
RScheme

The Implementation

Donovan Kolbly

RScheme

The Implementation

by Donovan Kolbly

Publication Number: RS-03-001

Rev	Date	Author(s)	Remarks
0.7	2006-02-14	DK	Incorporated Part II content from the old "Design and Implementation" volume.
0.6	2003-12-11	DK	Updated thread system chapter, together with other fixes and improvements.
0.5	2003-04-12	DK	Migration to new doc scheme.
0.4	1998-12-06	DK, PRW, RS, QZ	Relatively complete and widely distributed version.

Table of Contents

I	User Guide	1
1	Introduction	3
	1.1 Overview – 3	
	1.2 Conventions – 3	
2	Program Structure	5
	2.1 Lexical structure – 6	
	2.2 Naming conventions – 6	
	2.3 Expression Structure – 7	
3	Data Types	9
	3.1 Booleans – 9	
	3.2 Symbols – 9	
	3.3 Numbers – 9	
	3.4 Characters – 9	
	3.5 Strings – 9	
	3.6 Functions – 9	
	3.7 Collections – 9	
4	Flow of Control	11
5	Lists and pairs	13
	5.1 Print representation – 13	
	5.2 Side-effects vs. functional programming – 13	
	5.3 Functions – 14	
6	Object System	19
	6.1 Classes – 19	
7	Input/Output	21
	7.1 Input port constructors – 21	
8	String Manipulation	27
	8.1 String operations – 27	
	8.2 Character sets and tables – 28	
	8.3 Regular Expressions – 28	
9	Vectors	33
	9.1 Vector operations – 33	
10	Tables	35
	10.1 Table operations – 35	
11	Pathnames	39
	11.1 Functions – 39	
	11.2 Example usage – 41	
12	Exception handling	43
	12.1 Overview – 43	
	12.2 Forms – 43	
	12.3 Example usage – 45	
13	The Module System and the Module Compiler	47
	13.1 Introduction and Terminology – 47	

13.2 Organization of a Module – 47	
13.3 Compilation and its Semantics – 48	
14 Command-Line Interface	51
14.1 System Argument Processing – 51	
14.2 REPL Argument Processing – 51	
15 Compiling to C	53
15.1 Incorporation of object code using static linking – 54	
15.2 Incorporation of object code using dynamic linking – 56	
16 Thread System	57
16.1 Thread Objects – 58	
16.2 Synchronization – 59	
16.3 High Level Programming – 61	
16.4 Thread Safe Input and Output – 61	
17 Internet Connectivity	63
17.1 Server-side Procedures – 63	
18 Debugging in RScheme	67
19 Calendar	69
20 Persistent Object Store	73
20.1 Creating and Accessing an Object Store – 73	
20.2 Commit Records – 73	
20.3 Defining Pivot Points – 73	
20.4 Reachability-based Persistence – 74	
20.5 Functions Reference – 74	
21 PostgreSQL Interface	77
21.1 Initialization – 77	
21.2 Commands – 77	
21.3 Queries – 77	
21.4 Errors – 77	
21.5 Object mappings – 77	
21.6 Functions Reference – 77	
22 HTML and HTTP	83
22.1 HTML – 83	
22.2 Web Server Support – 83	
22.3 Web Client Support – 84	
22.4 CGI Support – 84	
22.5 Extended Example – 85	
23 POSIX System Call Interface	87
23.1 Time Functions – 87	
23.2 File Descriptor Functions – 91	
23.3 Filesystem Functions – 93	
23.4 <code>stat</code> Functions – 95	
23.5 Network Functions – 99	
23.6 Miscellaneous – 105	
24 Unix System Interface	107
II Technical Guide	113



User Guide

(This Page Intentionally Left Blank)

RScheme is an object-oriented, extended version of the Scheme dialect of Lisp, principally a merger of concepts from the Scheme language (see Revised⁴ Report on Scheme) and the Dylan language (see Dylan).

RScheme is freely redistributable, and offers reasonable performance despite being quite portable. Code written in RScheme can be compiled to C, and the C can then be compiled with a normal C compiler to generate machine code. 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.

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 at runtime, the interaction is fast.

1.1.1. Overview

This book is organized as reference material. No special effort is made to guide the user through programming in RScheme. For this purpose, we highly recommend Paul Wilson's Introduction to Scheme.

This book is intended as a general purpose user guide[1]. The target audience for this part is experienced programmers who wish to use RScheme as a language and system for fairly normal programming tasks. Included in this book are chapters describing the standard RScheme extensions – *packages* that ship with the distribution.

1.1.2. Conventions

In this book, we adopt certain usual typographical and stylistic conventions. These are designed to help the reader understand the meaning of a sequence of characters.

Content that is displayed like `this` is text that is likely to be typed literally into the system.

(This Page Intentionally Left Blank)

RScheme is a language with *modules*, and modules form the basis of a program's structure. That is, a program is simply a collection of modules, together with an entry point [1]. Some modules are required because the runtime system and application environment depend on their presence. Most modules, however, are optional, and are only included when necessary.

A *module* is primarily a name space for variables. That is, a module contains a particular set of mappings from names (identifiers) to variables (program definitions). A module specifies its *interface* in the form of its set of *exported variables*. Modules also specify which other modules they depend on, or *import*.

The content of a module is structured as a set of *definitions* plus a sequence of *top-level expressions*. Definitions are used to create module variables, while top-level expressions are used to provide code to be executed when the module is *loaded*.

When RScheme starts up, by default it presents an interactive environment known as the read-eval-print loop, or REPL for short. It is so called because the basic behavior of the REPL is to *read* an expression, *evaluate* it, *print* the result of the evaluation (that is, the value returned by the expression), and *loop* back to the beginning to do it again. Unlike some interactive language environments, RScheme doesn't *interpret* the expressions in order to evaluate them. Instead, the expressions are compiled to bytecodes and the compile code is executed by the usual bytecode interpreter.

The evaluation of expressions by the REPL is intended to simulate the occurrence of forms in a module.

The system has access to several modules that are suitable for normal interactive use. The default module for interactive use is the `user` module. Also available is the `usual-inlines` module.

Note that although we go to great lengths to simulate module semantics in interaction, receiving an expression at a time makes this simulation imperfect. For example, when all the definitions for a module are conceptually seen at once, top-level expressions can reference variables that are defined *later* in the source of the module. This is impractical in the interactive environment because the user expects the expressions to be executed as soon as they are entered; the user would find it disturbing if the system waited for later definitions to be encountered before the expressions are executed.

Since most definitions are simply means for defining variables whose values are the value returned by evaluating expressions, expressions are the focus of most of this syntax description.

Like most languages, expressions in RScheme return a value. Unlike many, however, (a) most constructs are expressions, including control, block, and binding constructs, and (b) expressions may

return more than one value or zero values.

Many expressions return very familiar values, although perhaps in an unfamiliar syntax:

$$(+ 1 2) \quad \Rightarrow \quad 3$$

In this book, we use the above typographical convention to indicate that the expression $(+ 1 2)$, when evaluated, returns the value 3. More precisely, the string to the left of the arrow is intended to be a prototypical expression as it might appear in a program, while the string to the right is the *print form* of the object (value) returned when that expression is evaluated.

Note that the right hand side is the print form of the resulting object(s), and *not* the expression necessary to create those objects. In other words, the left hand side denotes a program, and the right hand side denotes data. The arrow may be regarded as the “evaluates to” operator.

1.2.1. Lexical structure

Like most programming languages, RScheme has a *lexical* structure which underlies its syntactic structure.

Table 1. Token examples

Example	Description
	whitespace; used to separate tokens. All the ASCII whitespace characters are valid token delimiters (ie, space, newline, carriage return, form feed, vertical tab).
34	numbers
foo:	keywords
foo	symbols
"foo"	strings
#t #f	booleans
#key #rest	syntactic keywords
#\f	characters
(a b)	lists
#(a b)	vectors
'x	quotation shorthand
`x	quasiquote shorthand
,x	unquote shorthand

1.2.2. Naming conventions

The lexical flexibility for symbols in RScheme could be a curse if used indiscriminantly. Therefore, through experience and historical precedent, some conventions for variable names have been developed. (Note that in programs, identifiers are represented by what would be symbols in data)

Many permissible special characters are simply not used. The period and percent sign are particularly

discouraged[1]. This convention will be particularly important in RScheme because in an alternative Java-like syntax (under development), many valid scheme symbols are not identifiers (due to the use of special characters for operators).

Although RScheme is case sensitive, names are usually in all lower case with dashes to separate words.

Some specific naming conventions are used to distinguish different kinds of variables.

Table 2. Examples of identifiers following conventions

Identifier	Description
foo	normal, mutable module variable
<foo>	class variable
foo	function or special form
foo!	function or special form with side-effects (pronounced “bang”, as in <code>set!</code> which is read “set-bang”)
foo?	predicate or boolean-returning function
\$foo	constant

With respect to the notational convention for side-effects, the use of an exclamation mark (!) suffix, not all functions with side-effects are so marked. This convention applies primarily to low-level functions or functions whose primary purpose is to have side effects, such as `vector-set!` and `delq!`.

1.2.3. Expression Structure

Despite the apparent similarity between data and programs (their source representations are nearly identical), not all data are valid expressions.

Expressions are defined recursively, as is usual for programming language specifications.

1.2.3.1. Simple Expressions

1.2.3.1.1. Literals

The simplest expression is a *literal* expression. Literal expressions are also known as self-evaluating data, because the value of a literal expression is the corresponding datum. Characters, strings, numbers, booleans, and keywords are all valid literal expressions.

1	⇒	1
"foo"	⇒	"foo"
bob:	⇒	bob:

1.2.3.1.2. Variable References

The next simplest expression is the *variable reference*. An identifier is an expression which means “look up and return the current value of the variable with this name”.

1.2.3.1.3. Procedure Calls

The third kind of expression is the function call, or *combination*. Syntactically, a function call expression is an open parenthesis followed by one or more expressions followed by a close parenthesis.

There must be at least one expression between the parentheses, because the value of the first expression is the function to be called. The remaining expressions are all evaluated and their values are the arguments to the function.

The earlier example expression, $(+ 1 2)$ illustrates the three kinds of expressions, in order of execution: literal, variable reference, and procedure call.

1 and 2 are literal expressions which evaluate to the numbers 1 and 2 , respectively.

Then, $+$ is an expression which means “look up and return the value of the variable named $+$ ”.

Finally, the whole thing is a combination which means to call the procedure returned by the first expression (typically, the value of the variable named $+$ is a function which adds its arguments together) with the arguments 1 and 2 . The usual addition function would return 3 in this case.

1.2.3.2. Special Forms

The fourth kind of expression is really a large class of every other kind of expression, and are called *special forms*. Any identifier or parenthesized sequence of forms starting with an identifier is potentially a special form. The only way to tell is to know whether or not the variable named by the identifier is a special form compiler[1].

1.2.3.2.1. Binding constructs

A useful common class of special forms are those used to introduce new program variables. These are known as *binding constructs*.

`let`, `letrec`, `bind`, et.al.

1.2.3.2.2. Conditional constructs

`if`, `or`, `and`, et.al.

1.2.3.2.3. Procedural abstraction

`lambda`.

RScheme has a rich set of data types for general programming. Unlike in some hybrid lisp object systems, all data types are part of a complete class hierarchy. The RScheme data types are a superset of standard Scheme's required data types.

NOTE: Until we finish filling in all this material, please look in *R4RS* for reference material.

1.3.1. Booleans

Standard scheme `#t` and `#f`.

1.3.2. Symbols

Standard scheme, except that symbols are case sensitive.

1.3.3. Numbers

Standard scheme *required* types – float and small integers.

1.3.4. Characters

Standard scheme with extensions. You can write ascii control characters using their ascii control character names, for example `#\ff` or `#\esc`. Some characters can be entered using different names, such as `#\newLine` which is also `#\nl`.

1.3.5. Strings

Standard Scheme, except with standard C string escapes (*e.g.*, embed newline characters using `\n`)

1.3.6. Functions

Unlike some languages like Pascal, functions are regular data objects and can be stored in data structures, extracted, and the result of a computation can be invoked like any other `function[1]`

Note that the use of the term “function” does not imply anything in the sense of purely functional languages. The term is used here as in Dylan and C; functions can have side effects, and can return zero or more values (unlike C and Java which only support returning zero or one values)

1.3.7. Collections

1.3.7.1. Vectors

Standard Scheme. See chapter on vectors.

1.3.7.2. Tables

Constant-time mappings from keys to values. See chapter on tables.

All *forward control flow* in RScheme is *based* on the `if` special form. This form interprets all values which are not the `#f` object as indicating true. Hence:

```
(if #t 2 3)           ⇒ 2  
(if #f 2 3)          ⇒ 3  
(if 1 2 3)           ⇒ 2
```

In addition, RScheme implements the standard Scheme `cond`, `case[1]`, named `let`, and `do`.

RScheme also supports exceptions for *backward control flow*. See in particular `handler-case` and `signal` in the chapter on exception handling.

(This Page Intentionally Left Blank)

Pairs and the empty list are the building blocks of programs that make heavy use of Lisp list-processing style. Pairs are composed to create lists.

The terminology used here and in R4RS are slightly different, in that we consider a list to be anything that is either a pair or an empty list. A *proper list*, on the other hand, is a list that is acyclic in the cdrs and for which every cdr itself a proper list. In contrast, R4RS defines a list to be what we call a proper list.

1.5.1. Print representation

The print representation of a pair involves enclosing the contents in parentheses with a dot, or period, separating the car and cdr. This print representation is referred to as a *dotted pair*.

Since lists are common and involve repeated occurrences of pairs in cdr positions, a special case print representation is available to these kinds of pairs. In this case, the dot and recursive set of parentheses are elided. Furthermore, if the cdr is the empty list, then nothing is printed instead of “. ()”. Hence, a proper list is represented merely as the contents of the list within parentheses.

```
(list 1 2 3) ⇒ (1 2 3)
```

1.5.2. Side-effects vs. functional programming

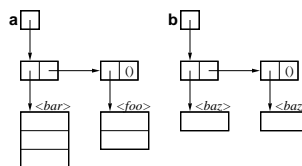
RScheme, like Scheme itself, is a language with side-effects. However, a “mostly functional” style is encouraged.

Consider the two short lists, constructed as follows:

```
(set! a (list (make-bar 1) (make-foo 2)))
(set! b (list (make-baz 3) (make-baz 4)))
```

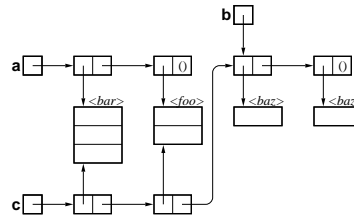
The resulting structure is as follows:

Figure 1. List structure after side-effects.



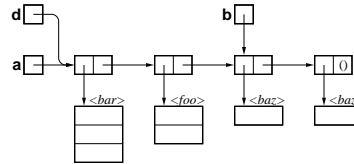
Call the return value of `(append a b)` `c`, and you have the following circumstance. Note that the `c` and `b` lists *share structure*

Figure 2. List after append



On the other hand, if *d* is the result of calling `(append! a b)`, then the following figure applies. Note that the *a* and *d* lists share structure in this case, in addition to the same kind of shared structure with the *b* list.

Figure 3. List after append!



1.5.3. Functions

cons		function
	Construct a new pair from two objects	
	<code>(cons a b)</code>	$\Rightarrow p$
	Arguments	
	<i>a</i> An instance of <object>	
	<i>b</i> An instance of <object>	
	Return Values	
	<i>p</i> An instance of <pair>	
	Description	
	This is the standard lisp constructor. <i>a</i> will be the <i>car</i> of the resulting pair, while <i>b</i> will be the <i>cdr</i> .	
car		function
	Get the car part of a pair.	
	<code>(car p)</code>	$\Rightarrow a$
	Arguments	
	<i>p</i> An instance of <pair>	
	Return Values	
	<i>a</i> An instance of <object>	
	Description	
	This is one of the two standard lisp accessors.	
cdr		function
	Get the cdr part of a pair.	
	<code>(cdr p)</code>	$\Rightarrow b$
	Arguments	
	<i>p</i> An instance of <pair>	
	Return Values	
	<i>b</i> An instance of <object>	
	Description	
	This is the other of the two standard lisp accessors.	
length		function
	Return the length of a list.	

	(length <i>l</i>)	\Rightarrow <i>n</i>	
Arguments	<i>l</i>	An instance of <list>	
Return Values	<i>n</i>	An instance of <fixnum>	
Description	Computes the length of a well-structured list. Signals an error if <i>l</i> is an acyclic improper list. Never returns if <i>l</i> is cyclic.[1]		
append	Appends some lists		function
	(append <i>l</i>)	\Rightarrow <i>b</i>	
Arguments	<i>l</i>	An instance of <list>	
Return Values	<i>b</i>	An instance of <list>	
Description	Appends some lists (possibly zero of them) into one big list. The resulting list shares structure with the final given list.		
	NOTE: This means the all lists except the last are copied, so the cost in time and space of append is proportional to the length of all the lists except the last, <i>even if the last list is empty</i>		
append!	Append some lists using side-effects.		function
	(append! <i>l</i>)	\Rightarrow <i>b</i>	
Arguments	<i>l</i>	An instance of <list>	
Return Values	<i>b</i>	An instance of <list>	
Description	Like append , but side effects the tails of all but the last list in order to concatenate them. Note however that append! must still traverse each pair in the lists except the last, so this function still costs time proportional to the length of the lists (except the last), though the space cost is zero.		
pair?	Test if an object is a pair.		function
	(pair? <i>x</i>)	\Rightarrow <i>q</i>	
Arguments	<i>x</i>	An instance of <object>	
Return Values	<i>q</i>	An instance of <boolean>	
Description	Returns true if and only if <i>x</i> is a pair.		
null?	Test if an object is the empty list.		function
	(null? <i>x</i>)	\Rightarrow <i>q</i>	
Arguments	<i>x</i>	An instance of <object>	
Return Values	<i>q</i>	An instance of <boolean>	

	Description		
	Returns true if and only if x is the empty list.		
list?			function
	Test if an object is a proper list.		
	(list? x)	\Rightarrow	q
	Arguments		
	x An instance of <object>		
	Return Values		
	q An instance of <boolean>		
	Description		
	Returns true if and only if x is a proper list – a sequence of zero or more cdr-linked pairs terminated with the empty list.		
	In particular, list? does finish and returns #f on cyclic lists. As a result, this function is fairly expensive (quadratic on long lists, although optimized for short ones).		
set-car!			function
	Set the car of a pair.		
	(set-car! p a)	\Rightarrow	
	Arguments		
	p An instance of <pair>		
	a An instance of <object>		
	Description		
	Side effects the given pair, p , so that its car is a . Returns no values.		
set-cdr!			function
	Set the cdr of a pair.		
	(set-cdr! p b)	\Rightarrow	
	Arguments		
	p An instance of <pair>		
	b An instance of <object>		
	Description		
	Side effects the given pair, p , so that its cdr is b . Returns no values.		
reverse			function
	Returns the reverse of a list.		
	(reverse l)	\Rightarrow	r
	Arguments		
	l An instance of <list>		
	Return Values		
	r An instance of <list>		
	Description		
	Returns a new list which is the reverse of the given list. Signals an error if l is not a proper list.		
reverse!			function
	Reverses a list using side effects.		
	(reverse! l)	\Rightarrow	r
	Arguments		
	l An instance of <list>		
	Return Values		
	r An instance of <list>		

Description

Accomplishes something list **reverse**, but does so by side-effecting *l*. Returns the new head of the list, which was its tail. Signals an error if *l* is not a proper list.

list-tail**function**

Returns the *k*-th cdr of a list.

(**list-tail** *l k*)

⇒ *r*

Arguments

l An instance of <list>

k An instance of <fixnum>

Return Values

r An instance of <list>

Description

Returns the *k*-th cdr of a given list. Signals an error if the list is not long enough.

(This Page Intentionally Left Blank)

By Robert Strandh

RScheme is a fully object-oriented language in that each RScheme object is an instance of some class. Contrary to common languages such as C++ or Java in which methods are associated with classes, the RScheme object system is based on generic functions much in the spirit of Dylan, which in turn is based on the Common Lisp object system (CLOS).

Every value in RScheme is an object. That is, RScheme is "objects all the way down" rather than being a hybrid language like C++ or most Lisp object systems--the built-in R4RS Scheme data types are all classes in the class hierarchy, as they should be. Currently, only a simple object system with single inheritance and single dispatching is supported. The object system is approximately a subset of TinyCLOS, with some Dylan-like extensions.

1.6.1. **Classes**

As mentioned above, every object in RScheme is an instance of a class. For example, 123 and 12.3 are both instances of the class named `<number>`, and "hello" and "hi there" are both instances of the class `<string>`.

Classes are organized into a hierarchy. The top class of the hierarchy is called `<object>`. If some class A is positioned immediately above some other class B in the hierarchy then we say that A is the immediate superclass of B, and B is the immediate subclass of A. All classes above A in the hierarchy are called (indirect) superclasses of B, and all classes below B in the hierarchy are called (indirect) subclasses of A. The term subclass is used to mean direct or indirect subclass, and the term superclass is used to mean direct or indirect superclass. A direct subclass B is said to inherit from its direct superclass A, because instances of B automatically contain everything that instances of A contain, plus information specific to B. An instance of a class B is also said to be an instance of every superclass (direct or indirect) of B. An object, instance of B but not of any subclass of B, is said to be a direct instance of B and an (indirect) instance of every superclass of B.

In the RScheme class hierarchy, each class has exactly one direct superclass (except `<object>` which has none). Another term for this kind of class organization is single inheritance.

We mentioned above that 123 and 12.3 are both instances of the class `<number>`. To be more specific, 123 is a direct instance of the class named `<fixnum>` which is a subclass of `<number>`, and 12.3 is a direct instance of the class named `<double-float>` which is also a subclass of `<number>`.

In some languages, for instance C++ and Java, classes exist only at compile time, and thus cannot be manipulated as objects at runtime. In other languages, for instance Smalltalk and RScheme, classes

are themselves objects (and thus instances of some classes) in the same way numbers and strings are objects. If classes can be manipulated in the same way other objects are manipulated, then we say that classes are first-class. If an object is first-class, it can be given as values to a variable, passed as an argument to a function, and returned as values of a function invocation.

Since classes are first-class objects, they do not necessarily have names. It is convenient, however, to be able to refer to classes by some name. Indeed, above we already talked about the classes named `<object>`, `<number>`, and `<string>`. In reality, these names are just names of ordinary global variables whose values happen to be classes. As a convention, we use angle brackets around variable names whose values are classes.

Most classes in RScheme are instances of the class named `<<standard-class>>`. Classes of class objects are called metaclasses. Names of metaclasses are surrounded by double angle brackets as a convention.

The most convenient way of creating a class and giving it a name, is to use the macro `define-class`. Here is the syntax:

define-class		special
	Defines a new class	
	(define-class <i>name superclass class-opt slot-spec</i>)	
Arguments		
	<i>name</i>	The name of the class being defined
	<i>superclass</i>	The parent class
	<i>class-opt</i>	A class option
	<i>slot-spec</i>	A slot specification
Description		
	This special form defines a new, immutable top-level variable with the given <i>name</i> , whose value is the class object described by the superclass, class options, slot specifications.	

The input/output facilities of RScheme use the concept of a "port". A port is a first-class RScheme object that is passed (implicitly or explicitly) to the input/output functions.

In RScheme, ports come in two varieties, namely input ports which are instances of <input-port> and output ports that are instances of <output-port> [we don't have input output ports?].

While some ports exist when RScheme is started, most have to be created explicitly by an open operation. The open operations thus create ports from other types of objects such as strings that represent file names, or just strings that are themselves used for input or output.

RScheme ports usually buffer input/output for efficiency. This means that you might not see the effect of an output operation immediately. Instead the output may be stored internally in a buffer associated with the port and only written to the underlying device when the buffer is full, or as a result of an explicit flush operation.

When all desired input/output has been accomplished from/to a particular port, you should close the port. Closing the port will free up resources of the operating system and flush the contents of the port.

1.7.1. Input port constructors

input-port? **function**

Checks whether an object is a input port.

(*input-port?* *obj*) ⇒ *bool*

Arguments

obj any RScheme object

Return Values

bool #t if the argument is an input port and #f otherwise

Description

This function tests whether the object given as argument is an instance of the class <input-port>. If so, it returns #t. Otherwise it returns #f.

current-input-port **function**

Returns the current default input port

(*current-input-port*) ⇒ *input-port*

Return Values

input-port An instance of <input-port>

Description

This function returns the input port used by default in some input operations such as read, read-char, peek-char.

open-input-file	<p>Creates an input port from a file name (open-input-file <i>string</i>) ⇒ <i>input-port</i></p> <p>Arguments</p> <p><i>string</i> An instance of <string> representing a file name.</p> <p>Return Values</p> <p><i>input-port</i> An instance of <input-port></p> <p>Description</p> <p>This function creates a new input port from a file name. This port can then subsequently be used in input operations such as read, read-char, and peek-char.</p>	function
with-input-from-file	<p>Redirect input from file (with-input-from-file <i>string proc</i>) ⇒</p> <p>Arguments</p> <p><i>string</i> An instance of <string> representing a file name</p> <p><i>proc</i> A function of one parameter</p> <p>Description</p> <p>This function creates a new input port from the file name in the string as in open-input-file. It then calls the procedure with the new port. If the procedure returns, then the port will be closed as with close-input-port. If the procedure does not return, the port will not be closed until and unless it can be shown that the port cannot be used for any input operations.</p>	function
open-input-process	<p>Open a port to the output from a shell pipeline. (open-input-process <i>shell</i>) ⇒ <i>port</i></p> <p>Arguments</p> <p><i>shell</i> An instance of <string></p> <p>Return Values</p> <p><i>port</i> An instance of <input-port></p> <p>Description</p> <p>Creates a subprocess for executing <i>shell</i> via the OS's shell interpreter, and returns an input port which reads from the output of that subprocess. The close-input-port method synchronizes with the subprocess.</p> <p>Note that this procedure is named for the kind of object you get back (an input port), and not for the behavior that shell process is expected to exhibit (ie, output).</p>	function
close-input-port	<p>Close an input port (close-input-port <i>port</i>) ⇒</p> <p>Arguments</p> <p><i>port</i> An instance of <input-port></p> <p>Description</p> <p>This function closes the input port given as argument.</p>	function
read-char	<p>Read a character (read-char) ⇒ <i>char</i></p> <p>Arguments</p> <p><i>input-port</i> An instance of <input-port>. Default is the current input port.</p>	function

	Return Values		
	<i>char</i>	A character object or the EOF object if the input port is at end of file.	
	Description		
		This function reads one character from the input port given, or from the current input port if no argument is given. It return the character read as an RScheme ascii character object. If there are no more characters to read from the port then the EOF object is returned (the only one for which eof-object? return #t).	
peek-char			function
	Read a character without consuming it		
	(peek-char)		⇒ <i>char</i>
	Arguments		
	<i>input-port</i>	An instance of <input-port>. Default is the current input port	
	Return Values		
	<i>char</i>	A character object or the EOF object if the input port is at end of file.	
	Description		
		This function is similar to read-char in that it reads one character from the input port given, or from the current input port if no argument is given. However, the character read is not consumed and is returned again by the next call to read-char or to peek-char. If there are no more characters to read from the port then the EOF object is returned.	
char-ready?			function
	Check whether it is safe to read a character		
	(char-ready?)		⇒ <i>bool</i>
	Arguments		
	<i>input-port</i>	An instance of <input-port>. Default is the current input port	
	Return Values		
	<i>bool</i>	#t or #f	
	Description		
		This function returns #t if a subsequent call to read-char on the port is guaranteed not to block. Otherwise it returns #f.	
read			function
	Read an S-expression		
	(read)		⇒ <i>expr</i>
	Arguments		
	<i>input-port</i>	An instance of <input-port>. Default is the current input port.	
	Return Values		
	<i>expr</i>	The internal representation of the next expression on the port.	
	Description		
		This function is the RScheme expression parser. It converts a sequence of characters on port to the internal representation of an expression. The value returned can be as simple as a character and as complicated as a vector of almost arbitrary RScheme objects. The object returned depends on the sequence of characters, which must obey the syntactic conventions of RScheme.	
output-port?			function
	Checks whether an object is a output port.		
	(output-port? obj)		⇒ <i>bool</i>
	Arguments		
	<i>obj</i>	Any RScheme object	

	<p>Return Values</p> <p><i>bool</i> #<i>t</i> if the argument is an output port and #<i>f</i> otherwise</p> <p>Description</p> <p>This function tests whether the object given as argument is an instance of the class <output-port>. If so, it returns #<i>t</i>. Otherwise it returns #<i>f</i>.</p>	
current-output-port	<p>Returns the current default output port</p> <p>(current-output-port) ⇒ <i>output-port</i></p>	function
	<p>Return Values</p> <p><i>output-port</i> An instance of <output-port></p> <p>Description</p> <p>This function returns the output port used by default in some output operations such as write, write-char, and display.</p>	
open-output-file	<p>Creates an output port from a file name</p> <p>(open-output-file <i>string</i>) ⇒ <i>output-port</i></p>	function
	<p>Arguments</p> <p><i>string</i> An instance of <string> representing a file name.</p> <p>Return Values</p> <p><i>output-port</i> An instance of <output-port>.</p> <p>Description</p> <p>This function creates a new output port from a file name. This port can then subsequently be used in output operations such as write, write-char, and display.</p>	
with-output-to-file	<p>Redirect output to a file</p> <p>(with-output-to-file <i>string</i> <i>thunk</i>) ⇒</p>	function
	<p>Arguments</p> <p><i>string</i> An instance of <string> representing a file name</p> <p><i>thunk</i> A procedure of no arguments</p> <p>Description</p> <p>This procedure creates a new output port referring to the system file <i>string</i>, as does the procedure open-output-file. It then calls the given <i>thunk</i> in a dynamic context such that the current output port is the new output port. If the procedure returns, then the port will be closed as with close-output-port. If the procedure does not return, the port will not be closed until and unless it can be shown (as by the reachability analysis done by the garbage collector) that the port cannot be used for any output operations.</p>	
close-output-port	<p>Close an output port</p> <p>(close-output-port <i>port</i>) ⇒</p>	function
	<p>Arguments</p> <p><i>port</i> An instance of <output-port></p> <p>Description</p> <p>Closes the output port <i>port</i>. Signals an error if the output port has already closed. Certain implementations may also signal an error if something goes wrong, like buffered data could not be written to the media.</p>	
open-output-process	<p>Open a port to the input to a shell pipeline.</p> <p>(open-output-process <i>shell</i>) ⇒ <i>port</i></p>	function

Arguments

shell An instance of <string>

Return Values

port An instance of <output-port>

Description

NOTE: This procedure is named for the kind of object you get back (an input port), and not for the behavior that shell process is expected to exhibit (ie, output).

(This Page Intentionally Left Blank)

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 4. Sample String

```

1  2  B u c k l e   m y   s h o e
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

```

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.

1.8.1. String operations

`string-search`

function

Search a string

`(string-search string item)`

\Rightarrow *index*

Arguments

string

An instance of `<string>`.

item

An item for which to search, a `<char>` or `<string>`.

offset

The offset in *string* at which to start searching. Defaults to 0.

Return Values

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 or a character.

`string-split`

function

Split a string into substrings

`(string-split string delim)`

\Rightarrow *list*

Arguments

string

An instance of `<string>`.

delim

The delimitation on which to break *string*. May be a `<string>`, `<char>`, or a `<function>`.

Return Values

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 *delim* is a procedure, *delim* 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

```
(string-join delim list)                => string
```

Arguments

delim The delimiter to separate elements of *list*.

list A proper list of <string> instances.

Return Values

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"
```

1.8.2. 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 to match characters.

1.8.3. Regular Expressions

RScheme comes with a regular expression facility, accessible as the `regex` module. This module provides a regular expression compiler; regular expressions in a structured scheme-like syntax are compiled into functions which know how to search for their pattern.

Compiling a regular expression is relatively expensive, so it is best to factor out the operation whenever convenient.

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:

Table 3. Regular Expression Operators

form	example
<char>	#\a
<string>	"foo"
<symbol>	space
eos	eos
(or P1 ...)	(or #\a space)
(seq P1 ...)	(seq #\a #\b)
(+ P)	(+ space)
(* P)	(* alpha)
(? P)	(? #\x)
(range C1 C2)	(range #\a #\z)
(not P)	(not space)
(save P)	(save (+ (not #\,)))
(let N P)	(let id (+ alpha))
(prefix P)	(prefix #\a)
(suffix P)	(suffix #\.)
(entire P)	(entire "foo")

1.8.3.1. Functions

reg-expr->proc

function

Compiles a regular expression

(reg-expr->proc *expr*)

⇒ *proc list*

Arguments

expr A regular expression.

Return Values

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, ... [really, `save` is the appropriate choice for unnamed `let`'s]

```
(reg-expr->proc '(seq (+ space) (let w (+ alpha)))) #f#(<procedure>*)
```

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 q (reg-expr->proc '(seq (+ #\*) (save (+ alpha))))
(define str "...**hello there")
(q str)                               => 3           => 11           => "hello"
(substring str 3 11)                   => "**hello"
(bind ((s e w (q str)) w)              => "hello"
```

unformat->proc**function**

Constructs an “unformat” procedure

(**unformat->proc** *format-string*) \Rightarrow *proc*

Arguments

format-string A format string.

anywhere? An optional <boolean>. Default value is #f.

Return Values

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 its 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 signaled 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 its given string of the substring that matched. Otherwise, by default, the returned procedure will require an exact match between the format string and its 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*.

1.8.3.2. Examples

The following examples illustrate the basic behavior of the regular expression module.

Example 1. Regular Expression Examples

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

(This Page Intentionally Left Blank)

The `<vector>` class corresponds to the 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 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, whereas it is a linear-time operation for vectors.

Vectors read like lists, except with a `#` at the beginning. For example,

```
#(1 2 3)
#()
```

1.9.1. Vector operations

<code>vector-append</code>		function
	Concatenates a sequence of vectors (<code>vector-append</code> <i>vector</i>)	\Rightarrow <i>result-vector</i>
	Arguments	
	<i>vector</i> Instances of <code><vector></code> .	
	Return Values	
	<i>result-vector</i> An instance of <code><vector></code> .	
	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.	
<code>subvector</code>		function
	Creates a subvector (<code>subvector</code> <i>vec</i> <i>start</i>)	\Rightarrow <i>result</i>
	Arguments	
	<i>vec</i> An instance of <code><vector></code> .	
	<i>start</i> An instance of <code><fixnum></code> .	

	<i>end</i>	An instance of <fixnum>. Optional. Default value is the length of the vector.	
	Return Values		
	<i>result</i>	An instance of <vector>.	
	Description	This function is to vectors what substring is for strings.	
vector-map			function
	Maps a function over a vector		
	(vector-map <i>proc</i> <i>vec</i>)	\Rightarrow <i>result</i>	
	Arguments		
	<i>proc</i>	An instance of <function>.	
	<i>vec</i>	An instance of <vector>.	
	Return Values		
	<i>result</i>	An instance of <vector>.	
	Description	This function is to vectors what map is to lists.	
		The length of <i>result</i> is the <i>minimum</i> of the lengths of the <i>vec</i> arguments. Hence, if one of <i>vec</i> is an empty vector (having 0 length), then an empty vector is returned and <i>proc</i> is never called.	
		<pre>(define x '(1 2 3)) (define y '(10 20 30)) (vector-map + x y) \Rightarrow #(11 22 33) (define z '(8 9)) (vector-map + y z) \Rightarrow #(18 19)</pre>	
vector-for-each			function
	Applies a function to elements of a vector		
	(vector-for-each <i>proc</i> <i>vector</i>)	\Rightarrow <i>object ...</i>	
	Arguments		
	<i>proc</i>	An instance of <function>.	
	<i>vector</i>	Instances of <vector>.	
	Return Values		
	<i>object</i>	Instances of <object>.	
	Description	This function is to vectors what for-each is to lists.	
		In particular, the procedure <i>proc</i> is called with corresponding vector elements, in order (up to the length of the shortest <i>vector</i> .) Also, vector-for-each returns whatever the last call to <i>proc</i> returns[1].	

RScheme provides an implementation of hash tables, both generic and specialized for certain kinds of keys. The hash table facilities are provided by the `tables` module.

```
top[0]=>(use tables)
top[1]=>(define t (make-table))
value := t
top[2]=>t
value := #[<generic-table> @000f_005b]
```

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.

1.10.1. Table operations

`make-table`

function

Constructs a new hash table

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

\Rightarrow `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.

Return Values

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
<code>eq?</code>	<code>symbol->hash</code>	<code><symbol-table></code>
<code>eq?</code>	<code>integer->hash</code>	<code><hash-integer-table></code>
<code>eq?</code>	<code>identity</code>	<code><integer-table></code>
<code>string=?</code>	<code>string->hash</code>	<code><string-table></code>
<code>string-ci=?</code>	<code>string-ci->hash</code>	<code><string-ci-table></code>
<code>eq?</code>	<i>any</i>	<code><eq-table></code>

`table-lookup`

generic

Searches a table for the given key

`(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.

Return Values*value*

The value associated with given key, or `#f` if the key is not present in the table[1].

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.

The following 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.

Methods

`(table-lookup generic-table key)`

\Rightarrow *value*

`table-key-present?`

generic

Checks a table for the presence of a given key

`(table-key-present? table key)`

\Rightarrow *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.

Return Values*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 <code>table-lookup</code> , 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 <code>#f</code> as a value.	
<code>table-insert!</code>	Associates a key with a value in a table (<code>table-insert! table key value</code>)	\Rightarrow <i>prev-value</i> generic
	Arguments	
	<i>table</i> An instance of <code><table></code> to be extended	
	<i>key</i> The key to insert.	
	<i>value</i> The value to be associated with the key.	
	Return Values	
	<i>prev-value</i> The value previously associated with given key, or <code>#f</code> 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.	
<code>table-remove!</code>	Removes a given key from a table (<code>table-remove! table key</code>)	\Rightarrow <i>value</i> generic
	Arguments	
	<i>table</i> An instance of <code><table></code> to be modified.	
	<i>key</i> The key to remove	
	Return Values	
	<i>value</i> The value that was associated with <i>key</i> , or <code>#f</code> 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 <code>#f</code> is returned. No error is signalled in this case.	
<code>table-size</code>	Computes the number of keys in the table. (<code>table-size table</code>)	\Rightarrow <i>integer</i> generic
	Arguments	
	<i>table</i> An instance of <code><table></code> .	
	Return Values	
	<i>integer</i> 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 <code>key-sequence</code> and <code>value-sequence</code> .	
<code>key-sequence</code>	Returns a sequence of table keys (<code>key-sequence table</code>)	\Rightarrow <i>list</i> generic
	Arguments	

	<i>table</i>	An instance of <table>	
Return Values	<i>list</i>	An instance of <list>	
Description	<p>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.</p> <p>The order of returned keys in the list is not specified, <i>except</i> to say that it is the same order as the values returned by value-sequence.</p>		
value-sequence			generic
	Returns a sequence of table values (value-sequence <i>table</i>)		⇒ <i>sequence</i>
Arguments	<i>table</i>	An instance of <table>	
Return Values	<i>sequence</i>	An instance of <list>	
Description	<p>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.</p> <p>In the current implementation, the returned sequence is a list. However, future versions may return a vector instead.</p> <p>The order of returned value in the sequence is not specified, <i>except</i> to say that it is the same order as the values returned by key-sequence.</p>		
table-for-each			function
	Applies a procedure to entries in a hash table (table-for-each <i>table proc</i>)		⇒
Arguments	<i>table</i>	An instance of <hash-table>.	
	<i>proc</i>	An instance of <function>.	
Description	<p>This function applies a given procedure to the entries in a hash table. The procedure <i>proc</i> must accept 3 arguments, which will be, for each entry in the hash table, the hash code, the key, and the associated value.</p> <p>The number of calls to the procedure will equal the table size (modulo the following statement:)</p> <p>The behavior of the iteration is <i>not</i> specified if the table is updated while the iteration is taking place.</p>		

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 separates 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.

1.11.1. Functions

<code>string->file</code>		function
	Creates a pathname object naming a file (<code>string->file string</code>)	\Rightarrow <i>filename</i>
	Arguments	
	<i>string</i> An instance of <string>.	
	Return Values	
	<i>filename</i> An instance of <file-name>.	
	Description	
	This function parses the given string in canonical filename notation (unix style) into a structured <file-name> instance.	
<code>string->dir</code>		function
	Creates a pathname object naming a directory (<code>string->dir string</code>)	\Rightarrow <i>dirname</i>
	Arguments	
	<i>string</i> An instance of <string>.	
	Return Values	
	<i>dir-name</i> 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

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

Arguments

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

Return Values

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

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

Arguments

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

Return Values

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

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

Arguments

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

Return Values

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

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

Arguments

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

extension A string.

Return Values

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 its 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

`(append-path dirname filename)` \Rightarrow *new-filename*

Arguments

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

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

Return Values

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.

`(append-dirs dir1 dir2)` \Rightarrow *dir3*

Arguments

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

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

Return Values

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

Description

This function is analogous to **append-path**, but works on a directory as its 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

`(pathname->os-path pathname)` \Rightarrow *string*

Arguments

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

Return Values

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.

1.11.2. Example usage

The following examples illustrate the basic usage.

Example 2. Filename Examples

```
(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/}
```

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

1.12.1. Overview

Exception handling is a language feature available in RScheme which supports to the development of more robust programs by providing a uniform mechanism for error dispatching. This facility is also used by RScheme's built-in error recognition as well, so error management is uniform across built-in and application-level error conditions.

1.12.2. Forms

Error conditions are injected into the dispatching system by the `signal` or `error` function. These two functions operate similarly; the only difference is that `error` never returns, whereas `signal` may return, depending on the handler that is in place.

On the receiving end, errors are *caught* by functions and bodies introduced by the `handler-case` and `handler-bind` forms. The former is more commonly used, but imposes the semantics of *terminating handlers*. `handler-bind` is more flexible, and can support *calling semantics* for its handlers.

<code>signal</code>	<code>function</code>
Signal a condition (<code>signal condition</code>) (<code>signal message arg</code>)	\Rightarrow <code>object ...</code> \Rightarrow <code>object ...</code>
Arguments	
<i>condition</i>	A condition object (an instance of <code><condition></code>).
<i>format-string</i>	An instance of <code><string></code> .
Return Values	
<i>object</i>	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 <code>signal</code> . See also <code>error</code> 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

(**error** *condition*)

(**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

(**handler-case** *expr class 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 its 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

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

Arguments

class A subclass of `<condition>`.

(This Page Intentionally Left Blank)

1.13.1. Introduction and Terminology

It is good practice to divide a large program into smaller units that can be independently understood. Such units are called modules. RScheme has full support for modules.

Traditionally, there are two different purposes of modules. The first and least interesting purpose is to function as a unit of compilation. Early versions of Fortran already supported separate compilation of this type of module. The second and most interesting purpose of a module is to encapsulate related data and functions with a restricted view from the outside. Typically, modules are used to hide implementation details of some abstract data type that can only be used by other modules (called the client modules) through a well-defined set of functions called the interface of the abstract data type.

In RScheme a module is both a unit of separate compilation and a unit of encapsulation. Currently, only the off-line compiler (or module compiler) `rsc` can create a module.

You may think of a module as an augmented top-level environment mapping names to variables, macros, or special forms (collectively called *bindings*). Bindings are not first-class objects so you cannot manipulate them the way you manipulate other RScheme objects such as numbers and strings. In addition to the collection of bindings, a module contains information about which bindings are visible from the outside (exported) and which bindings are purely local to the module. Finally, a module contains information about what other modules are needed by the module (called the imported modules).

When a module *M* imports a module *N*, the top-level environment of *M* is augmented with the name-to-binding mappings of *N* that are exported. RScheme provides functionality to give different names of these bindings from those used by *N*, so that both *M* and *N* share the same bindings, but under different names. This renaming is useful when you need to avoid name clashes between modules.

The collection of modules together with the imports relation in a program form a directed acyclic graph. In other words, it is not possible for modules to be mutually imported.

1.13.2. Organization of a Module

A module is organized as a collection of source files (extension `.scm`) and a module control file (extension `.mcf`). The source files are logically concatenated by the module compiler so that there is no encapsulation between different source files in a module. The module control file indicates the order of the concatenation of the source files. This order may be important, since a binding, such as a macro, may be created in one file and used in another.[1]

The source files are ordinary RScheme files that can be understood by the interactive on-line compiler. Typically, you would use the procedure `load` to add the contents of these source files to your interactive RScheme environment during the development phase. Loading the source files does not involve the module compiler at all. Instead the on-line interactive compiler translates the source to some internal form (e.g. bytecodes) that can be executed relatively fast. The on-line interactive compiler is faster than the off-line module compiler, so that in a phase of frequent modifications to source code you can rapidly test new versions of your code. However, the off-line compiler produced much faster executable code, so that when you are reasonably sure that your program is bug-free, you typically use the off-line module compiler on your code in order to produce fast executable code.

1.13.3. Compilation and its Semantics

We have already used the concept of loading above. The compiler introduces some other concepts that need to be understood in order for the compilation process to make sense.

Compiling a module involves translating the source code in the source files to a different format that is no longer in the form of source code. The off-line module compiler puts the result of this translation in a file called the module image file (extension `.mif`).

The module index (`.mx`) file contains various names that are important to the linker, in particular module entry points and a list of dynamically linked (extension `.so`) files. The module index file and the module image file are both logically part of the "object code" generated by the compiler and should always be manipulated together. In particular, they should both be installed in the same directory.

The contents of this module image file can then be added to your interactive RScheme environment in a way similar to loading described above. Conceptually, this process is divided into two steps, the link step and the import step. Linking simply takes the module image file and incorporates it into the interactive RScheme environment without making any connection between the linked module and the current module. Linking is just the extension, at runtime, of the set of modules present in the heap; the system starts out with a set of default modules already present and linked. Importing, on the other hand, simply binds names in the current module to the exported variables of the imported module.

In the interactive RScheme environment, you typically use the command `(use module)` to import a module, where `module` is the name (in the form of a symbol) of a module. The module image file is formed by adding a `.mif` extension to the string form of the symbol. Recall that forms that start with a comma are conceptually executed outside the interactive environment. The command `(use foo)` will first check whether the module `foo` has already been linked. If not, then it tries to find the module image file and link it. If it cannot find the module image file, an error message is given. If the module is already linked, or if the module file was found and successfully linked, the `(use foo)` command will proceed to the population step and add the exported top-level bindings to the interactive RScheme environment.

Conceptually, the meaning of the linking step is similar to the meaning of loading source. It is during the linking step that top-level forms in the source code (i.e. the `.scm` files that were mentioned during compilation) are conceptually evaluated in order. We say "conceptually" because the compiler is free to replace this top-level evaluation by something more efficient provided that the effect in the interactive RScheme environment is the same. The most important such transformation that the compiler does is to convert each lambda expression to machine code that has the same effect as the

lambda expression itself, except that it is much faster.

use

replcmd

Import module to current environment.
(**unquote** *module*)

Description

The **use** REPL command is used to import the exported variables of other modules. *module* gives the name of the module whose variables are to be bound in the current environment.

If the named module is not currently linked, the system will attempt to locate the corresponding mif file and link the module, before importing it.

rsc

unix

Compile a module.

Description

This **unix** command compiles a module, using the control and component instructions in the module control file given by *file*. The module's files are taken to be relative to *file*, not the current working directory.

(This Page Intentionally Left Blank)

The RScheme executable program in a unix-like environment accepts several switches, options, and arguments.

When the system starts up, the first thing it does is load the *system image*. When RScheme is configured, a default location for the system image is defined, and that location is compiled into the executable image. In particular, the image is sought in `$INSTALL_DIR/resource/system.img`, where `$INSTALL_DIR` is the installation directory for RScheme.

After loading the system image, the internal start function is called which executes the initialization procedures and then calls the current `main` function with the command line arguments in a list.

1.14.1. System Argument Processing

The C initialization code and the built in (system) start function interpret some but not all command arguments. The command-line arguments understood by the C initialization code will be shared by all “executable” images.

The following flags are processed by the system initialization code:

Table 4. Command-line Interface Flags defined for the RScheme executable environment. See also the following table.

flag	meaning
- -version	Print out the version and exit.
-image	Use an alternate system image file.
-q	Suppress output of greetings.
-qimage	Use an alternate system image file and suppress.
-script	Suppress output of greetings and set <i>script mode</i> .
-bcitrace	Turn on tracing of bytecodes; the system must have been configured with the - -enable-debug flag to make use of this flag.
-abt	Turn on apply backtrace tracing from the start. Useful in some cases when turning it on at runtime (using ,fg-abt) doesn't work.

1.14.2. REPL Argument Processing

As built, the `main` function comes from the `repl` module, which interprets the arguments as flags and names of files to be loaded. The following flags are understood:

Table 5. RScheme Shell Flags

flag	meaning
<code>-c path</code>	Save bootable image.
<code>-c.repl path</code>	Save bootable image with REPL's <code>main</code> as the main function.
<code>--</code>	Interpret remaining arguments as application arguments.
<code>-exit</code>	Exit successfully.
<code>-e expr</code>	Evaluate expression <i>expr</i> .
<code>-m [module=]file</code>	Link module <i>module</i> from mif at <i>file</i> . If <i>module</i> is not specified, the name will be the base name in <i>file</i> .
<code>+module</code>	Import module <i>module</i>

The default behavior is to interpret each of the other command-line arguments as the name of a file to load.

[unify this...] When we start the RScheme system, we can use command-line arguments to incorporate compiled modules and to execute expressions, and possibly to save the resulting image to an image file. The general syntax of the `rs` command is described above.

Each module-or-eval-argument is handled in the order given on the command line and can be `-m foo.mif`, `+foo`, or `-e expr`. Using `-m foo.mif` means that the code for the module `foo` is loaded into RScheme, but the exported variables in `foo` are not made visible until a `,(use foo)` command is executed. Using `+foo` means that the module is loaded in the same way as with `-m foo.mif`, but in addition the exported variables in `foo` are made visible to the current module as if `,(use foo)` had been typed. Using `-e expr` means evaluating an RScheme expression. Thus `rs +foo` is roughly equivalent to `rs -m foo.mif -e "(use foo)"`, but `-e` can of course be used to evaluate any valid RScheme expression.

Finally, the argument `-c file.img` saves the RScheme image resulting from the other arguments.

By Robert Strandh

The normal mode of operation (called strategy) of the off-line module compiler is to generate bytecodes. But there is another strategy which consists of generating C code which then is compiled with the ordinary C compiler to generate object code.

In this section, we describe the steps necessary to take in order for `rsc` to generate C code, or a mixture between C code and bytecodes. We also describe how to compile this C code and how to integrate it with the RScheme system.

As mentioned before, the off-line module compiler is invoked like this for ordinary user modules: %
`rsc -p foo.mcf`

where `-p` means "package" and where `foo1.mcf`, `foo2.mcf` etc are the module control files for modules `foo1` and `foo2` respectively.

In the default case where the strategy is to compile to bytecodes, `rsc` generates two files for each module, the module image file (extension `.mif`) and the module index file (extension `.mx`).

When the off-line module compiler compiles the Scheme code of a module, it uses a default strategy. If the option `-ccode` is given to `rsc` the default strategy is to generate C code, otherwise (if no argument `-ccode` has been given) the default strategy is to generate bytecodes. The default strategy can then be overridden for a sequence of top-level forms in the Scheme code. For that purpose, you use the special form (`%strategy strategy form ...`). Here `strategy` can be either `ccode` or `bytecode`. The forms are typically top-level procedure definitions, but can be any top-level forms.

If any form is compiled with strategy `ccode`, whether by command-line options or the use of the `%strategy` special form, then the off-line module compiler generates a number of files in addition to the module image file and the module index file. The compiler puts these additional files in the directory indicated by the `outdir` element in the module descriptor [is that what we called it?] in the module control file.

[why, btw is the extension `.mx` and not `.mxf`, or, alternatively, why are the extensions `.mcf` and `.mif` and not just `.mc` and `.mi`. I mean you use `.c` for a C file, not `.cf`. The fact that it is a file is kind of implicit, no?]

For the module as a whole, the compiler generates a Makefile, two header files and a C file. The Makefile is used by the Unix `make` command to generate the object code for the module. For module `foo`, the header files will be called `foo.h` and `foo_p.h`. The file `foo.h` is the ordinary header file that can be used by a client module to access exported symbols in the module, and `foo_p.h` is the so-called private header file containing declarations that should only be used by other parts of the module to

access data that is morally but not technically private to the module. The C file is called `foo_1.c` and is called the linkage file. This file contains C code needed to link the module to the RScheme system.

In addition to the files associated with the module as a whole, the compiler generates one C file for each Scheme file (extension `.scm`) that is part of the module (recall that the Scheme files of a module are listed in the `files` clause in the module control file) [is this true: and that contains at least one form to be compiled with the `ccode` strategy]. For Scheme file `foo.scm`, the name of this file is `foo.c`. This file is where the compiler puts the code for each form that needs to be translated to C.

Next, you have to decide whether you want to incorporate the module code by static or dynamic linking into your RScheme system. Static linking makes the RScheme executable bigger but the start-up time will be shorter. Dynamic linking is more flexible in that you can decide for each execution of your RScheme system whether you need the module or not.

1.15.1. Incorporation of object code using static linking

For static linking, the next step is to generate object (extension `.o`) code for the entire module from the C files. To do that, you use the Unix `make` command in the directory (`outdir`) where the compiler put the Makefile, the header files, and the C files.

Before you use the `make` command, however, you need to figure out where your RScheme system is installed. Usually, this place will be something like `/usr/local/lib/rs/VER` where `VER` is something like `0.7.2`. This location is referred to as the install directory. However, since it is possible to change the install directory when the RScheme system is compiled, you may have to consult the person who did this installation as to the exact location to give here.

Supposing that the install directory is `/usr/local/lib/rs/0.7.2`, you type `make INSTALL_DIR=/usr/local/lib/rs/0.7.2` to the Unix shell. This `make` command will generate a single file `foo.o` (where `foo` is the name of the module) [the bug is that it actually generates `bar.o` where `bar` is the name of the directory (`outdir`?)]

As part of the static linking process, you now have to build a new version of the `rs` command that, in addition to all the normal files, also contains the file `foo.o`.

But before you link the final executable, you have to make sure that the top-level forms of your module are executed when the RScheme system starts. To accomplish that, you have to inform the RScheme start-up code of the existence of your module.

To understand how this mechanism works, we need to describe the way the RScheme system is organized. The bulk of the functionality of RScheme is assembled in a Unix library called `libs.a` which is installed in `/usr/local/lib/rs/0.7.1` (for version 0.7.1). The only thing that has been left out of `libs.a` is the file that contains the main program. This file is called `shell.c`. The RScheme system is built from the file `shell.o` (`shell.c` compiled) and the library `libs.a`. The file `shell.c` contains, in addition to the main program, a vector of module descriptors of modules to be initialized when RScheme starts up. The descriptor of your module must be in this list.

The easiest way to integrate your module is to copy the `shell.c` file into the directory where your module object code `foo.o` lives. Next, you have to modify the `shell.c` file using your favorite text editor. Here is the general structure of `shell.c`:

```
static struct module_descr *std_modules[] =
{
```

```

    STD_MODULES_DECL
};
...
main()
{
    ...
}

```

To insert your own modules (say) `foo1` and `foo2` into this file, you need to modify it to look like this:

```

...
#include "foo1.h"
#include "foo2.h"
...
static struct module_descr *std_modules[] =
{
    &module_foo1,
    &module_foo2,
    STD_MODULES_DECL
};
...
main()
{
    ...
}

```

The order between the entries of the table is not significant as the initialization code is executed in the order determined by the dependencies between modules, not by the order in this vector.

Now that you have a modified `shell.c`, you can finally create the modified version of the `rs` command. We recommend that you call it something other than `rs` so as to avoid confusion. Here is the general structure of the command to use

```

% cc -I /usr/local/lib/rs/0.7.1/include
    -I the-directory-in-which-the-module-lives
    shell.c
    -L /usr/local/lib/rs/0.7.1/lib
    -o name-of-final-executable

```

While the resulting executable file contains the machine code for the forms compiled with strategy ccode, it does not contain forms compiled with strategy bytecode, nor other information that is kept in the module image file and the module index file.

So in order to use your module from RScheme, you still have to say `(use foo)` if `foo` is the name of the module. Alternatively, you can use the same method as we used for pure bytecode in order to create a new RScheme image in which the module is fully incorporated. Supposing you gave the name `rsf` to the name of the final executable containing object code for you module, you can now do:

```

% rsf -m foo.mif -c foo.img

```

to create an image `foo.img` in which all of your module exists. If you want your executable `rsf` to behave like the executable `rs` (except with more functionality), i.e., with a read-eval-print loop, you can do:

```

% rsf -m foo.mif -c.repl foo.img

```

The option `-c module` indicates that the procedure called `main` is taken from the module `module`. The `module` is not imported, however. `-c` means to use the `main` from the current environment.

To start the new system with the new image, use:

```
% rsf -image foo.img
```

1.15.2. Incorporation of object code using dynamic linking

Using dynamic linking instead of static linking to incorporate your module into RScheme is slightly more complicated.

First, you have to make sure your RScheme command `rs` was linked with a flag to `ld` indicating that the dynamic linker can use symbols in the executable to resolve unresolved symbols in dynamically linked code. The name of this flag varies between systems. On GNU/Linux, it is `-rdynamic`.

If `rs` was not built with this flag, you first have to rebuild it. Normally, if the object code is still around, you only have to redo the final link stage.

Next, you have to modify the Makefile generated by the off-line module compiler. You have to inform the C compiler to generate position independent code for all the C files of your module. This is necessary because the exact position of the final executable code may vary from one execution of the system to another. The exact form of this flag varies from system to system, but for the GNU C compiler it is `-fPIC`. In addition to informing the C compiler about position independent code, you have to tell the linker to generate a dynamically linkable library rather than just an object file, so instead of `foo.o` you obtain `libfoo.so`. Again the way to do this varies from system to system.

At this point you are almost done. You don't need to recompile the RScheme executable, only make sure that it can find your file `libfoo.so`. If your module is generally useful, you may want to install it together with other modules delivered with RScheme in the directory for this purpose (usually `/usr/local/lib/rs/0.7.1/resource/modules`). For application-specific modules, you simply need to make sure the directory in which the module is installed is in RScheme's search path for dynamically linked modules[1].

RScheme supports user-level threads portably, even on operating systems that don't have threads at all. RScheme multiplexes several user threads onto one OS-level process. These threads appear to be preemptive from the normal user's point of view. (For example, nonblocking I/O is used so that one thread reading from the network doesn't block the whole process; the reading thread actually issues a nonblocking read request and a thread switch activates a waiting thread.)

When threads support is compiled in, the REPL comes up by default running in a thread. In this case, the `,t1` directive will list all the threads in the system with their state, total run time, and the kind of object they are blocked on, if any. For example:

```
top[0]=>,t1
0 [main]      RUN   46.2 ms
1 [finalize]  BLOCK 27 us   #[<mailbox> "finalize"]
2 [monitor]  BLOCK 54 us   #[<mailbox> "signal"]
```

Synchronous exceptions (e.g., taking the car of 3) still generate a break loop in the thread causing the exception (typically the REPL thread itself).

NOTE: Need to talk about the backstop handler

However, when running a threaded REPL, an asynchronous interrupt (as from `^C`) will suspend the current thread group and create a new "break" thread group for a new REPL:

```
top[1]=>^C
** 0: Interrupt received: (SIGINT . #[<time> Thu Jul 17 16:21:00 2003])
break[0]=>,t1
0 [main]      SUSP 50 ms
1 [finalize]  BLOCK 27 us   #[<mailbox> "finalize"]
2 [monitor]  BLOCK 3.1 ms  #[<mailbox> "signal"]
3 [break]    RUN   1.68 ms
```

When the break loop ends (as by EOF, or `^D`), the suspended thread group is resumed.

In order to maintain thread progress, the low-level system call I/O and other blocking system calls should not normally be used (e.g., `fd-read`, `fd-write`, `socket-accept`, `wait-for`) since they will typically block the entire RScheme process waiting for the operation to complete.

The threads system uses `select()` internally to manage multiple outstanding non-blocking I/O requests. Therefore, when interacting with "slow" devices such as the network, use the appropriate procedures described in this chapter to create safe objects. See section §16.4 for more details.

NOTE: The procedures defined in this chapter are available from the `rs.sys.threads.manager` module.

1.16.1. Thread Objects

The `make-thread` procedure is used to create threads. The system maintains a notion of thread groups, which is a hierarchy of thread sets. You can also determine what thread is currently running, and its thread group.

`make-thread` **function**

Create a new thread.

`(make-thread thunk [name] [group])` \Rightarrow *thread*

Arguments

thunk An instance of <procedure>

name An instance of <string>

group An instance of <thread-group>

Return Values

thread An instance of <thread>

Description

Creates and returns a new thread of control.

The optional *name* provides a convenient way of referring to the thread to the user. If not supplied, the *name* defaults to "thread".

The optional *group* allows the new thread to be created as a member of a thread group other than the current thread group. Since a thread's group cannot be changed once created, this is the only way to create a thread running in a different group.

The newly created thread is initially suspended, with a suspend count of 1. Use `thread-resume` to make the thread available for execution.

The behavior of the thread is to invoke the given *thunk*. When *thunk* returns, the thread enters the `complete` state and `thread-join` returns (all) the values returned by *thunk*.

`current-thread-group` **function**

Return the thread group of the current thread.

`(current-thread-group)` \Rightarrow *group*

Return Values

group An instance of <thread-group>

Description

Returns the current thread group, which is to say, the thread group of the current thread.

`current-thread` **function**

Return the currently running thread.

`(current-thread)` \Rightarrow *thread*

Return Values

thread An instance of <thread>

Description

Returns the currently running thread.

`thread-suspend` **function**

Suspend a thread.

`(thread-suspend thread)` \Rightarrow

Arguments

thread An instance of <thread>

	Description	
	<p>Increments the suspend count of the given <i>thread</i>. This makes the thread ineligible for execution until a corresponding <code>thread-resume</code> is invoked.</p> <p>If <i>thread</i> is the current thread, this procedure does not return until the corresponding <code>thread-resume</code> is invoked.</p> <p>If <i>thread</i> is another, currently running, thread (<i>i.e.</i>, in a SMP context), this procedure blocks until the target thread has stopped running (which should be fairly soon). [1]</p>	
<code>thread-resume</code>		function
	<p>Resume a thread.</p> <p>(<code>thread-resume thread</code>) ⇒</p>	
	Arguments	
	<i>thread</i> An instance of <thread>	
	Description	
	<p>Decrements the suspend count of the thread. If the suspend count becomes zero, the thread is eligible to execute (unless it was blocked on something, that is).</p> <p>If the thread already has a zero suspend count, this procedure has no effect.</p>	
<code>thread-sleep</code>		function
	<p>Sleeps the current thread for <i>sec</i> seconds.</p> <p>(<code>thread-sleep sec</code>) ⇒</p>	
	Arguments	
	<i>ms</i> An instance of <real> denoting the number of seconds to sleep	
	Description	
	<p>The current thread blocks for approximately <i>sec</i> seconds, which may be a fractional quantity. That is, the current thread is marked as sleeping and after <i>sec</i> seconds have passed (real time), it is marked as runnable. When exactly the thread runs after it gets marked runnable is subject to other dynamics, especially the number and priority of other threads that are runnable.</p>	

1.16.2. Synchronization

<code>make-mailbox</code>		function
	<p>Create an empty mailbox.</p> <p>(<code>make-mailbox</code>) ⇒ <i>mbox</i></p>	
	Return Values	
	<i>mbox</i> An instance of <mailbox>	
	Description	
	<p>Creates and returns a new, empty mailbox. Mailboxes are the basic building block for high-level synchronization in RScheme threads. They are unbounded, synchronized FIFO queues which can pass arbitrary Scheme objects from sender to receiver.</p>	
<code>send-message!</code>		function
	<p>Send a message to a mailbox.</p> <p>(<code>send-message! mbox item</code>) ⇒</p>	
	Arguments	
	<i>mbox</i> An instance of <mailbox>	
	<i>item</i> An instance of <object>	
	Description	
	<p>This procedure enqueues <i>item</i> as a message in <i>mbox</i>. Since mailboxes are unbounded queues, this</p>	

procedure never blocks. Messages are inserted in order.

receive-message!	<p>Receive a message from a mailbox. <code>(receive-message! <i>mbox</i>)</code> ⇒ <i>item</i></p> <p>Arguments</p> <p><i>mbox</i> An instance of <mailbox></p> <p>Return Values</p> <p><i>item</i> An instance of <object></p> <p>Description</p> <p>This procedure extracts the next message from <i>mbox</i>. If nothing is available, the thread blocks until a corresponding message is enqueued. Multiple waiters on a single mailbox are queued in FCFS order.</p>	function
make-send-rights	<p>Extract send rights from mailbox. <code>(make-send-rights <i>mailbox</i>)</code> ⇒ <i>mbox</i></p> <p>Arguments</p> <p><i>mailbox</i> An instance of <mailbox></p> <p>Return Values</p> <p><i>mbox</i> An instance of <mailbox-send-rights></p> <p>Description</p> <p>Extracts only send rights from the given mailbox, returning a (possibly shared) handle to the underlying mailbox containing only send rights. Messages sent to <i>mbox</i> go to the underlying <i>mailbox</i>, but messages may not be received from <i>mbox</i>. (Attempting to do so signals a <only-send-rights> condition.)</p> <p style="margin-left: 40px;">NOTE: This is implemented using a proxy wrapper class for mailboxes. The <code>send-message!</code> primitive knows how to indirect through the proxy wrapper.</p> <p style="margin-left: 40px;">NOTE: This is not currently implemented.</p>	function
make-semaphore	<p>Create a new semaphore object, with initial count of <i>n</i>. <code>(make-semaphore <i>n</i>)</code> ⇒ <i>sem</i></p> <p>Arguments</p> <p><i>n</i> An instance of <integer></p> <p>Return Values</p> <p><i>sem</i> An instance of <semaphore></p> <p>Description</p> <p>Creates a new semaphore synchronization object, with an initial count of <i>n</i>. If omitted, <i>n</i> defaults to 0.</p>	function
semaphore-signal	<p>Increment semaphore count. <code>(semaphore-signal <i>sem</i>)</code> ⇒</p> <p>Arguments</p> <p><i>sem</i> An instance of <semaphore></p> <p>Description</p> <p>Increments the semaphore counter, unblocking exactly one thread if any threads are blocked on a <code>semaphore-wait</code>.</p>	function
semaphore-wait	<p>Decrement semaphore count, blocking if negative. <code>(semaphore-wait <i>sem</i>)</code> ⇒</p>	function

	<p>Return Values</p> <p><i>socket</i> An instance of <server-socket></p> <p>Description</p> <p>Creates a socket object which listens on the given port.</p> <p>If the <i>port</i> argument is an integer, then it represents a TCP port number and in which case the socket is bound to IN_ADDR_ANY and the socket will accept connections on any IP address. The <i>port</i> argument may also be a <inet-socket-addr>, in which case the socket will accept connections only on the particular IP address and port.</p>	
accept-client	<p>Accept a connection. (accept-client <i>serversock</i>)</p> <p>Arguments</p> <p><i>serversock</i> An instance of <server-socket></p> <p>Return Values</p> <p><i>peersock</i> An instance of <responder-socket></p> <p>Description</p> <p>Accepts the next connection from the socket.</p>	<p>function</p> <p>⇒ <i>peersock</i></p>
open-client-socket	<p>Create a TCP client socket (open-client-socket <i>addr</i> [<i>name</i>]) (open-client-socket <i>host port</i>)</p> <p>Arguments</p> <p><i>addr</i> An instance of <inet-socket-addr></p> <p><i>name</i> Optional; an instance of <object></p> <p><i>host</i> An instance of <string></p> <p><i>port</i> An instance of <fixnum></p> <p>Return Values</p> <p><i>socket</i> An instance of <initiator-socket></p> <p>Description</p> <p>Creates a socket object connected to a remote server.</p>	<p>function</p> <p>⇒ <i>socket</i></p> <p>⇒ <i>socket</i></p>
close	<p>Close a TCP socket (close <i>socket</i>)</p> <p>Arguments</p> <p><i>socket</i> An instance of <peer-socket> or <server-socket></p> <p>Description</p> <p>Close a TCP socket.</p>	<p>method</p> <p>⇒</p>

RScheme provides some basic procedures for building internet-aware applications, including both clients and servers. Additional packages provide specialized procedures for implementing particular internet protocols, such as `ident`, `HTTP`, and `X`.

1.17.1. Server-side Procedures

`inet-server` **function**

Create an internet server.

`(inet-server port)` \Rightarrow *fd*

Arguments

port An IP port as a `<fixnum>`, or a socket address as a `<inet-socket-addr>`.

Return Values

fd An instance of `<fixnum>`

Description

This procedure establishes a socket on the given *port* to listen for IP connections. If the *port* is a number, then connections to any IP address on the local machine will be accepted. Otherwise, only connections to the IP address and port specified in the `<inet-socket-addr>` will be accepted.

The listen queue is set to 3.

NOTE: If threads are being used, the `open-server-socket` (p.61) function is preferred.

`make-service` **function**

Create a service.

`(make-service fd)` \Rightarrow *service*

Arguments

fd An instance of `<fixnum>`

Return Values

service An instance of `<service>`

Description

This procedure arranges to listen for connections on *fd*, which was presumably created using something like `inet-server`. Use `get-next-client` to dequeue connecting clients.

`get-next-client` **function**

Get the next client connecting to a service.

`(get-next-client service)` \Rightarrow *fd peer*

Arguments

service An instance of `<server>`

Return Values

fd An instance of `<fixnum>`

	<i>peer</i>	An instance of <socket-addr>	
	Description	This procedure extracts the next client connecting to <i>service</i> . It returns the socket, <i>fd</i> , and the peer socket name.	
inet-client			function
	Connect to an internet server.		
	(inet-client <i>host port</i>)		⇒ <i>fd</i>
	Arguments		
	<i>host</i>	An instance of <hostspec>	
	<i>port</i>	An instance of <fixnum>	
	Return Values		
	<i>fd</i>	An instance of <fixnum>	
	Description	This procedure establishes a TCP/IP (stream) connection to the the given <i>port</i> on the given <i>host</i> . The host may be specified as either a <string> or a <inet-addr>. If host is a <string>, it is resolved using string->hostaddr .	
string->hostaddr			function
	Resolve a host name or dotted IP address.		
	(string->hostaddr <i>spec</i>)		⇒ <i>host</i>
	Arguments		
	<i>spec</i>	An instance of <string>	
	Return Values		
	<i>host</i>	An instance of <inet-addr>	
	Description	This procedure converts a string into a <inet-addr>. If the first character of the string is a digit, then the string is interpreted as a dotted IP address (e.g., 204.71.200.72). Otherwise, it is interpreted as a host name and resolved using gethostbyname .	
	If the host name cannot be found, or it has no address, or if the resolver library encounters some other error, a <resolver-error> condition is signalled.		
remote-port-owner			function
	Use ident protocol to find the remote owner of a socket.		
	(remote-port-owner <i>fd</i>)		⇒ <i>type info</i>
	Arguments		
	<i>fd</i>	An instance of <fixnum>, the file descriptor for the local socket.	
	Return Values		
	<i>type</i>	An instance of <string>	
	<i>info</i>	An instance of <string>	
	Description	This procedure contacts the ident server on the machine which is the peer of the given socket. If a recognizable ident response is returned, the response type and additional info are returned as two values.	
	<pre>(define svc (make-service (inet-server 5050))) (remote-port-owner (get-next-client svc)) ⇒ "USERID" ⇒ "UNIX : dkolbly"</pre>		
	If the peer host is not running an ident server, this procedure signals an error (an <os-error>),		

"Connection refused").

(This Page Intentionally Left Blank)

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 repercussion. 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.

<code>trace</code>	Trace function entry and exit. (<code>unquote function</code>)	<code>replcmd</code>
	Description This debugging command causes the named function(s) to be traced. That is, the function is reconfigured to print out its arguments and return values when it is called. Tracing is turned off using the <code>no-break</code> command.	
<code>break</code>	Break on function entry. (<code>unquote function</code>)	<code>replcmd</code>
	Description The <code>break</code> debugging command is similar to the <code>trace</code> command, except instead of printing out the function arguments when it is called, it creates a break-level read-eval-print loop, <code>BRK</code> .	

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	Remove tracing or breakpoint on a function. (<code>no-break function</code>)	replcmd
	Description Turn of breaking on the given <i>function</i> .	
abt	Print backtrace of function calls. (<code>abt</code>)	replcmd
	Description Show a backtrace of applications.	
bt	Print continuation chain (stack). (<code>bt</code>)	replcmd
	Description Show a backtrace of applications.	

This package (module `calendar`) provides basic date manipulation functions.

`date->string` **function**

Convert a date object into string format (YYYY.MM.DD)

`(date->string date)` \Rightarrow *str*

Arguments

date An instance of `<date>`

Return Values

str An instance of `<string>`

Description

This function converts the given date object, *date*, into a string in the default format, e.g., 1997.09.14.

`string->date` **function**

Parse a date in standard format (YYYY.MM.DD) into a date object

`(string->date string)` \Rightarrow *date*

Arguments

string An instance of `<string>`

Return Values

date An instance of `<date>`

Description

This function is the inverse of `date->string`, parsing a string representation of a date into the corresponding date object.

`ymd->date` **function**

Convert a day in separate year, month, and day to a date object.

`(ymd->date year month day)` \Rightarrow *date*

Arguments

year An instance of `<fixnum>`

month An instance of `<fixnum>`

day An instance of `<fixnum>`

Return Values

date An instance of `<date>`

Description

This function computes the date object corresponding to the given *year*, *month*, and *day* in the Gregorian calendar.

`(ymd->date 1997 9 14)` \Rightarrow 1997.09.14

`(date->day (ymd->date 1997 9 14))` \Rightarrow 729281

`date->ymd` **function**

	Returns the parts of a date in the Gregorian calendar. (<i>date</i> -> <i>ymd</i> <i>date</i>)	⇒ <i>year month day</i>	
	Arguments		
	<i>date</i>	An instance of <date>	
	Return Values		
	<i>year</i>	An instance of <fixnum>	
	<i>month</i>	An instance of <fixnum>	
	<i>day</i>	An instance of <fixnum>	
	Description		
	<i>date</i> -> <i>ymd</i> is the inverse of <i>ymd</i> -> <i>date</i> (as you might guess from the name), and returns the broken out year, month, and day for a given <i>date</i> object.		
date+			function
	Compute a new date relative to a given date. (<i>date+</i> <i>date</i> <i>days</i>)	⇒ <i>new-date</i>	
	Arguments		
	<i>date</i>	An instance of <date>	
	<i>days</i>	An instance of <fixnum>	
	Return Values		
	<i>new-date</i>	An instance of <date>	
	Description		
	This function returns the date which is <i>days</i> away from the given <i>date</i> . If <i>days</i> is negative, then a preceding date is returned.		
	(define t (ymd->date 1997 9 14))		
	(date+ t 3)	⇒ 1997.09.17	
	(date+ t -14)	⇒ 1997.08.31	
date=?			function
	Compare dates		
	(<i>date=?</i> <i>date1</i> <i>date2</i>)	⇒ <i>val</i>	
	(<i>date>?</i> <i>date1</i> <i>date2</i>)	⇒ <i>val</i>	
	(<i>date>=?</i> <i>date1</i> <i>date2</i>)	⇒ <i>val</i>	
	(<i>date<?</i> <i>date1</i> <i>date2</i>)	⇒ <i>val</i>	
	(<i>date<=?</i> <i>date1</i> <i>date2</i>)	⇒ <i>val</i>	
	Arguments		
	<i>date1</i>	An instance of <date>	
	<i>date2</i>	An instance of <date>	
	Return Values		
	<i>val</i>	An instance of <boolean>	
	Description		
	These functions compare dates, based on the ordering of days.		
	(define td (ymd->date 1997 9 14))		
	(define tm (ymd->date 1997 9 15))		
	(date<? td tm)	⇒ #t	
	(date>=? td td)	⇒ #t	
date->time			function
	Compute the time on a particular day (<i>date->time</i> <i>date</i> <i>secs</i>)	⇒ <i>time</i>	

Arguments			
	<i>date</i>	An instance of <date>	
	<i>secs</i>	An instance of <fixnum>	
Return Values			
	<i>time</i>	An instance of <time>	
Description			
		Returns the <i>time</i> corresponding to the given number of seconds, <i>secs</i> , since midnight GMT on the given <i>date</i> .	
		Because of the representation of time objects (from the <code>syscalls</code> module, btw), <i>date</i> must refer to January 1, 1970, or later.	
date->weekday			function
		The day of the week on which the date falls.	
		(<i>date->weekday date</i>)	\Rightarrow <i>weekday</i>
Arguments			
	<i>date</i>	An instance of <date>	
Return Values			
	<i>weekday</i>	An instance of <symbol>	
Description			
		Returns a symbol representing the day of the week on which the given <i>date</i> falls. In particular, returns exactly one of: <code>sunday</code> , <code>monday</code> , <code>tuesday</code> , <code>wednesday</code> , <code>thursday</code> , <code>friday</code> , <code>saturday</code> .	
date->week			function
		Breaks a date into week number and day of week.	
		(<i>date->week date</i>)	\Rightarrow <i>week day-of-week</i>
Arguments			
	<i>date</i>	An instance of <date>	
Return Values			
	<i>week</i>	An instance of <fixnum>	
	<i>day-of-week</i>	An instance of <fixnum>	
Description			
		This function factors a given <i>date</i> into its <i>week</i> and <i>day-of-week</i> components. The second return value, <i>day-of-week</i> , is in the range 0 to 6 (inclusive), indicating the weekdays Sunday through Saturday.	
week->date			function
		Compute the date for a given week and day of week.	
		(<i>week->date week day-of-week</i>)	\Rightarrow <i>date</i>
Arguments			
	<i>week</i>	An instance of <fixnum>	
	<i>day-of-week</i>	An instance of <fixnum>	
Return Values			
	<i>date</i>	An instance of <date>	
Description			
		This function returns the date corresponding to the given day of the week.	
		The following example shows how one might use <code>date->week</code> and <code>week->date</code> to find the Sunday before a given date.	

```

      (define td (ymd->date 1997 9 13))
      (date->week td)                    => 104182 =>      6
      (week->date 104182 0)              => 1997.09.07

```

short-weekday-name **function**

Return the short (English) names for date quantities

```

(short-weekday-name weekday)           => name
(short-month-name month)              => name

```

Arguments

month An instance of <fixnum>

Return Values

name An instance of <string>

Description

`short-weekday-name` maps weekday symbols, as returned by `date->weekday`, to their short (3-character) English name.

`short-month-name` does the same for month numbers (1-12) and month names.

The persistent object store module (`rs.db.rstore`) is an implementation based on the concept of pointer swizzling at page-fault time as described in Paul Wilson's Pointer Swizzling at Page-Fault Time. 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.

1.20.1. Creating and Accessing an Object Store

The life cycle of an object store starts off with its 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.

1.20.2. 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.

1.20.3. 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 opens 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.

1.20.4. 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.

One consequence of *copying* objects into the store to make them persistent is that *object identity of transient objects is lost* across a commit. If you hold on to a pointer to a transient object which is reachable from the root of the store, then after the commit, that pointer will no longer refer to the object within the store. [Need a picture here!]

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.)

1.20.5. Functions Reference

`open-persistent-store`

function

Opens an existing store
 (`open-persistent-store path`) \Rightarrow `pstore`

Arguments

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

Return Values

`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**

Creates a new persistent store
(*create-persistent-store path*) ⇒ *pstore*

Arguments

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

Return Values

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
(*root-object pstore*) ⇒ *object*

Description

This procedure extracts a pointer to the object last named as the root object of the repository. The root object may be changed with `set-root-object!`.

commit **function**

Commit changes to persistent store
(*commit pstore*) ⇒ *locator*

Arguments

pstore An instance of <persistent-store>.

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

Return Values

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

Description

This procedure synchronizes the in-memory object store with the representation on disk. It uses the `fsync` system call to ensure the data has been pushed onto disk. [We should support a faster mode, where all we protect against is the program crashing, in which case we don't need to `fsync`.]

setup-indirect-page **function**

Configure a page of pivot objects
(*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

(`alloc-indirect-pages` *pstore* *num-pages*)

⇒ *first-page*

Arguments

pstore An instance of <persistent-store>.

num-pages An instance of <fixnum>.

Return Values

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.

This chapter describes the RScheme interface to the PostgreSQL Data Base Management System (formerly known as Postgres95). The interface (glue) functionality is contained in an RScheme package, `pg95`, which provides a single module by the same name [1].

1.21.1. Initialization

In order to make use of a PostgreSQL database, you must establish a *connection* to it. You may connect to databases on other hosts or on ports other than the default by specifying appropriate keywords to the `postgres-connect` function.

1.21.2. Commands

Commands are submitted to the database using the `pg95-exec-command`

1.21.3. Queries

This section describes how queries are made.

1.21.4. Errors

All the glue functions that interface to the database check for error conditions. An error condition that arises is manifest as the signalling of a `<pg95-error>`, which is a kind of `<error>`.

Most functions will signal a `<pg95-exec-error>`, which indicates an error in the execution of a query or command. The other case is a `<pg95-connect-error>`, which denotes an error attempting to connect to the database. (The reason for the distinction is that in the latter case, a connection exists and can be used to identify the error. In the latter case, no valid connection exists.)

1.21.5. Object mappings

The `pg95` module has functions for interpreting tuples and rows of tables as objects in the RScheme object system. This is done by making use of PostgreSQL type introspection techniques and RScheme class introspection.

The general rule is that as long as the the name of the class slots are the same as the names of the tuple fields, then the mapper can fill in the appropriate slots. The slot with name `oid` can be used to refer to the PostgreSQL object id. [give details, esp. as regards value mappings and special cases]

1.21.6. Functions Reference

postgres-connect	<p>Connect to Postgres95 server (<code>postgres-connect database host port opts tty</code>) ⇒ <i>cnxn</i></p> <p>Arguments</p> <p><i>database</i> An instance of <string></p> <p><i>host</i> An instance of <string></p> <p><i>port</i> An instance of <string></p> <p><i>opts</i> An instance of <string></p> <p><i>tty</i> An instance of <string></p> <p>Return Values</p> <p><i>cnxn</i> An instance of <pg95-connection></p> <p>Description</p> <p>This function creates a connection to the PG95 database with the name <i>database</i>. If the connection cannot be established (ie, because of network, installation, or authentication problems), an error is signalled.</p> <pre>(postgres-connect "foo") #{a <pg95-connection>} (postgres-connect "bar" host: "dbsrv.bar.com") #{a <pg95-connection>}</pre>	function
pg95-connect	<p>Underlying function to connect to PG95 (<code>pg95-connect host port opts tty database</code>) ⇒ <i>cnxn</i></p> <p>Arguments</p> <p><i>host</i> An instance of <string></p> <p><i>port</i> An instance of <string></p> <p><i>opts</i> An instance of <string></p> <p><i>tty</i> An instance of <string></p> <p><i>database</i> An instance of <string></p> <p>Return Values</p> <p><i>cnxn</i> An instance of <pg95-connection></p> <p>Description</p> <p>This function, like <code>postgres-connect</code>, connects to a postgres 95 database. However, the arguments are explicit rather than being keywords. The string arguments should be empty strings if the intention is to pass NULL to the <code>PQsetdb()</code> function (which is the default).</p>	function
pg95-exec-command	<p>Execute a command against the database. (<code>pg95-exec-command cnxn command</code>) ⇒ <i>result</i></p> <p>Arguments</p> <p><i>cnxn</i> An instance of <pg95-connection></p> <p><i>command</i> An instance of <string></p> <p>Return Values</p> <p><i>result</i> An instance of <object></p> <p>Description</p> <p>This function executes a command (as opposed to a query) against the database. Common commands are things like "create table" and "insert into".</p>	function

The value returned is either an integer object id, in the case of insertions, or #f.

pg95-with-tuples **function**

Execute a query (like "select") against the database
(**pg95-with-tuples** *cnxn command proc*) ⇒ *result*

Arguments

cnxn An instance of <pg95-connection>
command An instance of <string>
proc An instance of <function>

Return Values

result An instance of <object>

Description

This function executes a query (as opposed to a command) against the database. The given *proc* is called with three arguments, which are respectively, the query result object (an instance of <pg95-result>), the number of tuples in the result, and the number of fields in each tuple (both <fixnum>s).

When the function returns, the C result object is cleared (freed?).

pg95-field-names **function**

Get the names of the fields in a result's tuples.
(**pg95-field-names** *result first-index index-limit*) ⇒ *names*

Arguments

result An instance of <pg95-result>
first-index An instance of <fixnum>
index-limit An instance of <fixnum>

Return Values

names An instance of <list>

Description

This function extracts the names of the fields that are present in the tuples of a result. *first-index* is typically 0, while *index-limit* is typically the number of fields. This would return a list of all the field names in the result.

pg95-field-number **function**

Determine which field has a given name.
(**pg95-field-number** *result name*) ⇒ *field-num*

Arguments

result An instance of <pg95-result>
name An instance of <string>

Return Values

field-num An instance of <fixnum>

Description

This function looks up the given *name* in the fields of a result, returning the index of the field, *field-num* (0-based).

pg95-field-size **function**

Get the size of the given field.
(**pg95-field-size** *result field-num*) ⇒ *size*

Arguments

result An instance of <pg95-result>

	<i>field-num</i>	An instance of <fixnum>	
	Return Values		
	<i>size</i>	An instance of <fixnum>, or #f	
	Description		
		Returns the size of the given field in the result structure, where fields are named by index, <i>field-num</i> . Returns #f instead of a size if the field has negative size (indicating what, again,...?)	
pg95-field-type			function
		Get the PG95 type id for the values in the given field. (pg95-field-type <i>result field-num</i>)	\Rightarrow <i>type-id</i>
	Arguments		
	<i>result</i>	An instance of <pg95-result>	
	<i>field-num</i>	An instance of <fixnum>	
	Return Values		
	<i>type-id</i>	An instance of (<fixnum>)	
	Description		
		Returns the PG95 type id (which is an oid in the <code>pg_type</code> table) for the given field (<i>field-num</i>) in a result structure.	
pg95-get-value			function
		Get the value of a particular field in a particular tuple of a result. (pg95-get-value <i>result tuple-num field-num</i>)	\Rightarrow <i>value</i>
	Arguments		
	<i>result</i>	An instance of <pg95-result>	
	<i>tuple-num</i>	An instance of <fixnum>	
	<i>field-num</i>	An instance of <fixnum>	
	Return Values		
	<i>value</i>	An instance of <string>, or #f	
	Description		
		Returns the value of a particular field in a particular tuple of a result array. The value returned is a string representation, and will be #f if the value of the field is null.	
pg95-type			function
		Get the type name for a given PG95 type id. (pg95-type <i>cnxn type-id</i>)	\Rightarrow <i>type-name</i>
	Arguments		
	<i>cnxn</i>	An instance of <pg95-connection>	
	<i>type-id</i>	An instance of <fixnum>	
	Return Values		
	<i>type-name</i>	An instance of <symbol>	
	Description		
		Looks up the given type id (an OID in the <code>pg_type</code> table) and returns the corresponding type name (<code>typename</code> column).	
		To improve performance, the results of the lookup are cached in a hash table associated with the connection.	
table->list			function
		Return all the rows of a table as a list of objects.	

<p>(table->list <i>cnxn table-name class</i>)</p> <p>Arguments</p> <p><i>cnxn</i> An instance of <pg95-connection></p> <p><i>table-name</i> An instance of <string></p> <p><i>class</i> An instance of <<class>></p> <p>Return Values</p> <p><i>table-rows</i> An instance of <list></p> <p>Description</p> <p>Builds objects (instances of <i>class</i>) for the entire contents the database table named <i>table-name</i>. Returns the contents of the table as a list of those objects. The length of the list is the number of rows in the table.</p>	<p>⇒ <i>table-rows</i></p>
<p>oid->instance</p> <p>Return the identified row as an instance.</p> <p>(oid->instance <i>cnxn table-name class oid</i>)</p> <p>Arguments</p> <p><i>cnxn</i> An instance of <pg95-connection></p> <p><i>table-name</i> An instance of <string></p> <p><i>class</i> An instance of <<class>></p> <p><i>oid</i> An instance of <fixnum></p> <p>Return Values</p> <p><i>instance</i> An instance of class</p> <p>Description</p> <p>Builds a single object (instance of <i>class</i>) from the row in table <i>table-name</i> with the given object id, <i>oid</i>. In PostgreSQL, every row (tuple) in the entire system has a unique identity. This is like object identity. (And, like object identity encoded in pointers, may be reused for a new row after the row is deleted).</p>	<p>function</p> <p>⇒ <i>instance</i></p>
<p>make-extractor</p> <p>Create a tuple extractor function.</p> <p>(make-extractor <i>class cnxn result num-fields</i>)</p> <p>Arguments</p> <p><i>class</i> An instance of <<class>></p> <p><i>cnxn</i> An instance of <pg95-connection></p> <p><i>result</i> An instance of <pg95-result></p> <p><i>num-fields</i> An instance of <fixnum></p> <p>Return Values</p> <p><i>extractor</i> An instance of <function></p> <p>Description</p> <p>Compiles an <i>extraction plan</i> and returns a function which uses the plan to extract the <i>i</i>th tuple of the given result. The function returned is a function of one argument, which is the tuple number. That function returns an instance of <i>class</i>.</p>	<p>function</p> <p>⇒ <i>extractor</i></p>

(This Page Intentionally Left Blank)

One of the many uses to which RScheme has been put is as an application server for building intranet and internet web servers. There are several modules which are used to make the construction of web servers and web-based services convenient. The `rs.net.html` module provides basic procedures for generating HTML programmatically.

1.22.1. HTML

Most web-enabled facilities have a strong requirement for the dynamic generation of HTML. RScheme meets this need by supplying procedures for producing marked-up output. One distinguishing characteristic of RScheme's approach to generating dynamic HTML is that the markup is passed "out-of-band" using a specialized `<output-port>` so that normal output can be automatically escaped.

For example, if you are generating a fragment of HTML dynamically, you do not need to worry about whether or not the text you're generating contains HTML special characters like '`<`' or '`&`'. Consider this program fragment:

```
(define (gen-user-name (fullname <string>) (email <string>))
  (bold (display fullname))
  (display " <")
  (emph (display email))
  (display ">"))
```

The `gen-user-name` procedure need not be concerned about whether the `fullname` might contain characters special to HTML, as in "Alice & Bob". An in displaying the "`<`" and "`>`" email address delimiters, the programmer need not recall that these should appear as "`<`" and "`>`" in HTML – the system will automatically substitute the correct translation.

1.22.2. Web Server Support

RScheme's strengths are best revealed when it can be the web server itself.

1.22.2.1. Connecting the service

To build an actual web server, the first thing to do is to hook up a protocol interpreter to an appropriate TCP/IP socket.

1.22.2.2. Session Management

Facilities are included to manage sessions in two ways: using cookies, and using virtual subspaces. We could also support the hidden-variable approach, but that is too inconvenient to deploy services based on it and there is no apparent advantage, so we leave that as an exercise for the user.

1.22.2.3. Web spaces

An web server built using rs.net.http has an internal representation of the shape of the web space it serves. This space is structured mostly as a tree, reflecting the hierarchy of a URL.

When a request comes in, the server starts at the "web root" and walks down the tree, following links labelled by the elements in the request. For example, if a request for the object "/pub/build/0.7.3.1" comes in (as, for example, from the URL "http://www.rscheme.org/pub/build/0.7.3.1"), then the system will start at the root, follow the "pub" link from there, follow the "build" link from that location, and finally deliver the request to the result of following the "0.7.3.1" link from there.

1.22.2.4. Forms (POST)

A web-enabled service likely has a need to put data *into* the system. This is typically done with the HTTP POST method.

1.22.2.5. Queries (GET)

Sometimes it is convenient to encode a request as part of the URL. This shows up as a '?' after the last element in the URL path, as in "/pub/inspect?cr=123", which is a reference to the queryable resource "/pub/inspect" with the parameter "cr" with a value of "123".

As a matter of convention, I reserve this notation for side-effect-free operations (hence, I call them "queries"). For operations which update some persistent representation, I use POST.

1.22.2.6. Authentication and Security

Some portions of the web space may be protected. There is built-in support for using HTTP-based authentication as well as identd-based authentication.

A protected web node has a value for the `protect` property. When the web space traversal procedure gets to such a node, the generic function `check-protect` is invoked on the value of the `protect` property. Typically, this value will be an instance of `<http-protection>`, which has a method for doing basic authentication.

```
(define *confidential*  
  (make <http-protection>  
    in-realm: "ABC Inc."  
    password-check-proc: my-password-ok?  
    use-identd: ("*.abc.com")))
```

The `check-protect` method will signal an appropriate condition if the request does not satisfy the security constraints. In the case of HTTP authorization, using `in-realm`, this condition may be an `<http-error>` with code 401 (Unauthorized), with appropriate headers to elicit authorization from the web client and/or user.

1.22.3. Web Client Support

There are also facilities for building http clients. This is useful if some service is being deployed over an underlying HTTP protocol, or if one is building a web browser or web-crawling agent of some sort.

1.22.4. CGI Support

If RScheme is being used to augment an existing web server via the standard service process interface, CGI, there are facilities to assist in parsing the CGI environment.

1.22.5. Extended Example

In this extended example, we will go walk the development of a web-enabled service for source code control and build tracking. This example is taken from the system and facilities used in the development of RScheme itself.

1.22.5.1. Object Analysis

The first step in building a web-enabled service is to understand the domain. This means understanding what the conceptual objects are, what their relationships are, and what procedures or operations are to be web-enabled.

For this example application, the domain is that of source code control and build tracking. The objects are things like virtual filesystems, source files, filesystem snapshots, groups, users, change requests, and builds.

There are several kinds of relationships among these objects; filesystems contain files, files are owned by groups and users, files changes are motivated by change requests, etc.

Builds use the source from filesystems at a particular snapshots.

1.22.5.2. Web-Enabling Analysis

The next step is to design the web space. This involves understanding how the conceptual objects map onto web pages, how their relationships map onto links among those pages, and how the operations and procedures are appropriately represented using forms.

For this application, I chose a fairly direct mapping of application objects to web pages; each object is represented by a single page in the web space, with top-level web space links encoding type information.

For example, the page that describes change request 803 has the path `/cr/803`.

Since some of the objects being mapped are files, we get what appears to be a virtual filesystem of metadata published on the web. For example, the page describing the file `/handc/configure.in` in the filesystem `rs-0.7` is located at `/fs/rs-0.7/file/handc/configure.in`. It isn't necessary to use file names in this case – the equivalent of inode numbers could be used as well – but using the file names corresponds more closely with the an intuition about the shape of the web space.

(This Page Intentionally Left Blank)

All the material in this chapter is available from the `syscalls` module, which is an add-on package (since not all systems support this functionality).

1.23.1. Time Functions

<code>epoch-seconds->time</code>		function
	Returns the time object which is the given number of seconds (<code>epoch-seconds->time secs</code>)	\Rightarrow <i>time</i>
Arguments		
<i>secs</i>	An instance of <number>	
Return Values		
<i>time</i>	An instance of <time>	
Description	Returns the time corresponding to the given number of seconds since the Unix Epoch (January 1, 1970).	
<code>interval->string</code>		function
	Format an interval into a string. (<code>interval->string dt</code>)	\Rightarrow <i>string</i>
Arguments		
<i>dt</i>	An instance of <interval>	
Return Values		
<i>string</i>	An instance of <string>	
Description	Returns a string representing the length of the interval. This function chooses units that are appropriate for the length of the interval. For example, 90000 seconds would be rendered as 1.04 days ("1.04 d") whereas 0.9 seconds would be rendered as "900 ms".	
<code>time+interval</code>		function
	Adds an interval to a time. (<code>time+interval t1 dt1</code>)	\Rightarrow <i>t2</i>
Arguments		
<i>t1</i>	An instance of <time>	
<i>dt1</i>	An instance of <interval>	
Return Values		
<i>t2</i>	An instance of <time>	
Description	Returns a new time which is <i>dt</i> later than the given time. (<i>dt</i> may be negative, in which case the	

	returned time is earlier)		
interval+interval			function
	Adds two intervals.		
	(interval+interval <i>dt1 dt2</i>)	\Rightarrow	<i>dt3</i>
	Arguments		
	<i>dt1</i>	An instance of <interval>	
	<i>dt2</i>	An instance of <interval>	
	Return Values		
	<i>dt3</i>	An instance of <interval>	
	Description		
	Returns a new interval which is the sum of the given intervals.		
interval - interval			function
	Interval difference.		
	(interval - interval <i>dt1 dt2</i>)	\Rightarrow	<i>dt3</i>
	Arguments		
	<i>dt1</i>	An instance of <interval>	
	<i>dt2</i>	An instance of <interval>	
	Return Values		
	<i>dt3</i>	An instance of <interval>	
	Description		
	Subtracts the given intervals, <i>dt1</i> minus <i>dt2</i> .		
negative - interval			function
	Compute negative interval.		
	(negative - interval <i>dt1</i>)	\Rightarrow	<i>dt3</i>
	Arguments		
	<i>dt1</i>	An instance of <interval>	
	Return Values		
	<i>dt3</i>	An instance of <interval>	
	Description		
	Equivalent to (interval - interval (seconds->interval 0) <i>dt1</i>).		
time - time			function
	Compute the interval between two times.		
	(time - time <i>t1 t2</i>)	\Rightarrow	<i>delta</i>
	Arguments		
	<i>t1</i>	An instance of <time>	
	<i>t2</i>	An instance of <time>	
	Return Values		
	<i>delta</i>	An instance of <interval>	
	Description		
	Returns an interval representing the difference in the given times.		
time<?			function
	Time comparison.		
	(time<? <i>t1 t2</i>)	\Rightarrow	<i>cmp</i>
	(time<=? <i>t1 t2</i>)	\Rightarrow	<i>cmp</i>
	(time>? <i>t1 t2</i>)	\Rightarrow	<i>cmp</i>
	(time>=? <i>t1 t2</i>)	\Rightarrow	<i>cmp</i>

	(<i>time</i> =? <i>t1</i> <i>t2</i>)	⇒ <i>cmp</i>	
Arguments			
	<i>t1</i>	An instance of < <i>time</i> >	
	<i>t2</i>	An instance of < <i>time</i> >	
Return Values			
	<i>cmp</i>	An instance of < <i>boolean</i> >	
Description	Compares two times.		
clock-thunk			function
	Measure the execution time (as by <code>clock()</code>) of a function.		
	(<i>clock-thunk</i> <i>thunk</i>)	⇒ <i>elapsed</i>	
Arguments			
	<i>thunk</i>	An instance of < <i>function</i> >	
Return Values			
	<i>elapsed</i>	An instance of < <i>interval</i> >	
Description	Measure the execution time (as by <code>clock()</code>) of a function. The <code>clock()</code> function supposedly measures CPU (process) time, not wall clock time.		
time-thunk			function
	Measure the wall-clock execution time of a function.		
	(<i>time-thunk</i> <i>thunk</i>)	⇒ <i>elapsed</i>	
Arguments			
	<i>thunk</i>	An instance of < <i>function</i> >	
Return Values			
	<i>elapsed</i>	An instance of < <i>interval</i> >	
Description	this function uses time, the elapsed interval is in wall clock time, not process time.		
clock			function
	Return the process clock.		
	(<i>clock</i>)	⇒ <i>t</i>	
Return Values			
	<i>t</i>	An instance of < <i>interval</i> >	
Description	Return the time since process start as by <code>clock()</code> .		
time			function
	Return the current time.		
	(<i>time</i>)	⇒ <i>t</i>	
Return Values			
	<i>t</i>	An instance of < <i>time</i> >	
Description	Return the current time as by <code>gettimeofday()</code> .		
day->time			function
	Construct a time from a day and seconds within the day.		
	(<i>day->time</i> <i>day</i> <i>sec</i>)	⇒ <i>t</i>	
Arguments			
	<i>day</i>	An instance of < <i>fixnum</i> >	
	<i>sec</i>	An instance of < <i>fixnum</i> >	

	Return Values		
	<i>t</i>	An instance of <time>	
	Description	Construct a time based on the day and the seconds (CUT) within the day. The day is a number based at 1/1/1, so that 1/1/70 is day number 719163. [doesn't work if day < 719163, because times are unix based]	
time-again!			function
	Save current time into object.		
	(time-again! <i>t</i>)	⇒	
	Arguments		
	<i>t</i>	An instance of <time>	
	Description	Side effect <i>t</i> with the current time.	
time->calendar			function
	Get calendar point for given time.		
	(time->calendar <i>t local?</i>)	⇒	<i>year month monthday</i>
	<i>hour min sec yearday weekday</i>		
	Arguments		
	<i>t</i>	An instance of <time>	
	<i>local?</i>	An instance of <boolean>	
	Return Values		
	<i>year</i>	An instance of <fixnum>	
	<i>month</i>	An instance of <fixnum>	
	<i>monthday</i>	An instance of <fixnum>	
	<i>hour</i>	An instance of <fixnum>	
	<i>min</i>	An instance of <fixnum>	
	<i>sec</i>	An instance of <fixnum>	
	<i>yearday</i>	An instance of <fixnum>	
	<i>weekday</i>	An instance of <fixnum>	
	Description	Gets the calendar information for the given time.	
time->string			function
	Render a time to a string.		
	(time->string <i>t fmt local</i>)	⇒	<i>str</i>
	Arguments		
	<i>t</i>	An instance of <time>	
	<i>fmt</i>	An instance of <string>	
	<i>local</i>	An instance of <boolean>	
	Return Values		
	<i>str</i>	An instance of <string>	
	Description	Formats <i>t</i> into a string. <i>fmt</i> is optional, as is <i>local</i> . Default <i>fmt</i> is as by <code>ctime()</code> .	

seconds->interval		function
	Convert number of seconds to interval. (seconds->interval <i>sec</i>)	⇒ <i>dt</i>
Arguments		
<i>sec</i>	An instance of <real>	
Return Values		
<i>dt</i>	An instance of <interval>	
Description	Create an interval corresponding to given number of seconds.	
time->epoch-seconds		function
	Convert time to number of seconds since epoch. (time->epoch-seconds <i>t</i>)	⇒ <i>sec</i>
Arguments		
<i>t</i>	An instance of <time>	
Return Values		
<i>sec</i>	An instance of <real>	
Description	Reverse of epoch-seconds->time.	
interval->seconds		function
	Get number of seconds in interval. (interval->seconds <i>dt</i>)	⇒ <i>sec</i>
Arguments		
<i>dt</i>	An instance of <interval>	
Return Values		
<i>sec</i>	An instance of <real>	
Description	Converts an interval to a number of seconds.	

1.23.2. File Descriptor Functions

fd-read		primop
	Read from an open file. (fd-read <i>fd buf offset len</i>)	⇒ <i>len</i>
Arguments		
<i>fd</i>	An instance of <fixnum>	
<i>buf</i>	An instance of <string>	
<i>offset</i>	An instance of <fixnum>	
<i>len</i>	An instance of <fixnum>	
Return Values		
<i>len</i>	An instance of <fixnum>	
Description	Reads <i>len</i> bytes from an open file into <i>buf</i> at offset <i>offset</i> . Returns the number of bytes read, or #f on error.	
fd-write		primop
	Write to an open file. (fd-write <i>fd buf offset len</i>)	⇒ <i>len</i>
Arguments		

	<i>fd</i>	An instance of <fixnum>	
	<i>buf</i>	An instance of <string>	
	<i>offset</i>	An instance of <fixnum>	
	<i>len</i>	An instance of <fixnum>	
	Return Values		
	<i>len</i>	An instance of <fixnum>	
	Description		
		Writes <i>len</i> bytes to an open file, taken from <i>buf</i> at offset <i>offset</i> . Returns the number of bytes written or #f on error.	
fd-close			primop
	Closes a file. (<i>fd-close</i> <i>fd</i>)		\Rightarrow <i>ok</i>
	Arguments		
	<i>fd</i>	An instance of <fixnum>	
	Return Values		
	<i>ok</i>	An instance of <boolean>	
	Description		
		Closes an open file.	
fd-open			primop
	Open a file. (<i>fd-open</i> <i>path</i> <i>mode</i> <i>perm</i>)		\Rightarrow <i>fd</i>
	Arguments		
	<i>path</i>	An instance of <string>	
	<i>mode</i>	An instance of <fixnum>	
	<i>perm</i>	An instance of <fixnum>	
	Return Values		
	<i>fd</i>	An instance of <fixnum>	
	Description		
		Opens a file. Get <i>mode</i> using <i>make-fd-open-mode</i> . Construct <i>perm</i> using <i>mode-list->bits</i> .	
fd-lseek			primop
	Seek in a file. (<i>fd-lseek</i> <i>fd</i> <i>offset</i> <i>whence</i>)		\Rightarrow <i>offt</i>
	Arguments		
	<i>fd</i>	An instance of <fixnum>	
	<i>offset</i>	An instance of <fixnum>	
	<i>whence</i>	An instance of <fixnum>	
	Return Values		
	<i>offt</i>	An instance of <integer>	
	Description		
		Set pointer in file. Returns new position relative to start. [Note: only works with fixnums for now, limiting usefulness to 500Mb files]	
fd-dup			primop
	Duplicate a file descriptor. (<i>fd-dup</i> <i>fd</i>)		\Rightarrow <i>fd2</i>

	<p>Arguments</p> <p><i>fd</i> An instance of <fixnum></p> <p>Return Values</p> <p><i>fd2</i> An instance of <fixnum></p> <p>Description</p> <p>Copies a file descriptor. Returns the new fd.</p>	
fd-dup2	<p>Dups an fd to a particular fd. (fd-dup2 <i>fromfd tofd</i>) ⇒ <i>rc</i></p> <p>Arguments</p> <p><i>fromfd</i> An instance of <fixnum></p> <p><i>tofd</i> An instance of <fixnum></p> <p>Return Values</p> <p><i>rc</i> An instance of <fixnum></p> <p>Description</p> <p>Copies a file descriptor to a particular one. [Check dup2() for additional semantics like closing <i>tofd</i> or not, etc.]</p>	primop
fd-stat	<p>Stat a file. (fd-stat <i>fd</i>) ⇒ <i>sb</i></p> <p>Arguments</p> <p><i>fd</i> An instance of <fixnum></p> <p>Return Values</p> <p><i>sb</i> An instance of <stat-buf></p> <p>Description</p> <p>Get status on an already open file.</p>	function
fd-truncate	<p>Truncate an open file. (fd-truncate <i>fd len</i>) ⇒</p> <p>Arguments</p> <p><i>fd</i> An instance of <fixnum></p> <p><i>len</i> An instance of <integer></p> <p>Description</p> <p>Sets the size of an open file. Signals an instance of <os-error> if not successful.</p>	primop

1.23.3. Filesystem Functions

scandir	<p>Scan directory for entries. (scandir <i>path</i>) ⇒ <i>entries</i></p> <p>Arguments</p> <p><i>path</i> An instance of <string></p> <p>Return Values</p> <p><i>entries</i> An instance of <list></p> <p>Description</p> <p>Returns a list of entries in the directory.</p>	function
rename		function

	Rename a file. (<code>rename oldpath newpath</code>)	⇒	
	Arguments		
	<i>oldpath</i> An instance of <string>		
	<i>newpath</i> An instance of <string>		
	Description		
	Changes a filename, like unix <code>mv</code> .		
link			function
	Create a new link to a file. (<code>link existpath newpath</code>)	⇒	
	Arguments		
	<i>existpath</i> An instance of <string>		
	<i>newpath</i> An instance of <string>		
	Description		
	Links a new entry to an old file, like unix <code>ln</code> .		
symlink			function
	Create a symbolic link to a file. (<code>symlink existpath newpath</code>)	⇒	
	Arguments		
	<i>existpath</i> An instance of <string>		
	<i>newpath</i> An instance of <string>		
	Description		
	Soft-links a new entry to an old file, like unix <code>ln -s</code> .		
unlink			function
	Delete a link. (<code>unlink path</code>)	⇒	
	Arguments		
	<i>path</i> An instance of <string>		
	Description		
	Removes a filesystem link, like unix <code>rm</code> .		
rmdir			function
	Delete a directory. (<code>rmdir path</code>)	⇒	
	Arguments		
	<i>path</i> An instance of <string>		
	Description		
	Removes an empty directory, like unix <code>rmdir</code> .		
mode-list->bits			function
	Compute bits for mode list. (<code>mode-list->bits lst</code>)	⇒ <i>bits</i>	
	Arguments		
	<i>lst</i> An instance of <list>		
	Return Values		
	<i>bits</i> An instance of <fixnum>		
	Description		
	Computes some mode bits using a list of symbols of the form <code>mode/entity-action</code> , where <i>entity</i> is one of		

user, group, anyone, and *action* is one of read, write, or execute. Also `mode/set-user-id` and `mode/set-group-id` are understood.

chmod	<p>Changes permissions. (<code>chmod path mode</code>)</p> <p>Arguments</p> <p><i>path</i> An instance of <string></p> <p><i>mode</i> An instance of <fixnum></p> <p>Description</p> <p>Changes permissions on a file. <i>mode</i> is from <code>mode-list->bits</code>.</p>	function
chown	<p>Changes file ownership. (<code>chown path uid gid</code>)</p> <p>Arguments</p> <p><i>path</i> An instance of <string></p> <p><i>uid</i> An instance of <fixnum></p> <p><i>gid</i> An instance of <fixnum></p> <p>Description</p> <p>Changes file ownership. Probably requires root authority.</p>	function
file-truncate	<p>Truncate a file. (<code>file-truncate path len</code>)</p> <p>Arguments</p> <p><i>path</i> An instance of <string></p> <p><i>len</i> An instance of <integer></p> <p>Description</p> <p>Sets the size of the given file. Signals an instance of <os-error> if not successful.</p>	primop
file-access?	<p>Check for file access. (<code>file-access? path mode</code>)</p> <p>Arguments</p> <p><i>path</i> An instance of <string></p> <p><i>mode</i> An instance of <fixnum></p> <p>Return Values</p> <p><i>ok</i> An instance of <boolean></p> <p>Description</p> <p>Checks to see if the current process has the given kind of access to the given file. Use <code>access-mask</code> to construct <i>mode</i>.</p>	function

1.23.4. stat Functions

stat	<p>Stat a file. (<code>stat path</code>)</p> <p>Arguments</p> <p><i>path</i> An instance of <string></p>	function
-------------	---	-----------------

	Return Values			
	<i>buf</i>	An instance of <stat-buf>		
	Description	Construct a stat buffer for the given file. If <i>path</i> is a soft link, the file referred to is stat'ed.		
lstat				function
	Stat a link.			
	(<i>lstat path</i>)		\Rightarrow <i>buf</i>	
	Arguments			
	<i>path</i>	An instance of <string>		
	Return Values			
	<i>buf</i>	An instance of <stat-buf>		
	Description	Like <i>stat</i> , but gets status on the link itself if <i>path</i> refers to a symbolic link.		
stat-type				function
	Get the type of the stat buffer.			
	(<i>stat-type buf</i>)		\Rightarrow <i>type</i>	
	Arguments			
	<i>buf</i>	An instance of <stat-buf>		
	Return Values			
	<i>type</i>	An instance of <symbol>		
	Description	Returns a symbol indicating the type of the stat'ed file. The returned symbol is one of directory, regular, fifo, character-special, or block-special. #f may also be returned if the type is not recognized (ie, for symbol links stat'ed using <i>lstat</i> .)		
stat-mode				function
	Get the mode of the stat'd file.			
	(<i>stat-mode buf</i>)		\Rightarrow <i>mode</i>	
	Arguments			
	<i>buf</i>	An instance of <stat-buf>		
	Return Values			
	<i>mode</i>	An instance of <fixnum>		
	Description	Returns the mode bits for the given stat buffer.		
stat-owner				function
	Get the owner of the file.			
	(<i>stat-owner buf</i>)		\Rightarrow <i>mode</i>	
	Arguments			
	<i>buf</i>	An instance of <stat-buf>		
	Return Values			
	<i>mode</i>	An instance of <fixnum>		
	Description	Returns the owner id (uid) for the given stat buffer.		
stat-eq?				function
	Check if two buffers refer to same object.			
	(<i>stat-eq? a b</i>)		\Rightarrow <i>same</i>	
	Arguments			
	<i>a</i>	An instance of <stat-buf>		

	<i>b</i>	An instance of <stat-buf>	
Return Values	<i>same</i>	An instance of <boolean>	
Description	Checks to see if two stat buffers refer to the same filesystem object (ie, they are on the same device and have the same inode. Hence, this should probably be called stat-eqv? instead)		
stat-id->hash			function
	Compute hash number for stat buffer.		
	(<i>stat-id->hash b</i>)		\Rightarrow <i>hash</i>
Arguments	<i>b</i>	An instance of <stat-buf>	
Return Values	<i>hash</i>	An instance of <fixnum>	
Description	Computes a hash number for a stat buffer based on the identity of the stat'd file (ie, using the device number and inode number).		
stat-id-vector			function
	Return a vector of file identity info.		
	(<i>stat-id-vector b</i>)		\Rightarrow <i>info</i>
Arguments	<i>b</i>	An instance of <stat-buf>	
Return Values	<i>info</i>	An instance of <vector>	
Description	Returns a four-element vector containing identity information for the given stat buffer. The elements are, in order, device high 16 bits, device number low 16 bits, inode high 16, and inode low 16.		
stat-mtime			function
	Return mtime for stat buffer.		
	(<i>stat-mtime b</i>)		\Rightarrow <i>t</i>
Arguments	<i>b</i>	An instance of <stat-buf>	
Return Values	<i>t</i>	An instance of <time>	
Description	Returns the mtime for the stat'd file.		
stat-times			function
	Returns file times.		
	(<i>stat-times b</i>)		\Rightarrow <i>mtime atime ctime</i>
Arguments	<i>b</i>	An instance of <stat-buf>	
Return Values	<i>mtime</i>	An instance of <time>	
	<i>atime</i>	An instance of <time>	
	<i>ctime</i>	An instance of <time>	
Description	Returns the three file times for the stat'd file; modification time, access time, and creation time.		

stat-size	<p>Returns file size. (stat-size <i>b</i>)</p> <p>Arguments</p> <p><i>b</i> An instance of <stat-buf></p> <p>Return Values</p> <p><i>size</i> An instance of <integer></p> <p>Description</p> <p>Returns the size of the file in bytes.</p>	function
	⇒ <i>size</i>	
stat-directory?	<p>Determine if a stat buffer belongs to a directory. (stat-directory? <i>sb</i>)</p> <p>Arguments</p> <p><i>sb</i> An instance of <stat-buf></p> <p>Return Values</p> <p><i>result</i> An instance of <boolean></p> <p>Description</p> <p>Returns #t if the filesystem object whose stat buffer is <i>sb</i> is a directory.</p>	function
	⇒ <i>result</i>	
stat-file?	<p>Determine if a stat buffer belongs to a file. (stat-file? <i>sb</i>)</p> <p>Arguments</p> <p><i>sb</i> An instance of <stat-buf></p> <p>Return Values</p> <p><i>result</i> An instance of <boolean></p> <p>Description</p> <p>Returns #t if the filesystem object whose stat buffer is <i>sb</i> is a file object.</p>	function
	⇒ <i>result</i>	
stat-access?	<p>Determine if an entity can access the stat'ed object in a given mode. (stat-access? <i>sb entity mode</i>)</p> <p>Arguments</p> <p><i>sb</i> An instance of <stat-buf></p> <p><i>entity</i> An instance of <symbol></p> <p><i>mode</i> An instance of <symbol></p> <p>Return Values</p> <p><i>result</i> An instance of <boolean></p> <p>Description</p> <p>Where entity may be one of <code>owner</code>, <code>group</code>, or <code>world</code>, and mode is one of <code>read</code>, <code>write</code>, <code>execute</code>.</p>	function
	⇒ <i>result</i>	
make-fd-open-mode	<p>Create an open mode for fd-open. (make-fd-open-mode <i>mainmode</i>)</p> <p>Arguments</p> <p><i>mainmode</i> An instance of <symbol></p> <p><i>option</i> Instances of <symbol></p> <p>Return Values</p> <p><i>mode</i> An instance of <fixnum></p>	function
	⇒ <i>mode</i>	

Description

Construct a unix "open" mode for use by **fd-open**. *mainmode* is one of `read`, `write`, and `read-write`. The options are taken from `append`, `create`, `exclusive`, and `truncate`.

access-mask**syntax**

Create an access mask for file-access?.

(**access-mask** *accessflag*)

⇒ *accessbits*

Arguments

accessflag A name

Return Values

accessbits An instance of <fixnum>

Description

Creates an access mask for the access types `read`, `write`, `execute`, and `exist`.

1.23.5. Network Functions**make-fd-set****function**

Create a file descriptor set for use by **fd-select**.

(**make-fd-set** *reads writes exceptions*)

⇒ *set*

Arguments

reads An instance of <list>

writes An instance of <list>

exceptions An instance of <list>

Return Values

set An instance of <fd-select-set>

Description

Constructs a file descriptor set (which is really made up of three sets, corresponding to watching for readability, writability, and exceptions). The resulting set can be used in the **fd-select** function.

fd-select**function**

Test for operations.

(**fd-select** *delays set*)
haveexception

⇒ *readable writable*

Arguments

delays An instance of <fixnum>

set An instance of <fd-select-set>

Return Values

readable An instance of <list>

writable An instance of <list>

haveexception An instance of <list>

Description

Uses the `select()` system function to determine which of the file descriptors in the given *set* can be operated on. Returns three lists of file descriptors.

set-socket-option**function**

Sets a socket option.

(**set-socket-option** *fd level option value*)

⇒

Arguments

fd An instance of <fixnum>

level An instance of <symbol>

option An instance of <symbol>

value An instance of <object>

Description

Sets a socket option using `setsockopt()`. The required type of *value* depends on the option being set.

get-socket-option**function**

Get a socket option value.

`(get-socket-option fd level option)` \Rightarrow *value*

Arguments

fd An instance of <fixnum>

level An instance of <symbol>

option An instance of <symbol>

Return Values

value An instance of <object>

Description

Gets the value of a socket option using `getsockopt()`. The returned type of *value* depends on the option being set.

The valid values for *level* are:

Option Level
level/socket
level/ip
level/tcp
level/udp

The valid values for *option* are:

Socket Option	Value Type
socket/debug	<boolean>
socket/reuse-addr	<boolean>
socket/keep-alive	<boolean>
socket/dont-route	<boolean>
socket/linger	<integer> or #f
socket/broadcast	<boolean>
socket/oob-inline	<boolean>
socket/send-buffer	<integer>
socket/receive-buffer	<integer>
socket/type	<integer>
socket/error	<integer>

inet-addr->string **function**

Render an internet address.

(*inet-addr->string in*) ⇒ *str*

Arguments

in An instance of <inet-addr>

Return Values

str An instance of <string>

Description

Renders an internet address in dotted list form (using `inet_ntoa()`).

string->inet-addr **function**

Create an internet address object.

(*string->inet-addr ip*) ⇒ *in*

Arguments

ip An instance of <string>

Return Values

in An instance of <inet-addr>

Description

Parses *ip* (in dotted quad format) and creates a <inet-addr> object.

make-inet-socket-addr **function**

Create an internet socket address.

(*make-inet-socket-addr host port*) ⇒ *addr*

Arguments

host An instance of <inet-addr>

port An instance of <fixnum>

Return Values

addr An instance of <inet-socket-addr>

Description

Creates a socket address using a host address and a port number.

inet-socket-addr-parts **function**

	Extract socket address parts. (<i>inet-socket-addr-parts addr</i>)	⇒ <i>in port</i>	
	Arguments		
	<i>addr</i>	An instance of <inet-socket-addr>	
	Return Values		
	<i>in</i>	An instance of <inet-addr>	
	<i>port</i>	An instance of <fixnum>	
	Description		
	Extracts the two parts (host and port) of an internet socket address.		
socket-address-family->integer			function
	Identify socket address family. (<i>socket-address-family->integer fam-name</i>)	⇒ <i>fam</i>	
	Arguments		
	<i>fam-name</i>	An instance of <symbol>	
	Return Values		
	<i>fam</i>	An instance of <fixnum>	
	Description		
	Find local name (integer) for socket address family. Supports <i>address-family/unix</i> and <i>address-family/internet</i> .		
socket-type->integer			function
	Identify socket type. (<i>socket-type->integer type-name</i>)	⇒ <i>type</i>	
	Arguments		
	<i>type-name</i>	An instance of <symbol>	
	Return Values		
	<i>type</i>	An instance of <fixnum>	
	Description		
	Find local name (integer) for socket address type. Supports <i>socket-type/stream</i> , <i>socket-type/datagram</i> , and <i>socket-type/raw</i> .		
socket-create			function
	Creates a socket. (<i>socket-create fam socket-type protocol</i>)	⇒ <i>fd</i>	
	Arguments		
	<i>fam</i>	An instance of <fixnum>	
	<i>socket-type</i>	An instance of <fixnum>	
	<i>protocol</i>	An instance of <fixnum>	
	Return Values		
	<i>fd</i>	An instance of <fixnum>	
	Description		
	Opens a socket of the given type in the given address family, using the given protocol (<i>protocol</i> is usually zero, which means the default protocol).		
socket-listen			function
	Listen on a socket. (<i>socket-listen sock queue-length</i>)	⇒ <i>ok</i>	
	Arguments		
	<i>sock</i>	An instance of <fixnum>	

	<i>queue-length</i>	An instance of <fixnum>	
	Return Values		
	<i>ok</i>	An instance of <boolean>	
	Description		
		Configure a socket for accepting connections.	
socket-bind/unix			function
		Bind a socket to a unix address.	
		(<i>socket-bind/unix sock path</i>)	⇒ <i>ok</i>
	Arguments		
	<i>sock</i>	An instance of <fixnum>	
	<i>path</i>	An instance of <string>	
	Return Values		
	<i>ok</i>	An instance of <boolean>	
	Description		
		Binds the socket to a unix domain address. A unix domain address is actually a filename that refers to a special socket file.	
socket-bind/inet			function
		Bind a socket to an internet address.	
		(<i>socket-bind/inet sock port</i>)	⇒ <i>ok</i>
	Arguments		
	<i>sock</i>	An instance of <fixnum>	
	<i>port</i>	An instance of <fixnum>	
	Return Values		
	<i>ok</i>	An instance of <boolean>	
	Description		
		Binds the socket to an internet socket address.	
socket-bind/inet-sockaddr			function
		Bind a socket to a socket address (IP and port)	
		(<i>socket-bind/inet-sockaddr sockfd sockaddr</i>)	⇒
	Arguments		
	<i>sockfd</i>	An instance of <fixnum>	
	<i>sockaddr</i>	An instance of <inet-socket-addr>	
	Description		
		Binds the socket specified by <i>sockfd</i> to an internet socket address, including the IP address and port number. This is useful when writing a server for a multi-homed host or host with IP aliases. For example, an HTTP server present on only one or a subset of IP addresses of a server. This function allows the application to bind to only one IP address, or bind different socket file descriptors to different IP addresses.	
socket-accept			function
		Accept a connection.	
		(<i>socket-accept sock</i>)	⇒ <i>fd</i>
	Arguments		
	<i>sock</i>	An instance of <fixnum>	
	Return Values		
	<i>fd</i>	An instance of <fixnum>	

	Description		
		Accepts the next connection from the socket (which was previously told to <code>socket-listen</code> .)	
socket-connect/inet			function
		Connect to remote address. (<code>socket-connect/inet sock port hostid</code>)	\Rightarrow <i>ok</i>
	Arguments		
	<i>sock</i>	An instance of <fixnum>	
	<i>port</i>	An instance of <fixnum>	
	<i>hostid</i>	An instance of <string>	
	Return Values		
	<i>ok</i>	An instance of <boolean>	
	Description		
		Connect the given socket to a remote address.	
host-name->address			function
		Look up a host by name. (<code>host-name->address hostname</code>)	\Rightarrow <i>addr aliases</i>
	Arguments		
	<i>hostname</i>	An instance of <string>	
	Return Values		
	<i>addr</i>	An instance of <string>	
	<i>aliases</i>	An instance of <list>	
	Description		
		Look up a host name and return its address (dotted list format) and any aliases (<i>aliases</i> includes the canonical name as the first element)	
host-address->name			function
		Look up a host by address. (<code>host-address->name addr</code>)	\Rightarrow <i>addr aliases</i>
	Arguments		
	<i>addr</i>	An instance of <string>	
	Return Values		
	<i>addr</i>	An instance of <string>	
	<i>aliases</i>	An instance of <list>	
	Description		
		Returns the same thing as <code>host-name->address</code> , but looks up the host by address (internet dotted list notation) instead.	
recv-from			function
		Receive a packet. (<code>recv-from sock buf offset len peek? out-of-band? peer-class</code>)	\Rightarrow <i>len peer</i>
	Arguments		
	<i>sock</i>	An instance of <fixnum>	
	<i>buf</i>	An instance of <string>	
	<i>offset</i>	An instance of <fixnum>	
	<i>len</i>	An instance of <fixnum>	

<i>peek?</i>	An instance of <boolean>
<i>out-of-band?</i>	An instance of <boolean>
<i>peer-class</i>	An instance of <<class>>
Return Values	
<i>len</i>	An instance of <fixnum>
<i>peer</i>	An instance of <sockaddr>

Description

Receive a packet from a datagram port. Returns #f instead if an error occurred.

send-to**function**

Send a packet.

(**send-to** *sock buf offset len out-of-band? to*) \Rightarrow *len*

Arguments

<i>sock</i>	An instance of <fixnum>
<i>buf</i>	An instance of <string>
<i>offset</i>	An instance of <fixnum>
<i>len</i>	An instance of <fixnum>
<i>out-of-band?</i>	An instance of <boolean>
<i>to</i>	An instance of <sockaddr>

Return Values

<i>len</i>	An instance of <fixnum>
------------	-------------------------

Description

Sends a packet out a datagram socket. Returns #f if an error occurred.

1.23.6. Miscellaneous**errno****primop**

Return errno.

(**errno**) \Rightarrow *errno*

Return Values

<i>errno</i>	An instance of <fixnum>
--------------	-------------------------

Description

Return the last error. Deprecated, but still needed as not everything signals errors (e.g., `fd-open` still returns status info, which breaks `w/threads`)

(This Page Intentionally Left Blank)

This chapter describes the functions available from the `unixm` module.

<code>get-env</code>		function
	Get an environment variable. (<code>get-env key</code>)	\Rightarrow <i>val</i>
Arguments	<i>key</i>	An instance of <code><string></code>
Return Values	<i>val</i>	An instance of <code><string-or-false></code>
Description	Get the value of an environment variable. Returns <code>#f</code> if the variable is not defined.	
<code>set-env</code>		function
	Set an environment variable. (<code>set-env key value</code>)	\Rightarrow
Arguments	<i>key</i>	An instance of <code><string></code>
	<i>value</i>	An instance of <code><string></code>
Description	Set the environment variable named <i>key</i> to the value <i>value</i> .	
<code>reset-env</code>		function
	Replace process environment. (<code>reset-env lst</code>)	\Rightarrow
Arguments	<i>lst</i>	An instance of <code><list></code>
Description	Replaces the process environment with the contents of the given list, which is a list of key-value pairs.	
<code>unset-env</code>		function
	Remove <i>key</i> from environment. (<code>unset-env key</code>)	\Rightarrow
Arguments	<i>key</i>	An instance of <code><string></code>
Description	Removes <i>key</i> from the process environment.	
<code>mkdir</code>		function
	Create directory (<code>mkdir dir</code>)	\Rightarrow

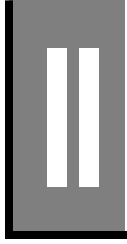
	Arguments		
	<i>dir</i>	An instance of <string>	
	Description	Creates a directory (unix mkdir())	
mkdirs			function
	Create directories		
	(mkdirs <i>dir</i>)		⇒ <i>dir</i>
	Arguments		
	<i>dir</i>	An instance of <string>	
	Return Values		
	<i>dir</i>	An instance of <string>	
	Description	Creates <i>dir</i> and any non-existing parent directories.	
system			function
	Invoke command using sh		
	(system <i>str</i>)		⇒ <i>rc</i>
	Arguments		
	<i>str</i>	An instance of <string>	
	Return Values		
	<i>rc</i>	An instance of <integer>	
	Description	This is the standard unix system() function, which invokes the given command as by sh -c.	
		Returns the exit code of the subprocess.	
getrusage			function
	Get process resource usage.		
	(getrusage)		⇒ <i>usage</i>
	Return Values		
	<i>usage</i>	An instance of <vector>	
	Description	Currently there is no way to get the rusage of a child process.	
		The vector contains 8 elements, which are: process user time (an <interval>), process system time (an <interval>), average memory (shared), average memory (unshared in data segment), input operations, output operations, number of messages sent, and number of messages received.	
open-input-process*			function
	Open an input process.		
	(open-input-process* <i>arg</i>)		⇒ <i>p</i>
	Arguments		
	<i>arg</i>	An instance of <string>s	
	Return Values		
	<i>p</i>	An instance of <input-port>	
	Description	This is functionally similar to open-input-process which uses popen(). However, this mechanism is more direct in its invocation of the subprogram (since the subcommand is not invoked by sh.). On the downside, you can't use shell expansions in the command, either. Like the difference between 'fork/exec' and 'system'.	
sleep			function

	Sleep for the given amount of time. (<i>sleep sleep_time</i>)	⇒	
Arguments	<i>sleep_time</i> An instance of <number>		
Description	Sleeps for the given number of seconds (may be a float if the system has <code>usleep()</code> , otherwise a rounds it off to the nearest whole number of seconds)		
fork			function
	Fork a new process (<i>fork</i>)	⇒ <i>pid</i>	
Return Values	<i>pid</i> An instance of <integer>		
Description	Create a new process. Returns the subprocess's id in the parent, and #f in the child.		
wait			function
	Wait for process completion. (<i>wait</i>)	⇒ <i>pid status</i>	
Return Values	<i>pid</i> An instance of <integer>		
	<i>status</i> An instance of <integer>		
Description	Wait for a child process to complete. If the <code>wait()</code> system call returns -1, this function returns #f #f instead.		
wait-for			function
	Wait for a particular subprocess to complete. (<i>wait-for pid</i>)	⇒ <i>pid status</i>	
Arguments	<i>pid</i> An instance of <raw-int>		
Return Values	<i>pid</i> An instance of <integer>		
	<i>status</i> An instance of <integer>		
Description	Wait for the given process to complete.		
getpid			function
	Get the current process id. (<i>getpid</i>)	⇒ <i>pid</i>	
Return Values	<i>pid</i> An instance of <integer>		
Description	Returns the current process id.		
getpgrp			function
	Get process group id. (<i>getpgrp</i>)	⇒ <i>pid</i>	
Return Values	<i>pid</i> An instance of <integer>		
Description	Returns the process group id (the process id of the group leader?).		

getppid			function
	Get parent process id.		
	(<i>getppid</i>)	\Rightarrow <i>pid</i>	
	Return Values		
	<i>pid</i>	An instance of <integer>	
	Description		
	Returns the process id of this process's parent		
getuid			function
	Get user id.		
	(<i>getuid</i>)	\Rightarrow <i>uid</i>	
	Return Values		
	<i>uid</i>	An instance of <integer>	
	Description		
	Returns the user id of the owner of the current process.		
geteuid			function
	Get effective user id.		
	(<i>geteuid</i>)	\Rightarrow <i>uid</i>	
	Return Values		
	<i>uid</i>	An instance of <integer>	
	Description		
	Returns the user id of the <i>effective</i> owner of the current process.		
getlogin			function
	Get the name of the login user.		
	(<i>getlogin</i>)	\Rightarrow <i>login</i>	
	Return Values		
	<i>login</i>	An instance of <string>	
	Description		
	Returns the login name of the process's owner, or #f if it cannot be determined.		
getpw			function
	Get the password entry for the given entity.		
	(<i>getpw who</i>)	\Rightarrow <i>pwent</i>	
	Arguments		
	<i>who</i>	An instance of <string-or-fixnum>	
	Return Values		
	<i>pwent</i>	An instance of <vector>	
	Description		
	Returns a vector describing the passwd entry for the given user. <i>who</i> may be either a string, in which case it is interpreted as a login name, or an integer, in which case it is interpreted as a user id.		
	Returns #f if the lookup fails.		
	The returned vector has 7 elements: The user login name, the user id, the user's primary group id, their home directory, their shell, their encrypted passwd, and their GECOS data. Some data may not be available, in which case #f is present in that slot.		
exec*			function
	Transform current process into a new program.		
	(<i>exec* path argv env</i>)	\Rightarrow	
	Arguments		
	<i>path</i>	An instance of <string>	

	<i>argv</i>	An instance of <vector>	
	<i>env</i>	An instance of <vector>	
	Description		
		Tail-call a process. This is a wrapper for the standard <code>execve()</code> system call, which transforms the current process into a new program, supplying an explicit <i>argv</i> and <i>environ</i> to the new process.	
		Signals an error if the <code>exec</code> fails.	
exec			function
		Exec a new program.	
		(<code>exec path arg</code>)	\Rightarrow
	Arguments		
	<i>path</i>	An instance of <string>	
	<i>arg</i>	An instance of <string>s	
	Description		
		Execs a new program (a wrapper for the <code>execv()</code> system call). The new program's environment is the same as the current process's environment.	
pipe			function
		Create a pipe pair.	
		(<code>pipe</code>)	\Rightarrow <i>fd1 fd2</i>
	Return Values		
	<i>fd1</i>	An instance of <integer>	
	<i>fd2</i>	An instance of <integer>	
	Description		
		Create a pair of file descriptors which are connected by a unix pipe.	
kill			function
		Send a signal to a process.	
		(<code>kill pid sig</code>)	\Rightarrow <i>rc</i>
	Arguments		
	<i>pid</i>	An instance of <fixnum>	
	<i>sig</i>	An instance of <fixnum>	
	Return Values		
	<i>rc</i>	An instance of <integer>	
	Description		
		Send the signal <i>sig</i> to the process <i>pid</i> .	
hostname			function
		Determine the hostname of the current machine.	
		(<code>hostname</code>)	\Rightarrow <i>hostname</i>
	Return Values		
	<i>hostname</i>	An instance of <string>	
	Description		
		Returns the hostname of the current machine (<code>unix gethostname()</code>).	

(This Page Intentionally Left Blank)



Technical Guide
