SISC for Seasoned Schemers

Scott G. Miller

Matthias Radestock

SISC for Seasoned Schemers

by Scott G. Miller and Matthias Radestock

Copyright © 2002-2006 Scott G. Miller,Matthias Radestock

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

The source code refers to the DocBook XML source files. The object code refers to the generated HTML, PostScript, PDF, and other forms.

This documentation is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License (in Appendix E) along with this documentation; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Table of Contents

List of Tables

Chapter 1. Introduction

SISC [1] is a lightweight, platform independent Scheme system whose primary goals are rapid execution of the complete R^5RS and future Scheme standards, plus a useful superset for real-world application development.

SISC's development progresses in two directions, to improve the core interpreter to be simpler, more elegant, and more efficient; and to add necessary functionality through extensions that do not complicate the core.

SISC as a project began as the successor to the Lightweight Interpreter of Scheme Code (*LISC*). *LISC* was a small, stack-based almost R4RS compliant Scheme. *SISC* was born out of the desire to create an interpreter that was of a similar footprint to *LISC*, but which executed Scheme code much faster, complied fully to the R^5RS standard, and which wasn't limited by the stack-based model. *SISC* met these goals very quickly, and has since progressed in active development to be a competitive Scheme system. As a successor to *LISC* the interpreter was named the Second Interpreter of Scheme Code.

1.1. Features

- Full R^5RS compliance
- Efficient number tower, with support for integers, floating-point numbers, rationals, and complex numbers of arbitrary precision
- Lightweight Scheme engine, implementing all R^5RS functionality in approximately 10,000 lines of native code plus approximately 5,000 lines of Scheme code.
- Flexible runtime extensibility through a scopeable module system, which may add arbitrary bindings and new first-class types.

1.2. About this document

This document explains the *SISC* Scheme system. It assumes knowledge of the Scheme language. As such, when discussing the Scheme language, we will focus primarily on differences between the Scheme standard and the language implemented by *SISC*.

Secondly, *SISC* implements the R^5RS standard. As such, any code written to that standard should run without reading any further.

1.2.1. About procedure documentation

Throughout this document, procedures will be defined using the following syntax:

procedure: (**`function-name`** `required-argument [optional-argument] [rest-argument] ...`) => return value

Description of the procedure's semantics.

A procedure is any function that takes zero or more arguments and returns a value.

parameter: (**`parameter-name`** `[new-parameter-value]`)

Description of the parameter.

A Parameter is a special function that is used to store a value in the dynamic-environment. When given no arguments, a parameter will return its current value. When provided an argument, the value of the parameter is set to the value of that argument.

syntax: (**`syntactic-keyword`** `structure ...`)

Description of the syntactic transform.

Syntax refers to a syntactic keyword that when compiled will be transformed into some expression composed only of basic Scheme forms. `let`, `cond`, and `or` are all examples of syntactic forms.

Procedures, parameters, and syntax may take one or more arguments. The name of an argument in a description summarizes the semantics of the expected argument. The type of an argument, when not clear, should be described in the procedure summary paragraph that follows the prototype. If the argument is enclosed in square brackets ([]), that argument is optional, and may be omitted. If an argument is optional and is followed by ellipses (...), then the argument is a *rest* argument, and may be satisfied by zero or more values.

Finally, some of the functions described in this document are encapsulated in *modules* (see Chapter 10). Sections that describe functions in modules will have a *requires* statement indicating the expression that must be evaluated to make those functions available in the Scheme environment. The requires statement will appear as follows:

Requires: **(import *module-name*)**

module-name is will be a name uniquely identifying the name of the module that contains the functions.

1.3. Where to obtain this document

This document is produced from DocBook sources and is made available in HTML, Adobe Acrobat (PDF), and PostScript forms. Other formats, and up-to-date versions of this document should be available from the *SISC* website, *http://sisc.sourceforge.net* (http://sisc.sourceforge.net).

The DocBook sources are available from the same site, packaged in the source distribution of *SISC*, and available from the project's CVS tree. Details for accessing both are linked off the project site.

Starting with *SISC* version 1.8.4, the HTML render of this manual is included in the full binary distributions of *SISC*.

1.4. Version Applicability

This document describes functionality present in *SISC* version 1.16.4. Some of the document may apply to previous and future versions. When in doubt, the source of this document can be found in the *SISC* CVS repository, or in the packaged source distribution that should be available wherever the binary distribution was obtained.

Notes

1. The pronunciation is written as `sIsk` in the International Phonetic Alphabet. This is similar to the English word 'whisk'.

Chapter 2. Installation and Invocation

2.1. Required Environment

SISC primarily targets the Java Virtual Machine, and the Java v1.3 and higher class libraries. The 1.2 libraries and VM are required due to a reliance on memory reference functionality used to properly garbage collect unused symbols while still maintaining pointer equality between those that remain active, while the 1.3 libraries are needed for the proxy functionality of the Java bridge library.

SISC does not require any particular operating system, though the official distribution currently provides launching assistance for Windows and Unix based systems. Theoretically any machine that supports Java 1.3 or higher should be able to run *SISC*.

2.2. The Read-Eval-Print-Loop

As in most Scheme systems, *SISC*'ss primary human interface is a REPL, or Read-Eval-Print-Loop. *SISC* presents the user with a prompt ('`#;>`'), then reads an s-expression from the console. *SISC* then evaluates the s-expression, producing a value or error, which it then prints. Finally the process begins again with a new prompt.

The process terminates with a forced kill of the JVM (with Control-C or similar), or when an end of file is detected on the console. Also, the `exit` procedure may be used to exit the current REPL.

procedure: (**`exit`** `[return-value]`) => does not return

Instructs the current REPL to exit. If the optional return value is given, the value will be passed out of the REPL to the calling code or environment.

Evaluation may be prematurely interrupted on some platforms by sending SIGQUIT (usually by pressing Control-C) to the *SISC* process. If supported, this will cause an interrupted error to be raised at whichever expression is currently being evaluated in the REPL thread. If not caught, this will cause the error to be raised at the console and a new prompt will be displayed. Issuing the signal twice will terminate *SISC*.

2.3. Display Conventions

Executing or loading code in the REPL ordinarily produces a value result which is displayed during the Print phase. However, two other message types may be displayed. First, if an error is raised that is not handled by the running program, the message will be described in one of several forms, depending on what combinations of a location, a message, and a parent message the error has:

```
Error.

Error in <error-location>.

Error: <description>

Error in <error-location>: <description>
Error in nested call.
<nested-error>

Error in nested call: <description>
<nested-error>

Error in nested call from <error-location>.
<nested-error>

Error in nested call from <error-location>: <description>
<nested-error>
```

The location of an error often refers to the name of a procedure, syntax, or parameter. Errors that do not contain a location often originate in an anonymous function. Nested errors occur when one function makes a call to another function and the second raises an error. If the first function wishes, it may catch this error and generate its own, with the second functions error as a parent. Thus a nested error consists of an error message on the first line, and the nested error message or messages on following lines.

In addition to errors, it is possible for code to produce warnings during compilation or run-time. A warning is a message to the user about a condition that is unusual but not terminal to the program flow. The compiler, for example, does a minimal amount of sanity checking on code and may produce warnings about code that has the potential to raise errors during execution. Warnings are always distinguished from ordinary messages by surrounding braces ({}) and starting with the word 'warning'.

```
{warning: <description>}
```

2.4. Running *SISC*

SISC's official distribution provides a startup script for Windows and Unix based hosts. Simply executing this script starts the *SISC* REPL. Zero or more Scheme source files can also be specified on the command line after the script name, which will be loaded in order before the REPL prompt is displayed.

It may be desirable to pass options to the underlying JVM. This can be done by setting the `JAVAOPT` environment variable to the options you wish to pass to the JVM. This includes switches for heap size, system properties, etc. *SISC* itself has a number of properties that can affect its operation. See Section 2.4.2 for a list of these properties and their meanings.

2.4.1. Command-line Switches

SISC can, in addition to loading Scheme source programs, also accept a few command-line switches to change its behavior. Any non-switch argument is considered a Scheme source file to load on startup, until the *end of options* characters "--" are reached. Any item after those characters are considered arguments to the function specified in the "call-with-args" switch.

All command-line switches have both a short and long form. These forms are equivalent in meaning.

Table 2-1. *SISC* Command line Switches

Long Switch	**Short Switch**	**Switch meaning**
`--call-with-args <name>`	`-c <name>`	Call the top-level procedure `name` with the remaining command-line arguments after the "--" sequence.
`--eval <expression>`	`-e <expression>`	Evaluate the provided expression.
`--no-repl`	`-x`	Exit after loading all the Scheme source files and processing all command-line switches.
`--heap <heap-file>`	`-h <heap-file>`	File containing pre-compiled code and data for a complete Scheme top-level environment. This parameter is mandatory if the heap cannot be located automatically.
`--properties <config-file>`	`-p <config-file>`	Specifies a Java properties file that contains application properties. Typically some of these properties define defaults for configuration parameters (Section 2.4.2. The file can be specified as a URL.
`--listen [<host>:]<port>`	`-l [<host>:]<port>`	Listen on host/port for REPL connections, i.e. connecting to the specified host/port will create a new REPL.

The order of processing the command line is as follows:

1. Process the entire command line, noting the settings of each switch and accumulating all Scheme source files and arguments after the end of options sequence.
2. Load the heap file.
3. Load each Scheme source file found in the order they occured on the command line. Note whether any errors occurred.
4. If present, evaluate the expression in an `--eval` switch. Note its success or failure.
5. If present, apply the named function in a `--call-with-args` switch to the arguments after the end of options sequence. Note its return value.
6. If `--no-repl` was not specified, invoke the REPL, otherwise exit.
7. If the REPL was run if its return value is an integer, return that integer as *SISC*'s overall return code. If the REPL was not run, and any return code supporting step above was run, return the most recent return code. If no return code step was performed, but a success/failure step was performed, return 1 if any failures occured, 0 otherwise.

2.4.2. Configuration Parameters

SISC's behaviour is affected by a number of configuration parameters. These fall into two categories:

1. Static configuration parameters that can only be specified at system startup and apply across all *SISC* applications that use the same classloader.
2. Dynamic configuration parameters that can be altered on a per-thread basis without impacting other threads. These kind of parameters are also called "thread locals", or, in Scheme terminology, dynamic parameters.

2.4.2.1. Static Parameters

Static configuration parameters can be set using Java system properties. Their value can be retrieved with a Scheme function, but it is not possible to alter it. Static configuration parameters default to pre-defined system-internal settings if left unspecified.

SISC has the following static configuration parameters:

Java property [a]	**Scheme parameter**	**default**	**description**
permitInterrupts	`permit-interrupts`	true	If set to `true`, `thread/interrupt` is permitted to interrupt running Scheme code, in addition to sending an interrupt signal to the host language.
minFloatPrecision	`min-float-precision`	16	Specifies the minimum precision, in decimal places, to be maintained by the Quantity lib if using arbitrary precision floats.
maxFloatPrecision	`max-float-precision`	32	Specifies the maximum precision, in decimal places, to be maintained by the Quantity lib if using arbitrary precision floats.

Notes:
a. The names of all properties for configuration parameters must be prefixed with sisc..

When static configuration parameters are retrieved with their associated Scheme function, the value is of the type specified for the parameter. See Section 2.4.2.3.

2.4.2.2. Dynamic Parameters

There are four ways to specify a configuration parameter, in decreasing order of precedence:

1. *Invoking a scheme function*. Dynamic configuration parameters are special Scheme parameters (see Section 3.1.8. Invoking the parameter with a value sets the parameter for the current thread without affecting other dynamic contexts / threads.
2. *Defining an application property*. A single *SISC* runtime can host multiple applications simultaneously. Application properties define default values for dynamic configuration parameters across all dynamic contexts / threads of an application. They can be specified at application initialisation time. See Section 8.1. When *SISC* is started from the command line, the location of a Java properties file containing application properties can be specified with a command line option. See Table 2-1.
3. *Defining a Java system property*. Java system properties define default values for dynamic configuration parameters that apply across all applications inside a single *SISC* runtime.
4. *System defaults*. All dynamic configuration parameters have a reasonable default value.

SISC has the following dynamic configuration parameters:

Java property [a]	**Scheme parameter**	**default**	**description**
caseSensitive	`case-sensitive`	false	Determines whether symbols read via the Scheme reader are to be treated case sensitively.
characterSet	`character-set`	"UTF8"	Defines the default character set used by character ports (see Section 5.1.3) if no character set is otherwise specified.
emitAnnotations	`emit-annotations`	true	If set to `true`, this parameter causes source files loaded with `load` or `import`, as well as source entered in the console, to be annotated by the Scheme reader. Annotations include source file location information, which simplifies debugging. See Section 4.1.

Java property a	Scheme parameter	default	description
emitDebuggingSymbols	`emit-debugging-symbols`	true	If set to `true`, additional annotations useful for debugging, such as function and variable names, are produced by *SISC*'s compiler. See Section 4.1.
maxStackTraceDepth	`max-stack-trace-depth`	0	Specifies the maximum depth of virtual call stacks, which are used to obtain proper stack traces even in the presence of tail recursion and continuation capture and invocation, while still preserving tail-call semantics and safe-for-space guarantees. See Chapter 4. Collecting this information is computationally expensive, so this feature is turned off by default. A value of `16` is about right for debugging most applications. Large values allow collection of more information but carry a large performance penalty, small values result in some call information being lost.

Java property [a]	**Scheme parameter**	**default**	**description**
permissiveParsing	`permissive-parsing`	false	If set to `true`, the Scheme parser will warn rather than raise an error for various syntactic errors such as unbalanced parentheses. This allows one to continue parsing a syntactically invalid file, finding many errors at once.
printShared	`print-shared`	true	If set to `true`, `write` and the REPL detect shared structures in data and invoke a version of `write` capable of emitting the shared structure's external representation of data. See Section 3.3.2.2. The user may wish to set this parameter to `false` because of the overhead of both scanning all data, and constructing this representation when shared structures are detected.
replPrompt	`repl-prompt`		String to be displayed as part of *SISC*'s REPL prompt.
stackTraceOnError	`stack-trace-on-error`	true	If set to `true`, whenever an uncaught error is encountered a full stack trace is displayed automatically. See Section 4.1.

Java property [a]	**Scheme parameter**	**default**	**description**
strictR5RSCompliance	`strict-r5rs-compliance`	false	If set to `true`, *strict* R5RS syntax and semantics are followed. This will cause SISC to raise errors in all situations described as "an error" in the Scheme standard. This will override and invalidate all the interpretation liberties described in Appendix B.
synopsisLength	`synopsis-length`	30	Limit on the length (in characters) of the external representation of data structures in error messages and warnings. When the limit is reached, an ellipsis (`...`) is appended to the curtailed external representation.
vectorLengthPrefixing	`vector-length-prefixing`	true	If set to `true`, this parameter will instruct the pretty-printer to emit length prefixed, trailing-duplicate-eliminated vectors in its output. If `false`, ordinary full-length vectors without prefixes will be emitted. See Section 3.1.6.

When dynamic configuration parameters are retrieved or set with their associated Scheme function, the value is of the type specified for the parameter. See Section 2.4.2.3.

2.4.2.3. Value Conversion

When specifying configuration parameters via Java properties, a Java-like notation is used, e.g. boolean parameters are specified as `true` and `false`. By contrast, when getting and setting configuration parameters from Scheme, their values are of the appropriate Scheme type, e.g. boolean parameters are specified as `#t` and `#f`. Strings and symbols undergo a similiar conversion; they are specified without their double/single quotes in the Java properties.

For parameter types other than boolean, string and symbol, Java properties are read as Scheme values, i.e. Scheme literal notation should be used in the properties.

2.5. Running Scheme Programs

2.5.1. Loading

SISC supports loading Scheme programs using the R^{5}RS optional procedure `load`. A number of file types for loading Scheme code are supported. The common extensions are:

- `scm` - Scheme source code
- `sce` - Scheme code expanded. This type of code has been processed by the syntax expander and contains only core Scheme forms.
- `scc` - *SISC* Compiled Code. This contains expanded and compiled code in a *SISC* specific representation. The code gets executed when loaded.

In addition to code loaded, *SISC* requires a *heap*, which contains the default set of libraries and functions for the initial environment. *SISC* will look for a heap file called `sisc.shp` in the current directory, the directory referenced by the `SISC_HOME` environment variable, and as a resource paired with the `HeapAnchor` class in the `sisc.boot` package of the classpath. The standard *SISC* distribution contains `sisc.shp` in the same directory as the supporting `.jar` files.

It isn't uncommon to want the heap to reside on the classpath, where it can be more easily resolved in applets or in web applications. This file, usually called `sisc-heap.jar` can be added to the classpath in any usual fashion, and the heap loading routines will discover it if not found elsewhere. To create a jar file containing the heap, create the following file structure in the jar:

```
sisc/
sisc/boot/
sisc/boot/HeapAnchor.class
sisc/boot/sisc.shp
```

The `HeapAnchor.class` class file is distributed in the `sisc-opt.jar` file of the full binary distribution.

2.5.2. Scheme Shell Scripts

On Unix or Unix-like systems, *SISC* supports SRFI-22, a mechanism for writing shell-like scripts that can be invoked directly as executable programs. The text of the SRFI, which can be found at *http://srfi.schemers.org* (http://srfi.schemers.org/srfi-22/srfi-22.html), describes how such programs are written.

Chapter 3. Scheme Language

In this chapter we will examine the language that *SISC* interprets, which is a superset of the R^{5}RS Scheme Standard.

3.1. Types

3.1.1. Numbers

The full Scheme number tower is supported:

- Integers
- Floating Point numbers
- Rational numbers
- Complex numbers

Depending on the numeric library compiled into *SISC*, floating point numbers have either 32 or 64 bit IEEE precision, or arbitrary [1] precision. Regardless, *SISC*'s complex numbers have floating point components of the same precision as the reals. Integers have arbitrary precision in all numeric libraries, and rational numbers are built with arbitrary precision components.

3.1.1.1. Numeric constants

The precision specifying exponents (`S, F, L,` and `D`) are ignored in *SISC*, all inexact numbers are kept in the precision of the numeric library. The exponents are read and used to scale the real number as expected. In the case of arbitrary precision floats, specific precision constraints are maintained to prevent a runaway increase of precision. The constraints can be set by minFloatPrecision and maxFloatPrecision configuration parameters on startup. See Section 2.4.2.

All four base specifiers (`#x, #o, #d, #b`) are supported for integers and rationals. Only decimal (`#d`), the default, is supported for floating point and complex numbers.

SISC will produce infinite or not-a-number quantities from some operations. Those quantities are represented and can be used in Scheme programs as `#!+inf` (positive infinity), `#!-inf` (negative infinity), and `#!nan` (not-a-number).

3.1.1.2. Exactness

Exactness and inexactness contagion behaves as expected. Rational's are made inexact through division. Floats are made exact by conversion to a rational number. *SISC* attempts as accurate a conversion as possible, by converting the decimal portion of the number to a ratio with a denominator of the form 10^n, where n is the scale of the floating point number. Then the fraction is reduced as usual.

Since complex numbers must have floating point components currently, conversion to an exact merely rounds the components to integers.

3.1.2. Characters

SISC's characters are double-byte wide. This means that they are capable of representing the full range of unicode characters. Unicode characters can be created with `number->character`; `#\nnnnnn`, where `nnnnnn` is an octal number in the range 000000 -> 177777; or `#\uxxxx`, where `xxxx` is a hexadecimal number in the range 0000 -> ffff. At least two zeros must be specified to distinguish from the '0' character when using an octal character literal. At least one zero must be specified to distinguish a hexadecimal character from the 'u' character.

SISC also provides additional named characters, to add to the Scheme standard's `space` and `newline`:

Table 3-1. Named character literals

Character Name	**Unicode Value (hex)**
`backspace`	`0008`
`newline`	`000a`
`nul`	`0000`
`page`	`000c`
`return`	`000d`
`rubout`	`007f`
`space`	`0020`
`tab`	`0009`

Formally, *SISC*'s lexer modifies the R^5RS grammar with the following productions for character literals:

```
<character> --> #\ <any character>
     | #\u <uinteger 16>
     | #\ <uinteger 8>
     | #\ <character name>
<character name> --> backspace | newline | nul
     | page | return | rubout | space | tab
```

Characters are *not* compared with respect to the locale of the running system. Character comparison is equivalent to numeric comparison of the character value as returned by `char->integer`.

There are a number of reasons why a full Unicode system is non-trivial, especially within the framework of the R^5RS string and character functions. Such a discussion is outside the scope of this document. Unicode compliant processing may be made available in the future as a library, however.

3.1.3. Symbols

SISC's symbols are ordinarily case-insensitive. *SISC* maintains true pointer equality between symbols with like contents, unless the symbol is created *uninterned*. An uninterned symbol is one which is guaranteed to be pointer distinct from any other symbol in the Scheme system, even another with the same contents. Uninterned symbols can be generated with:

procedure: (**`string->uninterned-symbol`** `string`) => symbol

Converts the provided string into an uninterned, pointer distinct symbol.

Uninterned symbols, while always pointer-distinct, may still be `equal?` to another symbol if it's representation matches another.

3.1.3.1. Case Sensitivity

SISC also allows symbols to be created that *are* case-sensitive. This can be done one of two ways. The first is by setting the caseSensitive configuration parameter (see Section 2.4.2. The second method is via a non-standard symbol syntax. If a symbol is enclosed in pipe ('|') characters, the reader will treat that individual symbol as cased. The syntax extends the R^{5}RS grammar with the following production:

```
<cased symbol> --> |<identifier>|
```

Example 3-1. Case sensitive Symbol literals

```
(eq? 'a '|A|) ; => #f
(eq? 'a '|a|) ; => #t
(eq? '|A| '|a|) ; => #f
```

3.1.3.2. Printed Representation

Symbols may contain characters that are disallowed by R^{5}RS using `symbol->string`. In such a case, the printed representation of that symbol will contain those characters, prefaced with the escape ('\') character. Likewise, such symbols may be created without `symbol->string` by escaping non-standard characters.

Symbols which contain characters that could only be present in a case-sensitive environment will be printed in one of two ways, depending on the value of the `case-sensitive` parameter. If true, the symbols will be printed as is, containing the upper and lower case letters. If false, the symbol will be printed surrounded by pipe characters.

3.1.4. Strings

Strings are built from Unicode characters, and are compared lexicographically in a manner derived from character comparison. In addition to using backslash to escape the double-quote (") character and the backspace character itself, *SISC* provides several escape codes to ease string literal construction.

Table 3-2. String escape codes

Escape	Value
`\f`	Inserts the formfeed character (unicode 000c)
`\n`	Inserts the newline character (unicode 000a)
`\r`	Inserts the rubout character (unicode 007f)
`\t`	Inserts the tab character (unicode 0009)
`\uxxxx`	Inserts the unicode character described by the hex number 'xxxx'. All four hex digits must be specified.

Escape	Value
`\\`	Inserts the backslash ('\') character
`\"`	Inserts the double quote ('"') character

3.1.5. Pairs and Lists

A function is provided to determine if a given pair is a *proper list*.

procedure: (**`proper-list?`** `datum`) => #t/#f

Returns `#t` if the given argument is a *proper-list*. That is, if the argument is a pair, whose `cdr` is either the empty-list or also a proper-list, and which contains no references to itself (is not circular).

3.1.6. Vectors

SISC supports the length prefix method of creating Vector constants. For example, `'#5(x)` creates a vector constant containing five identical symbols. In addition, the length-prefix form is used when printing vectors, and if elements repeat at the end of a Vector, only the last unique element is printed. This form is referred to as the *compact* vector representation. The unprefixed form with all elements displayed is called the *verbose* representation.

Vectors are displayed differently depending on the call used. When called with `display`, in addition to the ordinary R^{5}RS rules regarding the output of values displayed with `display`, the verbose representation is displayed. Using `write`, on the other hand produces the compact representation.

Displaying a vector with `pretty-print` may output either the verbose or compact representation of a vector. The behavior in this regard is controlled by the vectorLengthPrefixing configuration parameter (see Section 2.4.2). If set to #t, `pretty-print` will emit the compact representation. If #f, the verbose representation is produced.

3.1.7. Boxes

SISC supports boxes, a container for a Scheme value. Boxing is often used to implement call-by-reference semantics. Boxes are created and accessed using the following three functions:

procedure: (**`box`** `value`) => box

Creates a box filled with the given value.

procedure: (**`unbox`** `box`) => value

Returns the value contained in the given box.

procedure: (**`set-box!`** `box value`) => undefined

Replaces the value contained in the given box with the value provided.

In addition to the `box` function for creating boxes, *SISC* provides an external representation for boxes and boxed values. It extends the R^{5}RS grammar with the following:

```
<boxed value> --> #&<datum>
```

This syntax denotes a boxed value, with `<datum>` as the contained value.

Boxes are a distinct first class type. The `box?` predicate tests a value to see if is a box.

procedure: (**box?** `value`) => #t/#f

Returns `#t` only if the given value is a box.

Boxes, like pairs, are only equal in the sense of `eq?` and `eqv?` when a box is compared with itself. A box is equal to another in the sense of `equal?` if the value contained within the box is `equal?` to the value contained in the other.

3.1.8. Parameters

A parameter is a named dynamic variable that is accessed through a function. The function, when given no arguments, returns the current value of the parameter. When given an argument, the value of the parameter is set to the provided value.

SISC's parameters are fully compatible with those specified by SRFI-39. Consult the SRFI-39 specification at srfi.schemers.org (http://srfi.schemers.org) for documentation on how to construct and use parameters. SRFI-39 does not specify the semantics for parameters in the presence of threads. SISC's parameters bind into the dynamic environment, which means their semantics are defined based on the semantics of dynamic environments' interactions with threads, specified in Section 6.1.

3.1.9. Immutable types

SISC follows the R^{5}RS recommendation of immutable list, string, and vector constants. Quoted lists and vectors are immutable. Attempting to modify elements in these constants will raise an error. String constants are immutable as well when created with `symbol->string`.

3.2. Equivalence

SISC's storage model maintains true pointer equality between symbols, booleans, the end-of-file object, void, and the empty list. Thus two instances of any of those types is guaranteed to return `#t` from `eq?` if they would have produced `#t` from `equal?`.

Numbers and characters are not pointer equal ordinarily (unless actually occupying the same storage). *SISC* will return `#t` from `eqv?` if two numbers are both exact, or both inexact, and are numerically equal. Two characters are equivalent from `eqv?` if they occupy the same code-point in the unicode character set. This is the behavior specified by R^{5}RS.

Strings, vectors, lists, and boxes are containers for other Scheme types. As such they are not pointer equal unless they are referenced by two variables that point to the same storage location (i.e. they are actually pointer equal). *SISC* holds that only `equal?` will return `#t` if two objects are the same type and their contents contain equivalent values with respect to `equal?`.

3.3. Syntax and Lexical Structure

3.3.1. Comments

In addition to the single line comments of the Scheme standard, *SISC* supports both *s-expression commenting* and *nested, multiline comments*. An s-expression comment is used to comment out an entire s-expression. To do this, the sharp sequence `#;` is used. It extends the R^{5}RS grammar with the following production:

```
<expression-comment> --> #;<datum>
```

The reader, upon encountering this sharp sequence, will read and discard the next datum. The expression commented out must still be a valid s-expression, however.

Nested, multiline comments are as defined in SRFI-30. Briefly, a multiline comment begins with the sharp sequence `#|` and ends with the sequence `|#`. The comment may contain nested comments as well. Unfortunately, this extension cannot be represented in a stateless grammar for the lexical structure.

3.3.2. Shared Structures

3.3.2.1. Reader Syntax

SISC provides a parse-time syntax for creating data (primarily vectors and lists) that contain references to themselves or data which contains several pointer-equal elements. This can be useful to create streams, graphs, and other self-referencing structures while maintaining readability and avoiding complex construction code.

The reader syntax has two parts, defining a pointer, and later referencing the pointer to create the circular reference.

Below is an additional production in the R^{5}RS formal syntax (specifically section 7.1.2, external representations) to support circular structures:

```
<pointer definition> --> #<uinteger 10>=<datum>
<pointer reference> --> #<uinteger 10>#
```

The first form instructs the reader to create a pointer identified by the specified integer, which maps to the datum that follows, and is active during the reading of the datum on the right-hand side of the definition.

If a second definition occurs during the reading of the datum with the same integral identifier, the previous definition is overwritten for the duration of the read. The definitions are *not* scoped in any way. The pointer identifiers should be kept unique by the programmer to prevent any unintended effects of identifier collisions.

The second form references a previously created pointer definition. It is an error to reference an undefined pointer. The reader will handle a valid reference by placing a pointer at the current read position back to the location of the definition.

At this point some examples might be helpful:

Example 3-2. Circular Structures

```
(define x '#0=(1 2 . #0#))
(caddr x)        ; => 1
(list-ref x 15)  ; => 2

(define y '(1 2 #1=#(3 4) . #1#))
(eq? (caddr y) (cdddr y)) ; => #t
```

3.3.2.2. Writing

Ordinarily, the display of cyclical data would cause a problem for a Read-Eval-Print-Loop. For this reason, the REPL will attempt to check the structure it is about to print for circularities before printing. If a cycle is found in the structure, the REPL will refuse to print if the printShared configuration parameter, described below, is `false`. In that case the REPL will issue a warning to the user that the structure contains a cycle. If a circular structure is printed with `display`, `write`, etc, and the printShared parameter is set to `false`, the environment may enter an infinite loop which may or may not cause the Scheme system to exit with an error.

The printShared configuration parameter (see Section 2.4.2), if set to `true` enables *SISC* to scan data for circularity and data sharing before writing values. If such sharing is found, an alternate printer is invoked which will emit a representation compatible with the circular structure representation described in the previous section.

Alternately, *SISC* also supports SRFI-38, which describes the functions `write-showing-shared` and `read-with-shared-structure`.

3.3.3. Control Features

In addition to the R^{5}RS standard control features, two additional forms, `when` and `unless`, are supported by *SISC*.

syntax: (**`when`** `condition expression [expressions] ...`) => value

Evaluates *`condition`*, an expression. If true, the expressions that follow are evaluated, in order, the value of the last being returned. If not true, the result is unspecified.

syntax: (**`unless`** `condition expression [expressions] ...`) => value

Evaluates *`condition`*, an expression. If false, the expressions that follow are evaluated, in order, the value of the last being returned. If true, the result is unspecified.

3.3.4. Syntactic Extension

SISC provides a hygienic macro system that fully conforms to the R^{5}RS standard. The macro system is provided by the portable syntax-case macro expander. In addition to R^{5}RS macros, the expander provides a more flexible macro definition tool called `syntax-case`. A full description of the capabilities of the expander is best found in the *Chez Scheme Users Guide* (http://www.scheme.com/csug.html), specifically *Section 9.2, Syntax-Case* (http://www.scheme.com/csug/syntax.html#g2154).

In addition, *SISC* supports non-hygienic, legacy macro support in two forms; `define-macro` and `defmacro`. These forms, found in older Scheme code written for R^{4}RS compliant Scheme systems, should be used only for executing legacy code which relies on it. New code should use the safer and more flexible `syntax-case` or the standard `syntax-rules` macros.

syntax: (**`define-macro`** `(name . args) body ...`)

syntax: (**`define-macro`** `name transformer`)

In the first form, `define-macro` creates a macro transformer bound to *`name`*, which when applied will have raw s-expressions bound to one or more parameters (*`args`*). The `(name . args)` name and formal parameter form is identical to the short form for procedure definition with `define`.

The transformer's body will then, using the s-expressions bound to its arguments, return a new s-expression that is the result of the macro transformation.

The second form binds an arbitrary procedure to the syntactic keyword `name`, using that procedure to transform occurences of that named syntax during future evaluations.

syntax: (**`defmacro`** `name args body ...`)

`defmacro` is another macro definition form supported by some Scheme systems. Its semantics are equivalent to:

```
(define-macro (name . args) body ...)
```

3.4. Errors and Error Handling

Errors can be raised by primitives in libraries and Scheme-level code. *SISC* provides a sophisticated mechanism for handling these errors when they occur during program execution.

3.4.1. Failure Continuations

During the execution of any program, there is always a continuation that represents the *rest* of a computation. In addition, one can imagine all the activities that will occur as a result of an error. This sequence of actions is explicitly represented in *SISC* as a *failure continuation*.

Two values must be applied to a failure continuation. The first is an error record, a datastructure which describes the error (and may contain information about the name of the function that generated the error, a descriptive message about the error, etc.). The second is the continuation of the expression that raised the error. All errors raised in *SISC* automatically and implicitly obtain and apply these values to the active failure continuation. Applying the error record and error continuation to the failure continuation will not return to the continuation of the application, unless that continuation was captured and later invoked in a non-local entrance.

3.4.1.1. Creation

A programmer may wish to augment current failure continuation, choosing a different set of actions to occur for a body of code if it raises an error. To facilitate this, *SISC* provides the `with-failure-continuation` procedure.

procedure: (**`with-failure-continuation`** `handler thunk`) => value

procedure: (**`with/fc`** `handler thunk`) => value

`with-failure-continuation` takes as arguments a thunk (a zero-argument procedure) to be evaluated. The thunk will be evaluated in the continuation of the `with/fc` function, and with a failure continuation defined by the provided error handler. If during the evaluation of the thunk an error is raised, the first, two argument procedure is called with values describing the error and its context. If no error occurs, value of the thunk is applied to the continuation of the `with/fc` expression.

The error handler required as an argument to `with-failure-continuation` must accept two values. The first is a value containing information about the error that occurred. This is often an association list containing a number of attributes of the error. The second is a procedure encapsulating the continuation that was in place at the site of the error. This continuation is referred to as the *error continuation*

When an error occurs, the error handler may choose one of three courses in dealing with the error. First, the handler may choose to return an alternate value to be applied to the continuation of the `with/fc` expression. Second, the handler may restart the computation from the error site by invoking the error continuation with a value that should be returned in place of the expression that caused the error. Finally, the handler may choose to propagate the error (or a new error) to the failure continuation of the `with/fc` expression. This can be done with the `throw` function described in Section 3.4.2.3.

3.4.1.2. Capture

The currently active failure continuation may be obtained explicitly using the `call-with-failure-continuation` procedure. This continuation may be applied to appropriate values at any time in the future.

procedure: (**`call-with-failure-continuation`** `procedure`) => value

procedure: (**`call/fc`** `procedure`) => value

Calls the given one-argument procedure with the currently active failure continuation.

3.4.1.3. Interaction with Ordinary Continuations

Failure continuations exist as an attribute of the ordinary continuations of Scheme expressions. Because of this, the invocation of a continuation may cause a different failure continuation to become active in the region of the captured continuation. Specifically, the failure continuation in place at the `call/cc` expression will be reinstated when that continuation is later invoked.

Similarly, invoking a continuation that escapes a region of code will cause any created failure continuations to be abandoned, unless the region is itself captured in a continuation and later invoked.

See also Section 3.4.4.

3.4.2. Error Records

An error record is the value usually propagated with an error in *SISC*. It is a datastructure containing such information as the location of the error, a descriptive message about the error, and possibly other error metadata.

3.4.2.1. Creating Error Records

Error records can be created in advance of actually raising an error with the `make-error` function. The function allows the programmer to create error records that contain a location and a message or value. No field of an error record is required.

procedure: (**`make-error`** `[location] [message] [arguments] ...`) => error-record

Constructs an error record. If present, a symbol, and not `#f`, the first argument is the location of the error, which may be a symbol equivalent to a function identifier. If present, the message is a format-string processed with the optional arguments that follow as by `format` in SRFI-28. The remaining arguments must only be present if the format-string is present as the message.

procedure: (**`make-error`** `[location] error-value`) => error-record

Constructs an error record. If present, a symbol, and not `#f`, the first argument is the location of the error. The second argument is an arbitrary Scheme value that will be the error value. This value will be accessible with the `error-message` function.

None of the fields of an error-record are required. One may create an error record with no information, an error record with only a location, or an error record with only a message or value. Below are some examples (for an explanation of the `throw` procedure see Section 3.4.2.3).

```
(throw (make-error))
; => Error.

(throw (make-error 'foo))
; => Error in foo.

(throw (make-error "something ~a happened" 'bad))
; => Error: something bad happened

(throw (make-error 3))
; => Error: 3

(throw (make-error #f 'foo))
; => Error: foo

(throw (make-error 'foo "something ~a happened" 'bad))
; => Error in foo: something bad happened
```

In addition, an error record may be created that adds additional information to an error record that was already created. This is useful when an error was caught in an error handler, and one wishes to raise an error from the handler that contains additional information about the local location or error message as well as the error that was caught.

procedure: (**`make-nested-error`** `local-error parent-error parent-error-continuation`) => error-record

procedure: (**`make-nested-error`** `local-error exception`) => error-record

The first version creates an error record which has *`parent-error`* (and it's associated *`parent-error-continuation`*) as the root cause of an error-record passed as *`local-error`*.

The second version creates an error record which has *`exception`* (see Section 3.4.2.4) as the root cause of an error passed as *`local-error`*.

An example of the creating, throwing, and display of a nested error follows.

```
(with-failure-continuation
  (lambda (m e)
    (throw (make-nested-error
            (make-error 'foo "could not call bar.") m e)))
  (lambda ()
    (error 'bar "something went wrong.")))
;=> Error in foo: could not call bar.
;   Caused by Error in bar: something went wrong.
```

3.4.2.2. Accessors

An error record contains several useful pieces of information. The following functions allow the programmer to access that information.

procedure: (**`error-location`** `error-record`) => symbol

Obtains the location of the error, a symbol which may be a function identifier. If there is no location specified, `#f` is returned.

procedure: (**`error-message`** `error-record`) => value

Obtains the message of the error, which may be a string which is a descriptive message of the error, or an arbitrary value (as created by the second form of `make-error`). If there is no message specified, `#f` is returned.

procedure: (**`error-parent-error`** `error-record`) => error-record

Obtains the parent error of the error. This is the value of the second argument to the `make-nested-error` function. If there is no parent specified, `#f` is returned.

procedure: (**`error-parent-continuation`** `error-record`) => error-continuation

Obtains the parent error continuation of the error. This is the value of the third argument to the `make-nested-error` function. If there is no parent specified, `#f` is returned.

3.4.2.3. Raising Errors

The fundamental mechanism for raising an error in application code is provided by the `throw` procedure.

procedure: (**`throw`** `error-record [error-continuation]`) => does not return

procedure: (**`throw`** `exception`) => does not return

The first verison applies the given error record to the current failure continuation. If provided, the error continuation is designated by the optional parameter. If not, the continuation of the throw expression is used.

The second form applies the current failure continuation to the error record and error continuation extracted from the supplied *`exception`* (see Section 3.4.2.4).

If invoked from an error-handler with the values of the handler's formal parameters, throw has the effect of propagating the error in a manner that is equivalent to the absence of the modified failure-continuation.

`throw` could be defined in terms of `call-with-failure-continuation` as:

```
(define (throw error . args)
  (call-with-failure-continuation
     (lambda (fk)
       (if (null? args)
           (call-with-current-continuation (lambda (k) (fk error k)))
           (fk error (car args))))))
```

For convenience and compatibility with SRFI-23, the function `error` is provided. Its syntax is identical to `make-error`, but it immediately applies the resulting error record to the current failure continuation with the current continuation as the error continuation.

procedure: (**`error`** `[location] [message] [arguments] ...`) => does not return

Raises an error record whose location, if provided, is *`location`*, a symbol; and whose error message, if present, is *`message`*. If provided, the error message is a format-string that is processed, with the optional *`argument`*s, as with the `format` function in SRFI 28.

procedure: (**`error`** `[location] error-value`) => does not return

Raises an error record whose location, if present, is the symbol *`location`*, and and whose error-value is any arbitrary Scheme value.

`error` can be implemented in terms of `throw` and `make-error`:

```
(define (error . args)
  (throw (apply make-error args)))
```

3.4.2.4. Exceptions

Exceptions in *SISC* are a simple wrapper around an error record and an associated error continuation.

Exceptions are created with

procedure: (**`make-exception`** `error-record error-continuation`) => exception

Constructs an exception from an *`error-record`* and an *`error-continuation`*, e.g. as obtained from the arguments of a handler procedure passed to `with-fc`.

Accessors and a type-test are provided by the following procedures:

procedure: (**`exception-error`** `exception`) => error-record

Returns the *`exception`*'s error record.

procedure: (**`exception-continuation`** `exception`) => error-continuation

Returns the *`exception`*'s error continuation.

procedure: (**`exception?`** `value`) => #t/#f

Returns #t if *`value`* is an exception object, #f otherwise.

3.4.3. Examples

At this point, a few examples may be helpful:

```
(+ 1 (/ 1 0) 3)
; => A divide by zero error is raised
```

Example 3-3. Return a new value

```
(with-failure-continuation
    (lambda (error-record error-k)
      'error)
  (lambda () (+ 1 (/ 1 0) 3)))
; => The symbol 'error
```

Example 3-4. Restart with a different value

```
(with-failure-continuation
    (lambda (error-record error-k)
      (error-k 2))
  (lambda () (+ 1 (/ 1 0) 3)))
; => 6
```

Example 3-5. Propagate the error

```
(with-failure-continuation
    (lambda (error-record error-k)
      (throw error-record error-k))
  (lambda () (+ 1 (/ 1 0) 3)))
; => A divide by zero error is raised
```

Example 3-6. Propagate a different error with the same error continuation

```
(with-failure-continuation
    (lambda (error-record error-k)
      (throw (make-error '/ "could not perform the division.") error-k))
  (lambda () (+ 1 (/ 1 0) 3)))
; => An error is raised: Error in /: could not perform the division.
```

Example 3-7. Raise a new error

```
(with-failure-continuation
    (lambda (error-record error-k)
      (error 'example-function "could not evaluate the expression."))
  (lambda () (+ 1 (/ 1 0) 3)))
; => An error is raised: Error in example-function: could not evaluate the expression.
```

Note that the difference between Example 3-6 and Example 3-7 is that in the latter, the computation can still be restarted from the second argument of the addition if an outside handler catches the newly raised exception and applies the continuation. This is not true in the last example, as its a new error whose continuation is the same as the `with-failure-continuation` expression.

3.4.4. `dynamic-wind`

R^{5}RS does not specify the behavior of `dynamic-wind` in the case where an error is raised while evaluating the *during* thunk. *SISC* chooses to view an error raised in that section as an instance of the dynamic extent being exited. In other words, if an error is raised in the dynamic extent of a dynamic-wind expression, *SISC* will ensure that the *after* thunk is evaluated before the error is propagated to the failure-continuation of the dynamic-wind expression.

Example 3-8. Errors and `dynamic-wind`

```
(define x 0)
(dynamic-wind (lambda () (set! x (+ x 1)))
              (lambda () (/ 1 0))
              (lambda () (set! x (+ x 1))))

; => A divide by zero error is raised, and the value of x is 2
```

If an error is raised in either the *before* or *after* thunks, no additional measures are taken. The error is propagated to the failure-continuation of the dynamic-wind as if the dynamic-wind call was an ordinary function application. Explicitly, if an error is raised from *before*, neither *during* nor *after* will be executed. If an error is raised in *after*, the results of evaluating *before* and *during* remain valid.

Also noteworthy is what happens if a continuation is invoked that exits from either the *before* or *after* thunks. Such a case is treated just as if a continuation was invoked during the evaluation of an operand to an application. This is to say that no additional steps will be taken by *SISC*. If *before* is escaped by a continuation invocation, neither *during* nor *after* will be executed. If *after* is escaped, the results of *before* and *during* remain valid.

In summary, extraordinary evaluation is only possible during the evaluation of the *during* thunk. The *before* and *after* thunks are evaluated with the dynamic environment and dynamic-wind stack of the call to dynamic-wind itself.

3.5. Symbolic Environments and Property Maps

Symbolic environments and property maps provide additional named global environments useful for storing program specific data without exposing it to the general purpose top-level environment.

A property map is dictionary structure tied to the interaction environment which maps symbolic names to Scheme values. First-class symbolic environments provide a similar mapping, but can be used as first class values (including as an argument to `eval`). Symbolic environments are used to implement *SISC*'s global (top level)and report environments.

3.5.1. Access Functions

Access to symbolic environments is performed through the `getprop` and `putprop` functions. All symbolic environment operations are thread safe.

procedure: (**`getprop`** `binding-name plist-name [default-value]`) => value

procedure: (**`getprop`** `binding-name environment [default-value]`) => value

Attempts a lookup of *`binding-name`* in an environment.

In the first form, the the binding is resolved in the interaction-environment's property list named *`plist-name`*, a symbol. If the environment is not found or the binding doesn't exist, *`default-value`* is returned if provided, otherwise `#f` is returned.
In the second form, the binding is resolved in a first-class symbolic environment.

procedure: (**`putprop`** `binding-name plist-name value`) => undefined
procedure: (**`putprop`** `binding-name environment value`) => undefined

Sets the value of a binding named with the symbol *`binding-name`* in a property list or first class symbolic environment.

In the first form, the binding is resolved using a symbolic name (*`plist-name`*) in the interaction environment's property lists. If the map does not yet exist, it is created as an empty map.
In the second form, the binding is resolved in the provided first class symbolic environment. If the binding does not yet exist in the given environment, it is created. If a binding previously existed, its previous value is discarded.

3.5.2. Obtaining and Naming

Symbolic environments are a first class datatype in *SISC*. The top-level environment itself is merely a special cased symbolic environment. To obtain the top-level environment as a first class value, one can use the `interaction-environment` function that is an optional procedure in R^{5}RS. Another useful environment is the R^{5}RS report environment available by calling:

```
(scheme-report-environment 5)
```

Each call to `scheme-report-environment` returns a new environment that contains only the bindings available in the Scheme report. Finally, the initial environment available to the programmer when *SISC* starts can be retrieved using the `sisc-initial-environment` function:

procedure: (**`sisc-initial-environment`**) => environment

Returns the initial *SISC* interaction environment.

Like `scheme-report-environment`, each call to `sisc-initial-environment` returns a distinct environment which contains only the bindings initially available when *SISC* starts. An interesting use of this would be to define one or more distinct initial-environments, bound to toplevel variables. One could then define Scheme code and data in each environment that can use the full *SISC* language but cannot see any bindings in other environments.

Finally, R^{5}RS states that it is an error to modify the contents of a top-level variable that has not yet been created. *SISC* adheres to the standard, and raises an error when any unbound variable in a symbolic environment (including the top-level) is modified using `set!`. This differs from some Scheme systems that will silently create the binding and set it to the new value.

3.5.3. Chained Symbolic Environments

SISC contains a mechanism for creating a symbolic environment which is *chained* to another environment, such that new and modified bindings are created in the new, child environment, but bindings may also be resolved from the parent if not present in the child. *SISC* uses this functionality to protect the contents of the R^{5}RS and *SISC* initial environments from modification. One can use it in a similar way, protecting the bindings in the parent for sandboxing or other purposes.

procedure: (**`make-child-environment`** `parent-environment`) => environment

Creates a new environment, initially empty of its own bindings, but which chains to the provided *`parent-environment`* when resolving a binding.

procedure: (**`parent-environment`** `environment`) => environment

Obtains the parent environment of a symbolic environment. If the given environment has no parent (e.g. is not chained), `#f` is returned.

3.6. Miscellaneous Functions

The remaining functions in this chapter are not easily classified, but nevertheless are useful and worth describing.

procedure: (**`circular?`** `datum`) => #t/#f

Returns `#t` if the given datum is circular. A datum is circular if it is a compound datum (lists or vectors for example), and one of its elements is a reference to itself, or a reference to a sub-element which creates a cycle.

procedure: (**`compose`** `[function] ...`) => procedure

`compose` takes zero or more functions of one argument and returns a new function of one argument that will apply to that argument to the selected functions in reverse order. If no functions are provided, the identity function is returned.

For example, the function `caddr` could be simply defined as:

```
(define caddr (compose car cdr cdr))
```

procedure: (**`iota`** `n`) => pair

The `iota` function produces a list whose elements are the integers 0 to *`n`*-1 inclusive.

syntax: (**`time`** `[iterations] expression`) => list

Evaluates the given expression *`iterations`* times, or if *`iterations`* is not provided, only once. When complete, a list is returned of the following form:

```
(result (n ms))
```

where *result* is the Scheme value that resulted from the last evaluation of the expression, and *n* is the number of milliseconds taken to evaluate the expression. If more than one iteration occurred, then the average number of milliseconds elapsed during each iteration is returned.

Notes

1. Essentially arbitrary, see Section D.1 for a discussion of the physical limits of number representation

Chapter 4. Debugging Facilities

No Scheme system would be complete without facilities to assist the programmer in debugging his or her code. *SISC* provides aid for passive debugging (requiring no action on the part of the programmer) and active debugging (requiring code instrumentation to facilitate debugging).

4.1. Passive Debugging

Passive debugging facilities are provided that collect information on an error that occurred at runtime and was not caught by the executing code. The programmer can then inspect the last error, obtain information about the call stack of the error, or even attempt to restart the computation.

procedure: (**`get-last-exception`**) => exception

Retrieves the last exception that occurred in *SISC*.

One of the most common desires is to obtain a trace of the call stack, to determine what sequence of calls resulted in the error. *SISC* provides procedures for accessing the call stack of *any* continuation.

Requires: **(import *debugging*)**

procedure: (**`stack-trace`** `continuation`) => list

Returns the stack trace for *`continuation`* in form of a list. The format of the list is

stack-trace	:=	`(`call-frame ...`)`
call-frame	:=	sisc-expr | `(`sisc-expr overflown `.` stack`)`
overflown	:=	#t | #f
stack	:=	`(`stack-entry ...`)`
stack-entry	:=	sisc-expr | `(`repetitions `.` stack`)`
repetitions	:=	integer

Each element in the list represents one level in the *SISC* interpreter's call stack, starting from the top. The element contains the *SISC* virtual machine expression that would be executed next in that frame, and, if available, a compact representation of a virtual stack created by collecting information on tail calls carried out in that frame.

The virtual stack is bounded in size by the value of the see maxStackTraceDepth configuration parameter (see Section 2.4.2). If old information was dropped due to the bound being exceeded then the *`overflown`* flag is set.

Each entry in the virtual stack contains either the *SISC* virtual machine expression that was executed, or a sub-stack annotated with a repetition count, indicating that the entries in that sub-stack were repeated several times.

SISC expressions are annotated with source locations if the emitAnnotations parameter is set to `true`. Additional annotations are produced when emitDebuggingSymbols is set to `true`. See Section 2.4.2. The annotations can be retrieved using the `annotation` function with the keys `source-file`, `line-number`, `column-number`, `source-kind`, and `proc-name`.

Stack trace entries for expressions with a `source-kind` mentioned on `suppressed-stack-trace-source-kinds` are suppressed.

procedure: (**`suppressed-stack-trace-source-kinds`** `[list]`) => list

Retrieves or sets the list of source kinds to suppress in stack traces returned by `stack-trace`. This is a dynamic parameter.

The default value is `(#f)` which causes all stack trace entries for expressions with no specified source kind to be suppressed.
The annotation of expressions with source kinds and other information is controlled by the `source-annotations` parameter.

procedure: (**`source-annotations`** `[alist]`) => alist

Retrieves or sets the association list of additional annotations for expressions that are being read. This is a dynamic parameter.
All system and core library code is loaded with this parameter set to `()`, resulting in no additional annotations being produced. However, on entry to the REPL, the parameter is set to `((source-kind . user))`. In combination with the default settings for `suppressed-stack-trace-source-kinds` this results in system code being omitted from stack traces.

procedure: (**`print-stack-trace`** `continuation`) => void

Displays the call stack of the *`continuation`* in a human-readable form.

The error message, location information and call stack associated with an exception can be displayed in human-readable form using the following procedure.

procedure: (**`print-exception`** `exception [stack-trace?]`) => void

Displays the error message and location of *`exception`*. A stack trace is displayed if *`stack-trace?`* is absent or set to #t. Furthermore the procedure calls itself recursively in order to display similar information for nested exceptions.

In order to obtain the source file location of a call, your Scheme code must have been loaded while *SISC*'s reader had *annotations* enabled. Annotations are a means of attaching metadata to compiled Scheme code. To allow the reader to attach annotations related to the source file position of the code it reads, enable the emission of annotations with the emitAnnotations configuration parameter (see Section 2.4.2).

SISC can also produce more detailed stack traces if code was generated with *debugging symbols*. These are extra annotations generated by the compiler that track function and variable names that would ordinarily be discarded. By including these annotations, the stack trace can display the name of more of the calls involved. To enable the generation of debugging symbols, the emitDebuggingSymbols configuration parameter must be set to `true` (see Section 2.4.2).

Finally, when debugging a program for a long period of time, it may be desirable to have stack traces displayed whenever an error occurs, rather than needing to invoke `print-exception` or other functions each time. For this, the stackTraceOnError configuratin parameter must be set to `true` (see Section 2.4.2).

4.2. Active Debugging

Requires: **(import *debugging*)**

SISC provides active debugging aids; procedures and syntax that can be used in source code to assist in tracing the activities of running Scheme code.

4.2.1. Runtime Tracing

When a function is traced, each call to the function will be displayed to the console with the function's trace identifier and the values of the operands the function is being applied to. Each nested call is indented slightly, so as to illustrate the depth of calls. When the function application returns, the value of the function-call is displayed at the same indentation as the call itself. Once indented to a certain depth, the same indentation is kept for further nesting, but the depth of the call is displayed as an integer preceding the call.

syntax: (**`trace-lambda`** `trace-name formals body`) => procedure

When replaced with a trace-lambda, all calls to a lambda defined procedure are traced on the console. *`trace-name`* is an identifier which will disambiguate the procedure in the trace. *`formals`* and *`body`* have identical semantics to `lambda`.

syntax: (**`trace-let`** `loop-identifier formal-bindings body`) => value

Replaces a named-let expression in a similar manner to trace-lambda.

procedure: (**`trace`** `[symbol] ...`) => undefined

Begins traces on the procedures named by the symbols given. The procedures must be defined in the top-level environment.

If no procedures are given, a message is displayed indicating the names of top-level procedures currently being traced.

If a traced procedure is redefined, it will not retain the instrumenting installed by `trace`, until `trace` or `untrace` is called again (with any arguments). At that time, the traced procedures are reinspected and instrumenting reinstalled on redefined procedures.

procedure: (**`untrace`** `[symbol] ...`) => undefined

Stops tracing the top-level procedures named by the symbols given.

If no procedures are given, a message is displayed indicating the names of top-level procedures currently being traced.

`trace-lambda` and `trace-let` are useful for debugging anonymous lambdas and named-lets respectively. `trace` and `untrace` ar useful for tracing any top-level bound procedure, including calls to builtin procedures and stored continuations.

> **Note:** Tracing a function installs instrumentation code around the procedure which does not preserve the continuation of a call to that function. Thus, tail calls made in a traced function are no longer tail calls. This may affect the memory usage characteristics of running code.

4.2.2. Breakpoints

A user may wish to halt execution of a running Scheme program when a given procedure is called. *SISC* provides means to install breakpoints at top-level visible functions without having to redefine the function.

When a breakpoint is set using `set-breakpoint!`, and the function is called, execution will halt, returning to the REPL and displaying an informational message indicating a break, the procedure called, the arguments passed to the breakpointed procedure, and, if possible, the location in a source file of the call. The user may then continue execution using the `continue` procedure.

procedure: (**`set-breakpoint!`** `symbol`) => undefined

Instruments the top-level procedure named by the given symbol, such that when called, execution will halt and return to the REPL and the name of the breakpointed function and its arguments are displayed.

procedure: (**`clear-breakpoint!`** `symbol`) => undefined

Removes the instrumentation on the named top-level procedure, if present. Execution will continue normally through occurances of the formally breakpointed procedure.

procedure: (**`continue`**) => does not return

Continues execution from the most recent break. It is an error to call this procedure if a breakpoint has not been hit, or to call this procedure more than once for a given break.

procedure: (**`current-breakpoint-continuation`**) => continuation

Returns the continuation of the most recent breakpoint, or #f if execution is not currently interrupted at a breakpoint.

The continuation is useful for obtaining stack traces, e.g. with `(print-stack-trace (current-breakpoint-continuation))`.

procedure: (**`current-breakpoint-args`**) => list

Returns a list containing the current breakpoint's continuation procedure and arguments that will be used when execution is resumed with `continue`, or #f if execution is not currently interrupted at a breakpoint.

This procedure is useful for performing deep inspection of the breakpointed procedure and its arguments. The returned values are also handy for constructing modified breakpoint continuations with `set-current-breakpoint-args!`.

procedure: (**`set-current-breakpoint-args!`** `procedure value ...`) => #t/#f

Sets the current breakpoint's continuation procedure and arguments that will be used when execution is resumed with `continue`. If execution is not currently interrupted at a breakpoint then invoking this procedure has no effect and it returns #f. Otherwise it returns #t.

Chapter 5. I/O

SISC's I/O routines are implemented in a flexible manner, allowing extensions to create new I/O sources that will behave as standard Scheme port objects. The ports can then be operated on with all R^{5}RS port operations, as well as some *SISC* specific port functions.

5.1. Ports

5.1.1. URLs

In *SISC* all procedures that create ports for accessing files, e.g. `open-input-file`, `open-output-file` accept URLs in addition to ordinary file names. Here are some examples of valid URLs:

```
http://foo.com/bar/bar1.scm
file:/tmp/foo.scm
file:c:\bar\baz.scm
file:foo.scm
jar:http://foo.com/bar.jar!/bar/bar1.scm
```

The last is a URL referring to a file stored in a JAR on a remote web server. For further details on the format of URLs please consult this specification (http://www.ietf.org/rfc/rfc2396.txt). The format of JAR URLs is defined in the JDK API documentation (http://java.sun.com/j2se/1.4/docs/api/java/net/JarURLConnection.html). What types of URLs are supported by a particular installation of Java depends on the configured protocol handlers. See the JDK API documentation (http://java.sun.com/j2se/1.4/docs/api/java/net/URL.html) for details. [1]

Relative file names or URLs are resolved in relation to the following parameter:

parameter: (**`current-url`** `[url]`) => url

Retrieves or sets the URL which forms the basis for resolving relative filenames and URLs. It is initialized on start up with the path to the current directory. All parameters and the returned value are strings.

The algorithm for resolving relative URLs is defined in this specification (http://www.ietf.org/rfc/rfc2396.txt). For compatibility with other Schemes, *SISC* also supports the `current-directory` procedure, which is a simple wrapper around `current-url`.

A convenience procedure exists for executing a procedure while the `current-url` is temporarily set to a different value:

procedure: (**`with-current-url`** `url thunk`) => value

Sets the current URL to the URL obtained by normalizing *`url`* in relation to the current URL, then executes *`thunk`*, and then sets the current URL back to the previous value.

URLs can be normalized using

procedure: (**`normalize-url`** `url1 [url2]`) => url

When called with one string argument, `normalize-url` returns the normalized version of the given URL. Normalization involves, amongst other things, the replacement of relative path references such as `.` and `..`.

When called with two string arguments, the procedure returns the normalized version of the second URL when interpreted as a being relative to the first URL.

5.1.2. Buffered I/O

Buffered ports are provided in *SISC* to layer over any existing port, and read or write in larger, more efficient chunks. Buffered ports are created with the following constructors, which accept the underlying port and an optional size, indicating the number of bytes or characters to buffer before making an underlying read or write.

procedure: (**`open-buffered-binary-input-port`** `binary-input-port`) => binary-input-port

Creates a binary-input-port which is buffered, and reads its bytes from the provided port.

procedure: (**`open-buffered-binary-output-port`** `binary-output-port`) => binary-output-port

Creates a binary-output-port which is buffered, and writes its bytes to the provided port.

procedure: (**`open-buffered-character-input-port`** `character-input-port`) => character-input-port

Creates a character-input-port which is buffered, and reads its characters from the provided port.

procedure: (**`open-buffered-character-output-port`** `character-output-port`) => character-output-port

Creates a character-output-port which is buffered, and writes its characters to the provided port.

With buffered output ports, individual writes may not actually reach the eventual output source, so the programmer must explicitly flush the port when it the output data *must* reach its destination.

procedure: (**`flush-output-port`** `[output-port]`) => undefined

Causes the specified *`output-port`*'s buffered data to be written immediately. This operation is allowed on any output port, but may have no affect on some. *`output-port`* defaults to `current-output-port`.

5.1.3. Character Ports

The R^5RS I/O primitives implemented by *SISC* create *character ports*. Character ports read characters from input sources and treat the data as characters in a given character set. Correspondingly, character ports output bytes from characters according to a given character set's encoding rules.

By default character ports use the value of the string parameter `character-set` as the character encoding name. A list of many possible encoding names can be found in the Java Platform Documentation (http://java.sun.com/j2se/1.4/docs/guide/intl/encoding.doc.html)

One may temporarily change the default character set using the `with-character-set` function.

procedure: (**`with-character-set`** `encoding thunk`) => value

Changes the value of the `character-set` parameter, and the default character set to the named encoding while executing the body of *`thunk`*.

5.1.3.1. Port Creation

procedure: (**`open-input-file`** `url [encoding]`) => input-port

Creates an input port from the specified *`url`*. If the optional *`encoding`* parameter, a string, is supplied input will be decoded from the specified encoding rather than the default.

procedure: (**`open-output-file`** `url [encoding] [auto-flush]`) => output-port

Creates an output port to the specified *`url`*. If the optional *`encoding`* parameter, a string, is supplied output will be encoded in the specified encoding rather than the default. If the optional *`auto-flush`* argument is provided and is non-false, the port will automatically flush after each write call. If the specified file exists, it will be overwritten silently when the port is opened.

procedure: (**`open-character-input-port`** `binary-input-port [character-set]`) => input-port

Creates an input port which reads from the provided binary input port. One may specify the desired character set as a string, otherwise the character set is retrieved from the `character-set` dynamic parameter.

procedure: (**`open-character-output-port`** `binary-output-port [character-set] [auto-flush]`) => output-port

Creates an output port which writes to the provided binary output port. One may specify the desired character set as a string, otherwise the character set is retrieved from the `character-set` dynamic parameter. If the optional *`auto-flush`* argument is provided and is non-false, the port will automatically flush after each write call.

5.1.3.2. Bulk Character I/O

In addition to the R^{5}RS I/O primitives, *SISC* provides two functions for reading and writing blocks of characters from a character port.

procedure: (**`read-string`** `buffer offset count [character-input-port]`) => integer

Reads up to *`count`* characters from the implicit or specified character port into the string *`buffer`* starting from the position specified by the integer *`offset`*. The number of characters successfully read (which may be fewer than *`count`*) is returned, or the end-of-file value if the end-of-file was reached before any characters were encountered.

procedure: (**`write-string`** `buffer offset count [character-output-port]`) => undefined

Writes exactly *`count`* characters from the given string *`buffer`* starting from the integer position *`offset`* to the implicit or specified character output port.

5.1.3.3. Port Creation Wrappers

The next set of procedures assists in creating a port, followed by calling a given procedure with that port. When the procedure returns, the port is closed. Invoking escaping continuations from inside the procedure does not close the port, and invoking a continuation captured inside the procedure does not open the port.

procedure: (**`call-with-input-file`** `url [encoding] procedure`) => value

Calls *`procedure`* with a new input port attached to *`url`*. The result of the thunk is returned.

If the optional *`encoding`* parameter is provided, the character port created will use the specified encoding rather than the default.

procedure: (**`call-with-output-file`** `url [encoding] procedure`) => value

Calls *`procedure`* with a new character output port attached to *`url`*. The result of the thunk is returned.

If the optional *`encoding`* parameter is provided, the character port created will use the specified encoding rather than the default.

5.1.3.4. Replacing Standard Ports

The following procedures wrap a thunk, redirecting the input and output of the thunk while it is evaluating to an input or output port other than the `current-input-port` and `current-output-port`. Invoking escaping continuations from inside the procedure restores the original port, and invoking a continuation captured inside the procedure restores the redirection.

procedure: (**`with-input-from-port`** `input-port thunk`) => value

Evaluates *`thunk`* with *`input-port`* as the `current-input-port` for the duration of the evaluation.

procedure: (**`with-output-to-port`** `output-port thunk`) => value

Evaluates *`thunk`* with *`output-port`* as the `current-output-port` for the duration of the evaluation.

procedure: (**`with-input-from-file`** `url [encoding] thunk`) => value

Evaluates *`thunk`* with an input port attached to a file opened for reading from *`url`* as the `current-input-port` for the duration of the evaluation. The port is closed when *`thunk`* returns normally.

If the optional *`encoding`* parameter is provided, the character port created will use the specified encoding rather than the default.

procedure: (**`with-output-to-file`** `url [encoding] thunk`) => value

Evaluates *`thunk`* with an input port attached to a file opened for writing to *`url`* as the `current-output-port` for the duration of the evaluation. The port is closed when *`thunk`* returns normally.

If the optional *`encoding`* parameter is provided, the character port created will use the specified encoding rather than the default.

5.1.3.5. Port Predicates

procedure: (**`input-port?`** `value`) => #t/#f

Returns #t if *`value`* is an input port, #f otherwise.

procedure: (**`output-port?`** `value`) => #t/#f

Returns #t if *`value`* is an output port, #f otherwise.

5.1.4. String Ports

Requires: **(import *string-io*)**

String ports are input or output ports that read or write to a string rather than a file or other stream. String ports can be used to parse or emit formatted strings using the standard Scheme port operations. A String

Input port will read from a given string until the end of string is reached, at which point `#!eof` is returned.

String ports deal with characters as the atomic unit, and as such preserve full unicode width characters at all times.

procedure: (**`open-input-string`** `string`) => string-input-port

Creates a string input port whose characters are read from the provided string. Characters will be returned from any read operation on the port until the end of the string is reached. Read calls after reaching the end of the string will return #!eof.

procedure: (**`open-output-string`**) => string-output-port

Creates a string output port, which behaves as an ordinary output port, except that writes are used to create a string as output. The results of all the write operations are retrieved using `get-output-string`.

procedure: (**`get-output-string`** `string-output-port`) => string

Returns the string that was created by zero or more writes to a string output port. If no writes were performed on the string output port, an empty string ("") is returned. After this call, the provided string output port is reset to its initial, empty state.

procedure: (**`call-with-input-string`** `string procedure`) => value

Calls *`procedure`* with a new string input port created from *`string`*. The result of the thunk is returned.

procedure: (**`call-with-output-string`** `procedure`) => string

Calls *`procedure`* with a new string output port. The contents of the string-output-port are returned when the procedure returns.

procedure: (**`with-input-from-string`** `string thunk`) => value

Evaluates *`thunk`* with a string-input-port created from *`string`* as the `current-input-port` for the duration of the evaluation.

procedure: (**`with-output-to-string`** `thunk`) => string

Evaluates *`thunk`* with a string-output-port created as the `current-output-port` for the duration of the evaluation. When the thunk returns, the contents of the string-output-port are returned.

procedure: (**`string-input-port?`** `value`) => #t/#f

Returns #t if *`value`* is a string input port, #f otherwise.

procedure: (**`string-output-port?`** `value`) => #t/#f

Returns #t if *`value`* is a string output port, #f otherwise.

> **Note:** This interface complies with SRFI-6 (Basic String Ports).

5.1.5. Binary Ports and Block IO

Requires: **(import *binary-io*)**

In addition to the R^{5}RS I/O functions, *SISC* provides a symmetric set of functions for reading and writing binary data to and from ports with no character set translation. These ports are operated on using the binary I/O functions described below. Using character-oriented operations (such as the traditional R^{5}RS functions `read`, `read-char`, `display`, etc.) is an error. Binary ports also provide block input/output functions, that allow a Scheme program to read blocks of more than one byte of data at a time from binary ports. *SISC* stores data that is read or is to be written in block fashion in a binary buffer (see Section 9.1.5).

procedure: (**`open-binary-input-file`** `url`) => binary-input-port

Creates an input port in the same manner as R^{5}RS `open-input-file`, producing an input port that does no character-set decoding on the bytes read as input.

procedure: (**`open-binary-output-file`** `url [auto-flush]`) => binary-output-port

Creates an output port in the same manner as `open-output-file`, producing an output port that does no character-set encoding.

procedure: (**`call-with-binary-input-file`** `url procedure`) => value

Calls *`procedure`* with a new binary input port attached to *`url`*. The result of the thunk is returned.

procedure: (**`call-with-binary-output-file`** `url procedure`) => value

Calls *`procedure`* with a new binary output port attached to *`url`*. The result of the thunk is returned.

procedure: (**`with-binary-input-from-file`** `url thunk`) => value

Evaluates *`thunk`* with a binary input port attached to a file opened for reading from *`url`* as the `current-input-port` for the duration of the evaluation. The port is closed when *`thunk`* returns normally.

procedure: (**`with-binary-output-to-file`** `url thunk`) => value

Evaluates *`thunk`* with a binary input port attached to a file opened for writing to *`url`* as the `current-output-port` for the duration of the evaluation. The port is closed when *`thunk`* returns normally.

There are several operations specifically available for use on binary ports.

procedure: (**`peek-byte`** `[binary-input-port]`) => integer

Similar to `peek-char`, reads ahead one byte in the stream, returning the next byte available but not advancing the stream. The byte is returned as an integer. The current input port is used unless specified.

procedure: (**`read-byte`** `[binary-input-port]`) => integer

Reads a single byte from the stream, advancing the stream and returning the byte as an integer. The current input port is used unless specified.

procedure: (**`read-block`** `buffer offset count [binary-input-port]`) => integer

Reads up to *`count`* bytes of data from the current input port or the *`binary-input-port`* parameter if provided, into the binary buffer *`buffer`* starting at position *`offset`*. Note that less than *`count`* bytes may be read. The number of bytes actually read is returned. If the end-of-file is encountered before any bytes could be read, `#!eof` will be returned.

procedure: (**`write-byte`** `integer [binary-output-port]`) => undefined

Writes a single byte specified as an integer to the given *`binary-output-port`* if provided, current output port otherwise.

procedure: (**`write-block`** `buffer offset count [binary-output-port]`) => undefined

Writes *`count`* bytes of data from the provided *`buffer`* at starting point *`offset`* to the given *`binary-output-port`* or to the current output port if unspecified. Exactly *`count`* bytes will be written.

procedure: (**`binary-input-port?`** `value`) => #t/#f

Returns #t if *`value`* is a binary input port, #f otherwise.

procedure: (**`binary-output-port?`** `value`) => #t/#f

Returns #t if *`value`* is a binary output port, #f otherwise.

5.1.5.1. Buffer I/O

Requires: **(import *buffer-io*)**

A module is also provided for input and output to and from binary buffers using buffer ports, similar to character I/O to and from strings using string ports.

procedure: (**`open-input-buffer`** `buffer`) => buffer-input-port

Creates a buffer input port whose bytes are read from the provided buffer. Bytes will be returned from any read operation on the port until the end of the buffer is reached. Read calls after reaching the end of the buffer will return #!eof.

procedure: (**`open-output-buffer`**) => buffer-output-port

Creates a buffer output port, which behaves as an ordinary output port, except that writes are used to create a buffer as output. The results of all the write operations are retrieved using `get-output-buffer`.

procedure: (**`get-output-buffer`** `buffer-output-port`) => buffer

Returns the buffer that was created by zero or more writes to a buffer output port. If no writes were performed on the buffer output port, an empty buffer is returned. After this call, the provided buffer output port is reset to its initial, empty state.

procedure: (**`call-with-input-buffer`** `buffer procedure`) => value

Calls *`procedure`* with a new buffer input port created from *`buffer`*. The result of the thunk is returned.

procedure: (**`call-with-output-buffer`** `procedure`) => buffer

Calls *`procedure`* with a new buffer output port. The contents of the buffer-output-port are returned when the procedure returns.

procedure: (**`with-input-from-buffer`** `buffer thunk`) => value

Evaluates *`thunk`* with a buffer-input-port created from *`buffer`* as the `current-input-port` for the duration of the evaluation.

procedure: (**`with-output-to-buffer`** `thunk`) => buffer

Evaluates *`thunk`* with a buffer-output-port created as the `current-output-port` for the duration of the evaluation. When the thunk returns, the contents of the buffer-output-port are returned.

5.1.6. Java Ports

Requires: **(import *java-io*)**

This module provides procedures to convert between Scheme and Java I/O types. In general, binary Scheme ports map to plain Java streams, while character Scheme ports map to Java readers and writers.

The following example is somewhat contrived, but illustrates a common usage pattern:

```
;; A convoluted Hello World example.
(import s2j)
(import java-io)

(define-java-classes
  (<java.lang.system> |java.lang.System|))

(define-generic-java-field-accessors
  (:jout out))

(let ((stdout (open-character-output-port
               (->binary-output-port
                (:jout (java-null <java.lang.system>)) #t) #t)))
  (display "Hello, world!" stdout))
```

procedure: (**`->binary-input-port`** `jinput-stream`) => binary-input-port

Returns a binary input port associated to the `java.io.InputStream` object passed as the *`jinput-stream`* parameter.

procedure: (**`->binary-output-port`** `joutput-stream [aflush?]`) => binary-output-port

Returns a binary input port associated to the `java.io.OutputStream` object passed as the *`joutput-stream`* parameter. If the optional boolean *`aflush?`* parameter is not provided, the port will not autoflush by default.

procedure: (**`->character-input-port`** `jreader`) => character-input-port

Returns a character input port associated to the `java.io.Reader` object passed as the *`jreader`* parameter.

procedure: (**`->character-output-port`** `jwriter [aflush?]`) => character-output-port

Returns a binary input port associated to the `java.io.Writer` object passed as the *`joutput-stream`* parameter. If the optional boolean *`aflush?`* parameter is not provided, the port will not autoflush by default.

procedure: (**`->jinput-stream`** `input-port`) => jinput-stream

Returns a `java.io.InputStream` object associated to the given Scheme *`input-port`*. The function produces an error if a character port is passed.

procedure: (**`->joutput-stream`** `output-port`) => joutput-stream

Returns a `java.io.OutputStream` object associated to the given Scheme *`output-port`*. The function produces an error if a character port is passed.

procedure: (**`->jreader`** `character-input-port`) => jreader

Returns a `java.io.Reader` object associated to the given Scheme *`character-input-port`*.

procedure: (**`->jwriter`** `character-output-port`) => jwriter

Returns a `java.io.Writer` object associated to the given Scheme *`character-output-port`*.

Scheme ports can also be used from Java. See Section 8.1.5.3 for information.

5.1.7. Serialization

Requires: **(import *serial-io*)**

With `read` and `write`, Scheme values are read and written in a standardized, textual external representation. However, this external representation only fully describes a limited subset of Scheme types. For instance it is impossible to read/write a procedure, or closure, or continuation.

SISC provides a special composed port type and procedures for reading and writing *any* Scheme value using a binary representation. The (de)serialization preserves the referential structure of the object graph comprising the serialized values.

Serial ports are composed onto and thus are binary ports, i.e. all operations applicable to binary ports also apply to serial ports.

5.1.7.1. Port Creation and Identification

procedure: (**`open-serial-input-port`** `binary-input-port`) => serial-input-port

Creates a serial input port which reads external representations of Scheme values from the given binary input port.

procedure: (**`open-serial-output-port`** `binary-output-port [auto-flush]`) => serial-output-port

Creates a serial output port which can be used to write external representations of Scheme values to the provided binary output port.

procedure: (**`serial-input-port?`** `value`) => #t/#f

Returns #t if *`value`* is a serial input port, #f otherwise.

procedure: (**`serial-output-port?`** `value`) => #t/#f

Returns #t if *`value`* is a serial output port, #f otherwise.

5.1.7.2. Serial Port Wrappers

procedure: (**`call-with-serial-input-port`** `binary-input-port procedure`) => value

Calls *`procedure`* with a new serial input port attached to *`binary-input-port`*. The result of the thunk is returned.

procedure: (**`call-with-serial-output-port`** `binary-output-port procedure`) => value

Calls *`procedure`* with a new serial output port attached to *`binary-output-port`*. The result of the thunk is returned.

procedure: (**`with-serial-input-from-port`** `binary-input-port thunk`) => value

Evaluates *`thunk`* with a serial input port attached to the specified *`binary-input-port`* as the `current-input-port` for the duration of the evaluation. The port is *not* closed when *`thunk`* returns.

procedure: (**`with-serial-output-to-port`** `binary-output-port thunk`) => value

Evaluates *`thunk`* with a serial input port attached to the specified *`binary-output-port`* as the `current-output-port` for the duration of the evaluation. The port is *not* closed when *`thunk`* returns.

procedure: (**`call-with-serial-input-file`** `url procedure`) => value

Calls *`procedure`* with a new serial input port attached to *`url`*. The result of the thunk is returned.

procedure: (**`call-with-serial-output-file`** `url procedure`) => value

Calls *`procedure`* with a new serial output port attached to *`url`*. The result of the thunk is returned.

procedure: (**`with-serial-input-from-file`** `url thunk`) => value

Evaluates *`thunk`* with a serial input port attached to a file opened for reading from *`url`* as the `current-input-port` for the duration of the evaluation. The port is closed when *`thunk`* returns normally.

procedure: (**`with-serial-output-to-file`** `url thunk`) => value

Evaluates *`thunk`* with a serial input port attached to a file opened for writing to *`url`* as the `current-output-port` for the duration of the evaluation. The port is closed when *`thunk`* returns normally.

5.1.7.3. Serialization Procedures

procedure: (**`deserialize`** `[serial-input-port]`) => value

Reads a Scheme value from an external representation retrieved from *`serial-input-port`*. If *`serial-input-port`* is absent the data is read from the current input port.

procedure: (**`serialize`** `value [serial-output-port]`) => undefined

Writes an external representation of *`value`* to *`serial-output-port`*. If *`serial-output-port`* is absent the data is written to the current output port.

5.2. Networking

Requires: **(import *networking*)**

The *SISC* Networking library provides a mechanism for creating and manipulating IP network protocols as standard Scheme ports. *SISC* supports TCP, UDP, and Multicast UDP. Each is described in the sections that follow.

Each protocol provides one or more *socket constructors*. These functions produce a Socket handle, which is represented in *SISC* as `#<socket>`. A socket handle is then used to obtain Scheme ports.

IP addresses and network hostnames are represented as strings in *SISC*. Unless otherwise noted, the network library functions that require an address may take a network address as a string which may be any of:

- A hostname, to be resolved through the domain name system.

- An IPv4 network address in the standard dotted quad format. (RFC-791)
- An IPv6 network address in colon separated hexadecimal form, and zero-shortened form. (RFC-2373)

IPv6 addresses must be supported by the underlying operating system. An error may be raised if the address is not supported. All IP port values must be exact integers in the proper range.

5.2.1. IP Addressing

Several utility functions are provided for manipulating IP addresses. These are described below.

procedure: (**`get-host-ip-by-name`** `hostname`) => string

Attempts to resolve a hostname provided as a string into an IP address in dotted-quad form. If the host cannot be found, #f is returned.

procedure: (**`get-host-name-by-ip`** `ip-address`) => string

Attempts a reverse lookup of the given dotted-quad address to determine a registered domain name. If unsuccessful, #f is returned.

procedure: (**`get-local-host`**) => string

Attempts to determine the Internet visible IP address of the local machine. If successful, this address is returned in dotted-quad notation. #f is returned otherwise.

5.2.2. Socket Operations

Once obtained using a protocol specific constructor, a Socket Handle allows manipulation of common socket options, the creation of Scheme input/output ports, and closing of the socket.

procedure: (**`socket?`** `value`) => #t/#f

Returns true if and only if the provided value is a socket.

procedure: (**`open-socket-input-port`** `socket [encoding]`) => input-port

Opens a character input port to the socket. If the optional *`encoding`* parameter is provided, the character port created will use the specified encoding rather than the default.

procedure: (**`open-socket-output-port`** `socket [encoding] [auto-flush]`) => output-port

Opens a character output port to the socket. If provided, the boolean argument specifies whether the given port should be set to auto-flush mode. If unspecified, the port does *not* auto-flush. If the optional *`encoding`* parameter is provided, the character port created will use the specified encoding rather than the default.

procedure: (**`open-binary-socket-input-port`** `socket`) => binary-input-port

Opens a binary input port to the socket.

procedure: (**`open-binary-socket-output-port`** `socket [auto-flush]`) => binary-output-port

Opens a character output port to the socket. If provided, the boolean argument specifies whether the given port should be set to auto-flush mode. If unspecified, the port does *not* auto-flush.

procedure: (**`close-socket`** `socket`) => unspecified

Closes an IP socket.

The port-obtaining functions above work on most sockets. An exception applies for TCP server sockets, which are used only to obtain connected TCP sockets.

5.2.3. TCP

The most commonly used Internet protocol maps most favorably to Scheme's input/output model. Writing to an output port retrieved from a TCP socket writes the data to that socket. Reading from an input port reads from the connected socket. One important note is that one can control the amount of data that fills a TCP packet by using an output port that does not auto-flush. Data is written to the port until one considers the packet complete, and then uses `(flush-output-port port)` to complete the packet. Note also that this does not *guarantee* that one gets the desired packet size, but does allow one to construct reasonably sized packets.

TCP sockets are obtained one of two ways. Either one creates an outgoing connection to another listening host and then subsequently obtains a socket handle, or one creates a *listening socket* and then obtains a socket by waiting for an incoming connection on the specified port. In either case, the result is a socket handle with an available input and output port that can be obtained using a function in the previous section.

procedure: (**`open-tcp-socket`** `host port`) => socket

Attempts to connect to the host at the given hostname or IP address encoded as a string, at the given TCP port specified as an integer. An error is raised if the host cannot be found or the connection fails. If successful, a socket is returned.

procedure: (**`open-tcp-listener`** `port [interface-address]`) => server-socket

Creates a TCP server socket, which may only be used with accept-tcp-socket, or closed. The server socket will listen on the integer port specified. If provided, the interface-address, a string specifies the address of a local interface to bind to. If not provided, the port is bound on all available interfaces. An error is raised if the socket cannot be bound and set listening.

procedure: (**`server-socket?`** `value`) => #t/#f

Returns true if and only if the provided value is a server socket.

procedure: (**`accept-tcp-socket`** `server-socket`) => socket

Accepts an incoming connection on the provided server-socket, and returns a TCP socket handle. This function will block until an incoming connection is made, or, if set, the socket timeout is exceeded. If the latter happens, an error will be raised.

procedure: (**`set-so-timeout!`** `socket timeout`) => undefined

Sets the socket timeout on a socket. The socket can be either a server socket or connected socket. In the former case, this value specifies the number of milliseconds that an accept-tcp-socket can wait before timing out. In the latter, the value specifies the number of milliseconds that can elapse during a read call before timing out.

5.2.4. TLS and SSL

Sockets which are encrypted using the Secure Sockets Layer (SSL) or Transport Security Layer (TLS) can be created an accepted as well. This functionality can heavily depend on the underlying support in the JVM, including security settings, restrictions, installed certificates, etc. For information on the Java Secure Socket Extension (JSSE) and its setup, refer to the *JSSE documentation* (http://java.sun.com/products/jsse/reference/docs/index.html) page at Sun.

The SSL functionality is built atop the sockets and server sockets functionality as in TCP networking above. A separate constructor for sockets (`open-ssl-socket`) and listening sockets (`open-ssl-listener`) exist, and produce sockets which are used identically to TCP sockets as previously described.

procedure: (**`open-ssl-socket`** `host port [auto-close]`) => socket

Attempts to connect to the host at the given hostname or IP address encoded as a string, at the given TCP port specified as an integer, and establish an SSL connection using the default cipher suites and protocols available. An error is raised if the host cannot be found or the connection fails. If successful, a socket is returned.

procedure: (**`open-ssl-listener`** `port [interface-address]`) => server-socket

Creates an SSL TCP server socket, which may only be used with accept-tcp-socket, or closed. The server socket will listen on the integer port specified. If provided, the interface-address, a string specifies the address of a local interface to bind to. If not provided, the port is bound on all available interfaces. An error is raised if the socket cannot be bound and set listening.
The cipher suites, protocols, and modes which this server socket will accept are set with the following functions.

procedure: (**`get-enabled-cipher-suites`** `ssl-server-socket`) => list

Returns a list of strings naming the cipher suites which are enabled for this server socket.

procedure: (**`set-enabled-cipher-suites!`** `ssl-server-socket suite-list`) => list

Accepts an ssl server socket and a list of strings naming the cipher suites which should be available for negotiation with the remote end. The previous list is returned.

procedure: (**`get-enabled-protocols`** `ssl-server-socket`) => list

Returns a list of strings naming the security protocols which are enabled for this server socket.

procedure: (**`set-enabled-protocols!`** `ssl-server-socket protocol-list`) => list

Accepts an ssl server socket and a list of strings naming the protocols which should be available for negotiation with the remote end. The previous list is returned.

procedure: (**`session-creation-permitted?`** `ssl-server-socket`) => boolean

Returns true if new SSL/TLS sessions can be created by the sockets obtained from this server socket.

procedure: (**`set-session-creation-permitted!`** `ssl-server-socket boolean`) => boolean

Sets whether new SSL/TLS sessions can be created by the sockets obtained from this server socket. The previous value is returned.

procedure: (**`is-client-mode?`** `ssl-server-socket`) => boolean

Returns true if the SSL server socket will be in the rare client mode after accepting the connection, rather than the more common server mode.

procedure: (**`set-client-mode!`** `ssl-server-socket boolean`) => boolean

Sets whether the SSL server socket will be in client mode when accepting new connections. The previousvalue is returned.

procedure: (**`get-client-auth`** `ssl-server-socket`) => symbol or #f

Returns the necessity of the remote client to authenticate to this server socket. The returned values are either `needed`, indicating the remote must authenticate, `wanted`, indicating the remote will be requiested to authenticate but is not required to, or `#f`, indicating the remote will not be requested to authenticate.

procedure: (**`set-client-auth!`** `ssl-server-socket client-auth-mode`) => symbol or #f

Sets the client authentication requirements, as described in `get-client-auth` above. One of the above values must be specified. The previous value is returned.

5.2.5. UDP

UDP sockets can be obtained for both receive only and send/receive sessions.

> **Note:** The behavior of the `char-ready?` function is somewhat more difficult to predict on a UDP input port. The function will return `#t` only when a previous datagram contained more bytes than were requested by the read operation that received it.

procedure: (**`open-udp-listen-socket`** `listen-port [interface-address] [datagram-size]`) => udp socket

Opens a UDP socket that listens on *`listen-port`* (optionally bound to only the interface on *`interface-address`*). If provided, *`datagram-size`* specifies the buffer size (in bytes) for receiving UDP datagrams. Datagrams larger than *`datagram-size`* are truncated to that size. If unspecified, the default datagram size is 1500 bytes.

procedure: (**`open-udp-socket`** `remote-host remote-port`) => udp socket

Opens a UDP socket for sending datagrams to the Internet host specified by *`remote-host`*, on port *`remote-port`*.

After obtaining a UDP socket, input and output ports can be obtained in the usual manner. It is an error to attempt to obtain an output-port from a listening UDP socket, or an input port from a sending UDP socket.

UDP input ports behave as ordinary input ports. When a datagram arrives as a result of any read operation on the port, their entire contents are stored in a buffer of length *`datagram-size`* bytes. Successive read operations return data from that buffer until it is exhausted, at which point a read operation will cause the UDP socket to listen for another datagram.

UDP output ports should be treated with some care, however. If a UDP output port was obtained in auto-flush mode, each write operation to the output port will cause a new datagram to be sent. Control over the size of the datagram must be maintained by using a port that does not auto-flush, writing the desired data, and flushing once the amount of data that the user wants to occupy a single UDP datagram is reached. The behavior of constructing very large UDP packets is undefined. The packet may be silently dropped or (more likely) fragmented at the IP layer.

5.2.6. Multicast UDP

SISC provides support for IP multicast UDP datagrams as well. This allows a program to both send and receive to an IP multicast group. Multicast UDPs are an extension of ordinary UDP. Thus all I/O operations on a Multicast UDP socket are subject to the same semantics as an ordinary UDP socket.

> **Note:** The Multicast UDP library requires that the underlying operating system's IP networking stack support Multicast. The functions described here may produce an error if the operating system does not.

A program wishing to use multicast UDP sockets must first obtain a multicast socket for either listening to a multicast group, or for both listening and sending to such a group.

procedure: (**`open-multicast-socket`** `listen-port [interface-address] [datagram-size]`) => multicast udp socket

Opens a multicast UDP socket that listens on *`listen-port`* (optionally bound only to the interface addressed by *`interface-address`*. If provided, *`datagram-size`* specifies the buffer size (in bytes) for receiving UDP datagrams. Datagrams larger than *`datagram-size`* are truncated to that size. If unspecified, the default datagram size is 1500 bytes.

procedure: (**`open-multicast-socket`** `group port [interface-address] [datagram-size]`) => multicast udp socket

Opens a multicast UDP socket for sending datagrams to the specified multicast *`group`*, on the specified *`port`*. The returned socket will also be capable of listening to that group on the same port (and optionally bound only to the interface addressed by *`interface-address`*), though the socket will not initially be a member of the group. If provided, *`datagram-size`* specifies the buffer size (in bytes) for receiving UDP datagrams. Datagrams larger than *`datagram-size`* are truncated to that size. If unspecified, the default datagram size is 1500 bytes.

Once a sending socket has been obtained (the second form), an output-port can be obtained in the usual manner, and datagrams can be immediately sent to the multicast group. To receive datagrams, sockets returned from both forms must *join* a multicast group.

A multicast group is specified by a class D IP address and by a standard UDP port number. Class D IP addresses are in the range 224.0.0.0 to 239.255.255.255, inclusive. The address 224.0.0.0 is reserved and should not be used.

Groups are joined and left using the following functions:

procedure: (**`join-multicast-group`** `multicast-socket group`) => undefined

Causes the given multicast socket to join the group specified by the Internet address in *`group`*. Once joined, read operations on an obtained input-port will be able to receive datagrams destined to that group.

procedure: (**`leave-multicast-group`** `multicast-socket group`) => undefined

Causes the given multicast socket to leave the group specified by the Internet address in *`group`*. Read operations on any input ports obtained from this socket will no longer receive datagrams from that group.

A single multicast socket can simultaneously listen to more than one multicast group. A socket can only send to one group, however: the group it was constructed with.

Multicast packets are limited in extent by their time-to-live. Each time a multicast packet crosses a router, its ttl is decremented. In this manner, one can send datagrams only to local networks or subnetworks, as well as more grand scopes. The TTL of a socket is set using `set-multicast-ttl!`

procedure: (**`set-multicast-ttl!`** `multicast-socket ttl`) => undefined

Sets the multicast TTL of the given socket to *`ttl`*, an integer. All datagrams sent after this call will have their TTL set to the new value.

Valid multicast TTLs are in the range 0 (restricted to the same host) to 255 (unlimited in scope).

5.3. User Defined Ports

Requires: **(import *custom-io*)**

SISC provides the ability to create new I/O port types from within Scheme. To add a new port type, one invokes a custom port constructors described below, passing procedures (Scheme or otherwise) which implement the operations required by that port. Each custom port constructor returns a Scheme port which may be used with any of the *SISC* or R^{5}RS I/O functions. Port implementors may also use an associated *port-local* value to coordinate state for some specialized ports.

procedure: (**`set-port-local!`** `custom-port value`) => value

Sets the port-local value of the given custom port.

procedure: (**`port-local`** `wrapper-stream`) => value

Gets the port-local value of the given custom port.

procedure: (**`make-custom-character-input-port`** `read read-string ready? close`) => character-input-port

Creates a character-input-port whose functionality is implemented by the four provided fundamental procedures.

The fundamental procedures required are as follows:
(**`read`** `port`) => integer

Return a character as an integer value, or -1 if the end of stream has been reached. *`port`* is a reference to the custom Scheme port which contains the procedure.
(**`read-string`** `port mutable-string offset count`) => integer

Reads up to *`count`* characters into the given mutable string at position *`offset`*. The number of actual characters read are returned as an integer value, or -1 if the end of stream has been reached. *`port`* is a reference to the custom Scheme port which contains the procedure.
(**`ready?`** `port`) => boolean

Returns a non-false value if one or more characters are available for reading, false otherwise.
(**`close`** `port`) => undefined

Closes the port.

procedure: (**`make-custom-binary-input-port`** `read read-block available close`) => binary-input-port

Creates a binary-input-port whose functionality is implemented by the four provided fundamental procedures.

The fundamental procedures required are as follows:
(**`read`** `port`) => integer

Return a byte as an integer value, or -1 if the end of stream has been reached. *`port`* is a reference to the custom Scheme port which contains the procedure.
(**`read-block`** `port buffer offset count`) => integer

Reads up to *`count`* bytes into the given binary buffer at position *`offset`*. The number of actual bytes read are returned as an integer value, or -1 if the end of stream has been reached. *`port`* is a reference to the custom Scheme port which contains the procedure.
(**`available`** `port`) => integer

Returns a count of the number of bytes which are available for reading. This number need not be accurate.
(**`close`** `port`) => undefined

Closes the port.

procedure: (**`make-custom-character-output-port`** `write write-string flush close`) => character-output-port

Creates a character-output-port whose functionality is implemented by the four provided fundamental procedures.

The fundamental procedures required are as follows:
(**`write`** `port byte`) => undefined

Writes a character represented as an integer value. *`port`* is a reference to the custom Scheme port which contains the procedure.
(**`write-string`** `port string offset count`) => void

Writes *`count`* characters from the given string at position *`offset`*. *`port`* is a reference to the custom Scheme port which contains the procedure.
(**`flush`** `port`) => undefined

Flushes any unwritten characters to the stream.
(**`close`** `port`) => undefined

Closes the port.

procedure: (**`make-custom-binary-output-port`** `write write-block flush close`) => binary-output-port

Creates a binary-output-port whose functionality is implemented by the four provided fundamental procedures.

The fundamental procedures required are as follows:

The fundamental procedures required are as follows:
(**`write`** `port byte`) => undefined

Writes a byte represented as an integer value. *`port`* is a reference to the custom Scheme port which contains the procedure.
(**`write-block`** `port buffer offset count`) => void

Writes `count` bytes from the given binary buffer at position `offset`. `port` is a reference to the custom Scheme port which contains the procedure.
(**flush** `port`) => undefined

Flushes any unwritten characters to the stream.
(**close** `port`) => undefined

Closes the port.

Finally, the underlying procedures which implement a custom port can be retrieved with the following function:

procedure: (**custom-port-procedures** `custom-port`) => list

Returns the procedures which implement a custom port as a list.

As an example, here are String Ports, implemented as user defined ports:

Example 5-1. User defined string ports

```
(import custom-io)

;; String Input Ports

(define (sio/read port)
  (let ([local (port-local port)])
    (let ([ptr (vector-ref local 1)])
      (if (= (vector-ref local 2) ptr)
          -1
          (let ([c (char->integer
                     (string-ref (vector-ref local 0) ptr))])
            (vector-set! local 1 (+ ptr 1))
            c)))))

(define (sio/read-string local buffer offset length)
  (let ([local (port-local port)])
    (let ([str (vector-ref local 0)]
          [strlen (vector-ref local 2)])
      (do ([i offset (+ i 1)]
           [j (vector-ref local 1) (+ j 1)]
           [c 0 (+ c 1)])
          ((or (= i (+ offset length))
               (= j strlen))
           (vector-set! local 1 (+ ptr 1))
           c)
        (string-set! buffer i (string-ref str j))))))

(define null (lambda args (void)))

(define (open-input-string str)
  (unless (string? str)
    (error 'open-input-string "expected string, got '~a'.~%" str))
  (let ([port (make-custom-character-input-port
               sio/read sio/read-string null null)])
    ; Use the port local value to store the string and a pointer
    ; for the current position
    (set-port-local! port (vector str 0 (string-length str)))
    port))


;; String Output Ports

(define (sio/write port char)
  (set-port-local! port
    (cons char (port-local port))))

(define (sio/write-string port string offset length)
  (set-port-local! port
   (append (reverse (string->list (substring string offset (+ length offset))))
           (port-local port))))

(define (open-output-string)
  (let ([port
         (make-custom-character-output-port
          sio/write sio/write-string null null)])
    (set-port-local! port '())
    port))

(define (get-output-string port)
  (flush-output-port port)
  (let ([str (apply string
                    (reverse (port-local port)))])
    (set-port-local! port '())
    str))
```

5.4. Miscellaneous

5.4.1. Pretty-Printing

SISC includes a pretty-printer, a function that behaves like `write`, but introduces whitespace in order to make the output of data more readable to humans.

procedure: (**`pretty-print`** `value [output-port]`) => unspecified

Pretty-prints the specified value, either to the specified output-port, or to the console if no output-port is specified.

5.4.2. Source Loading

The `load` procedure accepts URLs as well as ordinary file names. See Section 5.1.1 for details on what kinds of URLs are supported.

The file name passed to `load` is resolved relative to the `current-url` parameter. During the execution of `load`, `current-url` is set to the loaded file, so that any invocations of `load` from the loaded file resolve the given file name relative to the file currently being loaded. For example, lets assume we have a web site that serves the following files:

```
 ;;;contents of http://foo.com/bar/bar1.scm ;;;
(load "bar2.scm")
;;;contents of http://foo.com/bar/bar2.scm ;;;
(load "/baz/baz1.scm")
;;;contents of http://foo.com/baz/baz1.scm ;;;
(load "../baz/baz2.scm")
;;;contents of http://foo.com/baz/baz2.scm ;;;
(display "Hello")
```

Invoking

```
(load "http://foo.com/bar/bar1.scm")
```

results in each file being loaded; with the last file in the chain, `baz2.scm`, displaying `Hello`.

The `load` function supports many types of files which contain executable code. `load` will attempt to determine (primarily by the file's extension) which type of file is being loaded to load that file in the correct manner. The file types currently supported are those described in Section 2.5.

When a pure source file is loaded, each s-expression is evaluated in sequence, exactly as if entered into the REPL one s-expression at a time.

5.4.3. Location Tracking

SISC allows the location of input, i.e. the file name, line number, and column number, to be tracked when reading from an input port.

procedure: (**`open-source-input-file`** `url`) => input-port

This procedure behaves the same as `open-input-file`, except that it also tracks the location of the input.

procedure: (**`input-port-location`** `input-port`) => list

Returns the current location information associated with *`input-port`*. The return value is an association list containing the following keys: `source-file`, `line-number`, `column-number`. If no location information is available, #f is returned.

5.4.4. Locating Resources

SISC provides a mechanism for locating and subsequently loading named resources, such as Scheme source files, Scheme data files, property files. The resources are located using the mechanism described in Section 5.4.6. This allows Scheme programs to load resources in a portable, J2EE-compliant manner.

procedure: (**`find-resource`** `string`) => url

Locates the resource named by *`string`* on the Java class path. The resource location is returned as a URL suitable for *SISC* I/O operations. If the resource cannot be found, #f is returned.

procedure: (**`find-resources`** `string`) => url-list

Locates the resource named by `string` on the Java class path. The resource locations are returned as a list of URLs suitable for *SISC* I/O operations. If the resource cannot be found, an empty list is returned.

5.4.5. File Manipulation

Requires: **(import *file-manipulation*)**

The file-manipulation library provides access to a number of functions for reading and manipulating files and their attributes. The file-manipulation library acts on filenames in the same manner as other Scheme file related functions, e.g. it accepts file and directory names as strings, which are resolved relative to the current URL.

The following functions act on both files and directories. With the exception of `file-exists?` and `get-parent-url`, the behavior when applying these to non-existant files or directories is undefined.

procedure: (**`file-delete!`** `filename`) => #t/#f

Attempts to remove the given file or directory. If successful, `#t` is returned.

procedure: (**`file-exists?`** `filename`) => #t/#f

Returns true if the given file or directory exists.

procedure: (**`file-is-directory?`** `filename`) => #t/#f

Returns true if the given string names an existing directory.

procedure: (**`file-is-file?`** `filename`) => #t/#f

Returns true if the given string names an existing file.

procedure: (**`file-last-modified`** `filename`) => integer

Returns the number of milliseconds since the Unix epoch (Jan 1, 1970) of the date the file or directory was last modified.

procedure: (**`file-rename!`** `source-filename dest-filename`) => #t/#f

Renames the given source file or directory to the destination. This can be used both to rename a file or directory or to move a file/directory in the same filesystem. If successful, `#t` is returned.

procedure: (**`file-set-last-modified!`** `filename unixtime`) => #t/#f

Sets the last modified date of the given filename to the given integer (in number of milliseconds since the epoch). Returns `#t` if successful.

procedure: (**`get-parent-url`** `filename`) => string

Given any URL, returns the URL of its parent. For filenames, as an example, the parent directory is returned.

The following functions operate only on files. Their behavior when applied to directories or non-existant files is undefined.

procedure: (**`file-is-readable?`** `filename`) => #t/#f

Returns `#t` if the file can be opened for reading.

procedure: (**`file-is-writeable?`** `filename`) => #t/#f

Returns `#t` if the file can be opened for writing.

procedure: (**`file-length`** `filename`) => integer

Returns the length, in bytes, of the given file.

procedure: (**`file-set-read-only!`** `filename`) => #t/#f

Sets the given file read-only. Returns `#t` if successful.

Finally, the following functions are specific to directories. Their behavior on files is undefined. The behavior of `directory-list` is undefined on non-existant directories.

procedure: (**`directory-list`** `directory`) => list of strings

Retrieves the children of the given directory, as a list of strings. Each string names one child, and is a filename relative to the given directory.

procedure: (**`make-directory!`** `directoryname`) => #t/#f

Attempts to creates the given directory. Returns `#t` if successful.

procedure: (**`make-directories!`** `directoryname`) => #t/#f

Attempts to creates the given directory and all non-existing parent directories. Returns `#t` if successful.

5.4.6. Class Loading

Some *SISC* features require classes and other resources to be loaded. By default, *SISC* will use the current thread's class loader, or, if none is present, the system class loader. *SISC* maintains a list of class path extensions onto which class and resource loading falls back. This list can be inspected and extended using the following functions:

parameter: (**`class-path-extension`**) => string-list

Retrieves the current class path extension. The class path extension is a list of strings representing URLs, typically pointing to jar files or directories. It is used a fall back during class and resource loading.

parameter: (**`class-path-extension-append!`** `class-path`) => undefined

Appends a list of URLs to the current class path extension.

The elements of the *`class-path`* list are normalized using the *`current-url`* (see Section 5.1.1), thus permitting the usage of relative URLs.

Note that the class path extension is part of the dynamic environment,, so each thread has its own setting, initially inherited from the parent thread. See Chapter 6 for more details on *SISC*'s threading semantics.

Notes

1. Handling of JAR files in URLS may be dependent on the *SISC* host language, as well as some uncommon protocols. FILE, HTTP and FTP should be expected to work with any host language.

Chapter 6. Threads and Concurrency

Requires: **(import *threading*)**

SISC provides a comprehensive library for executing Scheme code in parallel in multiple concurrent threads. This allows code for simple code for handling blocking I/O sources (such as network servers) or for the ability to do parallel computation across multiple processors.

In addition, functions are provided to ensure mutually exclusive access to data (mutexes), to assign priorities to Scheme threads, and for inter-thread signaling and synchronization.

6.1. Scheme Thread Semantics

Care has been taken to ensure that Scheme code executing concurrently in two or more threads does not result in unpredictable behavior. Assuming the executing threads do not share data, executing code in multiple threads should behave just as executing the code in a single thread.

All threads executing in the system share some resources. The top-level and symbolic environments are shared by all threads. If a thread makes a change to these environments, the change will be visible in all other threads. Unless changed by the thread, all threads share the same console input and output ports. As threads originate from a thunk created in the primordial thread or a child thread, the lexical environment captured by the thunk may include some lexical variables from the parent thread. These variables will be visible by both the parent and child threads. The lexical environments created in an executing thread are visible by that thread only (unless that thread spawns a child thread whose thunk binds it's parent's lexicals).

Some resources can be shared but may also be distinct from thread to thread. These resources are inherited from the parent thread, but may be changed by the child or parent without affecting the other. The dynamic environment (including the console input and output port, parameters, etc.) are inherited from the parent, as is the dynamic-wind stack.

When a thread begins, it is considered to be isolated from its parent in terms of the dynamic-wind stack. If a parent spawns a thread in the *during* section of a dynamic-wind call, the spawned thread *escapes* the restrictions of the dynamic-wind call. This is logical, as the parent thread may then exit from the dynamic extent even as the child thread executes, or may remain there waiting for the child thread to finish, in which case the parent has not left the dynamic extent of the call. In short, the dynamic-wind is protecting only the parent thread.

It is possible for more than one thread to access the same memory location (be it a lexical variable or a named variable in the top-level or another symbolic environment), it is also possible that interactions on shared variables can have unpredictable results. As in any multi-threaded language, unprotected access to shared variables can result in race conditions and other concurrency mishaps. If the programmer anticipates concurrent access to a shared variable and if any thread is to write to the variable, sections of code that access the variable should use a protection mechanism from Section 6.5.

A thread can complete in one of two ways. If the thunk that contains the thread's code exits, the thread will terminate and the thread handle will contain the return value of the thunk. This completion condition is called a *clean* exit. If during the execution of the thunk's body an error is raised and is not subsequently caught, the thread will terminate and the thread handle will trap the error. The error will be raised to any caller that attempts to retrieve the return value of the thread.

It is perfectly legal for a thunk to both capture and invoke continuations, even continuations created by other threads. When applying a continuation captured outside of the thread, the resources of the executing thread are used, though the thread may be accessing lexical environments created by other threads.

Once created, a thread can be in one of four states: *ready*, *running*, *finished*, or *finished-with-error*. The first two states indicate a newly created thread and a running thread, repectively. The last two represent the end stages of a thread, *finished* indicating a thread that has exited cleanly, and *finished-with-error* indicating exit with an error.

There are no guarantees that a Scheme thread will ever exit. It is perfectly valid for a thread to execute indefinitely. Furthermore, the *SISC* environment will not exit until all threads have completed, either cleanly or with a failure, unless all remaining threads are so-called *daemon* threads. Daemon threads are threads that may run indefinitely but will be forcibly terminated if no non-daemon threads (of which the primordial thread is one) are still running. Termination of a daemon thread when no non-daemon threads exist is the *only* instance where a thread can be forcibly terminated. **There is no guaranteed thread stop or destroy operation.**

6.2. Basic Thread Operations

This section describes the basic, low-level operations on threads, including how to create a thread, how to start it, how to wait for it to terminate, and how to retrieve it's result. A thread is managed in Scheme by its *thread handle*, an opaque value that is used to identify the thread. A thread handle is present as an argument to most of the thread library functions.

Threads are created with `thread/new`. This function takes as its sole argument a thunk. The body of the thunk is the code that the thread will execute when started.

procedure: (**`thread?`** `value`) => #t/#f

Returns true if and only if the provided value is a thread handle.

procedure: (**`thread/new`** `thunk`) => thread-handle

Creates a new thread handle whose code is defined in the provided *`thunk`*.

Once a thread-handle is created, the thread is in the ready state, and can be started at any time by calling `thread/start`. At this point one can also set various thread parameters, such as thread priority and daemon status.

procedure: (**`thread/start`** `thread-handle`) => undefined

Starts the thread identified by *`thread-handle`*. The thread must be in the ready state. It is an error to start a thread in any other state.

procedure: (**`thread/daemon!`** `thread-handle boolean`) => undefined

Sets the daemon status of a thread. It is an error to change the status of a thread that is not in the `'ready` state.

procedure: (**`thread/daemon?`** `thread-handle`) => #t/#f

Returns `#t` if the given thread is a daemon thread.

It is common pattern to create a thread and immediately start it. The convenience method `thread/spawn` encapsulates this operation.

procedure: (**`thread/spawn`** `thunk`) => thread-handle

Creates a new thread handle whose code is defined in the provided *`thunk`*, and starts the thread.

Once started, the thread will be in the running state. The body of the thunk is now being evaluated in parallel to the parent thread. The thread will remain in the running state until it completes and enters one of the two finished states. The state can be read using `thread/state`.

procedure: (**`thread/state`** `thread-handle`) => symbol

Returns the state of the thread identified by *`thread-handle`*. The state is one of `'ready`, `'running`, `'finished`, or `'finished-with-error`.

The parent thread may continue executing its own code, or may attempt to *join* the child thread. To join another thread is to wait until the other thread has completed. The parent thread can join a child using `thread/join`. The parent can wait indefinitely or may specify a timeout, after which the `thread/join` command will return with `#f`.

procedure: (**`thread/join`** `thread-handle [timeout]`) => boolean

Attempts to join with the indicated thread. If the thread terminates, thread/join will return a non-false value. If a timeout is specified thread/join will only wait *`timeout`* milliseconds for the thread to complete. If the thread does not terminate before the timeout, `#f` will be returned.

It is possible to join on an already completed thread. In such a case the join will immediately return #t. The behavior of a join on a thread in the ready state is unspecified, and may cause an error. Finally, it is possible that a join may return `#f`, even if no timeout is specified. Though unlikely, programmers who wish to wait indefinitely for a thread to complete should check the return value of `thread/join` and repeat the join until `#t` is returned.

If enabled (using the permitInterrupts configuration parameter, see Section 2.4.2), running threads may be interrupted at both the host language level and when executing Scheme code with the `thread/interrupt` function. If disabled (the default), only a host-language interrupt signal can be sent.

procedure: (**`thread/interrupt`** `thread-handle`) => undefined

Sends an interrupt signal to the given thread. This will cause an error to be raised from Scheme code, and a thread interrupt in the host language.

When invoked, a signal is sent to the running thread which will cause an error to be raised from some point in its execution. If not caught, the thread will terminate in the *finished-with-error* state, and the raised error will be rethrown if `thread/result` is called. The error-continuation of the error thrown inside the thread, when invoked with no arguments, will restart the computation exactly where it left off. A thread may not properly resume if its code calls back into Scheme using any of the mechanisms described in Section 8.1.

Final Continuation of a Thread

If a thread is interrupted and later its computation is resumed by calling the error-continuation, the thread that hosts the resumed computation will exit when the computation terminates. As a consequence, if a computation that ran in a separate thread is resumed in the primordial thread (that usually hosts the REPL), the primordial thread will terminate as soon as the computation completes. This will circumvent the REPL entirely and cause *SISC* to exit if no non-daemon threads remain. In general, interrupted threads should be resumed in a newly created thread to avoid this scenario.

It is important to note that `thread/interrupt` does *not* guarantee the termination of a thread. A thread may still capture the error at the scheme level with a `with/fc`, or catch the interrupt signal if executing in the host language. In either case, the running code is not required to rethrow the error.

Once a thread has completed, the parent thread may wish to retrieve the result of the thread's thunk, be it an error or a valid result. This can be done with the `thread/result` function.

procedure: (**`thread/result`** `thread-handle`) => value

Returns the return value from a completed thread. If the thread completed with error, that error is raised from this call.

An error will be raised if an attempt is made to retrieve the result of a thread before that thread has completed.

6.3. High-level Functions

In addition to the basic thread operations, some high level syntax is provided to simplify some general case thread use.

procedure: (**`parallel`** `thunk1 thunk2 [thunks] ...`) => multiple values

Executes each thunk in its own thread in parallel. The call to parallel blocks until all the threads have finished. If all threads completed without error, the results of each thunk are returned as multiple values. If any thread raised an error, that error is raised from the call to parallel. The error is raised only after all other thunks have also completed. If more than one thunk raises an error, it is undefined which error will be raised to the caller.

6.4. Thread Scheduling

All Scheme threads created in *SISC* are preemptive and managed by a scheduler. It is possible for a program to manage the priorities of threads in order to give execution preference to higher priority threads. It is also possible for threads to give up their execution time to other blocked threads.

The priority of a thread is represented by an integer. The range of priorities and the default priority of a thread is unspecified and may be platform specific. Larger integers represent higher priorities then smaller integers. If a higher or lower priority thread is desired, the recommended procedure is to get the current priority of a thread and increment or decrement it. Though unspecified, it is possible that an error will be raised if a priority level outside the platform specific range is selected.

Though not guaranteed, the behavior of the scheduler when two threads, one with higher priority than another are both runnable but only one processor is available to run a thread, is that the higher priority thread will be selected. If only equal priority threads are available to be run, the scheduler can choose any thread to run. No guarantees are made about latency or fairness.

Thread priorities can be set by the parent thread or the thread itself. The behavior of a child thread attempting to set the parent's priority, or a sibling's priority is undefined.

procedure: (**`thread/priority`** `thread-handle`) => integer

Retrieves the current priority of the given thread.

procedure: (**`thread/priority!`** `thread-handle new-priority`) => undefined

Attempts to set the given thread's priority to the integer *`new-priority`*.

In addition to setting priority levels, a program may wish to yield its execution time temporarily to other threads. Performing a yield allows the scheduler to select a thread to run on the processor of the thread that just yielded control. It is possible that the yielding thread may be selected again, or another thread may be chosen.

procedure: (**`thread/yield`**) => undefined

Causes the currently executing thread to yield to other threads.

Finally, a thread (including the main thread) may sleep for a specified amount of time, allowing other threads to execute in the mean time. If no other threads are running, sleeping effectively pauses the program for the given time period.

procedure: (**`sleep`** `milliseconds`) => undefined

Causes the currently executing thread to sleep for the given number of milliseconds, specified as an exact integer.

6.5. Synchronization Primitives

SISC provides an implementation of mutexes and condition variables designed to support a wide range of synchronization models, including the monitor paradigm previously found in *SISC*. The implementation provides both a *mutex* and *condition-variable* first-class value for concurrency protection and inter-thread communication. Mutexes provide mutual exclusion locking, while condition variables faciliate synchronization of threads with the change in state of monitored data. *SISC*'s mutexes are reentrant, that is, if a thread locks a mutex and then attempts to lock it again, the lock is granted immediately- it does not block. Correspondingly, the mutex is not unlocked until the same number of unlock operations as previous lock operations are performed by the owning thread.

SISC's synchronization primitives map very closely to SRFI-18, which is supported by *SISC*. In fact, it is recommended that programmers write to SRFI-18 if at all possible, as they will gain maximum portability with little or no loss in efficiency on *SISC*.

Throughout the following sections, a thread is often said to *block* because of some circumstance. While a thread is blocked on some resource, other threads are allowed to execute freely, at the discretion of the scheduler.

Several functions exist for performing low level operations such as creating mutexes and condition variables. All require the mutex or condition variable as the first parameter. Mutexes are represented in

Scheme as opaque values displayed as `#<mutex>` while condition variables are represented as `#<condition-variable>`. It is important to understand that all the synchronization operations depend on the same mutex and/or condition variable being shared between any threads using the functionality.

6.5.1. Mutex Operations

In order to protect a segment of Scheme code from concurrent access, one can create a mutex that is shared by all threads that may access the segment. When entering the contested region of code (the *critical section*), a thread would call `mutex/lock`. Upon exiting the region, `mutex/unlock` is called.

procedure: (**`mutex?`** `value`) => #t/#f

Returns true if and only if the provided value is a mutex.

procedure: (**`mutex/new`**) => mutex

Creates and returns a new mutex object.

procedure: (**`mutex-of`** `value`) => mutex

Returns a mutex that is uniquely associated to the given value. Subsequent (or concurrent) calls to this function are guaranteed to return the same mutex if given the same value.

procedure: (**`mutex/lock!`** `mutex`) => undefined

Attempts to acquire the lock on the given mutex. Returns only when the lock has been successfully acquired.

procedure: (**`mutex/unlock!`** `mutex`) => undefined

Releases the lock on the given mutex. The behavior when unlocking a mutex when the running thread does not have the lock is undefined, and may raise an error.

The semantics of `mutex/lock!` ensure that only one thread can execute beyond the lock call at any one time. The first thread that reaches the call *acquires* the lock on the mutex. Any later threads will block at the call to `mutex/lock!` until the thread that owns the lock releases the lock with `mutex/unlock!`.

6.5.2. Condition Variable Operations

A condition variable allows one thread to sleep until another wakes it. A common situation is for one thread to check the status of a variable, and sleep if the *condition* is not met. While the thread sleeps, one or more separate threads may execute and satisfy the condition (by changing the state of the variable) and then *notify* the sleeping thread via the condition variable. The thread then awakes, checks the state variable, and proceeds if the condition is met. If not, it sleeps again. This construct allows for cooperation between multiple threads on a computation.

To wait on a condition variable, the `mutex/unlock!` function is again used with a condition variable as an additional argument. applied to a monitor. This will cause the thread to unlock the given mutex, then block until notified by another thread. When it is notified, or the provided timeout expires, the lock is *not* reacquired.

procedure: (**`condvar?`** `value`) => #t/#f

Returns true if the provided value is a condition variable.

procedure: (**`condvar/new`**) => condition variable

Creates a new condition variable.

procedure: (**`mutex/unlock!`** `mutex condvar [timeout]`) => #t/#f

Causes the thread to sleep until notified on the provided condition variable by another thread. This call will not return until notified, unless the optional timeout is specified and *`timeout`* milliseconds have elapsed without a notification. Before blocking, the lock on *`mutex`* is released. If the timeout is reached before notification, `#f` is returned, otherwise `#t` is returned.

Another thread may wake a single waiting thread with the `condvar/notify` operation. When called, one thread waiting on the condition variable is woken. If no threads are waiting this call has no effect. If more than one thread is waiting, exactly one will be woken. Which is woken is unspecified. If a thread wishes to wake *all* threads waiting on a given monitor, it may use the `condvar/notify-all` function.

procedure: (**`condvar/notify`** `condvar`) => undefined

Wakes exactly one thread waiting on the condition variable, if any such threads exist. If the notifying thread holds the lock on *`condvar`*, the waiting thread will not proceed until the notifying thread releases the lock.

procedure: (**`condvar/notify-all`** `condvar`) => undefined

Wakes all threads waiting on the condition variable, if any waiting threads exist. If the notifying thread holds the lock on *`condvar`*, the waiting thread will not procede until the notifying thread releases the lock.

6.5.3. High-level Concurrency

In addition to the low-level operations on mutexes and condition variables, two library functions are provided to greatly ease the construction and readability of thread-safe code.

procedure: (**`mutex/synchronize`** `mutex thunk`) => value

Protects execution of thunk as a critical section by holding the mutex's lock during evaluation of the thunk. The result of the thunk's evaluation becomes the result of the `mutex/synchronize` expression.

procedure: (**`mutex/synchronize-unsafe`** `mutex thunk`) => value

Behaves exactly as mutex/synchronize without automatic unlocking when an error is raised or a continuation escapes when executing thunk.

`mutex/synchronize` locks the mutex while the thunk provided is being executed. The lock is automatically released when the expression has completed. Also, if an error is raised or a continuation is invoked that escapes the call to `mutex/synchronize`, the lock is automatically released.

The added safety provided by `mutex/synchronize` may slow the execution of code that repeatedly calls a critical section. If the programmer is absolutely sure that no error can be raised and that no continuations will be applied to escape the call, `mutex/synchronize-unsafe` may be used. It provides no safety guarantees in those situations. If an error is raised or an escaping continuation invoked, the mutex will remain locked which could cause a deadlock if another thread attempts to acquire the lock.

Chapter 7. Types and Objects

7.1. Type System

Requires: **(import *type-system*)**

SISC's extensible type system provides programmatic access to the type information of values and provides a core set of type testing and comparison procedures. The type system is extensible in two ways. Firstly any new native types are recognised automatically. Secondly, hooks are provided for Scheme-level extensions of the type-system.

By convention, type names start with < and end with >, with normal Scheme identifier naming conventions applying for everything in-between, i.e. all lower-case with words separated by dashes. For example, `<foo-bar-baz>`. This convention helps to visually distinguish type names from names of procedures and top-level data bindings.

7.1.1. Core Procedures and Predicates

procedure: (**`type-of`** `value`) => type

Returns the type of *`value`*. There is no standard representation for types, leaving type extensions free to choose a representation that suits them most.

The procedure is equipped with an extension hook, `type-of-hook`. See Section 7.1.3 for more details on hooks. The default implementation of the hook `type-of` returns a type based on the Java type of the internal representation of *`value`*.

```
(type-of 1)  ;=> #<scheme sisc.data.Quantity>
(type-of (lambda (x) x))  ;=> #<scheme sisc.data.Closure>
```

procedure: (**`type<=`** `type1 type2`) => #t/#f

Returns #t if *`type1`* is a sub-type of *`type2`*.

The predicate is equipped with an extension hook, `type<=-hook`. See Section 7.1.3 for more details on hooks. The default implementation of the hook determines sub-typing based on the inheritance relationship of the Java types representing native types.

```
(type<= (type-of 'a) (type-of 'b))  ;=> #t
(type<= (type-of 'b) (type-of 'a))  ;=> #t
(type<= (type-of 1) (type-of 'a))  ;=> #f
(type<= (type-of 1) <number>)  ;=> #t
(type<= (type-of 'a) <symbol>)  ;=> #t
(type<= <number;> <symbol>)  ;=> #f
(type<= <symbol;> <number>)  ;=> #f
(type<= <number> <value>)  ;=> #t
(type<= <symbol> <value>)  ;=> #t
```

procedure: (**`compare-types`** `type1 type2 type3`) => ’equal,’more-specific,’less-specific

Determines the relationship of *`type1`* and *`type2`* with respect to *`type3`*. *`type3`* must be a sub-type of *`type1`* and *`type2`*. *`type1`* and *`type2`* are first compared using `type<=`. If that comparison indicates that the types are disjoint (i.e. *`type1`* is not sub-type of *`type2`*, *`type2`* is not a sub-type of *`type1`* and the types are not equal) then additional information from *`type3`* is taken into account for the comparison.

The predicate is equipped with an extension hook, `compare-types-hook` that is invoked in the case the comparison of *`type1`* with *`type2`* using `type<=` finds the two types to be disjoint. See Section 7.1.3 for more details on hooks. The default implementation of the hook returns an error

```
(compare-types <number> <value> <number>)
  ;=> 'more-specific
(compare-types <value> <number> <number>)
  ;=> 'less-specific
(compare-types <number> <number> <number>)
  ;=> 'equal
```

7.1.2. Derived Procedures and Predicates

The type system’s derived procedures and predicates are implemented in terms of the core procedures and predicates.

procedure: (**`instance-of?`** `value type`) => #t/#f

Determines whether *`value`* is an instance of *`type`*.

The predicate obtains *`value`*’s type using `type-of` and then compares it to *`type`* using `type<=`.

```
(instance-of? 1 <number>)  ;=> #t
(instance-of? 'a <symbol>)  ;=> #t
(instance-of? 1 <symbol>)  ;=> #f
(instance-of? 1 <value>)  ;=> #t
```

procedure: (**`type=`** `type1 type2`) => #t/#f

Determines whether two types are equal by comparing them using `type<=`.

```
(type= (type-of 'a) (type-of 'b))  ;=> #t
(type= (type-of 1) (type-of 'a))  ;=> #f
(type= (type-of 1) <number>)  ;=> #t
(type= (type-of 'a) <symbol>)  ;=> #t
(type= <number;> <symbol>)  ;=> #f
```

procedure: (**`types<=`** `type-list1 type-list2`) => #t/#f

Determines whether all of the types in *`type-list1`* are sub-types of the the corresponding (by position) types in *`type-list2`*.

A pair-wise comparison of the elements in the two lists using `type<=` is performed until a test returns #f, in which case #f is returned, or one (or both) of the lists has been exhausted, in which case #t is returned.

```
(types<= (list <number> <symbol>)
         (list <value> <value>))  ;=> #t
(types<= (list <number> <symbol>)
         (list <number> <number>))  ;=> #f
```

procedure: (**`instances-of?`** `value-list type-list`) => #t/#f

Determines whether all of the values in *`value-list`* are instances of the the corresponding (by position) types in *`type-list`*.

A pair-wise comparison of the elements in the two lists using `instance-of?` is performed until a test returns #f, in which case #f is returned, or one (or both) of the lists has been exhausted, in which case #t is returned.

```
(instances-of? (list 1 'a)
               (list <number> <symbol>))  ;=> #t
(instances-of? (list 1 'a)
               (list <number> <number>))  ;=> #f
```

procedure: (**`types=`** `type-list1 type-list2`) => #t/#f

Determines whether all of the types in *`type-list1`* are equal to the the corresponding (by position) types in *`type-list2`*.

A pair-wise comparison of the elements in the two lists using `type=` is performed until a difference is found, in which ase #f is returned, or one (or both) of the lists has been exhausted, in which case #t is returned.

```
(types= (list <number> <symbol>)
        (list <number> <symbol>))  ;=> #t
(types= (list <number> <symbol>)
        (list <number> <number>))  ;=> #f
```

7.1.3. Hooks

Hooks are the main mechanism by which Scheme code can extend the default type system. The core type system procedures `type-of`, `type<=` and `compare-types` all provide such hooks, called `type-of-hook`, `type-<=-hook`, and `compare-types-hook` respectively.

Extension takes place by installing labelled handler procedures on the hook, which is done by invoking the hook procedure. Installing a handler procedure with a label of an already installed procedure replaces the latter with the former.

The handler procedures are called with a *`next`* procedure as the first argument and all the arguments of the call to the hook-providing procedures as the remaining arguments. Typically a handler procedure first determines whether it is applicable, i.e. is capable of performing the requested comparison etc. If not it calls the *`next`* handler procedure, which invokes the next hook or, if no further hooks exist, the default implementation of the hooked procedure.

Example 7-1. Hook Installation

This example shows how *SISC*'s record type module adds record types to the type system by installing handler procedure on `type-of-hook` and `type<=-hook`.

```
(type-of-hook 'record
  (lambda (next o)
    (if (record? o)
        (record-type o)
        (next o))))

(type<=-hook 'record
  (lambda (next x y)
    (cond [(record-type? x)
           (if (record-type? y)
               (eq? x y)
               (type<= <record> y))]
          [(record-type? y) #f]
          [else (next x y)])))
```

7.1.4. Standard Types

The type system pre-defines bindings for the native types corresponding to all the data types defined in R5RS: `<eof>`, `<symbol>`, `<list>`, `<procedure>`, `<number>`, `<boolean>`, `<char>`, `<string>`, `<vector>`, `<input-port>`, `<output-port>`. One notable exception is that pairs and null are combined into a `<list>` type.

The type system also defines a `<value>` type that is the base type of all *SISC* values, i.e. all *SISC* values are instances of `<value>` and all types are sub-types of `<value>`.

The representations of other native types can be obtained using

procedure: (**`make-type`** `symbol`) => type

Constructs a type representing a built-in type. The *`symbol`* must denote a Java class that is a sub-class of `sisc.data.Value`, the base of the *SISC* value type hierarchy.

```
(define <record> (make-type '|sisc.modules.record.Record|))
(type<= <record> <value>)  ;=> #t
```

7.2. Generic Procedures

Requires: **(import *generic-procedures*)**

Generic procedures are procedures that select and execute methods based on the types of the arguments. Methods have a type signature, which generic procedures use for method selection, and contain a procedure which is invoked by generic procedures when the method has been selected for execution.

Generic procedures have several advantages over ordinary procedures:

- It is not necessary to come up with unique names for procedures that perform the same operation on different types of objects. This avoids cluttering the name space. All these procedures can be defined separately but yet be part of the same, single generic procedure.
- The functionality of a generic procedure can be extended incrementally through code located in different places. This avoids "spaghetti code" where adding a new type of objects requires changes to existing pieces of code in several locations.
- Code using generic procedures has a high degree of polymorphism without having to resort to ugly and hard-to-maintain test-type-and-dispatch branching.

Generic procedures make extensive use of *SISC*s type system. See Section 7.1.

The use of generic procedures proceeds through three stages:

1. definition of the generic procedure
2. adding of methods to the generic procedure
3. adding of methods to the generic procedure

The adding of methods can be interleaved with invocation, i.e. methods can be added to generic procedures while they are in use.

7.2.1. Defining Generic Procedures

There are one procedure and two special forms for defining generic procedures. Typical usage will employ one of the special forms.

procedure: (**`make-generic-procedure`** `generic-procedure ...`) => generic-procedure

Creates a generic procedure. If *`generic-procedure`* parameters are specified, then their method lists are merged, in effect combining the generic procedures into one. For more details on generic procedure combination see Section 7.2.5.1.

```
(define pretty-print1 (make-generic-procedure))
(define pretty-print2 (make-generic-procedure))
;=> <procedure>
(define pretty-print (make-generic-procedure pretty-print1
                                             pretty-print2))
```

syntax: (**`define-generic`** `name generic-procedure ...`) => void

Creates a binding for *`name`* to a new generic procedure.

This form is equivalent to `(define` *`name`* `(make-generic-procedure` *`generic-procedure`* `...))`.

```
(define-generic pretty-print1)
(define-generic pretty-print2)
(define-generic pretty-print pretty-print1 pretty-print2)
```

syntax: (**define-generics** form ...) => void
where *form* is of the form *name* or (*name generic-procedure* ...)

Creates bindings for several new generic procedures.

The form expands into several `define-generic` forms.

```
(define-generic pretty-print1)
(define-generic pretty-print2)
(define-generics
  foo
  (pretty-print pretty-print1 pretty-print2)
  bar)
```

7.2.2. Defining Methods

Methods can be define and subsequently added to generic procedures, or the two operations can be combined, which is the typical usage.

There is one procedure and one special form to create methods:

procedure: (**make-method** procedure type-list rest?) => method

Creates a new method containing *procedure* whose type signature is *type-list*. If *rest?* is #t then the procedure can take rest arguments.

Generic procedures always invoke method procedures with a special *next:* argument as the first parameter (see Section 7.2.5.3), followed by all the arguments of the generic procedure invocation. Hence *procedure* needs to accept (length *type-list*)+1 arguments.

```
(make-method (lambda (next x y) (next x y))
             (list <number> <number>)
             #f)
  ;=> <method>
(make-method (lambda (next x . rest) (apply + x rest))
             (list <number>)
             #t)
  ;=> <method>
```

syntax: (**method** signature . body) => method
where *signature* is of the form ([(next: *next*)] (*type param*) ... [. *rest*])
and *body* can contain anything that is valid inside the body of a `lambda`.

Creates a method.

This form is similar to a `lambda` form, except that all parameters must be typed. The form expands into an invocation of the `make-method` procedure.

The first parameter name in the method's signature can be the special `next:` parameter. See Section 7.2.5.3.

```
(method ((next: next)(<number> x)(<number> y)) (next x y))
  ;=> <method>
(method ((<number> x) . rest) (apply + x rest))
  ;=> <method>
```

There are two procedures to add methods to a generic procedure:

procedure: (**add-method** generic-procedure method) => void

Adds *method* to *generic-procedure*. Any existing method with the same signature as *method* is removed.
Method addition is thread-safe.

```
(define-generic m)
(add-method m (method ((next: next)(<number> x)(<number> y)) (next x y)))
(add-method m (method ((<number> x) . rest) (apply + x rest)))
```

procedure: (**add-methods** generic-procedure method-list) => void

Adds all methods in *method-list* to *generic-procedure*. Any existing method with the same signature as one of the methods in *method-list* is are removed. When several methods in *method-list* have the same signature, only the last of these methods is added.

Method addition is thread-safe. Calling add-methods instead of add-method when adding several methods to a generic procedure is more efficient.

```
(define-generic m)
(add-methods m (list (method ((next: next)(<number> x)(<number> y)) (next x y))
                     (method ((<number> x) . rest) (apply + x rest))))
```

The creation of methods and adding them to generic procedures can be combined using one of two special forms:

syntax: (**define-method** (generic-procedure . signature) . body) => void

Creates a method and adds it to *generic-procedure*.

This form is equivalent to (add-method *generic-procedure* (method *signature* . *body*)

```
(define-generic m)
(define-method (m (next: next)(<number> x)(<number> y)) (next x y)))
(define-method (m (<number> x) . rest) (apply + x rest))
```

syntax: (**define-methods** generic-procedure (signature . body) ...) => void

Creates several methods and adds them to *generic-procedure*.

This form is equivalent to (add-methods *generic-procedure* (list (method *signature* . *body*) ...)

```
(define-generic m)
(define-methods m
  [((next: next)(<number> x)(<number> y)) (next x y)]
  [((<number> x) . rest) (apply + x rest)])
```

The list of methods contained in a generic procedure can be obtained as follows:

procedure: (**`generic-procedure-methods`** `generic-procedure`) => method-list

Returns the list of methods currently associated with *`generic-procedure`*.

```
(define-generic m)
(define-methods m
  [((next: next)(<number> x)(<number> y)) (next x y)]
  [((<number> x) . rest) (apply + x rest)])
(generic-procedure-methods m)  ;=> (<method> <method>)
```

7.2.3. Invoking Generic Procedures

Generic procedures are invoked like ordinary procedures. Upon invocation, generic procedures compute a list of applicable methods, ordered by their specificity, based on the types of the parameters supplied in the invocation. If the resulting list is empty an error is raised. Otherwise the first (i.e. most specific) method is invoked. The remaining methods come into play when a method invokes the "next best matching method". See Section 7.2.5.3 for details on the method selection algorithms.

The logic by which generic procedures select methods for invocation is made accessible to the programmer through the following procedures:

procedure: (**`applicable-methods`** `generic-procedure type-list`) => method-list

Returns all methods of *`generic-procedure`* that are applicable, as determined by `method-applicable?` to parameters of the types specified in *`type-list`*. The methods are returned ordered by their specificity, determined by pair-wise comparison using `compare-methods`.

```
(define-generic m)
(define-methods m
  [((next: next)(<number> x)(<number> y)) (next x y)]
  [((<number> x) . rest) (apply + x rest)])
(applicable-methods m (list <number> <number>))
  ;=> (<method> <method>)
```

procedure: (**`method-applicable?`** `method type-list`) => #t/#f

Determines whether *`method`* is applicable to arguments of the types specified in *`type-list`*.

The rules for determining method applicability are defined in Section 7.2.5.3.

```
(method-applicable? (method ((next: next)(<number> x)(<number> y)) (next x y))
                    (list <number>))
  ;=> #f
(method-applicable? (method ((<number> x) . rest) (apply + x rest))
                    (list <number>))
  ;=> #t
```

procedure: (**`compare-methods`** `method method type-list`) => 'equal,'more-specific,'less-specific

Determines the relationship of two methods by comparing their type signatures against each other and using the supplied *`type-list`* for disambiguation. Both methods must be applicable to *`type-list`*, as determined by `method-applicable?`.

The comparison algorithm is described in Section 7.2.5.3.

```
(compare-methods (method ((next: next)(<number> x)(<number> y)) (next x y))
                 (method ((<number> x) . rest) (apply + x rest))
                 (list <number> <number>))
                ;=> 'more-specific
```

Calling a generic will dispatch on the argument types as described above. This dispatch can change if new methods are added to a generic procedure. On occasion the programmer may wish to fix the dispatch of a particular generic function, either to guarantee a specific function is called for a given part of a Scheme program, or to improve performance by avoiding the type dispatch at each call. *SISC* provides a syntactic form which allows the programmer to bind/rebind a generic procedure to a new lexical variable which is the monomorphized variant of the function call.

syntax: (**`let-monomorphic`** `bindings expressions ...`) => void
  where *`bindings`* are of the form `((`*`generic type`*`...) ...)`
  or `(((`*`binding generic`*`)` *`type`*`...) ...)`

In the former binding form, the generic procedure specified by *`generic`* is rebound lexically with the same name, and monomorphized to the method which is applicable for the given types. In the latter form, the generic is rebound lexically to the new name specified by *`binding`*. Both forms may be used in a given call to `let-monomorphic`.

The bindings are made as if by `let`, i.e. no assumptions can be made as to the order in which they are bound. The expressions are evaluated as in `let` as well, in order using an implicit `begin`.

```
(let-monomorphic ([foo-generic <number> <string>]
                  [(bar bar-generic) <char>])
  (foo-generic 3 "four")
  (bar #\x))
```

7.2.4. Procedures on Methods

Methods are instances of the abstract data type `<method>`, which has range of procedures:

procedure: (**`method?`** `value`) => #t/#f

Returns #t if *`value`* is a method, #f otherwise.

```
(method? (method ((<number> x) . rest) (apply + x rest)))
  ;=> #t
(method? (lambda (x) x))
  ;=> #f
```

procedure: (**`method-procedure`** `method`) => procedure

Returns *`method`*'s body as a procedure. Note that a method's procedure always takes a "next method" procedure as the first argument. See Section 7.2.5.3.

```
((method-procedure (method ((<number> x) . rest) (apply + x rest)))
 #f 1 2 3)
  ;=> 6
```

procedure: (**method-types** `method`) => type-list

Returns *`method`*'s type signature, i.e. the types of the declared mandatory parameters.

```
(method-types (method ((next: next)(<number> x)(<number> y) . rest) (next x y)))
  ;=> (<number> <number>)
```

procedure: (**method-rest?** `method`) => #t/#f

Returns #t if *`method`* has a rest parameter, #f otherwise.

```
(method-rest? (method ((<number> x) . rest) (apply + x rest)))
  ;=> #t
(method-rest? (method ((next: next)(<number> x)(<number> y) . rest) (next x y)))
  ;=> #f
```

procedure: (**method-arity** `method`) => number

Returns the number of mandatory arguments of *`method`*. Note that the special `next:` parameter is not counted.

```
(method-arity (method ((<number> x) . rest) (apply + x rest)))
  ;=> 1
(method-arity (method ((next: next)(<number> x)(<number> y) . rest) (next x y)))
  ;=> 2
```

procedure: (**method=** `method method`) => #t/#f

Returns #t if the two methods have identical signatures, i.e. have equal parameter types (as determined by `types=`) and rest parameter flag. #f is returned otherwise.

```
(method= (method ((next: next)(<number> x)(<number> y) . rest) (next x y))
         (method ((<number> a)(<number> b) . rest) (+ x y)))
  ;=> #t
(method= (method ((<number> x)(<number> y) . rest) (+ x y))
         (method ((<number> a)(<number> b)) (+ x y)))
  ;=> #f
(method= (method ((<number> x)(<value> y)) (+ x y))
         (method ((<number> a)(<number> b)) (+ x y)))
  ;=> #f
```

7.2.5. Miscellaneous

7.2.5.1. Generic Procedure Combination

Generic procedure combination merges the method lists of multiple generic procedures. The typical scenario for using this features is when several modules have defined generic procedure (and procedures using these generic procedures) that perform identical operations but on different data types. Generic procedure combination extends to coverage of the individual generic procedures and the dependent procedures to the combined set of data types. Furthermore, the coverage of the dependent procedures is implicitly extended to the combined set of data types.

The following example illustrates how generic procedure combination can be used to combine the functionality of two `p-append` procedures defined independently by two modules. It also shows how generic procedure combination implicitly extends the coverage of the `p-reverse-append` and `p-repeat` procedures defined by the modules.

```
(import* misc compose)
(module foo
    (p-append p-reverse-append)
  (define (p-reverse-append . args)
    (apply p-append (reverse args)))
  (define-generic p-append)
  (define-methods p-append
    [((<list> x) . rest)
     (apply append x rest)]
    [((<vector> x) . rest)
     (list->vector (apply append
                          (vector->list x)
                          (map vector->list rest)))]))
(module bar
    (p-append p-repeat)
  (define (p-repeat n x)
    (let loop ([res '()]
               [n   n])
      (if (= n 0)
          (apply p-append res)
          (loop (cons x res) (- n 1)))))
  (define-generic p-append)
  (define-methods p-append
    [((<string> x) . rest)
     (apply string-append x rest)]
    [((<symbol> x) . rest)
     (string->symbol (apply string-append
                            (symbol->string x)
                            (map symbol->string rest)))]))

(import* foo (p-append1 p-append) p-reverse-append)
(import* bar (p-append2 p-append) p-repeat)
(define-generic p-append p-append1 p-append2)
(define-method (p-append (<procedure> x) . rest)
  (apply compose x rest))
```

```
(p-append '(a b))  ;=> '(a b)
(p-append '(a b) '(c d) '(e f))  ;=> '(a b c d e f)
(p-append '#(a b))  ;=> '#(a b)
(p-append '#(a b) '#(c d) '#(e f))  ;=> '#(a b c d e f)
(p-append "ab")  ;=> "ab"
(p-append "ab" "cd" "ef")  ;=> "abcdef"
(p-append 'ab)  ;=> 'ab
(p-append 'ab 'cd 'ef)  ;=> 'abcdef
((p-append car cdr cdr cdr) '(1 2 3 4))  ;=> 4

(p-reverse-append "ab" "cd" "ef")  ;=> "efcdab"
(p-repeat 3 '(a b))  ;=> '(a b a b a b)
((p-reverse-append cdr cdr cdr car) '(1 2 3 4))  ;=> 4
((p-repeat 3 cdr) '(1 2 3 4))  ;=> (4)
```

7.2.5.2. Scoping Rules

Generic procedures are lexically scoped, but their methods are not. Hence defining methods in a local scope is generally a bad idea. One exception are module definitions. It is perfectly safe for modules to define private (i.e. not exported) generic procedures and add methods to them without interfering with other modules. However, care must be taken when generic procedures are imported or exported - methods are added to generic procedures when the module gets *defined* rather then when it gets imported.

The following example illustrates the scoping rules.

```
(define-generic m)
(define-method (m (<value> v)) v)
(m 1)  ;=> 1
(let ([x 1])
  (define-method (m (<number> v)) (+ x v))
  (m 1))  ;=> 2
(m 1)  ;=> 2

(module foo
    (m)
  (define-generic m)
  (define-method (m (<value> v)) v))
(import foo)
(m 1)  ;=> 1
(module bar
    ()
  (import foo)
  (define-method (m (<number> v)) (+ 1 v)))
(m 1)  ;=> 2
```

7.2.5.3. Method Selection

When generic procedures are invoked they select the most specific applicable method and call it, with the remaining applicable methods being made available to the invoked method via the *`next:`*.

Method applicability is determined on the basis of the types of the parameters passed in the invocation of the generic procedure. A method is applicable to a list of parameter types if and only if the following conditions are met:

- If the method accepts rest arguments then the length of the list of parameter types must be equal or greater than the method arity (as returned by `method-arity`).
- If the method does not accepts rest arguments then the length of the list of parameter types must be equal to the method arity (as returned by `method-arity`).
- All the types in the method's type signature (as returned by `method-types`) must be super-types of the corresponding parameter types. This comparison is performed using the `types<=` procedure.

This algorithm is encapsulated by the `method-applicable?` procedure.

Method specificity is an ordering relation on applicable methods with respect to a specific list of parameter types. Informally, the relative specificity of two methods is determined by performing a left-to-right comparison of the type signatures of the two methods and the parameter types using `compare-types`, returning the result of the type comparison at the point of the first discernable difference.

More formally, the relative specificity of two applicable methods is computed by a triple-wise comparison on successive elements of the method signatures (as returned by `method-types`) and actual parameter types, using `compare-types`, such that

- If we run out of elements in both method signatures then
  - If both or neither method return rest arguments (as determined by `method-rest?`) then the methods are of equal specificity.
  - If the first method takes rest arguments (as determined by `method-rest?`) then the first method is less specific than the second.
  - If the second method takes rest arguments (as determined by `method-rest?`) then the first method is more specific than the second.
- If we run out of elements in the first method's signature only then the first method is less specific than the second.
- If we run out of elements in the second method's signature only then the first method is more specific than the second.
- If `compare-types` returns `'equal` we proceed to the next triple.
- If `compare-types` returns `'less-specific` then the first method is less specific than the second.
- If `compare-types` returns `'more-specific` then the first method is more specific than the second.

This algorithm is encapsulated by the `compare-methods` procedure.

The `method` form and derived forms (i.e. `define-method` and `define-methods`) permit the specification of a special first parameter to the method invocation. When a generic procedure invokes a method, this parameter is bound to a procedure that when called will invoke the "next best matching" method. This is the next method in the list of applicable methods returned by `applicable-methods` when it was called by the generic procedure upon invocation.

If no "next best matching" method exists, i.e. the current method is the last in the list, then the next parameter is #f. This allows methods to invoke the next best matching method selectively depending on whether it is present. This is an important feature since the dynamic nature of method selection makes it impossible to determine at the time of writing the method whether there is going to be a next best matching method.

The next best matching method must be invoked with arguments to which the current method is applicable.

The following example illustrates the method selection algorithm, and use of the `next:` parameter:

```
(define-generic m)
(define-methods m
  [((next: next) (<number> x) (<value> y) (<number> z) . rest)
   (cons 'a (if next (apply next x y z rest) '()))]
  [((next: next) (<number> x) (<value> y) (<number> z))
   (cons 'b (if next (next x y z) '()))]
  [((next: next) (<number> x) (<number> y) . rest)
   (cons 'c (if next (apply next x y rest) '()))]
  [((next: next) (<number> x) (<number> y) (<value> z))
   (cons 'd (if next (next x y z) '()))])
(m 1 1 1)  ;=> '(d c b a)
(m 1 1)  ;=> '(c)
(m 1 'x 2)  ;=> '(b a)
(m 1 1 'x)  ;=> '(d c)
(m 1 'x 'x)  ;=> error
```

7.3. Object System

Requires: **(import *oo*)**

Programming in the *SISC* object system usually entailse the definition of generic procedures, so typically one also has to **(import *generic-procedures*)** .

The key features of the object system are:

- class-based, with a restricted form of multiple inheritance
- instance variables (aka *slots*) are accessed and modified via generic procedures
- generic procedures implement all behaviour; there is no separate notion of *methods*
- introspection API
- complete integration into *SISC*'s extensible type system

The examples in this section follow a few naming conventions:

Classes are types in the *SISC* type system and therefore class names follow the naming convention for type names (see
Generic procedures whose sole purpose it is to access slots of objects have names starting with `:` and otherwise follow
Generic procedures whose sole purpose it is to modify slots of objects, are named after the corresponding accessor pro

7.3.1. Classes

Classes have a name, a list of direct superclasses, and a list of direct slot descriptions. All classes are instances of the type <class> and are themselves types in *SISC*'s extensible type system. All classes are direct or indirect sub-classes of the class <object>, except for <object> itself, which has no super-classes.

Classes are created as follows:

procedure: (**`make-class`** `symbol class-list slot-list [guid-symbol]`) => class

Creates a class named *`symbol`* with the classes in *`class-list`* as its direct super-classes. See Section 7.3.4 for restrictions on super-classes. When no super-classes are specified, the superclass is <object>.

`slot-list` is a list of slot names (symbols).

Slots are inherited by sub-classes. For details on slot inheritance see Section 7.3.4.

If *`guid-symbol`* is specified then the new class is *non-generative*: if *`guid-symbol`* is already bound to a class then that class is modified, instead of a new class being created. Non-generative classes are serialised specially such that deserialising them also performs this check. By contrast, deserialisation of ordinary, generative classes and their instances results in duplicate types being created, which is usually not desirable.

```
(define-generics :x :y :y!)
(define <foo> (make-class '<foo> '() '()))
(define <bar> (make-class '<bar> '() '()))
(define <baz> (make-class '<baz> (list <foo> <bar>)
                          '(x y)))
```

syntax: (**`define-class`** `name-and-supers slot-def ...`) => void
where *`name-and-supers`* is of the form `(`*`class-name super-class`*` ...)`
and *`slot-def`* is of the form `(` *`slot-name`* [*`accessor`* [*`modifier`*]] `)`

Binds *`class-name`* to a newly created class.

This form expands into a definition with call to `make-class` on the right hand side.

`slot-name` names a slot. *`accessor`* must be a generic procedure. An accessor method for the slot will be added to it. *`modifier`* must be a generic procedure. A modifier method for the slot will be added to it.

```
(define-generics :x :y :y!)
(define-class (<foo>))
(define-class (<bar>))
(define-class (<baz> <foo> <bar>)
  (x :x)
  (y :y :y!))
```

syntax: (**`define-nongenerative-class`** `name-and-supers guid slot-def ...`) => void

This is the same as `define-class`, except that the resulting class is *non-generative* with *`guid`*, a symbol, as the unique identifier. The significance of this is explained in `make-class`.

One can test whether a particular value is a class:

procedure: (**`class?`** `value`) => #t/#f

Returns #t if *value* is a class, #f otherwise.

```
(class? (make-class '<foo> '() '()))  ;=> #t
(class? (lambda (x) x))  ;=> #f
(define-class (<foo>))
(class? <foo>)  ;=> #t
```

The following procedures provide access to the various elements of the <class> abstract data type:

procedure: (**class-name** class) => symbol

Returns the name of the class *class*.

```
(class-name (make-class '<foo> '() '()))  ;=> '<foo>
(define-class (<foo>))
(class-name <foo>)  ;=> '<foo>
```

procedure: (**class-direct-superclasses** class) => class-list

Returns the list of direct super-classes of the class *class*.

```
(define-class (<foo>))
(define-class (<bar>))
(define-class (<baz> <foo> <bar>))
(map class-name (class-direct-super-classes <foo>)  ;=> '())
(map class-name (class-direct-super-classes <baz>)  ;=> '(<foo> <bar>))
```

procedure: (**class-direct-slots** class) => slot-list

Returns the list of descriptions of the direct slots of the class *class*.

Slot descriptions are created by `make-class` for each slot definitions. The procedures operating on slot descriptions are documented in Section 7.3.2.

```
(class-direct-slots (make-class '<foo> '() '()))  ;=> '()
(define-generics :x :y :y!)
(define-class (<baz>)
  (x :x)
  (y :y :y!))
(class-direct-slots <baz> ;=> (<slot> <slot>))
```

7.3.2. Slots

Slot descriptions are instances of the abstract data type <slot>. They are created implicitly when classes are created; it is not possible to create them directly.

procedure: (**slot?** value) => #t/#f

Returns #t if *value* is a slot description, #f otherwise.

```
(define-class (<baz>) (x) (y))
(map slot? (class-direct-slots <baz>))  ;=> '(#t #t)
```

procedure: (**`slot-name`** `slot`) => symbol

Returns the name of the slot described by *`slot`*. This is the name given to the slot when its class was created.

```
(define-class (<baz>) (x) (y))
(map slot-name (class-direct-slots <baz>))  ;=> '(x y)
```

procedure: (**`slot-class`** `slot`) => class

Returns the class to which the slot description *`slot`* belongs.

```
(define-class (<baz>) (x) (y))
(eq? (slot-class (car (class-direct-slots <baz>))) <baz>)  ;=> #t
```

Slot definitions can produce procedures that allow access and modification of slots. Since slots can be defined without accessors and modifiers, this may be the only way to access/modify a particular slot.

procedure: (**`slot-accessor`** `slot`) => procedure

Returns a procedure than when applied to an instance of *`slot`*'s class, will return the slot's value on that instance.

```
(define-class (<baz>) (x))
(define baz (make <baz>))
((slot-accessor (car (class-direct-slots <baz>))) baz) ;=> 1
```

procedure: (**`slot-modifier`** `slot`) => procedure

Returns a procedure than when applied to an instance of *`slot`*'s class and a value, will set the slot's value on that instance to the supplied value.

```
(define-generics :x :x!)
(define-class (<baz>) (x :x))
(define baz (make <baz>))
(:x baz)  ;=> 1
((slot-modifier (car (class-direct-slots <baz>))) baz 2)
(:x baz)  ;=> 2
```

In addition to *procedures* for slot access and modification, slot definitions can also produce equivalent *methods*. These are suitable for adding to generic procedures and can, for instance, be used to achieve the same effect as specifying generic procedures for slot access/modification at class creation time.

procedure: (**`slot-accessor-method`** `slot`) => method

This is the same as `slot-accessor`, except it returns a method instead of a procedure.

procedure: (**`slot-modifier-method`** `slot`) => method

This is the same as `slot-modifier`, except it returns a method instead of a procedure.

7.3.3. Instantiation

Class instantiation is a two-stage process. First a new object is created whose type is that of the instantiated class. Then the `initialize` generic procedure is called with the newly created instance and additional arguments. `initialize` serves the same purpose as constructors in other object systems.

All the phases of class instantiation are carried out by a single procedure:

procedure: (**`make`** `class value ...`) => instance

Creates a new object that is an instance of *`class`*. The instance and *`value`*s are then passed as parameters to a call to the `initialize` generic procedure. The result of calling `make` is the instance.

```
(define-generics :x :x!)
(define-class (<baz>)
  (x :x :x!))
(define-method (initialize (<baz> b) (<number> x)) (:x! b x))
(define baz (make <baz> 2))
(:x baz)  ;=> 2
```

By default the `initialize` contains a no-op method for initialising objects of type <object> with no further arguments. Since all objects are instances of <object> all classes can be instantiated by calling `(make` *`class`*`)`.

```
(define-generics :x :x!)
(define-class (<baz>)
  (x :x :x!))
(define baz (make <baz>))
(:x baz)  ;=> #f
```

7.3.4. Inheritance

Inheritance is a form of sub-typing. A class is a sub-type of all its direct and indirect superclasses. Method selection in generic procedures (and hence also slot access/modification) is based on a relationship called *class precedence*; a total order of classes based on the partial orders established by the direct super-classes:

procedure: (**`class-precedence-list`** `class`) => class-list

Returns the total order of *`class`* and all direct and indirect super-classes, as determined by the partial orders obtained from calling `class-direct-superclasses`.

The ordering of classes returned by `class-direct-superclasses` is considered "weak" whereas the ordering of the class itself to its direct super-classes is considered strong. The significance of this is that when computing the class precedence list weak orderings are re-arranged if that is the only way to obtain a total order. By contrast, strong orderings are never rearranged.

```
(define-class (<foo>))
(define-class (<bar> <foo>))
(define-class (<baz> <foo>))
(define-class (<boo> <bar> <baz>))
(define-class (<goo>))
```

```
(define-class (<moo1> <bar> <baz> <goo>))
(map class-name (class-precedence-list <moo1>))
  ;=> (<moo1> <bar> <baz> <goo> <foo> <object>)
(define-class (<moo2> <bar> <baz> <foo> <goo>))
(map class-name (class-precedence-list <moo2>))
  ;=> (<moo1> <bar> <baz> <foo> <goo> <object>)
(define-class (<moo3> <baz> <bar> <boo>))
(map class-name (class-precedence-list <moo3>))
  ;=> (<moo3> <boo> <baz> <bar> <foo> <object>)
```

For any two classes in the class precedence list that have direct slots, one must be a sub-class of the other.

In effect this enforces single inheritance of slots while still giving control over the order of classes in the class-precedence list.

Slots from a superclass are only ever inherited once, regardless of the number of paths in the inheritance graph to that class.

If a sub-class defines a slot with the same name as one of its super-classes, instances of the resulting class ends up with *two* slots of the same name. If the slots are define with different accessors and modifiers then this does not cause any problems at all. If they are not then an invocation of the accessor/modifier will acess/modify the slot of the sub-class. Note that it is still possible to access/modify the slot of the superclass by using procedures/methods obtained from the slot descriptor. See Section 7.3.2.

The following example illustrates the rules governing slot inheritance.

```
;;ordinary slot access and modification
(define-generics
  :x :x! :y :y! :z :z!
  :xb :xb! :yb :yb! :zb :zb!)
(define-class (<foo>)
  (x :x :x!)
  (y :y :y!)
  (z :z))
(define f (make <foo>))
(:x f)  ;=> #f
(:y f)  ;=> #f
(:z f)  ;=> #f
(:x! f 2)
(:y! f 2)
(:x f)  ;=> 2
(:y f)  ;=> 2

;;overloading slots in sub-class
(define-class (<bar> <foo>)
  (x :x :x!)
  (y :yb :yb!)
  (zb :zb :zb!))
(define b (make <bar>))
(:x b)  ;=> #f
(:y b)  ;=> #f
```

```
(:z b)  ;=> #f
(:yb b)  ;=> #f
(:zb b)  ;=> #f
(:y! b 3)
(:yb! b 4)
(:y b)  ;=> 3
(:yb b)  ;=> 4

;;accessing a fully shadowed slot
(define :foo-x
  (cdr (assq 'x (map (lambda (x)
                       (cons (slot-name x) (slot-accessor x)))
                     (class-direct-slots <foo>)))))
(:foo-x b)  ;=> 1

;;inheritance restriction on slots
(define-class (<boo> <baz>)
  (yb :yb :yb!))
(define-class (<goo>))
(define-class (<moo> <bar> <boo> <goo>))  ;=> error
```

Chapter 8. Java Interaction

SISC can be used as a scripting language for Java, or Java may be used to provide functionality to Scheme. Such activity is collectively termed 'bridging'. In *SISC* bridging is accomplished by a Java API for executing Scheme code and evaluating Scheme expressions, and a module that provides Scheme-level access to Java objects and implementation of Java interfaces in Scheme.

8.1. Calling Scheme from Java

Calling programmer-defined Scheme code from Java is a simple process, but requires some initialization before proceeding. Depending on the use case of *SISC*, the initialization may be largely automatic. Any Scheme call first requires four resources:

1. An Application Context (`AppContext`), which encapsulates the entirety of a single application's interactions with *SISC*. This is initialized with ...
2. A Heap, which provides the environments and base code to execute programs using ...
3. A Dynamic Environment, which contains thread specific values such as the current I/O ports, and
4. An `Interpreter`, which provides the engine and API for actually evaluating Scheme code.

Each resource is described in the sections below. The gateway for interacting with these resources is primarily the `sisc.interpreter.Context` utility class.

8.1.1. Application Contexts

The Application Context, represented by the class `sisc.interpreter.AppContext`, holds references to the top-level environment, global configuration properties, and other resources which are unique to a single usage of *SISC*. This should not be confused with multiple threads or invocations into the same application.

An AppContext is created simply using the default constructor, or a constructor which takes a Java `Properties` class with overrides for global Scheme properties as described throughout this manual.

The application context is the token used by most of the API for calling Scheme from Java code, though many of the API calls can either infer the `AppContext` from the currently executing thread, or defer to a default context.

A default application context may be necessary when Scheme code is called through callbacks or other Java mechanisms in code which is not aware of *SISC* or the Scheme code which is being called. The default application context can be set explicitly after creating the context, but is also set implicitly by *SISC* if there is not already a default context when one is needed. To set the default application context, use the following method from `Context`:

public void **setDefaultAppContext**(sisc.interpreter.AppContext ctx);

Set the default `AppContext` to the instance provided.

public sisc.interpreter.AppContext **getDefaultAppContext**();

Retrieves the current default `AppContext`, possibly creating and initializing one using the default heap if no default was set.

Remember that an application context is nearly always useless until initialized with a heap, as described in the next section.

8.1.2. The *SISC* Heap

The heap is a serialization of all the pre-compiled code which makes up both the base Scheme language provided by *SISC*, and many of its libraries. Distributions of *SISC* come with a prepackaged heap which is sufficient for most usages, and customizing a heap should be viewed as a last resort, as precompiled libraries usually solve the same problem.

A heap is a randomly accessed structure which must be loaded into an application context. The heap can be automatically located if it is in the directory pointed to by the `SISC_HOME` environment variable, the current directory, or on the Java classpath. In the last case however, it will be loaded into memory entirely, rather than read in as needed. The automatic heap location involves calling `addDefaultHeap` on the `AppContext`, and is used when an application context is created implicitly.

public void **addDefaultHeap**();

Uses `findHeap`, `openHeap`, and `addHeap` to find and register a default heap with this application context

The URL that is produced through this discovery can be obtained using the `findHeap` method in `AppContext`:

public static java.net.URL **findHeap**(String heapName);

Searches for the heap with the given name, or if null, `sisc.shp` in several candidate locations. If a heap is found, a URL pointing to it is returned, otherwise null is returned.

One of the locations searched is the classpath, by searching for a *heap anchor*, and loading the heap file from the same package. To do this, the heap must be on the classpath in the same location as `HeapAnchor.class`. A utility script, `build-heapjar.scm`, executed as an SRFI-22 script, is included in the `scheme-src` directory of the full distribution. It is invoked from the directory containing the heap, and will by default emit `sisc-heap.jar` into the current directory. If that jar file is on the classpath, `findHeap` will locate it automatically.

Once located, the heap is opened and registered with the `AppContext`, allowing the context to then be used for Scheme evaluation. This is done with the `openHeap` and `addHeap` methods:

public static sisc.ser.SeekableInputStream **openHeap**(java.net.URL heapURL);

Opens the heap pointed to by the given URL, returning an appropriate random access input stream suitable for passing to `addHeap`.

public boolean **addHeap**(sisc.ser.SeekableInputStream heap);

Registers the given heap stream with the application context. Returns true if the registration is successful, false otherwise.

8.1.3. The Dynamic Environment

The dynamic environment, represented by the `sisc.env.DynamicEnvironment` class, is the datastructure which stores dynamic variables. That is, variables whose values are not scoped lexically and are associated with threads rather than code. This includes such values as the current input and output ports, Scheme parameters, and the class path.

Each interpreter contains a reference to its current dynamic environment, and the dynamic environment cannot be shared between two threads, or by more than one application context called in the same thread. They should be shared across call boundaries in a single thread and single application, but must be created anew for external calls and cross application calls, and cloned for new threads.

It should seem obvious, then, that maintaining the correct dynamic environment in all call situations can be tricky. Fortunately, the supported API calls in `Context` detect and do the Right Thing in most situations.

8.1.4. The Interpreter

The `Interpreter` class contains the engine for evaluating Scheme code, and API methods for triggering that evaluation. Each thread of execution must have its own `Interpreter` instance. Interpreters are obtained before one or more calls into Scheme, and using methods on the `Context` helper class depending on whether the call is an *Internal* or *External* call.

A Scheme application can execute in multiple threads. Each thread must have its own dynamic environment, containing entities such as the current input and output ports. Dynamic environments are represented by instances of the `sisc.env.DynamicEnvironment` class.

Internal calls need to create fresh interpreters in order to preserve and subsequently restore the state of the surrounding Scheme execution. Thus a single thread may be home to several interpreters.

Warning

In *any* case, the programmer must ensure that all calls to an interpreter are made by the same thread. If another thread wishes to execute Scheme code, it must follow the API below to obtain a different interpreter.

8.1.4.1. External Calls

An external call is a call made from Java to Scheme with no preceding call from Scheme to Java, in other words, the call is entirely external to the Scheme environment. For example, this may occur as a result of a timer expiration or a thread created by Java. In this case, the application context must be specified, and a new dynamic environment (containing thread specific information such as the current input and output ports) must be created. The preferred method for an external call is called a *managed external call*, and uses the visitor pattern with one of the following methods in the `Context` class:

public static Object **execute**(sisc.interpreter.AppContext ctx, sisc.interpreter.SchemeCaller caller) throws sisc.interpreter.SchemeException;

Creates an Interpreter context for the application context if provided, or the default application context if `ctx` is null, and calls `execute` in the caller with the new Interpreter. When the caller returns, the Interpreter is freed, and the return value of the caller is returned.

public static Object **execute**(sisc.interpreter.SchemeCaller caller) throws sisc.interpreterSchemeException;

Creates an Interpreter context for the default application context, loading the default heap if necessary, and calls the `execute` method of the `caller` object with the new Interpreter (that is, it is equivalent to `execute(null, caller)`). This simple interface is the one you should use if you do not have a particular reason to use a non-default application context.

The visitor instances implement `sisc.interpreter.SchemeCaller` and are responsible for implementing the following method:

public Object **execute**(sisc.interpreter.Interpreter r) throws sisc.interpreter.SchemeException;

Utilizes the given Interpreter to make Java to Scheme calls. Any Object may be returned.

As an alternative to this managed external call, a call from Java to Scheme can be made by using the `enter` method of `Context` to obtain an interpreter, then making several calls to the interpreter, then releasing it using `exit`. This has some weaknesses, however. Because this pair of calls cannot enforce that all intervening calls to the Interpreter run in the same thread, subtle issues can arrise if the Interpreter context is saved as a reference in a program and used by competing threads. Other, more subtle issues exist, such as the association of a thread handle (as retrievable using SRFI-18) in Scheme with the Java thread which is making the call. For this reason, the managed external call form is preferred whenever possible.

public sisc.interpreter.Interpreter **enter**(sisc.interpreter.AppContext ctx, sisc.env.DynamicEnvironment ctx);

Obtains an instance of Interpreter for the provided application context if provided, the current or default otherwise. If provided, the given dynamic environment is used rather than the environment selected automatically for the type of call.

The dynamic environment optional argument in the `enter` method may be specified if one wants to use a different mechanism for finding applications and dynamic environments. For instance, threads created from Scheme should probably execute within the application that created them and using a dynamic environment that is cloned from the dynamic environment present when the thread is started. The `enter` method can therefore be used as general mechanism for obtaining a new interpreter that uses a specific application and dynamic environment.

When the Interpreter is no longer by the current thread needed it *must* be released using:

public void **exit**();

Release the resources of the current interpreter.

8.1.4.2. Internal Calls

Internal calls are calls to Scheme made from Java, from a previous call into Java from Scheme. In other words, the call to Scheme is made internally from other Scheme code. One can determine if a call is internal in the following manner:

```
Interpreter current = Context.currentInterpreter();
if (current == null) { ...make external call...}
else { ...make internal call... }
```

This case is more complex, as it requires maintaining the correct dynamic environment and Interpreter instance to preserve return context. When making an internal call, one typically wants to make the call in an interpreter that shares the same application context and dynamic enviornment as the calling interpreter.

Fortunately, the details are managed by the `Context` helper class. The same calling mechanisms are used as in an external call, but the application context is ommited as a parameter to the functions.

8.1.4.3. The Interpreter API

No matter which mechanism is used, the `Interpreter` is eventually used to call Scheme code, using any of the following methods:

- public Value **eval**(String expr) throws sisc.interpreter.SchemeException, java.io.IOException;

  Evaluates expressions read from a string, returning the result of the evaluation of the last expression.
- public Value **eval**(InputStream stream, String sourceId) throws sisc.interpreter.SchemeException, java.io.IOException;

  Evaluates expressions read from an input stream, returning the result of the evaluation of the last expression. The *`sourceId`* identifies the source of the stream for display purposes.
- public Value **eval**(Value val) throws sisc.interpreter.SchemeException;

  This is the same as calling `(eval` *`val`*`)` in Scheme.
- public Value **eval**(Procedure proc, Value[] args) throws sisc.interpreter.SchemeException;

  This is the same as calling `(`*`proc arg`* `...)` in Scheme. This is the most efficient type of call, as it requires no parsing or compilation.

Several such calls can be made consecutively on the same interpreter. A `SchemeException` may be thrown by any of these methods if an error occurs in the Scheme code.

8.1.5. Miscellaneous Features

8.1.5.1. Error Handling

`Interpreter.eval()` throws a `sisc.interpreter.SchemeException` when an evaluation causes an exception that is not caught inside the evaluation. When making internal calls the exception can be propagated to the calling interpreter in one of four ways:

- by throwing a `RuntimeException` - this will be reported as`"Error in "`*`prim-name`*`:` *`description`*`"`.
- by calling `Module.throwPrimException("`*`description`*`")` - this will be reported as `"Error in` *`prim-name`*`:` *`description`*`"`.
- by calling `throwNestedPrimException("`*`description`*`",` *`schemeException`*`)` - this will be reported as `"Error in` *`prim-name`*`:` *`description`*`\n`*`nested-description`*`"`.
- by calling `throwNestedPrimException(`*`schemeException`*`)` - this will be reported as `"Error in` *`prim-name`*`: exception during nested call\n`*`nested-description`*`"`.

Scheme code can throw Java exceptions (see Section 8.2.9). The `sisc.modules.s2j.Util` class contains a static method `javaException(`*`schemeException`*`)` that extracts the Java exception from a `SchemeException`, or, if no Java exception is present, returns the `SchemeException`.

8.1.5.2. Continuations

Continuations do not cross the Scheme/Java boundary. In the embedded call scenario invoking a continuation inside the embedded call will not discard the computation of the caller. The embedded call will return when the continuation returns. If the continuation contains the read-eval-print loop the return will never happen. Similarly, capturing a continuation inside a call (embedded or external) will only capture the continuation to point where the call was made.

Capturing and invoking a continuation within the *same* call works correctly.

8.1.5.3. Scheme I/O

Scheme ports are thin wrappers around the Java I/O hierarchy, adding some functionality needed by Scheme. As such, it is trivial to obtain Java compatible I/O objects from Scheme ports. For information on obtaining Scheme and Java compatible I/O objects in Scheme, see the Java I/O module described in Section 5.1.6.

Scheme ports are encapsulated in the `SchemeBinaryInputPort`, `SchemeBinaryOutputPort`, `SchemeCharacterInputPort`, `SchemeCharacterOutputPort` classes in the `sisc.data` package. An instance of a port class contains an accessor which returns the relevant Java I/O type, as described for each class below.

```
public class sisc.data.SchemeBinaryInputPort {
          public java.io.InputStream getInputStream();

}
```

Return a Java `InputStream` for accessing this Scheme port.

```
public class sisc.data.SchemeBinaryOutputPort {
          public java.io.OutputStream getOutputStream();

}
```

Return a Java `OutputStream` for accessing this Scheme port.

```
public class sisc.data.SchemeCharacterInputPort {
          public java.io.Reader getReader();

}
```

Return a Java `Reader` for accessing this Scheme port.

```
public class sisc.data.SchemeCharacterOutputPort {
          public java.io.Writer getWriter();

}
```

Return a Java `Writer` for accessing this Scheme port.

8.1.6. Quick Reference

The tables below cover the most common use cases for calling Scheme from Java, and provide simple pseudocode examples.

Table 8-1. Typical Java to Scheme, External Calls

Situation	Code
Default Application Context	
Heap can be located automatically...	*No action required.*

...or, with a custom heap	`AppContext ctx=new AppContext();`
	`SeekableInputStream myHeap=AppContext.openHeap(my`
	`ctx.addHeap(myHeap);`
	`Context.setDefaultAppContext(ctx);`

Then, making the external call.	`Context.execute(mySchemeCaller);`
Custom Application Context	
Creating the context.	`AppContext ctx=new AppContext(myProperties);`
Heap can be located automatically ...	`ctx.addDefaultHeap();`

... or, with a custom heap	`SeekableInputStream myHeap=AppContext.openHeap(my`
	`ctx.addHeap(myHeap);`

Then, making the external call.	`Context.execute(ctx, mySchemeCaller);`

Table 8-2. Typical Java to Scheme, Internal Calls

Situation	Code
Making the internal call.	`Context.execute(mySchemeCaller);`

8.2. Calling Java from Scheme

Requires: **(import *s2j*)**

The High-Level S2J API allows Scheme code to instantiate Java classes, call methods on Java objects, access/modify fields of Java objects and implement Java interfaces that delegate to Scheme code.

8.2.1. Classes

Java classes are types in *SISC*'s extensible type system (see Section 7.1). They are made accessible to Scheme code by one of the following procedures / special forms:

procedure: (**`java-class`** `symbol`) => jclass

Returns the Java class of name *`symbol`*, which can also be the name of a primitive type or an array type.

```
(java-class '|java.lang.String|)  ;=> <jclass>
(define <java.io.object-input-string/get-field**>
  (java-class '|java.io.ObjectInputStream$GetField[][]|))
```

syntax: (**`define-java-class`** `scheme-name [java-name]`) => void

Binds *`scheme-name`* to the Java class named by *`java-name`*, or, if no such parameter is supplied, by the mangled *`scheme-name`*.

```
(define-java-class <jstring> |java.lang.String|)
(define-java-class <java.io.object-input-string/get-field**>)
```

syntax: (**`define-java-classes`** `form ...`) => void
where *`form`* is of the form *`scheme-name`* or `(`*`scheme-name java-name`*` )`

Creates bindings for several Java classes.

The form expands into several `define-java-class` forms.

```
(define-java-classes
  (<jstring> |java.lang.String|)
  <java.io.object-input-string/get-field**>)
```

Mangling of class names allows classes to be identified more schemely, e.g. `<java.io.object-input-stream/get-field**>` corresponds to the Java type `java.io.ObjectInputStream.GetField[][]` (note that GetField is a nested class). More formally, mangling of class names checks for the presence of angle brackets (<>) around the *`scheme-name`*. If they are present they are stripped. All the identifiers between the dots (`.`) are passed through field name mangling (see Section 8.2.3). The character following the last dot is upcased. A slash (`/`) is treated as a nested class indicator; it is replaced with dollar (`$` and the character following it is upcased. Trailing stars (`*`) characters are replaced with pairs of brackets (`[]`).

There are predicates for determining whether a Scheme value is a Java class or interface. All Java interfaces are also Java classes.

procedure: (**`java-class?`** `value`) => #t/#f

Returns #t if *`value`* is a Java class, #f otherwise.

```
(java-class? (java-class '|java.lang.String|)  ;=> #t
(define <java-io-object-input-string/get-field**>
  (java-class '|java.io.ObjectInputStream$GetField[][]|))
(java-class? <java.io.object-input-string/get-field**>)  ;=> #t
```

procedure: (**`java-interface?`** `value`) => #t/#f

Returns #t if *`value`* is a Java interface, #f otherwise.

```
(java-interface? (java-class '|java.util.Map|))  ;=> #t
(java-interface? (java-class '|java.lang.String|)  ;=> #f
```

Java classes are serializable by the *SISC* runtime.

8.2.2. Methods

Java methods are made accessible to Scheme code as procedures that can invoke any method of a given name on any Java object. Method selection is performed based on the types of the arguments to the procedure call. Static Java methods can be invoked by passing an instance of the appropriate class or an appropriately typed null object (see Section 8.2.4) as the first argument to the procedures.

procedure: (**`generic-java-method`** `symbol`) => procedure

Returns a procedure that when invoked with a Java object as the first argument and Java values as the remaining arguments, invokes the best matching named method named *`symbol`* on the Java object and returns the result.

```
(generic-java-method '|getURL|)  ;=> <jmethod>
(define empty-list? (generic-java-method '|isEmptyList|))
```

syntax: (**`define-generic-java-method`** `scheme-name [java-name]`) => void

Binds *`scheme-name`* to the generic Java method named by *`java-name`*, or, if no such parameter is supplied, by the mangled *`scheme-name`*.

```
(define-generic-java-method get-url |getURL|)
(define-generic-java-method empty-list?)
```

syntax: (**`define-generic-java-methods`** `form ...`) => void
 where *`form`* is of the form *`scheme-name`* or `(`*`scheme-name java-name`* `)`

Creates bindings for several generic Java methods.

The form expands into several `define-generic-java-method` forms.

```
(define-generic-java-methods
  (get-url |getURL|)
  empty-list?)
```

Method name mangling allows methods to be identified more schemely, e.g. `empty-list?` corresponds to the Java method name `isEmptyList`. More formally, mangling of method names removes trailing exclamation marks (`!`) and replaces trailing question marks (`?`) with a leading `is-`. The result of this mangling is passed through field mangling (see Section 8.2.3).

Generic Java methods are serializable by the *SISC* runtime.

8.2.3. Fields

Fields are made accessible to Scheme code as procedures that can get / set any field of a given name on any Java object. If several fields of the same name are present in the object due to the object class's inheritance chain, the most specific field, i.e. the one bottommost in the inheritance hierarchy, is selected. Static Java fields can be accessed / modified by passing an instance of the appropriate class or an appropriately typed null object (see Section 8.2.4) as the first argument to the procedures.

Generic Java field accessors, i.e. procedures that allow Scheme code to obtain the value of a Java field, can be defined as follows:

procedure: (**`generic-java-field-accessor`** `symbol`) => procedure

Returns a procedure that when invoked with a Java object as the first argument, retrieves the value of the Java field named *`symbol`* in the Java object.

```
(generic-java-field-accessor '|currentURL|)  ;=> <jfield>
(define :current-input-port (generic-java-field-accessor '|currentInputPort|))
```

syntax: (**`define-generic-java-field-accessor`** `scheme-name [java-name]`) => void

Binds *`scheme-name`* to the generic Java field accessor for fields named *`java-name`*, or, if no such parameter is supplied, the mangled *`scheme-name`*.

```
(define-generic-java-field-accessor :current-url |currentURL|)
(define-generic-java-field-accessor :current-input-port)
```

syntax: (**`define-generic-java-field-accessors`** `form ...`) => void
where *`form`* is of the form *`scheme-name`* or `(`*`scheme-name java-name`* `)`

Creates bindings for several generic Java field accessors.
The form expands into several `define-generic-java-field-accessor` forms.

```
(define-generic-java-field-accessor
  (:current-url |currentURL|)
  :current-input-port)
```

Generic Java field modifiers, i.e. procedures that allow Scheme code to set the value of a Java field, can be defined as follows:

procedure: (**`generic-java-field-modifier`** `symbol`) => procedure

Returns a procedure that when invoked with a Java object as the first argument and a Java value as the second argument, sets the value of the Java field named *`symbol`* in the Java object to that value.

```
(generic-java-field-modifier '|currentURL|)  ;=> <jfield>
(define :current-input-port! (generic-java-field-modifier '|currentInputPort|))
```

syntax: (**`define-generic-java-field-modifier`** `scheme-name [java-name]`) => void

Binds *`scheme-name`* to the generic Java field modifier for fields named *`java-name`*, or, if no such parameter is supplied, the mangled *`scheme-name`*.

```
(define-generic-java-field-modifier :current-url! |currentURL|)
(define-generic-java-field-modifier :current-input-port!)
```

syntax: (**`define-generic-java-field-modifiers`** `form ...`) => void
  where *`form`* is of the form *`scheme-name`* or `(`*`scheme-name java-name`* `)`

Creates bindings for several generic Java field modifiers.
The form expands into several `define-generic-java-field-modifier` forms.

```
(define-generic-java-field-modifier
  (:current-url! |currentURL|)
  :current-input-port!)
```

The mangling of field names allows fields to be identified more schemely. By convention field accessors should be named with a leading colon (`:`) followed by the field name, and field modifiers with a leading colon (`:`) followed by the field name followed by an exclamation mark (`!`), e.g. `:foo-bar` and `:foo-bar!` are the names of the accessor and modifier for Java fields named `fooBar`. Mangling of field names upcases any character following a dash (`-`) and removes all characters that are not legal as part of Java identifiers.

Generic Java field accessors and modifiers are serializable by the *SISC* runtime.

8.2.4. Instances

Scheme code can instantiate Java classes with a call to the following procedure:

procedure: (**`java-new`** `jclass jobject ...`) => jobject

Selects a constructor of *`jclass`* based on the types of the *`jobject`*s and calls it, returning the newly created object.

```
(define-java-class <java.util.linked-hash-set>)
(java-new <java.util.linked-hash-set> (->jint 100))  ;=> <jobject>
```

There is a predicate for determining whether a value is a Java object:

procedure: (**`java-object?`** `value`) => #t/#f

Returns #t if *`value`* is a Java object, #f otherwise.
Note that, unlike in Java, instances of primitive Java types are considered to be Java objects.

```
(define-java-class <java.util.linked-hash-set>)
(define hs (java-new <java.util.linked-hash-set> (->jint 100)))
(java-object? hs)  ;=> #t
(java-object? (->jint 100))  ;=> #t
```

Unlike in Java, null objects are typed. Typed null objects play a key role in invoking static methods an accessing / modifying static fields.

procedure: (**`java-null`** `jclass`) => jnull

Returns a Java null object of type *`jclass`*.

```
(define-java-class <java.util.linked-hash-set>)
(java-null <java.util.linked-hash-set>)  ;=> <jnull>
```

There is a predicate for determining whether a value is a Java null. All nulls are also Java objects.

procedure: (**`java-null?`** `value`) => #t/#f

Returns #t if *`value`* is a Java null object, #f otherwise.

```
(define-java-class <java.util.linked-hash-set>)
(java-null? (java-null <java.util.linked-hash-set>))  ;=> #t
```

For convenience, `jnull` is bound to the typed null object obtained by calling `java-null` on `java.lang.Object`.

Any invocation of a Java method, or access to a Java fields that returns a Java null does so typed based on the declared return / field type.

Comparison of Java objects using `eqv?` compares the objects using Java's == comparison. `equal?`, on the other hand, compares the objects using Java's `equals` method. `eq?` uses pointer equality on the Scheme objects representing the Java objects and is therefore not generally useful. Applying `eq?`, `eqv?` or `equal?` to a mixture of Java objects and other Scheme values returns #f.

Java objects are only serializable by the *SISC* runtime if they support Java serialization. Java nulls are always serializable.

8.2.5. Arrays

Scheme code can create Java arrays with a call to the following procedure:

procedure: (**`java-array-new`** `jclass size`) => jarray

Creates an array of component type *`jclass`* with dimensions *`size`*, which can be a number (for a single-dimensional array), or a vector / list of numbers (for multi-dimensional arrays).

```
(java-array-new <jint> 2)  ;=> <jarray>
(define-java-class <java.lang.string>)
(java-array-new <java.lang.string> '#(2 2 2))  ;=> <jarray>
```

There is a predicate for determining whether a value is a Java array. All Java arrays are also Java objects.

procedure: (**`java-array?`** `value`) => #t/#f

Returns #t if `value` is a Java array, #f otherwise.

```
(java-array? (java-array-new <jint> 2))  ;=> #t
```

Elements of arrays are accessed and modified with:

procedure: (**`java-array-ref`** `jarray index`) => jobject

Returns the element at index `index` of `jarray`. `index` can be a number, for indexing into the first dimension of the array, or vector / list of numbers for multi-dimensional indexing.

```
(define a (->jarray (map ->jint (iota 10)) <jint>))
(java-array-ref a 1)  ;=> <java int 1>
(java-array-ref a '(1))  ;=> <java int 1>
```

procedure: (**`java-array-set!`** `jarray index jobject`) => void

Sets the element at index `index` of `jarray` to `jobject`. `index` can be a number, for indexing into the first dimension of the array, or vector / list of numbers for multi-dimensional indexing.

```
(define a (->jarray (map ->jint (iota 10)) <jint>))
(java-array-set! a 1 (->jint 2))
(java-array-ref a 1)  ;=> <java int 2>
(define a (java-array-new <jint> '#(2 2 2)))
(java-array-set! a '#(1 1 1) (->jint 1))
```

The length of a Java array can be determined with

procedure: (**`java-array-length`** `jarray`) => number

Returns the length of `jarray`.

```
(define a (->jarray (map ->jint (iota 10)) <jint>))
(java-array-length a)  ;=> 10
(define a (java-array-new <jint> '#(2 3 4)))
(java-array-length a)  ;=> 2
```

Scheme vectors and lists can be converted to Java array and vice versa.

procedure: (**`->list`** `jarray`) => list

Creates a list containing the elements of `jarray`.

```
(define a (->jarray (map ->jint (iota 5)) <jint>))
(map ->number (->list a))  ;=> '(0 1 2 3 4)
```

procedure: (**`->vector`** `jarray`) => vector

Creates a vector containing the elements of `jarray`.

```
(define a (->jarray (map ->jint (iota 5)) <jint>))
```

```
(map ->number (vector->list (->vector a)))  ;=> '(0 1 2 3 4)
```

procedure: (**->jarray** `list-or-vector jclass`) => jarray

Creates a one-dimensional array of type *`jclass`* and fills it with the values obtained from the Scheme vector or list.

```
(define a (->jarray (map ->jint (iota 5)) <jint>))
```

8.2.6. Proxies

Scheme code cannot create sub-classes of existing Java classes. It is, however, possible to create classes implementing existing Java interfaces. These classes are called proxies. Calling a method on a proxy invokes a user-definable Scheme procedure, based on the name of the method, passing the proxy object and the parameters of the method invocation as arguments. The result of the invocation is returned as the result of the method call.

syntax: (**define-java-proxy** `signature interfaces method ...`) => void

where *`signature`* is of the form `(`*`name param`*` ...)`,
`interfaces` is of the form `(`*`interface`*` ...)`, and
`method` is of the form `(define `*`method-name procedure`*`)`, or `(define ( `*`method-name method-arg`*` ...) . `*`body`*`)`

Creates a proxy generator procedure and binds it to *`name`*. A proxy class is created that implements all the *`interface`*s. When the generator is invoked, an instance of the proxy class is returned that delegates all method invocations to the Scheme procedures in the method definition list, based on the names of the methods.

The first kind of definition form defines *`procedure`* to be the method handler for the java method named *`method-name`*. *`method-name`* undergoes name mangling as described in Section 8.2.2. Note that *`procedure`* is inside the lexical scope of the generator procedure, so *`param`*s are accessible inside it.

The second kind of definition form is equivalent to the following first-type form: `(define `*`method-name`*` (lambda (`*`method-arg`*` ...) . `*`body`*`))`.

If a method is invoked on a proxy for which no method handler exists and error is returned to the caller.

```
(define-java-classes
  <java.util.comparator>
  <java.util.arrays>
  <java.lang.object>)
(define-java-proxy (comparator fn)
  (<java.util.comparator>)
  (define (.compare p obj1 obj2)
    (let ([x (java-unwrap obj1)]
          [y (java-unwrap obj2)])
      (->jint (cond [(fn x y) -1]
                    [(fn y x) +1]
                    [else 0])))))
(define-generic-java-method sort)
(define-java-class <java.lang.object>)
(define (list-sort fn l)
  (let ([a (->jarray (map java-wrap l) <java.lang.object>)])
    (sort (java-null <java.util.arrays>) a (comparator fn))
    (map java-unwrap (->list a))))
(list-sort < '(3 4 2 1))  ;=> '(1 2 3 4)
```

```
(list-sort string<? '("foo" "bar" "baz"))  ;=> '("bar" "baz" "foo")
```

8.2.7. Types and Conversions

For convenience, all the primitive Java types, i.e. `void`, `boolean`, `double`, `float`, `long`, `int`, `short`, `byte`, `char`, are predefined and bound to `<jvoid>`, `<jboolean>`, `<jdouble>`, `<jfloat>`, `<jlong>`, `<jint>`, `<jshort>`, `<jbyte>`, `<jchar>`, respectively.

When calling Java methods, invoking Java constructors, accessing or modifying Java fields, no automatic conversion is performed between ordinary Scheme values and Java values. Instead explicit conversion of arguments and results is required. Automatic conversion is not performed for the following reasons:

- For some Scheme types, such as numbers, the mapping to Java types is one-to-many, e.g. a Scheme number could be converted to a `byte`, `short`, `int`, etc. This causes ambiguities when automatic conversion of parameters is attempted.
- Some Java types have several corresponding Scheme types, e.g. a Java array could be represented as Scheme list or vector - this causes ambiguities when automatic conversion of results is attempted.
- Conversion carries an overhead that can be significant. For instance, Java strings have to be copied "by value" to Scheme strings since the former are immutable and the latter aren't. In a chained-call scenario, i.e. where the results of one method invocation are passed as arguments to another, the conversion is unnecessary and a wasted effort.
- Conversion breaks the object identity relationship. In a chained-call scenario, the identities of the objects passed to the second call are different from the ones returned by the first. This causes problems if the called Java code relies on the object identity being preserved.
- Conversion conflicts with generic procedures. The method selection mechanism employed by generic procedures relies on objects having exactly one type. Automatic conversion effectively gives objects more than one type - their original type and the type of the objects they can be converted to. While it would be technically possible to devise a method selection algorithm that accommodates this, the algorithm would impose a substantial overhead on generic procedure invocation and also make it significantly harder for users to predict which method will be selected when invoking a generic procedure with a particular set of arguments.

Conversion functions are provided for converting instances of primitive Java types to instances of standard Scheme types:

procedure: (**->boolean** `jboolean`) => #t/#f

procedure: (**->character** `jchar`) => character

procedure: (**->number** `jbyte/jshort/jint/jlong/jfloat/jdouble`) => number

Conversion functions also exists for the opposite direction, i.e. converting instances of standard Scheme types to instances of primitive Java types

procedure: (**->jboolean** `boolean`) => jboolean

procedure: (**->jchar** `character`) => jchar *procedure:* (**->jbyte** `number`) => jbyte

procedure: (**->jshort** `number`) => jshort

procedure: (**->jint** `number`) => jint

procedure: (**->jlong** `number`) => jlong

procedure: (**->jfloat** `number`) => jfloat

procedure: (**->jdouble** `number`) => jdouble

Finally, there are conversion functions for converting between Java strings and Scheme strings and symbols:

procedure: (**->string** `jstring`) => string

procedure: (**->symbol** `jstring`) => symbol

procedure: (**->jstring** `string/symbol`) => jstring

Scheme values are not Java objects and hence cannot be passed as arguments in Java method or constructor invocations or when setting Java fields. However, all Scheme values are *internally* represented by instances of classes in the SISC runtime. S2J provides a mechanism to get hold of this internal representation as an S2J Java object. The converse operation is also supported - a Java instance obtained via a Java method or constructor invocation or field access in S2J can be turned into a Scheme value if it is an instance of an appropriate *SISC* runtime class. These two operations are called "wrapping" and "unwrapping" respectively because conceptually the scheme object is wrapped to make it appear like a Java object and the wrapper is removed in order to recover the original Scheme object.

procedure: (**java-wrap** `value`) => jobject

Returns the Java object that represents the Scheme *`value`* in *SISC*'s runtime.

procedure: (**java-unwrap** `jobject`) => value

Returns the Scheme value represented by the *`jobject`*. If *`jobject`* is not an object representing a Scheme value in SISC's runtime and error is thrown.

```
(define-java-class <java.lang.object>)
(define a (java-array-new <java.lang.object> '#(1)))
(java-array-set! a '#(0) (java-wrap 'foo))
(java-unwrap (java-array-ref a '#(0))) ;=> 'foo
```

Wrapping and unwrapping allows Scheme values to be used in generic (i.e. not type-specific) Java operations, such as those of the Java collection API. It is also frequently used in connection with proxies when Scheme objects are passed back and forth through layers of Java to a Scheme-implemented proxy that manipulates them. Finally, wrapping and unwrapping permit *SISC* Scheme code to interface to the *SISC* runtime.

8.2.8. Multi-threading

In Java each object is a potential thread synchronization point. Therefore Scheme code needs to be able to synchronize on Java objects in order for it to interoperate properly with Java in a multi-threaded application. This is accomplished by the following procedure:

procedure: (**`java-synchronized`** `jobject thunk`) => value

Runs *`thunk`* in a block synchronized on *`jobject`*, returning the result returned by *`thunk`*. This is the equivalent to `synchronized (`*`jobject`*`) { return `*`thunk`*`(); }` in Java.

It is illegal for *`thunk`* to invoke continuations that escape *`thunk`*, or for code outside *`thunk`* to invoke a continuation captured inside *`thunk`*.

```
(define-java-class <java.lang.object>)
(define mtx (java-new <java.lang.object>))
(define v 0)
(define (inc-v)
  (java-synchronized mtx (lambda () (set! v (+ v 1)) v)))
(define (dec-v)
  (java-synchronized mtx (lambda () (set! v (- v 1)) v)))
(import threading)
(begin (parallel inc-v dec-v inc-v inc-v dec-v dec-v) v)  ;=> 0
```

8.2.9. Exception Handling

Java exceptions are propagated to scheme and can be caught like any other exception, e.g. with `with/fc` as defined in Section 3.4.1. The `s2j` module exports augmented versions of the `print-stack-trace` and `print-exception` functions that handle Java exceptions. For example

```
(define-generic-java-method char-at)
(with/fc (lambda (m e) (print-exception (make-exception m e)))
  (lambda () (char-at (->jstring "foo") (->jint 3))))
```

will catch the `IndexOutOfBoundsException`, print its stack trace and return #f.

In Scheme, Java exceptions can be thrown by raising an error containing the Java exception as the message, e.g.

```
(define-java-class <java.util.no-such-element-exception>)
(error (java-new <java.util.no-such-element-exception>))
```

or

```
(throw (make-error (java-new <java.util.no-such-element-exception>)))
```

If this occurs inside a proxy method (see Section 8.2.6), the exception is propagated to the invoking Java code.

8.2.10. Access Permissions

Invoking `[define-]java-class[es]`, `java-new` or any of the procedures defined with `[define-]generic-java-{method,field-accessor,field-modifier}[s]` causes S2J to perform reflection on the named Java class(es), the class passed as the first argument, or the class corresponding to the type first argument passed to the other procedures, respectively. This process collects information about all the constructors, methods and fields of the class and its superclasses/interfaces.

The only class members processed during this automatic reflection are public ones declared in public classes. This almost exactly mimics the visibility rules in Java for code residing in packages other than the one the member is residing in. It is also in line with the default permissions granted to the Java reflection API. There is one rare case where this rule is more restrictive than Java's: public members of package-protected classes are not visible even when accessed via a public sub-class.

Depending on the security settings, the Java reflection API is in fact capable of granting access to *any* members of *any* class. However, using this in the automatic reflection performed by S2J would constitute a significant departure from normal Java behaviour and result in unpredictable results to the user. For instance, undocumented private methods would be invoked in preference to documented public methods if the formers type signature provided a better match.

Automatic reflection ignores security exceptions thrown by the Java reflection API, i.e. the class in question will appear to have no constructors, methods and fields. This is designed to cope with situations where the default security settings have been altered in a way that prevents access to members of some (or even all) classes.

In some applications the reflection API permissions depend on the context of the invocation. For instance, in applets it is usually possible to access class member information as part of the initialisation but not after that. Since `[define-]java-class[es]` triggers automatic reflection, it can be used to control when automatic reflection for specific classes takes place.

8.2.11. Common Usage

This section provides a summary of all the commonly used S2J features, correlating them with the corresponding Java code. It makes use of some functions from the `srfi-1`, `srfi-26` and `misc` modules

```
(require-library 'sisc/libs/srfi)
(import* srfi-1 fold)
(import* srfi-26 cut cute)
(import* misc compose)
```

Table 8-3. Common S2J Usage

Java	Scheme
create bindings for classes, methods and fields	

`n/a`	`(define-java-classes <foo.bar-baz> <foo.bar-boo>)`
	`(define-generic-java-methods get-bar get-baz set-`
	`(define-generic-java-field-accessors :bar :baz)`
	`(define-generic-java-field-modifiers :bar! :baz!)`

instantiate class	

Java	Scheme
`foo.BarBaz fooObj = new foo.BarBaz(a, b, c);`	`(define foo-obj (java-new <foo.bar-baz> a b c))`
invoke method on instance	
`Object res = fooObj.barBaz(a, b, c)`	`(define res (bar-baz foo-obj a b c))`
invoke method on class	
`Object res = foo.Bar.baz(a, b, c)`	`(define res (baz (java-null <foo.bar>) a b c))`
access instance field	
`Object res = fooObj.bar;`	`(define res (:bar foo-obj))`
access class field	
`Object res = foo.Bar.baz;`	`(define res (:bar (java-null <foo.bar>)))`
modify instance field	
`fooObj.bar = val;`	`(:bar! foo-obj val)`
modify class field	
`foo.Bar.baz = val;`	`(:bar! (java-null <foo.bar>) val)`
chained field access	

`Object res = fooObj.bar.baz.boo`	`(define res (fold (cut <> <>) foo-obj (list :bar`
	 or
	 `(define res ((compose :boo :baz :bar) foo-obj))`
	 This works equally well for bean fields.

chained field modification	

`fooObj.bar.baz.boo = moo;`	`(:boo! (fold (cut <> <>) foo-obj (list :bar :baz)`
	 or
	 `(:boo! ((compose :baz :bar) foo-obj) moo)`
	 This works equally well for bean fields.

accessing several fields	

`a = fooObj.bar; b = fooObj.baz;`
 `c = fooObj.boo;`
	`(apply (lambda (a b c) ...)`
	 `(map (cute <> foo-obj) (list :bar :baz :boo)))`
	 This works equally well for bean fields.

modifying several fields	

Java	Scheme
`fooObj.bar = a; fooObj.baz = b; fooObj.boo = c;`	`(for-each (cute <> foo-obj <>) (list :bar! :baz! :boo!) (list a b c))` This works equally well for bean fields.
creating an array	
`int[][] ar = new int[2][2];`	`(define ar (java-array-new <jint> '(2 2)))` This works equally well for bean fields.
accessing an array element	
`int res = ar[1][1];`	`(define res (java-array-ref ar '(1 1)))`
modifying an array element	
`ar[1][1] = val;`	`(java-array-set! ar '(1 1) val)`
iterating over an array	
`for(int i=0; i<ar.length; i++) ar[i].fooBar(a,b);`	`(for-each (cute foo-bar <> a b) (->list ar))`
implementing interfaces	
`public class Foo implements Bar, Baz { private int x; private int y; public Foo(int x, int y) { this.x = x; this.y = y; } public int barMethod(int z) { return x+y+z; } public int bazMethod(int z) { return x+y-z; } } ... Foo fooObj = new Foo(1, 2);`	`(define-java-proxy (foo x y) (<bar> <baz>) (define (bar-method p z) (->jint (+ x y (->number z)))) (define (baz-method p z) (->jint (+ x y (- (->number z)))))) ... (define foo-obj (foo 1 2))`

8.3. Java Reflection Interface

Requires: **(import *s2j*)**

The S2J Reflection API lets Scheme code access all the core functions of the Java reflection API. It underpins the High Level S2J Interface (see Section 8.2). Normal interaction with Java from Scheme does not require knowledge of this API, just like normal use of Java does not require knowledge of the Java reflection API.

8.3.1. Classes

These functions access attributes and members of classes.

procedure: (**`java-class-name`** `jclass`) => symbol

Returns the name of *`jclass`*.

procedure: (**`java-class-flags`** `jclass`) => list of symbols

Returns the modifiers of *`jclass`*, for example `public static final`.

procedure: (**`java-class-declaring-class`** `jclass`) => jclass

Returns the Java class in which *`jclass`* was declared, or null if it was declared at the top level.

procedure: (**`java-class-declared-superclasses`** `jclass`) => list of jclass

Returns the direct superclasses of *`jclass`*.

Normally this is the class' superclass followed by all of its interfaces in the order they were specified in the class declaration. There are a number of exceptions which ensure that the result is consistent with the precedence order employed by Java for method lookup on overloaded method. Interfaces and classes that directly inherit from `java.lang.Object` are all given `java.lang.Object` as the *last* element in their superclass list. For primitive and array types the direct superclass or superclasses reflect the widening conversions performed by Java. For example, `<jint>`'s superclass is `<jlong>` and `<java.util.array-list[][]>` 's superclasses are:

- `<java.util.abstract-list[][]>`
- `<java.util.list[][]>`
- `<java.util.random-access[][]>`
- `<java.lang.cloneable[][]>`
- `<java.io.serializable[][]>`

Note that this behavior is different from the corresponding method in the Java reflection API.

procedure: (**`java-class-declared-classes`** `jclass`) => list of jclasses/#f

Returns all the classes declared by *`jclass`*, or `#f` if access to this information is prohibited.

procedure: (**`java-class-declared-constructors`** `jclass`) => list of jconstructor/#f

Returns all the constructors declared by *`jclass`*, or `#f` if access to this information is prohibited.

procedure: (**`java-class-declared-methods`** `jclass`) => list of jmethods/#f

Returns all the methods declared by *`jclass`*, or `#f` if access to this information is prohibited.

procedure: (**`java-class-declared-fields`** `jclass`) => list of jfields/#f

Returns all the fields declared by *`jclass`*, or `#f` if access to this information is prohibited.

procedure: (**`java-class-precedence-list`** `jclass`) => list of jclasses

Returns the total order of *`jclass`* and all direct and indirect superclasses, as determined by the partial orders obtained from calling `java-class-declared-superclasses`.

The class precedence list is important when comparing types using the type system's `compare-types` procedure, which is used by the generic procedure method selection algorithm (see `compare-methods` in Section 7.2.3). Since generic Java methods and field accessors/mutators are implemented in terms of generic procedures they are all affected by the class precedence list.

8.3.2. Constructors

procedure: (**`java-constructor?`** `value`) => #t/#f

Determines whether *`value`* is a Java constructor.

procedure: (**`java-constructor-name`** `jconstructor`) => symbol

Returns the name of *`jconstructor`*.

procedure: (**`java-constructor-flags`** `jconstructor`) => list of symbols

Returns the modifiers of *`jconstructor`* , such as `public static final`.

procedure: (**`java-constructor-declaring-class`** `jconstructor`) => jclass

Returns the Java class in which *`jconstructor`* was declared.

procedure: (**`java-constructor-parameter-types`** `jconstructor`) => list of jclasses

Returns the declared types of the parameters of *`jconstructor`*.

procedure: (**`java-constructor-procedure`** `jconstructor`) => procedure

Returns a procedure that when called invokes the constructor with the passed arguments, returning the newly created objected.

procedure: (**`java-constructor-method`** `jconstructor`) => method

Returns a method suitable for adding to generic procedures that, when called invokes the underlying Java constructor with the passed arguments. The resulting newly created object is returned.

8.3.3. Methods

procedure: (**`java-method?`** `value`) => #t/#f

Determines whether *`value`* is a Java method.

procedure: (**`java-method-name`** `jmethod`) => symbol

Returns the name of *`jmethod`*.

procedure: (**`java-method-flags`** `jmethod`) => list of symbols

Returns the modifiers of *`jmethod`*, such as `public static final`.

procedure: (**`java-method-declaring-class`** `jmethod`) => jclass

Returns the Java class in which *`jmethod`* was declared.

procedure: (**`java-method-parameter-types`** `jmethod`) => list of jclasses

Returns the declared types of the parameters of *`jmethod`*.

procedure: (**`java-method-procedure`** `jmethod`) => procedure

Returns a procedure that when called invokes the method with the passed arguments, returning the newly created objected.

procedure: (**`java-method-method`** `jmethod`) => method

Returns a method suitable for adding to generic procedures that, when called invokes the underlying Java method on the object passed as the first argument, and with the remaining arguments passed as parameters. The result of the method invocation is returned. Static methods can be invoked by passing a typed null object as the first parameter to the generic procedure.

8.3.4. Fields

procedure: (**`java-field?`** `value`) => #t/#f

Determines whether *`value`* is a Java field.

procedure: (**`java-field-name`** `jfield`) => symbol

Returns the name of *`jfield`*.

procedure: (**`java-field-flags`** `jfield`) => list of symbols

Returns the modifiers of *`jfield`*, such as `public static final`.

procedure: (**`java-field-declaring-class`** `jfield`) => jclass

Returns the Java class in which *`jfield`* was declared.

procedure: (**`java-field-type`** `jfield`) => jclass

Returns the declared type of *`jfield`*.

procedure: (**`java-field-accessor-procedure`** `jfield`) => procedure

procedure: (**`java-field-modifier-procedure`** `jfield`) => procedure

Returns a procedure that when called returns or sets (respectively) the value of the field on the object specified by the first parameter to the invocation. Static fields can be accessed/modified by passing a null object.

procedure: (**`java-field-accessor-method`** `jfield`) => method

procedure: (**`java-field-modifier-method`** `jfield`) => method

Returns a method suitable for adding to generic procedures that, when called returns/sets the value of the field on the object specified by the first argument to the generic procedure invocation. Static fields can be accessed/modified by passing a typed null object as the first parameter to the generic procedure.

8.3.5. Arrays

procedure: (**`java-array-class`** `jclass dimensions`) => jclass

Returns a class representing the array type that has *`jclass`* as the component type and *`dimensions`* as the number of dimensions. For example, the following expressions are equivalent:

```
(java-array-class <jint> 2)
(java-class '|int[][]|)
```

The list of direct superclasses returned by `java-class-declared-superclasses` for an array class is consistent with the widening conversion performed by Java, e.g. the direct superclasses of `java.util.ArrayList[][]` are:

- `java.util.AbstractList[][]`
- `java.util.List[][]`
- `java.util.RandomAccess[][]`

- `java.lang.Cloneable[][]`
- `java.io.Serializable[][]`

  This is different from what the Java reflection APIs return.

8.3.6. Proxies

procedure: (**`java-proxy-class`** `jinterface`) => jclass

Creates a Java class that implements the specified interfaces. The class can be instantiated with an invocation handler, such as the one returned by `java-proxy-dispatcher` below, that delegates method invocation to Scheme code.

procedure: (**`java-proxy-dispatcher`** `alist`) => invocation-handler

Creates an invocation handler suitable for use in the instantiation of a proxy (see `java-proxy-class` above). The keys in *`alist`* are Java method names and the values are Scheme procedures.

When a method is invoked on a proxy, the procedure matching the method's name is invoked with the proxy object and the parameters of the method invocation as arguments. The result of the invocation is returned as the result of the method call. If *`alist`* does not contain a binding for the method name, an error is signalled.

Chapter 9. Additional Libraries

9.1. Optional *SISC* Libraries

The optional *SISC* libraries are modules whose definition is included in the full *SISC* distribution, but not the lite distribution.

9.1.1. Definitions

Requires: **(import *misc*)**

In addition to the standard R^{5}RS definition syntaxes, *SISC* provides an additional value definition and syntax definition form.

First, `define-values`, which allows more than one binding to be created at once, given the multiple-value return of its body.

syntax: (**`define-values`** `(binding binding ... ) expression`) => undefined

Evaluates the expression in the body, which must return the same number of values as there are binding names. Each value is then bound (in an undefined order) to each binding name.

`define-values` behaves like `define` in terms of which environment the bindings are created. If the `define-values` statement is at the top-level then bindings are created in the top-level environment. If the statement is in a lexical environment, then it behaves just as an internal define.

Second, `define-simple-syntax` provides a shorthand for syntax definition when the syntactic form's appearance is similar to a function.

syntax: (**`define-simple-syntax`** `(name vars ... ) body`) => undefined

Creates a syntactic form with the given name, and any number of listed syntactic variables, which expands to the given body (with instances of the syntactic variables hygienically expanded).

Here is an example usage of `define-simple-syntax` to define the `when` macro:

```
(define-simple-syntax (when condition body ...)
  (if condition
      (begin body ...)))
```

9.1.2. Bitwise Logical Operations

Requires: **(import *logicops*)**

In addition to the R^{5}RS set of procedures that deal with numbers, *SISC* provides operators for performing bitwise logic operations on exact integers.

procedure: (**`logand`** `integer [integer] ...`) => integer

Performs the logical AND of all the provided arguments.

procedure: (**`logor`** `integer [integer] ...`) => integer

Performs the logical OR of all the provided arguments.

procedure: (**`logxor`** `integer [integer] ...`) => integer

Performs the logical exclusive-OR of all the provided arguments.

procedure: (**`lognot`** `integer`) => integer

Performs the logical NOT of the provided integer.

procedure: (**`logcount`** `integer`) => integer

Returns the count of the number of 1 bits in the representation of a given positive integer, or 0 bits in a negative integer.

In addition, two operators are provided to perform arithmetic shifts on any integer (these operators do not have the range limitation the previous logical functions do). The shift operators return a newly generated number formed by shifting the provided number left or right by the given number of bits.

procedure: (**`ashl`** `integer bits`) => integer

Arithmetically shifts *`integer`* left by *`bits`* bits.

procedure: (**`ashr`** `integer bits`) => integer

Arithmetically shifts *`integer`* right by *`bits`* bits.

Mathematically, if r is the number, and s is the number of bits, ashl calculates:

$r \times 2^{s}$

while ashr calculates

$r / 2^{s}$

in the integer domain. Both ashl and ashr operate on exact integers and produce only exact integers.

9.1.3. Records

Requires: **(import *record*)**

SISC provides a native implementation of record types as defined in SRFI-9. See http://srfi.schemers.org/srfi-9/ for details. In addition to the `define-record-type` syntax provided by SRFI-9, a more compact (but less flexible) `define-struct` syntax is offered.

syntax: (**`define-struct`** `name (field ...)`) => void

Defines a SRFI-9 record type as follows:

```
(define-record-type (make-name field ...)
  name?
  (field name-field set-name-field!)
  ...)
```

i.e. naming conventions are used to determine the names of the record type constructor, predicate, field access and field modifier procedures.

Records are `eq?` and `eqv?` if and only if they are identical. Records are `equal?` if and only if they are instances of the same record type and all their fields are `equal?`.

It is also possible to define *non-generative* record types, using `define-nongenerative-record-type` and `define-nongenerative-struct`. Non-generative record types are associated with a user-specified guid. If an attempt is made to define a record type with a guid that is already bound to an existing record type then the existing record type is modified, instead of a new record type being created. Non-generative record types are serialised specially such that deserialising them also performs this check. By contract, deserialisation of ordinary, generative record types and their instances results in duplicate types being created, which is usually not desirable.

syntax: (**`define-nongenerative-record-type`** `name guid (constructor-name field ...) (predicate ...) (field-spec ...)`) => void

This is the same as `define-record-type`, except that the resulting record type is *non-generative* with *`guid`*, a symbol, as the unique identifier.

syntax: (**`define-nongenerative-struct`** `name guid (field ...)`) => void

This is the same as `define-struct`, except that the resulting struct is *non-generative* with *`guid`*, a symbol, as the unique identifier.

9.1.4. Hash Tables

Requires: **(import *hashtable*)**

Hash tables store mappings of keys to values. Hence they are similar to association lists, except that hash tables allow retrieval, addition and modification in constant time whereas association lists typically perform these operations in linear time based on the number of elements.

9.1.4.1. Creation and Introspection

Hash tables are a distinct data type. They can be created empty or filled with the contents of an association list. The converse, creating an association list from a hash table, is also supported.

procedure: (**`make-hashtable`** `[equivalence-predicate] [hash-function] [thread-safe?] [weak?]`) => hashtable

Creates a hash table. The first optional argument supplies the equivalence test procedure that the hashtable should use for comparison of keys. This must be a function accepting two arguments and returning a boolean. It defaults to `equal?`.

The second optional argument supplies the hash function, which must accept one argument and return a numeric value. For the equivalence predicates `eq?`, `eqv?`, `equal?`, `string=?`, `string-ci=?` it defaults to `hash-by-eq`, `hash-by-eqv`, `hash-by-equal`, `hash-by-string=`, `hash-by-string-ci=?` respectively, and `hash-by-equal` otherwise.

The third optional argument determines whether operations on the hash table should be made thread-safe. The default is #t. Thread synchronization (see Chapter 6) is required if there are potentially several threads operating concurrently on the hash table and one of these threads performs a structural modification (i.e. adds or removes an entry; merely changing the value of an entry is not a structural modification). Failure to enforce proper thread synchronization has unpredicatable results.

The fourth optional argument determines whether the keys in the hash table are held with weak references, allowing them to be garbage collected, and automatically removed from the hashtable when they are not referenced from elsewhere. The default is #t.

For reasons of disambiguation, the hash function argument can only be supplied if the preceeding equivalence predicate was also supplied, and the weakness argument can only be supplied if the preceeding thread-safety argument was also supplied.

The equivalence and hash function must produce stable results for the keys in a hash table.
The effects of invoking an escaping continuation inside the equivalence predicate or hash function, or invoking a continuation captured inside the equivalence predicate or hash function after that function has returned, are unspecified.

procedure: (**`alist->hashtable`** `alist [equivalence-predicate] [hash-function] [thread-safe?] [weak?]`) => hashtable

Creates a hashtable and initializes it with the keys and values found in *`alist`*. *`alist`* must be a list of pairs, with the `car` of each pair representing a key and the `cdr` representing its associated value. The optional arguments are the same as for `make-hashtable`.
If there are multiple pairs which contain the same key (with respect to chosen equivalence test) then the resulting hash table will associate the key with the value of the last such pair.

procedure: (**`hashtable?`** `value`) => #t/#f

Returns #t if *`value`* is a hash table, #f otherwise.

procedure: (**`hashtable/equivalence-function`** `hashtable`) => procedure

Returns the equivalence predicate of *`hashtable`*.

procedure: (**`hashtable/hash-function`** `hashtable`) => procedure

Returns the hash function of *`hashtable`*.

procedure: (**`hashtable/thread-safe?`** `hashtable`) => #t/#f

Returns #t if *`hashtable`* is thread safe, #f otherwise.

procedure: (**`hashtable/weak?`** `hashtable`) => #t/#f

Returns #t if the keys in *`hashtable`* are held by weak references, #f otherwise.

procedure: (**`hashtable/size`** `hashtable`) => number

Returns the number of key/value pairs stored in *`hashtable`*.
procedure: (**`hashtable->alist`** `hashtable`) => alist

Returns an association list comprising the elements of *`hashtable`*. The list contains pairs whose `car`s are they keys found in *`hashtable`* and whose `cdr`s contain the associated values.

9.1.4.2. Hash Functions

Several hash functions that return results consistent with common equivalence predicates are predefined.

procedure: (**`hash-by-eq`** `value`) => number

procedure: (**`hash-by-eqv`** `value`) => number

procedure: (**`hash-by-equal`** `value`) => number

procedure: (**`hash-by-string=`** `string`) => number

procedure: (**`hash-by-string-ci=`** `string`) => number

These procedures return a hash code of their argument that is consistent with `eq?`, `eqv?`, `equal?`, `string=?`, `string-ci=?`, respectively.

9.1.4.3. Access and Modification

All hash table access operations follow a similar pattern. They return the value that was associated with the the given key at the time the operation was invoked. If no binding for the key existed, an optionally supplied value is returned that defaults to #f. This allows the programmer to associate keys with #f values and distinguish this case from not having any association for a key.

procedure: (**`hashtable/put!`** `hashtable key val [nobinding]`) => value

Associates *`key`* with *`val`* in *`hashtable`*. Returns the previous association of *`key`* or *`nobinding`*, which defaults to #f, if *`key`* has no previous association.

procedure: (**`hashtable/get`** `hashtable key [nobinding]`) => value

Returns the value associated with *`key`* in *`hashtable`*, or *`nobinding`*, which defaults to #f, if *`key`* has no association.

procedure: (**`hashtable/get!`** `hashtable key thunk [unsafe?]`) => value

Returns the value associated with *`key`* in *`hashtable`*. If *`key`* has no association then *`thunk`* is called and the result is associated with *`key`* in *`hashtable`* and also returned. The *`unsafe?`*, which defaults to #t, indicates whether *`thunk`* may invoke escaping continuations or raise errors. Setting *`unsafe?`* to #f results in more efficient execution but may cause deadlocks if *`thunk`* is unsafe. See also `mutex/synchronize-unsafe` in Section 6.5.3.

When *`hashtable`* is thread-safe this operation is atomic.

procedure: (**`hashtable/contains?`** `hashtable key`) => #t/#f

Returns the #t if *`hashtable`* contains an entry for *`key`*, #f otherwise.

procedure: (**`hashtable/remove!`** `hashtable key [nobinding]`) => value

Removes the association of *`key`* in *`hashtable`*. Returns the associated value of *`key`* or *`nobinding`*, which defaults to #f, if *`key`* has no association.

9.1.4.4. Bulk Operations

Bulk operations are operations that apply to all elements of a hash table.

procedure: (**`hashtable/clear!`** `hashtable`) => void

Removes all elements from *`hashtable`*.

procedure: (**`hashtable/keys`** `hashtable`) => list

Returns the keys contained in *`hashtable`*.

procedure: (**`hashtable/for-each`** `proc hashtable`) => void

Applies *`proc`* to each element of *`hashtable`*. *`proc`* is called with two parameters - the key and the value of the element.

procedure: (**`hashtable/map`** `proc hashtable`) => list

Applies *`proc`* to each element of *`hashtable`*. *`proc`* is called with two parameters - the key and the value of the element. The results of calling *`proc`* are returned as a list.

9.1.5. Binary Buffers

Requires: **(import *buffers*)**

Binary buffers provide an opaque container for a fixed amount of binary data. The binary buffer library provides a number of functions for creating and accessing those buffers. The buffer is very similar to a vector, in that it is a randomly accessable, zero-based structure. But as a tradeoff for space efficiency, binary buffers are only capable of storing bytes. The bytes are stored as 8-bit, unsigned fixed integers (of the range 0-255).

procedure: (**`buffer?`** `value`) => #t/#f

Returns true if and only if the provided argument is a binary buffer.

procedure: (**`make-buffer`** `size [fill-value]`) => buffer

Creates a new buffer capable of storing *`size`* bytes. *`size`* must be a fixed non-negative integer. If provided, the value of all bytes in the buffer is initialized to *`fill-value`*. If not provided, the contents of the buffer is unspecified.

procedure: (**`buffer`** `[value] ...`) => buffer

Creates a new buffer whose size is equal to the number of arguments given and whose contents are the bytes given as arguments.

procedure: (**`buffer-length`** `buffer`) => fixed integer

Returns the capacity of the given buffer.

procedure: (**`buffer-ref`** `buffer index`) => fixed integer

Returns the byte at offset *`index`* in the specified buffer. It is an error if *`index`* is out of range.

procedure: (**`buffer-set!`** `buffer index new-value`) => undefined

Sets the byte at offset *`index`* of the specified buffer to the given fixed integer *`new-value`*. It is an error if *`index`* is out of range.

procedure: (**`buffer-copy!`** `source-buffer source-offset dest-buffer dest-offset [count]`) => undefined

Copies *`count`* bytes starting from index *`source-offset`* in the source buffer to successive bytes starting at index *`dest-offset`* in the destination buffer. If *`count`* is unspecified, it is assumed to be the length of the source buffer. It is an error to copy more bytes from the source buffer than are available, or to copy more bytes into the destination buffer than its capacity allows.

Buffers are serializable (can exist in loadable libraries or a *SISC* heap), but are not representable in an s-expression. For this reason, they bear the printed representation of `#<buffer>`.

9.1.6. Procedure Properties

Requires: **(import *procedure-properties*)**

SISC allows key/value bindings to be associated with procedures. This has a number of applications. For instance, generic procedures store their methods in a procedure property.

Keys must be symbols. Values are any valid Scheme value. All operations are thread-safe.

procedure: (**`procedure-property`** `proc symbol [nobinding]`) => value

Returns the value associated with the property *`symbol`* of procedure *`proc`*, or *`nobinding`*, which defaults to #f, if the property is not set.

procedure: (**`set-procedure-property!`** `proc symbol val [nobinding]`) => value

Sets the property *`symbol`* of procedure *`proc`* to the value *`val`*. Returns the previous value of the property or *`nobinding`*, which defaults to #f, if the property was unset.

procedure: (**`procedure-property!`** `proc symbol thunk [unsafe?]`) => value

Returns the value associated with the property *`symbol`* of procedure *`proc`*. If the property is unset then *`thunk`* is called and the property is set to the result, which is also returned. The *`unsafe?`*, which defaults to #t, indicates whether *`thunk`* may invoke escaping continuations or raise errors. Setting *`unsafe?`* to #f results in more efficient execution but may cause deadlocks if *`thunk`* is unsafe. See also `mutex/synchronize-unsafe` in Section 6.5.3.

9.1.7. Loadable Scheme Libraries

Requires: **(import *libraries*)** [1]

Scheme code can be packaged into libraries that can have dependencies on other libraries and can be loaded as required. Libraries are identified by a name that follows Java package file naming conventions, i.e. using path-style names typically containing domain, organisation name, project name and library name. For instance, if company Foo produces a library Baz for project Bar and that library contains three files, the file structure might look as follows:

```
com/foo/bar/baz.scm
com/foo/bar/baz/baz1.scm
com/foo/bar/baz/baz2.scm
com/foo/bar/baz/baz3.scm
```

This library can be made accessible from *SISC* by adding the base directory or a jar file containing these files to the Java class path. Libraries are loaded by the following procedure.

procedure: (**`require-library`** `name`) => undefined

Checks whether the library identified by *`name`* (a string), has already been loaded and, if not, loads it. An error is raised if the library cannot be found.

Libraries are loaded using the `load` procedure from a resource located by the `find-resource` procedure. The name of the resource is derived from the name of the library by appending ".scc", ".sce" and, if that does not succeed, ".scm".

Note that `require-library` only loads a single file. The definition of dependencies on other libraries and the loading of other files therefore needs to happen within that file. For instance, the file `com/foo/bar/baz.scm` from the above example might contain the following:

```
(require-library 'com/foo/bar/boo)
(load "baz/baz1.scm")
(load "baz/baz2.scm")
(load "baz/baz3.scm")
```

It is possible to programmatically check whether a particular library exists and whether it has been loaded:

procedure: (**`library-exists?`** `name`) => #t/#f

Returns #t if the library identified by *`name`* (a string) exists, #f otherwise.

procedure: (**`library-loaded?`** `name`) => #t/#f

Returns #t if the library identified by *`name`* (a string) has been loaded, #f otherwise.

9.1.8. Operating System Interface

Requires: **(import *os*)**

The operating system interface currently contains functions for spawning external processes on the host operating system, obtaining input/output ports to the resulting process, and monitoring their status.

Two procedures exist for spawning processes:

procedure: (**`spawn-process`** `program/commandline [arglist]`) => process

Spawns a process, returning a process handle. If the optional argument list is provided, then the first argument is the binary to run with those arguments. If omitted, the first argument is tokenized as a commandline and used to spawn the process.

procedure: (**`spawn-process-with-environment`** `program arglist environment [working-directory]`) => process

procedure: (**`spawn-process/env`** `program arglist environment [working-directory]`) => process

Spawns a process named by *`program`* with the arguments given in *`arglist`*, in the given *`environment`*. The environment is an association list of strings to strings. The key in the association list is an environment variable name, and the corresponding value is the value to assign to that environment variable. If the environment parameter is `#f`, the environment variables of the current *SISC* instance are used.

The optional parameter `working-directory` specifies the directory which will be set as the current directory when the process is spawned. If ommited, the value of the `current-directory` parameter (i.e. the current directory of the running Scheme program) is used instead.

procedure: (**`process?`** `value`) => #t/#f

Returns `#t` if the given value is a process handle.

Once started, a process will run in parallel to the current Scheme program according to the usual scheduling of the host platform. The process handle obtained can be used to obtain the input, output, and error streams of the process using the following functions:

procedure: (**`get-process-stdout`** `process`) => binary-input-port

Returns a binary input port which will read bytes which the given process has written to its standard output stream.

procedure: (**`get-process-stderr`** `process`) => binary-input-port

Returns a binary input port which will read bytes which the given process has written to its standard error stream.

procedure: (**`get-process-stdin`** `process`) => binary-output-port

Returns a binary output port which when written to will send bytes to the given process' standard input stream.

Finally, functions are provided to check the status of a spawned process, and to wait for a process to complete:

procedure: (**`process-terminated?`** `process`) => integer or #f

Checks to see if the given process has terminated, and returns the process' return code if so. If the process is still running, `#f` is returned.

procedure: (**`wait-for-process`** `process`) => integer or #f

Waits for the given process to terminate, and returns the process' return code if so. The wait operation may be interrupted by other code, in which `#f` is returned.

9.2. Third-Party Libraries

SISC provides hooks for accessing a number of third-party Scheme libraries.

Warning

This functionality has not undergone much testing.

9.2.1. SRFIs

The Scheme Requests For Implementation (SRFI) process aims to coordinate libraries and other additions to the Scheme language between different Scheme implementations. For details see http://srfi.schemers.org/ which describes the process and contains a list of all available SRFIs.

9.2.1.1. SRFI Modules

In *SISC* each SRFI is encapsulated in a module. See Chapter 10 for details of *SISC*'s module system. The definitions for SRFI modules are not included in the standard *SISC* heap build and hence must be loaded separately from various compiled library files in the `sisc-lib.jar` jar file in the root directory of the *SISC* binary distribution. As long as this jar file is on the classpath, which is the case by default, any SRFI's module definition may be loaded with the expression `(require-library 'sisc/libs/srfi/srfi-`*`n`*`)`, where *`n`* is the SRFI's number. For example:

```
(require-library 'sisc/libs/srfi/srfi-9)
```

All SRFI's may be loaded at once by requiring `sisc/libs/srfi`.

9.2.1.2. Using SRFIs

SISC currently supports SRFIs 0, 1, 2, 5, 6, 7, 8, 9, 11, 13, 14, 16, 18, 19, 22, 23, 25, 26, 27, 28, 29, 30, 31, 34, 35, 37, 38, 39, 40, 42, 43, 45, 48, 51, 54, 55, 59, 60, 61, 62, 66, 67, 69 and 78. Once the SRFI module definitions have been loaded as described above, a SRFI *`n`* can be imported using

```
(import srfi-n)
```

e.g.

```
(import srfi-1)
(xcons 1 2) ;=> (2 . 1)
```

SRFI modules, like all modules in *SISC*, can be imported/used by other modules. Doing so does not pollute the top-level environment with the definitions exported by the module, i.e. any code outside the importing module remains unaffected.

If, however, an SRFI is to be imported into the top-level, one can use the `require-extension` mechanism (see Section 10.2.1).

9.2.1.3. SRFI Extensions

Some SRFIs have built-in extension points that Scheme implementations can use to augment a SRFI's functionality. It is also the case that some SRFIs would benefit from slightly extended APIs.

This section documents the SRFI extensions implemented by *SISC*.

9.2.1.3.1. SRFI 69 (Basic hash tables)

The `make-hash-table` function takes two additional optional arguments: *`thread-safe?`* and *`weak?`*. See `make-hashtable` in Section 9.1.4.1 for details.

The basic hash table API is extended in a separate module:

Requires: **(import *srfi-69-ext*)**

procedure: (**`hash-table-thread-safe?`** `hashtable`) => #t/#f

Returns #t if *`hashtable`* is thread safe, #f otherwise.

procedure: (**`hash-table-weak?`** `hashtable`) => #t/#f

Returns #t if the keys in *`hashtable`* are held by weak references, #f otherwise.

procedure: (**`hash-table-ref!`** `hashtable key thunk`) => value

Returns the value associated with *`key`* in *`hashtable`*. If *`key`* has no association then *`thunk`* is called and the result is associated with *`key`* in *`hashtable`* and also returned.

When *`hashtable`* is thread-safe this operation is atomic.

procedure: (**`hash-table-ref!`** `hashtable key default`) => value

Returns the value associated with *`key`* in *`hashtable`*. If *`key`* has no association then *`default`* is associated with *`key`* in *`hashtable`* and also returned.

When *`hashtable`* is thread-safe this operation is atomic.

9.2.2. SLIB

The SLIB portable scheme library provides compatibility and utility functions for standard Scheme implementations. It is supported by many Schemes, including *SISC*.

9.2.2.1. Downloading and Installation

The latest version of SLIB is available from http://swissnet.ai.mit.edu/~jaffer/SLIB.html as both a zip file and RPM. The site also hosts an online version of the SLIB manual.

Download SLIB and install it in a convenient location. The RPM will by default be installed in `/usr/share/slib/`. Do not worry when you see some errors about missing programs such as mzscheme and scheme48 when installing the RPM - these happen because SLIB tries to auto-configure itself for various Schemes that you may not have installed on your system.

9.2.2.2. Environment

Using SLIB in *SISC* requires two Java system properties to be set:

- **sisc.home.** This should (but does not actually *have* to) point to the location where you have installed *SISC*. If you are using one of the scripts from the binary *SISC* distribution in order to run *SISC* then this property will automatically be set to the value of the SISC_HOME environment variable.
- **sisc.slib.** This must point to the location where you installed SLIB. Other Schemes supporting SLIB tend to use an environment variable SCHEME_LIBRARY_PATH, so it is advisable to define that (if it is not already defined) and run Java with a `-Dsisc.slib=...` option based on the environment variable. If you are using the scripts from the binary $SISC; distribution in order to run *SISC* then you can set the property by adding the `-Dsisc.slib=...` to the JAVAOPT environment variable.

  Note: Note that the value of this property should be a fully qualified url, e.g. `file:///usr/share/slib`

You need to ensure that all potential users of SLIB have *read* permissions to files in the directories referred to by the above system properties.

9.2.2.3. Building the Catalog

Make sure that the above system properties are set and that you have *write* permissions to the sisc.home directory; often this means you need to be logged in as a privileged user.

Start *SISC* as you normally would. At the prompt type

```
(require-library 'sisc/libs/slib)
(require 'new-catalog)
(exit)
```

The above should create a file `slibcat` in the sisc.home directory. It is a good idea to check that this has indeed happened.

9.2.2.4. Using SLIB

Make sure the above system properties are set. Start *SISC* as you normally would. At the prompt load the *SISC* SLIB as described above, i.e.

```
(require-library 'sisc/libs/slib)
```

You can now load SLIB modules using `require`, e.g.

```
(require 'tsort)
(tsort '((shirt tie belt)
         (tie jacket)
         (belt jacket)
         (watch)
         (pants shoes belt)
         (undershorts pants shoes)
         (socks shoes))
       eq?)
```

loads the topological sorting module and invokes one of the procedures defined by it.

Please refer to the SLIB manual for further details of what modules are available. Note however that, as with most other Schemes supported by SLIB, there will be some modules that are not available or do not work in *SISC*.

9.3. Creating Libraries

Requires: **(import *libraries*)**

SISC allows the creation of *compiled libraries* which contain compiled scheme code. These libraries can then be executed into a running *SISC* session in order to extend the functionality without processing or possessing the original source. Such libraries can be loaded using `load` as would any ordinary Scheme source file.

Compiled code files (`.scc`) are created using the `compile-file` function, which takes a Scheme source file and a target output file, and processes the Scheme source through the various expansion and compilation phases, and then serializes the resulting *SISC* microexpressions to the given target file. The resulting file may then be loaded with `load` as any ordinary Scheme file would, or can be placed in the library path and resolved using `require-library`.

procedure: (**`compile-file`** `source-file target-file`) => undefined

Compiles the Scheme source present in *`source-file`*, writing the resulting micro-expressions into *`target-file`*, suitable for loading.

As a side effect, the micro-expressions are also evaluated, i.e. in effect `compile-file` compiles *and evaluates* the *`source-file`*. The latter is necessary because the compilation of an expression may depend on the results of evaluating a previous expression, e.g. as is typically the case for libraries that depend on other libraries.

> **`.sll` Deprecation:** Scheme Loadable Libraries (`.sll` files) were deprecated in version 1.9. This was due to unresolvable incompatibilities in the engine's closure representation and the `.sll` functionality.

Notes

1. This module gets imported by default.

Chapter 10. Modules and Libraries

10.1. Modules

Modules provide an additional level of scoping control, allowing symbolic and syntactic bindings to be bundled in a named or anonymous package. The package can then be imported into any scope, making the bindings contained in the module visible in only that scope.

SISC's modules are provided by the portable syntax-case macro expander by R. Kent Dybvig and Oscar Waddell. A comprehensive explanation of the provided module system is best found in the *Chez Scheme Users Guide* (http://www.scheme.com/csug.html), specifically *Section 9.3, Modules* (http://www.scheme.com/csug/syntax.html#g2187). What follows is an informal introduction to that module system.

10.1.1. Overview

The basic unit of modularization in *SISC* is a module. A typical module definition has this appearance:

```
(module foo
    (bar baz)
  (import boo1)
  (import boo2)
  (include "file.scm")
  (define (bar x) ...)
  (define-syntax baz ...)
  (define (something-else ...) ...)
  (do-something)
  (do-something-else))
```

A module definition consists of a name (`foo`), a list of exports (`bar` and `baz`) and a body. Expressions which can appear in the body of a module are the same as those which can appear in a `lambda` body. The `import` form imports bindings from a named module (in this case `boo1` and `boo2`) into the current lexical scope. The `include` form performs a textual inclusion of the source code found in the named file (`file.scm`). In other words, it works as if the contents of the file had appeared literally in place of the `include` statement.

All identifiers appearing in the export list must be `define`d or `define-syntax`ed in the body of the module, or `import`ed from another module.

10.1.2. Style

It is recommended to clearly separate modularization from actual code. The best way to accomplish this is to

- List all imports in the module body rather than in included files
- Include all files directly from the module body, avoiding nested includes

- Place all definitions and expressions in included files, avoiding them in the module body

There are several reasons for this. First, it makes refactoring easier, as one can move relevant code from module to module merely by rewriting the module definitions, leaving the implementation code unchanged. Second, it makes debugging easier, as one can load the implementation code directly into the Scheme system to have access to all bindings, or load the module definition to view the finished, encapsulated exports. Finally, it stylistically separates interface (the modules) from implementation (the included Scheme source).

10.1.3. Modularizing Existing Code

Since module bodies are treated like the bodies of `lambda`s, the R^{5}RS rules of how internal definitions are treated apply to all the definitions in the module body (both ordinary and syntax), including all code `include`d from files. This is often a source of errors when moving code from the top-level into a module because:

- *All* definitions must appear *before all* expressions,
- The list of definitions is translated into `letrec`/`letrec-syntax`, which means it must be possible to evaluate each right-hand side without assigning or referring to the value of any of the variables being defined.

This often necessitates re-arranging the code and the introduction of `set!` expressions. Here is an example of a sequence of top-level definitions/expressions and how they need to be rewritten so that they may appear in a module body:

```
(define (foo) 1)
(define bar (foo))
(do-some-stuff)
(define (baz) (bar))
==>
(define (foo) 1)
(define bar)
(define (baz) (bar))
(set! bar (foo))
(do-some-stuff)
```

The general strategy is to go through the list of expressions/definitions from top to bottom and build two lists - one of definitions and one of expressions - as follows:

- If a non-definition is encountered, append it to the expression list
- If a "naked" definition (i.e. a definition whose right-hand side is not a function) that refers to a binding defined within the module is encountered, append an empty definition to the definition list and append a `set!` with the right-hand side expression to the expression list
- Otherwise, i.e. for an ordinary definition, append it to the definition list

The concatenation of the resulting definition list with the expression list makes a suitable module body.

10.1.4. Evaluation

Modules are lexically scoped. It is possible to define modules inside `lambda`s and inside other modules and to export modules from modules. Example:

```
(define (f c)
  (module foo
      (bar)
    (module bar
        (baz)
      (define (baz x y) (- x y))
      (display "defining baz\n")))
  (if (> c 0)
      (let ((a 1))
         (import foo)
         (let loop ((b c))
            (import bar)
            (if (> b 0) (loop (baz b a)) (f (- c 1)))))))
```

The expressions in a module body get executed at the time and in the context of module definition. So, in the above example, the body of bar containing the display statement is executed once for every call to `f` rather than once for every iteration of the inner loop containing the import of the `bar` module.

There are quite a few more things you can do with modules. For instance one can define anonymous modules, which are a short cut for defining a named module and then importing it, import selected bindings from a module and renaming them rather then importing all bindings as is etc etc. For more details again refer to the Chez Scheme user manual.

10.2. Libraries

Libraries provide a means of encapsulating code that can be shared by many, independently developed applications.

Libraries are simply bundles of Scheme code, usually precompiled, which are packaged so that they may be resolved relative to a library path. Libraries are typically compiled using the meachanism from Section 9.3. Loading the resulting library makes the library available to the loading code. To create a compiled library from a module, compile a source file which contains any necessary `require-library` calls, followed by the module definition. When loaded, this will cause the necessary libraries to be loaded, and then define the module into the environment. For example, the source file may resemble:

```
(require-library 'sisc/libs/srfi/srfi-1)
(require-library 'com/foo/lib2)

(module lib3
    (a-function)
  (import srfi-1)
  (import com/foo/lib2)
  (define (a-function)
    (do-something (another-function)))
  (define (another-function)
    (something-else))
```

Libraries should not depend on any top-level definitions outside the standard *SISC* top-level, except the definition of other library modules. Otherwise it is not possible to use the libraries portably.

Libraries can be packaged with supporting code (e.g. ordinary Java code and native modules) and other libraries into jar files. A typical structure for such a jar file would be

```
com/foo/lib1.scc
com/foo/lib1/Class1.class
com/foo/lib1/Class2.class
com/foo/lib2.scc
com/foo/lib2/Class1.class
com/foo/lib2/Class2.class
com/foo/lib3.scc
com/foo/lib3/Class1.class
com/foo/lib3/Class2.class
```

It is usually a good idea to name a module after the path names in the jar, for example `com/foo/lib{1,2,3}` in the above example.

10.2.1. `require-extension`

SISC supports SRFI-55 (http://srfi.schemers.org/srfi-55/srfi-55.html) for loading libraries and extensions as well. SRFI-55 provides `require-extension`, which in *SISC* simultaneously loads a library, then imports its module definition into the current interaction environment. This may be more convenient than the combination of `require-library` and `import`, when one is loading dependent top-level libraries for a program. It is less flexible, though, since you cannot import into a lexical scope.

SRFI-55 is supported in the initial *SISC* environment, no `require-library` or `import` is needed to use `require-extension`.

At the time of this writing, *SISC* supports two extension identifier schemes, the `srfi` scheme as required by SRFI-55 itself, and a *SISC* specific `lib` scheme for loading a *SISC* library. Some examples:

Example 10-1. Loading and importing with `require-extension`

```
; Load and import SRFI 1
(require-extension (srfi 1))

; Load and import SISC library com/foo/lib1
(require-extension (lib com/foo/lib1))

; Load and import SRFIs 13 and 14,
; and SISC libraries com/foo/lib2 and com/foo/lib3
(require-extension (srfi 13 14) (lib com/foo/lib2 com/foo/lib3))
```

SISC modules loaded using the `lib` extension scheme must use the full path and file as the module name. For example, in the above example, `com/foo/lib1`'s module name must be `com/foo/lib1`.

Chapter 11. Extensibility

Occasionally functionality may be desired that is not easily accomplished at the Scheme level. A new first-class type may be desired, or efficient access to a Java library. *SISC* provides a simple API for such modifications.

11.1. Adding Types

A Scheme value is represented in *SISC* as a subclass of the abstract Java class `sisc.data.Value`.

11.1.1. External Representation of Values

In order to be able to display the value in the Scheme system, all Values must implement the `display` method:

public void **display**(sisc.io.ValueWriter writer);

Uses the various output methods of sisc.io.ValueWriter to construct an external representation of the given Value suitable for output from the `display` Scheme function.

If the programmer desires the Value to have a different representation when written with `write`, the `write` method must be overridden. If it is not, the output of `display` is used for `write` as well.

public void **write**(sisc.io.ValueWriter writer);

Uses the various output methods of sisc.io.ValueWriter to construct an external representation of the given Value suitable for output from the `write` Scheme function. If not implemented, the output constructed the by the `display` method is used as output from `write`.

Finally, if the external representation of a new Value is likely to be long, the programmer should implement the `synopsis` method, which generates a *summary representation* of the value.

public String **synopsis**(int limit);

Returns approximately *`limit`* characters from the printable representation of this value as if returned by `write`. This method is used for displaying the value in error messages where the entire representation may be superfluous.

The sisc.io.ValueWriter type that is passed as an argument to both `display` and `write` contains a number of methods to generate the representation. First there are several methods for appending Java Strings, characters, and SISC Values. In each, the called ValueWriter is returned, to allow for easy chaining of calls.

public sisc.io.ValueWriter **append**(char c);

Appends a single character to the external representation.

public sisc.io.ValueWriter **append**(String s);

Appends the contents of a Java String to the external representation.

public sisc.io.ValueWriter **append**(sisc.data.Value v);

Appends the contents of a Scheme value to the external representation. The value is converted to an external representation using the print style with which the ValueWriter was constructed (for example, `display` or `write`).

In addition to the above `append` methods, the programmer may wish to force `display` or `write` rather than use the same method as the ValueWriter. To do this, one can call the `display` or `write` methods on the ValueWriter.

public void **display**(sisc.data.Value v);

Appends the external representation of the given value as returned by `display`.

public void **write**(sisc.data.Value v);

Appends the external representation of the given value as returned by `write`.

11.1.2. Equality

If the one wants a Value to be comparable for any more than pointer equality, or for the concept of pointer equality to be less strict than actual pointer equality, one or more of the equality methods must be overridden.

public boolean **eq**(Object other);

Returns true if another provided Java object is equal in the sense of `eq?` to this Value.

public boolean **valueEqual**(Value other);

Returns true if another provided Scheme value is equal in the sense of `equal?` to this Value.

11.1.3. Serializable Values

If the type that is being added will be serialized in a *SISC* heap, and it contains one or more member variables, the Value must include a default constructor (a constructor with no arguments), and implement the `deserialize`, `serialize`, and `visit` methods described in Section 11.3.

11.2. Adding Native Bindings

One can add native bindings to the Scheme environment by implementing a subclass of the abstract class `sisc.nativefun.NativeLibrary`. Such a subclass needs to implement four methods:

public String **getLibraryName**();

Returns the name of this library. The name should also be acceptable for use in filenames.

public float **getLibraryVersion**();

Returns the version of this library.

public sisc.data.Symbol[] **getLibraryBindingNames**();

Returns an array of the names of the bindings exported by this library. Each name is a Scheme symbol.

public Value **getBindingValue**(sisc.data.Symbol name);

Returns the value of a given binding exported by this library.

11.2.1. Native Procedures

It is possible to implement Scheme functions whose behavior is implemented natively in Java code. Many of *SISC*'s procedures are implemented this way. Native procedures extend the `sisc.nativefun.NativeProcedure` abstract class. Working NativeProcedure subclasses must implement the `doApply` method, as described below.

public Value **doApply**(sisc.interpreter.Interpreter interp);

Perform the necessary computations of the NativeProcedure, returning a Value as the result of the procedure call. The arguments to the procedure can be found in the value rib array field, `vlr`, of the Interpreter passed as an argument. The number of arguments to the procedure can be found from the length of the array (`vlr.length`).

If the native procedure wishes to raise an error, it may do so by throwing any Java runtime exception (subclass of `java.lang.Runtime`). For a more descriptive error, one may raise a *SISC* error using any of a number of `error` forms in `sisc.util.Util`. Consult the source of Util or inquire on the sisc-devel mailinglist for assistance.

11.2.2. Fixable Native Procedures

Often it is unnecessary to have access to the full Interpreter context to implement a native procedure. If the arguments to the procedure are sufficient and the procedure is purely functional (causes no side effects), it is recommended that the programmer create a *fixable* native procedure. These native procedures may be inlined into generated code when enabled, allowing much faster execution. In addition, the fixable native procedure interface is simpler to use.

The `FixableProcedure` abstract class consists of five methods which may or may not be subclassed. These five methods correspond to the case of calling the procedure with no, one, two, three, and more than three arguments respectively. Not overriding one of these methods will cause a call to the fixable procedure to throw the invalid number of arguments error to the caller.

public Value **apply**();

Perform the necessary computations of the FixableNativeProcedure, returning a Value as the result of the procedure call. No argument variant.

public Value **apply**(Value v1);

Perform the necessary computations of the FixableNativeProcedure, returning a Value as the result of the procedure call. One argument variant.

public Value **apply**(Value v1, Value v2);

Perform the necessary computations of the FixableNativeProcedure, returning a Value as the result of the procedure call. Two argument variant.

public Value **apply**(Value v1, Value v2, Value v3);

Perform the necessary computations of the FixableNativeProcedure, returning a Value as the result of the procedure call. Three argument variant.

public Value **apply**(Value[] v);

Perform the necessary computations of the FixableNativeProcedure, returning a Value as the result of the procedure call. More than three argument variant.

11.2.3. Indexed Native Libraries

In the most common case, a Library is created to define several bindings, including procedures whose implementations are in Java code. For this common case, a skeleton subclass of `NativeLibrary`, `sisc.nativefun.IndexedLibraryAdapter` is provided. The IndexedLibraryAdapter class provides implementations for all four required NativeLibrary methods, and introduces a new abstract method which must be implemented, called `construct`. In addition, the method `define` is provided.

In an indexed native library, each binding is associated with a Java `int` unique to that binding within the library. The IndexedLibraryAdapter subclass should in its constructor call `define` for each binding provided by the library, according to the contract of the method:

public void **define**(String name, int id);

Register the native binding with the given name, and assign it the given library-unique id.

In implementing the `getBindingValue` method of the `NativeLibrary` class, an `IndexedLibraryAdapter` will call the abstract method `construct` required by the its subclasses:

public sisc.data.Value **construct**(int id);

Return an instance of the indexed binding.

Most frequently, the bindings created in an indexed library are native procedures. In such a case, a second class is created which subclasses `sisc.nativefun.IndexedProcedure`. IndexedProcedure is subclass of `NativeProcedure`. An IndexedProcedure subclass' constructor must call the superconstructor with an `int`, the unique id for that binding. That `int` is stored in the `id` field of `IndexedProcedure`. A subclass can then use the `id` instance variable to dispatch to many native procedures in the body of the `doApply` method required by native procedures.

So, typically, an IndexedNativeLibrary subclass is created whose `construct` method creates instances of IndexedProcedure subclasses. The IndexedNativeLibrary subclass its itself nested in the IndexedProcedure class which it is constructing. See the various indexed libraries in `sisc.modules` for concrete examples.

11.3. Serialization

SISC provides an API for serializing the state of a running Interpreter. The *SISC* heap is a dump of the state of an Interpreter with the necessary code to implement R^{5}RS Scheme, for example. In order to facilitate this serialization, *SISC* Expressions and Values can implement helper methods to define the serialization of the object. If the Expression or Value contains no internal state that need be serialized, the serialization methods may be ignored. If not, the Expression or Value must contain a default (no argument) constructor, and implement the following three methods:

public void **serialize**(sisc.ser.Serializer serializer) throws java.io.IOException;

Serializes the contents of the Expression to the given Serialization context.

public void **deserialize**(sisc.ser.Deserializer deserializer) throws java.io.IOException;

Sets the state of the Expression to the serialized data read from *`deserializer`*.

public boolean **visit**(sisc.util.ExpressionVisitor visitor);

When called, the Expression should call `visitor.visit(n)` on any nested Expressions.

The `Serializer` and `Deserializer` objects implement Java's `java.io.DataOutput` and `java.io.DataInput` interfaces, respectively. This means that you can use any of the write/read functions in those interfaces to serialize the state of your Expression or Value. In addition, a number of methods are provided that are helpful for this domain.

The ExpressionVisitor passed to visit contains only one method, `visit`, which bears the same contract as the `visit` above. When called, an Expression would then call the ExpressionVisitor's `visit` method once for each nested Expression. This method is used during serialization and during printing to detect cycles in data and code structures.

11.3.1. Deserializer methods

public BigInteger **readBigInteger**() throws java.io.IOException;

Reads a BigInteger from the stream.

public BigDecimal **readBigDecimal**() throws java.io.IOException;

Reads a BigDecimal from the stream.

public Class **readClass**() throws java.io.IOException;

Reads a Java Class object from the stream.

public Expression **readExpression**() throws java.io.IOException;

Reads a *SISC* Expression from the stream.

public Expression **readInitializedExpression**() throws java.io.IOException;

Reads a *SISC* Expression from the stream, fully initialized. This method should *only* be used if fields internal to the Expression returned must be available during deserialization.

public sisc.data.Expression[] **readExpressionArray**() throws java.io.IOException;

Reads an array of `Expression`s from the stream.

public sisc.data.Value[] **readValueArray**() throws java.io.IOException;

Reads an array of `Value`s from the stream.

public SymbolicEnvironment **readSymbolicEnvironment**() throws java.io.IOException;

Reads a *SISC* Symbolic Environment from the stream.

11.3.2. Serializer methods

public void **writeBigInteger**(BigInteger bigint) throws java.io.IOException;

Writes a BigInteger to the stream.

public void **writeBigDecimal**(BigDecimal bigdecim) throws java.io.IOException;

Writes a BigDecimal to the stream.

public void **writeClass**(Class clazz) throws java.io.IOException;

Writes a Java Class object to the stream.

public void **writeExpression**(Expression expr) throws java.io.IOException;

Writes a *SISC* Expression to the stream.

public void **writeInitializedExpression**(Expression expr) throws java.io.IOException;

Writes a *SISC* Expression to the stream. This method should *only* be used if fields internal to the Expression returned must be available during deserialization.

public void **writeExpressionArray**(sisc.data.Expression[] ary) throws java.io.IOException;

Writes an array of `Expression`s to the stream.

public void **writeValueArray**(sisc.data.Value[] ary) throws java.io.IOException;

Writes an array of `Value`s to the stream.

public void **writeSymbolicEnvironment**(SymbolicEnvironment e) throws java.io.IOException;

Writes a *SISC* Symbolic Environment to the stream.

Appendix A. Errata

This appendix describes where this manual and the implementation of SISC depart. This section should ideally remain small or empty, as it is the goal of the system to conform to this document, not for this document to describe the idiosyncrasies of the system.

Appendix B. R^5RS Liberties and Violations

This section lists all ways in which *SISC* interprets the R^5RS specification, where the standard is not particular clear. Such interpretations may allow non-portable code to be written and executed on *SISC*. Additionally, all known R^5RS violations are listed. Actual violations are considered *SISC* bugs, and have a high priority for being fixed. Violations of the standard are written in **bold text**.

1. 2.3 - *SISC* allows identifiers to start with '+', '-', or '.' if they cannot be read as numbers.
2. 2.3 - *SISC* uses the reserved characters '[' and ']' as synonyms for '(' and ')' respectively.
3. 2.3 - *SISC* does not raise any warning or error when encountering the reserved characters "[]{}|", and allows "{}|" to appear in identifiers.
4. 4.1.3 - *SISC* does not distinguish between () and the quoted empty list '().
5. 6.2.3 - The standard desires that that operations such as `sqrt` try to provide exact results when given exact arguments. While *SISC* meets the requirement for `sqrt`, it is not clear what other mathematical functions should have this behavior. *SISC* takes no heroic efforts to meet this requirement.
6. 6.2.6 - *SISC* allows radixes other than those specified in the contract for `number->string` and `string->number`. In particular, any radix up to 36 is allowed, and any unsupported value causes *SISC* to revert to base 10.
7. **6.5 - *SISC* currently returns an environment from the `scheme-report-environment` which contains four bindings not specified in R^5RS, bindings which are needed by R^5RS syntactic keywords.**
8. **6.6.2 - *SISC* currently only warns when end-of-file is reached in `read`, rather than signaling an error. The unterminated datum is discarded.**

All of the liberties described above are implemented for the convenience of the programmer. If desired, *strict* R^5RS syntax and semantics may be enabled with the strictR5RSCompliance configuration parameter (see Section 2.4.2), causing *SISC* to raise errors in all R^5RS situations that result in "an error", as well as respect the lexical syntax's reserved characters. When in *strict* compliance mode, the above mentioned deviations no longer apply and *SISC* is entirely R^5RS compliant.

Appendix C. Troubleshooting

This appendix covers issues with running *SISC* in certain environments.

C.1. Kaffe

There is a known limitation with Kaffe, whose default stack size is too small for *SISC* to parse s-expressions. To fix this, either edit the *SISC* startup scripts to pass `-Xss32m`, or set the `JAVAOPT` environment variable to the same.

Appendix D. Backend Details

Note: This appendix is under development

This appendix describes details of *SISC* on particular backends. This is not intended to guide programming. The programmer should code according to the main body of this document. However, this section still describes useful performance tips and limitations of *SISC*'s operation.

D.1. Limits

This appendix lays out the various limits in *SISC* running on a JVM backend. These limits are *not* specifications for an expected set of limits on all platforms, but serve as a real-world guide.

D.1.1. Datastructure Limits

Table D-1. *SISC* Limits

Description	**Limit**
Fixed-point Exact Integers	$-2^{31} < n < +2^{31}-1$
AP Exact Integers	$-2^{(2^{32}-1)} < x < +2^{(2^{32}-1)}-1$.
Inexacts (`32Float`)	See IEEE 754-1985 Floating Point Standard
Inexacts (`64Float`)	See IEEE 754-1985 Floating Point Standard
Inexact Mantissa (`APFloat`)	same as AP Exact Integers
Inexact Exponent (`APFloat`)	same as fixed-point exact integer
Max vector elements	Same as max fixed-point integer
Max string elements	Same as max fixed-point integer (?)
Representable characters	see Section 3.1.2
Maximum formal parameters	Same as max fixed-point integer
Maximum lexical depth	Same as max fixed-point integer
Maximum symbolic-environment bindings	Same as max fixed-point integer (?)
Addressable file size	min of $2^{64}-1$ and operating system limit

Arbitrary precision integers aren't quite arbitrary precision. *SISC* has a hard limit to the number of bits in an exact integer and thus to the range of representable numbers. Exact integers are stored as two's complement signed integers, with a bit limit (including the sign bit) of 2^{32}. This limits the range of representable exact integers to the numbers quoted above.

Likewise, arbitrary precision inexact numbers (when present) have a similar hard limit. The arbitrary precision inexact is constructed with an arbitrary precision exact number with the limits described above as the number's mantissa, and an exponent whose range is equivalent to that of a fixed-point exact integer. The inexact is then then `mantissa*10`$^{\text{exponent}}$.

D.1.2. Symbol Uniqueness

In order to support compilation in multiple threads, on multiple machines, or in multiple times, generated symbols for module bindings, lexical variables, etc. must be *4d-unique*, that is they must be unique across space and time. *SISC* attempts to balance this requirement with the space inefficiency of generating symbols with very long names.

SISC's unique symbols are generated by creating a number of the form `current-time + (random-16-bit-natural*311040000000) + (counter*155520000000)`. The current time variable is only updated when the value of counter reaches 65536. In this manner, two entities that generate the same symbol will only generate a colliding symbol if they generate the symbol on the same system millisecond, with the same counter value, and the same random number, *or*, if one entity happens to generate the symbol with the same millisecond and does so (counter-1)*50 years, (counter-2)*100 years... in the future, and with the same random or (random-1)*100 years, (random-2)*200 years, etc in the future. Only if all of these factors align will a colliding symbol be generated. This is not as unlikely as say a Microsoft GUID or Java VMID number, but it should be sufficiently unlikely. The advantage over other GUID algorithms is that the value produced by *SISC*'s is significantly smaller (and thus does not bloat expanded code).

D.2. Performance and Efficiency considerations

D.2.1. Math

The *SISC* numeric library is most efficient when operating on fixed bitlength numbers. Exact numbers are in their fixed bitlength mode if they are in the representable range for fixed exact integers, as described in Table D-1. Fixed bitlength inexact numbers are only available in the `64Float` and `32Float` libraries. For *SISC* on Java on the x86 32-bit architecture, the `64Float` library is generally more efficient than the `32Float` library, while both are more efficient than the `APFloat` library.

Fixed bitlength exact integers are only used for whole numbers. Rational numbers use arbitrary precision components and thus are less efficient than whole fixed integers.

Arbitrary precision inexact numbers are progressively slower as the bitlength of the mantissa and the scale of the exponent increase. Using the precision constraints can prevent an unbounded increase in the scale of arbitrary precision inexacts which will very rapidly slow calculations.

D.2.2. Strings

At the time of this writing, the Scheme string type can be represented either as a character array, a native string, or simultaneously as both. The character array representation allows efficient, constant time modification of a mutable string (using `string-set!` for example). The native string representation allows efficient output to ports, string comparison, and substring operations.

By default, *SISC* allows the Scheme string to contain both representations simultaneously, ensuring that there is not a costly representation conversion necessary to perform certain operations. However, in this default mode, strings may occupy twice the memory as a string in a single representation. If a program uses many strings or several very large strings, the programmer may wish to create strings that may only be in one representation at any given time. *SISC* provides a parameter to control this behavior.

parameter: (**`compact-string-rep`** `[boolean]`) => #t/#f

This parameter, if set `#t`, will force strings to be represented either as a character array, or as a native string, but never both. If false, simultaneous representations are possible.

D.2.3. Interrupts

Interrupts allow running Scheme code to be forcibly broken from another thread, causing the Scheme code to raise an error. The interrupt signal handling code does add an appreciable overhead (usually between 1-5% depending on the JVM) to execution. It can disabled using the sisc.permitInterrupts system property.

Appendix E. GNU General Public License

E.1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software - to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

1. copyright the software, and
2. offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

E.2. TERMS AND CONDITIONS FOR COPYING,

DISTRIBUTION AND MODIFICATION

E.2.1. Section 0

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a “work based on the Program ” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification ”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

E.2.2. Section 1

You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

E.2.3. Section 2

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License.

Exception:: If the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

E.2.4. Section 3

You may copy and distribute the Program (or a work based on it, under Section 2 in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

E.2.5. Section 4

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

E.2.6. Section 5

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

E.2.7. Section 6

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

E.2.8. Section 7

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

E.2.9. Section 8

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

E.2.10. Section 9

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

E.2.11. Section 10

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

E.2.12. NO WARRANTY Section 11

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

E.2.13. Section 12

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL

DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

E.3. How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index of Functions, Parameters, and Syntax