The RScheme System

Donovan Kolbly Paul R. Wilson

Revision 0.4.1: November 21, 1994

Contents

1 Introduction

This paper describes the design and implementation of a fairly straightforward and very portable Scheme system of our devising called *RScheme* (because its reflective[1] and, hey, it's *our* scheme). This system is intended to be suitable for language design and implementation experiments, educational purposes, scripting, and real applications where maximal speed is not required.

Because RScheme *is* a design and implementation of a language and it's easy to understand, experiments in this area can be conducted using RScheme as a starting point. For example, research into language construct for parallel programming could use the RScheme system to rapidly prototype the language constructs. Experiments conducted on the prototyper would yield useful information, because the performance of the system is sufficient that it doesn't mask the importance of the features under observation.

As a pedagogical tool RScheme has all the advantages of Scheme – a clean, well-structured language, with a standard interaction environment. In addition, we provide a fairly uncontroversial (for Lispy systems) object system, an important Scheme extension for pedagogical purposes.

It is also intended that RScheme be configurable to have a small footprint and low startup cost (although we haven't worked out the design to achieve this goal). If successful, RScheme will be extremely useful as a scripting language.

The reasonable performance that we achieve in RScheme makes it quite acceptable as a system in which to write real applications, as long as blazing speed is not required (and many applications require blazing speed in only a few, well-understood places, which, in our system, can be coded in C, leaving the bulk of the application in an easy-to-understand and develop language like Scheme).

Also, RScheme is designed to integrate fairly easily with C, so it can be used as a programmability extension library, even for applications written largely in C.

The core of the runtime system (mostly the garbage collector and some primitive operations [and binary code management system?]) is written in C, but most of the system is written in itself – in Scheme with object-oriented extensions.

We do occasionally sacrifice ease of understanding for a modicum of performance. For example, an interpreter is easier to understand than a compiler, but the performance of an interpreter is unacceptable for our purposes (The goal is ease of understanding *with* reasonable performance.)

Hence, instead of interpreting, we compile. For interaction, we generate an executable representation by stitching together pre-compiled fragments of code. For higher performance, we generate C code. This code is then run through a C compiler and linked into the system (statically, or dynamically if the underlying OS supports it and that is what you want).

This document is divided into two major sections. The first section is a User's Guide or a Reference Manual. The

[1] well, it's not very reflective, but it will be...

second section describes the implementation techniques used to achieve our design goals and the hackerly features provided by RScheme.

The User's Guide describes the language features provided by the system. It starts with an overview of the key features of R4RS Scheme, and then describes the extensions to Scheme we provide that are intended for general use, such as the object-oriented extensions, and the subset of R4RS macros that we provide.

The Implementation section describes how the compiler works, the design of the runtime system, and serves as a reference guide for the hackerly features provided by RScheme. This section would be suitable reading material for people who are interested in language design and implementation issues.

2 Language

Because RScheme is a Scheme system, the basic language features are those of Scheme, as described in the Revised^4 Report on the Algorithmic Language Scheme [R4RS].

2.1 Syntactic Extension w/o Macro Expansion

One major feature of this system is that it supports what some call lexically scoped macros. We do not view these syntactic extensions as macros at all, but rather as more analogous to inline call-by-name procedures.

This syntactic extension mechanism is used to provide some of the more complicated constructs required by the (Scheme) language without building support for them into the compiler. In our implementation, `case`, `or`, `and`, `delay`, and `cond` are implemented as these syntactic extensions.

The current version does not fully implement the fully general capability. For example, it is currently impossible to write a macro which writes macros.

For example, the following fragment of code, assuming that (through the use of module scope) `basic-define-syntax` is bound to the built-in define-syntax special form, should create a `define-syntax` macro which is a wrapper for the standard special form except that an extra form is recognized.

```
(basic-define-syntax define-syntax
  (syntax-form ((name . args) . body)
    (define-syntax name
      (syntax-form args . body)))
  (else basic-define-syntax))
```

The basic mechanism for these kinds of hygenic syntactic extensions is that of *compilation within environments.*

Another use of these cool syntactic extensions is embodied by the notion of "helper macros", where a macro is written not for the purpose of generating "code", but rather for the purpose of massaging what may wind up being arguments to special forms.

For example, if you were to implement "let" as a macro, you might want to have macros that do nothing more than destructure the "special" arguments to let. That is, you'd like a macro like:

```
(define-syntax let-vars
  (syntax-form ((var init) . more-bdgs)
    var . (let-vars more-bdgs))
  (syntax-form ()
    ))
```

In order for such a macro to be possible, we need the peculiar notion that a macro can "compile to" a sequence of symbols, rather than code.

Besides that, of course, we need to adopt a better syntactic structure, because the following two s-exprs are indistinguishable:

```
(var . (let-vars more-bdgs))
(var let-vars more-bdgs)
```

which is probably why Dylan does not use dotted pairs to encode *rest* arguments, and would instead write the first of the above as:

```
(var #rest (let-vars more-bdgs))
```

2.2 Object System

The object system is a fairly conventional Lisp object system based on first-class generic functions and anonymous classes. Its syntax and semantics are intended to be a compatible subset of Dylan(TM). The subset that we currently support is a single-inheritance class structure, with single-argument dispatch for methods.

By *anonymous class* we mean that there is no language-level connection between the name of a class and the class. It just so happens that pointers to class objects are often stored in top-level variables by the same name (classes *have* names, but that is for debugging and print purposes – there is no decision based on it). This is in contrast to the Meroon system (and CLOS?).

In this document, a *general instance* of a class is an object whose class is either the given class or a subclass of the given class. A *direct instance* is an object whose class is exactly (`eq?`) the given class.

`(define-class` *name* ⟨supers⟩ ⟨slots⟩`)` object system
`(define-generic-function` *name*`)` object system
`(define-method` *name* *gf* ⟨decls⟩ ⟨body⟩`)` object system

These are the basic definition forms for the object system. For example:

```
(define-class <point> (<object>)
    x
    y)
```

defines a top-level variable, `<point>`, whose value is a (constant) class object. This class object records its name (`<point>`), its inheritance (from `<object>`), and its instance fields (`x` and `y`).

This class definition implicitly creates generic functions which are the accessors and setters for instances. The accessors take on the name of the fields which they access, and the setters are of the form `set-*!` where `*` is the name of the field. So in this example, there are four generic functions, `x`, `y`, `set-x!`, and `set-y!`.

We can now create instances of this class using the special form `make`:

```
(define p1
   (make <point> x: 10
                 y: 20))
p1                          ⟹ #[<point>]
```

and apply the accessors and setters:

```
(x p1)                      ⟹ 10
(set-x! p1 3)
(x p1)                      ⟹ 3
(y p1)                      ⟹ 20
```

You can also define your own generic functions:

```
(define-generic-function range)
```

which you can then define methods on:

```
(define-method range ((self <point>))
    (sqrt (+ (* (x self) (x self))
             (* (y self) (y self)))))
(range p1)                  ⟹ 22.36068
```

In particular, you may want to define your own print function:

```
(define-method write ((self <point>) port)
    (format port "(~a,~a)" (x self) (y self)))
p1                          ⟹ (10,20)
```

2.2.1 Class Accessors

`(class-gvec?` *class*`)` primop
`(class-bvec?` *class*`)` primop
`(class-immob?` *class*`)` primop

These are the `primop` accessors for `<<class>>` objects. ⟦These aren't for normal users; should be moved to hackerly features section?⟧

`(direct-superclasses` *class*`)` object system
`(all-superclasses` *class*`)` object system
`(subclass?` *class1 class2*`)` object system
`(slot-descriptors` *class*`)` object system
`(direct-superclasses` *class*`)` object system
`(class-name` *class*`)` object system

These functions are the basic introspective functions for classes.

`(init-keyword` *slotd*`)` object system
`(init-value` *slotd*`)` object system
`(slot-type` *slotd*`)` object system
`(slot-value` *obj slotd*`)` object system

These functions provide introspective access to slot descriptors. Currently, every slot has both an init-value and an init-keyword, but this is likely to change to allow either or both. ⟦also need to implement `init-function`'s.⟧

`(generic-function-methods` *gf*`)` object system
`(method-specializers` *method*`)` object system
`(find-method` *gf specializers*`)` object system

These function permit the manipulation (introspection) of generic functions and methods, and the connection between the two. Currently (and most likely ever), the only kind of specializers we support are classes. Hence, `find-method` takes a generic function and a list of classes. Currently also, and this is likely to change, only single-dispatch is supported, so `find-method` ignores all but its first argument.

`(instance?` *obj class*`)` object system

This function makes the connection between instances and classes in the following way: It returns `#t` if and only if *obj* is an general instance of *class*. ⟦Replace current uses of `isa?` with `instance?`.⟧

2.3 Thread System

This system incorporates Scheme-level threads. The strategy for implementing threads is to capture continuations preemptively.

`(make-thread` *thunk priority*`)` threads procedure

thunk is a procedure of no arguments which will be executed as the body of the thread. The thread is not scheduled. *priority* is the priority at which the thread runs. It is a real number, nominally between -128 and 128, that determines how many slices the thread gets to run.

For example, if there are two threads with priority 10, and one thread with priority 20, then the priority sum is 40, so out of 100 time slices, the first two threads each run for 25 time slices and the second runs for 50 slices. Time sliced are currently 10ms. ⟦Get Rob's document in here⟧

2.4 Graphical User Interface

This system also provides a portable graphical user interface, originally based on the Vibrant layer from NCBI at NIH.

2.5 Interaction Mode

RScheme provides a command processor, or Read-Eval-Print loop. The function of the command processor is to repeatedly request an expression from the user, evaluate it, and print the result.

useful command: set-compile-verbose!, passing T or F according as whether you want the compiler to print out when it is compiling (ie, loading) declarations, etc.

2.6 Stylistic Conventions

The names of top-level variables by convention may encode some information about what the variable is used for.

Variables holding class names are surrounded by angle brackets (e.g., <pair>, <object>).

Mutable top-level variables are surrounded by asterisks (e.g., *ready-queue*, *current-window*).

Top-level variables holding program constants are prefixed with a dollar sign (e.g., $car-slot, $version).

Setters are suffixed with an exclamation point (e.g., set-car!, set-compile-verbose!).

An obsolescent convention is the prefix for unsafe or insanely hackerly operations. (e.g., %gvec-read, %car for an unsafe car operation).

Indention is as illustrated in the following:

```
(define foo value)

(define (foo arg1 arg2)
    (body1)
    (body2))

(let ((var1 init1)
      (var2 init2))
    (body1)
    (body2))

(if (a)
    (b)
    (c))

(lambda (arg1 arg2)
    (body1)
    (body2))
```

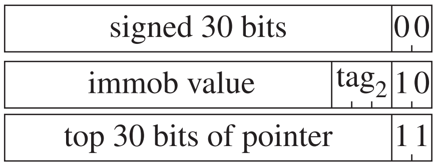

Figure 1: Structure of tagged references

3 Implementation

3.1 Runtime System

3.1.1 Object Representation

References to program and data objects are represented as 32 bit quantities, with 2 bit low primary tags and 3 bit secondary tags. The primary tags, call them INT, IMMOB, and PTR, are for integers (fixnums), [other] immediate objects, and pointers, respectively. Note that one tag (01) is not used – it is reserved for future expansion (possibly short floats). ⟦Should the INT tag be renamed FIXNUM?⟧ The following objects are represented immediately, all except for integers using the IMMOB tag:

- Integers
- ASCII and Unicode characters
- Boolean constants, true (#t) and false (#f)
- The empty list, ()
- The Dylan syntactic keywords, #rest and #key
- Flags for marking/detecting invalid values or usages:
  - #undef ⇒ the value that would be here is not defined by the system
  - #uninit ⇒ this value has not been initialized (as in a letrec)
  - #unbound ⇒ the binding from which this value came doesn't exist.
  - #none ⇒ this value doesn't exist.

Examples of how some of the invalid values/usages might occur are:

```
(if #f 3)                     ⟹ #undef
(letrec ((a b) (b 1)) a)      ⟹ #uninit
(car (cons (values) 1))       ⟹ #none
foo                           ⟹ #unbound
```

In normal development mode, all these except the last actually do occur. The last would not normally happen; an error is signalled when an unbound top-level variable is referenced.

Some of these immediate objects are grouped into a category of "unique" objects, meaning that there is only one instance of the conceptual class. (*ie*, **#none**, **#unbound**, etc.) These objects do not get their own class; they are (the only) instances of the <unique-obj> class. This allows the following statement to be made: The meta-object for an IMMOB (ie, the *class*) is a function of the secondary tag (tag_2). Hence, there are eight classes of IMMOB's:

0	<boolean>	Either #t or #f
1	<null>	()
2	<ascii-char>	ASCII character
3	<unicode-char>	Unicode character
4	<unique-obj>	A unique object

where secondary tags 5-7 are not used at present.

The unique objects are identified by the IMMOB "value" and are as follows:

0	#none
1	#undef
2	#uninit
3	#unbound
4	#rest
5	#key

Objects other than fixnums and IMMOBs are represented on the heap, using a PTR tag. Heap objects are divided into two broad catagories: gvec's (general vectors) and bvec's (byte vectors). The former are used when normal references to data objects are being stored inside the heap object, and the latter is used when raw bits are to be stored on the heap (e.g., ASCII strings are not represented as general vectors of character data objects, but rather as byte vectors of the byte values representing the low-level ASCII characters.)

Memory allocation occurs in units of 32 bit words. This can be done at no additional fragmentation cost because the underlying (gc) system allocates at that and coarser resolution anyway. Exposing this fact to the runtime system permits wordwise operations on byte vectors, e.g., string=? can do word compares.

The runtime system prepends a 2-word header to every heap object, containing the size (in bytes) of the object and its class. Among other things, the class of an object is used during GC traversal to determine how to scan an object. To this end, slot 1 of all <<class>> instances contains a "heap type", which is an integer:

0	gvec	General vector; contents are taken to be arbitrary Scheme objects
1	bvec	Byte vector; contents are bits and are skipped in scanning
2	immob	Immediate object; instances of this class cannot exist on the heap

The introspective functions ptr-class and ptr-size return the value of these fields. These functions are not normally used directly; use class-of and gvec-length or bvec-length instead. ⟦implement these functions and the integer values (we currently have #t, #f, and 0!)⟧. In a future version, heap types may themselves be pointers to "type descriptors", which will allow arbitrary "mixvec"s (*ie*, heap objects that contain a mixture of both normal object references and raw bits.)

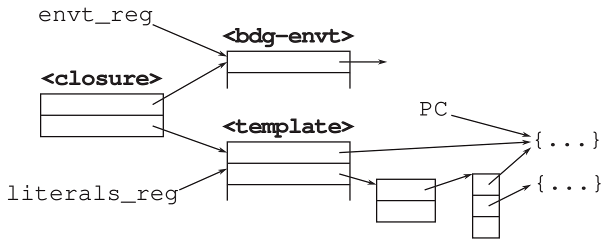

Figure 2: Illustrating the calling convention

(ptr? *thing*) primop
(immob? *thing*) primop
(fixnum? *thing*) primop

These functions are used to discriminate the various kinds of data objects that can be encountered.

3.1.2 Calling Convention

The calling convention is register-based, with the caller saving any state it needs to resume the computation in a continuation. "Registers" are implemented as global variables in the underlying language system (C). Some C compilers (e.g., the GNU compiler from the Free Software Foundation) allow global variables to be assigned fixed, H/W (CPU) registers, in which case some of "our" registers (not all – up to 10 in the current version) can be assigned real H/W regs.

The caller sets up envt_reg, literals_reg, and arg_count_reg, and transfers control to the code pointed to by the code pointer in the <template>.

The calling convention is uniform across C-compiled and closure-threaded code. From this point of view, closure-threaded code just passes around one argument, which is the runtime environment (because the <binding-envt>s captured by the closures which make up the closure-threaded code are used to hold the parameters to the CTC primitives.)

3.1.3 Stack Cache

Conceptually, the partial continuations which are used to save the control state of the program are allocated on the heap. However, as an optimization, partial continuations are initially allocated in a *stack cache* (called a *stack* cache because it is optimized for stack-like allocation/deallocation behavior.)

The stack cache is for continuations only. Binding environments are always allocated on the heap (when they aren't optimized completely away). Note that optimizing

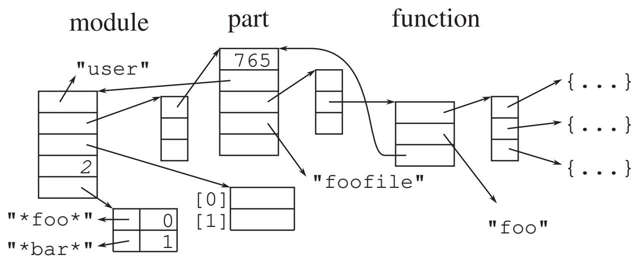

Figure 3: The static initialized data and how it points to code for a module named user containing a part with tag 765 that was built from foofile.scm, the second function in which is foo which is made up of 3 monotones. The module has two root variables, named *foo* and *bar*.

away a binding environment means putting its bindings in registers, which means that the values wind up in the continuation (note also that in the current implementation, bindings are only allocated in registers if no continuation is saved, because we don't bother to notice which bindings can be part of the immutable control state.)

The stack cache contains only partial continuations which are pointed to either by the current continuation register or by another partial continuation which is in the stack cache. Because of this constraint, whenever a pointer is copied out of the current continuation register (*ie*, using call-with-current-continuation), the contents of the cache are flushed to the heap. The cache is also flushed when it overflows.

3.1.4 Code Symbol Table (Linking)

In order to support the persistent storage of code pointers, a hierarchical (fixed 3-level) code symbol table is implemented. Modules are sets of parts, parts are sequences of functions, and functions are sequences of monotones[2]. A code pointer that appears in the system (that the system recognizes as such, as in a <template>) is always a monotone starting address.

In more detail: The runtime system maintains a master table of all modules (implemented as a pointer to an array of pointers to struct module_descr's. The top-level indirection through a pointer is to support dynamic linking of modules.) Modules are recognized by name. Each module has a list of parts (ie, a struct module_descr contains a pointer to an array of pointers to struct part_descr's. In this case, the first indirection through a pointer is because C doesn't seem to let you embed a variable-size array in a struct and initialize it.) See Figure 3 for an example.

Parts within a module are named by 29-bit integer "tags" (so that they are positive fixnum values). The tags are intended to be probably unique identifiers for a version of a part of a module, to ease the ability to recompile part of a module and still have saved images work (so long as they don't have any code pointers in the recompiled part, and the tags are to *detect* those cases).

[2] A monotone is a maximal sequence of "virtual" instructions that ends in a jump, apply, or return. See the Compiler section.

Actually, separate recompilation of sub-module units to C is no longer supported, although the linkage structure remains (Sub-module recompilation may be re-introduced, anyway, or some hack that gives the same effect (*eg*, recognizing versions of source files and basing tags on that, so that you can have persistent code pointers to parts of a module that you aren't changing).)

Each part contains a list of functions (that is, a struct part_descr contains a pointer to an array of pointers to struct function_descr's). Functions within a part are identified by their index in this array (0, 1, 2, ...).

Each function (scheme-level procedure) contains a list of monotones. Monotones within a function are identified by their index in the array, with the additional statement that the first monotone (index 0) is always the entry point of the procedure (the code pointer in a template always points to the first monotone.)

In general, there is a one-to-one correspondence between <template>'s in the heap and struct function_descr's. The exceptions are when doing dynamic linking hacking and closure-threaded code entry points, where a <template> is manufactured for each closure-threaded-code procedure, but they share a common piece of C code.

On normal systems, code pointers can escape templates by gvec-read, but a code pointer appears in this case to just be an integer or an immob (possibly an invalid one), because code is aligned on 32 or 16 bits boundaries (exception: 80x86 systems). ⟦Should we protect in all cases by having a <code-ptr> (a bvec) to encapsulate raw code pointers? And a function, template-code-ptr for reading the code pointer out of a template. What about object identity?⟧

3.1.5 Heap Image Saving

RScheme provides a fairly general purpose data structure *pickler*, which is used by the system itself to save heap images.

A pickler is a function that writes out a data structure, preserving the shared structure *within* the structure.

This pickler has an explicit notion of what constitutes the "edge" of the data structure, implemented as a list of objects that are to be mapped to names in the output. When the structure is read back in, the names are mapped to objects that already exist in the heap. Presumably, they are "similar" objects. For example, the builtin class objects like <pair> and <vector> are usually designated as edges of a data structure. When an image is loaded, these names map to the local image's version of those metaobjects.

Apart from "edges", some classes have special image-saving properties. Symbols and templates are handled specially. ⟦expand⟧

3.1.6 Scheme-Level Interrupts

The runtime system may generate interrupts at various times. To support this capability, the runtime system provides a flag and an array of immobs. When some facility wishes to signal a Scheme-level interrupt, it sets the appropriate array element to a non-`#f` value and sets the flag (the array is indexed by the interrupt number. The runtime system will notice the flag and dispatch to the scheme (builtin) function `interrupt-handler`.

```
bool interrupt_pending;
bool interrupts_enabled;
obj interrupt_flag[];
```

`interrupt_pending` is `YES` if any `interrupt_flag` is non-`#f`

`interrupts_enabled` is `YES` if the runtime system should dispatch to the interrupt handler when it notices that `interrupt_pending` is set. Interrupts should not remain disabled for very long.

The times that the system currently generates scheme-level interrupts for are:

- User interrupt (`^C` or analogue)
- Data is available on an file that has been registered for asynchronous input. ⟦What about async. output?⟧
- Timer has expired
- Garbage collection has happened

All interrupts dispatch through a single function defined in the `bltin` module, `interrupt-handler`. This function receives as arguments the interrupt information array (currently 4 values). All it does is dispatch to the higher-level interrupt handlers if the corresponding value is non-`#f`. It does this in a fixed order, which is (1) finalize, (2) user break (cntl-C), (3) data ready, (4) timeout. The timeout dispatch really should be last because if the thread system does a context switch at this point, the call to the timeout handler may not return for a long time (or possibly not ever, if the thread is killed before it gets to run again). Hence, the other system interrupts have to be handled first.

Usually, there is only one bit of information content in the interrupt info flag: `#t` indicating the interrupt has happened, as opposed to `#f` meaning the interrupt has not happened yet. Interrupt information must be immediate objects, because the pending-interrupts data structure is not part of the root set and is not on the heap. (Note that we can't make write barrier calls in process-level signal handlers... can we?)

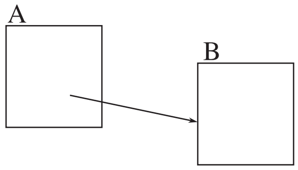

Figure 4: Objects A and B are finalizable. A contains the only pointer to B. Because all normal tracing completes before tracing of finalizing objects begins, both A and B will get their finalization routines called.

The `interrupt-handler` function finds the higher-level interrupt handlers by looking in `exception-vector`, which is a vector of 4 thunks. The runtime system arranges (via continuations) so that when `interrupt-handler` returns, the interrupted process continues where it left off. (This involves saving an arbitrary number of registers; otherwise, it is like a normal continuation save.) Because this is done with continuations, an exception handler can save the current continuation and restore another and implement scheme-level threads (this is exactly what we do in our threads system.)

3.1.7 Garbage Collection

The garbage collector used in this system is the Real-Time Garbage Collector of Johnstone and Wilson (et.al. ?) This section describes the interface between the runtime system and the garbage collector, and the parts of the garbage collector that are exposed to the scheme level.

Finalization Routines This system allows finalization routines to be registered for objects. All normal tracing completes before tracing of finalizing objects begins. See Figure 4 for a consequence of this two-phase behavior.

(`after-last-ptr` *ptr proc*) procedure

proc is a procedure of one argument that will be called when there are no other (non-weak) pointers to *ptr*, with *ptr* as the single argument. *ptr* must be a heap object. When *proc* is called, it is un-registered, so if finalization is still desired, the procedure must re-register itself.

Weak Pointers This system implements weak pointers both first-class objects and as embedded in a "weak pair", the latter being a useful optimization of the former.

(`weak-cell` *obj*) procedure
(`contents` *weak-cell*) procedure

A weak cell is used to hold a weak reference to an object. `contents` returns two values. The first is the pointer itself, or `#f` if the object has been garbage-collected. The second is used for distinguishing between a dead object and the live value #f, so it returns #t if the value is valid, and #f if the value is #f because it's been garbage collected.

```
(define c (weak-cell '(1 2)))
c                        ⟹ #[weak-cell]
(contents c)             ⟹ (1 2)
                         2 ⇒ #t
 later...
(contents c)             ⟹ #f
                         2 ⇒ #f
```

(weak-cons *obj*$_1$ *obj*$_2$) procedure
(weak-car *weak-pair*) procedure
(weak-cdr *weak-pair*) procedure

Weak pairs are convenient optimizations of weak cells, which could be implemented (approximately) as follows:

```
(define (weak-cons a b)
    (cons (weak-cell a) b))
(define (weak-car wp)
    (contents (car wp)))
(define weak-cdr cdr)
```

although they are implemented directly for efficiency. (and hence, (not (pair? (weak-cons a b)))).

Request to Clarify: Should weak pairs be immutable? Probably not. What does the write barrier look like for writing into a weak field? Probably a NOP. If we *do* call the write barrier (as if it were a normal field), then we may delay the collection of the pointed to object.

3.1.8 Hash Tables

<table> class
(table-insert *table key value*) generic fn
(table-lookup *table key*) generic fn

This is the basic interface for hash tables. ⟦Currently everything is named in terms of hash-tables, e.g., <hash-table>, make-hash-table⟧.

(make-table *eq-proc hash-proc*) procedure

This function creates an (instance) of a generic hash table. *eq-proc* should be a procedure of two arguments. It will be asked to compare two key values, the first argument is a key that's been handed to a hash-table procedure, and the second argument is a key that's been installed in the hash table. (This distinction may be important when keys are large objects that are only templatized when passed to hash table functions. For example, a <person> object may be instantiated empty except for a name when doing a name lookup. The *eq-proc* may need to know which is the template object and which is the real object.)

(make-object-table *weak-keys? weak-values?*) primitive procedure

This function is for constructing special hash tables. They are special in that they can efficiently map arbitrary objects to values. If *weak-keys?* is true, then the fact that a key has been entered into the hash table will not keep the object live. Similarly for *weak-values?* and values. If an object is garbage collected, the mapping is removed from the table. eq? is hard-coded as the equality test for keys.

Request to Clarify: When keys/values are held weakly, should image saving conditionally stop there (a la NXWriteObjectReference)?

Rationale: Object tables have to be known specially to the system for not just garbage collection reasons. Even w/o a copying garbage collector, image saving and restoring and/or page-wise persistence may move objects around. For these reasons, we don't want to expose any notion of transient pointers even to most system code.

3.2 Compiler

We compile for an abstract register machine (see description of runtime system).

The strategy for compiling to C code is to compile Scheme expressions into nested C expressions, insofar as possible while preserving the semantics of Scheme and using the semantics of C, and to glom those together into forward-executing sequences of expressions called "monotones". A good C compiler is smart about allocating temporary values when it encounters a nested expression, so we leverage off that to get good code most of the time.

⟦rephrase this:⟧ There are two things going on with the way we compile to monotones. One is that we can be sure each monotone will execute in finite time, with a (usually[3]) bounded amount of heap allocation, so we insert garbage-collector "safe points" and checks for system interrupts between monotones. The other thing is that we can do as much invocation of C functions and nesting of C expressions as we like within a single monotone (subject to the finite time/bounded allocation constraint) and rely on the C compiler to optimize within a monotone.

Since we are compiling for a register a-machine, we try to allocate binding environments to our registers as much as possible. In many cases, the alleviates the need to create binding environments at run-time (on the heap, since our stack cache is only for continuations.) Also, in implementations where we can assigned our registers to real H/W registers, this strategy puts local variables and arguments in the fastest memory.

Since arguments are passed in registers, we leave them there whenever possible, and assign registers for other bindings in increasing order. ⟦did we want to do some non-straightforward register assignment?⟧

[3]the exception is data-dependent stuff

3.2.1 Phases and Representations

The structure of the compiler is essentially two phases. The first phase takes as input source expressions (s-exprs) and produces as output intermediate code. The second phase takes as input intermediate code and produces essentially C code as output.

Each phase does work in both the top-down and bottom-up directions. The first phase, on the way down, traverses the source code, recognizing and interpreting macros, and builds the compile-time environments so indicated. On the way up in the first phase, intermediate code is glued together. The first phase is completely machine independent (even independent whether we are generating C code or CTC code[4].)

The first phase is what is implemented by the `compile` function, which takes an s-expr as input (and a ct-envt and some other stuff) and returns intermediate code representing the input expression.

The second phase is embodied by the "back end", which traverses intermediate code and generates appropriate back-end code on the way up. We currently have two back ends, for C and closure-threaded code generation. We may at some point have another back end which more closely resembles a traditional byte-code interpreter, for code compactness and speed.

3.2.2 Intermediate Code Format

Intermediate code is essentially tree-structured with decorations. The decorations include compile-time environments and references to lexical variable objects.

`<icode>`	icode
`(does-lambda?` *ic*`)`	icode
`(does-save?` *ic*`)`	icode

This is the root class for intermediate code. The functions `does-lambda?` and `does-save?` recognize when the *implementation* (not the *semantics*) of a fragment of icode capture environments or create continuations. ⟦We still just have `does-lambda?` and `does-apply?`⟧ This information is used in deciding when and where to do register allocation of binding environments.

Request to Clarify: Is this way of breaking up icode good? Do we want to introduce an additional phase, so that the first phase can generate ASTs (or something like icode for semantics) where the implementation decision (such as inlining or turning tail calls into jumps) is deferred. An additional pass may provide a good place to insert whatever unknown optimizations or reflective operations, but it would be harder to explain (maybe).

[4]except for loops – the CTC back end doesn't understand `<ic-loop>` intermediate code

3.2.3 Closure-Threaded Code Back End

3.2.4 C Code Back End

3.2.5 Virtual Instruction Format

Immediately above the C code that is generated is a layer of "virtual instructions", so called because the generated C code and the runtime system together conspire to form a register-based abstract (virtual) machine. These so-called virtual instructions are very unlike assembly language instructions in that they are expression-nestable (that is, vinsn exprs can be compositions of vinsn exprs, and vinsn statements can be compositions of vinsn statements.)

The second pass of the compilation to C code, which traverses intermediate code, generates virtual instructions. These virtual instructions are then fed into an "assembler" which generates the actual C code.

A "vinsn ea" is an effective address of the virtual instruction set, and is a way of referring to storage. An ea can be the target of either an assignment (store) or a reference (load).

`(reg` *r name*`)`	vinsn ea
`(lex-var` *frame slot name*`)`	vinsn ea
`(tl-var` *lit name*`)`	vinsn ea
`(root` *name*`)`	vinsn ea

These are the effective addresses of the virtual instruction language. *lit* is the literal number of the pointer to a top-level variable object. The *names* may be `#f` to denote a lack of knowledge or unnamedness. The names are emitted as comments in the underlying language.

`(ref` *ea*`)`	vinsn expr
`(closure` *lit*`)`	vinsn expr
`(literal` *lit*`)`	vinsn expr
`(seq` e_1 ...`)`	vinsn expr
`(if` *test t-expr f-expr*`)`	vinsn expr
`(c-call` *allocs str* e_1 e_2 ...`)`	vinsn expr
`(c-expr` *allocs str*`)`	

A string by itself may also serve as an expression, which denotes raw text, or a string at the head of a list, which denotes a call in the underlying language. *allocs* describes the allocation behavior of a C call or expression. ⟦More detail on *allocs*: should it be represented as total bytes (incl headers) or as num objects + num data bytes?⟧

Run-time size dependencies, such as to implement `make-vector`, are all hand-coded and are out-of-line (Scheme) calls. Hence, all vinsns have a bounded allocation behavior (the bound is possible partly because there is a bound on the number of arguments; the collecting of arguments into a list can only allocate so many (about 1000) `<pair>`s.).

`(label` *label*`)`	vinsn stmt
`(jump` *num label*`)`	vinsn stmt

`(apply` *num expr*`)`	vinsn stmt
`(return` *num*`)`	vinsn stmt
`(save` *num label*`)`	vinsn stmt
`(restore` *num*`)`	vinsn stmt
`(seq` s_1 s_2 ...`)`	vinsn stmt
`(do` *expr*`)`	vinsn stmt
`(set!` *ea expr*`)`	vinsn stmt
`(if` *test t-stmt f-stmt*`)`	vinsn stmt
`(check=` *num*`)`	vinsn stmt
`(check>=` *num*`)`	vinsn stmt
`(set-false<` *num*`)`	vinsn stmt
`(collect>` *num*`)`	vinsn stmt
`(bind` e_1 e_2 ...`)`	vinsn stmt
`(unbind)`	vinsn stmt

Note that every statement that can end a monotone (jump, apply, and return) has a *num* argument. This argument is the number of live registers at that point and is how the compiler communicates register usage information to the runtime system. The runtime system uses this info in case of gc work, in which case we need to know which registers have valid data in them.

3.3 System Modules

3.3.1 `primop` – Primitive Operations

This module provides mainly unsafe access to the runtime system.

Object Representation This group of primitive operations are used for examining, controlling, and creating object representations.

You should not assume that `ptr?` being true implies that either `bvec?` or `gvec?` is true, although this happens to be the case in the current version. Instead, use `bvec?` and `gvec?` explictly to determine if the respective `bvec-*` and `gvec-*` operations are appropriate. Note that:

```
(eq? (gvec? x)
     (class-gvec? (class-of x)))
```

always holds, and likewise for `bvec?` and `immob?`.

`(gvec?` *thing*`)`	primop
`(gvec-make` *class* obj_1 ...`)`	primop
`(gvec-alloc` *class len fill*`)`	primop
`(gvec-read` *obj index*`)`	primop
`(gvec-write` *obj index value*`)`	primop
`(gvec-length` *obj*`)`	primop

These functions provide access to the primitive notion of `gvec`. They are type unsafe, but the following facts had better hold:

- *class* is a general instance of `<<class>>` for which `(class-gvec?` *class*`)`.
- *index* is a fixnum, and $0 \leq index < len$ where *len* is the actual length of *obj*,
- *len* is a fixnum, and $len \geq 0$.

`(bvec?` *thing*`)`	primop
`(bvec-alloc` *class len value*`)`	primop
`(bvec-read` *b index*`)`	primop
`(bvec-write` *b index value*`)`	primop
`(bvec-length` *b*`)`	primop

These functions provide access to the primitive notion of `bvec`. Similar conditions apply to these operations as to the `gvec` operations, except that, in addition, *value* is a fixnum and $0 \leq value < 256$. Note that the length returned by `bvec-length` is in bytes, while the length returned by `gvec-length` is in words. In both cases, `*-length` returns the limit on the *index* of the accessors and setters.

For manipulating `IMMOB` and `INT` objects, we have:

`(immob?` *thing*`)`	primop
`(fixnum?` *thing*`)`	primop
`(fixnum->immob` *fixnum*`)`	primop
`(immob->fixnum` *immob*`)`	primop

These functions are the height of danger. `fixnum->immob` changes the tag of it's argument from `INT` to `IMMOB`, and `immob->fixnum` does the reverse. These functions can be used, for example, to extend the set of unique objects (currently, for example, the only syntactic keywords are `#rest` and `#key`. You could add `#value` by tweaking the scanner and tweaking the `<unique-object>` display methods. These functions would allow you to do so.)

Safer conversions between fixnums and certain immediate objects are done with:

`(fixnum->char` *fixnum*`)`	primop
`(fixnum->unicode-char` *fixnum*`)`	primop

These functions are type safe unless you supply the escape keyword `:trust-me` (after the argument), in which case neither class (fixnum) checking nor range checking (0..255 and 0..65535, respectively) is done. ⟦should we have seperate `:trust-me`'s for class and range? e.g., `:trust-my-class` and `:trust-my-value`?⟧

⟦Should we use only one kind of keyword (`foo:`)? I've been using two, `foo:` for argumented keywords, and `:foo` for arg-less keywords (flags) (but Dylan used `foo:` only, and I think CL uses `:foo` even for arg'd keywords!⟧

All other immobs are usually referenced literally (e.g., `#t`, `#f`, ...)

Many Scheme functions that operate on immediate objects are defined in the primop module. The type predicates are all type safe (which, in the context of a predicate, just means that it checks to see that it got exactly one argument) and will turn into constants in the presence of corresponding static type declarations.

`(fixnum? `*thing*`)` primop
`(immob? `*thing*`)` primop

These predicate test for an `INT` and an `IMMOB` primary tag, respectively.

`(boolean? `*thing*`)` primop
`(null? `*thing*`)` primop
`(ascii-char? `*thing*`)` primop
`(unicode-char? `*thing*`)` primop
`(unique-obj? `*thing*`)` primop

These predicates test for an `IMMOB` primary tag and the appropriate secondary tag. Actually, `null?` does *not* check only the tag – it checks all the top bits, too. Hence,

```
(null? (fixnum->immob 1))      ==> #t
(null? (fixnum->immob 9))      ==> #f
(boolean? (fixnum->immob 0))   ==> #t
(boolean? (fixnum->immob 8))   ==> #t
(boolean? (fixnum->immob 16))  ==> #t
```

Be aware of that if you are overloading the secondary tags.

Garbage Collector Interface This group of primitive operations provide introspective and effective access to the underlying garbage collector. It is from these primitives that weak pointers are built.

Hash Tables The runtime system provides hand-crafted support for hash tables. These primitive operations are the access to that which are used to construct higher-level `<table>`'s.

Heap Images The system image saver (pickler) is accessed using the following functions.

Dynamic Linking Access to the code linkage tables is provided using the following functions. When the underlying OS supports dynamic linking, that capability is accessed through some of these functions, too.

`(name->module `*str*`)` primop
`(module+tag->part `*module tag*`)` primop
`(part+index->function `*part index*`)` primop

These functions traverse the code linkage data structures and return raw C pointers (which appear to be fixnums) to the indicated structures, or `#f` if the structure cannot be found. `name->module` returns a pointer to the `struct module_descr` by the given name. `module+tag->part` returns a pointer to a `struct part_descr`.

`part+index->function` returns two values; the first is a raw C pointer to the entry point of the function (the first monotone), and the second is a pointer to the `struct function_descr` from which it was extracted. Note that the code pointer in a template corresponds to the first return value, and the "linkage" pointer in a template corresponds to the second return value.

Control Structures The runtime system provides interrupts, a stack cache, and a dynamic state registers. These functions provide access to that part of the runtime system.