cell a = make_integer(-23),
     b = make_integer(7);
cell r = bignum_divide(a, b);
```

the result would be equal to

```
car(r) = make_integer(-3); /* trunc(-23/7) */
cdr(r) = make_integer(-2); /* -23 - trunc(-23/7)*7 */
```

int bignum_equal_p(cell a, cell b);

This predicate returns 1, if its arguments are equal.

int bignum_even_p(cell a);

This predicate returns 1, if its argument is divisible by 2 with a remainder of 0. See **bignum_divide()**.

int bignum_less_p(cell a, cell b);

This predicate returns 1, if its argument a has a smaller value than its argument b.

```
cell bignum_multiply(cell a, cell b);
```

bignum_multiply() multiplies two integers and returns their product.

```
cell bignum_negate(cell a);
```

This function returns its argument with its sign reversed.

```
cell bignum_shift_left(cell a, int fill);
```

The **bignum_shift_left()** function shifts its argument a to the left by one decimal digit and then replaces the rightmost digit with *fill*. Note that $0 \le \mathit{fill} \le 9$ must hold!

Example:

```
cell n = make_integer(1234);
bignum_shift_left(x, 5); /* 12345 */
```

```
cell bignum_shift_right(cell a);
```

bignum_shift_right() shifts its argument to the right by one decimal digit. It returns a node with the shifted argument in the car part. The cdr part will contain the digit that "fell out" on the right side.

Example:

```
cell n = make_integer(12345);
cell r = bignum_shift_right(n);
```

The result would be equal to the following:

```
car(r) = make_integer(1234);
cdr(r) = make_integer(5);
```

```
cell bignum_subtract(cell a, cell b);
```

This function returns the difference $a - b$.

```
cell bignum_to_real(cell a);
```

The **bignum_to_real()** function converts a bignum integer to a real number. Note that for big integers, this will lead to a loss of precision. E.g., converting the integer

340282366920938463463374607431768211456

to real on a machine with a mantissa size of 18 digits will yield:

3.40282366920938463e+38

Converting it back to bignum integer will give:

340282366920938463000000000000000000000

```
cell bignum_to_string(cell x);
```

`bignum_to_string()` will return a string object containing the decimal representation of the given bignum integer. The string will be allocated in the vector pool, so it is safe to convert *really* big integers.

Real Number Arithmetics

All real number operations except those with a `Real_` or `S9_` prefix (capital first letter) accept bignum operands and convert them to real numbers silently. Of course, this may cause a loss of precision when large bignums are involved in a computation.

When *both* operands of a real number operation are bignums, the function will perform a precise bignum computation instead (except for `real_divide()`, which will always perform a real number division).

Note that S9core real numbers are base-10 (ten), so 1/2, 1/4, 1/5, 1/8 have exact results, but 1/3, 1/6, 1/7, and 1/9 do not.

```
Epsilon
```

`Epsilon` (ε) is a very small number ($10^{-(S9\_MANTISSA\_SIZE+1)}$). By all practical means, two numbers a and b should be considered to be equal, if their difference is not greater than ε, i.e. $|a - b| \le \varepsilon$.

Of course, much smaller numbers can be expressed *and ordered* by S9core (using `real_less_p()`), but the difference between two very small numbers becomes insignificant as it approaches ε.

This is particularly important when computing converging series. Here the precision cannot increase any further when the difference between the current guess x_i and previous guess x_{i-1} drops below ε. So the computation has reached a fixed point

when $|x_i - x_{i-1}| \le \varepsilon$.

Technically, the value of **Epsilon** is chosen in such a way that its number of fractional digits is one more than the mantissa size, so it cannot represent an *exact* difference between *any* two exact real numbers. For example (given a mantissa size of 9 digits:)

$0.999999999 + 0.000000001 = 1.0$

but

$0.999999999 + 0.0000000001 = 0.999999999$

In this example, the smaller value in the second equation would be equal to ε.

```
Real_flags(x)
Real_exponent(x)
Real_mantissa(x)
Real_negative_flag(x)
```

The **Real_mantissa()** and **Real_exponent()** macros are just more efficient versions of the **real_mantissa()** and **real_exponent()** functions. Unlike their function counterparts [pg 16], they accept real number operands exclusively. **Real_flags()** is described in the section on tagged types [pg 28]. **Real_negative_flag()** extracts the "negative sign" flag from the flags field of the given real number.

Note: Real_mantissa() returns a chain of integer segments *without* a type tag!

```
Real_zero_p(x)
Real_negative_p(x)
Real_positive_p(x)
```

These predicate macros test whether the given real number is zero, negative, or positive, respectively.

```
Real_negate(a)
```

This macro negates the given real number (returning a new real number object). **It does not protect its argument!**

```
cell real_abs(cell a);
```

The `real_abs()` function returns the magnitude (absolute value) of its argument (the original value with a positive sign).

```
cell real_add(cell a, cell b);
```

This function returns the sum of its arguments.

Caveat: When the arguments a and b differ by n orders of magnitude, where $n \geq S9\_MANTISSA\_SIZE$, then the sum will be equal to the larger of the two arguments. E.g. (given a mantissa size of 9):

1000000000.0 + 9.0 = 1000000000.0

because the result (1000000009) would not fit in a mantissa. Even with values that overlap only partially, the result will be truncated, resulting in loss of precision.

This is not a bug, but an inherent property of floating point arithmetics.

```
cell real_divide(cell x, cell a, cell b);
```

This function returns the quotient a/b. Loss of precision may occur, e.g.:

1.0 / 3 * 3 = 0.999999999

(given a mantissa size of 9).

The function *always* performs a real number division, even if both arguments are integers.

```
int real_equal_p(cell a, cell b);
```

The `real_equal_p()` predicate succeeds, if its arguments are equal. In S9core, two real numbers are equal, if they look equal when printed with `print_real()`.

However, the result of a real number operation may not be equal to a specific real number, even if expected. For instance,

$$1.0 / 3 \times 3 \neq 1.0$$

Generally, equality of real numbers implemented using a floating point representation should be considered with care. This applies not only to the S9core operations, but even to common hardware

implementations of real numbers. See also: **Epsilon** [pg 27].

```
cell real_floor(cell x);
cell real_trunc(cell x);
cell real_ceil(cell x);
```

These functions round the given real number as shown in figure 2.

function	round toward	sample	rounded
real_floor	$-\infty$	1.5	1.0
		-1.5	-2.0
real_trunc	0	1.5	1.0
		-1.5	-1.0
real_ceil	$+\infty$	1.5	2.0
		-1.5	-1.0

Fig 2. Rounding

```
cell real_integer_p(cell x);
```

This predicate succeeds, if the given number is an integer, i.e. has a fractional part of 0. This is trivially true for bignum integers.

```
int real_less_p(cell a, cell b);
```

This predicate succeeds, if $a < b$.

```
cell real_multiply(cell a, cell b);
```

This function returns the product of its arguments.

```
cell real_negate(cell a);
```

This function returns its argument with its sign reversed.

```
cell real_negative_p(cell a);
cell real_positive_p(cell a);
cell real_zero_p(cell a);
```

These predicates test whether the given number is zero, negative, or positive, respectively.

```
cell real_power(cell a, cell b);
```

This function returns a^b. Both a and b may be real numbers, but when b has a fractional part, a must be positive (i.e. the result of **real_power()** may not be a complex number).

```
cell real_subtract(cell a, cell b);
```

The **real_subtract()** function returns the difference $a - b$. The caveats regarding real number addition (see **real_add()**) also apply to subtraction.

```
cell real_to_bignum(cell r);
```

This function converts a integers in real number format to bignum integers. Real numbers with a non-zero fractional part cannot be converted and will yield a result of **UNDEFINED**.

Note that converting large real number will result in bignum integers with lots of zeros. Converting very large numbers may terminate the S9core process or, in case the memory limit has been removed, result in allocation of huge amounts of memory. For example, converting the number **1e+1000000** would create a string of 1 million zeros (and one one) and allocate about 25M bytes of memory in the process (on a 64-bit system). Also, the process would take a very long time.

This function is most useful for real numbers with a magnitude not larger than the mantissa size.

Input/Output

S9core input and output is based on "ports". A port is a handle to a garbage-collected object. On the C level, a port is a small integer (an index to the **Ports** array). On the S9core level, a **T_INPUT_PORT** or **T_OUTPUT_PORT** type tag is attached to the handle to make it distinguishable to the type checker.

There are input ports and output ports, but no bidirectional ports for both input and output.

When the garbage collector can prove that a port is inaccessible, it will finalize and recycle it. Of course, this works only for S9core ports. At C level, a port has to be *locked* (see **lock_port()**) to protect it from being recycled.

Input ports are finalized by closing them, output ports by flushing and closing them.

All I/O operations are performed on two implicit ports called the *current input port* and *current output port*. There are procedures for selecting these ports (e.g. `set_input_port()`).

The standard I/O files **stdin**, **stdout**, and **stderr** are assigned to the port handles 0, 1, and 2 when S9core is initialized. These ports are locked from the beginning.

```
int blockread(char *s, int k);
```

This function reads up to k character from the current input port and stores them in s. It returns the number of characters read. When an I/O error occurs, it updates the internal I/O status (see `io_status()`).

```
int readc(void)
void rejectc(int c)
```

`readc()` reads a single character from the current input port and returns it. A return value of -1 indicates the EOF or an error.

The `rejectc()` function inserts a character into the input stream, so the next `readc()` (or `blockread()`) will return it. In combination with `readc()`, it can be used to look ahead in the input stream.

Example:

```
cell peek = readc();
rejectc(peek);
```

At most two characters may be rejected subsequently, i.e. the reject buffer has a length of two characters.

```
void blockwrite(char *s, int k);
```

This function writes k characters from s to the current output port. It returns the number of characters written. When an I/O error occurs, it updates the internal I/O status (see `io_status()`).

```
int port_eof(int p);
```

This function returns a non-zero value, if reading beyond the EOF has been attempted on the given port. Otherwise it returns 0.

```
void prints(char *s);
```

`prints()` writes the C string s to the current output port.

```
void print_bignum(cell n);
```

The `print_bignum()` function writes the decimal representation of the bignum integer n to the current output port.

```
void print_expanded_real(cell n);
void print_real(cell n);
void print_sci_real(cell n);
```

These functions all write representations of the real number n to the current output port. `print_expanded_real()` prints all digits of the real number, both the integer and fractional parts. `print_sci_real()` prints numbers in "scientific" notation with a normalized mantissa and an exponent. E.g., 123.45 will print as $1.2345e{+}2$, meaning 1.2345×10^2. The exponent character may vary; see the `exponent_chars()` function [pg 38] for details.

The `print_real()` function will print numbers in expanded notation when there is an exact representation for that number, and otherwise it will print it in scientific notation.

```
nl()
```

`nl()` is short for `prints("\n");`.

```
void flush(void);
```

`flush()` commits all pending write operations on the current output port.

```
int io_status(void);
void io_reset(void);
```

The `io_status()` function returns the internal I/O state. When it returns 0, no I/O error has occurred since the call of `io_reset()` (or the initialization of S9core). When it returns -1, an I/O error has occurred in between.

`io_reset()` resets the I/O status to zero.

These two functions can be used to perform multiple I/O operations in a row without having to check each return value. Once the I/O state was changed to −1, it will stay that way until explicitly reset using `io_reset()`.

```
int open_input_port(char *path);
int open_output_port(char *path, int append);
void close_port(int port);
```

`open_input_port()` opens a file for reading and returns a port handle for accessing that file. `open_output_port()` opens the given file for output and returns a handle. When the *append* flag is zero, it creates the file. It will truncate any preexisting file to zero length. When the *append* flag is one, it will append data to an existing file. It still creates the file, if it does not exist.

The port opening functions return a negative value in case of an error.

The `close_port()` function closes the file associated with the given port handle and frees the handle. It can be used to close locked ports (see below), thereby unlocking them in the process.

```
char *open_input_string(char *s);
void close_input_string(void);
```

`open_input_string()` opens a string as input source and immediately redirects the current input port to that string. `readc()`, and `rejectc()` work as expected on string input, but `blockread()` does not. The function returns the previous input string, if any, and `NULL` otherwise.

`close_input_string()` ends input from a string and reestablishes the current input port that was in effect before opening the string (it does not reestablish a previous input string, though!).

```
int lock_port(int port);
int unlock_port(int port);
```

These functions *lock* and *unlock* a port, respectively. Locking a port protects it from being finalized and recycled by the garbage collector. For example, a function opening a file and packaging the resulting port in a `T_INPUT_PORT` object, would need to lock the port:

```
int port = open_input_port("some-file");
lock_port(port);
cell n = make_port(port, T_INPUT_PORT);
unlock_port(port);
```

Without locking the port, the `make_port()` function might close the freshly opened port when it triggers a GC. After unlocking the port, the `T_INPUT_PORT` object protects the port, *if* it is accessible through a GC root (on the stack, bound to a symbol, etc).

```
int input_port(void);
int output_port(void);
```

These functions return the *current input port* and *current output port*, respectively. `input_port` returns -1 when input is currently being read from a string.

```
cell set_input_port(cell port);
cell set_output_port(cell port);
void reset_std_ports(void);
```

The `set_input_port()` functions redirect all input to the given port. All read operations (`readc()`, `blockread()`) will use the given port after calling this function. The given port will become the new "current input port".

`set_output_port()` changes the current output port, affecting `blockwrite()`, `prints()`, etc.

The `reset_std_ports()` function sets the current input stream (handle 0) to `stdin`, the current output stream (handle 1) to `stdout`, and port handle 2 to `stderr`. It also clears the error and EOF flags of all standard ports.

```
void set_printer_limit(int k);
int printer_limit(void);
```

When `set_printer_limit()` is used to specify a non-zero "printer limit" k, then the output functions (like `prints()`, `blockwrite()`, etc) will write k characters at most and discard any excess output. The `printer_limit()` function returns a non-zero value, if the printer limit has been reached (so that no further characters will be written).

Specifying a printer limit of zero will remove any existing limit.

Printer limits are useful for printing partial data, for instance in error messages. This is especially useful when outputting cyclic structures, which would otherwise print indefinitely.

Heap Images

```
char *dump_image(char *path, char *magic);
```

The `dump_image()` function writes a heap image to the given path. The "*magic*" parameter must be a string of up to 16 characters that will be used for a magic ID when loading images.

Heap images work only, if *all* state of the language implementation using S9core is kept on the heap. Internal variables referring to the heap must be included as image variables. See the `image_vars()` function, below.

`dump_image()` will return `NULL` on success or an error message in case of failure.

```
void image_vars(cell **v);
void add_image_vars(cell **v);
```

The parameter of `image_vars()` is a list of addresses of cells that need to be saved to a heap image. This basically includes all non-temporary `cell` variables that reference the node pool when an image is dumped, for example: a symbol table, an interpreter stack, etc.

`add_image_vars()` is similar to `image_vars()`, but adds image variables to an existing list. Calling `image_vars()` will clear any previously existing list.

All variables that are GC roots [pg 5] and all global symbols [pg 17] also have to be included in the image.

Internal S9core variables are included automatically and do not have to be specified here.

```
char *load_image(char *path, char *magic);
```

The `load_image()` function loads a heap image file from the given path. It expects the heap image to contain the given magic ID (or the load will fail). See `dump_image()` for details.

When an image could be successfully loaded, the function will return `NULL`. In case of failure, it will deliver an explanatory error message in plain English.

Note: If `load_image()` fails, it leaves the heap in an undefined state. In this case, the following options exist:

• Load a different image

• Restart S9core by calling `s9_fini()` and then `s9_init()`

• Terminate the S9core process by calling `fatal()`

Memory Management

```
int gc(void);
int gcv(void);
```

The `gc()` function starts a node pool garbage collection and returns the number of nodes reclaimed. `gcv()` starts a vector pool garbage collection and compaction and returns the number of free cells in the vector pool.

GC is normally triggered by the allocator functions, but sometimes you might want to start from some known state (e.g. when benchmarking).

```
void gc_verbosity(int n);
```

When the parameter n of `gc_verbosity()` is set to 1, S9core will print information about pool growth to `stdout`. When $n = 2$, it will also print the number of nodes/cells reclaimed in each GC. $n = 0$ disables informational messages.

String/Number Conversion

```
void exponent_chars(char *s);
```

This function specifies the characters that will be interpreted as exponent signs in real numbers by `string_numeric_p()` and `string_to_real()`.

The first character of the string passed to this function will be used to denote exponents in the output of `print_sci_real()`.

The default exponent characters are `"eE"`.

```
int integer_string_p(char *s);
int string_numeric_p(char *s);
```

`string_numeric_p()` checks whether the given string represents a number. A number consists of the following parts:

- an optional $+$ or $-$ sign

- a non-empty sequence of decimal digits with an optional decimal point at any position

- an optional exponent character followed by another optional sign and another non-empty sequence of decimal digits

Subsequently, valid numbers would include, for instance:

```
0   +123   -1   .1   +1.23e+5   1e6   .5e-2
```

`integer_string_p()` checks whether a string represents an integer, i.e. a non-empty sequence of digits with an optional leading $+/-$ sign. Each integer is trivially a number by the above rules.

```
cell string_to_bignum(char *s);
```

The `string_to_bignum()` function converts a numeric string (see `integer_string_p()`) to a bignum integer and returns it. The result of this function is undefined, if its argument does not represent an integer.

```
cell string_to_real(char *s);
```

The `string_to_real()` function converts a numeric string (as recognized by `string_numeric_p()`) to a real number and returns it. The result of this function is undefined, if its argument

does not represent a real number.

It returns **UNDEFINED**, if the given exponent is too large. Converting the string to real will lead to loss of precision, if the mantissa does not fit in the internal representation, e.g.

```
string_to_real("3.1415926535897932384626")
```

will result in 3.14159265 when the internal format uses a 9-digit mantissa. In this case, the result will be truncated (rounded towards zero).

```
cell string_to_number(char *s);
```

This function converts integer representations to bignums and real number representations (containing decimal points or exponent characters) to real numbers. Its result is undefined for non-numeric strings. See also: **string_to_bignum()**, **string_to_real**() **integer_string_p()**.

Counters

```
counter
```

A **counter** is a structure for counting events. It can be reset, incremented, and read. See the following functions for details.

```
void reset_counter(counter *c);
```

This function resets the given counter to zero.

```
void count(counter *c);
```

This function increments the given counter by one. Counters overflow at one quadrillion (10^{15}). There is no overflow checking.

```
cell read_counter(counter *c);
```

This function converts the value of the given counter into a list of numbers in the range $0 . . 999$, where the first number represents the trillions, the second one the billions, etc. The last number contains the "ones" of the counter. E.g. reading a counter with a value of $12, 345, 678$ would return

```
(0 0 12 345 678)
```

Internal Counters

```
void run_stats(int run);
```

When `run_stats()` is called with a non-zero arguments, it resets all internal S9core counters and starts counting. When passed a zero argument, it stops counting and leaves the counters untouched. The counter values can be extracted using the `get_counters()` function.

```
void cons_stats(int on);
```

Passing a non-zero value to `cons_stats()` activates the the internal c (cons) counter of S9core. Passing zero to the function deactivates the counter (but does not reset it).

Cons counting is usually activated before dispatching a primitive function and immediately deactivated thereafter. It counts allocation requests made by a *program being interpreted* rather than requests made by the interpreter.

```
void get_counters(counter **n, counter **c, counter **g);
```

This function retrieves the values of the three internal S9core counters that start when `run_stats()` is called with a non-zero argument. These counters count

- the number of nodes allocated in total (n)
- the number of nodes allocated by a program (c)
- the number of garbage collections performed (g)

The n, c, and g varialbles can be passed to `read_counter` to convert them to a (machine-)readable form.

Utility Functions

```
cell argv_to_list(char **argv);
```

The `argv_to_list()` function converts a C-style `argv` argument vector to a LISP-style list of strings, containing one command line argument per string. It returns the list.

```
long asctol(char *s);
```

The `asctol()` function is like `atol()`, but does not interpret a leading 0 as a base-8 prefix, like Plan 9's `atol()` does.

```
void fatal(char *msg);
```

This function prints the given message and then aborts program execution.

```
cell flat_copy(cell n, cell *lastp);
```

`flat_copy()` copies the "spine" of the list n, i.e. the cons nodes connecting the elements of the list, giving a "shallow" or "flat" copy of the list, i.e. new spine, but identical elements.

When *lastp* is not **NULL**, it will be filled with the last cons of the fresh list, allowing, for instance, an O(1) destructive append. *lastp* will be ignored, if n is **NULL**.

```
int length(cell n);
```

This function returns the number of elements in the list n.

Caveats

Note: All caveats outlined here are due to garbage collection. This means that code exhibiting any of these issues *may* run properly most of the time and then fail unexpectedly.

Temporary Values

A *temporary* value is a **cell** that is not part of any GC-protected structure, like the symbol table, the stack, or any other GC root. Temporary values are not protected in S9core and subject to recycling by the garcage collector. E.g. the value n in

```
cell n = cons(One, NIL);
cell m = cons(Two, NIL); /* n is unprotected */
```

is *not* protected during the allocation of m and may therefore be recycled.

Most S9core functions allocate nodes, so a conservative premise would be that calling *any* S9core function (with the obvious exception of accessors, like **car()**, **string()**, or **port_no()**), will destroy temporary values.

There are several ways to protect temporary values. The most obvious one is to push the value on the stack during a critical phase:

```
cell m, n = cons(One, NIL);
save(n);
m = cons(Two, NIL);
unsave(1);
```

A less versatile, but more lightweight approach would be to create a temporary protection object (*Tmp*) and add that to the GC root as specified in **s9_init()** [5]. Using such an object, you could write:

```
cell m, n = cons(One, NIL);
Tmp = n;
m = cons(Two, NIL);
Tmp = NIL;
```

Finally, all symbols created by `symbol_ref()` or interned by `intern_symbol()` are automatically protected, because they are stored in the internal S9core symbol table. So the following code is safe:

```
cell n = symbol_ref("foo");
cell m = cons(Two, NIL);
```

Note that uninterned symbols (created by `make_symbol()`) are *not* protected!

Locations of Vector Objects

Nodes never move once allocated, e.g., the location of N will never change after executing

```
N = make_vector(10);
```

given that N is protected from GC.

However, *vector objects* (vectors, strings, and symbols) *will* be moved during garbage collection by the *vector pool compactor*. Therefore, no S9core function may be called between retrieving the payload of a vector and accessing it. For example, the following code **will not work**:

```
cell S = make_string("foo", 3);
char *s = string(S);
cell n = make_string("", 10); /* s may move */
printf("%s\n", s);
```

Because `make_string()` may trigger a vector pool garbage collection and compaction, the location of s may change before it is printed by `printf()`. In this simple example, the issue can be resolved by swapping the first two statements.

Things are more complicated in statements like

```
make_string(string(S), strlen(string(S)));
```

As explained earlier [pg 16], this statement will *not* create a copy of the string S, because the location delivered by `string(S)` may become invalid before `make_string()` has a chance to copy it. See page 16 for the proper procedure for copying strings.

The same applies to locations delivered by the **vector()** and **symbol_name()** accessors.

Mixing Assignments and Allocators

Assignments to accessors must *never* have an allocator in their rvalues. The statement

```
car(n) = cons(One, Two); /* pool may move! */
```

will fail at some point, because the pool containing n may move due to node pool reallocation.

The **cell** n is an index to an internal pool and **car** accesses a slot in that pool. When the **cons** in the above statement causes the node pool to grow, the pool will be **realloc**'ed, so the original address of the pool may become invalid *before* **car** can access the pool.

The above works with some C compilers and does not with others, but either way, *it is not covered by any C standard and should be avoided.* The proper way to write the above would be:

```
m = cons(One, Two);
car(n) = m;
```

For similar reasons, statements like

```
return cdr(bignum_divide(a, b));
```

will fail. Even here, storing the result in a temporary variable before taking the **cdr** would be the proper way.

Index