RScheme

Donovan Kolbly

Version 0.1.1 (i3), Sunday, February 02, 1997 – 21:21

Contents

Chapter 1
Introduction

RScheme is an object-oriented, extended version of the Scheme dialect of Lisp.

RScheme is freely redistributable, and offers reasonable performance despite being extraordinarily portable. RScheme can be compiled to C, and the C can then compiled with a normal C compiler to generate machine code. This can be done from a running system, and the resulting object code can be dynamically linked into RScheme as a program executes. By default, however, RScheme compiles to bytecodes which are interpreted by a (runtime) virtual machine. This ensures that compilation is fast and keeps code size down. In general, we recommend using the (default) bytecode code generation system, and only compiling your time-critical code to machine code. This allows a nice adjustment of space/time tradeoffs. To the casual user, RScheme appears to be an interpreted language. You can type RScheme code at a read-eval-print loop, and it executes the code and prints the result. In reality, every expression you type to the read-eval-print-loop is compiled and the resulting code is executed. Since RScheme compiles to bytecodes by default, this is fast – RScheme doesn't need to fork a separate process to run the C compiler to generate machine code.

1.1. Meta comments

This book is currently serving multiple purposes, some of which conflict. It's *primary* goal is to serve as a specification or target for the **`0.9`** release of RScheme. However, because of the high amount of overlap in much of the system, it is also being used as documentation for the current (**`0.7.1`**) release, since there is no other serious documentation extant.

The intention is that by the time **`0.9`** is implemented, this document will be substantially complete and serve as a user guide and technical reference for RScheme. Also at that point, it will no longer need to nor serve as a description of the **`0.7`** or **`0.8`** release.

Chapter 2
Input/Output

The input/ouput facilities in RScheme are based on the facilities in R4RS, which are defined in terms of “port” types. RScheme maps the input and output port concept of R4RS to the classes `<input-port>` and `<output-port>`, which are the abstract classes that are the focus of input and ouput operations, respectively.

2.1. Input port constructors

`open-input-string` **[function]**

Creates an input port to read from a string

Signature:

`(open-input-string` *string* `)` ⇒ *input-port*

Arguments:

string An instance of `<string>`.

Returns:

input-port A general instance of `<input-port>`.

Description:

This function creates an input port which reads the characters from the given string.

`file->string` **[function]**

Reads a file into a string

Signature:

`(file->string` *path* `)` ⇒ *string*

Arguments:

path An instance of `<string>` denoting a pathname.

Returns:

string An instance of `<string>`.

Description:

This functions reads an entire file's contents into a single string. No conversions are done on the data in the file (it is interpreted as binary data). Hence, the returned string is exactly as long as the input file.

If the file with the given path cannot be opened for reading, or if the actual read fails, then an error is signalled.

2.2. Input operations

`read-line` **[function]**

Reads a line of input

Signature:

`(read-line [` *input-port* `])` ⇒ *string*

Arguments:

input-port An instance of `<input-port>`. Default is the current input port.

Returns:

string An instance of `<string>`.

Description:

This function reads a line of text from the given port. The returned string is the contents of that line, not including the terminating newline character.

If no characters can be read because the input port is at EOF, then the EOF object is returned. On the other hand, if EOF is encountered after reading some characters but before reading a newline character, then the EOF terminates the line and the returned string will consist of the characters up to the EOF.

2.3. Output port constructors

`open-output-string` **[function]**

Creates an output port for writing to a string

Signature:

`(open-input-string)` ⇒ *output-port*

Returns:

output-port A general instance of `<output-port>`.

Description:

This function creates an output port which will accumulate the data which is written to it.

When the output port is eventually closed, the return value from `close-output-port` is a string containing everything that was written to the port.

See also `with-output-to-string`.

`open-output-append-file` **[function]**

Creates an output port for appending to a file

Signature:

`(open-output-append-file` *path* `)` ⇒ *output-port*

Arguments:

path An instance of `<string>` denoting the path of the file to be appended.

Returns:

output-port A general instance of `<output-port>`.

Description:

This function creates an output port which is similar to that created by `open-output-file`, except the data written to the port is *appended* to the file.

`with-output-to-string` **[function]**

Creates an output port for writing to a string

Signature:

`(with-output-to-string` *thunk* `)` ⇒ *string*

Arguments:

thunk A procedure of no arguments.

Returns:

string An instance of `<string>`.

Description:

This function establishes a string output port (see `open-output-string`), establishes it as the current output port, and then calls the given procedure (thunk) with no arguments.

When the thunk returns normally, the output port is closed and the resulting string is returned.

2.4. Output operations

`flush-output-port` **[generic]**

Flushes an output port

Signature:

`(flush-output-port` *output-port* `)` ⇒ #rest

Arguments:

output-port An instance of `<output-port>`.

Description:

Methods on this generic function flush any data that may be buffered internally on an output port.

The return value is not specified in general, though some specific methods may specify the return value.

If flushing the data to the OS generates an error (as may happen when flushing stdio streams and the disk is full), then an error is signalled.

Chapter 3
String Manipulation

RScheme has a wide variety of string manipulation functions, providing facilities that are useful to normal programmers and going well beyond what's available in R4RS.

Consider for example the following string:

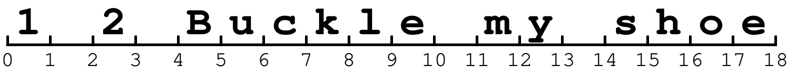

Figure 3.1. Sample string

Note the numbering assigned to the characters in the string. For purposes of array operations like `string-ref`, since arrays are 0-based in Scheme, the referenced character is the character that would make up a substring with an offset equal to the given index, and having length 1.

3.1. String operations

`string-search` **[function]**

Search a string

Signature:

`(string-search` *string* *item* `[` *offset* `]` `)` ⇒ *index*

Arguments:

string	An instance of `<string>`.
item	An item for which to search.
offset	The offset in *string* at which to start searching. Defaults to 0.

Returns:

index	An instance of `<fixnum>` if found, otherwise **`#f`**

Description:

This function searches the given *string* for an occurrence of the *item* , which may be a string, a character.

`string-split` **[function]**

Split a string into substrings

Signature:

`(string-split` *string delim* `)`⇒ *list*

Arguments:

string	An instance of `<string>`.
delim	The delimiter on which to break *string* .

Returns:

list	The list of substrings of *string* which are delimited by *delim* .

Description:

This function breaks the *string* up into substrings delimited by *delim* , which may be a string, a character, or a procedure.

In the latter case, where *item* is a procedure, *item* is presumed to have the interface of a regular expression search procedure. That is, it must be a procedure of two arguments (a string and an offset), and return two values if the pattern is found (the start and ending indexes) and no values or #f if the pattern is not found.

```
(string-split "foo bar baz" #\space) ⇒ ("foo" "bar" "baz")
(string-split ".foo..bar...baz" #\.) ⇒ ("" "foo" "" "bar" "" "" "baz")
(define p (reg-expr->proc '(+ #\,)))
(string-split "foo,,bar,,,baz" p) ⇒ ("foo" "bar" "baz")
```

If the delimitation is by function, and the function matches the empty string at some point in the process of trying to delimit the input string, then an error is signalled.

`string-join` **[function]**

Join some strings

Signature:

`(string-join` *delim list* `)`⇒ *string*

Arguments:

delim	The delimiter to seperate elements of *list* .
list	A proper list of `<string>` instances.

Returns:

string	An instance of `<string>`.

Description:

This function does the reverse of `string-split` and joins together the elements of *list* with occurrences of *delim* in between. *delim* may be a character or a string.

```
(string-join #\. '("foo" "bar" "baz")) ⇒ "foo.bar.baz"
(string-join ".." '("hi" "there" "bob")) ⇒ "hi..there..bob"
```

3.2. Character sets and tables

Different parts of the system make frequent use of character class testing. For example, the lexical analyzer defines some character classes that are used to tokenize scheme data and programs. Also, the regular expression facility can use character sets

3.3. Regular Expressions

RScheme comes with a regular expression facility, accessible as the **`regex`** module. The facility is basically in the form of a regular expression compiler, so the way it is used is that a facility function takes a regular expression specification and returns a procedure.

The returned procedure takes one or two arguments. It's first argument is the string to search for itself in. The second (optional) argument is the offset at which to start looking, which defaults to 0.

Regular expressions are composed of the following operators:

form	example	description
<char>	`#\a`	match the specified character
<string>	`"foo"`	match the given sequence of characters
<symbol>	`space`	match a character in the named character class valid character classes are: **any digit alpha uppercase lowercase hex-digit printable space**
eos	`eos`	match the end of the string
(or P1 …)	`(or #\a space)`	match any one of the given patterns (may be any pattern, not just characters)
(seq P1 …)	`(seq #\a #\b)`	match the sequence of patterns
(+ P)	`(+ space)`	match one or more occurrences of P
(* P)	`(* alpha)`	match zero or more occurrences of P
(? P)	`(? #\x)`	match zero or one P's
(range C1 C2)	`(range #\a #\z)`	match a character in the range C1..C2 (C1 and C2 must be literal characters)
(not P)	`(not space)`	match a character not matched by P (P must be a primitive pattern (ie, a character pattern, character class, or 'or' or 'not' of primitive patterns))
(save P)	`(save (+ (not #\,)))`	match P and return the matched substring as an additional value
(let N P)	`(let id (+ alpha))`	match P and return the matched substring as an additional value, using N as the name of the field.
(prefix P)	`(prefix #\a)`	match P if it is at the beginning of the string
(suffix P)	`(suffix #\.)`	match P if it is at the end of the string
(entire P)	`(entire "foo")`	match P if it is the entire string

3.3.1. Functions

`reg-expr->proc` **[function]**

Compiles a regular expression

Signature:

`(reg-expr->proc` *expr* `)` ⇒ *proc list*

Arguments:

expr A regular expression.

Returns:

proc An instance of `<function>`.

list A list describing the proc's return values.

Description:

This function compiles a regular expression into a callable object. The valid forms of regular expressions are given in the table at the beginning of this section.

The second return value, *list* , describes the values that will be returned from *proc* when it finds a match. The first element is always the symbol **`substring`**, indicating that the proc returns the start and end index of the substring that matched. The remaining elements are the names from each occurrence of a **`let`** form in *expr* , indicating the strings returned for each use of **`let`**. **`let`**'s without names are assigned integer names 1, 2, 3, …

```
(reg-expr->proc '(seq (+ space) (let w (+ alpha)))) ⇒ {a <function>}
⇒ (substring w)
```

If the *proc* finds a match, it return two values – the start and end index of the substring that matched – plus one string for each **`let`** construct in the regular expression.

Since the *proc* returns multiple values, the `bind` construct may be used to capture the multiple values.

```
(define p (reg-expr->proc '(seq (+ #\*) (let word (+ alpha)))))
(define str "…***hello there")
(p str) ⇒ 3
⇒ 11
⇒ "hello"
(substring str 3 11) ⇒ "***hello"
(bind ((s e w (p str))) w) ⇒ "hello"
```

`unformat->proc` **[function]**

Constructs an "unformat" procedure

Signature:

`(unformat->proc` *format-string* [*anywhere?*] `)` ⇒ *proc*

Arguments:

format-string	A format string.
anywhere?	An optional `<boolean>`. Default value is **`#f`**.

Returns:

proc	An instance of `<function>`.

Description:

This function constructs a procedure which will do the inverse of `format` when applied to a string.

However, the *format-string* can only contain the following format specifiers, and no modifiers are currently supported:

specifier	**meaning**
~s	consume as much input as possible and interpret it using `read`.
~a	consume as much input as possible, returning it as a string.
~d	consume a sequence of digit and dot characters and interpret it using `string->number`.

The way this function (usually) works is by using the format string to build a regular expression, using the regex binding facility to extract substrings corresponding to the format specifiers in *format-string* . The function returned by this function invokes the regular expression matcher on it's argument, and, if it matches, maps the appropriate interpretation procedures (e.g., `string->number`) over the substrings.

If the returned function does not find a match, it returns no values.

If the format string contains no format characters, then an error is signalled by `unformat->proc`, since there will be no way to distinguish success from failure cases of invocations of *proc* .

If *anywhere?* is **#t**, then the returned procedure will try to find something that matches anywhere inside the argument string. In addition, it will return two additional values (the first two), which are the start and ending offset in it's given string of the substring that matched. Otherwise, by default, the returned procedure will require an exact match between the format string and it's argument string.

(define p (unformat->proc "foo(~d,~a)"))
p ⇒ *{a <function>}*
(p "blah")
(p "foo(10,hello)") ⇒ 10
⇒ "hello"

As usual, you can use `bind` to capture the multiple values from *proc* .

(define p (unformat->proc "rev(~a,~a)"))
(define str "rev(hi,bob)")
(bind ((a b (p str))) (string-append b "," a)) ⇒ "bob,hi"

Note that an error may be signalled during the interpretation process even though a string matches the pattern. For example, the pattern for **~s** is quite permissive, but valid readable strings are more restricted. The `read` function applied to the substring that matched in the **~s** position may signal an error. Likewise, for example, the pattern for **~d** will match a string with multiple periods which is not really a number. For numbers, the interpretation procedure will signal an error in this case (rather than simply returning **#f** as `string->number` does).

The interpretation procedure for **~s** will signal an error if `read` returns EOF (e.g., if the substring is empty or contains only whitespace and comments.)

3.3.2. Examples

```
(define p (reg-expr->proc '(+ alpha)))
(p "1 2 Buckle my shoe") ⇒ 4
⇒ 10
(define p2 (reg-expr->proc '(let x (+ alpha))))
(p2 "5 6 pick up sticks") ⇒ 4
⇒ 8
⇒ "pick"
(p2 "5 6 pick up sticks" 8) ⇒ 9
⇒ 11
⇒ "up"
```

Chapter 4
Vectors

The `<vector>` class corresponds to the scheme vector type, and is essentially an array which can contain arbitrary scheme objects. The size of a vector is fixed when the vector is created.

The scheme functions `make-vector` and `vector` are used to create vectors, and the elements are accessed using `vector-ref` and `vector-set!`.

In addition to the required scheme operations, RScheme defines many functions which allow vectors to be used where lists might otherwise be used.

Since vectors are arrays, with constant-time element access, they have different performance characteristics than lists. For example, finding the length of a vector is a constant-time operation, compared to a list for which it takes time proportional to the length of the list. On the other hand, adding an element to the front of a list is a constant time operation, wheras it is a linear-time operation for vectors.

4.1. Vector operations

`vector-append` **[function]**
Concatenates a sequence of vectors

Signature:

`(vector-append` *vector* `...)` ⇒ *result-vector*

Arguments:

vector Instances of `<vector>`.

Returns:

result-vector An instance of `<vector>`.

Description:

This function appends a sequence of vectors together into one vector. The returned vector has all the elements of the given vectors, in order. Hence, the length of the returned vector is the sum of the lengths of the given vectors.

`subvector` **[function]**
Creates a subvector

Signature:

`(subvector` *vec start* `[` *end* `]` `)` ⇒ *result*

Arguments:

vec	An instance of `<vector>`.
start	An instance of `<fixnum>`.
end	An instance of `<fixnum>`. Optional. Default value is the length of the vector.

Returns:

result	An instance of `<vector>`.

Description:

This function is to vectors what `substring` is for strings.

`vector-map` **[function]**

Maps a function over a vector

Signature:

`(vector-map` *proc vec* `...)` ⇒ *result*

Arguments:

proc	An instance of `<function>`.
vec	An instance of `<vector>`.

Returns:

result	An instance of `<vector>`.

Description:

This function is to vectors what `map` is to lists.

The length of *result* is the *minimum* of the lengths of the *vec* arguments. Hence, if one of *vec* is an empty vector (having 0 length), then an empty vector is returned and *proc* is never called.

```
(define x '#(1 2 3))
(define y '#(10 20 30))
(vector-map + x y) ⇒ '#(11 22 33)
(define z '#(8 9))
(vector-map + y z) ⇒ '#(18 19)
```

`vector-for-each` **[function]**

Applies a function to elements of a vector

Signature:

`(vector-for-each` *proc vector* `...)` ⇒ *object* ...

Arguments:

proc	An instance of `<function>`.
vector	Instances of `<vector>`.

Returns:

object	Instances of `<object>`.

Description:

This function is to vectors what `for-each` is to lists.

In particular, the procedure *proc* is called with corresponding vector elements, in order (up to the length of the shortest *vector* .) Also, `vector-for-each` returns whatever the last call to *proc* returns.

Chapter 5
Tables

The `<table>` class is the abstract class of collections which map an explicit set of keys to values. The only concrete subclasses of `<table>` are subclasses of the class `<hash-table>`.

Hash tables are created using the `make-table` function.

5.1. Table operations

`make-table` **[function]**

Constructs a new hash table

Signature:

`(make-table` *test-fn* *hash-fn* `)` ⇒ *table*

Arguments:

test-fn	A function of two arguments that returns a boolean, and representing the equality predicate for the hash table.
hash-fn	A function of one argument, which will be a table key, that returns the `<fixnum>` hash value.

Returns:

table	The freshly allocated hash table.

Description:

This function creates a new hash table with the given test and hash functions. If the test function and hash function are chosen from a particular set of common pairs, then a specialized class of hash table may be returned.

The *test-fn* is a function of two arguments which will be used to test if two keys are equal. The first argument will always be a key already in the table, and the second argument will be a key being sought. The function should return `#f` if and only if the keys are not equal. As usual, returning anything else is considered to indicate that the keys are equal.

The *hash-fn* is a procedure of one argument which is used to compute a hash code from a key. The return value of the procedure must be a `<fixnum>`. As usual for hashing techniques, the *hash-fn* must return an identical value for keys that are equal according to the test-fn. Also as usual, the efficiency of a particular table depends on the even distribution in number space of the hash codes for the keys in the table.

This function will detect certain special cases and return an appropriate specialized subclass of `<hash-table>`, so whenever appropriate for the application, one of the following combinations should be used:

test function	hash function	specialized class
`eq?`	`symbol->hash`	`<symbol-table>`
`eq?`	`integer->hash`	`<hash-integer-table>`
`eq?`	`identity`	`<integer-table>`
`string=?`	`string->hash`	`<string-table>`
`string-ci=?`	`string-ci->hash`	`<string-ci-table>`
`eq?`	*any*	`<eq-table>`

`table-lookup` **[generic]**

Searches a table for the given key

Signature:

`(table-lookup` *table key* `)` $\Rightarrow$ *value*

Arguments:

table	An instance of `<table>` to be searched
key	The key for which to search. Must be an instance of the key class appropriate for the given table.

Returns:

value	The value associated with given key, or `#f` if the key is not present in the table.

Description:

This generic function on tables is used to search for the presence of a given key in a table.

The given key must be compatible with the table, or an error is signalled.

Method:

`(table-lookup` *`generic-table key`* `)` ⇒ *value*

This method searches a generic hash table (ie, with arbitrary test and hash functions) for the given key.

The given key is first handed to the table's hash function to compute a hash code. If the hash function does not return a `<fixnum>`, then an error is signalled.

The hash code is then sought in the table; if found, then the test function is called with two arguments, the first being the key in the table that has the same hash code, and the second being the key given to `table-lookup`. If that function returns non-**`#f`**, then the two keys are considered equal and a "table hit" has occurred.

`table-key-present?` **[generic]**

Checks a table for the presence of a given key

Signature:

`(table-key-present?` *table key* `)` ⇒ *boolean*

Arguments:

table	An instance of `<table>` to be searched
key	The key for which to search. Must be an instance of the key class appropriate for the given table.

Returns:

boolean	**`#t`** if the key is present, **`#f`** otherwise.

Description:

This generic function on tables is used to check for the presence of a given key in a table. This function is similar to `table-lookup`, but it returns a boolean indicator rather than the value associated with the key. Hence, it is of primary use for tables that may contain **`#f`** as a value.

`table-insert!` **[generic]**

Associates a key with a value in a table

Signature:

`(table-insert!` *table key value* `)` ⇒ *prev-value*

Arguments:

table	An instance of `<table>` to be extended
key	The key to insert.
value	The value to be associated with the key.

Returns:

prev-value	The value previously associated with given key, or **`#f`** if the key was not present in the table.

Description:

This generic function on tables is used to insert entries into a table.

The given key must be compatible with the table, or an error is signalled.

`table-remove!` **[generic]**

Removes a given key from a table

Signature:

`(table-remove!` *table key* `)`$\Rightarrow$ *value*

Arguments:

table	An instance of `<table>` to be modified.
key	The key to remove

Returns:

value	The value that was associated with *key* , or **`#f`** if the key was not present in the table.

Description:

This generic function on tables is used to remove entries from a table.

The given key must be compatible with the table, or an error is signalled.

If the given key is not present in the table, then **`#f`** is returned. No error is signalled in this case.

`table-size` **[generic]**

Computes the number of keys in the table.

Signature:

`(table-size` *table* `)`$\Rightarrow$ *integer*

Arguments:

table	An instance of `<table>`.

Returns:

integer	The number of entries in the table.

Description:

This generic function on tables returns the size of the table, which in particular will be the number of elements in the collections returned by `key-sequence` and `value-sequence`.

`key-sequence` **[generic]**

Returns a sequence of table keys

Signature:

`(key-sequence` *table* `)`⇒ *list*

Arguments:

table An instance of `<table>`

Returns:

list An instance of `<list>`

Description:

This function extracts the set of keys from a `<table>` in the form of a sequence. The length of the returned list is exactly the size of the table as reported by `table-size`.

The order of returned keys in the list is not specified, *except* to say that it is the same order as the values returned by `value-sequence`.

`value-sequence` **[generic]**

Returns a sequence of table values

Signature:

`(value-sequence` *table* `)`⇒ *sequence*

Arguments:

table An instance of `<table>`

Returns:

sequence An instance of `<list>`

Description:

This function extracts the set of values from a `<table>` in the form of a sequence. The length of the returned sequence is exactly the size of the table as reported by `table-size`.

In the current implementation, the returned sequence is a list. However, future versions may return a vector instead.

The order of returned value in the sequence is not specified, *except* to say that it is the same order as the values returned by `key-sequence`.

`table-for-each` **[function]**

Applies a procedure to entries in a hash table

Signature:

`(table-for-each` *table proc* `)`⇒

Arguments:

table	An instance of `<hash-table>`.
proc	An instance of `<function>`.

Description:

This function applies a given procedure to the entries in a hash table. The procedure *proc* must accept 3 arguments, which will be, for each entry in the hash table, the hash code, the key, and the associated value.

The number of calls to the procedure will equal the table size (modulo the following statement:)

The behavior of the iteration is *not* specified if the table is updated while the iteration is taking place.

Chapter 6
Pathnames

RScheme provides a *pathname abstraction* which provides some insulation against the different representation of path names on different operating systems. This facility is only concerned with the *naming* of files, not with the actual files and directories in the system.

The basic support provided is the notion of a directory, a base filename, and an extension.

The canonical representation is that of Unix pathnames of the general form **`foo/bar/baz.scm`**. In this case, **`foo/bar`** is the directory, **`baz`** is the base filename, and **`scm`** is the extension.

If there are multiple periods in the part after the directory, the last seperates the filename from the extension (hence the filename part can contain periods). Periods inside directory names are not treated specially.

The directory notion also includes the notion of "up" a directory (the `..` directory in unix and MS-DOS.) A canonicalized directory will consist of zero or more "up"s followed by zero or more directory names. The canonicalization process removes occurrences of the form **`foo/..`**, so an "up" will never occur in the middle of a directory pathname object.

6.1. Functions

`string->file` **[function]**

Creates a pathname object naming a file

Signature:

`(string->file` *string* `)` $\Rightarrow$ *filename*

Arguments:

string An instance of `<string>`.

Returns:

filename An instance of `<file-name>`.

Description:

This function parses the given string in canonical filename notation (unix style) into a structured `<file-name>` instance.

`string->dir` **[function]**

Creates a pathname object naming a directory

Signature:

`(string->dir` *string* `)`⇒ *dirname*

Arguments:

string An instance of `<string>`.

Returns:

dir-name An instance of `<directory-name>`.

Description:

This function parses the given string into an internal `<directory-name>` object. This involves recognizing "up" (`..`) entries and removing them when possible, and recognizing root directory indicators.

`pathname->string` **[generic]**

Converts a pathname object to the corresponding string

Signature:

`(pathname->string` *pathname* `)`⇒ *string*

Arguments:

pathname A pathname object (an instance of `<file-name>` or `<directory-name>`)

Returns:

string The string form of the pathname.

Description:

This generic function is applied to a pathname to recover the string representation of the pathname. Directory names are returned with a trailing slash.

(define p (string->file "foo/bar/baz.scm"))
(pathname->string p) ⇒ "foo/bar/baz.scm"
(pathname->string (file-directory p)) ⇒ "foo/bar/"

The returned string does not expand any special root locations. See also `pathname->os-path`.

`file-directory` **[generic]**

Accesses the directory part of a pathname

Signature:

`(file-directory` *pathname* `)`⇒ *dirname*

Arguments:

pathname A pathname object (an instance of `<file-name>`)

Returns:

dirname An instance of `<directory-name>`.

Description:

The methods on this generic function extract the directory portion of pathname objects.

`file-within-dir` **[function]**

Compute the name of the file within it's directory

Signature:

`(file-within-dir` *pathname* `)` ⇒ *string*

Arguments:

pathname A pathname object (an instance of `<file-name>`)

Returns:

string An instance of `<string>`.

Description:

This function returns the name of the file represented by the given pathname object within it's directory. That is, if the pathname is **`foo/bar/baz.scm`**, then the file is within the **`foo/bar`** directory (see `file-directory`) and the file name within the directory is **`baz.scm`**.

`extension-related-path` **[function]**

Compute the pathname of a "related" file

Signature:

`(extension-related-path` *pathname* *extension* `)` ⇒ *new-pathname*

Arguments:

pathname A pathname object (an instance of `<file-name>`).
extension A string.

Returns:

new-pathname An instance of `<file-name>`.

Description:

This function computes and returns the pathname of a file that is "close" to the file denoted by the given pathname, differing (possibly) only in it's extension.

(define f (string->file "stuff/quux.c"))
(extension-related-path f "o") ⇒ *{filename* **`stuff/quux.o`***}*

`append-path` **[function]**

Append a pathname onto a directory

Signature:

`(append-path` *dirname filename* `)` ⇒ *new-filename*

Arguments:

dirname	A directory name (an instance of `<directory-name>`).
filename	A file name (an instance of `<file-name>`).

Returns:

new-filename	An instance of `<file-name>`.

Description:

This function computes the pathname that results when the given *filename* is interpreted relative to the given *dirname* .

If *filename* is an absolute path, then it is returned. Otherwise, a new file name object is constructed.

`append-dirs` **[function]**

Append two directory pathnames.

Signature:

`(append-path` *dir1 dir2* `)` ⇒ *dir3*

Arguments:

dir1	A directory name (an instance of `<directory-name>`).
dir2	A directory name (an instance of `<directory-name>`).

Returns:

dir2	A directory name (an instance of `<directory-name>`).

Description:

This function is analagous to `append-path`, but works on a directory as it's right-hand-side argument. That is, the function computes the directory referred to when *dir2* is interpreted relative to *dir1* .

If *dir2* is an absolute path, then it is returned. Otherwise, a new directory-name object is constructed.

`pathname->os-path` **[generic]**

Converts a pathname object to the corresponding string

Signature:

`(pathname->os-path` *pathname* `)` ⇒ *string*

Arguments:

pathname A pathname object (an instance of `<file-name>` or `<directory-name>`)

Returns:

string The string form of the pathname.

Description:

This generic function is applied to a pathname to compute the native representation of the given path, suitable to passing to functions that actually operate on the file system, such as `open-input-file` and `stat`.

In unix, directory names are returned *without* a trailing slash.

Special root locations are expanded to their current value.

6.2. Example usage

(define f (string->file "foo/bar/baz.scm"))
f ⇒ *{filename* **`foo/bar/baz.scm`***}*
(file-within-dir f) ⇒ "baz.scm"
(extension-related-path f "o") ⇒ *{filename* **`foo/bar/baz.o`***}*
(define d (file-directory f))
d ⇒ *{dirname* **`foo/bar/`***}*
(append-path d (string->file "INDEX")) ⇒ *{filename* **`foo/bar/INDEX`***}*
(append-path d (string->file "../INDEX")) ⇒ *{filename* **`foo/INDEX`***}*
(append-dirs d (string->dir "../CVS")) ⇒ *{dirname* **`foo/CVS/`***}*

Chapter 7
Exception handling

The exception handling model of RScheme is based on that of Dylan(TM). All exceptions are represented by instances of a condition class, which is a subclass of `<condition>`.

`signal` **[function]**
Signal a condition

Signature:
`(signal` *condition* `)` $\Rightarrow$ *object* ...

Signature:
`(signal` *message* *arg* ...`)` $\Rightarrow$ *object* ...

Arguments:

condition	A condition object (an instance of `<condition>`).
format-string	An instance of `<string>`.

Returns:

object	An object.

Description:
This function submits a condition object to the exception handling system for delivery. The appropriate exception handler is invoked to handle the condition.

In the interactive environment, the default condition handler supplied by the read-eval-print loop will catch any condition not otherwise caught and create a "break" loop.

If the recovery protocol of the condition permits returning, and a handler returns, then the values it returns are returned from the call to `signal`. See also `error` if no recovery is permitted.

The second form of the function is a convenient interface to creating instances of `<simple-warning>` with the given message and arguments.

`error` **[function]**
Signal a non-recoverable condition

Signature:
`(error` *condition* `)`

Signature:

`(error message arg …)`

Arguments:

condition	A condition object (an instance of `<condition>`).
format-string	An instance of `<string>`.

Description:

This function signals a condition from which no recovery is possible. If the signal handler returns, then another error is signalled (a recovery protocol failure). Hence, this function never returns.

In the interactive environment, the default condition handler supplied by the read-eval-print loop will catch any condition not otherwise caught and create a "break" loop.

The second form of the function is a convenient interface to creating instances of `<simple-error>` with the given message and arguments.

`handler-case` **[special]**

Catches signalled exceptions and aborts computation

Signature:

`(handler-case expr ((class [ condition: var ] ) body …) …)`

Arguments:

expr	An expression to be evaluated
class	A subclass of `<condition>`.
var	The name for a binding in the corresponding *body* forms.
body	A sequence of expressions.

Description:

`handler-case` establishes exception handlers for the duration of the execution of *expr* .

If during the execution of *expr* , a condition is signalled, and the condition is an instance of the *class* of one of the handler clauses, then the corresponding *body* forms are executed. In this case, the value(s) of the `handler-case` is the value(s) of the last *body* form in the handler clause. Furthermore, during the execution of the *body* forms, the handlers established by this `handler-case` are no longer active (the handler taking control unwinds the dynamic stack to this point before executing the *body* forms).

If the **`condition:`** keyword is used in a clause, then the condition object is bound to a variable *var* with the *body* forms in it's scope.

The clauses are checked in the order they are given in the `handler-case`.

If *no* condition is signalled during the execution of *expr* , then the values of the entire `handler-case` are the values returned by the *expr* .

`handler-bind` **[special]**

Establishes signal handlers

Signature:

`(handler-bind (`*class function* `)` *body* `…)`

Arguments:

class	A subclass of `<condition>`.
function	A function of two arguments.
body	A sequence of expressions.

Description:

This special form arranges for the *body* forms to be evaluated in a context in which the given *function* is the handler for the set of conditions specified by the *class* and it's subclasses.

That is, if a condition is signalled during the execution of the *body* forms, and the condition object is a general instance of *class* , then the *function* is called with two arguments.

The first argument is the condition object that was signalled, which will be an instance of *class* .

The second argument is the "next handler" procedure, which is a procedure of no arguments which, when called, will arrange to invoke the next outermost handler.

The next-handler procedure may be used to be more selective in which conditions to handle, by invoking it if the *function* chooses not to handle the condition.

The *function* is called in the context of the signalling expression. In particular, the handling context established by this `handler-bind` and any interior `handler-bind`'s will still be in force. Hence, if a condition which is an instance of *class* is signalled within the execution of *function* , then *function* may be called again (and will be, if an interior `handler-bind` does not handle the condition).

Chapter 8
Debugging in RScheme

Debugging RScheme programs is in some ways both simpler and more difficult than debugging programs in a traditional language like C.

It is more difficult because the tools available are different than those that the typical C programmer is familiar with, such as **`gdb`**.

However, it is also easier, because RScheme is a much more dynamic system. Because RScheme is usually used (at least during program development) in an *interactive* mode, the developer can interact with the running program even more concretely than the most advanced features of a traditional debugger like **`gdb`**.

For example, in RScheme, you can redefine procedures interactively without having to reload your entire program.

(Unfortunately, not all kinds of objects can be redefined sans repurcussion. Class objects, for example, often have pointers to them stored in various places, such as superclass links and in functions and slots that have type restrictions on the class. Since redefining a class amounts to assigning a new value (a distinct class object) to the class variable of the given name, pointers to the old class will remain in the system, typically necessitating recompiling the definitions of many functions and any subclasses, as well as clearing out any data structures that may have instances of the old class)

In addition to the ability to interact with the target program directly, RScheme also provides some more traditional debugging facilities.

`trace` **[replcmd]**

Trace function entry and exit.

Signature:

`,(trace` *function* …`)`

Description:

This debugging command causes the named function(s) to be traced. That is, the function is reconfigured to print out it's arguments and return values when it is called.

Tracing is turned off using the `no-break` command.

`break` **[replcmd]**

Break on function entry.

Signature:

`,(break` *function* …`)`

Description:

The `break` debugging command is similar to the `trace` command, except instead of printing out the function arguments when it is called, it creates a break-level read-eval-print loop, **`BRK`**.

Within that loop, the debug command `return` may be used to cause the break'ed function call to return a specific value. The `continue` command may also be used to resume execution of the function.

The breakpoint can be removed using the `no-break` command.

`no-break` **[replcmd]**

Remove tracing or breakpoint on a function.

Signature:

`,(no-break` *function* `…)`

`abt` **[replcmd]**

Print backtrace of function calls.

Signature:

`,abt`

`bt` **[replcmd]**

Print continuation chain (stack).

Signature:

`,bt`

Chapter 9
Persistent Object Store

The persistent object store module (**rstore**) is an implementation based on the concept of pointer swizzling at page-fault time [1]. This allows the system to access databases larger than the virtual memory of the machine, and with efficient translation costs (pages are faulted in and translated on-demand, and once faulted, incur no additional run-time overhead).

The underlying storage for the object store is log-structured (and, in fact, currently no mechanism exists to automatically or incrementally compact the log). Only one file is used to store all the information in an object store, and that file is called the *backing store file*.

The interior structure of an object store – that is, the data structures which are stored – are determined completely by the application program. This facility provides access to a single value per persistent store, the *root object*, which is read using the `root-object` function and written using the `commit` function.

9.1. Creating and Accessing an Object Store

The life cycle of an object store starts off with it's creation (obviously). A newly created object store has no contents, and takes up about 600 bytes in the backing store. The initial root object is **#f**.

Thereafter, a sequence of *commit* operations take place, each establishing a (possibly) new root object. Each commit copies all dirty pages (pages modified since the last commit) to the end of the backing store file, and then writes out a *commit record* which describes the state of the object store.

An existing object store is accessed using the `open-persistent-store` function.

9.2. Commit Records

A *commit record* describes the state of the object store; each commit operation writes out a new commit record. A previous state of the store can even be accessed (in read mode) by specifying the location of the commit record to use when the store is opened.

9.3. Defining Pivot Points

It is usually necessary to allow objects within an object store to point to objects that are part of the "program", such as standard class objects (recall that everything is an object, and every object has a class. Furthermore, the representation is such that objects that are actually in the heap have actual pointers to their class objects. Hence, in order to have, for example, a vector located in the persistent store, it is necessary to somehow allow that object to point to the (single) `<vector>` class object. Furthermore, since the location in memory of built-in class objects may change from process to process, a different *object naming* scheme must be employed to resolve the references to built-in or other application objects.)

The mechanism that this module provides to allow persistent objects to refer to transient objects (but in a persistent way) is called *pivots*.

The application defines a sequence of *pivot objects* with well-known names (they are defined in collections on "indirect pages" – see `setup-indirect-page`); the system automatically resolves references to such objects when a page is loaded, and recognizes references to them when a page is written out. The next time the application open an object store, it configures the same sequence of pivot objects – then, a page that is loaded will correctly resolve to the corresponding application object.

Under normal circumstances, this module provides a pre-defined collection of pivots for the standard built-in classes such (for example) as the `<vector>` and `<pair>` class objects.

9.4. Reachability-based Persistence

When the persistence system is writing out a page, if it encounters a pointer to a transient object that is not a pivot, then something must be done – after all, a persistent page cannot *have* a pointer to a transient object except as allowed by the pivot mechanism.

The default behavior depends on what kind of object it is. Most normal data-structuring objects will be implicitly copied into the store (ie, made persistent). These are objects like lists, vectors, strings, regular application objects, etc. Because of this, the basic persistence metaphor is one of reachability-based persistence; that is, the *root object* and any object reachable from it will be made persistent.

Symbols are turned into pivots, in order to preserve the usual meaning of symbols (so that when the image is later loaded, the symbols will maintain the correct object identity.)

A class object that is encountered, however, probably denotes an application error – presumably either a failure to declare the class as a pivot, or the installation of an unexpected kind of object into the persistent data structure. (Note that class objects can't be automatically turned into pivots the way symbols can, because there isn't a well-known, fixed way of naming them.)

9.5. Functions Reference

`open-persistent-store` **[function]**

Opens an existing store

Signature:

`(open-persistent-store` *path* `)` ⇒ *pstore*

Arguments:

path An instance of `<string>` denoting the path to the backing store file.

Returns:

pstore An instance of `<persistent-store>`.

Description:

This function opens a backing store file and prepares it for access. The file is opened for writing, and is locked in exclusive mode.

If another process (or this process) already has the file open in exclusive mode, then an error is signalled.

`create-persistent-store` **[function]**

Opens an existing store

Signature:

`(create-persistent-store` *path* `)` ⇒ *pstore*

Arguments:

path An instance of `<string>` denoting the path to the backing store file.

Returns:

pstore An instance of `<persistent-store>`.

Description:

This function is much like `open-persistent-store`, except that the file is created.

If a file at *path* already exists, its contents are overwritten with the new object store's log.

`root-object` **[function]**

Return the root object for a persistent store

Signature:

`(root-object` *pstore* `)` ⇒ *object*

`commit` **[function]**

Commit changes to persistent store

Signature:

`(commit` *pstore* [*object*] `)` ⇒ *locator*

Arguments:

pstore An instance of `<persistent-store>`.

object An instance of `<object>`. Optional; default is to not change the root object.

Returns:

locator An object describing the location of the commit record (currently an instance of `<pair>`).

`setup-indirect-page` **[function]**

Configure a page of pivot objects

Signature:

`(setup-indirect-page` *pstore page-num vector* `)` ⇒

Arguments:

pstore	An instance of `<persistent-store>`.
page-num	An instance of `<fixnum>`.
vector	An instance of `<vector>`.

Description:

This function configures an "indirect page" in the persistent store's address space. An indirect page is used to hold pivot objects. Each indirect page can hold up to 64 pivot objects, which in this case are the objects referenced by the elements of *vector* .

The *page-num* is which page to configure with pivot objects. Page numbers 0-63 are reserved for use by the system[1], and page numbers 64-255 are available for the application.

If more pivot pages are needed, they may be obtained with the `alloc-indirect-pages` function.

`alloc-indirect-pages` **[function]**

Allocate some indirect page numbers

Signature:

`(alloc-indirect-pages` *pstore num-pages* `)` ⇒ *first-page*

Arguments:

pstore	An instance of `<persistent-store>`.
num-pages	An instance of `<fixnum>`.

Returns:

first-page	An instance of `<fixnum>`.

Description:

If the well-known indirect page numbers (0-255) are insufficient for an applications needs, more page numbers may be obtained using this function, which allocates blocks of indirect page numbers.

[1]Although currently only page 0 is used; it is used to store the standard pivot objects, and it is common practice to use pages 1-9 for application class object pivots.

References

[1] Paul R. Wilson and Sheetal V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. *International Workshop on Object Orientation in Operating Systems*, 364-377.

Index