

# Theme-D Standard Library Reference

Tommi Höynälänmaa

May 15, 2019



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Exceptions</b>	<b>3</b>
<b>3</b>	<b>Numerical Tower</b>	<b>5</b>
<b>4</b>	<b>Module (standard-library core)</b>	<b>7</b>
4.1	Control Structures . . . . .	7
4.1.1	Data Types . . . . .	7
4.1.2	Simple Procedures . . . . .	7
4.2	Command Line . . . . .	14
4.2.1	Simple Procedures . . . . .	14
4.3	Equality Predicates . . . . .	15
4.3.1	Simple Procedures . . . . .	15
4.3.2	Methods . . . . .	20
4.4	Class Membership Predicates . . . . .	20
4.4.1	Data Types . . . . .	20
4.4.2	Simple Procedures . . . . .	20
4.5	Lists, Tuples, and Pairs . . . . .	25
4.5.1	Data Types . . . . .	25
4.5.2	Simple Procedures . . . . .	25
4.5.3	Parametrized Procedures . . . . .	26
4.6	Logical Operations . . . . .	54
4.6.1	Simple Procedures . . . . .	54
4.7	Strings . . . . .	55
4.7.1	Data Types . . . . .	55
4.7.2	Simple Procedures . . . . .	56
4.8	Vectors . . . . .	67
4.8.1	Parametrized Procedures . . . . .	67
4.9	Arithmetic Operations . . . . .	72
4.9.1	Simple Procedures . . . . .	72
4.9.2	Methods . . . . .	102
4.10	Other . . . . .	103
4.10.1	Data Types . . . . .	103
<b>5</b>	<b>Module (standard-library core-forms)</b>	<b>105</b>
5.1	Macros . . . . .	105

<b>6 Module (standard-library bitwise-arithmetic)</b>	<b>115</b>
6.1 Simple Procedures . . . . .	115
<b>7 Module (standard-library promise)</b>	<b>119</b>
7.1 Data Types . . . . .	119
7.2 Macros . . . . .	119
7.3 Parametrized Procedures . . . . .	120
<b>8 Module (standard-library stream)</b>	<b>123</b>
8.1 Data Types . . . . .	123
8.2 Simple Procedures . . . . .	124
8.3 Parametrized Procedures . . . . .	124
<b>9 Module (standard-library iterator)</b>	<b>133</b>
9.1 Data Types . . . . .	133
9.2 Parametrized Procedures . . . . .	133
<b>10 Module (standard-library nonpure-iterator)</b>	<b>141</b>
10.1 Data Types . . . . .	141
10.2 Parametrized Procedures . . . . .	142
<b>11 Module (standard-library object-string-conversion)</b>	<b>151</b>
11.1 Simple Procedures . . . . .	151
11.2 Methods . . . . .	155
<b>12 Module (standard-library text-file-io)</b>	<b>157</b>
12.1 Data Types . . . . .	157
12.2 Simple Procedures . . . . .	157
12.3 Methods . . . . .	166
<b>13 Module (standard-library console-io)</b>	<b>169</b>
13.1 Simple Procedures . . . . .	169
<b>14 Module (standard-library system)</b>	<b>173</b>
14.1 Simple Procedures . . . . .	173
<b>15 Module (standard-library rational)</b>	<b>175</b>
15.1 Data Types . . . . .	175
15.2 Simple Procedures . . . . .	175
15.3 Methods . . . . .	198
<b>16 Module (standard-library real-math)</b>	<b>201</b>
16.1 Data Types . . . . .	201
16.2 Constants . . . . .	201
16.3 Simple Procedures . . . . .	202
16.4 Methods . . . . .	220
<b>17 Module (standard-library complex)</b>	<b>221</b>
17.1 Data Types . . . . .	221
17.2 Simple Procedures . . . . .	221
17.3 Methods . . . . .	254

<b>18 Module (standard-library math)</b>	<b>257</b>
18.1 Data Types . . . . .	257
18.2 Simple Procedures . . . . .	257
18.3 Methods . . . . .	261
<b>19 Module (standard-library extra-math)</b>	<b>267</b>
19.1 Wrapper Procedures for Standard C Functions . . . . .	267
19.2 Other Simple Procedures . . . . .	268
19.3 Methods . . . . .	269
<b>20 Module (standard-library posix-math)</b>	<b>273</b>
20.1 Wrapper Procedures for POSIX C Functions . . . . .	273
20.2 Methods . . . . .	273
<b>21 Module (standard-library matrix)</b>	<b>275</b>
21.1 Data Types . . . . .	275
21.2 Parametrized Procedures . . . . .	275
21.3 Parametrized Methods . . . . .	287
<b>22 Module (standard-library dynamic-list)</b>	<b>297</b>
22.1 Simple Procedures . . . . .	297
<b>23 Module (standard-library singleton)</b>	<b>303</b>
23.1 Data Types . . . . .	303
23.2 Parametrized Procedures . . . . .	303
<b>24 Module (standard-library hash-table)</b>	<b>307</b>
24.1 Data Types . . . . .	307
24.2 Simple Procedures . . . . .	309
24.3 Parametrized Procedures . . . . .	311
<b>25 Module (standard-library hash-table2)</b>	<b>323</b>
25.1 Data Types . . . . .	323
25.2 Simple Procedures . . . . .	324
25.3 Parametrized Procedures . . . . .	327
<b>26 Module (standard-library statprof)</b>	<b>339</b>
26.1 Simple Procedures . . . . .	339



# Chapter 1

## Introduction

Here is an overview for the modules in the Theme-D standard library:

- Module `core` includes basic functionality of the Theme-D environment and it should generally be always included by the user source code.
- Module `core-forms` defines some basic control structures as macros.
- Module `bitwise-arithmetic` implements basic bit operations.
- Module `promise` implements promises (delayed evaluation).
- Module `stream` implements streams.
- Module `iterator` implements purely functional iterators.
- Module `nonpure-iterator` implements nonpure iterators analogous to the pure ones.
- Module `object-string-conversion` implements readable string forms of the Theme-D objects.
- Module `text-file-io` defines primitive classes for input and output ports and basic operations with them.
- Module `console-io` implements console input and output.
- Module `system` implements some OS level functionality.
- Module `rational` implements rational numbers.
- Module `complex` implements complex numbers.
- Module `real-math` implements operations between real and rational values and real scientific functions. Basic numerical operations are defined in `core`.
- Module `math` implements the numerical tower and generic scientific functions.

- Module `extra-math` implements wrapper procedures for many standard C mathematical functions. Some methods using the wrappers are implemented, too.
- Module `posix-math` implements wrapper procedures for some POSIX C mathematical functions not present in the C standard. Some methods using the wrappers are implemented, too.
- Module `matrix` implements matrices.
- Module `dynamic-list` implements dynamically type checked lists.
- Module `singleton` implements singletons.
- Module `hash-table` implements hash tables.
- Module `hash-table2` implements hash tables.
- Module `statprof` provides an interface to the guile `statprof` profiler.

Modules `hash-table`, `extra-math`, and `posix-math` work only for the target platform Guile.

When we document procedures and methods we use the notation

```
proc-name: (arg-type-1 ... arg-type-n) → result-type
```

to indicate that method `method-name` takes arguments of types `arg-type-i` and returns a value of type `result-type`.

When we document methods we use the notation

```
method-name: (arg-type-1 ... arg-type-n) → result-type = procedure
```

that the method is additionally equal to the procedure `procedure`.

The notation

```
method-name: (arg-type-1 ... arg-type-n) → result-type abstract
```

means that the method only raises an exception and its purpose is to help in the static dispatch of the result type. We write “pure” in a procedure definition to indicate that the procedure is pure.



## Chapter 2

# Exceptions

Error handling in Theme-D is based on exceptions that resemble Scheme R7RS [2] exceptions. The module `core` in the standard library defines custom primitive class `<condition>`, which represents Scheme exceptions in the runtime target environment. Note that the exceptions do not have to be objects of class `<condition>`. Exceptions are raised with procedure `raise` (or procedures using `raise`) and exception handlers are defined by forms `guard` (see section 5.1) and `guard-general` (see section 6.11.6 in the language manual). See section 4.1.2 for description of the procedure `raise`.

The Theme-D runtime environment and standard library create runtime environment (RTE) exceptions. An RTE exception is an object of class `<condition>` having *kind* and *info*. *Kind* is a symbol that defines the kind of the exception. *Info* is an association list containing properties (pairs) with a key (a symbol).

When the exception info is enabled the runtime environment adds some extra information into the exceptions it generates. When a Theme-D program is run the flag is initially `#f` and it is set to `#t` when the main procedure is called. The reason for this behaviour is that the Theme-D objects in the exception info could cause ugly output with the default Guile error handler. When the info is disabled then exceptions raised by the Theme-D runtime environment contain only the exception kind (a symbol). Note that this is not true for the exceptions raised by the Scheme code called by the runtime environment. The exception info can be enabled with procedure `enable-rte-exception-info` and disabled with procedure `disable-rte-exception-info` (section 4.1.2).

A Theme-D program can be terminated with procedures `exit` or `raw-exit` (section 4.1.2). In general, you should normally use Theme-D procedure `exit` unless you exit the program in a toplevel expression in a script.



## Chapter 3

# Numerical Tower

The Theme-D Standard Library implements a numerical tower similar to Scheme. Easiest way to import it is to import module (`standard-library math`). Numerical types `<integer>` and `<real>` are built in the Theme-D language and types `<rational>` and `<complex>` are implemented by the standard library.

We say that a rational number is in the *simplified form* if its denominator is positive and the greatest common divisor between the numerator and the denominator is 1. Additionally we require that if the numerator is 0 the denominator is 1.

Terms NaN, inf, and -inf mean the exceptional floating point values (not a number, positive infinity, and negative infinity) defined by the IEEE 754 standard. If your system does not support these the behaviour of the procedures in case they are specified to return an exceptional value is undefined.

Numbers belonging to classes `<integer>` and `<rational>` are called *exact* and numbers belonging to classes `<real>` and `<complex>` *inexact*. The terms *exact 0* and *exact 1* mean the integer and rational values 0 and 1.

When the results of the real math functions in module `real-math` are complex or not defined these procedures usually return an exceptional value. When the optimized real functions are used (this is on by default) the complex value is not actually computed but the exceptional values are returned directly.

When applied to integer or real numbers the equality predicate `equal?` follows the rules for Scheme predicate `eqv?` in the R7RS standard [2]. The equality predicate `=` follows the rules for Scheme predicate `=` in R7RS for integer and real numbers. Predicate `=` returns `#t` when its arguments are numerically equal.



## Chapter 4

# Module (standard-library core)

### 4.1 Control Structures

#### 4.1.1 Data Types

*Data type name:* <condition>

*Type:* <class>

*Description:* The data type for Scheme conditions

*Data type name:* <rte-exception-kind>

*Type:* <class>

*Description:* The kind of an exception.

*Definition:* <symbol>

*Data type name:* <rte-exception-info>

*Type:* :union

*Description:* Auxiliary information for an exception.

*Definition:* (:alist <symbol> <object>)

#### 4.1.2 Simple Procedures

**exit**

*Syntax:*

(exit exit-code)

*Arguments:*

Name: `exit-code`  
Type: `<integer>`  
Description: The exit code passed to the operating system

No result value.

*Purity of the procedure:* nonpure

Procedure `exit` terminates a program. The exit code given as an argument is passed to the operating system. This procedure actually rows an RTE exception with kind `exit` and property `i-exit-code` containing the exit code. The Theme-D main program exception handler terminates the program with the exit code when it receives this exception.

## `raw-exit`

*Syntax:*

```
(raw-exit exit-code)
```

*Arguments:*

Name: `exit-code`  
Type: `<integer>`  
Description: The exit code passed to the operating system

No result value.

*Purity of the procedure:* nonpure

This procedure calls the Guile procedure `exit` in order to terminate the program. In general, you should normally use Theme-D procedure `exit` unless you exit the program in a toplevel expression in a script.

## `raise`

*Syntax:*

```
(raise exception-object)
```

*Arguments:*

Name: `exception-object`  
Type: `<object>`  
Description: The exception object to be raised

No result value.

*Purity of the procedure:* pure

Procedure **raise** raises an exception. Exceptions can be caught with **guard-general** and **guard** forms, see the language manual and section 5.1 in this guide. The semantics of **guard** and **raise** are similar to their semantics in Scheme.

## **enable-rte-exception-info**

*Syntax:*

```
(enable-rte-exception-info)
```

No arguments.

No result value.

*Purity of the procedure:* pure

This procedure sets the exception info flag on. See chapter 2.

## **disable-rte-exception-info**

*Syntax:*

```
(disable-rte-exception-info)
```

No arguments.

No result value.

*Purity of the procedure:* pure

This procedure sets the exception info flag off. See chapter 2.

## **make-rte-exception**

*Syntax:*

```
(make-rte-exception s-kind al-info)
```

*Arguments:*

Name: `s-kind`  
 Type: `<rte-exception-kind>`  
 Description: The kind of the exception

Name: `al-info`  
 Type: `<rte-exception-info>`  
 Description: Auxiliary information for the exception

*Result value:* An exception object

*Result type:* `<condition>`

*Purity of the procedure:* pure

**rte-exception?***Syntax:*

```
(rte-exception? x)
```

*Arguments:*

Name: `x`  
 Type: `<object>`  
 Description: An object

*Result value:* `#t` iff the object is an RTE exception

*Result type:* `<boolean>`

*Purity of the procedure:* pure

**get-rte-exception-kind0***Syntax:*

```
(get-rte-exception-kind0 exc)
```

*Arguments:*

Name: `exc`  
 Type: `<condition>`  
 Description: A condition object



*Result value:* The kind of the RTE exception

*Result type:* (:alt-maybe <rte-exception-kind>)

*Purity of the procedure:* pure

This procedure returns #f if the argument is not a RTE exception.

## get-rte-exception-kind

*Syntax:*

```
(get-rte-exception-kind exc)
```

*Arguments:*

Name: exc

Type: <condition>

Description: An RTE exception

*Precondition:* The argument has to be a RTE exception.

*Result value:* The kind of the RTE exception

*Result type:* <rte-exception-kind>

*Purity of the procedure:* pure

## get-rte-exception-info0

*Syntax:*

```
(get-rte-exception-info0 exc)
```

*Arguments:*

Name: exc

Type: <condition>

Description: A condition object

*Result value:* The RTE exception auxiliary info

*Result type:* (:alt-maybe <rte-exception-info>)

*Purity of the procedure:* pure

This procedure returns `#f` if the argument is not a RTE exception.

## `get-rte-exception-info`

*Syntax:*

```
(get-rte-exception-info exc)
```

*Arguments:*

Name: `exc`  
Type: `<condition>`  
Description: An RTE exception

*Precondition:* The argument has to be a RTE exception.

*Result value:* The RTE exception info

*Result type:* `<rte-exception-info>`

*Purity of the procedure:* pure

## `make-simple-exception`

*Syntax:*

```
(make-simple-exception s-kind)
```

*Arguments:*

Name: `s-kind`  
Type: `<symbol>`  
Description: The kind of the exception

*Result value:* An RTE exception object

*Result type:* `<condition>`

*Purity of the procedure:* pure

The result is an RTE exception object whose kind is the value of the argument and info is `null`.

## `raise-simple`

*Syntax:*

```
(raise-simple s-kind)
```

*Arguments:*

Name: `s-kind`  
Type: `<symbol>`  
Description: The kind of the exception

No result value.

*Purity of the procedure:* pure

This procedure raises a simple exception and the procedure never returns. The exception is created with procedure `make-simple-exception`.

## `make-numerical-overflow`

*Syntax:*

```
(make-numerical-overflow s-procedure)
```

*Arguments:*

Name: `s-procedure`  
Type: `<symbol>`  
Description: The name of the procedure from which the exception is raised

*Result value:* An RTE exception object

*Result type:* `<condition>`

*Purity of the procedure:* pure

The result is an RTE exception object whose kind is `numerical-overflow` and the value corresponding to key `s-proc-name` is the value the argument `s-procedure`.

## `%debug-print`

*Syntax:*

```
(%debug-print x-message)
```

*Arguments:*

Name: `x-message`  
Type: `<object>`  
Description: The object to be printed

No result value.

*Purity of the procedure:* nonpure

This procedure prints the argument with Scheme procedure `display` and flushes the ports. This procedure may be called body and program toplevel without prelinking any module bodies.

## `raise-numerical-overflow`

*Syntax:*

```
(raise-numerical-overflow s-procedure)
```

*Arguments:*

Name: `s-procedure`  
Type: `<symbol>`  
Description: The name of the procedure from which the exception is raised

No result value.

*Purity of the procedure:* pure

This procedure raises a numerical overflow exception and the procedure never returns.

## 4.2 Command Line

### 4.2.1 Simple Procedures

#### `command-line-arguments`

*Syntax:*

```
(command-line-arguments)
```

No arguments.

*Result value:* List of command line arguments

*Result type:* (:uniform-list <string>)

*Purity of the procedure:* pure

## 4.3 Equality Predicates

### 4.3.1 Simple Procedures

=

Procedure = is an alias to procedure equal?.

**boolean=?**

*Syntax:*

```
(boolean=? object1 object2)
```

*Arguments:*

Name: object1

Type: <boolean>

Description: A boolean value to be compared

Name: object2

Type: <boolean>

Description: A boolean value to be compared

*Result value:* #t iff object1 is equal to object2

*Result type:* <boolean>

*Purity of the procedure:* pure

**character=?**

*Syntax:*

```
(character=? object1 object2)
```

*Arguments:*

Name: `object1`  
Type: `<character>`  
Description: A character to be compared

Name: `object2`  
Type: `<character>`  
Description: A character to be compared

*Result value:* `#t` iff `object1` is equal to `object2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `integer=?`

*Syntax:*

`(integer=? object1 object2)`

*Arguments:*

Name: `object1`  
Type: `<integer>`  
Description: An integer value to be compared

Name: `object2`  
Type: `<integer>`  
Description: An integer value to be compared

*Result value:* `#t` iff `object1` is equal to `object2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `integer=`

*Syntax:*

`(integer= object1 object2)`

*Arguments:*

Name: `object1`  
 Type: `<integer>`  
 Description: An integer value to be compared

Name: `object2`  
 Type: `<integer>`  
 Description: An integer value to be compared

*Result value:* `#t` iff `object1` is numerically equal to `object2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

### `integer-real=`

*Syntax:*

`(integer-real= object1 object2)`

*Arguments:*

Name: `object1`  
 Type: `<integer>`  
 Description: An integer value to be compared

Name: `object2`  
 Type: `<real>`  
 Description: A real value to be compared

*Result value:* `#t` iff `object1` is numerically equal to `object2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

### `real=?`

*Syntax:*

`(real=? object1 object2)`

*Arguments:*

Name: `object1`  
Type: `<real>`  
Description: A real value to be compared

Name: `object2`  
Type: `<real>`  
Description: A real value to be compared

*Result value:* `#t` iff `object1` is equal to `object2`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `real=`

*Syntax:*

```
(real= object1 object2)
```

*Arguments:*

Name: `object1`  
Type: `<real>`  
Description: A real value to be compared

Name: `object2`  
Type: `<real>`  
Description: A real value to be compared

*Result value:* `#t` iff `object1` is numerically equal to `object2`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `real-integer=`

*Syntax:*

```
(real-integer= object1 object2)
```

*Arguments:*

Name: `object1`



Type: `<real>`  
Description: A real value to be compared

Name: `object2`  
Type: `<integer>`  
Description: An integer value to be compared

*Result value:* `#t` iff `object1` is numerically equal to `object2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## **string=?**

*Syntax:*

```
(string=? object1 object2)
```

*Arguments:*

Name: `object1`  
Type: `<string>`  
Description: A string to be compared

Name: `object2`  
Type: `<string>`  
Description: A string to be compared

*Result value:* `#t` iff `object1` is equal to `object2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

This procedure compares the contents of the argument strings.

## **symbol=?**

*Syntax:*

```
(symbol=? object1 object2)
```

*Arguments:*

Name: `object1`

Type: <symbol>  
 Description: A symbol to be compared

Name: object2  
 Type: <symbol>  
 Description: A symbol to be compared

*Result value:* #t iff object1 is equal to object2  
*Result type:* <boolean>

*Purity of the procedure:* pure

### 4.3.2 Methods

equal?: (<object> <object>) → <boolean> pure = equal-values?  
 equal?: (<boolean> <boolean>) → <boolean> pure = boolean=?  
 equal?: (<integer> <integer>) → <boolean> pure = integer=?  
 equal?: (<real> <real>) → <boolean> pure = real=?  
 equal?: (<symbol> <symbol>) → <boolean> pure = symbol=?  
 equal?: (<string> <string>) → <boolean> pure = string=?  
 equal?: (<character> <character>) → <boolean> pure = character=?

=: (<integer> <integer>) → <boolean> pure = integer=  
 =: (<real> <real>) → <boolean> pure = real=  
 =: (<integer> <real>) → <boolean> pure = integer-real=  
 =: (<real> <integer>) → <boolean> pure = real-integer=

## 4.4 Class Membership Predicates

### 4.4.1 Data Types

*Data type name:* <type-predicate>  
*Type:* :procedure  
*Description:* The data type for type membership predicates

### 4.4.2 Simple Procedures

boolean?

*Syntax:*

(boolean? object)

*Arguments:*

Name: `object`  
Type: `<object>`  
Description: An object to be tested

*Result value:* `#t` iff `object` is an instance of `<boolean>`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `character?`

*Syntax:*

`(character? object)`

*Arguments:*

Name: `object`  
Type: `<object>`  
Description: An object to be tested

*Result value:* `#t` iff `object` is an instance of `<character>`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `eof?`

*Syntax:*

`(eof? obj)`

*Arguments:*

Name: `obj`  
Type: `<object>`  
Description: An arbitrary object

*Result value:* `#t` iff `obj` is the eof object  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## integer?

*Syntax:*

(integer? object)

*Arguments:*

Name: object  
Type: <object>  
Description: An object to be tested

*Result value:* #t iff object is an instance of <integer>

*Result type:* <boolean>

*Purity of the procedure:* pure

## null?

*Syntax:*

(null? object)

*Arguments:*

Name: object  
Type: <object>  
Description: An object to test

*Result value:* #t iff object is null

*Result type:* <boolean>

*Purity of the procedure:* pure

## not-null?

*Syntax:*

(not-null? object)

*Arguments:*

Name: object  
Type: <object>  
Description: An object to test

*Result value:* #t iff object is not null

*Result type:* <boolean>

*Purity of the procedure:* pure

## pair?

*Syntax:*

(pair? object)

*Arguments:*

Name: object  
Type: <object>  
Description: An object to be tested

*Result value:* #t iff object is an instance of <pair>

*Result type:* <boolean>

*Purity of the procedure:* pure

This procedure returns #t for any pair.

## real?

*Syntax:*

(real? object)

*Arguments:*

Name: object  
Type: <object>  
Description: An object to be tested

*Result value:* #t iff `object` is an instance of `<real>`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## string?

*Syntax:*

`(string? object)`

*Arguments:*

Name: `object`

Type: `<object>`

Description: An object to be tested

*Result value:* #t iff `object` is an instance of `<string>`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## symbol?

*Syntax:*

`(symbol? object)`

*Arguments:*

Name: `object`

Type: `<object>`

Description: An object to be tested

*Result value:* #t iff `object` is an instance of `<symbol>`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## 4.5 Lists, Tuples, and Pairs

### 4.5.1 Data Types

*Data type name:* `:alist`

*Type:* `<param-logical-type>`

*Number of type parameters:* 2

*Description:* An association list

*Data type name:* `:nonempty-alist`

*Type:* `<param-logical-type>`

*Number of type parameters:* 2

*Description:* An association list containing at least one element

*Data type name:* `:maybe`

*Type:* `<param-logical-type>`

*Number of type parameters:* 1

*Description:* A value that is either `null` or an instance of the component type

*Data type name:* `:nonempty-uniform-list`

*Type:* `<param-logical-type>`

*Number of type parameters:* 1

*Description:* A uniform list with at least one element

*Data type name:* `<list>`

*Type:* `:union`

*Description:* A list consisting of any objects

*Data type name:* `<nonempty-list>`

*Type:* `:pair`

*Description:* A nonempty list consisting of any objects

*Data type name:* `<pair>`

*Type:* `:pair`

*Description:* A pair consisting of any objects

### 4.5.2 Simple Procedures

#### `length`

*Syntax:*

`(length lst)`

*Arguments:*

Name: `lst`  
Type: `(:uniform-list <object>)`  
Description: A list

*Result value:* Number of elements in the list  
*Result type:* `<integer>`

*Purity of the procedure:* pure

### 4.5.3 Parametrized Procedures

#### `car`

*Syntax:*

`(car pair)`

*Type parameters:* `%type1, %type2`

*Arguments:*

Name: `pair`  
Type: `(:pair %type1 %type2)`  
Description: A pair

*Result value:* The first element of the pair  
*Result type:* `%type1`

*Purity of the procedure:* pure

#### `cdr`

*Syntax:*

`(cdr pair)`

*Type parameters:* `%type1, %type2`

*Arguments:*

Name: `pair`  
Type: `(:pair %type1 %type2)`



Description: A pair

*Result value:* The second element of the pair

*Result type:* %type2

*Purity of the procedure:* pure

## gen-car

*Syntax:*

```
(gen-car pair)
```

*Type parameters:* %type1, %type2

*Arguments:*

Name: pair

Type: (:union (:pair %type1 %type2) <null>)

Description: A pair

*Result value:* The first element of the pair

*Result type:* %type1

*Purity of the procedure:* pure

If the argument is `null` an exception is raised.

## gen-cdr

*Syntax:*

```
(gen-cdr pair)
```

*Type parameters:* %type1, %type2

*Arguments:*

Name: pair

Type: (:union (:pair %type1 %type2) <null>)

Description: A pair

*Result value:* The second element of the pair

*Result type:* %type2

*Purity of the procedure:* pure

If the argument is `null` an exception is raised.

## **cons**

*Syntax:*

```
(cons first second)
```

*Type parameters:* %type1, %type2

*Arguments:*

Name: `first`

Type: %type1

Description: The first object of the new pair

Name: `second`

Type: %type2

Description: The second object of the new pair

*Result value:* A pair with values `first` and `second`

*Result type:* (:pair %type1 %type2)

*Purity of the procedure:* pure

## **list**

*Syntax:*

```
(list item-1 ... item-n)
```

*Type parameters:* %arglist

*Arguments:*

Name: `item-k`

Type:  $t_k$

Description: A list item

*Result value:* A list constructed from the arguments

*Result type:* %arglist

*Purity of the procedure:* pure

Metavariable  $t_k$  is the type of `item-k` for each  $k$ .. Type variable `%arglist` is equivalent to `(:tuple t1 ... tn)`.

## drop

*Syntax:*

```
(drop lst count)
```

*Type parameters:* `%type`

*Arguments:*

Name: `lst`  
Type: `(:uniform-list %type)`  
Description: A list

Name: `count`  
Type: `<integer>`  
Description: Number of elements to be dropped

*Result value:* A list constructed by dropping away the first `count` elements of list `lst`

*Result type:* `(:uniform-list %type)`

*Purity of the procedure:* pure

If `count` is larger than the length of `lst` an exception is raised.

## drop-right

*Syntax:*

```
(drop-right lst count)
```

*Type parameters:* `%type`

*Arguments:*

Name: `lst`  
Type: `(:uniform-list %type)`  
Description: A list

Name: `count`  
Type: `<integer>`  
Description: Number of elements to be dropped

*Result value:* A list constructed by dropping away the last `count` elements of list `lst`

*Result type:* `(:uniform-list %type)`

*Purity of the procedure:* pure

If `count` is larger than the length of `lst` an exception is raised.

## take

*Syntax:*

```
(take lst count)
```

*Type parameters:* `%type`

*Arguments:*

Name: `lst`  
Type: `(:uniform-list %type)`  
Description: A list

Name: `count`  
Type: `<integer>`  
Description: Number of elements to be taken

*Result value:* A list containing the first `count` elements of list `lst`

*Result type:* `(:uniform-list %type)`

*Purity of the procedure:* pure

If `count` is larger than the length of `lst` an exception is raised.

## take-right

*Syntax:*

```
(take-right lst count)
```

*Type parameters:* `%type`

*Arguments:*

Name: `lst`  
Type: `(:uniform-list %type)`  
Description: A list

Name: `count`  
Type: `<integer>`  
Description: Number of elements to be taken

*Result value:* A list containing the last `count` elements of list `lst`

*Result type:* `(:uniform-list %type)`

*Purity of the procedure:* pure

If `count` is larger than the length of `lst` an exception is raised.

## last

*Syntax:*

```
(last lst)
```

*Type parameters:* `%type`

*Arguments:*

Name: `lst`  
Type: `(:nonempty-uniform-list %type)`  
Description: A nonempty list

*Result value:* The last element of the list `lst`

*Result type:* `%type`

*Purity of the procedure:* pure

## for-each

*Syntax:*

```
(for-each proc lst-1 ... lst-n)
```

*Type parameters:* `%arglist`

*Arguments:*

Name: `proc`  
 Type: `(:procedure ((splice %arglist)) <none> nonpure)`  
 Description: A procedure to apply

Name: `lst-k`  
 Type: `(:uniform-list  $t_k$ )`  
 Description: A list to take arguments from

No result value.

*Purity of the procedure:* nonpure

The semantics resembles Scheme `for-each`. Procedure `proc` is applied to the  $j$ th elements of the `lst-k`'s for each  $j = 1, \dots, m$  (in this order) where  $m$  is the minimum of the lengths of `lst-k`'s. You may use a procedure with result type `<none>` as the first argument to `for-each`. The value of the type parameter `%arglist` is a tuple type consisting of types  $t_k$ ,  $k = 1, \dots, n$ . Procedure `proc` takes arguments with types  $t_k$ ,  $k = 1, \dots, n$ .

**for-each1***Syntax:*

`(for-each1 proc lst)`

*Type parameters:* `%arg-type`

*Arguments:*

Name: `proc`  
 Type: `(:procedure (%arg-type) <none> nonpure)`  
 Description: A procedure to apply

Name: `lst`  
 Type: `(:uniform-list %arg-type)`  
 Description: Values to which the procedure is applied

No result value.

*Purity of the procedure:* nonpure

The semantics resembles Scheme `for-each`. You may use a procedure with result type `<none>` as the first argument to `for-each`.

**map***Syntax:*

```
(map proc lst-1 ... lst-n)
```

*Type parameters:* %arglist, %result-type*Arguments:*Name: `proc`Type: `(:procedure ((splice %arglist)) %result-type pure)`

Description: A procedure to apply

Name: `lst-k`Type: `(:uniform-list  $t_k$ )`

Description: Lists to take arguments from

*Result value:* A list constructed by applying `proc` to the  $j$ th elements of the `lst-k`'s for each  $j = 1, \dots, m$  where  $m$  is the minimum of the list lengths*Result type:* `(:uniform-list %result-type)`*Purity of the procedure:* pure

The semantics resembles Scheme `map`. The value of the type parameter `%arglist` is a tuple type consisting of types  $t_k$ ,  $k = 1, \dots, n$ . Note that you cannot use procedure with result type `<none>` as the first argument of `map`.

**map1***Syntax:*

```
(map1 proc lst)
```

*Type parameters:* %arg-type, %result-type*Arguments:*Name: `proc`Type: `(:procedure (%arg-type) %result-type pure)`

Description: A procedure to apply

Name: `lst`Type: `(:uniform-list %arg-type)`

Description: Values to which the procedure is applied

*Result value:* A list constructed by applying `proc` to the elements of list `lst`  
*Result type:* `(:uniform-list %result-type)`

*Purity of the procedure:* pure

The semantics resembles Scheme `map`. Note that you cannot use procedure with result type `<none>` as the first argument of `map1`.

## map-nonpure

*Syntax:*

```
(map-nonpure proc lst-1 ... lst-n)
```

*Type parameters:* `%arglist`, `%result-type`

*Arguments:*

Name: `proc`  
 Type: `(:procedure ((splice %arglist)) %result-type nonpure)`  
 Description: A procedure to apply

Name: `lst-k`  
 Type: `(:uniform-list  $t_k$ )`  
 Description: A list to take arguments from

*Result value:* A list constructed by applying `proc` to the  $j$ th elements of the `lst-k`'s for each  $j = 1, \dots, m$  where  $m$  is the minimum of the list lengths

*Result type:* `(:uniform-list %result-type)`

*Purity of the procedure:* nonpure

The semantics of `map-nonpure` resemble `map` except `proc` may be nonpure and the applications of `proc` are guaranteed to be done in the order of increasing  $j$ . The value of the type parameter `%arglist` is a tuple type consisting of types  $t_k$ ,  $k = 1, \dots, n$ . Note that you cannot use procedure with result type `<none>` as the first argument of `map-nonpure`.

## map-nonpure1

*Syntax:*

```
(map-nonpure1 proc lst)
```

*Type parameters:* `%arg-type`, `%result-type`



*Arguments:*

Name: `proc`  
 Type: `(:procedure (%arg-type) %result-type nonpure)`  
 Description: A procedure to apply

Name: `lst`  
 Type: `(:uniform-list %arg-type)`  
 Description: Values to which the procedure is applied

*Result value:* A list constructed by applying `proc` to the elements of list `lst`

*Result type:* `(:uniform-list %result-type)`

*Purity of the procedure:* `nonpure`

The semantics resembles Scheme `map`. Note that you cannot use procedure with result type `<none>` as the first argument of `map-nonpure1`.

## and-map?

*Syntax:*

```
(and-map? proc lst-1 ... lst-n)
```

*Type parameters:* `%arglist`

*Arguments:*

Name: `proc`  
 Type: `(:procedure ((splice %arglist)) <boolean> pure)`  
 Description: A procedure to apply

Name: `lst-k`  
 Type: `(:uniform-list  $t_k$ )`  
 Description: Lists to take arguments from

*Result value:* `#t` iff `proc` returns `#t` for each elementwise application to lists `lst-k`

*Result type:* `<boolean>`

*Purity of the procedure:* `pure`

Note that if any of the applications of `proc` returns `#f` the rest of the elements are not evaluated. If the lengths of the lists are different the number of evaluations is the length of the shortest list. If all the argument lists are `null` return `#t`. The value of the type parameter `%arglist` is a tuple type consisting of types  $t_k$ ,  $k = 1, \dots, n$ . Procedure `proc` takes arguments with types  $t_k$ ,

$k = 1, \dots, n.$

## and-map1?

*Syntax:*

```
(and-map1? proc lst)
```

*Type parameters:* %argtype

*Arguments:*

Name: `proc`  
 Type: `(:procedure (%argtype)) <boolean> pure`  
 Description: A procedure to apply

Name: `lst`  
 Type: `(:uniform-list %argtype)`  
 Description: A list to take arguments from

*Result value:* `#t` iff `proc` returns `#t` for each application to the elements of list `lst`

*Result type:* `<boolean>`

*Purity of the procedure:* `pure`

Note that if any of the applications of `proc` returns `#f` the rest of the elements are not evaluated. If `lst` is `null` return `#t`.

## and-map-nonpure?

*Syntax:*

```
(and-map-nonpure? proc lst-1 ... lst-n)
```

*Type parameters:* %arglist

*Arguments:*

Name: `proc`  
 Type: `(:procedure ((splice %arglist)) <boolean> nonpure)`  
 Description: A procedure to apply

Name: `lst-k`  
 Type: `(:uniform-list  $t_k$ )`

Description: Lists to take arguments from

*Result value:* #t iff `proc` returns #t for each elementwise application to lists `lst-k`

*Result type:* <boolean>

*Purity of the procedure:* nonpure

This procedure is similar to `and-map?` except that `proc` may have side effects.

### `and-map-nonpure1?`

*Syntax:*

```
(and-map-nonpure1? proc lst)
```

*Type parameters:* %argtype

*Arguments:*

Name: `proc`  
 Type: (:procedure (%argtype)) <boolean> nonpure)  
 Description: A procedure to apply

Name: `lst`  
 Type: (:uniform-list %argtype)  
 Description: A list to take arguments from

*Result value:* #t iff `proc` returns #t for each application to the elements of list `lst`

*Result type:* <boolean>

*Purity of the procedure:* nonpure

This procedure is similar to `and-map1?` except that `proc` may have side effects.

### `or-map?`

*Syntax:*

```
(or-map? proc lst-1 ... lst-n)
```

*Type parameters:* %arglist

*Arguments:*

Name: `proc`  
 Type: `(:procedure ((splice %arglist)) <boolean> pure)`  
 Description: A procedure to apply

Name: `lst-k`  
 Type: `(:uniform-list  $t_k$ )`  
 Description: Lists to take arguments from

*Result value:* `#t` iff `proc` returns `#t` for any elementwise application to lists `lst-k`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

The value of the type parameter `%arglist` is a tuple type consisting of types  $t_k$ ,  $k = 1, \dots, n$ . Procedure `proc` takes arguments with types  $t_k$ ,  $k = 1, \dots, n$ . Note that if any of the applications of `proc` returns `#t` the rest of the elements are not evaluated. If the lengths of the lists are different the number of evaluations is the length of the shortest list. If all the argument lists are `null` return `#f`.

**or-map1?***Syntax:*

```
(or-map1? proc lst)
```

*Type parameters:* `%argtype`

*Arguments:*

Name: `proc`  
 Type: `(:procedure (%argtype)) <boolean> pure)`  
 Description: A procedure to apply

Name: `lst`  
 Type: `(:uniform-list %argtype)`  
 Description: A list to take arguments from

*Result value:* `#t` iff `proc` returns `#t` for some application to the elements of list `lst`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

Note that if any of the applications of `proc` returns `#t` the rest of the elements

are not evaluated. If `lst` is `null` return `#f`.

## or-map-nonpure?

*Syntax:*

```
(or-map-nonpure? proc lst-1 ... lst-n)
```

*Type parameters:* `%arglist`

*Arguments:*

Name: `proc`  
 Type: `(:procedure ((splice %arglist)) <boolean> nonpure)`  
 Description: A procedure to apply

Name: `lst-k`  
 Type: `(:uniform-list  $t_k$ )`  
 Description: Lists to take arguments from

*Result value:* `#t` iff `proc` returns `#t` for any elementwise application to lists `lst-k`

*Result type:* `<boolean>`

*Purity of the procedure:* `nonpure`

This procedure is similar to `or-map?` except that `proc` may have side effects.

## or-map-nonpure1?

*Syntax:*

```
(or-map-nonpure1? proc lst)
```

*Type parameters:* `%argtype`

*Arguments:*

Name: `proc`  
 Type: `(:procedure (%argtype)) <boolean> nonpure)`  
 Description: A procedure to apply

Name: `lst`  
 Type: `(:uniform-list %argtype)`  
 Description: A list to take arguments from

*Result value:* `#t` iff `proc` returns `#t` for some application to the elements of list `lst`

*Result type:* `<boolean>`

*Purity of the procedure:* nonpure

This procedure is similar to `or-map1?` except that `proc` may have side effects.

## map-car

*Syntax:*

```
(map-car lst)
```

*Type parameters:* `%arglist`

*Arguments:*

Name: `lst`

Type: `(:tuple (:nonempty-uniform-list  $t_1$ ) ... (:nonempty-uniform-list  $t_n$ ))`

Description: Lists to take arguments from

*Result value:* A list constructed by taking the first element of each component list of `lst`

*Result type:* `%arglist`

*Purity of the procedure:* pure

The value of the type parameter `%arglist` is a tuple type consisting of types  $t_k$ ,  $k = 1, \dots, n$ .

## map-cdr

*Syntax:*

```
(map-cdr lst)
```

*Type parameters:* `%arglist`

*Arguments:*

Name: `lst`

Type: `(:tuple (:nonempty-uniform-list  $t_1$ ) ... (:nonempty-uniform-list  $t_n$ ))`

Description: Lists to take arguments from

*Result value:* A list constructed by taking the tail of each component list of `lst`

*Result type:* `(type-loop %type %arglist (:uniform-list %type))`

*Purity of the procedure:* pure

The value of the type parameter `%arglist` is a tuple type consisting of types  $t_k$ ,  $k = 1, \dots, n$ .

## assoc-general

*Syntax:*

```
(assoc-general key association-list default my-eq?)
```

*Type parameters:* `%type1`, `%type2`, `%default`

*Arguments:*

Name: `key`

Type: `%type1`

Description: the key to be searched

Name: `association-list`

Type: `(:alist %type1 %type2)`

Description: the association list to be searched

Name: `default`

Type: `%default`

Description: the value returned if no association is found

Name: `my-eq?`

Type: `(:procedure (%type1 %type1) <boolean> pure)`

Description: the equivalence predicate to be used in the search

*Result value:* The result of the search

*Result type:* `(:union (:pair %type1 %type2) %default)`

*Purity of the procedure:* pure

The association list `association-list` is searched for `key`. If `key` is found return the first association having the key. Otherwise return `default`. The keys are compared with the equivalence predicate `my-eq?`.

**ASSOC**

*Syntax:*

```
(assoc key association-list default)
```

*Type parameters:* %type1, %type2, %default

*Arguments:*

Name: **key**

Type: %type1

Description: the key to be searched

Name: **association-list**

Type: (:alist %type1 %type2)

Description: the association list to be searched

Name: **default**

Type: %default

Description: the value returned if no association is found

*Result value:* The result of the search

*Result type:* (:union (:pair %type1 %type2) %default)

*Purity of the procedure:* pure

The association list **association-list** is searched for **key**. If **key** is found return the first association having the key. Otherwise return **default**. The keys are compared with the equivalence predicate **equal?**.

**assoc-values**

*Syntax:*

```
(assoc-values key association-list default)
```

*Type parameters:* %type1, %type2, %default

*Arguments:*

Name: **key**

Type: %type1

Description: the key to be searched

Name: **association-list**



Type: `(:alist %type1 %type2)`  
 Description: the association list to be searched

Name: `default`  
 Type: `%default`  
 Description: the value returned if no association is found

*Result value:* The result of the search

*Result type:* `(:union (:pair %type1 %type2) %default)`

*Purity of the procedure:* pure

The association list `association-list` is searched for `key`. If `key` is found return the first association having the key. Otherwise return `default`. The keys are compared with the equivalence predicate `equal-values?`.

## assoc-objects

*Syntax:*

```
(assoc-objects key association-list default)
```

*Type parameters:* `%type1`, `%type2`, `%default`

*Arguments:*

Name: `key`  
 Type: `%type1`  
 Description: the key to be searched

Name: `association-list`  
 Type: `(:alist %type1 %type2)`  
 Description: the association list to be searched

Name: `default`  
 Type: `%default`  
 Description: the value returned if no association is found

*Result value:* The result of the search

*Result type:* `(:union (:pair %type1 %type2) %default)`

*Purity of the procedure:* pure

The association list `association-list` is searched for `key`. If `key` is found return the first association having the key. Otherwise return `default`. The keys are compared with the equivalence predicate `equal-objects?`.

**assoc-contents***Syntax:*`(assoc-contents key association-list default)`*Type parameters:* %type1, %type2, %default*Arguments:*Name: **key**

Type: %type1

Description: the key to be searched

Name: **association-list**

Type: (:alist %type1 %type2)

Description: the association list to be searched

Name: **default**

Type: %default

Description: the value returned if no association is found

*Result value:* The result of the search*Result type:* (:union (:pair %type1 %type2) %default)*Purity of the procedure:* pure

The association list **association-list** is searched for **key**. If **key** is found return the first association having the key. Otherwise return **default**. The keys are compared with the equivalence predicate **equal-contents?**.

**general-alist-delete***Syntax:*`(general-alist-delete key alist my-eq?)`*Type parameters:* %type1, %type2*Arguments:*Name: **key**

Type: %type1

Description: the key to be searched

Name: **alist**

Type: (:alist %type1 %type2)  
 Description: the association list to be searched

Name: my-eq?  
 Type: (:procedure (%type1 %type1) <boolean> pure)  
 Description: the equivalence predicate to be used in the search

*Result value:* The association list obtained by removing all bindings for key `key` from `alist`

*Result type:* (:alist %type1 %type2)

*Purity of the procedure:* pure

The keys are compared with the equivalence predicate `my-eq?`.

## alist-delete

*Syntax:*

```
(alist-delete key alist)
```

*Type parameters:* %type1, %type2

*Arguments:*

Name: key  
 Type: %type1  
 Description: the key to be searched

Name: alist  
 Type: (:alist %type1 %type2)  
 Description: the association list to be searched

*Result value:* The association list obtained by removing all bindings for key `key` from `alist`

*Result type:* (:alist %type1 %type2)

*Purity of the procedure:* pure

The keys are compared with the equivalence predicate `equal?`.

## value-alist-delete

*Syntax:*

```
(value-alist-delete key alist)
```

*Type parameters:* %type1, %type2

*Arguments:*

Name: **key**  
 Type: %type1  
 Description: the key to be searched

Name: **alist**  
 Type: (:alist %type1 %type2)  
 Description: the association list to be searched

*Result value:* The association list obtained by removing all bindings for key **key** from **alist**

*Result type:* (:alist %type1 %type2)

*Purity of the procedure:* pure

The keys are compared with the equivalence predicate `equal-values?`.

## object-alist-delete

*Syntax:*

```
(object-alist-delete key alist)
```

*Type parameters:* %type1, %type2

*Arguments:*

Name: **key**  
 Type: %type1  
 Description: the key to be searched

Name: **alist**  
 Type: (:alist %type1 %type2)  
 Description: the association list to be searched

*Result value:* The association list obtained by removing all bindings for key **key** from **alist**

*Result type:* (:alist %type1 %type2)

*Purity of the procedure:* pure

The keys are compared with the equivalence predicate `equal-objects?`.

**content-alist-delete***Syntax:*`(content-alist-delete key alist)`*Type parameters:* %type1, %type2*Arguments:*

Name: `key`  
 Type: %type1  
 Description: the key to be searched

Name: `alist`  
 Type: `(:alist %type1 %type2)`  
 Description: the association list to be searched

*Result value:* The association list obtained by removing all bindings for key `key` from `alist`*Result type:* `(:alist %type1 %type2)`*Purity of the procedure:* pure

The keys are compared with the equivalence predicate `equal-contents?`.

**member-general?***Syntax:*`(member-general? object lst my-eq?)`*Type parameters:* %type*Arguments:*

Name: `object`  
 Type: %type  
 Description: the object to be searched

Name: `lst`  
 Type: `(:uniform-list %type)`  
 Description: the list to be searched

Name: `my-eq?`  
 Type: `(:procedure (%type %type) <boolean> pure)`

Description: equivalence predicate to be used in the search

*Result value:* Result of the search

*Result type:* <boolean>

*Purity of the procedure:* pure

The list `lst` is searched for `object`. If `object` is found return `#t`. Otherwise return `#f`. The list items are compared with the equivalence predicate `my-eq?`.

## member?

*Syntax:*

```
(member? object lst)
```

*Type parameters:* %type

*Arguments:*

Name: `object`

Type: %type

Description: the object to be searched

Name: `lst`

Type: (:uniform-list %type)

Description: the list to be searched

*Result value:* Result of the search

*Result type:* <boolean>

*Purity of the procedure:* pure

The list `lst` is searched for `object`. If `object` is found return `#t`. Otherwise return `#f`. The list items are compared with the equivalence predicate `equal?`.

## member-values?

*Syntax:*

```
(member-values? object lst)
```

*Type parameters:* %type

*Arguments:*

Name: `object`  
 Type: `%type`  
 Description: the object to be searched

Name: `lst`  
 Type: `(:uniform-list %type)`  
 Description: the list to be searched

*Result value:* Result of the search

*Result type:* `<boolean>`

*Purity of the procedure:* pure

The list `lst` is searched for `object`. If `object` is found return `#t`. Otherwise return `#f`. The list items are compared with the equivalence predicate `equal-values?`.

## member-objects?

*Syntax:*

```
(member-objects? object lst)
```

*Type parameters:* `%type`

*Arguments:*

Name: `object`  
 Type: `%type`  
 Description: the object to be searched

Name: `lst`  
 Type: `(:uniform-list %type)`  
 Description: the list to be searched

*Result value:* Result of the search

*Result type:* `<boolean>`

*Purity of the procedure:* pure

The list `lst` is searched for `object`. If `object` is found return `#t`. Otherwise return `#f`. The list items are compared with the equivalence predicate `equal-objects?`.

## member-contents?

*Syntax:*

```
(member-contents? object lst)
```

*Type parameters:* %type

*Arguments:*

Name: `object`  
 Type: %type  
 Description: the object to be searched

Name: `lst`  
 Type: (:uniform-list %type)  
 Description: the list to be searched

*Result value:* Result of the search

*Result type:* <boolean>

*Purity of the procedure:* pure

The list `lst` is searched for `object`. If `object` is found return `#t`. Otherwise return `#f`. The list items are compared with the equivalence predicate `equal-contents?`.

## append

*Syntax:*

```
(append list-1 ... list-n)
```

*Type parameters:* %types

*Arguments:*

Name: `list-k`  
 Type: (:uniform-list  $t_k$ )  
 Description: A list to be merged

*Result value:* A list constructed by appending the arguments

*Result type:* (:uniform-list (:union  $t_1$  ...  $t_n$ ))

*Purity of the procedure:* pure



**append-tuples***Syntax:*`(append-tuples tuple-1 ... tuple-n)`*Type parameters:* `%tuples`*Arguments:*

Name: `tuple-k`  
 Type: `(:tuple tk,1 ... tk,m(k))`  
 Description: A tuple to be merged

*Result value:* A tuple constructed by appending the arguments*Result type:* `(:tuple t1,1 ... t1,m(1) ... tn,1 ... tn,m(n))`*Purity of the procedure:* pure**reverse***Syntax:*`(reverse lst)`*Type parameters:* `%type`*Arguments:*

Name: `lst`  
 Type: `(:uniform-list %type)`  
 Description: A list to be reversed

*Result value:* A list constructed by reversing the argument list*Result type:* `(:uniform-list %type)`*Purity of the procedure:* pure**uniform-list-ref***Syntax:*`(uniform-list-ref lst index)`

*Type parameters:* %type

*Arguments:*

Name: `lst`  
Type: `(:uniform-list %type)`  
Description: A uniform list

Name: `index`  
Type: `<integer>`  
Description: Index to the list

*Result value:* Element of `lst` at position `index`

*Result type:* %type

*Purity of the procedure:* pure

The indices start from zero.

## filter

*Syntax:*

`(filter pred lst)`

*Type parameters:* %type

*Arguments:*

Name: `pred`  
Type: `(:procedure (%type) <boolean> pure)`  
Description: the predicate used for picking the elements

Name: `lst`  
Type: `(:uniform-list %type)`  
Description: The list to be searched

*Result value:* The list computed by picking all the elements in `lst` for which `pred` returns `#t`

*Result type:* `(:uniform-list %type)`

*Purity of the procedure:* pure

## distinct-elements?

*Syntax:*

```
(distinct-elements? lst my-eq?)
```

*Type parameters:* %type

*Arguments:*

Name: `lst`

Type: `(:uniform-list %type)`

Description: The list to be checked

Name: `my-eq?`

Type: `(:procedure (%type %type) <boolean> pure)`

Description: the equivalence predicate used for checking the elements

*Result value:* #t iff no two elements of `lst` are equal by `my-eq?`

*Result type:* <boolean>

*Purity of the procedure:* pure

## find

*Syntax:*

```
(find pred l x-default)
```

*Type parameters:* %type, %default

*Arguments:*

Name: `pred`

Type: `(:procedure (%type) <boolean> pure)`

Description: the predicate used for the search

Name: `l`

Type: `(:uniform-list %type)`

Description: The list to be searched

*Result value:* The first element satisfying the predicate in the list

*Result type:* `(:union %type %default)`

*Purity of the procedure:* pure

If no list element satisfies the predicate value `x-default` is returned.

## count

*Syntax:*

```
(count pred l)
```

*Type parameters:* %type

*Arguments:*

Name: pred

Type: (:procedure (%type) <boolean> pure)

Description: the predicate used for counting

Name: l

Type: (:uniform-list %type)

Description: The list to be searched

*Result value:* The number of the list elements satisfying the predicate

*Result type:* <integer>

*Purity of the procedure:* pure

## 4.6 Logical Operations

### 4.6.1 Simple Procedures

## not

*Syntax:*

```
(not boolean-value)
```

*Arguments:*

Name: boolean-value

Type: <boolean>

Description: A boolean value

*Result value:* #t iff the value of boolean-value is #f

*Result type:* <boolean>

*Purity of the procedure:* pure

## not-object

*Syntax:*

```
(not-object obj)
```

*Arguments:*

Name: obj  
Type: <object>  
Description: Any object

*Result value:* #t iff obj is false, #f otherwise

*Result type:* <boolean>

*Purity of the procedure:* pure

## XOR

*Syntax:*

```
(xor boolean-value1 boolean-value2)
```

*Arguments:*

Name: boolean-value1  
Type: <boolean>  
Description: A boolean value

Name: boolean-value2  
Type: <boolean>  
Description: A boolean value

*Result value:* #t iff exactly one of the values boolean-value1 and boolean-value2 is #t

*Result type:* <boolean>

*Purity of the procedure:* pure

## 4.7 Strings

### 4.7.1 Data Types

*Data type name:* <string-match-result>

*Type:* :union

*Description:* Return value of procedure `string-match`

## 4.7.2 Simple Procedures

### replace-char

*Syntax:*

```
(replace-char str ch-src ch-dest)
```

*Arguments:*

Name: `str`

Type: <string>

Description: A string

Name: `ch-src`

Type: <character>

Description: The character to be replaced

Name: `ch-dest`

Type: <character>

Description: The destination character

*Result value:* A string obtained by replacing character `ch-src` with `ch-dest` in string `str`

*Result type:* <string>

*Purity of the procedure:* pure

### replace-char-with-string

*Syntax:*

```
(replace-char-with-string str ch-src str-dest)
```

*Arguments:*

Name: `str`

Type: <string>

Description: A string

Name: `ch-src`  
Type: `<character>`  
Description: The character to be replaced

Name: `str-dest`  
Type: `<string>`  
Description: The destination string

*Result value:* A string obtained by replacing character `ch-src` with `str-dest` in string `str`

*Result type:* `<string>`

*Purity of the procedure:* pure

## join-strings-with-sep

*Syntax:*

```
(join-strings-with-sep lst str-separator)
```

*Arguments:*

Name: `lst`  
Type: `(:uniform-list <string>)`  
Description: A list of strings to join

Name: `str-separator`  
Type: `<string>`  
Description: The separator

*Result value:* A string obtained to join the strings in `lst` in order

*Result type:* `<string>`

*Purity of the procedure:* pure

## search-substring

*Syntax:*

```
(search-substring str str-match)
```

*Arguments:*

Name: `str`  
Type: `<string>`  
Description: A string

Name: `str-match`  
Type: `<string>`  
Description: The string to be searched

*Purity of the procedure:* pure

*Result value:* Index of the first occurrence of string `str-match` in string `str` (-1 if the string is not found)

*Result type:* `<integer>`

## search-substring-from-end

*Syntax:*

```
(search-substring-from-end str str-match)
```

*Arguments:*

Name: `str`  
Type: `<string>`  
Description: A string

Name: `str-match`  
Type: `<string>`  
Description: The string to be searched

*Purity of the procedure:* pure

*Result value:* Search for string `str-match` in string `str` starting from the end of `str` and return the index of the first match (-1 if the search does not succeed)

*Result type:* `<integer>`

## split-string

*Syntax:*

```
(split-string str ch)
```

*Arguments:*



Name: `str`  
Type: `<string>`  
Description: A string to be split

Name: `ch`  
Type: `<character>`  
Description: The separator character

*Result value:* A list constructed by splitting the string `str`

*Result type:* `(:uniform-list <string>)`

*Purity of the procedure:* pure

The character `ch` is used as a separator in splitting.

## **string**

*Syntax:*

`(string char-1 ... char-n)`

*Arguments:*

Name: `char-k`  
Type: `<character>`  
Description: A character

*Result value:* A string consisting of characters `char-1 ... char-n`

*Result type:* `<string>`

*Purity of the procedure:* pure

## **string->symbol**

*Syntax:*

`(string->symbol str)`

*Arguments:*

Name: `str`  
Type: `<string>`  
Description: A string

*Result value:* The argument string converted to a symbol

*Result type:* <symbol>

*Purity of the procedure:* pure

## string-append

*Syntax:*

```
(string-append str-1 ... str-n)
```

*Arguments:*

Name: **str-k**  
Type: <string>  
Description: A string

*Purity of the procedure:* pure

*Result value:* A string obtained by concatenating strings **str-1 ... str-n**

*Result type:* <string>

## string-char-index

*Syntax:*

```
(string-char-index str ch)
```

*Arguments:*

Name: **str**  
Type: <string>  
Description: A string

Name: **ch**  
Type: <character>  
Description: A character to be searched

*Purity of the procedure:* pure

*Result value:* Index of the first occurrence of character **ch** in string **str** (-1 if the character is not found)

*Result type:* <integer>

## string-char-index-right

*Syntax:*

```
(string-char-index-right str ch)
```

*Arguments:*

Name: `str`  
Type: <string>  
Description: A string

Name: `ch`  
Type: <character>  
Description: A character to be searched

*Purity of the procedure:* pure

*Result value:* Index of the last occurrence of character `ch` in string `str` (-1 if the character is not found)

*Result type:* <integer>

## string-contains-char?

*Syntax:*

```
(string-contains-char? str ch)
```

*Arguments:*

Name: `str`  
Type: <string>  
Description: A string

Name: `ch`  
Type: <character>  
Description: A character

*Purity of the procedure:* pure

*Result value:* `#t` iff string `str` contains character `ch`

*Result type:* <boolean>

## string-drop

*Syntax:*

```
(string-drop str count)
```

*Arguments:*

Name: `str`  
Type: <string>  
Description: A string

Name: `count`  
Type: <integer>  
Description: Number of characters to be dropped

*Result value:* A string constructed of by dropping away the first `count` characters of `str`

*Result type:* <string>

*Purity of the procedure:* pure

If `count` is larger than the length of `str` an exception is raised.

## string-drop-right

*Syntax:*

```
(string-drop-right str count)
```

*Arguments:*

Name: `str`  
Type: <string>  
Description: A string

Name: `count`  
Type: <integer>  
Description: Number of characters to be dropped

*Result value:* A string constructed of by dropping away the last `count` characters of `str`

*Result type:* <string>

*Purity of the procedure:* pure

If `count` is larger than the length of `str` an exception is raised.

## string-empty?

*Syntax:*

```
(string-empty? str)
```

*Arguments:*

Name: `str`  
Type: <string>  
Description: A string

*Result value:* `#t` iff the string is empty

*Result type:* <boolean>

*Purity of the procedure:* pure

## string-exact-match?

*Syntax:*

```
(string-exact-match? str-pattern str-source)
```

*Arguments:*

Name: `str-pattern`  
Type: <string>  
Description: A pattern

Name: `str-source`  
Type: <string>  
Description: The source string for matching

*Result value:* `#t` iff the pattern matches the whole source string

*Result type:* <boolean>

*Purity of the procedure:* pure

## string-last-char

*Syntax:*

```
(string-last-char str)
```

*Arguments:*

Name: `str`  
Type: `<string>`  
Description: A string

*Result value:* The last character of the string `str`

*Result type:* `<character>`

*Purity of the procedure:* pure

If `str` is empty raise an exception.

## string-length

*Syntax:*

```
(string-length str)
```

*Arguments:*

Name: `str`  
Type: `<string>`  
Description: A string

*Result value:* The length of the string `str`

*Result type:* `<integer>`

*Purity of the procedure:* pure

## string-match

*Syntax:*

```
(string-match str-pattern str-source)
```

*Arguments:*

Name: `str-pattern`  
Type: `<string>`  
Description: A pattern

Name: `str-source`  
Type: `<string>`  
Description: The source string for matching

*Result value:* The results of the matching

*Result type:* `<string-match-results>`

*Purity of the procedure:* pure

If the matching fails return `null`. Otherwise the result is a tuple consisting of three elements: the first element is the substring to which the pattern matched, the second item is the index to the source string where the matching started, and the third item the index where the matching stopped.

## string-ref

*Syntax:*

```
(string-ref str index)
```

*Arguments:*

Name: `str`  
Type: `<string>`  
Description: A string

Name: `index`  
Type: `<integer>`  
Description: An index to the string

*Result value:* The character at the `index`th position of string `str`

*Result type:* `<character>`

*Purity of the procedure:* pure

## string-take

*Syntax:*

```
(string-take str count)
```

*Arguments:*

Name: `str`  
Type: `<string>`  
Description: A string

Name: `count`  
Type: `<integer>`  
Description: Number of characters to be taken

*Result value:* A string consisting of the first `count` characters of `str`

*Result type:* `<string>`

*Purity of the procedure:* pure

If `count` is larger than the length of `str` an exception is raised.

## string-take-right

*Syntax:*

```
(string-take-right str count)
```

*Arguments:*

Name: `str`  
Type: `<string>`  
Description: A string

Name: `count`  
Type: `<integer>`  
Description: Number of characters to be taken

*Result value:* A string consisting of the last `count` characters of `str`

*Result type:* `<string>`

*Purity of the procedure:* pure

If `count` is larger than the length of `str` an exception is raised.

## substring

*Syntax:*

```
(substring str i-start i-end)
```



*Arguments:*

Name: `str`  
 Type: `<string>`  
 Description: A string

Name: `i-start`  
 Type: `<integer>`  
 Description: Index from which to start the extraction

Name: `i-end`  
 Type: `<integer>`  
 Description: Index to which to stop the extraction

*Result value:* A substring of `str`

*Result type:* `<integer>`

*Purity of the procedure:* pure

Note that the character at the position `i-end` is not included in the substring.

## 4.8 Vectors

### 4.8.1 Parametrized Procedures

#### `mutable-value-vector-length`

*Syntax:*

```
(mutable-value-vector-length vec)
```

*Type parameters:* `%type`

*Arguments:*

Name: `vec`  
 Type: `(:mutable-value-vector %type)`  
 Description: A vector

*Result value:* Length of the vector `vec`

*Result type:* `<integer>`

*Purity of the procedure:* pure

## mutable-value-vector-ref

*Syntax:*

```
(mutable-value-vector-ref vec index)
```

*Type parameters:* %type

*Arguments:*

Name: `vec`  
Type: `(:mutable-value-vector %type)`  
Description: A vector

Name: `index`  
Type: `<integer>`  
Description: Index to the vector

*Result value:* Element of vector `vec` at the position `index`

*Result type:* %type

*Purity of the procedure:* pure

## mutable-value-vector-set!

*Syntax:*

```
(mutable-value-vector-set! vec index element)
```

*Type parameters:* %type

*Arguments:*

Name: `vec`  
Type: `(:mutable-value-vector %type)`  
Description: A vector

Name: `index`  
Type: `<integer>`  
Description: Index to the vector

Name: `element`  
Type: %type  
Description: The new value of the element

No result value.

*Purity of the procedure:* nonpure

## mutable-vector-length

*Syntax:*

```
(mutable-vector-length vec)
```

*Type parameters:* %type

*Arguments:*

Name: `vec`  
Type: `(:mutable-vector %type)`  
Description: A vector

*Result value:* Length of the vector `vec`

*Result type:* `<integer>`

*Purity of the procedure:* pure

## mutable-vector-ref

*Syntax:*

```
(mutable-vector-ref vec index)
```

*Type parameters:* %type

*Arguments:*

Name: `vec`  
Type: `(:mutable-vector %type)`  
Description: A vector

Name: `index`  
Type: `<integer>`  
Description: Index to the vector

*Result value:* Element of vector `vec` at the position `index`

*Result type:* %type

*Purity of the procedure:* pure

## mutable-vector-set!

*Syntax:*

```
(mutable-vector-set! vec index element)
```

*Type parameters:* %type

*Arguments:*

Name: `vec`  
Type: `(:mutable-vector %type)`  
Description: A vector

Name: `index`  
Type: `<integer>`  
Description: Index to the vector

Name: `element`  
Type: `%type`  
Description: The new value of the element

No result value.

*Purity of the procedure:* nonpure

## value-vector-length

*Syntax:*

```
(value-vector-length vec)
```

*Type parameters:* %type

*Arguments:*

Name: `vec`  
Type: `(:value-vector %type)`  
Description: A vector

*Result value:* Length of the vector `vec`

*Result type:* `<integer>`

*Purity of the procedure:* pure

## value-vector-ref

*Syntax:*

```
(value-vector-ref vec index)
```

*Type parameters:* `%type`

*Arguments:*

Name: `vec`  
Type: `(:value-vector %type)`  
Description: A vector

Name: `index`  
Type: `<integer>`  
Description: Index to the vector

*Result value:* Element of vector `vec` at the position `index`

*Result type:* `%type`

*Purity of the procedure:* pure

## vector-length

*Syntax:*

```
(vector-length vec)
```

*Type parameters:* `%type`

*Arguments:*

Name: `vec`  
Type: `(:vector %type)`  
Description: A vector

*Result value:* Length of the vector `vec`

*Result type:* <integer>

*Purity of the procedure:* pure

## vector-ref

*Syntax:*

```
(vector-ref vec index)
```

*Type parameters:* %type

*Arguments:*

Name: `vec`  
Type: `(:vector %type)`  
Description: A vector

Name: `index`  
Type: <integer>  
Description: Index to the vector

*Result value:* Element of vector `vec` at the position `index`

*Result type:* %type

*Purity of the procedure:* pure

## 4.9 Arithmetic Operations

### 4.9.1 Simple Procedures

## ceiling

*Syntax:*

```
(ceiling r)
```

*Arguments:*

Name: `r`  
Type: <real>

Description: A real number

*Result value:* Rounded value

*Result type:* <integer>

*Purity of the procedure:* pure

This procedure rounds a real number towards infinity.

## factorial

*Syntax:*

```
(factorial i)
```

*Arguments:*

Name: *i*

Type: <integer>

Description: A nonnegative integer number

*Result value:* The factorial of the argument

*Result type:* <integer>

*Purity of the procedure:* pure

## finite?

*Syntax:*

```
(finite? r)
```

*Arguments:*

Name: *r*

Type: <real>

Description: A real number

*Result value:* Returns #t iff *r* is a finite value

*Result type:* <boolean>

*Purity of the procedure:* pure

## floor

*Syntax:*

```
(floor r)
```

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real number

*Result value:* Rounded value

*Result type:* `<integer>`

*Purity of the procedure:* pure

This procedure rounds a real number towards minus infinity.

## gcd

*Syntax:*

```
(gcd i1 i2)
```

*Arguments:*

Name: `i1`  
Type: `<integer>`  
Description: An integer number

Name: `i2`  
Type: `<integer>`  
Description: An integer number

*Result value:* The greatest common divisor of the arguments

*Result type:* `<integer>`

*Purity of the procedure:* pure

## i-log10-exact

*Syntax:*



`(i-log10-exact i)`

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: A positive integer number

*Result value:* The base 10 logarithm of the argument or `null` if the logarithm is not an integer

*Result type:* `(:maybe <integer>)`

*Purity of the procedure:* pure

`i-log2-exact`

*Syntax:*

`(i-log2-exact i)`

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: A positive integer number

*Result value:* The base 2 logarithm of the argument or `null` if the logarithm is not an integer

*Result type:* `(:maybe <integer>)`

*Purity of the procedure:* pure

`infinite?`

*Syntax:*

`(infinite? r)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real number

*Result value:* Returns #t iff *r* is an infinite value

*Result type:* <boolean>

*Purity of the procedure:* pure

## integer+

*Syntax:*

```
(integer+ int1 int2)
```

*Arguments:*

Name: *int1*

Type: <integer>

Description: An integer value

Name: *int2*

Type: <integer>

Description: An integer value

*Result value:* The sum of the arguments

*Result type:* <integer>

*Purity of the procedure:* pure

## integer-

*Syntax:*

```
(integer- int1 int2)
```

*Arguments:*

Name: *int1*

Type: <integer>

Description: An integer value

Name: *int2*

Type: <integer>

Description: An integer value

*Result value:* The difference of the arguments

*Result type:* <integer>

*Purity of the procedure:* pure

## integer\*

*Syntax:*

```
(integer* int1 int2)
```

*Arguments:*

Name: int1

Type: <integer>

Description: An integer value

Name: int2

Type: <integer>

Description: An integer value

*Result value:* The product of the arguments

*Result type:* <integer>

*Purity of the procedure:* pure

## integer<

*Syntax:*

```
(integer< int1 int2)
```

*Arguments:*

Name: int1

Type: <integer>

Description: An integer value

Name: int2

Type: <integer>

Description: An integer value

*Result value:* #t iff int1 < int2

*Result type:* <boolean>

*Purity of the procedure:* pure

## **integer>**

*Syntax:*

(integer> int1 int2)

*Arguments:*

Name: int1  
Type: <integer>  
Description: An integer value

Name: int2  
Type: <integer>  
Description: An integer value

*Result value:* #t iff  $\text{int1} > \text{int2}$

*Result type:* <boolean>

*Purity of the procedure:* pure

## **integer>=**

*Syntax:*

(integer>= int1 int2)

*Arguments:*

Name: int1  
Type: <integer>  
Description: An integer value

Name: int2  
Type: <integer>  
Description: An integer value

*Result value:* #t iff  $\text{int1} \geq \text{int2}$

*Result type:* <boolean>

*Purity of the procedure:* pure

## `integer<=`

*Syntax:*

```
(integer<= int1 int2)
```

*Arguments:*

Name: `int1`  
Type: `<integer>`  
Description: An integer value

Name: `int2`  
Type: `<integer>`  
Description: An integer value

*Result value:* `#t` iff  $\text{int1} \leq \text{int2}$

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `integer->real`

*Syntax:*

```
(integer->real int)
```

*Arguments:*

Name: `int`  
Type: `<integer>`  
Description: An integer value

*Result value:* The integer value converted to a real value

*Result type:* `<real>`

*Purity of the procedure:* pure

**i-abs**

*Syntax:*

(i-abs n)

*Arguments:*

Name: n  
Type: <integer>  
Description: An integer number

*Result value:* Absolute value of the argument

*Result type:* <integer>

*Purity of the procedure:* pure

**i-nonneg-expt**

*Syntax:*

(i-nonneg-expt i-base i-expt)

*Arguments:*

Name: i-base  
Type: <integer>  
Description: An integer number

Name: i-expt  
Type: <integer>  
Description: A nonnegative integer number

*Result value:* i-base raised to the power i-expt

*Result type:* <integer>

*Purity of the procedure:* pure

**i-sign**

*Syntax:*

(i-sign i)

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer number

*Result value:* Return 0 if `i = 0`, 1 if `i > 0`, and -1 if `i < 0`

*Result type:* `<integer>`

*Purity of the procedure:* pure

## `i-square`

*Syntax:*

`(i-square n)`

*Arguments:*

Name: `n`  
Type: `<integer>`  
Description: An integer number

*Result value:* Square of the argument

*Result type:* `<integer>`

*Purity of the procedure:* pure

## `inf`

*Syntax:*

`(inf)`

No arguments.

*Result value:* The exceptional floating point value `inf` (positive infinity)

*Result type:* `<real>`

*Purity of the procedure:* pure

**integer-float?**

*Syntax:*

(integer-float? r)

*Arguments:*

Name: r  
Type: <real>  
Description: A real number

*Result value:* #t iff r is an integer value

*Result type:* <integer>

*Purity of the procedure:* pure

**i-neg**

*Syntax:*

(i-neg n)

*Arguments:*

Name: n  
Type: <integer>  
Description: An integer number

*Result value:* The opposite number of the argument

*Result type:* <integer>

*Purity of the procedure:* pure

**integer-real+**

*Syntax:*

(integer-real+ i r)

*Arguments:*

Name: i



Type: `<integer>`  
Description: An integer value

Name: `r`  
Type: `<real>`  
Description: A real value

*Result value:* The sum of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

### `integer-real-`

*Syntax:*

```
(integer-real- i r)
```

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value

Name: `r`  
Type: `<real>`  
Description: A real value

*Result value:* The difference of the arguments

*Result type:* `<integer>`

*Purity of the procedure:* pure

### `integer-real*`

*Syntax:*

```
(integer-real* i r)
```

*Arguments:*

Name: `i`  
Type: `<integer>`

Description: An integer value

Name: `r`

Type: `<real>`

Description: A real value

*Result value:* The product of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

## `integer-real/`

*Syntax:*

`(integer-real/ i r)`

*Arguments:*

Name: `i`

Type: `<integer>`

Description: An integer value

Name: `r`

Type: `<real>`

Description: A real value

*Result value:* The quotient of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

## `integer-real<`

*Syntax:*

`(integer-real< i r)`

*Arguments:*

Name: `i`

Type: `<integer>`

Description: An integer value

Name: `r`  
Type: `<real>`  
Description: A real value

*Result value:* `#t` iff `i < r`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

### `integer-real<=`

*Syntax:*

`(integer-real<= i r)`

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value

Name: `r`  
Type: `<real>`  
Description: A real value

*Result value:* `#t` iff `i <= r`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

### `integer-real>`

*Syntax:*

`(integer-real> i r)`

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value

Name: `r`  
Type: `<real>`  
Description: A real value

*Result value:* `#t` iff `i > r`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `integer-real>=`

*Syntax:*

`(integer-real>= i r)`

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value

Name: `r`  
Type: `<real>`  
Description: A real value

*Result value:* `#t` iff `i >= r`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `nan`

*Syntax:*

`(nan)`

No arguments.

*Result value:* The exceptional floating point value NaN (not a number)  
*Result type:* `<real>`

*Purity of the procedure:* pure

**nan?**

*Syntax:*

(nan? r)

*Arguments:*

Name: r  
Type: <real>  
Description: A real number

*Result value:* Returns #t iff r is a NaN value

*Result type:* <boolean>

*Purity of the procedure:* pure

**neg-inf**

*Syntax:*

(neg-inf)

No arguments.

*Result value:* The exceptional floating point value -inf (negative infinity)

*Result type:* <real>

*Purity of the procedure:* pure

**quotient**

*Syntax:*

(quotient int1 int2)

*Arguments:*

Name: int1  
Type: <integer>  
Description: An integer value

Name: int2

Type: `<integer>`  
Description: An integer value

*Result value:* The quotient of the arguments  
*Result type:* `<integer>`

*Purity of the procedure:* pure

If the second argument is 0 raise exception `numerical-overflow`. Note that this procedure always returns an integer.

## **r-abs**

*Syntax:*

`(r-abs r)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real number

*Result value:* Absolute value of the argument  
*Result type:* `<real>`

*Purity of the procedure:* pure

## **r-ceiling**

*Syntax:*

`(r-ceiling r)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real number

*Result value:* Rounded value  
*Result type:* `<real>`

*Purity of the procedure:* pure

This procedure rounds a real number towards infinity and returns a real number.

### **r-floor**

*Syntax:*

```
(r-floor r)
```

*Arguments:*

Name: **r**  
Type: **<real>**  
Description: A real number

*Result value:* Rounded value

*Result type:* **<real>**

*Purity of the procedure:* pure

This procedure rounds a real number towards minus infinity and returns a real number.

### **r-int-expt**

*Syntax:*

```
(r-int-expt r-base i-expt)
```

*Arguments:*

Name: **r-base**  
Type: **<real>**  
Description: A real number

Name: **i-expt**  
Type: **<integer>**  
Description: An integer number

*Result value:* **r-base** raised to the power **i-expt**

*Result type:* **<real>**

*Purity of the procedure:* pure

## r-neg

*Syntax:*

(r-neg r)

*Arguments:*

Name: r  
Type: <real>  
Description: A real number

*Result value:* The opposite number of the argument

*Result type:* <real>

*Purity of the procedure:* pure

## r-nonneg-int-expt

*Syntax:*

(r-nonneg-int-expt r-base i-expt)

*Arguments:*

Name: r-base  
Type: <real>  
Description: A real number

Name: i-expt  
Type: <integer>  
Description: A nonnegative integer number

*Result value:* r-base raised to the power i-expt

*Result type:* <real>

*Purity of the procedure:* pure

## r-round

*Syntax:*

(r-round r)



*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real number

*Result value:* Rounded value

*Result type:* `<real>`

*Purity of the procedure:* pure

This procedure rounds a real number and return a real number.

## **r-sign**

*Syntax:*

`(r-sign r)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real number

*Result value:* Return 0 if `r = 0.0`, 1 if `r > 0.0`, and -1 if `r < 0.0`

*Result type:* `<integer>`

*Purity of the procedure:* pure

Return 0 for the exceptional value NaN.

## **r-square**

*Syntax:*

`(r-square r)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real number

*Result value:* Square of the argument

*Result type:* <real>

*Purity of the procedure:* pure

## r-truncate

*Syntax:*

```
(r-truncate r)
```

*Arguments:*

Name: `r`  
Type: <real>  
Description: A real number

*Result value:* Rounded value

*Result type:* <real>

*Purity of the procedure:* pure

This procedure rounds a real number towards zero and returns a real number.

## real+

*Syntax:*

```
(real+ real1 real2)
```

*Arguments:*

Name: `real1`  
Type: <real>  
Description: A real value

Name: `real2`  
Type: <real>  
Description: A real value

*Result value:* The sum of the arguments

*Result type:* <real>

*Purity of the procedure:* pure

**real-**

*Syntax:*

(real- real1 real2)

*Arguments:*

Name: **real1**  
Type: **<real>**  
Description: A real value

Name: **real2**  
Type: **<real>**  
Description: A real value

*Result value:* The difference of the arguments

*Result type:* **<real>**

*Purity of the procedure:* pure

**real\***

*Syntax:*

(real\* real1 real2)

*Arguments:*

Name: **real1**  
Type: **<real>**  
Description: A real value

Name: **real2**  
Type: **<real>**  
Description: A real value

*Result value:* The product of the arguments

*Result type:* **<real>**

*Purity of the procedure:* pure

**real/**

*Syntax:*

(real/ real1 real2)

*Arguments:*

Name: `real1`  
Type: `<real>`  
Description: A real value

Name: `real2`  
Type: `<real>`  
Description: A real value

*Result value:* The quotient of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

**real<**

*Syntax:*

(real< real1 real2)

*Arguments:*

Name: `real1`  
Type: `<real>`  
Description: A real value

Name: `real2`  
Type: `<real>`  
Description: A real value

*Result value:* `#t` iff `real1 < real2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

**real>**

*Syntax:*

```
(real> real1 real2)
```

*Arguments:*

Name: `real1`  
Type: `<real>`  
Description: A real value

Name: `real2`  
Type: `<real>`  
Description: A real value

*Result value:* `#t` iff `real1 > real2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `real<=`

*Syntax:*

```
(real<= real1 real2)
```

*Arguments:*

Name: `real1`  
Type: `<real>`  
Description: A real value

Name: `real2`  
Type: `<real>`  
Description: A real value

*Result value:* `#t` iff `real1 ≤ real2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `real>=`

*Syntax:*

```
(real>= real1 real2)
```

*Arguments:*

Name: `real1`  
 Type: `<real>`  
 Description: A real value

Name: `real2`  
 Type: `<real>`  
 Description: A real value

*Result value:* `#t` iff `real1 ≥ real2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

**real->integer***Syntax:*

`(real->integer r)`

*Arguments:*

Name: `r`  
 Type: `<real>`  
 Description: An integer value of type `<real>`

*Result value:* The real value converted to an integer value of type `<integer>`

*Result type:* `<integer>`

*Purity of the procedure:* pure

If `r` is not an integer value (`xxx.0`) an exception is raised.

**real-integer+***Syntax:*

`(real-integer+ r i)`

*Arguments:*

Name: `r`  
 Type: `<real>`

Description: A real value

Name: `i`

Type: `<integer>`

Description: An integer value

*Result value:* The sum of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

### `real-integer-`

*Syntax:*

```
(real-integer- r i)
```

*Arguments:*

Name: `r`

Type: `<real>`

Description: A real value

Name: `i`

Type: `<integer>`

Description: An integer value

*Result value:* The difference of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

### `real-integer*`

*Syntax:*

```
(real-integer* r i)
```

*Arguments:*

Name: `r`

Type: `<real>`

Description: A real value

Name: `i`  
Type: `<integer>`  
Description: An integer value

*Result value:* The product of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

## `real-integer/`

*Syntax:*

`(real-integer/ r i)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value

Name: `i`  
Type: `<integer>`  
Description: An integer value

*Result value:* The quotient of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

## `real-integer<`

*Syntax:*

`(real-integer< r i)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value



Name: `i`  
 Type: `<integer>`  
 Description: An integer value

*Result value:* `#t` iff `r < i`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

### `real-integer<=`

*Syntax:*

`(real-integer<= r i)`

*Arguments:*

Name: `r`  
 Type: `<real>`  
 Description: A real value

Name: `i`  
 Type: `<integer>`  
 Description: An integer value

*Result value:* `#t` iff `r <= i`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

### `real-integer>`

*Syntax:*

`(real-integer> r i)`

*Arguments:*

Name: `r`  
 Type: `<real>`  
 Description: A real value

Name: `i`

Type: `<integer>`  
Description: An integer value

*Result value:* `#t` iff `r > i`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `real-integer>=`

*Syntax:*

`(real-integer>= r i)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value

Name: `i`  
Type: `<integer>`  
Description: An integer value

*Result value:* `#t` iff `r >= i`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `remainder`

*Syntax:*

`(remainder dividend divisor)`

*Arguments:*

Name: `dividend`  
Type: `<integer>`  
Description: The dividend

Name: `divisor`  
Type: `<integer>`

Description: The divisor

*Result value:* The remainder obtained by dividing the dividend with the divisor

*Result type:* <integer>

*Purity of the procedure:* pure

The semantics of `remainder` is the same as the semantics of procedure `remainder` in Scheme (R6RS).

## round

*Syntax:*

```
(round r)
```

*Arguments:*

Name: r

Type: <real>

Description: A real number

*Result value:* Rounded value

*Result type:* <integer>

*Purity of the procedure:* pure

This procedure rounds a real number and returns an integer.

## truncate

*Syntax:*

```
(truncate r)
```

*Arguments:*

Name: r

Type: <real>

Description: A real number

*Result value:* Rounded value

*Result type:* <integer>

*Purity of the procedure:* pure

This procedure rounds a real number towards zero.

### 4.9.2 Methods

```
+: (<integer> <integer>) → <integer> pure = integer+
+: (<integer> <real>) → <real> pure = integer-real+
+: (<real> <integer>) → <real> pure = real-integer+
+: (<real> <real>) → <real> pure = real+
```

```
-: (<integer> <integer>) → <integer> pure = integer-
-: (<integer> <real>) → <real> pure = integer-real-
-: (<real> <integer>) → <real> pure = real-integer-
-: (<real> <real>) → <real> pure = real-
```

```
*: (<integer> <integer>) → <integer> pure = integer*
*: (<integer> <real>) → <real> pure = integer-real*
*: (<real> <integer>) → <real> pure = real-integer*
*: (<real> <real>) → <real> pure = real*
```

```
/: (<integer> <real>) → <real> pure = integer-real/
/: (<real> <integer>) → <real> pure = real-integer/
/: (<real> <real>) → <real> pure = real/
```

Note that division / between two integers is not defined in the core module as its result is a rational number.

```
<: (<integer> <integer>) → <boolean> pure = integer<
<: (<integer> <real>) → <boolean> pure = integer-real<
<: (<real> <integer>) → <boolean> pure = real-integer<
<: (<real> <real>) → <boolean> pure = real<
```

```
<=: (<integer> <integer>) → <boolean> pure = integer<=
<=: (<integer> <real>) → <boolean> pure = integer-real<=
<=: (<real> <integer>) → <boolean> pure = real-integer<=
<=: (<real> <real>) → <boolean> pure = real<=
```

```
>: (<integer> <integer>) → <boolean> pure = integer>
>: (<integer> <real>) → <boolean> pure = integer-real>
>: (<real> <integer>) → <boolean> pure = real-integer>
>: (<real> <real>) → <boolean> pure = real>
```

```
>=: (<integer> <integer>) → <boolean> pure = integer>=
>=: (<integer> <real>) → <boolean> pure = integer-real>=
>=: (<real> <integer>) → <boolean> pure = real-integer>=
>=: (<real> <real>) → <boolean> pure = real>=
```

```
-: (<integer>) → <integer> pure = i-neg
-: (<real>) → <real> pure = r-neg
abs: (<integer>) → <integer> pure = i-abs
abs: (<real>) → <real> pure = r-abs
square: (<integer>) → <integer> pure = i-square
```

square: (<real>) → <real> pure = r-square  
sign: (<integer>) → <integer> pure = i-sign  
sign: (<real>) → <integer> pure = r-sign

## 4.10 Other

### 4.10.1 Data Types

*Data type name:* :unary-predicate  
*Type:* parametrized procedure class  
*Description:* Unary predicate

*Data type name:* :binary-predicate  
*Type:* parametrized procedure class  
*Description:* Binary predicate



## Chapter 5

# Module (standard-library core-forms)

### 5.1 Macros

#### **with-syntax**

See [3].

#### **syntax-rules**

See [3].

#### **identifier-syntax**

See [3].

#### **quasiquote**

See [3].

#### **quasisyntax**

See [3].

**cond***Syntax:***(cond** [*clause-list*] [*else-clause*] )*clause-list* ::= *clause*<sub>1</sub>, ..., *clause*<sub>*n*</sub>*clause*<sub>*k*</sub> ::= (*condition*<sub>*k*</sub> *expr*<sub>*k,1*</sub>, ..., *expr*<sub>*k,m(k)*</sub> )*else-clause* ::= (**else** *else-expr*<sub>1</sub>, ..., *else-expr*<sub>*p*</sub> )

Each condition must have type `<boolean>`. The type of each *clause*<sub>*k*</sub> is the type of *expr*<sub>*k,m(k)*</sub> (the last expression in the clause). If *else-clause* is present its type is the type of *else-expr*<sub>*p*</sub> (the last expression in the else clause). If *else-expression* is defined the type of the **cond** expression is the union of the types of each clause and the type of the *else-clause*. Otherwise the type of the **cond** expression is `<none>`.

Each *condition*<sub>*k*</sub> is evaluated in order until some of them returns `#t`. When some *condition*<sub>*k*</sub> returns `#t` the expressions *expr*<sub>*k,1*</sub>, ..., *expr*<sub>*k,m(k)*</sub> are evaluated in order. If the result type of the **cond** expression is not `<none>` the value of the last expression *expr*<sub>*k,m(k)*</sub> is returned as the value of the **cond** expression. If none of the conditions return `#t` and *else-clause* is present the expressions *else-expr*<sub>1</sub>, ..., *else-expr*<sub>*p*</sub> are evaluated in order. If the result type of the **cond** expression is not `<none>` the value of the last expression *else-expr*<sub>*p*</sub> is returned as the value of the **cond** expression.

**and***Syntax:***(and** *arg*<sub>1</sub> ...*arg*<sub>*n*</sub> )

The type of each *arg*<sub>*k*</sub> has to be `<boolean>`. The arguments are evaluated in order until some of the arguments returns `#f`. If all of the arguments return `#t` the result of the **and** expression is `#t`. Otherwise the result value is `#f`. Note that all of the arguments are not necessarily evaluated at all.

**or***Syntax:***(or** *arg*<sub>1</sub> ...*arg*<sub>*n*</sub> )



The type of each  $arg_k$  has to be `<boolean>`. The arguments are evaluated in order until some of the arguments returns `#t`. If all of the arguments return `#f` the result of the `or` expression is `#f`. Otherwise the result value is `#t`. Note that all of the arguments are not necessarily evaluated at all.

## cond-object

*Syntax:*

`(cond-object [clause-list] [else-clause] )`

$clause\text{-}list ::= clause_1, \dots, clause_n$

$clause_k ::= (condition_k\ expr_{k,1}, \dots, expr_{k,m(k)}) | (condition_k => handler_k)$

$else\text{-}clause ::= (else\ else\text{-}expr_1, \dots, else\text{-}expr_p)$

This form works as `cond` except all nonfalse values are implemented as true in the conditions. When a clause is of type  $(condition_k => handler_k)$  expression  $handler_k$  has to be a procedure accepting a single argument. When this kind of clause is encountered the  $condition_k$  is evaluated and if its result is not false it is passed to the procedure  $handler_k$  whose result is returned.

## and-object

*Syntax:*

`(and-object expression ... )`

Start evaluating the argument expressions from the left. If any argument returns `#f` stop the evaluation and return `#f`. Otherwise return the value of the last expression.

## or-object

*Syntax:*

`(or-object expression ... )`

Start evaluating the argument expressions from the left. If any argument returns a nonfalse value stop the evaluation and return this value. Otherwise return `#f`.

## let\*

*Syntax:*

(**let\*** (*var-spec*<sub>1</sub> ... *var-spec*<sub>*n*</sub>) *let-body-expressions* )

*var-spec*<sub>*k*</sub> ::= (*var-name*<sub>*k*</sub> [*var-type*<sub>*k*</sub>] *value*<sub>*k*</sub> )

*var-name*<sub>*k*</sub> ::= identifier

*let-body-expressions* ::= expression ...

The **let\*** form is similar to **let** except that the expressions *value*<sub>*k*</sub> are evaluated in order and each expression may use the variables defined before it.

## let\*-mutable

*Syntax:*

(**let\*-mutable** (*var-spec*<sub>1</sub> ... *var-spec*<sub>*n*</sub>) *let-body-expressions* )

*var-spec*<sub>*k*</sub> ::= (*var-name*<sub>*k*</sub> *var-type*<sub>*k*</sub> *value*<sub>*k*</sub> )

*var-name*<sub>*k*</sub> ::= identifier

*let-body-expressions* ::= expression ...

The **let\*-mutable** form is similar to **let-mutable** except that the expressions *value*<sub>*k*</sub> are evaluated in order and each expression may use the variables defined before it.

## let\*-volatile

*Syntax:*

(**let\*-volatile** (*var-spec*<sub>1</sub> ... *var-spec*<sub>*n*</sub>) *let-body-expressions* )

*var-spec*<sub>*k*</sub> ::= (*var-name*<sub>*k*</sub> *var-type*<sub>*k*</sub> *value*<sub>*k*</sub> )

*var-name*<sub>*k*</sub> ::= identifier

*let-body-expressions* ::= expression ...

The **let\*-volatile** form is similar to **let-volatile** except that the expressions *value*<sub>*k*</sub> are evaluated in order and each expression may use the variables defined before it.

**case***Syntax:*

```
(case value [clause-list] [else-clause] )
```

```
clause-list ::= clause1, ..., clausen
```

```
clausek ::= ((keyk,1 ...keyk,p(k)) exprk,1, ..., exprk,m(k))
```

```
else-clause ::= (else else-expr1, ..., else-exprq)
```

The clauses are processed in order. If *value* is equal to some of the keys for clause *k* in the sense of the equality predicate `equal?` processing the clauses is stopped and expressions *expr*<sub>*k,j*</sub> are evaluated in order and the value of the last of these expressions is returned as the value of the **case** expression. If none of the clauses match and the else clause is present the expressions *else-expr*<sub>*j*</sub> are evaluated in order and the value of the last of these expressions is returned. If none of the clauses match and the else clause is not present the **case** expression returns nothing.

**do***Syntax:*

```
(do (var-spec1 ... var-specn)
      (condition [result-expression] )
      body-expression1 ...body-expressionn )
```

```
var-speck ::= (var-namek var-typek init-valuek update-exprk )
```

```
var-namek ::= identifier
```

The type of *condition* has to be `<boolean>`. At the beginning of each iteration *condition* is evaluated. If it returns `#t` the iteration is stopped and the value of *result-expression* is returned as the result of the **do** expression. Otherwise the body expressions are evaluated in order, variables *var-name*<sub>*k*</sub> are assigned new values obtained by evaluating each *update-expr*<sub>*k*</sub> in order, and the next iteration is started from the beginning. If *result-type* is not specified the type of the **do** expression is `<none>`. Expression

```
(do ((var-name1 var-type1 init-value1 update-expr1) ...
      (var-namem var-typem init-valuem update-exprm ))
      (condition [result-expression] )
      body-expression1 ...body-expressionn )
```

is equivalent to

```

(let-mutable ((var-name1 var-type1 init-value1 )...
              (var-namem var-typem init-valuem ))
  (until (condition [result-expression] )
           body-expression1 ...
           body-expressionn
           (set! var-name1 update-expr1 )...
           (set! var-namem update-exprm )))

```

**\$let\***  
**\$letrec**  
**\$letrec\***

*Syntax:*

```

({$let* | $letrec | $letrec* } (var-spec1 ... var-specn ) let-body-expressions )

```

*var-spec*<sub>*k*</sub> ::= (*var-name*<sub>*k*</sub> *value*<sub>*k*</sub> )

*var-name*<sub>*k*</sub> ::= identifier

*let-body-expressions* ::= expression ...

These forms work like the corresponding Scheme forms without the leading '\$', see [2]. These forms may only be used in macro transformers.

**\$and**

*Syntax:*

```

($and expression ... )

```

This form works like Theme-D **and-object** and Scheme form **and**. This form may only be used in macro transformers.

**\$or**

*Syntax:*

```

($or expression ... )

```

This form works like Theme-D **or-object** and Scheme form **or**. This form may only be used in macro transformers.

## define-normal-goops-class

*Syntax:*

```
(define-normal-goops-class name target-name superclass inheritable? im-
mutable? equal-by-value? )
name ::=identifier
target-name ::=identifier
inheritable? ::=boolean
immutable? ::=boolean
equal-by-value? ::=boolean
```

This keyword defines a GOOPS class with the default equivalence predicates (Scheme `eqv?` for `equal?` and `equal-contents?` and Scheme `eq?` for `equal-objects?`) and no zero object.

## define-param-method

*Syntax:*

```
(define-param-method method-name (type1 ... typen) (argument-list result-
type attribute-list) body-expr1, ..., body-exprn )
```

```
method-name ::=identifier
argument-list ::=([arg1 ... argn])
argk ::=(arg-namek arg-typek)
arg-namek ::=identifier
attribute-list ::=(attribute ...) | attribute
attribute ::=pure | nonpure | force-pure
           | always-returns | may-return | never-returns | static
```

The **define-param-method** defines a parametrized method. Note that the argument list may be (). Expressions *arg-type<sub>k</sub>* and *result-type* have to be static type expressions. It is an error if the result type is not `<none>` and the type of the last body expression is not a subtype of *result-type*. If *result-type* is not `<none>` the result value of the procedure is the value of the last body expression.

## define-param-proc

*Syntax:*

**(define-param-proc** *procedure-name* (*type*<sub>1</sub> ... *type*<sub>*n*</sub>) (*argument-list* *result-type* *attribute-list*) *body-expr*<sub>1</sub>, ..., *body-expr*<sub>*n*</sub>)

*procedure-name* ::= identifier

*type*<sub>*k*</sub> ::= identifier

*argument-list* ::= ([*arg*<sub>1</sub> ... *arg*<sub>*n*</sub>])

*arg*<sub>*k*</sub> ::= (*arg-name*<sub>*k*</sub> *arg-type*<sub>*k*</sub>)

*arg-name*<sub>*k*</sub> ::= identifier

*attribute-list* ::= (*attribute* ... ) | *attribute*

*attribute* ::= pure | nonpure | force-pure

| always-returns | may-return | never-returns | static

Keyword **define-param-proc** defines constant *procedure-name* with a parametrized procedure value. Note that the argument list may be (). Expressions *arg-type*<sub>*k*</sub> and *result-type* have to be static type expressions. It is an error if the type of the last body expression is not a subtype of *result-type*. If *result-type* is not <none> the result value of the procedure is the value of the last body expression.

## define-simple-method

*Syntax:*

**(define-simple-method** *method-name* (*argument-list* *result-type* *attribute-list*) *body-expr*<sub>1</sub>, ..., *body-expr*<sub>*n*</sub>)

*method-name* ::= identifier

*argument-list* ::= ([*arg*<sub>1</sub> ... *arg*<sub>*n*</sub>])

*arg*<sub>*k*</sub> ::= (*arg-name*<sub>*k*</sub> *arg-type*<sub>*k*</sub>)

*arg-name*<sub>*k*</sub> ::= identifier

*attribute-list* ::= (*attribute* ... ) | *attribute*

*attribute* ::= pure | nonpure | force-pure

| always-returns | may-return | never-returns | static

Keyword **define-simple-method** defines a simple method. Note that the argument list may be (). Expressions *arg-type*<sub>*k*</sub> and *result-type* have to be static type expressions. It is an error if the result type is not <none> and the type of the last body expression is not a subtype of *result-type*. If *result-type* is not <none> the result value of the procedure is the value of the last body expression.

## define-simple-proc

*Syntax:*

(**define-simple-proc** *procedure-name* (*argument-list* *result-type* *attribute-list*)  
*body-expr*<sub>1</sub>, ..., *body-expr*<sub>*n*</sub> )

*procedure-name* ::= identifier  
*argument-list* ::= ([*arg*<sub>1</sub> ... *arg*<sub>*n*</sub>] )  
*arg*<sub>*k*</sub> ::= (*arg-name*<sub>*k*</sub> *arg-type*<sub>*k*</sub>)  
*arg-name*<sub>*k*</sub> ::= identifier  
*attribute-list* ::= (*attribute* ... ) | *attribute*  
*attribute* ::= pure | nonpure | force-pure  
| always-returns | may-return | never-returns | static

Keyword **define-simple-proc** defines constant *procedure-name* with a simple procedure value. Note that the argument list may be (). Expressions *arg-type*<sub>*k*</sub> and *result-type* have to be static type expressions. It is an error if the result type is not <none> and the type of the last body expression is not a subtype of *result-type*. If *result-type* is not <none> the result value of the procedure is the value of the last body expression.

## guard

*Syntax:*

(**guard** (*exception-variable* *clause*<sub>1</sub> ... *clause*<sub>*n*</sub> [*else-clause*] )  
*body-expr*<sub>1</sub> ... *body-expr*<sub>*n*</sub> )  
*clause*<sub>*k*</sub> ::= (*condition*<sub>*k*</sub> *expr*<sub>*k,1*</sub>, ..., *expr*<sub>*k,m(k)*</sub> )  
*else-clause* ::= (**else** *else-expr*<sub>1</sub>, ..., *else-expr*<sub>*p*</sub> )

The syntax of *clause*<sub>*k*</sub> and *else-clause* is similar to the same syntax elements in **cond** form, see section 5.1. When a **guard** form is executed it starts evaluating the body expressions in order. If an exception is raised during the body expression evaluation do the following:

- Bind the variable *exception-variable* to the exception.
- Evaluate conditions in clauses *clause*<sub>*k*</sub> in order. When the first condition returns true evaluate the corresponding clause expressions and return the value of the last expression as the value of the **guard** expression.
- If none of the conditions returns true evaluate the *else-clause* and return its value as the value of the **guard** expression.
- If none of the conditions returns true and *else-clause* is not present reraise the exception to be handled by the surrounding exception handler.

## make

*Syntax:*

(**make** *class* *field-value*<sub>1</sub> ...*field-value*<sub>*n*</sub> )

Keyword **make** creates an instance of *class* calling the constructor of *class* and passing the arguments *field-value*<sub>*k*</sub>. Expression *class* has to be a static type expression and its value has to be a class.



## Chapter 6

# Module (standard-library bitwise-arithmetic)

### 6.1 Simple Procedures

bitwise-not

*Syntax:*

```
(bitwise-not i)
```

*Arguments:*

Name: i  
Type: <integer>  
Description: An integer number

*Result value:* The ones-complement of the argument

*Result type:* <integer>

*Purity of the procedure:* pure

bitwise-and

*Syntax:*

```
(bitwise-and i1 i2)
```

*Arguments:*

Name: `i1`  
Type: `<integer>`  
Description: An integer number

Name: `i2`  
Type: `<integer>`  
Description: An integer number

*Result value:* The bitwise AND between the arguments

*Result type:* `<integer>`

*Purity of the procedure:* pure

## `bitwise-ior`

*Syntax:*

```
(bitwise-ior i1 i2)
```

*Arguments:*

Name: `i1`  
Type: `<integer>`  
Description: An integer number

Name: `i2`  
Type: `<integer>`  
Description: An integer number

*Result value:* The bitwise OR between the arguments

*Result type:* `<integer>`

*Purity of the procedure:* pure

## `bitwise-xor`

*Syntax:*

```
(bitwise-xor i1 i2)
```

*Arguments:*

Name: `i1`  
Type: `<integer>`  
Description: An integer number

Name: `i2`  
Type: `<integer>`  
Description: An integer number

*Result value:* The bitwise XOR between the arguments

*Result type:* `<integer>`

*Purity of the procedure:* pure

## `bitwise-arithmetic-shift`

*Syntax:*

```
(bitwise-arithmetic-shift i i-shift)
```

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer number

Name: `i-shift`  
Type: `<integer>`  
Description: The number of bits to be shifted (maybe negative)

*Result value:* Value of `i` with bits shifted by `i-shift` positions

*Result type:* `<integer>`

*Purity of the procedure:* pure

Positive values of `i-shift` mean shifting bits left and negative values shifting right. Name `ash` is an alias for `bitwise-arithmetic-shift`.

## `bitwise-arithmetic-shift-right`

*Syntax:*

```
(bitwise-arithmetic-shift-right i i-shift)
```

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer number

Name: `i-shift`  
Type: `<integer>`  
Description: The number of bits to be shifted (maybe negative)

*Result value:* Value of `i` with bits shifted right by `i-shift` positions

*Result type:* `<integer>`

*Purity of the procedure:* pure

Negative values of `i-shift` shift bit to the left.

## `bitwise-arithmetic-shift-left`

*Syntax:*

```
(bitwise-arithmetic-shift-left i i-shift)
```

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer number

Name: `i-shift`  
Type: `<integer>`  
Description: The number of bits to be shifted (maybe negative)

*Result value:* Value of `i` with bits shifted left by `i-shift` positions

*Result type:* `<integer>`

*Purity of the procedure:* pure

Negative values of `i-shift` shift bit to the right.

## Chapter 7

# Module (standard-library promise)

This module implements delayed evaluation with the promise objects. The promises resemble Scheme promises, see [2].

### 7.1 Data Types

*Data type name:* `:promise`

*Type:* `:procedure`

*Number of type parameters:* 1

*Description:* A promise object

*Data type name:* `:nonpure-promise`

*Type:* `:procedure`

*Number of type parameters:* 1

*Description:* A promise object that can have side effects

### 7.2 Macros

#### **delay**

*Syntax:*

`(delay expression )`

This macro creates a promise that delays the evaluation of the given expression. This is a frontend to the procedure `make-promise`. The argument expression has to be pure.

## delay-nonpure

*Syntax:*

(**delay-nonpure** *expression* )

This macro creates a promise that delays the evaluation of the given expression. This is a frontend to the procedure `make-promise`. The argument expression may be nonpure.

## 7.3 Parametrized Procedures

### force

*Syntax:*

(**force** *promise*)

*Type parameters:* %type

*Arguments:*

Name: `promise`  
Type: (`:promise` %type)  
Description: A promise

*Result value:* The value of the promise

*Result type:* %type

*Purity of the procedure:* pure

This procedure evaluates the promise if it has not already been done and returns the value.

### force-nonpure

*Syntax:*

(**force-nonpure** *promise*)

*Type parameters:* %type

*Arguments:*

Name: `promise`  
Type: `(:nonpure-promise %type)`  
Description: A nonpure promise

*Result value:* The value of the promise

*Result type:* `%type`

*Purity of the procedure:* nonpure

This procedure evaluates the promise if it has not already been done and returns the value.

## `make-nonpure-promise`

*Syntax:*

```
(make-nonpure-promise proc)
```

*Type parameters:* `%type`

*Arguments:*

Name: `proc`  
Type: `(:procedure () %type nonpure)`  
Description: A procedure

*Result value:* A promise

*Result type:* `(:nonpure-promise %type)`

*Purity of the procedure:* pure

This procedure creates a promise that delays the evaluation of the given procedure.

## `make-promise`

*Syntax:*

```
(make-promise proc)
```

*Type parameters:* `%type`

*Arguments:*

Name: `proc`  
Type: `(:procedure () %type pure)`  
Description: A procedure

*Result value:* A promise  
*Result type:* `(:promise %type)`

*Purity of the procedure:* pure

This procedure creates a promise that delays the evaluation of the given procedure. The procedure has to be pure.



## Chapter 8

# Module (standard-library stream)

Streams are kind of abstract sequences. A stream is defined by the following operations:

- *stream-value*: Return the current value of the stream.
- *stream-next*: Read one stream element forward and return the stream with the new element as its current value.
- *stream-empty?*: Return true iff the stream is empty.

See programs `test451.thp`, `test452.thp`, and `test453.thp` in directory `theme-d-code/tests` for examples.

### 8.1 Data Types

*Data type name:* `:stream`

*Type:* `:union`

*Number of type parameters:* 1

*Description:* A stream

*Data type name:* `:nonempty-stream`

*Type:* `:pair`

*Number of type parameters:* 1

*Description:* A nonempty stream

*Data type name:* `:nonpure-stream`

*Type:* `:union`

*Number of type parameters:* 1

*Description:* A nonpure stream

*Data type name:* `:nonempty-nonpure-stream`

*Type:* `:pair`

*Number of type parameters:* 1

*Description:* A nonempty nonpure stream

## 8.2 Simple Procedures

### make-input-expr-stream

*Syntax:*

```
(make-input-expr-stream ip)
```

*Arguments:*

Name: ip

Type: <input-port>

Description: An input port

*Result value:* A nonpure stream that reads from the given input port

*Result type:* (:nonpure-stream <object>)

*Purity of the procedure:* pure

## 8.3 Parametrized Procedures

### stream-value

*Syntax:*

```
(stream-value stm)
```

*Type parameters:* %type

*Arguments:*

Name: stm

Type: (:stream %type)

Description: A stream

*Result value:* The current value of the stream

*Result type:* %type

*Purity of the procedure:* pure

If the stream `stm` is empty this procedure raises an exception.

### **stream-next**

*Syntax:*

```
(stream-next stm)
```

*Type parameters:* %type

*Arguments:*

Name: `stm`  
Type: `(:stream %type)`  
Description: A stream

*Result value:* A stream located one step forward from the given stream

*Result type:* `(:stream %type)`

*Purity of the procedure:* pure

If the stream `stm` is empty this procedure raises an exception.

### **stream-empty?**

*Syntax:*

```
(stream-empty? stm)
```

*Type parameters:* %type

*Arguments:*

Name: `stm`  
Type: `(:stream %type)`  
Description: A stream

*Result value:* `#t` iff the stream is empty

*Result type:* `<boolean>`

*Purity of the procedure:* pure

**stream->list**

*Syntax:*

```
(stream->list stm)
```

*Type parameters:* %type

*Arguments:*

Name: **stm**  
Type: (:stream %type)  
Description: A stream

*Result value:* A list

*Result type:* (:uniform-list %type)

*Purity of the procedure:* pure

This procedure constructs a list by reading the stream until it is empty.

**list->stream**

*Syntax:*

```
(list->stream l)
```

*Type parameters:* %type

*Arguments:*

Name: **stm**  
Type: (:uniform-list %type)  
Description: A list

*Result value:* A stream that processes the given list

*Result type:* (:stream %type)

*Purity of the procedure:* pure

**nonpure-stream-value**

*Syntax:*

```
(nonpure-stream-value stm)
```

*Type parameters:* %type

*Arguments:*

Name: `stm`  
Type: `(:nonpure-stream %type)`  
Description: A nonpure stream

*Result value:* The current value of the stream

*Result type:* %type

*Purity of the procedure:* pure

If the stream `stm` is empty this procedure raises an exception.

## `nonpure-stream-next`

*Syntax:*

```
(nonpure-stream-next stm)
```

*Type parameters:* %type

*Arguments:*

Name: `stm`  
Type: `(:nonpure-stream %type)`  
Description: A nonpure stream

*Result value:* A nonpure stream located one step forward from the given nonpure-stream

*Result type:* `(:nonpure-stream %type)`

*Purity of the procedure:* nonpure

If the stream `stm` is empty this procedure raises an exception.

## `nonpure-stream-empty?`

*Syntax:*

```
(nonpure-stream-empty? stm)
```

*Type parameters:* %type

*Arguments:*

Name: `stm`  
Type: `(:nonpure-stream %type)`  
Description: A nonpure stream

*Result value:* `#t` iff the stream is empty

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `nonpure-stream->list`

*Syntax:*

`(nonpure-stream->list stm)`

*Type parameters:* %type

*Arguments:*

Name: `stm`  
Type: `(:nonpure-stream %type)`  
Description: A nonpure stream

*Result value:* A list

*Result type:* `(:uniform-list %type)`

*Purity of the procedure:* nonpure

This procedure constructs a list by reading the stream until it is empty.

## `stream-map`

*Syntax:*

`(stream-map proc stm)`

*Type parameters:* %type1, %type2

*Arguments:*

Name: `proc`  
 Type: `(:procedure (%type1) %type2 pure)`  
 Description: The procedure to be applied

Name: `stm`  
 Type: `(:stream %type1)`  
 Description: The source stream

*Result value:* The target stream  
*Result type:* `(:stream %type2)`

*Purity of the procedure:* pure

This procedure applies the argument procedure to the source stream elements with delayed evaluation. Another stream is returned.

### `stream-map-nonpure`

*Syntax:*

```
(stream-map-nonpure proc stm)
```

*Type parameters:* `%type1`, `%type2`

*Arguments:*

Name: `proc`  
 Type: `(:procedure (%type1) %type2 nonpure)`  
 Description: The procedure to be applied

Name: `stm`  
 Type: `(:stream %type1)`  
 Description: The source stream

*Result value:* The target stream  
*Result type:* `(:nonpure-stream %type2)`

*Purity of the procedure:* nonpure

This procedure applies the argument procedure to the source stream elements with delayed evaluation. Another stream is returned. The applied procedure may have side effects and the result is a nonpure stream.

### `stream-for-each`

*Syntax:*

```
(stream-for-each proc stm)
```

*Type parameters:* %type1

*Arguments:*

Name: `proc`  
 Type: (:procedure (%type1) <none> nonpure)  
 Description: The procedure to be applied

Name: `stm`  
 Type: (:stream %type1)  
 Description: A stream

No result value.

*Purity of the procedure:* nonpure

This procedure applies the argument procedure to the source stream elements. The evaluation is not delayed and no value is returned.

## nonpure-stream-map

*Syntax:*

```
(nonpure-stream-map proc stm)
```

*Type parameters:* %type1, %type2

*Arguments:*

Name: `proc`  
 Type: (:procedure (%type1) %type2 nonpure)  
 Description: The procedure to be applied

Name: `stm`  
 Type: (:nonpure-stream %type1)  
 Description: The source stream

*Result value:* The target stream

*Result type:* (:nonpure-stream %type2)

*Purity of the procedure:* nonpure



This procedure applies the argument procedure to the source nonpure stream elements with delayed evaluation. Another stream is returned. The applied procedure may have side effects and the result is a nonpure stream.

### `nonpure-stream-for-each`

*Syntax:*

```
(nonpure-stream-for-each proc stm)
```

*Type parameters:* `%type1`

*Arguments:*

Name: `proc`  
Type: `(:procedure (%type1) <none> nonpure)`  
Description: The procedure to be applied

Name: `stm`  
Type: `(:nonpure-stream %type1)`  
Description: A nonpure stream

No result value.

*Purity of the procedure:* nonpure

This procedure applies the argument procedure to the source nonpure stream elements. The evaluation is not delayed and no value is returned.



## Chapter 9

# Module (standard-library iterator)

This module implements purely functional iterators, see [1].

### 9.1 Data Types

*Data type name:* `:iterator`

*Type:* `<param-logical-type>`

*Number of type parameters:* 1

*Definition:* `(:param-proc (%target) ((:consumer %source %target)) %target pure)`

*Description:* An iterator

*Data type name:* `:iterator-inst`

*Type:* `<param-logical-type>`

*Number of type parameters:* 2

*Definition:* `(:procedure ((:consumer %source %target)) %target pure)`

*Description:* An instance of an iterator for which the target type or the iteration is fixed

*Data type name:* `:consumer`

*Type:* `<param-logical-type>`

*Number of type parameters:* 2

*Definition:* `(:procedure ((:maybe %source) <boolean> (:maybe (:iterator-inst %source %target)))) %target pure)`

*Description:* A procedure that “consumes” values yielded by an iterator

### 9.2 Parametrized Procedures

`end-iter`

*Syntax:*

```
(end-iter consumer)
```

*Type parameters:* %source, %target

*Arguments:*

```
Name: consumer
Type: (:consumer %source %target)
Description: A consumer procedure
```

*Result value:* Target object

*Result type:* %target

*Purity of the procedure:* pure

This procedure is used when the iterator reaches its end.

## gen-list

*Syntax:*

```
(gen-list l consumer iterator-inst)
```

*Type parameters:* %source, %target

*Arguments:*

```
Name: l
Type: (:uniform-list %source)
Description: A list for which to create an iterator
```

```
Name: consumer
Type: (:consumer %source %target)
Description: A consumer procedure
```

```
Name: iterator-inst
Type: (:iterator-inst %source %target)
Description: An iterator instance
```

*Result value:* Target object

*Result type:* %target

*Purity of the procedure:* pure

This procedure is used internally to create a list iterator.

## get-list-iterator

*Syntax:*

```
(get-list-iterator l)
```

*Type parameters:* %source

*Arguments:*

Name: l  
Type: (:uniform-list %source)  
Description: A list for which to create an iterator

*Result value:* An iterator for the given list

*Result type:* (:iterator %source)

*Purity of the procedure:* pure

This procedure is used to create a list iterator.

## gen-mutable-vector

*Syntax:*

```
(gen-mutable-vector v consumer iterator-inst)
```

*Type parameters:* %source, %target

*Arguments:*

Name: v  
Type: (:mutable-vector %source)  
Description: A mutable vector for which to create an iterator

Name: consumer  
Type: (:consumer %source %target)  
Description: A consumer procedure

Name: iterator-inst  
Type: (:iterator-inst %source %target)  
Description: An iterator instance

*Result value:* Target object

*Result type:* %target

*Purity of the procedure:* pure

This procedure is used internally to create a mutable vector iterator.

## get-mutable-vector-iterator

*Syntax:*

```
(get-mutable-vector-iterator v)
```

*Type parameters:* %source

*Arguments:*

Name: v

Type: (:mutable-vector %source)

Description: A mutable vector for which to create an iterator

*Result value:* An iterator for the given mutable vector

*Result type:* (:iterator %source)

*Purity of the procedure:* pure

This procedure is used to create an iterator for a mutable vector.

## iter-map1

*Syntax:*

```
(iter-map1 proc iterator)
```

*Type parameters:* %source, %component

*Arguments:*

Name: proc

Type: (:procedure (%source) %component pure)

Description: A procedure to apply to the given iterator

Name: iterator

Type: (:iterator %source)

Description: An iterator to iterate the given procedure

*Result value:* A list constructed by applying the given procedure to the values yielded by the iterator

*Result type:* (:uniform-list %component)

*Purity of the procedure:* pure

This procedure maps the given procedure to each element yielded by the iterator and constructs a list from the result values.

## iter-map2

*Syntax:*

```
(iter-map2 proc iterator1 iterator2)
```

*Type parameters:* %source1, %source2, %component

*Arguments:*

Name: `proc`

Type: (:procedure (%source1 %source2) %component pure)

Description: A procedure to apply to the given iterator

Name: `iterator1`

Type: (:iterator %source1)

Description: An iterator to iterate the given procedure

Name: `iterator2`

Type: (:iterator %source2)

Description: Another iterator to iterate the given procedure

*Result value:* A list constructed by applying the given procedure to the values yielded by the iterators

*Result type:* (:uniform-list %component)

*Purity of the procedure:* pure

This procedure maps pairwise the given procedure to all the elements yielded by the iterators and constructs a list from the result values.

## iter-every1

*Syntax:*

```
(iter-every1 proc iterator)
```

*Type parameters:* %source

*Arguments:*

Name: `proc`  
Type: `(:procedure (%source) <boolean> pure)`  
Description: A procedure to apply to the given iterator

Name: `iterator`  
Type: `(:iterator %source)`  
Description: An iterator to iterate the given procedure

*Result value:* #t iff the procedure is returns true for all iterated values

*Result type:* <boolean>

*Purity of the procedure:* pure

This procedure maps the given procedure to each element yielded by the iterator and returns #t iff all the results are #t. If some application returns #f the application is terminated and #f returned.

## iter-every2

*Syntax:*

```
(iter-every2 proc iterator1 iterator2)
```

*Type parameters:* %source1, %source2

*Arguments:*

Name: `proc`  
Type: `(:procedure (%source1 %source2) <boolean> pure)`  
Description: A procedure to apply to the given iterator

Name: `iterator1`  
Type: `(:iterator %source1)`  
Description: An iterator to iterate the given procedure

Name: `iterator2`  
Type: `(:iterator %source2)`  
Description: Another iterator to iterate the given procedure

*Result value:* #t iff the procedure is returns true for all iterated values

*Result type:* <boolean>



*Purity of the procedure:* pure

This procedure maps pairwise the given procedure to all the elements yielded by the iterators and returns **#t** iff all the results are **#t**. If some application returns **#f** the application is terminated and **#f** returned.



## Chapter 10

# Module (standard-library nonpure-iterator)

This module `nonpure` implements nonpure iterators analogous to the purely functional ones presented in the previous section. Nonpure iterators are needed in following cases:

- The operation done to the values yielded by iterators has side effects, e.g. printing.
- The generation of values for an iterator has side effects, e.g. reading values from a file.

### 10.1 Data Types

*Data type name:* `:nonpure-iterator`

*Type:* `<param-logical-type>`

*Number of type parameters:* 1

*Definition:* `(:param-proc (%target) ((:nonpure-consumer %source %target)) %target nonpure)`

*Description:* An iterator

*Data type name:* `:nonpure-iterator-inst`

*Type:* `<param-logical-type>`

*Number of type parameters:* 2

*Definition:* `(:procedure ((:nonpure-consumer %source %target)) %target nonpure)`

*Description:* An instance of an iterator for which the target type or the iteration is fixed

*Data type name:* `:nonpure-consumer`

*Type:* `<param-logical-type>`

*Number of type parameters:* 2

*Definition:* `(:procedure ((:maybe %source) <boolean> (:maybe (:nonpure-iterator-inst %source %target)))) %target nonpure)`

*Description:* A procedure that “consumes” values yielded by an iterator

## 10.2 Parametrized Procedures

### nonpure-end-iter

*Syntax:*

```
(nonpure-end-iter consumer)
```

*Type parameters:* %source, %target

*Arguments:*

```
Name: consumer
Type: (:nonpure-consumer %source %target)
Description: A consumer procedure
```

*Result value:* Target object

*Result type:* %target

*Purity of the procedure:* nonpure

This procedure is used when the iterator reaches its end.

### gen-list-nonpure

*Syntax:*

```
(gen-list-nonpure l consumer iterator-inst)
```

*Type parameters:* %source, %target

*Arguments:*

```
Name: l
Type: (:uniform-list %source)
Description: A list for which to create an iterator
```

```
Name: consumer
Type: (:nonpure-consumer %source %target)
Description: A consumer procedure
```

```
Name: iterator-inst
```

Type: (:nonpure-iterator-inst %source %target)  
Description: An iterator instance

*Result value:* Target object  
*Result type:* %target

*Purity of the procedure:* nonpure

This procedure is used internally to create a list iterator.

## get-list-nonpure-iterator

*Syntax:*

```
(get-list-nonpure-iterator l)
```

*Type parameters:* %source

*Arguments:*

Name: l  
Type: (:uniform-list %source)  
Description: A list for which to create an iterator

*Result value:* An iterator for the given list  
*Result type:* (:nonpure-iterator %source)

*Purity of the procedure:* nonpure

This procedure is used to create a list iterator.

## gen-mutable-vector-nonpure

*Syntax:*

```
(gen-mutable-vector-nonpure v consumer iterator-inst)
```

*Type parameters:* %source, %target

*Arguments:*

Name: v  
Type: (:mutable-vector %source)  
Description: A mutable vector for which to create an iterator

Name: `consumer`  
 Type: `(:nonpure-consumer %source %target)`  
 Description: A consumer procedure

Name: `iterator-inst`  
 Type: `(:nonpure-iterator-inst %source %target)`  
 Description: An iterator instance

*Result value:* Target object  
*Result type:* `%target`

*Purity of the procedure:* nonpure

This procedure is used internally to create a mutable vector iterator.

## `get-mutable-vector-nonpure-iterator`

*Syntax:*

```
(get-mutable-vector-nonpure-iterator v)
```

*Type parameters:* `%source`

*Arguments:*

Name: `v`  
 Type: `(:mutable-vector %source)`  
 Description: A mutable vector for which to create an iterator

*Result value:* An iterator for the given mutable vector  
*Result type:* `(:nonpure-iterator %source)`

*Purity of the procedure:* nonpure

This procedure is used to create an iterator for a mutable vector.

## `nonpure-iter-map1`

*Syntax:*

```
(nonpure-iter-map1 proc iterator)
```

*Type parameters:* `%source`, `%component`

*Arguments:*

Name: `proc`  
 Type: `(:procedure (%source) %component nonpure)`  
 Description: A procedure to apply to the given iterator

Name: `iterator`  
 Type: `(:nonpure-iterator %source)`  
 Description: An iterator to iterate the given procedure

*Result value:* A list constructed by applying the given procedure to the values yielded by the iterator

*Result type:* `(:uniform-list %component)`

*Purity of the procedure:* nonpure

This procedure maps the given procedure to each element yielded by the iterator and constructs a list from the result values.

## nonpure-iter-map2

*Syntax:*

```
(nonpure-iter-map2 proc iterator1 iterator2)
```

*Type parameters:* `%source1`, `%source2`, `%component`

*Arguments:*

Name: `proc`  
 Type: `(:procedure (%source1 %source2) %component nonpure)`  
 Description: A procedure to apply to the given iterator

Name: `iterator1`  
 Type: `(:nonpure-iterator %source1)`  
 Description: An iterator to iterate the given procedure

Name: `iterator2`  
 Type: `(:nonpure-iterator %source2)`  
 Description: Another iterator to iterate the given procedure

*Result value:* A list constructed by applying the given procedure to the values yielded by the iterators

*Result type:* `(:uniform-list %component)`

*Purity of the procedure:* nonpure

This procedure maps pairwise the given procedure to all the elements yielded

by the iterators and constructs a list from the result values.

## nonpure-iter-every1

*Syntax:*

```
(nonpure-iter-every1 proc iterator)
```

*Type parameters:* %source

*Arguments:*

Name: `proc`  
 Type: `(:procedure (%source) <boolean> nonpure)`  
 Description: A procedure to apply to the given iterator

Name: `iterator`  
 Type: `(:nonpure-iterator %source)`  
 Description: An iterator to iterate the given procedure

*Result value:* `#t` iff the procedure is returns true for all iterated values

*Result type:* `<boolean>`

*Purity of the procedure:* nonpure

This procedure maps the given procedure to each element yielded by the iterator and returns `#t` iff all the results are `#t`. If some application returns `#f` the application is terminated and `#f` returned.

## nonpure-iter-every2

*Syntax:*

```
(nonpure-iter-every2 proc iterator1 iterator2)
```

*Type parameters:* %source1, %source2

*Arguments:*

Name: `proc`  
 Type: `(:procedure (%source1 %source2) <boolean> nonpure)`  
 Description: A procedure to apply to the given iterator

Name: `iterator1`  
 Type: `(:nonpure-iterator %source1)`



Description: An iterator to iterate the given procedure

Name: `iterator2`

Type: `(:nonpure-iterator %source2)`

Description: Another iterator to iterate the given procedure

*Result value:* `#t` iff the procedure is returns true for all iterated values

*Result type:* `<boolean>`

*Purity of the procedure:* nonpure

This procedure maps pairwise the given procedure to all the elements yielded by the iterators and returns `#t` iff all the results are `#t`. If some application returns `#f` the application is terminated and `#f` returned.

## nonpure-iter-for-each1

*Syntax:*

```
(nonpure-iter-for-each1 proc iterator)
```

*Type parameters:* `%source`

*Arguments:*

Name: `proc`

Type: `(:procedure (%source) <none> nonpure)`

Description: A procedure to apply to the given iterator

Name: `iterator`

Type: `(:nonpure-iterator %source)`

Description: An iterator to iterate the given procedure

No result value.

*Purity of the procedure:* nonpure

This procedure maps the given procedure to each element yielded by the iterator.

## nonpure-iter-for-each2

*Syntax:*

```
(nonpure-iter-for-each2 proc iterator1 iterator2)
```

*Type parameters:* %source1, %source2

*Arguments:*

Name: `proc`  
 Type: (:procedure (%source1 %source2) <none> nonpure)  
 Description: A procedure to apply to the given iterator

Name: `iterator1`  
 Type: (:nonpure-iterator %source1)  
 Description: An iterator to iterate the given procedure

Name: `iterator2`  
 Type: (:nonpure-iterator %source2)  
 Description: Another iterator to iterate the given procedure

No result value.

*Purity of the procedure:* nonpure

This procedure maps pairwise the given procedure to all the elements yielded by the iterators.

## gen-generator

*Syntax:*

```
(gen-generator generator terminate? consumer iterator-inst)
```

*Type parameters:* %source, %target

*Arguments:*

Name: `generator`  
 Type: (:procedure () %source nonpure)  
 Description: A generator from which to create an iterator

Name: `terminate?`  
 Type: (:procedure (%source) <boolean> pure)  
 Description: A procedure that determines when to end the iteration

Name: `consumer`  
 Type: (:nonpure-consumer %source %target)  
 Description: A consumer procedure

Name: `iterator-inst`  
 Type: (:nonpure-iterator-inst %source %target)

Description: An iterator instance

*Result value:* Target object

*Result type:* %target

*Purity of the procedure:* nonpure

This procedure is used internally to create an iterator from a generator.

## generator->iterator

*Syntax:*

```
(generator->iterator generator terminate?)
```

*Type parameters:* %source

*Arguments:*

Name: `generator`

Type: (:procedure () %source nonpure)

Description: A generator from which to create an iterator

Name: `terminate?`

Type: (:procedure (%source) <boolean> pure)

Description: A procedure that determines when to end the iteration

*Result value:* An iterator for the given generator

*Result type:* (:nonpure-iterator %source)

*Purity of the procedure:* nonpure

This procedure is used to create an iterator that obtains its values from a generator.



# Chapter 11

## Module (standard-library object-string-conversion)

This module contains procedures to compute string output for different objects.

### 11.1 Simple Procedures

#### `general-object->string`

*Syntax:*

```
(general-object->string obj)
```

*Arguments:*

Name: `obj`  
Type: `<object>`  
Description: Any object

*Result value:* The name of the class of the object enclosed in square brackets

*Result type:* `<string>`

*Purity of the procedure:* pure

This procedure is used as the default implementation of generic procedure `object->string` in case no explicit method is defined.

#### `boolean->string`

*Syntax:*

(boolean->string obj)

*Arguments:*

Name: obj  
Type: <boolean>  
Description: A boolean value

*Result value:* "#t" or "#f"

*Result type:* <string>

*Purity of the procedure:* pure

**character->string**

*Syntax:*

(character->string obj)

*Arguments:*

Name: obj  
Type: <character>  
Description: A character value

*Result value:* Return a string consisting of the given character

*Result type:* <string>

*Purity of the procedure:* pure

**integer->string**

*Syntax:*

(integer->string obj)

*Arguments:*

Name: obj  
Type: <integer>  
Description: An integer value

*Result value:* Output string for the given value

*Result type:* <string>

*Purity of the procedure:* pure

## null->string

*Syntax:*

```
(null->string obj)
```

*Arguments:*

Name: obj  
Type: <null>  
Description: null

*Result value:* ()

*Result type:* <string>

*Purity of the procedure:* pure

## real->string

*Syntax:*

```
(real->string obj)
```

*Arguments:*

Name: obj  
Type: <real>  
Description: A real value

*Result value:* Output string for the given value

*Result type:* <string>

*Purity of the procedure:* pure

## string->string

*Syntax:*

(string->string obj)

*Arguments:*

Name: obj  
Type: <string>  
Description: A string

*Result value:* The same string as the argument

*Result type:* <string>

*Purity of the procedure:* pure

symbol->string

*Syntax:*

(symbol->string obj)

*Arguments:*

Name: obj  
Type: <symbol>  
Description: A symbol

*Result value:* Output string for the given value

*Result type:* <string>

*Purity of the procedure:* pure

pair->string

*Syntax:*

(pair->string obj)

*Arguments:*

Name: p  
Type: <pair>  
Description: A pair

*Result value:* Output string for the given value



*Result type:* <string>

*Purity of the procedure:* pure

This procedure converts pairs, lists and tree structures implemented with pairs to strings. Method `object->string` is called recursively for the contents. This procedure should be safe for cyclic structures.

## 11.2 Methods

```
object->string: (<object>) → <string> pure = general-object->string
object->string: (<integer>) → <string> pure = integer->string
object->string: (<real>) → <string> pure = real->string
object->string: (<string>) → <string> pure = string->string
object->string: (<symbol>) → <string> pure = symbol->string
object->string: (<null>) → <string> pure = null->string
object->string: (<character>) → <string> pure = character->string
object->string: (<boolean>) → <string> pure = boolean->string
object->string: (<pair>) → <string> pure = pair->string
```



## Chapter 12

# Module (standard-library text-file-io)

### 12.1 Data Types

*Data type name:* <input-port>

*Type:* <class>

*Description:* An input port (input file)

*Data type name:* <output-port>

*Type:* <class>

*Description:* An output port (output file)

### 12.2 Simple Procedures

#### character-ready?

*Syntax:*

(character-ready? input-port)

*Arguments:*

Name: input-port

Type: <input-port>

Description: The input port to check

*Result value:* #t iff there is a character ready in the given input port

*Result type:* <boolean>

On i/o error exception (`io-error character-ready?:runtime-error filename`) is raised.

## close-input-port

*Syntax:*

```
(close-input-port input-port)
```

*Arguments:*

Name: `input-port`

Type: `<input-port>`

Description: The input port to be closed

No result value.

## close-output-port

*Syntax:*

```
(close-output-port output-port)
```

*Arguments:*

Name: `output-port`

Type: `<output-port>`

Description: The output port to be closed

No result value.

## current-input-port

*Syntax:*

```
(current-input-port)
```

No arguments.

*Result value:* The current input port

*Result type:* `<input-port>`

## current-output-port

*Syntax:*

```
(current-output-port)
```

No arguments.

*Result value:* The current output port

*Result type:* <output-port>

## write-character

*Syntax:*

```
(write-character output-port ch)
```

*Arguments:*

Name: output-port

Type: <output-port>

Description: An output port where to write

Name: ch

Type: <character>

Description: A character to be written

No result value.

This procedure writes the external representation of a character in the specified output port. If the operation fails an exception (`io-error` `error-displaying-object filename`) is raised.

## write-string

*Syntax:*

```
(write-string output-port str)
```

*Arguments:*

Name: output-port

Type: <output-port>

Description: An output port where to write

Name: `str`  
Type: `<string>`  
Description: A string to be written

No result value.

This procedure writes the external representation of a string into the specified output port. If the operation fails an exception (`io-error` `error-displaying-object filename`) is raised.

## `write-line`

*Syntax:*

```
(write-line output-port obj)
```

*Arguments:*

Name: `output-port`  
Type: `<output-port>`  
Description: An output port where to write

Name: `obj`  
Type: `<object>`  
Description: An object to be written

No result value.

This function uses the generic procedure `write` to write the object and prints a newline after that.

## `display-character`

*Syntax:*

```
(display-character output-port ch)
```

*Arguments:*

Name: `output-port`  
Type: `<output-port>`  
Description: An output port where to display

Name: `ch`  
Type: `<character>`

Description: A character to be displayed

No result value.

This procedure displays a character into the given output port. If the operation fails an exception (`io-error` `error-displaying-object` *filename*) is raised.

## display-string

*Syntax:*

```
(display-string output-port str)
```

*Arguments:*

Name: `output-port`  
Type: `<output-port>`  
Description: An output port where to display

Name: `str`  
Type: `<string>`  
Description: A string to be displayed

No result value.

This procedure displays a string into the given output port. If the operation fails an exception (`io-error` `error-displaying-object` *filename*) is raised.

## display-line

*Syntax:*

```
(display-line output-port obj)
```

*Arguments:*

Name: `output-port`  
Type: `<output-port>`  
Description: An output port where to display

Name: `obj`  
Type: `<object>`  
Description: An object to be displayed

No result value.

This function uses the generic procedure `display` to display the object and prints a newline.

## `newline`

*Syntax:*

```
(newline output-port)
```

*Arguments:*

Name: `output-port`

Type: `<output-port>`

Description: An output port where to print

No result value.

This procedure prints a newline to the given output port. If the operation fails an exception (`io-error error-displaying-newline filename`) is raised.

## `open-input-file`

*Syntax:*

```
(open-input-file filename)
```

*Arguments:*

Name: `filename`

Type: `<string>`

Description: Name of the file to be opened

*Result value:* An object representing the opened file

*Result type:* `<input-port>`

This procedure opens an input file. If the operation fails an exception (`io-error error-opening-input-file filename`) is raised.

## `open-output-file`

*Syntax:*



(open-output-file filename)

*Arguments:*

Name: `filename`  
Type: `<string>`  
Description: Name of the file to be opened

*Result value:* An object representing the opened file

*Result type:* `<output-port>`

This procedure opens an output file. If the operation fails an exception (`io-error error-opening-output-file filename`) is raised.

## peek-character

*Syntax:*

(peek-character input-port)

*Arguments:*

Name: `input-port`  
Type: `<input-port>`  
Description: An input port where to read from

*Result value:* The read character or an eof object

*Result type:* `(:union <character> <eof>)`

This procedure peeks a character from an input port. On i/o error an exception (`io-error peek-character:io-error filename`) is raised.

## read

*Syntax:*

(read input-port)

*Arguments:*

Name: `input-port`  
Type: `<input-port>`  
Description: An input port where to read from

*Result value:* The object read or an eof object

*Result type:* <object>

This procedure reads a Theme-D expression from an input port. The Theme-D runtime environment checks that the result object does not contain any data types unknown to Theme-D. On i/o error an exception (`io-error read:io-error filename`) is raised. If a Scheme vector constant is encountered in the data raise exception (`io-error io:illegal-vector filename`). If a Scheme complex number is encountered in the data raise exception (`io-error io:illegal-complex-number filename`). On some other Scheme object whose data type is not known by Theme-D raise exception (`io-error io:illegal-data-type filename`).

## read-all

*Syntax:*

```
(read-all input-port)
```

*Arguments:*

Name: `input-port`

Type: <input-port>

Description: An input port where to read from

*Result value:* The objects read

*Result type:* <object>

This procedure uses procedure `read` to read all the expressions from the given input-port.

## read-character

*Syntax:*

```
(read-character input-port)
```

*Arguments:*

Name: `input-port`

Type: <input-port>

Description: An input port where to read from

*Result value:* The read character or an eof object

*Result type:* (:union <character> <eof>)

This procedure reads a character from an input port. On i/o error an exception (`io-error read-character:io-error filename`) is raised.

## read-line

*Syntax:*

```
(read-line input-port)
```

*Arguments:*

Name: `input-port`  
Type: `<input-port>`  
Description: An input port where to read from

*Result value:* Contents of a line

*Result type:* `<string>`

This procedure reads a single line from the given input port as a string. On i/o error an exception (`io-error read-character:io-error filename`) is raised.

## read-string

*Syntax:*

```
(read-string input-port)
```

*Arguments:*

Name: `input-port`  
Type: `<input-port>`  
Description: An input port where to read from

*Result value:* The contents of the file

*Result type:* `<string>`

This procedure reads the contents of the given input port as a single string.

## call-with-input-string

*Syntax:*

```
(call-with-input-string str proc)
```

*Arguments:*

Name: `str`

Type: `<string>`  
 Description: A string where to read from

Name: `proc`  
 Type: `(:procedure (<input-port>) <object> nonpure)`  
 Description: A procedure to call

*Result value:* The object returned by the procedure  
*Result type:* `<object>`

This procedure creates an input port whose input comes from the argument string and passes it to the given procedure.

## call-with-output-string

*Syntax:*

```
(call-with-output-string proc)
```

*Arguments:*

Name: `proc`  
 Type: `(:procedure (<output-port>) <none> nonpure)`  
 Description: A procedure to call

*Result value:* A string consisting of the output of the procedure  
*Result type:* `<string>`

This procedure creates an output port and passes it to the given procedure. A string consisting of the output of the procedure into the port is returned.

## 12.3 Methods

```
display: (<output-port> <object>) → <none>
display: (<output-port> <null>) → <none>
display: (<output-port> <boolean>) → <none>
display: (<output-port> <integer>) → <none>
display: (<output-port> <real>) → <none>
display: (<output-port> <string>) → <none>
display: (<output-port> <character>) → <none>
display: (<output-port> <symbol>) → <none>
display: (<output-port> <pair>) → <none>
```

These methods display an object as a string in the specified port.

```
write: (<output-port> <object>) → <none>
write: (<output-port> <null>) → <none>
write: (<output-port> <boolean>) → <none>
```

```
write: (<output-port> <integer>) → <none>  
write: (<output-port> <real>) → <none>  
write: (<output-port> <string>) → <none>  
write: (<output-port> <character>) → <none>  
write: (<output-port> <symbol>) → <none>  
write: (<output-port> <pair>) → <none>
```

These methods write the external representation of an object to the specified port for primitive objects.

You may define methods `write` and `display` for your own classes. It is usually better to define method `object->string` instead of method `display`. Note that generic procedure `write` calls `display` by default.



## Chapter 13

# Module (standard-library console-io)

This module implements input and output for the standard input and standard output.

### 13.1 Simple Procedures

#### console-character-ready?

*Syntax:*

```
(console-character-ready?)
```

No arguments.

*Result value:* #t iff there is a character ready in the standard input

*Result type:* <boolean>

#### console-display

*Syntax:*

```
(console-display obj)
```

*Arguments:*

Name: obj

Type: <object>

Description: An object to be displayed

No result value.

This function uses the procedure `atom-to-string` to obtain the string representation of the object and displays the string with procedure `console-display-string`.

## `console-display-character`

*Syntax:*

```
(console-display-character ch)
```

*Arguments:*

Name: `ch`  
Type: `<character>`  
Description: A character to be displayed

No result value.

## `console-display-line`

*Syntax:*

```
(console-display-line obj)
```

*Arguments:*

Name: `obj`  
Type: `<object>`  
Description: An object to be displayed

No result value.

This function uses the procedure `atom-to-string` to obtain the string representation of the object and displays the string with procedure `console-display-string`. A newline is displayed after the object.

## `console-display-string`

*Syntax:*



```
(console-display-string str)
```

*Arguments:*

Name: `str`  
Type: `<string>`  
Description: A string to be displayed

No result value.

## console-newline

*Syntax:*

```
(console-newline)
```

No arguments.

No result value.

This procedure prints a newline to the standard output.

## console-read

*Syntax:*

```
(console-read)
```

No arguments.

*Result value:* The object read or an eof object

*Result type:* `<object>`

The Theme-D runtime environment checks that the result object does not contain any data types unknown to Theme-D.

## console-read-character

*Syntax:*

```
(console-read-character)
```

No arguments.

*Result value:* The read character or an eof object

*Result type:* (:union <character> <eof>)

This procedure reads a character from the standard input.

## console-write

*Syntax:*

```
(console-write obj)
```

*Arguments:*

Name: `obj`

Type: <object>

Description: An object to be written

No result value.

This function uses the procedure `atom-to-string` to obtain the source code representation of the object and displays the string with procedure `console-display-string`.

## console-write-line

*Syntax:*

```
(console-write-line obj)
```

*Arguments:*

Name: `obj`

Type: <object>

Description: An object to be written

No result value.

This function uses the procedure `atom-to-string` to obtain the source code representation of the object and displays the string with procedure `console-display-string`. A newline is written after the object.

## Chapter 14

# Module (standard-library system)

### 14.1 Simple Procedures

#### delete-file

*Syntax:*

```
(delete-file str-filename)
```

*Arguments:*

Name: `str-filename`

Type: `<string>`

Description: The name of the file to be deleted

No result value.

*Purity of the procedure:* nonpure

This procedure deletes the named file. If the file does not exist an exception in raised.

#### file-exists?

*Syntax:*

```
(file-exists? str-filename)
```

*Arguments:*

Name: `str-filename`  
Type: `<string>`  
Description: The name of the file

*Result value:* Returns `#t` iff the file exists

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `getenv`

*Syntax:*

`(getenv str-var-name)`

*Arguments:*

Name: `str-var-name`  
Type: `<string>`  
Description: The name of the environment variable

*Result value:* The value of the given environment variable

*Result type:* `(:maybe <string>)`

*Purity of the procedure:* pure

If the environment variable does not exist return `()`.

# Chapter 15

## Module (standard-library rational)

### 15.1 Data Types

*Data type name:* <rational>

*Type:* <class>

*Description:* A rational number

Class <rational> is immutable, equal by value, and not inheritable.

*Data type name:* <rational-number>

*Type:* :union

*Description:* A rational valued number

Type <rational-number> is equal to the union of <rational> and <integer>.

### 15.2 Simple Procedures

rational

*Syntax:*

(rational i-numer i-denom)

*Arguments:*

Name: i-numer

Type: <integer>

Description: The numerator

Name: `i-denom`  
Type: `<integer>`  
Description: The denominator

*Result value:* The quotient of `i-numer` and `i-denom`  
*Result type:* `<rational>`

*Purity of the procedure:* pure

This procedure returns the rational number in simplified form. If the denominator is zero a numerical overflow exception is raised.

## numerator

*Syntax:*

```
(numerator rat)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational number

*Result value:* The numerator of the argument  
*Result type:* `<integer>`

*Purity of the procedure:* pure

## denominator

*Syntax:*

```
(denominator rat)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational number

*Result value:* The denominator of the argument  
*Result type:* `<integer>`

*Purity of the procedure:* pure

**rational=?**

*Syntax:*

(rational=? rat1 rat2)

*Arguments:*

Name: rat1  
Type: <rational>  
Description: A rational value to be compared

Name: rat2  
Type: <rational>  
Description: A rational value to be compared

*Result value:* #t iff rat1 is equal to rat2

*Result type:* <boolean>

*Purity of the procedure:* pure

Rational numbers  $a/b$  and  $c/d$  are equal iff  $ad = bc$ .

**rational=**

*Syntax:*

(rational= rat1 rat2)

*Arguments:*

Name: rat1  
Type: <rational>  
Description: A rational value to be compared

Name: rat2  
Type: <rational>  
Description: A rational value to be compared

*Result value:* #t iff rat1 is numerically equal to rat2

*Result type:* <boolean>

*Purity of the procedure:* pure

Rational numbers  $a/b$  and  $c/d$  are equal iff  $ad = bc$ .

## rational-integer=

*Syntax:*

```
(rational-integer= rat i)
```

*Arguments:*

Name: `rat`

Type: `<rational>`

Description: A rational value to be compared

Name: `i`

Type: `<integer>`

Description: An integer value to be compared

*Result value:* `#t` iff `rat` is numerically equal to `i`

*Result type:* `<boolean>`

Rational number  $a/b$  and integer  $c$  are equal iff  $a = bc$ .

*Purity of the procedure:* pure

## integer-rational=

*Syntax:*

```
(integer-rational= i rat)
```

*Arguments:*

Name: `i`

Type: `<integer>`

Description: An integer value to be compared

Name: `rat`

Type: `<rational>`

Description: A rational value to be compared

*Result value:* `#t` iff `i` is numerically equal to `rat`



*Result type:* <boolean>

## rational<

*Syntax:*

```
(rational< rat1 rat2)
```

*Arguments:*

Name: rat1  
Type: <rational>  
Description: A rational value to be compared

Name: rat2  
Type: <rational>  
Description: A rational value to be compared

*Result value:* #t iff rat1 is less than rat2

*Result type:* <boolean>

*Purity of the procedure:* pure

## rational-integer<

*Syntax:*

```
(rational-integer< rat i)
```

*Arguments:*

Name: rat  
Type: <rational>  
Description: A rational value to be compared

Name: i  
Type: <integer>  
Description: An integer value to be compared

*Result value:* #t iff rat is less than i

*Result type:* <boolean>

*Purity of the procedure:* pure

**integer-rational<**

*Syntax:*

```
(integer-rational< i rat)
```

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value to be compared

Name: `rat`  
Type: `<rational>`  
Description: A rational value to be compared

*Result value:* `#t` iff `i` is less than `rat`

*Result type:* `<boolean>`

**rational<=**

*Syntax:*

```
(rational<= rat1 rat2)
```

*Arguments:*

Name: `rat1`  
Type: `<rational>`  
Description: A rational value to be compared

Name: `rat2`  
Type: `<rational>`  
Description: A rational value to be compared

*Result value:* `#t` iff `rat1` is less than or equal to `rat2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

**rational-integer<=**

*Syntax:*

```
(rational-integer<= rat i)
```

*Arguments:*

Name: `rat`  
 Type: `<rational>`  
 Description: A rational value to be compared

Name: `i`  
 Type: `<integer>`  
 Description: An integer value to be compared

*Result value:* `#t` iff `rat` is less than or equal to `i`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## integer-rational<=

*Syntax:*

```
(integer-rational<= i rat)
```

*Arguments:*

Name: `i`  
 Type: `<integer>`  
 Description: An integer value to be compared

Name: `rat`  
 Type: `<rational>`  
 Description: A rational value to be compared

*Result value:* `#t` iff `i` is less than or equal to `rat`

*Result type:* `<boolean>`

## rational>

*Syntax:*

```
(rational> rat1 rat2)
```

*Arguments:*

Name: `rat1`  
Type: `<rational>`  
Description: A rational value to be compared

Name: `rat2`  
Type: `<rational>`  
Description: A rational value to be compared

*Result value:* `#t` iff `rat1` is greater than `rat2`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

### `rational-integer>`

*Syntax:*

```
(rational-integer> rat i)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value to be compared

Name: `i`  
Type: `<integer>`  
Description: An integer value to be compared

*Result value:* `#t` iff `rat` is greater than `i`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

### `integer-rational>`

*Syntax:*

```
(integer-rational> i rat)
```

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value to be compared

Name: `rat`  
Type: `<rational>`  
Description: A rational value to be compared

*Result value:* `#t` iff `i` is greater than `rat`  
*Result type:* `<boolean>`

`rational>=`

*Syntax:*

```
(rational>= rat1 rat2)
```

*Arguments:*

Name: `rat1`  
Type: `<rational>`  
Description: A rational value to be compared

Name: `rat2`  
Type: `<rational>`  
Description: A rational value to be compared

*Result value:* `#t` iff `rat1` is greater than or equal to `rat2`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

`rational-integer>=`

*Syntax:*

```
(rational-integer>= rat i)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value to be compared

Name: `i`  
Type: `<integer>`  
Description: An integer value to be compared

*Result value:* `#t` iff `rat` is greater than or equal to `i`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `integer-rational>=`

*Syntax:*

```
(integer-rational>= i rat)
```

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value to be compared

Name: `rat`  
Type: `<rational>`  
Description: A rational value to be compared

*Result value:* `#t` iff `i` is greater than or equal to `rat`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `simplify-rational`

*Syntax:*

```
(simplify-rational rat)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The argument in the simplified form

*Result type:* <rational>

*Purity of the procedure:* pure

See chapter 3.

## rat-integer-valued?

*Syntax:*

```
(rat-integer-valued? rat)
```

*Arguments:*

Name: rat

Type: <rational>

Description: A rational value

*Result value:* #t iff the argument is integer valued

*Result type:* <boolean>

*Purity of the procedure:* pure

A rational number is integer valued iff its denominator is 1 in the simplified form.

## rat-sign

*Syntax:*

```
(rat-sign rat)
```

*Arguments:*

Name: rat

Type: <rational>

Description: A rational value

*Result value:* The sign of the argument

*Result type:* <integer>

*Purity of the procedure:* pure

Return 0 if `rat = 0`, 1 if `rat > 0`, and -1 if `rat < 0`.

**integer->rational**

*Syntax:*

```
(integer->rational i)
```

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value

*Result value:* The argument converted to a rational number

*Result type:* `<rational>`

*Purity of the procedure:* pure

The numerator of the result is `i` and the denominator 1.

**rational->integer**

*Syntax:*

```
(rational->integer rat)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The argument converted to an integer number

*Result type:* `<integer>`

*Purity of the procedure:* pure

If the argument is not integer valued an exception `rational->integer:not-an-integer` is raised.

**rat-zero**

*Syntax:*

```
(rat-zero)
```



No arguments.

*Result value:* The rational number 0

*Result type:* <rational>

*Purity of the procedure:* pure

The numerator of the result is 0 and the denominator 1.

## rat-one

*Syntax:*

```
(rat-one)
```

No arguments.

*Result value:* The rational number 1

*Result type:* <rational>

*Purity of the procedure:* pure

The numerator and the denominator of the result are 1.

## rat-zero?

*Syntax:*

```
(rat-zero? rat)
```

*Arguments:*

Name: rat

Type: <rational>

Description: A rational value

*Result value:* #t iff the argument is equal to 0

*Result type:* <boolean>

*Purity of the procedure:* pure

A numerical overflow exception is raised if the denominator of the argument is 0.

**rat-one?**

*Syntax:*

```
(rat-one? rat)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* `#t` iff the argument is equal to 1

*Result type:* `<boolean>`

*Purity of the procedure:* pure

A numerical overflow exception is raised if the denominator of the argument is 0.

**rational+**

*Syntax:*

```
(rational+ rat1 rat2)
```

*Arguments:*

Name: `rat1`  
Type: `<rational>`  
Description: A rational value

Name: `rat2`  
Type: `<rational>`  
Description: A rational value

*Result value:* The sum of the arguments in the simplified form

*Result type:* `<rational>`

*Purity of the procedure:* pure

**rational-integer+**

*Syntax:*

```
(rational-integer+ rat i)
```

*Arguments:*

Name: `rat`  
 Type: `<rational>`  
 Description: A rational value

Name: `i`  
 Type: `<integer>`  
 Description: An integer value

*Result value:* The sum of the arguments in the simplified form

*Result type:* `<rational>`

*Purity of the procedure:* pure

## integer-rational+

*Syntax:*

```
(integer-rational+ i rat)
```

*Arguments:*

Name: `i`  
 Type: `<integer>`  
 Description: An integer value

Name: `rat`  
 Type: `<rational>`  
 Description: A rational value

*Result value:* The sum of the arguments in the simplified form

*Result type:* `<rational>`

*Purity of the procedure:* pure

## rational-

*Syntax:*

```
(rational- rat1 rat2)
```

*Arguments:*

Name: `rat1`  
Type: `<rational>`  
Description: A rational value

Name: `rat2`  
Type: `<rational>`  
Description: A rational value

*Result value:* The difference of the arguments in the simplified form

*Result type:* `<rational>`

*Purity of the procedure:* pure

**rational-integer-***Syntax:*

`(rational-integer- rat i)`

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value

Name: `i`  
Type: `<integer>`  
Description: An integer value

*Result value:* The difference of the arguments in the simplified form

*Result type:* `<rational>`

*Purity of the procedure:* pure

**integer-rational-***Syntax:*

`(integer-rational- i rat)`

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The difference of the arguments in the simplified form

*Result type:* `<rational>`

*Purity of the procedure:* pure

## `rational*`

*Syntax:*

```
(rational* rat1 rat2)
```

*Arguments:*

Name: `rat1`  
Type: `<rational>`  
Description: A rational value

Name: `rat2`  
Type: `<rational>`  
Description: A rational value

*Result value:* The product of the arguments in the simplified form

*Result type:* `<rational>`

*Purity of the procedure:* pure

## `rational-integer*`

*Syntax:*

```
(rational-integer* rat i)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value

Name: `i`  
Type: `<integer>`  
Description: An integer value

*Result value:* The product of the arguments in the simplified form  
*Result type:* `<rational>`

*Purity of the procedure:* pure

## `integer-rational*`

*Syntax:*

`(integer-rational* i rat)`

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The product of the arguments in the simplified form  
*Result type:* `<rational>`

*Purity of the procedure:* pure

## `rational/`

*Syntax:*

`(rational/ rat1 rat2)`

*Arguments:*

Name: `rat1`

Type: <rational>  
 Description: A rational value

Name: `rat2`  
 Type: <rational>  
 Description: A rational value

*Result value:* The quotient of the arguments in the simplified form

*Result type:* <rational>

*Purity of the procedure:* pure

If the divisor is equal to 0 raise a numerical overflow exception.

### `rational-integer/`

*Syntax:*

`(rational-integer/ rat i)`

*Arguments:*

Name: `rat`  
 Type: <rational>  
 Description: A rational value

Name: `i`  
 Type: <integer>  
 Description: An integer value

*Result value:* The quotient of the arguments in the simplified form

*Result type:* <rational>

*Purity of the procedure:* pure

If the divisor is equal to 0 raise a numerical overflow exception.

### `integer-rational/`

*Syntax:*

`(integer-rational/ i rat)`

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The quotient of the arguments in the simplified form  
*Result type:* `<rational>`

*Purity of the procedure:* pure

If the divisor is equal to 0 raise a numerical overflow exception.

## `rat-neg`

*Syntax:*

```
(rat-neg rat)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The opposite of the argument  
*Result type:* `<rational>`

*Purity of the procedure:* pure

## `rat-abs`

*Syntax:*

```
(rat-abs rat)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value



*Result value:* The absolute value of the argument

*Result type:* <rational>

*Purity of the procedure:* pure

## rat-square

*Syntax:*

```
(rat-square rat)
```

*Arguments:*

Name: rat

Type: <rational>

Description: A rational value

*Result value:* The square of the argument

*Result type:* <rational>

*Purity of the procedure:* pure

## rat-inverse

*Syntax:*

```
(rat-inverse rat)
```

*Arguments:*

Name: rat

Type: <rational>

Description: A rational value

*Result value:* The inverse of the argument

*Result type:* <rational>

*Purity of the procedure:* pure

If the argument is equal to 0 a numerical overflow exception is raised.

## rat-nonneg-int-expt

*Syntax:*

```
(rat-nonneg-int-expt rat-base i-exponent)
```

*Arguments:*

Name: `rat-base`  
Type: `<rational>`  
Description: A rational value

Name: `i-exponent`  
Type: `<integer>`  
Description: A nonnegative integer value

*Result value:* `rat-base` raised to the power `i-exponent`

*Result type:* `<rational>`

*Purity of the procedure:* pure

## rat-int-expt

*Syntax:*

```
(rat-int-expt rat-base i-exponent)
```

*Arguments:*

Name: `rat-base`  
Type: `<rational>`  
Description: A rational value

Name: `i-exponent`  
Type: `<integer>`  
Description: An integer value

*Result value:* `rat-base` raised to the power `i-exponent`

*Result type:* `<rational>`

*Purity of the procedure:* pure

If `rat-base` is equal to 0 and `i-exponent` is negative raise a numerical overflow exception.

## i-expt

*Syntax:*

```
(i-expt i-base i-exponent)
```

*Arguments:*

Name: `i-base`  
 Type: `<integer>`  
 Description: An integer number

Name: `i-exponent`  
 Type: `<integer>`  
 Description: An integer number

*Result value:* `i-base` raised to the power `i-exponent`

*Result type:* `<rational-number>`

*Purity of the procedure:* pure

If the base is 0 and the exponent is negative raise a numerical overflow exception. If the exponent is nonnegative the result is always an `<integer>`.

## rat-log10-exact

*Syntax:*

```
(rat-log10-exact rat)
```

*Arguments:*

Name: `rat`  
 Type: `<rational>`  
 Description: A rational number

*Result value:* The base 10 logarithm of the argument or `null` if the logarithm is not an integer

*Result type:* `(:maybe <integer>)`

*Purity of the procedure:* pure

This procedure is able compute logarithms of the negative integer powers of 10, too.

## rat-log2-exact

*Syntax:*

```
(rat-log2-exact rat)
```

*Arguments:*

Name: `rat`  
 Type: `<rational>`  
 Description: A rational number

*Result value:* The base 2 logarithm of the argument or `null` if the logarithm is not an integer

*Result type:* `(<maybe <integer>)`

*Purity of the procedure:* pure

This procedure is able compute logarithms of the negative integer powers of 2, too.

## rational-to-string

*Syntax:*

```
(rational-to-string rat repr?)
```

*Arguments:*

Name: `rat`  
 Type: `<rational>`  
 Description: A rational value

Name: `repr?`  
 Type: `<boolean>`  
 Description: Should we write a representation or a user-friendly value

*Result value:* The argument value converted to a string

*Result type:* `<string>`

*Purity of the procedure:* pure

## 15.3 Methods

`equal?: (<rational> <rational>) → <boolean>` pure = `rational=?`

```

=: (<rational> <rational>) → <boolean> pure    =    rational=
=: (<rational> <integer>) → <boolean> pure    =    rational-integer=
=: (<integer> <rational>) → <boolean> pure    =    integer-rational=

<: (<rational-number> <rational-number>) → <boolean>  pure abstract
<: (<rational> <rational>) → <boolean> pure    =    rational<
<: (<rational> <integer>) → <boolean> pure    =    rational-integer<
<: (<integer> <rational>) → <boolean> pure    =    integer-rational<

<=: (<rational-number> <rational-number>) → <boolean>  pure abstract
<=: (<rational> <rational>) → <boolean> pure    =    rational<=
<=: (<rational> <integer>) → <boolean> pure    =    rational-integer<=
<=: (<integer> <rational>) → <boolean> pure    =    integer-rational<=

>: (<rational-number> <rational-number>) → <boolean>  pure abstract
>: (<rational> <rational>) → <boolean> pure    =    rational>
>: (<rational> <integer>) → <boolean> pure    =    rational-integer>
>: (<integer> <rational>) → <boolean> pure    =    integer-rational>

>=: (<rational-number> <rational-number>) → <boolean>  pure abstract
>=: (<rational> <rational>) → <boolean> pure    =    rational>=
>=: (<rational> <integer>) → <boolean> pure    =    rational-integer>=
>=: (<integer> <rational>) → <boolean> pure    =    integer-rational>=

+: (<rational-number> <rational-number>) → <rational-number>  pure
abstract
+: (<rational> <rational>) → <rational> pure    =    rational+
+: (<rational> <integer>) → <rational> pure    =    rational-integer+
+: (<integer> <rational>) → <rational> pure    =    integer-rational+

-: (<rational-number> <rational-number>) → <rational-number>  pure
abstract
-: (<rational> <rational>) → <rational> pure    =    rational-
-: (<rational> <integer>) → <rational> pure    =    rational-integer-
-: (<integer> <rational>) → <rational> pure    =    integer-rational-

*: (<rational-number> <rational-number>) → <rational-number>  pure
abstract
*: (<rational> <rational>) → <rational> pure    =    rational*
*: (<rational> <integer>) → <rational> pure    =    rational-integer*
*: (<integer> <rational>) → <rational> pure    =    integer-rational*

/: (<rational-number> <rational-number>) → <rational-number>  pure
abstract
/: (<rational> <rational>) → <rational> pure    =    rational/
/: (<rational> <integer>) → <rational> pure    =    rational-integer/
/: (<integer> <rational>) → <rational> pure    =    integer-rational/

-: (<rational-number>) → <rational-number>  pure abstract
-: (<rational>) → <rational> pure    =    rat-neg

```

```
abs: (<rational-number>) → <rational-number> pure abstract  
abs: (<rational>) → <rational> pure = rat-abs
```

```
square: (<rational-number>) → <rational-number> pure abstract  
square: (<rational>) → <rational> pure = rat-square
```

```
sign: (<rational-number>) → <rational-number> pure abstract  
sign: (<rational>) → <integer> pure = rat-sign
```

```
atom-to-string: (<rational> <boolean>) → <string> pure = rational-to-string
```

# Chapter 16

## Module (standard-library real-math)

### 16.1 Data Types

*Data type name:* <real-number>

*Type:* :union

*Description:* A real valued number

Type <real-number> is equal to the union of <real>, <rational>, and <integer>.

### 16.2 Constants

The following constants are defined:

- `gl-r-pi`: The value  $\pi$
- `gl-r-pi/2`: The value  $\pi/2$
- `gl-r-pi/4`: The value  $\pi/4$
- `gl-r-1/pi`: The value  $1/\pi$
- `gl-r-2/pi`: The value  $2/\pi$
- `gl-r-2/sqrtpi`: The value  $1/\sqrt{\pi}$
- `gl-r-sqrt2`: The value  $\sqrt{2}$
- `gl-r-1/sqrt2`: The value  $1/\sqrt{2}$
- `gl-r-e`: The value  $e$  (Napier's constant)
- `gl-r-log2e`: The value  $\log_2 e$
- `gl-r-log10e`: The value  $\log_{10} e$

- `gl-r-ln2`: The value  $\ln 2$
- `gl-r-ln10`: The value  $\ln 10$
- `gl-r-pi/ln2`: The value  $\pi/\ln 2$
- `gl-r-pi/ln10`: The value  $\pi/\ln 10$

### 16.3 Simple Procedures

#### `rational->real`

*Syntax:*

`(rational->real rat)`

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The rational value converted to a real value

*Result type:* `<real>`

*Purity of the procedure:* pure

#### `real-rational=`

*Syntax:*

`(real-rational= r rat)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value to be compared

Name: `rat`  
Type: `<rational>`  
Description: A rational value to be compared



*Result value:* #t iff `r` is numerically equal to `rat`

*Result type:* <boolean>

*Purity of the procedure:* pure

## rational-real=

*Syntax:*

```
(rational-real= rat r)
```

*Arguments:*

Name: `rat`

Type: <rational>

Description: A rational value to be compared

Name: `r`

Type: <real>

Description: A real value to be compared

*Result value:* #t iff `rat` is numerically equal to `r`

*Result type:* <boolean>

*Purity of the procedure:* pure

## real-rational<

*Syntax:*

```
(real-rational< r rat)
```

*Arguments:*

Name: `r`

Type: <real>

Description: A real value to be compared

Name: `rat`

Type: <rational>

Description: A rational value to be compared

*Result value:* #t iff `r` is less than `rat`

*Result type:* <boolean>

*Purity of the procedure:* pure

## rational-real<

*Syntax:*

(rational-real< rat r)

*Arguments:*

Name: `rat`

Type: <rational>

Description: A rational value to be compared

Name: `r`

Type: <real>

Description: A real value to be compared

*Result value:* #t iff `rat` is less than `r`

*Result type:* <boolean>

*Purity of the procedure:* pure

## real-rational<=

*Syntax:*

(real-rational<= r rat)

*Arguments:*

Name: `r`

Type: <real>

Description: A real value to be compared

Name: `rat`

Type: <rational>

Description: A rational value to be compared

*Result value:* #t iff `r` is less than or equal to `rat`

*Result type:* <boolean>

*Purity of the procedure:* pure

### rational-real<=

*Syntax:*

```
(rational-real<= rat r)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value to be compared

Name: `r`  
Type: `<real>`  
Description: A real value to be compared

*Result value:* `#t` iff `rat` is less than or equal to `r`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

### real-rational>

*Syntax:*

```
(real-rational> r rat)
```

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value to be compared

Name: `rat`  
Type: `<rational>`  
Description: A rational value to be compared

*Result value:* `#t` iff `r` is greater than `rat`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## rational-real>

*Syntax:*

```
(rational-real> rat r)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value to be compared

Name: `r`  
Type: `<real>`  
Description: A real value to be compared

*Result value:* `#t` iff `rat` is greater than `r`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## real-rational>=

*Syntax:*

```
(real-rational>= r rat)
```

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value to be compared

Name: `rat`  
Type: `<rational>`  
Description: A rational value to be compared

*Result value:* `#t` iff `r` is greater than or equal to `rat`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

**rational-real>=***Syntax:*`(rational-real>= rat r)`*Arguments:*

Name: `rat`  
 Type: `<rational>`  
 Description: A rational value to be compared

Name: `r`  
 Type: `<real>`  
 Description: A real value to be compared

*Result value:* `#t` iff `rat` is greater than or equal to `r`*Result type:* `<boolean>`*Purity of the procedure:* pure**real-rational+***Syntax:*`(real-rational+ r rat)`*Arguments:*

Name: `r`  
 Type: `<real>`  
 Description: A real value

Name: `rat`  
 Type: `<rational>`  
 Description: A rational value

*Result value:* The sum of the arguments*Result type:* `<real>`*Purity of the procedure:* pure**rational-real+**

*Syntax:*

```
(rational-real+ rat r)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value

Name: `r`  
Type: `<real>`  
Description: A real value

*Result value:* The sum of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

## real-rational-

*Syntax:*

```
(real-rational- r rat)
```

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The difference of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

## rational-real-

*Syntax:*

(rational-real- rat r)

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value

Name: `r`  
Type: `<real>`  
Description: A real value

*Result value:* The difference of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

**real-rational\***

*Syntax:*

(real-rational\* r rat)

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The product of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

**rational-real\***

*Syntax:*

(rational-real\* rat r)

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value

Name: `r`  
Type: `<real>`  
Description: A real value

*Result value:* The product of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

`real-rational/`*Syntax:*

`(real-rational/ r rat)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The quotient of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

In case `rat` is equal to 0 return

- `inf`, if `rat`  $>$  0
- `-inf`, if `rat`  $<$  0
- `NaN`, if `rat` = 0

`rational-real/`



*Syntax:*

```
(rational-real/ rat r)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: A rational value

Name: `r`  
Type: `<real>`  
Description: A real value

*Result value:* The quotient of the arguments

*Result type:* `<real>`

*Purity of the procedure:* pure

In case `r` is equal to 0 return

- `inf`, if `r > 0`
- `-inf`, if `r < 0`
- `NaN`, if `r = 0`

## r-sqrt

*Syntax:*

```
(r-sqrt r)
```

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real number

*Result value:* Square root of the argument

*Result type:* `<real>`

*Purity of the procedure:* pure

If the argument is negative return `NaN`.

## r-expt

*Syntax:*

(r-expt x y)

*Arguments:*

Name: x  
Type: <real>  
Description: A real number

Name: y  
Type: <real>  
Description: A real number

*Result value:* x to the power of y

*Result type:* <real>

*Purity of the procedure:* pure

If x is less than 0 return NaN. If x is equal to 0 return 1.0.

## r-exp

*Syntax:*

(r-exp r)

*Arguments:*

Name: r  
Type: <real>  
Description: A real number

*Result value:* e to the power of r

*Result type:* <real>

*Purity of the procedure:* pure

Number e is the base of natural logarithms (approx. 2.718).

## r-log

*Syntax:*

(r-log r)

*Arguments:*

Name: **r**  
Type: **<real>**  
Description: A real number

*Result value:* The natural logarithm of **r**  
*Result type:* **<real>**

*Purity of the procedure:* pure

If the argument is 0.0 return -inf. If the argument is less than 0.0 return NaN.

## **r-log10**

*Syntax:*

(**r-log10 r**)

*Arguments:*

Name: **r**  
Type: **<real>**  
Description: A real number

*Result value:* The base 10 logarithm of **r**  
*Result type:* **<real>**

*Purity of the procedure:* pure

If the argument is 0.0 return -inf. If the argument is less than 0.0 return NaN.

## **r-sin**

*Syntax:*

(**r-sin r**)

*Arguments:*

Name: **r**  
Type: **<real>**  
Description: A real number

*Result value:* The sine of the argument

*Result type:* <real>

*Purity of the procedure:* pure

## **r-cos**

*Syntax:*

(r-cos r)

*Arguments:*

Name: r

Type: <real>

Description: A real number

*Result value:* The cosine of the argument

*Result type:* <real>

*Purity of the procedure:* pure

## **r-tan**

*Syntax:*

(r-tan r)

*Arguments:*

Name: r

Type: <real>

Description: A real number

*Result value:* The tangent of the argument

*Result type:* <real>

*Purity of the procedure:* pure

## **r-asin**

*Syntax:*

```
(r-asin r)
```

*Arguments:*

Name: r  
Type: <real>  
Description: A real number

*Result value:* The arcsine of the argument

*Result type:* <real>

*Purity of the procedure:* pure

If the result is not real return NaN.

## **r-acos**

*Syntax:*

```
(r-acos r)
```

*Arguments:*

Name: r  
Type: <real>  
Description: A real number

*Result value:* The arccosine of the argument

*Result type:* <real>

*Purity of the procedure:* pure

If the result is not real return NaN.

## **r-atan**

*Syntax:*

```
(r-atan r)
```

*Arguments:*

Name: r

Type: `<real>`  
Description: A real number

*Result value:* The arctangent of the argument  
*Result type:* `<real>`

*Purity of the procedure:* pure

## **r-sinh**

*Syntax:*

`(r-sinh r)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real number

*Result value:* The hyperbolic sine of the argument  
*Result type:* `<real>`

*Purity of the procedure:* pure

## **r-cosh**

*Syntax:*

`(r-cosh r)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real number

*Result value:* The hyperbolic cosine of the argument  
*Result type:* `<real>`

*Purity of the procedure:* pure

**r-tanh***Syntax:* $(\text{r-tanh } r)$ *Arguments:*

Name: **r**  
Type: **<real>**  
Description: A real number

*Result value:* The hyperbolic tangent of the argument*Result type:* **<real>***Purity of the procedure:* pure**r-asinh***Syntax:* $(\text{r-asinh } r)$ *Arguments:*

Name: **r**  
Type: **<real>**  
Description: A real number

*Result value:* The hyperbolic arcsine of the argument*Result type:* **<real>***Purity of the procedure:* pure**r-acosh***Syntax:* $(\text{r-acosh } r)$ *Arguments:*

Name: **r**

Type: `<real>`  
Description: A real number

*Result value:* The hyperbolic arccosine of the argument  
*Result type:* `<real>`

*Purity of the procedure:* pure

If the result is not real return NaN.

## `r-atanh`

*Syntax:*

`(r-atanh r)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real number

*Result value:* The hyperbolic arctangent of the argument  
*Result type:* `<real>`

*Purity of the procedure:* pure

If the result is not real return NaN.

## `r-atan2`

*Syntax:*

`(r-atan2 y x)`

*Arguments:*

Name: `y`  
Type: `<real>`  
Description: A real number

Name: `x`  
Type: `<real>`  
Description: A real number



*Result value:* The angle between point (x, y) and the positive x axis (in radians)

*Result type:* <real>

*Purity of the procedure:* pure

## i-sqrt

*Syntax:*

```
(i-sqrt i)
```

*Arguments:*

Name: i

Type: <integer>

Description: An integer number

*Result value:* Square root of the argument

*Result type:* (:union <real> <integer>)

*Purity of the procedure:* pure

If the square root is integer valued it is converted to class <integer>.

## rat-sqrt

*Syntax:*

```
(rat-sqrt rat)
```

*Arguments:*

Name: rat

Type: <rational>

Description: A rational number

*Result value:* Square root of the argument

*Result type:* (:union <real> <rational> <integer>)

*Purity of the procedure:* pure

If the square root is rational or integer valued it is converted to class <rational> or <integer>, respectively.

## 16.4 Methods

```

=: (<real> <rational>) → <boolean> pure    =   real-rational=
=: (<rational> <real>) → <boolean> pure    =   rational-real=

<: (<real-number> <real-number>) → <boolean> pure abstract
<: (<real> <rational>) → <boolean> pure    =   real-rational<
<: (<rational> <real>) → <boolean> pure    =   rational-real<
<=: (<real-number> <real-number>) → <boolean> pure abstract
<=: (<real> <rational>) → <boolean> pure    =   real-rational<=
<=: (<rational> <real>) → <boolean> pure    =   rational-real<=
>: (<real-number> <real-number>) → <boolean> pure abstract
>: (<real> <rational>) → <boolean> pure    =   real-rational>
>: (<rational> <real>) → <boolean> pure    =   rational-real>
>=: (<real-number> <real-number>) → <boolean> pure abstract
>=: (<real> <rational>) → <boolean> pure    =   real-rational>=
>=: (<rational> <real>) → <boolean> pure    =   rational-real>=

+: (<real-number> <real-number>) → <real-number> pure abstract
+: (<real> <rational>) → <real> pure      =   real-rational+
+: (<rational> <real>) → <real> pure      =   rational-real+
-: (<real-number> <real-number>) → <real-number> pure abstract
-: (<real> <rational>) → <real> pure      =   real-rational-
-: (<rational> <real>) → <real> pure      =   rational-real-
*: (<real-number> <real-number>) → <real-number> pure abstract
*: (<real> <rational>) → <real> pure      =   real-rational*
*: (<rational> <real>) → <real> pure      =   rational-real*
/: (<real-number> <real-number>) → <real-number> pure abstract
/: (<real> <rational>) → <real> pure      =   real-rational/
/: (<rational> <real>) → <real> pure      =   rational-real/

-: (<real-number>) → <real-number> pure abstract
abs: (<real-number>) → <real-number> pure abstract
square: (<real-number>) → <real-number> pure abstract
sign: (<real-number>) → <integer> pure abstract

```

## Chapter 17

# Module (standard-library complex)

### 17.1 Data Types

*Data type name:* <complex>

*Type:* <class>

*Description:* A complex number

Class <complex> is immutable, equal by value, and not inheritable.

### 17.2 Simple Procedures

real->complex

*Syntax:*

(real->complex r)

*Arguments:*

Name: r

Type: <real>

Description: A real number

*Result value:* The complex number corresponding to the given real number

*Result type:* <complex>

*Purity of the procedure:* pure

**integer->complex**

*Syntax:*

(integer->complex n)

*Arguments:*

Name: n  
Type: <integer>  
Description: An integer number

*Result value:* The complex number corresponding to the given integer number

*Result type:* <complex>

*Purity of the procedure:* pure

**rational->complex**

*Syntax:*

(rational->complex rat)

*Arguments:*

Name: rat  
Type: <rational>  
Description: An integer number

*Result value:* The complex number corresponding to the given rational number

*Result type:* <complex>

*Purity of the procedure:* pure

**complex=?**

*Syntax:*

(complex=? cx1 cx2)

*Arguments:*

Name: cx1

Type: `<complex>`  
Description: A complex value to be compared

Name: `cx2`  
Type: `<complex>`  
Description: A complex value to be compared

*Result value:* `#t` iff `cx1` is equal to `cx2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `complex=`

*Syntax:*

```
(complex= cx1 cx2)
```

*Arguments:*

Name: `cx1`  
Type: `<complex>`  
Description: A complex value to be compared

Name: `cx2`  
Type: `<complex>`  
Description: A complex value to be compared

*Result value:* `#t` iff `cx1` is numerically equal to `cx2`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `complex-integer=`

*Syntax:*

```
(complex-integer= cx i)
```

*Arguments:*

Name: `cx`  
Type: `<complex>`

Description: A complex value to be compared

Name: `i`

Type: `<integer>`

Description: An integer value to be compared

*Result value:* `#t` iff `cx` is numerically equal to `i`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

### `integer-complex=`

*Syntax:*

```
(integer-complex= i cx)
```

*Arguments:*

Name: `i`

Type: `<integer>`

Description: An integer value to be compared

Name: `cx`

Type: `<complex>`

Description: A complex value to be compared

*Result value:* `#t` iff `i` is numerically equal to `cx`

*Result type:* `<boolean>`

*Purity of the procedure:* pure

### `complex-real=`

*Syntax:*

```
(complex-real= cx r)
```

*Arguments:*

Name: `cx`

Type: `<complex>`

Description: A complex value to be compared

Name: `r`  
Type: `<real>`  
Description: A real value to be compared

*Result value:* `#t` iff `cx` is numerically equal to `r`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

### `real-complex=`

*Syntax:*

```
(real-complex= r cx)
```

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value to be compared

Name: `cx`  
Type: `<complex>`  
Description: A complex value to be compared

*Result value:* `#t` iff `r` is numerically equal to `cx`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

### `complex-rational=`

*Syntax:*

```
(complex-rational= cx rat)
```

*Arguments:*

Name: `cx`  
Type: `<complex>`  
Description: A complex value to be compared

Name: `rat`  
Type: `<rational>`  
Description: A rational value to be compared

*Result value:* `#t` iff `cx` is numerically equal to `rat`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `rational-complex=`

*Syntax:*

```
(rational-complex= rat cx)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: An rational value to be compared

Name: `cx`  
Type: `<complex>`  
Description: A complex value to be compared

*Result value:* `#t` iff `rat` is numerically equal to `cx`  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `complex+`

*Syntax:*

```
(complex+ cx1 cx2)
```

*Arguments:*

Name: `cx1`  
Type: `<complex>`  
Description: A complex value

Name: `cx2`



Type: <complex>  
Description: A complex value

*Result value:* The sum of the arguments

*Result type:* <complex>

*Purity of the procedure:* pure

## complex-integer+

*Syntax:*

(complex-integer+ cx i)

*Arguments:*

Name: cx  
Type: <complex>  
Description: A complex value

Name: i  
Type: <integer>  
Description: An integer value

*Result value:* The sum of the arguments

*Result type:* <complex>

*Purity of the procedure:* pure

## integer-complex+

*Syntax:*

(integer-complex+ i cx)

*Arguments:*

Name: i  
Type: <integer>  
Description: An integer value

Name: cx  
Type: <complex>

Description: A complex value

*Result value:* The sum of the arguments

*Result type:* <complex>

*Purity of the procedure:* pure

## complex-real+

*Syntax:*

```
(complex-real+ cx r)
```

*Arguments:*

Name: `cx`

Type: <complex>

Description: A complex value

Name: `r`

Type: <real>

Description: A real value

*Result value:* The sum of the arguments

*Result type:* <complex>

*Purity of the procedure:* pure

## real-complex+

*Syntax:*

```
(real-complex+ r cx)
```

*Arguments:*

Name: `r`

Type: <real>

Description: A real value

Name: `cx`

Type: <complex>

Description: A complex value

*Result value:* The sum of the arguments

*Result type:* <complex>

*Purity of the procedure:* pure

### complex-rational+

*Syntax:*

```
(complex-rational+ cx rat)
```

*Arguments:*

Name: cx

Type: <complex>

Description: A complex value

Name: rat

Type: <rational>

Description: A rational value

*Result value:* The sum of the arguments

*Result type:* <complex>

*Purity of the procedure:* pure

### rational-complex+

*Syntax:*

```
(rational-complex+ rat cx)
```

*Arguments:*

Name: rat

Type: <rational>

Description: An rational value

Name: cx

Type: <complex>

Description: A complex value

*Result value:* The sum of the arguments

*Result type:* <complex>

*Purity of the procedure:* pure

## complex-

*Syntax:*

```
(complex- cx1 cx2)
```

*Arguments:*

Name: cx1

Type: <complex>

Description: A complex value

Name: cx2

Type: <complex>

Description: A complex value

*Result value:* The difference of the arguments

*Result type:* <complex>

*Purity of the procedure:* pure

## complex-integer-

*Syntax:*

```
(complex-integer- cx i)
```

*Arguments:*

Name: cx

Type: <complex>

Description: A complex value

Name: i

Type: <integer>

Description: An integer value

*Result value:* The difference of the arguments

*Result type:* <complex>

*Purity of the procedure:* pure

## integer-complex-

*Syntax:*

```
(integer-complex- i cx)
```

*Arguments:*

Name: *i*  
Type: <integer>  
Description: An integer value

Name: *cx*  
Type: <complex>  
Description: A complex value

*Result value:* The difference of the arguments

*Result type:* <complex>

*Purity of the procedure:* pure

## complex-real-

*Syntax:*

```
(complex-real- cx r)
```

*Arguments:*

Name: *cx*  
Type: <complex>  
Description: A complex value

Name: *r*  
Type: <real>  
Description: A real value

*Result value:* The difference of the arguments

*Result type:* <complex>

*Purity of the procedure:* pure

## real-complex-

*Syntax:*

```
(real-complex- r cx)
```

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value

Name: `cx`  
Type: `<complex>`  
Description: A complex value

*Result value:* The difference of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

## complex-rational-

*Syntax:*

```
(complex-rational- cx rat)
```

*Arguments:*

Name: `cx`  
Type: `<complex>`  
Description: A complex value

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The difference of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

## rational-complex-

*Syntax:*

```
(rational-complex- rat cx)
```

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: An rational value

Name: `cx`  
Type: `<complex>`  
Description: A complex value

*Result value:* The difference of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

## complex\*

*Syntax:*

```
(complex* cx1 cx2)
```

*Arguments:*

Name: `cx1`  
Type: `<complex>`  
Description: A complex value

Name: `cx2`  
Type: `<complex>`  
Description: A complex value

*Result value:* The product of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

**complex-integer\***

*Syntax:*

```
(complex-integer* cx i)
```

*Arguments:*

Name: `cx`  
Type: `<complex>`  
Description: A complex value

Name: `i`  
Type: `<integer>`  
Description: An integer value

*Result value:* The product of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

**integer-complex\***

*Syntax:*

```
(integer-complex* i cx)
```

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value

Name: `cx`  
Type: `<complex>`  
Description: A complex value

*Result value:* The product of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

**complex-real\***



*Syntax:*

```
(complex-real* cx r)
```

*Arguments:*

Name: `cx`  
Type: `<complex>`  
Description: A complex value

Name: `r`  
Type: `<real>`  
Description: A real value

*Result value:* The product of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

## `real-complex*`

*Syntax:*

```
(real-complex* r cx)
```

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value

Name: `cx`  
Type: `<complex>`  
Description: A complex value

*Result value:* The product of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

## `complex-rational*`

*Syntax:*

(complex-rational\* cx rat)

*Arguments:*

Name: `cx`  
Type: `<complex>`  
Description: A complex value

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The product of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

**rational-complex\***

*Syntax:*

(rational-complex\* rat cx)

*Arguments:*

Name: `rat`  
Type: `<rational>`  
Description: An rational value

Name: `cx`  
Type: `<complex>`  
Description: A complex value

*Result value:* The product of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

**complex/**

*Syntax:*

(complex/ cx1 cx2)

*Arguments:*

Name: `cx1`  
Type: `<complex>`  
Description: A complex value

Name: `cx2`  
Type: `<complex>`  
Description: A complex value

*Result value:* The quotient of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

## `complex-integer/`

*Syntax:*

`(complex-integer/ cx i)`

*Arguments:*

Name: `cx`  
Type: `<complex>`  
Description: A complex value

Name: `i`  
Type: `<integer>`  
Description: An integer value

*Result value:* The quotient of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

## `integer-complex/`

*Syntax:*

`(integer-complex/ i cx)`

*Arguments:*

Name: `i`  
Type: `<integer>`  
Description: An integer value

Name: `cx`  
Type: `<complex>`  
Description: A complex value

*Result value:* The quotient of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

## `complex-real/`

*Syntax:*

`(complex-real/ cx r)`

*Arguments:*

Name: `cx`  
Type: `<complex>`  
Description: A complex value

Name: `r`  
Type: `<real>`  
Description: A real value

*Result value:* The quotient of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

## `real-complex/`

*Syntax:*

`(real-complex/ r cx)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real value

Name: `cx`  
Type: `<complex>`  
Description: A complex value

*Result value:* The quotient of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

### `complex-rational/`

*Syntax:*

```
(complex-rational/ cx rat)
```

*Arguments:*

Name: `cx`  
Type: `<complex>`  
Description: A complex value

Name: `rat`  
Type: `<rational>`  
Description: A rational value

*Result value:* The quotient of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

### `rational-complex/`

*Syntax:*

```
(rational-complex/ rat cx)
```

*Arguments:*

Name: `rat`

Type: `<rational>`  
Description: An rational value

Name: `cx`  
Type: `<complex>`  
Description: A complex value

*Result value:* The quotient of the arguments

*Result type:* `<complex>`

*Purity of the procedure:* pure

## **c-neg**

*Syntax:*

`(c-neg c)`

*Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* The opposite number of the given complex number

*Result type:* `<complex>`

*Purity of the procedure:* pure

## **c-abs**

*Syntax:*

`(c-abs c)`

*Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* The absolute value of the given complex number

*Result type:* `<real>`

*Purity of the procedure:* pure

## c-square

*Syntax:*

```
(c-square c)
```

*Arguments:*

Name: c  
Type: <complex>  
Description: A complex number

*Result value:* The square of the given complex number

*Result type:* <complex>

*Purity of the procedure:* pure

## real-part

*Syntax:*

```
(real-part c)
```

*Arguments:*

Name: c  
Type: <complex>  
Description: A complex number

*Result value:* The real part of the given complex number

*Result type:* <real>

*Purity of the procedure:* pure

## imag-part

*Syntax:*

(imag-part c)

*Arguments:*

Name: c  
Type: <complex>  
Description: A complex number

*Result value:* The imaginary part of the given complex number

*Result type:* <real>

*Purity of the procedure:* pure

## make-polar

*Syntax:*

(make-polar magnitude angle)

*Arguments:*

Name: magnitude  
Type: <real>

Name: angle  
Type: <real>

*Result value:* The complex number having the given magnitude and angle

*Result type:* <complex>

*Purity of the procedure:* pure

## c-angle

*Syntax:*

(c-angle c)

*Arguments:*

Name: c  
Type: <complex>  
Description: A complex number



*Result value:* The angle of the given complex number

*Result type:* <real>

*Purity of the procedure:* pure

## r-complex-log

*Syntax:*

```
(r-complex-log r)
```

*Arguments:*

Name: **r**

Type: <real>

Description: A real number

*Result value:* The natural logarithm of **r**

*Result type:* <complex>

*Purity of the procedure:* pure

The result is computed correctly for the negative values of **r**, too.

## r-log-neg

*Syntax:*

```
(r-log-neg r)
```

*Arguments:*

Name: **r**

Type: <real>

Description: A negative real number

*Result value:* The natural logarithm of **r**

*Result type:* <complex>

*Purity of the procedure:* pure

The natural logarithm of a negative real number  $r$  is  $\ln r = \ln |r| + i\pi$ .

**r-log10-neg***Syntax:*`(r-log10-neg r)`*Arguments:*

Name: `r`  
 Type: `<real>`  
 Description: A negative real number

*Result value:* The base 10 logarithm of `r`*Result type:* `<complex>`*Purity of the procedure:* pure

The base 10 logarithm of a negative real number  $r$  is  $\log_{10} r = \log_{10} |r| + i\pi / \ln 10$ .

**r-complex-expt***Syntax:*`(r-complex-expt r-base r-exponent)`*Arguments:*

Name: `r-base`  
 Type: `<real>`  
 Description: A real number

Name: `r-exponent`  
 Type: `<real>`  
 Description: A real number

*Result value:* `r-base` raised to the power of `r-exponent`*Result type:* `<complex>`*Purity of the procedure:* pure

The result is computed correctly for the negative values of `r-base`, too.

**complex-real-expt**

*Syntax:*

```
(complex-real-expt cx-base r-exponent)
```

*Arguments:*

Name: `cx-base`  
Type: `<complex>`  
Description: A complex number

Name: `r-exponent`  
Type: `<real>`  
Description: A real number

*Result value:* `cx-base` raised to the power of `r-exponent`

*Result type:* `<complex>`

*Purity of the procedure:* pure

## real-complex-expt

*Syntax:*

```
(real-complex-expt r-base cx-exponent)
```

*Arguments:*

Name: `r-base`  
Type: `<real>`  
Description: A real number

Name: `cx-exponent`  
Type: `<complex>`  
Description: A complex number

*Result value:* `r-base` raised to the power of `cx-exponent`

*Result type:* `<complex>`

*Purity of the procedure:* pure

This procedure works for negative values of `r-base`, too.

## c-exp2

*Syntax:*

(c-exp2 cx)

*Arguments:*

Name: cx  
Type: <complex>  
Description: A complex number

*Result value:* 2 raised to the power of cx-exponent

*Result type:* <complex>

*Purity of the procedure:* pure

## c-nonneg-int-expt

*Syntax:*

(c-nonneg-int-expt cx-base i-expt)

*Arguments:*

Name: cx-base  
Type: <complex>  
Description: A complex number

Name: i-expt  
Type: <integer>  
Description: A nonnegative integer number

*Result value:* cx-base raised to the power i-expt

*Result type:* <complex>

*Purity of the procedure:* pure

## c-int-expt

*Syntax:*

(c-int-expt cx-base i-expt)

*Arguments:*

Name: `cx-base`  
Type: `<complex>`  
Description: A complex number

Name: `i-expt`  
Type: `<integer>`  
Description: An integer number

*Result value:* `cx-base` raised to the power `i-expt`

*Result type:* `<complex>`

*Purity of the procedure:* pure

## `c-sqrt`

*Syntax:*

`(c-sqrt c)`

*Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* Square root of the argument

*Result type:* `<complex>`

*Purity of the procedure:* pure

## `c-expt`

*Syntax:*

`(c-expt x y)`

*Arguments:*

Name: `x`  
Type: `<complex>`  
Description: A complex number

Name: `y`  
Type: `<complex>`  
Description: A complex number

*Result value:* `x` to the power of `y`  
*Result type:* `<complex>`

*Purity of the procedure:* pure

## `c-exp`

*Syntax:*

`(c-exp c)`

*Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:*  $e$  to the power of `r`  
*Result type:* `<complex>`

*Purity of the procedure:* pure

Number  $e$  is the base of natural logarithms (approx. 2.718).

## `c-log`

*Syntax:*

`(c-log c)`

*Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* The natural logarithm of `r`  
*Result type:* `<complex>`

*Purity of the procedure:* pure

**c-log10***Syntax:* $(\text{c-log10 } c)$ *Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* The base 10 logarithm of `r`*Result type:* `<complex>`*Purity of the procedure:* pure**c-sin***Syntax:* $(\text{c-sin } c)$ *Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* The sine of the argument*Result type:* `<complex>`*Purity of the procedure:* pure**c-cos***Syntax:* $(\text{c-cos } c)$ *Arguments:*

Name: `c`

Type: `<complex>`  
Description: A complex number

*Result value:* The cosine of the argument  
*Result type:* `<complex>`

*Purity of the procedure:* pure

## **c-tan**

*Syntax:*

`(c-tan c)`

*Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* The tangent of the argument  
*Result type:* `<complex>`

*Purity of the procedure:* pure

## **c-asin**

*Syntax:*

`(c-asin c)`

*Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* The arcsine of the argument  
*Result type:* `<complex>`

*Purity of the procedure:* pure

This procedure sometimes returns a value from a different branch compared



to guile (2.2.2) `asin`. Our convention is similar to Octave (version 3.8.1).

### **c-acos**

*Syntax:*

```
(c-acos c)
```

*Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* The arccosine of the argument

*Result type:* `<complex>`

This procedure sometimes returns a value from a different branch compared to guile (2.2.2) `acos`. Our convention is similar to Octave (version 3.8.1).

*Purity of the procedure:* pure

### **c-atan**

*Syntax:*

```
(c-atan c)
```

*Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* The arctangent of the argument

*Result type:* `<complex>`

*Purity of the procedure:* pure

### **c-sinh**

*Syntax:*

`(c-sinh c)`

*Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* The hyperbolic sine of the argument

*Result type:* `<complex>`

*Purity of the procedure:* pure

**c-cosh**

*Syntax:*

`(c-cosh c)`

*Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* The hyperbolic cosine of the argument

*Result type:* `<complex>`

*Purity of the procedure:* pure

**c-tanh**

*Syntax:*

`(c-tanh c)`

*Arguments:*

Name: `c`  
Type: `<complex>`  
Description: A complex number

*Result value:* The hyperbolic tangent of the argument

*Result type:* <complex>

*Purity of the procedure:* pure

## c-asinh

*Syntax:*

(c-asinh c)

*Arguments:*

Name: c

Type: <complex>

Description: A complex number

*Result value:* The hyperbolic arcsine of the argument

*Result type:* <complex>

*Purity of the procedure:* pure

## c-acosh

*Syntax:*

(c-acosh c)

*Arguments:*

Name: c

Type: <complex>

Description: A complex number

*Result value:* The hyperbolic arccosine of the argument

*Result type:* <complex>

*Purity of the procedure:* pure

## c-atanh

*Syntax:*

`(c-atanh c)`

*Arguments:*

Name: `c`  
 Type: `<complex>`  
 Description: A complex number

*Result value:* The hyperbolic arctangent of the argument

*Result type:* `<complex>`

*Purity of the procedure:* pure

This procedure sometimes returns a value from a different branch compared to guile (2.2.2) `atanh`. Our convention is similar to Octave (version 3.8.1).

## complex-to-string

*Syntax:*

`(complex-to-string c)`

*Arguments:*

Name: `c`  
 Type: `<complex>`  
 Description: A complex number

*Result value:* The complex number as a string

*Result type:* `<string>`

*Purity of the procedure:* pure

## 17.3 Methods

`complex: (<real-number> <real-number>) → <complex>` pure abstract

`complex: (<real> <real>) → <complex>` pure

`complex: (<real> <integer>) → <complex>` pure

`complex: (<real> <rational>) → <complex>` pure

`complex: (<integer> <real>) → <complex>` pure

`complex: (<integer> <integer>) → <complex>` pure

`complex: (<integer> <rational>) → <complex>` pure

`complex: (<rational> <real>) → <complex>` pure

`complex: (<rational> <integer>) → <complex>` pure

`complex: (<rational> <rational>) → <complex>` pure

These methods construct a complex number from the real and imaginary parts given as arguments. The arguments are converted to `<real>` if necessary.

```

equal?: (<complex> <complex>) → <boolean> pure    =    complex=?

=: (<complex> <complex>) → <boolean> pure    =    complex=
=: (<complex> <integer>) → <boolean> pure    =    complex-integer=
=: (<integer> <complex>) → <boolean> pure    =    integer-complex=
=: (<complex> <real>) → <boolean> pure    =    complex-real=
=: (<real> <complex>) → <boolean> pure    =    real-complex=
=: (<complex> <rational>) → <boolean> pure    =    complex-rational=
=: (<rational> <complex>) → <boolean> pure    =    rational-complex=

+: (<complex> <complex>) → <complex> pure    =    complex+
+: (<complex> <integer>) → <complex> pure    =    complex-integer+
+: (<integer> <complex>) → <complex> pure    =    integer-complex+
+: (<complex> <real>) → <complex> pure    =    complex-real+
+: (<real> <complex>) → <complex> pure    =    real-complex+
+: (<complex> <rational>) → <complex> pure    =    complex-rational+
+: (<rational> <complex>) → <complex> pure    =    rational-complex+

-: (<complex> <complex>) → <complex> pure    =    complex-
-: (<complex> <integer>) → <complex> pure    =    complex-integer-
-: (<integer> <complex>) → <complex> pure    =    integer-complex-
-: (<complex> <real>) → <complex> pure    =    complex-real-
-: (<real> <complex>) → <complex> pure    =    real-complex-
-: (<complex> <rational>) → <complex> pure    =    complex-rational-
-: (<rational> <complex>) → <complex> pure    =    rational-complex-

*: (<complex> <complex>) → <complex> pure    =    complex*
*: (<complex> <integer>) → <complex> pure    =    complex-integer*
*: (<integer> <complex>) → <complex> pure    =    integer-complex*
*: (<complex> <real>) → <complex> pure    =    complex-real*
*: (<real> <complex>) → <complex> pure    =    real-complex*
*: (<complex> <rational>) → <complex> pure    =    complex-rational*
*: (<rational> <complex>) → <complex> pure    =    rational-complex*

/: (<complex> <complex>) → <complex> pure    =    complex/
/: (<complex> <integer>) → <complex> pure    =    complex-integer/
/: (<integer> <complex>) → <complex> pure    =    integer-complex/
/: (<complex> <real>) → <complex> pure    =    complex-real/
/: (<real> <complex>) → <complex> pure    =    real-complex/
/: (<complex> <rational>) → <complex> pure    =    complex-rational/
/: (<rational> <complex>) → <complex> pure    =    rational-complex/

-: (<complex>) → <complex> pure    =    c-neg
square: (<complex>) → <complex> pure    =    c-square
abs: (<complex>) → <real> pure    =    c-abs

atom-to-string: (<complex> <boolean>) → <string> pure    =    complex-to-string

```



## Chapter 18

# Module (standard-library math)

This module reexports modules `rational`, `real-math`, and `complex`.

### 18.1 Data Types

*Data type name:* `<number>`

*Type:* `:union`

*Description:* A number

Type `<number>` is equal to the union of `<complex>`, `<real>`, `<rational>`, and `<integer>`.

### 18.2 Simple Procedures

#### `integer-valued?`

*Syntax:*

```
(integer-valued? x)
```

*Arguments:*

Name: `x`

Type: `<object>`

Description: An object

*Result value:* `#t` iff the argument is integer valued

*Result type:* `<boolean>`

*Purity of the procedure:* pure

An object is integer valued if one of the following holds:

- It is an `<integer>`
- It is an integer valued `<rational>`
- It is an integer valued `<real>`
- It is a `<complex>` whose imaginary part is 0.0 and real part is integer valued

## real-valued?

*Syntax:*

```
(real-valued? x)
```

*Arguments:*

Name: `x`  
Type: `<object>`  
Description: An object

*Result value:* `#t` iff the argument is real valued

*Result type:* `<boolean>`

*Purity of the procedure:* pure

An object is real valued if

- It is an `<integer>`,
- It is a `<rational>`,
- It is a `<real>`, or
- It is a `<complex>` whose imaginary part is 0.0

## rational-valued?

*Syntax:*

```
(rational-valued? x)
```



*Arguments:*

Name: `x`  
 Type: `<object>`  
 Description: An object

*Result value:* `#t` iff the argument is rational valued

*Result type:* `<boolean>`

*Purity of the procedure:* pure

An object is rational valued if

- It is an `<integer>`,
- It is a `<rational>`,
- It is an integer valued `<real>`, or
- It is a `<complex>` whose imaginary part is 0.0 and real part is integer valued

## exact?

*Syntax:*

`(exact? nr)`

*Arguments:*

Name: `nr`  
 Type: `<number>`  
 Description: A number

*Result value:* `#t` iff the argument is exact

*Result type:* `<boolean>`

*Purity of the procedure:* pure

A number is exact iff it is an `<integer>` or a `<rational>`.

## inexact?

*Syntax:*

`(inexact? nr)`

*Arguments:*

Name: `nr`  
Type: `<number>`  
Description: A number

*Result value:* `#t` iff the argument is inexact  
*Result type:* `<boolean>`

*Purity of the procedure:* pure

A number is exact iff it is not inexact, i.e, it is an `<real>` or a `<complex>`.

## `real->exact`

*Syntax:*

`(real->exact r)`

*Arguments:*

Name: `r`  
Type: `<real>`  
Description: A real number

*Result value:* The argument converted to a rational or an integer  
*Result type:* `<rational-number>`

*Purity of the procedure:* pure

## `complex->exact`

*Syntax:*

`(complex->exact cx)`

*Arguments:*

Name: `cx`  
Type: `<complex>`  
Description: A complex number

*Result value:* The argument converted to a rational or an integer  
*Result type:* `<rational-number>`

*Purity of the procedure:* pure

## real->exact

*Syntax:*

```
(real->exact r)
```

*Arguments:*

Name: r  
 Type: <real>  
 Description: A real number

*Result value:* The argument converted to a rational or an integer

*Result type:* <rational-number>

*Purity of the procedure:* pure

If the imaginary part of the argument is not 0.0 exception `complex->exact:invalid-argument` is raised.

## 18.3 Methods

```
exact->inexact: (<number>) → <number> pure abstract
exact->inexact: (<real-number>) → <real-number> pure abstract
exact->inexact: (<integer>) → <real> pure = integer->real
exact->inexact: (<rational>) → <real> pure = rational->real
```

```
exact->inexact: (<real>) → <real> pure
```

Returns the argument.

```
exact->inexact: (<complex>) → <complex> pure
```

Returns the argument.

```
inexact->exact: (<number>) → <rational-number> pure abstract
inexact->exact: (<complex>) → <rational-number> pure = complex->exact
inexact->exact: (<real>) → <rational-number> pure = real->exact
```

```
inexact->exact: (<rational>) → <rational> pure
```

Returns the argument.

`inexact->exact: (<integer>) → <integer> pure`

Returns the argument.

`+: (<number> <number>) → <number> pure abstract`  
`-: (<number> <number>) → <number> pure abstract`  
`*: (<number> <number>) → <number> pure abstract`  
`/: (<number> <number>) → <number> pure abstract`

`-: (<number>) → <number> pure abstract`  
`abs: (<number>) → <real-number> pure abstract`  
`square: (<number>) → <number> pure abstract`

`sqrt: (<number>) → <number> pure abstract`  
`sqrt: (<integer>) → <number> pure`  
`sqrt: (<rational>) → <number> pure`  
`sqrt: (<real>) → (:union <real> <complex>) pure`  
`sqrt: (<complex>) → <complex> pure = c-sqrt`

These methods compute the square root of the argument. They return exact 0 when the argument is exact 0.

`expt: (<number> <number>) → <number> pure abstract`  
`expt: (<integer> <integer>) → <rational-number> pure`  
`expt: (<integer> <real>) → (:union <real> <complex>) pure`  
`expt: (<integer> <rational>) → <number> pure`  
`expt: (<integer> <complex>) → <complex> pure`  
`expt: (<real> <integer>) → (:union <real> <integer>) pure`  
`expt: (<real> <real>) → (:union <real> <complex>) pure`  
`expt: (<real> <rational>) → (:union <integer> <real> <complex>) pure`  
`expt: (<real> <complex>) → <complex> pure`  
`expt: (<rational> <integer>) → <rational> pure = rat-int-expt`  
`expt: (<rational> <real>) → (:union <integer> <real> <complex>) pure`  
`expt: (<rational> <rational>) → <number> pure`  
`expt: (<rational> <complex>) → <complex> pure`  
`expt: (<complex> <integer>) → (:union <complex> <integer>) pure`  
`expt: (<complex> <real>) → <complex> pure`  
`expt: (<complex> <rational>) → <complex> pure`  
`expt: (<complex> <complex>) → <complex> pure = c-expt`

These methods raise the first argument to the power of the second argument. If any of the arguments is equal to exact 0 the value 0 or 1 is returned in many cases.

`exp: (<number>) → <number> pure abstract`  
`exp: (<real-number>) → (:union <real> <integer>) pure abstract`  
`exp: (<integer>) → (:union <real> <integer>) pure`  
`exp: (<rational>) → (:union <real> <integer>) pure`  
`exp: (<real>) → <real> pure = r-exp`  
`exp: (<complex>) → <complex> pure = c-exp`

These methods compute the exponential of the argument. They return exact 1 when the argument is exact 0.

```
log: (<number>) → <number> pure abstract
log: (<integer>) → (:union <integer> <real> <complex>) pure
log: (<rational>) → (:union <integer> <real> <complex>) pure
log: (<real>) → (:union <real> <complex>) pure
log: (<complex>) → <complex> pure = c-log
```

These methods compute the natural logarithm of the argument. They return exact 0 when the argument is exact 1.

```
log10: (<number>) → <number> pure abstract
log10: (<integer>) → (:union <integer> <real> <complex>) pure
log10: (<rational>) → (:union <integer> <real> <complex>) pure
log10: (<real>) → (:union <real> <complex>) pure
log10: (<complex>) → <complex> pure = c-log10
```

These methods compute the logarithm with base 10 of the argument. They return exact 0 when the argument is exact 1.

```
sin: (<number>) → <number> pure abstract
sin: (<real-number>) → (:union <real> <integer>) pure abstract
sin: (<integer>) → (:union <real> <integer>) pure
sin: (<rational>) → (:union <real> <integer>) pure
sin: (<real>) → <real> pure = r-sin
sin: (<complex>) → <complex> pure = c-sin
```

These methods compute the sine of the argument. They return exact 0 when the argument is exact 0.

```
cos: (<number>) → <number> pure abstract
cos: (<real-number>) → (:union <real> <integer>) pure abstract
cos: (<integer>) → (:union <real> <integer>) pure
cos: (<rational>) → (:union <real> <integer>) pure
cos: (<real>) → <real> pure = r-cos
cos: (<complex>) → <complex> pure = c-cos
```

These methods compute the cosine of the argument. They return exact 1 when the argument is exact 0.

```
tan: (<number>) → <number> pure abstract
tan: (<real-number>) → (:union <real> <integer>) pure abstract
tan: (<integer>) → (:union <real> <integer>) pure
tan: (<rational>) → (:union <real> <integer>) pure
tan: (<real>) → <real> pure = r-tan
tan: (<complex>) → <complex> pure = c-tan
```

These methods compute the tangent of the argument. They return exact 0 when the argument is exact 0.

```

asin: (<number>) → <number>   pure abstract
asin: (<integer>) → (:union <integer> <real> <complex>) pure
asin: (<rational>) → (:union <integer> <real> <complex>) pure
asin: (<real>) → (:union <real> <complex>) pure
asin: (<complex>) → <complex> pure   =   c-asin

```

These methods compute the arcsine of the argument. They return exact 0 when the argument is exact 0. See the note for `c-asin`, which may also apply when the argument is real.

```

acos: (<number>) → <number>   pure abstract
acos: (<integer>) → (:union <integer> <real> <complex>) pure
acos: (<rational>) → (:union <integer> <real> <complex>) pure
acos: (<real>) → (:union <real> <complex>) pure
acos: (<complex>) → <complex> pure   =   c-acos

```

These methods compute the arccosine of the argument. They return exact 0 when the argument is exact 1. See the note for `c-acos`, which may also apply when the argument is real.

```

atan: (<number>) → <number>   pure abstract
atan: (<real-number>) → <real-number>   pure abstract
atan: (<integer>) → (:union <real> <integer>) pure
atan: (<rational>) → (:union <real> <integer>) pure
atan: (<real>) → <real> pure   =   r-atan
atan: (<complex>) → <complex> pure   =   c-atan

```

These methods compute the arctangent of the argument. They return exact 0 when the argument is exact 0.

```

sinh: (<number>) → <number>   pure abstract
sinh: (<real-number>) → (:union <real> <integer>)   pure abstract
sinh: (<integer>) → (:union <real> <integer>) pure
sinh: (<rational>) → (:union <real> <integer>) pure
sinh: (<real>) → <real> pure   =   r-sinh
sinh: (<complex>) → <complex> pure   =   c-sinh

```

These methods compute the hyperbolic sine of the argument. They return exact 0 when the argument is exact 0.

```

cosh: (<number>) → <number>   pure abstract
cosh: (<real-number>) → (:union <real> <integer>)   pure abstract
cosh: (<integer>) → (:union <real> <integer>) pure
cosh: (<rational>) → (:union <real> <integer>) pure
cosh: (<real>) → <real> pure   =   r-cosh
cosh: (<complex>) → <complex> pure   =   c-cosh

```

These methods compute the hyperbolic cosine of the argument. They return exact 1 when the argument is exact 0.

```

tanh: (<number>) → <number>   pure abstract

```

```

tanh: (<real-number>) → (:union <real> <integer>) pure abstract
tanh: (<integer>) → (:union <real> <integer>) pure
tanh: (<rational>) → (:union <real> <integer>) pure
tanh: (<real>) → <real> pure = r-tanh
tanh: (<complex>) → <complex> pure = c-tanh

```

These methods compute the hyperbolic tangent of the argument. They return exact 0 when the argument is exact 0.

```

asinh: (<number>) → <number> pure abstract
asinh: (<real-number>) → (:union <real> <integer>) pure abstract
asinh: (<integer>) → (:union <real> <integer>) pure
asinh: (<rational>) → (:union <real> <integer>) pure
asinh: (<real>) → <real> pure = r-asinh
asinh: (<complex>) → <complex> pure = c-asinh

```

These methods compute the inverse hyperbolic sine of the argument. They return exact 0 when the argument is exact 0.

```

acosh: (<number>) → <number> pure abstract
acosh: (<integer>) → (:union <integer> <real> <complex>) pure
acosh: (<rational>) → (:union <integer> <real> <complex>) pure
acosh: (<real>) → (:union <real> <complex>) pure
acosh: (<complex>) → <complex> pure = c-acosh

```

These methods compute the inverse hyperbolic cosine of the argument. They return exact 0 when the argument is exact 1.

```

atanh: (<number>) → <number> pure abstract
atanh: (<integer>) → (:union <integer> <real> <complex>) pure
atanh: (<rational>) → (:union <integer> <real> <complex>) pure
atanh: (<real>) → (:union <real> <complex>) pure
atanh: (<complex>) → <complex> pure = c-atanh

```

These methods compute the inverse hyperbolic tangent of the argument. They return exact 0 when the argument is exact 0. See the note for `c-atanh`, which may also apply when the argument is real.





## Chapter 19

# Module (standard-library extra-math)

This module implements wrapper procedures to many of the mathematical functions in standard C. Additionally, some helper procedures and methods are defined. This module works only for the target platform Guile.

### 19.1 Wrapper Procedures for Standard C Functions

The following procedures are wrappers to the corresponding functions in the standard C (without the prefix “r-”):

```
fmod: (<real> <real>) → <real> pure
r-remainder: (<real> <real>) → <real> pure
r-fma: (<real> <real> <real>) → <real> pure
fmin: (<real> <real>) → <real> pure
fmax: (<real> <real>) → <real> pure
fdim: (<real> <real>) → <real> pure
r-exp2: (<real>) → <real> pure
r-expm1: (<real>) → <real> pure
r-log2: (<real>) → <real> pure
r-log1p: (<real>) → <real> pure
logb: (<real>) → <real> pure
ilogb: (<real>) → <integer> pure
r-cbrt: (<real>) → <real> pure
r-hypot: (<real> <real>) → <real> pure
r-erf: (<real>) → <real> pure
r-erfc: (<real>) → <real> pure
r-lgamma: (<real>) → <real> pure
r-tgamma: (<real>) → <real> pure
r-nearbyint: (<real>) → <real> pure
rint: (<real>) → <real> pure
frexp: (<real>) → (:pair <real> <integer>) pure
ldexp: (<real> <integer>) → <real> pure
```

```

modf: (<real>) → (:pair <real> <real>) pure
r-nextafter: (<real> <real>) → <real> pure
r-copysign: (<real> <real>) → <real> pure
fpclassify: (<real>) → <integer> pure
r-isnormal?: (<real>) → <boolean> pure
r-signbit: (<real>) → <integer> pure

```

Procedure `frexp` returns the fraction of its argument in the head of the result and the exponent in the tail. Procedure `modf` returns the fractional part of its argument in the head of the result and the integer part in the tail. There are no wrappers for `isinf` and `isnan` since similar functions are defined in the core module. See e.g.

[https://en.wikipedia.org/wiki/C\\_mathematical\\_functions](https://en.wikipedia.org/wiki/C_mathematical_functions)

and the GNU libc Reference for further documentation.

## 19.2 Other Simple Procedures

### r-log2-neg

*Syntax:*

```
(r-log2-neg r)
```

*Arguments:*

Name: `r`  
 Type: `<real>`  
 Description: A negative real number

*Result value:* The base 2 logarithm of `r`

*Result type:* `<complex>`

*Purity of the procedure:* pure

The base 2 logarithm of a negative real number  $r$  is  $\log_2 r = \log_2 |r| + i\pi / \ln 2$ .

### i-cbrt

*Syntax:*

```
(i-cbrt i)
```

*Arguments:*

Name: `i`  
 Type: `<integer>`  
 Description: An integer number

*Result value:* The cubic root of the argument

*Result type:* `(:union <real> <integer>)`

*Purity of the procedure:* pure

It seems that this procedure can't always detect integer valued results because of floating point errors.

## rat-cbrt

*Syntax:*

```
(rat-cbrt rat)
```

*Arguments:*

Name: `rat`  
 Type: `<rational>`  
 Description: A rational number

*Result value:* The cubic root of the argument

*Result type:* `(:union <real> <rational> <integer>)`

*Purity of the procedure:* pure

It seems that this procedure can't always detect integer or rational valued results because of floating point errors.

## 19.3 Methods

```
exp2: (<number>) → <number> pure abstract
exp2: (<real-number>) → <real-number> pure abstract
exp2: (<complex>) → <complex> pure
exp2: (<real>) → <real> pure = r-exp2
exp2: (<rational>) → <real-number> pure
exp2: (<integer>) → <rational-number> pure
```

These methods compute  $2^x$ .

```
expm1: (<number>) → <number> pure abstract
expm1: (<real-number>) → <real-number> pure abstract
expm1: (<complex>) → <complex> pure
```

```

expm1: (<real>) → <real> pure    =    r-expm1
expm1: (<rational>) → (:union <real> <integer>) pure
expm1: (<integer>) → (:union <real> <integer>) pure

```

These methods compute  $\exp x - 1$ .

```

log2: (<number>) → <number>   pure abstract
log2: (<complex>) → <complex> pure
log2: (<real>) → (:union <real> <complex>) pure
log2: (<rational>) → (:union <integer> <real> <complex>) pure
log2: (<integer>) → (:union <integer> <real> <complex>) pure

```

These methods compute  $\log_2 x$ .

```

log1p: (<number>) → <number>   pure abstract
log1p: (<complex>) → <complex> pure
log1p: (<real>) → (:union <complex> <real>) pure
log1p: (<rational>) → (:union <complex> <real> <integer>) pure
log1p: (<integer>) → (:union <complex> <real> <integer>) pure

```

These methods compute  $\ln(1 + x)$ .

```

cbirt: (<number>) → <number>   pure abstract
cbirt: (<real-number>) → <real-number>   pure abstract
cbirt: (<complex>) → <complex> pure
cbirt: (<real>) → <real> pure    =    r-cbirt
cbirt: (<rational>) → (:union <real> <rational> <integer>) pure
cbirt: (<integer>) → (:union <real> <integer>) pure

```

These methods compute the cubic root of  $x$ . They are defined for negative arguments, too.

```

hypot: (<real-number> <real-number>) → <real-number>   pure abstract
hypot: (<real> <real>) → <real> pure    =    r-hypot
hypot: (<real> <rational>) → <real> pure
hypot: (<real> <integer>) → <real> pure
hypot: (<rational> <real>) → <real> pure
hypot: (<rational> <rational>) → <real-number> pure
hypot: (<rational> <integer>) → <real-number> pure
hypot: (<integer> <real>) → <real> pure
hypot: (<integer> <rational>) → <real-number> pure
hypot: (<integer> <integer>) → (:union <real> <integer>) pure

```

These methods compute  $\sqrt{x^2 + y^2}$ .

```

erf: (<real-number>) → (:union <real> <integer>)   pure abstract
erf: (<real>) → <real> pure    =    r-erf
erf: (<rational>) → (:union <real> <integer>) pure
erf: (<integer>) → (:union <real> <integer>) pure

```

These methods compute  $\operatorname{erf} x$  (the error function).

```

erfc: (<real-number>) → (:union <real> <integer>) pure abstract
erfc: (<real>) → <real> pure = r-erfc
erfc: (<rational>) → (:union <real> <integer>) pure
erfc: (<integer>) → (:union <real> <integer>) pure

```

These methods compute  $\operatorname{erfc} x = 1 - \operatorname{erf} x$  (the complementary error function).

```

lgamma: (<real-number>) → (:union <real> <integer>) pure abstract
lgamma: (<real>) → <real> pure = r-lgamma
lgamma: (<rational>) → (:union <real> <integer>) pure
lgamma: (<integer>)) → (:union <real> <integer>) pure

```

These methods compute  $\ln |\Gamma(x)|$ .

```

tgamma: (<real-number>) → (:union <real> <integer>) pure abstract
tgamma: (<real>) → <real> pure = r-tgamma
tgamma: (<rational>) → (:union <real> <integer>) pure
tgamma: (<integer>)) → (:union <real> <integer>) pure

```

These methods compute  $\Gamma(x)$  (the gamma function). For positive integers the exact value is computed as a factorial.



## Chapter 20

# Module (standard-library posix-math)

This module implements wrapper procedures to many of the mathematical functions in POSIX C not belonging to the C standard. Additionally, some methods are defined. This module works only for the target platform Guile.

See e.g.

[https://en.wikipedia.org/wiki/C\\_POSIX\\_library](https://en.wikipedia.org/wiki/C_POSIX_library)

and the GNU libc Reference for further documentation.

### 20.1 Wrapper Procedures for POSIX C Functions

r-j0: (<real>) → <real> pure  
r-j1: (<real>) → <real> pure  
r-jn: (<integer> <real>) → <real> pure

r-y0: (<real>) → <real> pure  
r-y1: (<real>) → <real> pure  
r-yn: (<integer> <real>) → <real> pure

### 20.2 Methods

j0: (<real-number>) → <real> pure abstract  
j0: (<real>) → <real> pure = r-j0  
j0: (<rational>) → <real> pure  
j0: (<integer>) → <real> pure

These methods compute the Bessel function j0.

```

j1: (<real-number>) → <real>  pure abstract
j1: (<real>) → <real> pure    =    r-j1
j1: (<rational>) → <real> pure
j1: (<integer>) → <real> pure

```

These methods compute the Bessel function `j1`.

```

jn: (<integer> <real-number>) → <real>  pure abstract
jn: (<integer> <real>) → <real> pure    =    r-jn
jn: (<integer> <rational>) → <real> pure
jn: (<integer> <integer>) → <real> pure

```

These methods compute the Bessel functions `jn`.

```

y0: (<real-number>) → <real>  pure abstract
y0: (<real>) → <real> pure    =    r-y0
y0: (<rational>) → <real> pure
y0: (<integer>) → <real> pure

```

These methods compute the Bessel function `y0`.

```

y1: (<real-number>) → <real>  pure abstract
y1: (<real>) → <real> pure    =    r-y1
y1: (<rational>) → <real> pure
y1: (<integer>) → <real> pure

```

These methods compute the Bessel function `y1`.

```

yn: (<integer> <real-number>) → <real>  pure abstract
yn: (<integer> <real>) → <real> pure    =    r-yn
yn: (<integer> <rational>) → <real> pure
yn: (<integer> <integer>) → <real> pure

```

These methods compute the Bessel functions `yn`.



## Chapter 21

# Module (standar1d-library matrix)

### 21.1 Data Types

*Data type name:* `:matrix`

*Type:* `<param-class>`

*Number of type parameters:* 1

*Description:* A complex number

*Data type name:* `:diagonal-matrix`

*Type:* `<param-class>`

*Number of type parameters:* 1

*Description:* A complex number

Class `<complex>` is equal by value, not inheritable, and not immutable. Note that the indices of the matrices have base zero.

### 21.2 Parametrized Procedures

`matrix=`

*Syntax:*

`(matrix= mx1 mx2)`

*Type parameters:* `%number`

*Arguments:*

Name: `mx1`

Type: (:matrix %number)  
Description: A matrix

Name: mx2  
Type: (:matrix %number)  
Description: A matrix

*Result value:* #t iff the arguments are numerically equal  
*Result type:* <boolean>

*Purity of the procedure:* pure

## diagonal-matrix=

*Syntax:*

(diagonal-matrix= mx1 mx2)

*Type parameters:* %number

*Arguments:*

Name: mx1  
Type: (:diagonal-matrix %number)  
Description: A matrix

Name: mx2  
Type: (:diagonal-matrix %number)  
Description: A matrix

*Result value:* #t iff the arguments are numerically equal  
*Result type:* <boolean>

*Purity of the procedure:* pure

## matrix-diagonal-matrix=

*Syntax:*

(matrix-diagonal-matrix= mx1 mx2)

*Type parameters:* %number

*Arguments:*

Name: `mx1`  
 Type: `(:matrix %number)`  
 Description: A matrix

Name: `mx2`  
 Type: `(:diagonal-matrix %number)`  
 Description: A matrix

*Result value:* `#t` iff the arguments are numerically equal

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `diagonal-matrix-matrix=`

*Syntax:*

```
(diagonal-matrix-matrix= mx1 mx2)
```

*Type parameters:* `%number`

*Arguments:*

Name: `mx1`  
 Type: `(:diagonal-matrix %number)`  
 Description: A matrix

Name: `mx2`  
 Type: `(:matrix %number)`  
 Description: A matrix

*Result value:* `#t` iff the arguments are numerically equal

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## `column-vector`

*Syntax:*

```
(column-vector lst)
```

*Type parameters:* %number

*Arguments:*

Name: `lst`  
Type: `(:uniform-list %number)`  
Description: The contents of the vector

*Result value:* A column vector constructed from the argument list

*Result type:* `(:matrix %number)`

*Purity of the procedure:* pure

## diagonal-matrix

*Syntax:*

`(diagonal-matrix lst)`

*Type parameters:* %number

*Arguments:*

Name: `lst`  
Type: `(:uniform-list %number)`  
Description: The contents of the diagonal

*Result value:* A diagonal matrix constructed from the argument list

*Result type:* `(:diagonal-matrix %number)`

*Purity of the procedure:* pure

## diagonal-matrix\*

*Syntax:*

`(diagonal-matrix* mx1 mx2)`

*Type parameters:* %number

*Arguments:*

Name: `mx1`

Type: (:diagonal-matrix %number)  
Description: A diagonal matrix

Name: mx2  
Type: (:diagonal-matrix %number)  
Description: A diagonal matrix

*Result value:* Product of the given diagonal matrices

*Result type:* (:diagonal-matrix %number)

*Purity of the procedure:* pure

## diagonal-matrix+

*Syntax:*

```
(diagonal-matrix+ mx1 mx2)
```

*Type parameters:* %number

*Arguments:*

Name: mx1  
Type: (:diagonal-matrix %number)  
Description: A diagonal matrix

Name: mx2  
Type: (:diagonal-matrix %number)  
Description: A diagonal matrix

*Result value:* Sum of the given diagonal matrices

*Result type:* (:diagonal-matrix %number)

*Purity of the procedure:* pure

## diagonal-matrix-

*Syntax:*

```
(diagonal-matrix- mx1 mx2)
```

*Type parameters:* %number

*Arguments:*

Name: `mx1`  
Type: `(:diagonal-matrix %number)`  
Description: A diagonal matrix

Name: `mx2`  
Type: `(:diagonal-matrix %number)`  
Description: A diagonal matrix

*Result value:* Difference of the given diagonal matrices

*Result type:* `(:diagonal-matrix %number)`

*Purity of the procedure:* pure

## diagonal-matrix-copy

*Syntax:*

```
(diagonal-matrix-copy mx)
```

*Type parameters:* `%number`

*Arguments:*

Name: `mx`  
Type: `(:diagonal-matrix %number)`  
Description: A diagonal matrix

*Result value:* A copy of the given diagonal matrix

*Result type:* `(:diagonal-matrix %number)`

*Purity of the procedure:* pure

The contents of the argument and result matrices will be different objects.

## diagonal-matrix-ref

*Syntax:*

```
(diagonal-matrix-ref mx index)
```

*Type parameters:* `%number`

*Arguments:*

Name: `mx`  
Type: `(:diagonal-matrix %number)`  
Description: A diagonal matrix

Name: `index`  
Type: `<integer>`  
Description: Index to the element

*Result value:* An element of the diagonal matrix

*Result type:* `%number`

*Purity of the procedure:* pure

**diagonal-matrix-set!***Syntax:*

```
(diagonal-matrix-set! mx index value)
```

*Type parameters:* `%number`

*Arguments:*

Name: `mx`  
Type: `(:diagonal-matrix %number)`  
Description: A diagonal matrix

Name: `index`  
Type: `<integer>`  
Description: Index to the element

Name: `value`  
Type: `%number`  
Description: The new value of the element

No result value.

*Purity of the procedure:* nonpure

**make-column-vector**

*Syntax:*

```
(make-column-vector len element-value)
```

*Type parameters:* %number

*Arguments:*

Name: `len`  
Type: `<integer>`  
Description: The length of the vector

Name: `element-value`  
Type: %number  
Description: A value to fill the vector

*Result value:* A column vector  
*Result type:* (:matrix %number)

*Purity of the procedure:* pure

## make-diagonal-matrix

*Syntax:*

```
(make-diagonal-matrix len element-value)
```

*Type parameters:* %number

*Arguments:*

Name: `len`  
Type: `<integer>`  
Description: The number of rows and columns in the diagonal matrix

Name: `element-value`  
Type: %number  
Description: A value to fill the diagonal

*Result value:* A diagonal matrix  
*Result type:* (:diagonal-matrix %number)

*Purity of the procedure:* pure



**make-matrix**

*Syntax:*

```
(make-matrix rows columns element-value)
```

*Type parameters:* %number

*Arguments:*

Name: **rows**  
Type: <integer>  
Description: Number of rows in the matrix

Name: **columns**  
Type: <integer>  
Description: Number of columns in the matrix

Name: **element-value**  
Type: %number  
Description: A value to fill the matrix

*Result value:* A matrix

*Result type:* (:matrix %number)

*Purity of the procedure:* pure

**make-row-vector**

*Syntax:*

```
(make-row-vector len element-value)
```

*Type parameters:* %number

*Arguments:*

Name: **len**  
Type: <integer>  
Description: The length of the vector

Name: **element-value**  
Type: %number  
Description: A value to fill the vector

*Result value:* A row vector  
*Result type:* (:matrix %number)

*Purity of the procedure:* pure

## matrix

*Syntax:*

```
(matrix lst)
```

*Type parameters:* %number

*Arguments:*

Name: `lst`  
Type: (:uniform-list (:uniform-list %number))  
Description: The contents of the matrix

*Result value:* A matrix constructed from the argument list  
*Result type:* (:matrix %number)

*Purity of the procedure:* pure

The argument type shall be a list of number lists. Each sublist gives the contents of one row in the matrix. All of the sublists must have equal lengths.

## matrix\*

*Syntax:*

```
(matrix* mx1 mx2)
```

*Type parameters:* %number

*Arguments:*

Name: `mx1`  
Type: (:matrix %number)  
Description: A matrix

Name: `mx2`  
Type: (:matrix %number)  
Description: A matrix

*Result value:* Product of the given matrices

*Result type:* (:matrix %number)

*Purity of the procedure:* pure

## matrix+

*Syntax:*

```
(matrix+ mx1 mx2)
```

*Type parameters:* %number

*Arguments:*

Name: mx1

Type: (:matrix %number)

Description: A matrix

Name: mx2

Type: (:matrix %number)

Description: A matrix

*Result value:* Sum of the given matrices

*Result type:* (:matrix %number)

*Purity of the procedure:* pure

## matrix-

*Syntax:*

```
(matrix- mx1 mx2)
```

*Type parameters:* %number

*Arguments:*

Name: mx1

Type: (:matrix %number)

Description: A matrix

Name: `mx2`  
Type: `(:matrix %number)`  
Description: A matrix

*Result value:* Difference of the given matrices  
*Result type:* `(:matrix %number)`

*Purity of the procedure:* pure

## matrix-copy

*Syntax:*

```
(matrix-copy mx)
```

*Type parameters:* `%number`

*Arguments:*

Name: `mx`  
Type: `(:matrix %number)`  
Description: A matrix

*Result value:* A copy of the given matrix  
*Result type:* `(:matrix %number)`

*Purity of the procedure:* pure

The contents of the argument and result matrices will be different objects.

## row-vector

*Syntax:*

```
(row-vector lst)
```

*Type parameters:* `%number`

*Arguments:*

Name: `lst`  
Type: `(:uniform-list %number)`  
Description: The contents of the vector

*Result value:* A row vector constructed from the argument list

*Result type:* (:matrix %number)

*Purity of the procedure:* pure

## 21.3 Parametrized Methods

=

*Syntax:*

(= mx1 mx2)

*Type parameters:* %number

*Arguments:*

Name: mx1

Type: (:matrix %number) or (:diagonal-matrix %number)

Description: A matrix

Name: mx2

Type: (:matrix %number) or (:diagonal-matrix %number)

Description: A matrix

*Result value:* #t iff the arguments are numerically equal

*Result type:* <boolean>

*Purity of the procedure:* pure

All combinations of (:matrix %number) and (:diagonal-matrix %number) as argument types are supported.

\*

*Syntax:*

(\* mx1 mx2)

*Type parameters:* %number

*Arguments:*

Name: `mx1`  
 Type: `(:matrix %number)` or `(:diagonal-matrix %number)`  
 Description: A matrix

Name: `mx2`  
 Type: `(:matrix %number)` or `(:diagonal-matrix %number)`  
 Description: A matrix

*Result value:* The product of the matrices

*Result type:* `%number`

*Purity of the procedure:* pure

All combinations of `(:matrix %number)` and `(:diagonal-matrix %number)` as argument types are supported.

\*

*Syntax:*

`(* nr mx)`

*Type parameters:* `%number`

*Arguments:*

Name: `nr`  
 Type: `%number`  
 Description: A scalar

Name: `mx`  
 Type: `(:matrix %number)` or `(:diagonal-matrix %number)`  
 Description: A matrix

*Result value:* The product of the number and the matrix

*Result type:* `(:matrix %number)` or `(:diagonal-matrix %number)`

*Purity of the procedure:* pure

The result type is the same as the type of argument `mx`.

\*

*Syntax:*

`(* mx nr)`

*Type parameters:* %number

*Arguments:*

Name: `mx`  
 Type: `(:matrix %number)` or `(:diagonal-matrix %number)`  
 Description: A matrix

Name: `nr`  
 Type: %number  
 Description: A scalar

*Result value:* The product of the matrix and the number

*Result type:* `(:matrix %number)` or `(:diagonal-matrix %number)`

*Purity of the procedure:* pure

The result type is the same as the type of argument `mx`.

/

*Syntax:*

`(/ mx nr)`

*Type parameters:* %number

*Arguments:*

Name: `mx`  
 Type: `(:matrix %number)` or `(:diagonal-matrix %number)`  
 Description: A matrix

Name: `nr`  
 Type: %number  
 Description: A scalar

*Result value:* The quotient of the matrix and the number

*Result type:* `(:matrix %number)` or `(:diagonal-matrix %number)`

*Purity of the procedure:* pure

The result type is the same as the type of argument `mx`.

+

*Syntax:*

(+ mx1 mx2)

*Type parameters:* %number

*Arguments:*

Name: mx1

Type: (:matrix %number) or (:diagonal-matrix %number)

Description: A matrix

Name: mx2

Type: (:matrix %number) or (:diagonal-matrix %number)

Description: A matrix

*Result value:* The sum of the matrices

*Result type:* %number

*Purity of the procedure:* pure

All combinations of (:matrix %number) and (:diagonal-matrix %number) as argument types are supported.

-

*Syntax:*

(- mx)

*Type parameters:* %number

*Arguments:*

Name: mx

Type: (:matrix %number) or (:diagonal-matrix %number)

Description: A matrix

*Result value:* The opposite matrix

*Result type:* %number

*Purity of the procedure:* pure

The result type is the same as the type of argument mx.



-

*Syntax:*`(- mx1 mx2)`*Type parameters:* `%number`*Arguments:*Name: `mx1`Type: `(:matrix %number)` or `(:diagonal-matrix %number)`

Description: A matrix

Name: `mx2`Type: `(:matrix %number)` or `(:diagonal-matrix %number)`

Description: A matrix

*Result value:* The difference of the matrices*Result type:* `%number`*Purity of the procedure:* pure

All combinations of `(:matrix %number)` and `(:diagonal-matrix %number)` as argument types are supported.

## **matrix-ref**

*Syntax:*`(matrix-ref mx row column)`*Type parameters:* `%number`*Arguments:*Name: `mx`Type: `(:matrix %number)`

Description: A matrix

Name: `row`Type: `<integer>`

Description: Row index

Name: `column`Type: `<integer>`

Description: Column index

*Result value:* The element of the matrix at the given position

*Result type:* `%number`

*Purity of the procedure:* pure

## `matrix-ref`

*Syntax:*

```
(matrix-ref mx row column)
```

*Type parameters:* `%number`

*Arguments:*

Name: `mx`

Type: `(:diagonal-matrix %number)`

Description: A matrix

Name: `row`

Type: `<integer>`

Description: Row index

Name: `column`

Type: `<integer>`

Description: Column index

*Result value:* The element of the matrix at the given position

*Result type:* `%number`

*Purity of the procedure:* pure

Note that elements outside the diagonal are zero.

## `matrix-set!`

*Syntax:*

```
(matrix-set! mx row column element-value)
```

*Type parameters:* `%number`

*Arguments:*

Name: `mx`  
 Type: `(:matrix %number)`  
 Description: A matrix

Name: `row`  
 Type: `<integer>`  
 Description: Row index

Name: `column`  
 Type: `<integer>`  
 Description: Column index

Name: `element-value`  
 Type: `%number`  
 Description: The new value at the specified position

No result value.

*Purity of the procedure:* nonpure

**matrix-set!***Syntax:*

```
(matrix-set! mx row column element-value)
```

*Type parameters:* `%number`

*Arguments:*

Name: `mx`  
 Type: `(:diagonal-matrix %number)`  
 Description: A matrix

Name: `row`  
 Type: `<integer>`  
 Description: Row index

Name: `column`  
 Type: `<integer>`  
 Description: Column index

Name: `element-value`  
 Type: `%number`

Description: The new value at the specified position

No result value.

*Purity of the procedure:* nonpure

The row and column indices have to be equal.

## number-of-columns

*Syntax:*

```
(number-of-columns mx)
```

*Type parameters:* %number

*Arguments:*

Name: mx  
Type: (:matrix %number)  
Description: A matrix

*Result value:* Number of columns in the matrix

*Result type:* <integer>

*Purity of the procedure:* pure

## number-of-columns

*Syntax:*

```
(number-of-columns mx)
```

*Type parameters:* %number

*Arguments:*

Name: mx  
Type: (:diagonal-matrix %number)  
Description: A matrix

*Result value:* Length of the diagonal

*Result type:* <integer>

*Purity of the procedure:* pure

## number-of-rows

*Syntax:*

```
(number-of-rows mx)
```

*Type parameters:* %number

*Arguments:*

Name: mx  
Type: (:matrix %number)  
Description: A matrix

*Result value:* Number of rows in the matrix

*Result type:* <integer>

*Purity of the procedure:* pure

## number-of-rows

*Syntax:*

```
(number-of-rows mx)
```

*Type parameters:* %number

*Arguments:*

Name: mx  
Type: (:diagonal-matrix %number)  
Description: A matrix

*Result value:* Length of the diagonal

*Result type:* <integer>

*Purity of the procedure:* pure



## Chapter 22

# Module (standard-library dynamic-list)

### 22.1 Simple Procedures

#### d-append

*Syntax:*

```
(d-append lst-1 ... lst-n)
```

*Arguments:*

Name: `lst-k`  
Type: `<object>`  
Description: A list

*Result value:* A list constructed by concatenating the argument lists

*Result type:* `<object>`

*Purity of the procedure:* pure

The lists are concatenated in the order they are given.

#### d-car

*Syntax:*

```
(d-car obj)
```

*Arguments:*

Name: `obj`  
Type: `<object>`  
Description: An object

*Result value:* The head of the pair

*Result type:* `<object>`

*Purity of the procedure:* pure

If the argument is not a pair an exception is raised.

## **d-cdr**

*Syntax:*

```
(d-cdr obj)
```

*Arguments:*

Name: `obj`  
Type: `<object>`  
Description: An object

*Result value:* The tail of the pair

*Result type:* `<object>`

*Purity of the procedure:* pure

If the argument is not a pair an exception is raised.

## **d-for-each**

*Syntax:*

```
(d-for-each proc lst-1 ... lst-n)
```

*Arguments:*

Name: `proc`  
Type: `(:procedure (<object>) <none> nonpure)`  
Description: A procedure to be applied into the given lists

Name: `lst-k`



Type: <object>  
Description: A list

No result value.

*Purity of the procedure:* nonpure

This procedure is similar to `for-each`, see section 4.5.3. The given procedure is applied to the given lists and the results are discarded.

## d-for-each1

*Syntax:*

```
(d-for-each1 proc lst)
```

*Arguments:*

Name: `proc`  
Type: (:procedure (<object>) <none> nonpure)  
Description: A procedure to be applied into the given list

Name: `lst`  
Type: <object>  
Description: A list

No result value.

*Purity of the procedure:* nonpure

This procedure applies the given procedure to the given list and discards the results.

## d-list

*Syntax:*

```
(d-list obj-1 ... obj-n)
```

*Arguments:*

Name: `obj-k`  
Type: <object>  
Description: An object

*Result value:* A list constructed from the arguments

*Result type:* <object>

*Purity of the procedure:* pure

## d-list-ref

*Syntax:*

```
(d-list-ref lst index)
```

*Arguments:*

Name: `lst`

Type: <object>

Description: A list

Name: `index`

Type: <integer>

Description: Index to the list

*Result value:* The object at the specified position in the given list

*Result type:* <object>

*Purity of the procedure:* pure

## d-map

*Syntax:*

```
(d-map proc lst-1 ... lst-n)
```

*Arguments:*

Name: `proc`

Type: (:procedure (<object>) <object> pure)

Description: A procedure to be applied into the given list

Name: `lst-k`

Type: <object>

Description: A list

*Result value:* A list constructed by applying the procedure to the elements of

the lists

*Result type:* <object>

*Purity of the procedure:* pure

This procedure is similar to `map`, see section 4.5.3.

## d-map1

*Syntax:*

```
(d-map1 proc lst)
```

*Arguments:*

Name: `proc`

Type: (:procedure (<object>) <object> pure)

Description: A procedure to be applied into the given list

Name: `lst`

Type: <object>

Description: A list

*Result value:* A list constructed by applying the procedure to each element of the list

*Result type:* <object>

*Purity of the procedure:* pure

## d-map-nonpure

*Syntax:*

```
(d-map-nonpure proc lst-1 ... lst-n)
```

*Arguments:*

Name: `proc`

Type: (:procedure (<object>) <object> nonpure)

Description: A procedure to be applied into the given lists

Name: `lst`

Type: <object>

Description: A list

*Result value:* A list constructed by applying the procedure to the elements of the lists

*Result type:* <object>

*Purity of the procedure:* nonpure

This procedure is similar to `map-nonpure`, see section 4.5.3.

## d-map-nonpure1

*Syntax:*

```
(d-map-nonpure1 proc lst)
```

*Arguments:*

Name: `proc`

Type: `(:procedure (<object>) <object> nonpure)`

Description: A procedure to be applied into the given list

Name: `lst`

Type: <object>

Description: A list

*Result value:* A list constructed by applying the procedure to each element of the list

*Result type:* <object>

*Purity of the procedure:* nonpure

## Chapter 23

# Module (standard-library singleton)

### 23.1 Data Types

*Data type name:* `:singleton`

*Type:* `<param-logical-type>`

*Number of type parameters:* 1

*Description:* A singleton object

A singleton is an object containing a single value.

### 23.2 Parametrized Procedures

#### `make-singleton`

*Syntax:*

```
(make-singleton element)
```

*Type parameters:* `%type`

*Arguments:*

Name: `element`

Type: `%type`

Description: An object

*Result value:* A new singleton object containing the given value

*Result type:* `(:singleton %type)`

*Purity of the procedure:* pure

## singleton-get-element

*Syntax:*

```
(singleton-get-element sgt)
```

*Type parameters:* %type

*Arguments:*

Name: `sgt`  
Type: `(:singleton %type)`  
Description: A singleton

*Result value:* The value contained in the argument object

*Result type:* %type

*Purity of the procedure:* pure

## singleton-set-element!

*Syntax:*

```
(singleton-set-element! sgt new-element)
```

*Type parameters:* %type

*Arguments:*

Name: `sgt`  
Type: `(:singleton %type)`  
Description: A singleton

Name: `new-element`  
Type: %type  
Description: The new element value

No result value.

*Purity of the procedure:* nonpure

The element of the singleton `sgt` is set to `new-element`.





## Chapter 24

# Module (standard-library hash-table)

This module is deprecated and works only for Guile target platform. You should use module (standard-library hash-table2) instead.

When a hash table is used the hash procedure and the equality predicate used by the association procedure must be compatible with each other, i.e. the hash procedure shall never compute different hash values for objects that are equal by the equality predicate.

When you create object, string, or symbol hash tables you have to manually dispatch the value type. For example to create a string hash table with symbols as the value type use code

```
((param-proc-dispatch make-string-hash-table-with-size <symbol>)
 100)
```

### 24.1 Data Types

*Data type name:* <raw-hash-table>

*Type:* <class>

*Description:* The low-level guile hash table class. This class should not be used directly.

*Data type name:* :hash-proc

*Type:* parametrized procedure class

*Number of type parameters:* 1

*Description:* The type of a hash procedure. The type parameter is the type of the values to be hashed.

*Data type name:* :assoc-proc

*Type:* parametrized procedure class

*Number of type parameters:* 2

*Description:* The type of an association procedure for hash tables. The first type parameter is the type of the key and the second the type of the values with which the keys are associated.

*Data type name:* `:hash-table`

*Type:* `<param-class>`

*Number of type parameters:* 2

*Description:* The parametrized class for hash tables. The first parameter is the type of the keys and the second the type of the values with which the keys are associated.

*Data type name:* `:object-hash-table`

*Type:* `<param-class>`

*Number of type parameters:* 1

*Description:* The parametrized class for hash tables for which the keys are arbitrary objects. The type parameter is the type of the associated values.

The hash procedure of an object hash table is compatible with the association procedure `assoc-objects1` with the following key types:

- symbols
- booleans
- characters
- strings
- user defined nonprimitive classes
- pairs
- vectors (all four kinds of vectors)

Procedure `assoc-objects1` uses equivalence predicate `equal-objects?`. Note that if you use this class with string or pair keys the keys are considered equal if they are the same object.

*Data type name:* `:string-hash-table`

*Type:* `<param-class>`

*Number of type parameters:* 1

*Description:* The parametrized class for hash tables for which the keys are strings. The type parameter is the type of the associated values.

*Data type name:* `:symbol-hash-table`

*Type:* `<param-class>`

*Number of type parameters:* 1

*Description:* The parametrized class for hash tables for which the keys are symbols. The type parameter is the type of the associated values.

## 24.2 Simple Procedures

### object-hash

*Syntax:*

(object-hash obj size)

*Arguments:*

Name: `obj`

Type: `<object>`

Description: The object for which the hash value is computed

Name: `size`

Type: `<integer>`

Description: The size of the hash table for which the hash value is computed.

*Result value:* Hash value

*Result type:* `<integer>`

*Purity of the procedure:* pure

### string-hash

*Syntax:*

(string-hash str size)

*Arguments:*

Name: `str`

Type: `<string>`

Description: The string for which the hash value is computed

Name: `size`

Type: `<integer>`

Description: The size of the hash table for which the hash value is computed.

*Result value:* Hash value

*Result type:* `<integer>`

*Purity of the procedure:* pure

## hashq

*Syntax:*

```
(hashq x size)
```

*Arguments:*

Name: **x**

Type: <object>

Description: The object for which the hash value is computed

Name: **size**

Type: <integer>

Description: The size of the hash table for which the hash value is computed.

*Result value:* Hash value

*Result type:* <integer>

*Purity of the procedure:* pure

This procedure computes a hash value with Scheme procedure `hashq`. This procedure is compatible with the Scheme predicate `eq?`. See [2, chapter 6.1].

## hashv

*Syntax:*

```
(hashv x size)
```

*Arguments:*

Name: **x**

Type: <object>

Description: The object for which the hash value is computed

Name: **size**

Type: <integer>

Description: The size of the hash table for which the hash value is computed.

*Result value:* Hash value

*Result type:* <integer>

*Purity of the procedure:* pure

This procedure computes a hash value with Scheme procedure `hashv`. This procedure is compatible with the Scheme predicate `eqv?`. See [2, chapter 6.1].

## hash-contents

*Syntax:*

```
(hash-contents x size)
```

*Arguments:*

Name: `x`

Type: <object>

Description: The object for which the hash value is computed

Name: `size`

Type: <integer>

Description: The size of the hash table for which the hash value is computed.

*Result value:* Hash value

*Result type:* <integer>

*Purity of the procedure:* pure

This procedure computes a hash value with Scheme procedure `hash`. This procedure is compatible with the Scheme predicate `equal?` (not the similar Theme-D predicate). See [2, chapter 6.1].

## 24.3 Parametrized Procedures

### hash-clear!

*Syntax:*

```
(hash-clear! hashtable)
```

*Type parameters:* %key, %value

*Arguments:*

Name: `hashtable`  
Type: `(:hash-table %key %value)`  
Description: A hash table

No result value.

*Purity of the procedure:* nonpure

This procedure removes all the element from the argument hash table.

## hash-count-elements

*Syntax:*

`(hash-count-elements hashtable)`

*Type parameters:* `%key`, `%value`

*Arguments:*

Name: `hashtable`  
Type: `(:hash-table %key %value)`  
Description: A hash table

*Result value:* The number of elements in the hash table

*Result type:* `<integer>`

*Purity of the procedure:* pure

## hash-exists?

*Syntax:*

`(hash-exists? hashtable key)`

*Type parameters:* `%key`, `%value`

*Arguments:*

Name: `hashtable`  
Type: `(:hash-table %key %value)`  
Description: A hash table

Name: `key`  
Type: `%key`  
Description: Key to be searched

*Result value:* Returns `#t` iff the given key is found from the hash table

*Result type:* `<boolean>`

*Purity of the procedure:* pure

## hash-for-each

*Syntax:*

```
(hash-for-each proc hashtable)
```

*Type parameters:* `%key`, `%value`

*Arguments:*

Name: `proc`  
Type: `(:procedure (%key %value) <none> nonpure)`  
Description: The procedure to be called

Name: `hashtable`  
Type: `(:hash-table %key %value)`  
Description: A hash table

No result value.

*Purity of the procedure:* nonpure

This procedure calls the given procedure for all elements in the hash table.

## hash-ref

*Syntax:*

```
(hash-ref hashtable key default)
```

*Type parameters:* `%key`, `%value`, `%default`

*Arguments:*

Name: `hashtable`  
 Type: `(:hash-table %key %value)`  
 Description: A hash table

Name: `key`  
 Type: `%key`  
 Description: Key to be searched

Name: `default`  
 Type: `%default`  
 Description: The default value

*Result value:* The value associated with the given key in the hash table. Returns `default` if the key is not found.

*Result type:* `(:union %value %default)`

*Purity of the procedure:* pure

## `hash-remove!`

*Syntax:*

```
(hash-remove! hashtable key)
```

*Type parameters:* `%key`, `%value`

*Arguments:*

Name: `hashtable`  
 Type: `(:hash-table %key %value)`  
 Description: A hash table

Name: `key`  
 Type: `%key`  
 Description: Key to be defined

No result value.

*Purity of the procedure:* nonpure

This procedure removes the given key from the hash table. If the key is not found the procedure does nothing.

## `hash-set!`



*Syntax:*

```
(hash-set! hashtable key value)
```

*Type parameters:* %key, %value

*Arguments:*

Name: `hashtable`  
 Type: `(:hash-table %key %value)`  
 Description: A hash table

Name: `key`  
 Type: `%key`  
 Description: Key to be defined

Name: `value`  
 Type: `%value`  
 Description: Value to be associated

No result value.

*Purity of the procedure:* nonpure

This procedure associates the given key with the given value in the hash table.

## make-hash-table

*Syntax:*

```
(make-hash-table proc-hash proc-assoc)
```

*Type parameters:* %key, %value

*Arguments:*

Name: `proc-hash`  
 Type: `(:hash-proc %key)`  
 Description: A procedure to compute hash values

Name: `proc-assoc`  
 Type: `(:assoc-proc %key %value)`  
 Description: A procedure to associate keys and values

*Result value:* A hash table

*Result type:* `(:hash-table %key %value)`

*Purity of the procedure:* pure

## make-hash-table-with-size

*Syntax:*

```
(make-hash-table-with-size proc-hash proc-assoc i-size)
```

*Type parameters:* %key, %value

*Arguments:*

Name: proc-hash  
Type: (:hash-proc %key)  
Description: A procedure to compute hash values

Name: proc-assoc  
Type: (:assoc-proc %key %value)  
Description: A procedure to associate keys and values

Name: i-size  
Type: <integer>  
Description: Size of the hash table

*Result value:* A hash table with given size  
*Result type:* (:hash-table %key %value)

*Purity of the procedure:* pure

## make-object-hash-table

*Syntax:*

```
(make-object-hash-table)
```

*Type parameters:* %value

No arguments.

*Result value:* An object hash table  
*Result type:* (:object-hash-table %value)

*Purity of the procedure:* pure

### make-object-hash-table-with-size

*Syntax:*

```
(make-object-hash-table-with-size i-size)
```

*Type parameters:* %value

*Arguments:*

Name: i-size  
Type: <integer>  
Description: Size of the hash table

*Result value:* An object hash table with given size

*Result type:* (:object-hash-table %value)

*Purity of the procedure:* pure

### make-string-hash-table

*Syntax:*

```
(make-string-hash-table)
```

*Type parameters:* %value

No arguments.

*Result value:* A string hash table

*Result type:* (:string-hash-table %value)

*Purity of the procedure:* pure

### make-string-hash-table-with-size

*Syntax:*

```
(make-string-hash-table-with-size i-size)
```

*Type parameters:* %value

*Arguments:*

Name: i-size  
Type: <integer>  
Description: Size of the hash table

*Result value:* A string hash table with given size

*Result type:* (:string-hash-table %value)

*Purity of the procedure:* pure

## make-symbol-hash-table

*Syntax:*

```
(make-symbol-hash-table)
```

*Type parameters:* %value

No arguments.

*Result value:* A symbol hash table

*Result type:* (:symbol-hash-table %value)

*Purity of the procedure:* pure

## make-symbol-hash-table-with-size

*Syntax:*

```
(make-symbol-hash-table-with-size i-size)
```

*Type parameters:* %value

*Arguments:*

Name: i-size  
Type: <integer>

Description: Size of the hash table

*Result value:* A symbol hash table with given size

*Result type:* (:symbol-hash-table %value)

*Purity of the procedure:* pure

## assoc1

*Syntax:*

```
(assoc1 key lst)
```

*Type parameters:* %type1, %type2

*Arguments:*

Name: key

Type: %type1

Description: The key to be searched

Name: lst

Type: (:alist %key %value)

Description: The association list from which the object is searched

*Result value:* The association or #f if none is found.

*Result type:* (:union (:pair %type2 %type2) <boolean>)

*Purity of the procedure:* pure

This procedure uses the equivalence predicate `equal?`.

## assoc-values1

*Syntax:*

```
(assoc-values1 key lst)
```

*Type parameters:* %type1, %type2

*Arguments:*

Name: key

Type: %type1

Description: The key to be searched

Name: `lst`

Type: `(:alist %key %value)`

Description: The association list from which the object is searched

*Result value:* The association or `#f` if none is found.

*Result type:* `(:union (:pair %type2 %type2) <boolean>)`

*Purity of the procedure:* pure

This procedure uses the equivalence predicate `equal-values?`.

## assoc-objects1

*Syntax:*

```
(assoc-objects1 key lst)
```

*Type parameters:* `%type1`, `%type2`

*Arguments:*

Name: `key`

Type: `%type1`

Description: The key to be searched

Name: `lst`

Type: `(:alist %key %value)`

Description: The association list from which the object is searched

*Result value:* The association or `#f` if none is found.

*Result type:* `(:union (:pair %type2 %type2) <boolean>)`

*Purity of the procedure:* pure

This procedure uses the equivalence predicate `equal-objects?`.

## assoc-contents1

*Syntax:*

```
(assoc-contents1 key lst)
```

*Type parameters:* `%type1`, `%type2`

*Arguments:*

Name: `key`

Type: `%type1`

Description: The key to be searched

Name: `lst`

Type: `(:alist %key %value)`

Description: The association list from which the object is searched

*Result value:* The association or `#f` if none is found.

*Result type:* `(:union (:pair %type2 %type2) <boolean>)`

*Purity of the procedure:* pure

This procedure uses the equivalence predicate `equal-contents?`.





## Chapter 25

# Module (standard-library hash-table2)

### 25.1 Data Types

*Data type name:* `:hash-proc`

*Type:* parametrized procedure class

*Number of type parameters:* 1

*Description:* The type of a hash procedure. The type parameter is the type of the values to be hashed.

*Data type name:* `:assoc-proc`

*Type:* parametrized procedure class

*Number of type parameters:* 2

*Description:* The type of an association procedure for hash tables. The first type parameter is the type of the key and the second the type of the values with which the keys are associated.

*Data type name:* `:alist-delete-proc`

*Type:* parametrized procedure class

*Number of type parameters:* 2

*Description:* The type of a procedure used to delete objects from an association list. The first type parameter is the type of the key and the second the type of the values with which the keys are associated.

*Data type name:* `:hash-table`

*Type:* `<param-class>`

*Number of type parameters:* 2

*Description:* The parametrized class for hash tables. The first parameter is the type of the keys and the second the type of the values with which the keys are associated.

*Data type name:* `:object-hash-table`

*Type:* `<param-class>`

*Number of type parameters:* 1

*Description:* The parametrized class for hash tables for which the keys are arbitrary objects. The type parameter is the type of the associated values.

The hash procedure of an object hash table is compatible with the association procedure `assoc-objects1` with the following key types:

- symbols
- booleans
- characters
- strings
- user defined nonprimitive classes
- pairs
- vectors (all four kinds of vectors)

Procedure `assoc-objects1` uses equivalence predicate `equal-objects?`. Note that if you use this class with string or pair keys the keys are considered equal if they are the same object.

*Data type name:* `:string-hash-table`

*Type:* `<param-class>`

*Number of type parameters:* 1

*Description:* The parametrized class for hash tables for which the keys are strings. The type parameter is the type of the associated values.

*Data type name:* `:symbol-hash-table`

*Type:* `<param-class>`

*Number of type parameters:* 1

*Description:* The parametrized class for hash tables for which the keys are symbols. The type parameter is the type of the associated values.

## 25.2 Simple Procedures

### `object-hash`

*Syntax:*

`(object-hash obj size)`

*Arguments:*

Name: `obj`

Type: `<object>`

Description: The object for which the hash value is computed

Name: `size`

Type: `<integer>`

Description: The size of the hash table for which the hash value is computed.

*Result value:* Hash value

*Result type:* `<integer>`

*Purity of the procedure:* pure

## string-hash

*Syntax:*

```
(string-hash str size)
```

*Arguments:*

Name: `str`

Type: `<string>`

Description: The string for which the hash value is computed

Name: `size`

Type: `<integer>`

Description: The size of the hash table for which the hash value is computed.

*Result value:* Hash value

*Result type:* `<integer>`

*Purity of the procedure:* pure

## hashq

*Syntax:*

```
(hashq x size)
```

*Arguments:*

Name: `x`

Type: `<object>`

Description: The object for which the hash value is computed

Name: `size`

Type: `<integer>`

Description: The size of the hash table for which the hash value is computed.

*Result value:* Hash value

*Result type:* `<integer>`

*Purity of the procedure:* pure

This procedure computes a hash value with Scheme procedure `hashq`. This procedure is compatible with the Scheme predicate `eq?`. See [2, chapter 6.1].

## hashv

*Syntax:*

```
(hashv x size)
```

*Arguments:*

Name: `x`

Type: `<object>`

Description: The object for which the hash value is computed

Name: `size`

Type: `<integer>`

Description: The size of the hash table for which the hash value is computed.

*Result value:* Hash value

*Result type:* `<integer>`

*Purity of the procedure:* pure

This procedure computes a hash value with Scheme procedure `hashv`. This procedure is compatible with the Scheme predicate `eqv?`. See [2, chapter 6.1].

## hash-contents

*Syntax:*

```
(hash-contents x size)
```

*Arguments:*

Name: `x`  
 Type: `<object>`  
 Description: The object for which the hash value is computed

Name: `size`  
 Type: `<integer>`  
 Description: The size of the hash table for which the hash value is computed.

*Result value:* Hash value

*Result type:* `<integer>`

## 25.3 Parametrized Procedures

### `make-hash-table`

*Syntax:*

```
(make-hash-table proc-hash proc-assoc proc-delete)
```

*Type parameters:* `%key`, `%value`

*Arguments:*

Name: `proc-hash`  
 Type: `(:hash-proc %key)`  
 Description: A procedure to compute hash values

Name: `proc-assoc`  
 Type: `(:assoc-proc %key %value)`  
 Description: A procedure to associate keys and values

Name: `proc-delete`  
 Type: `(:alist-delete-proc %key %value)`  
 Description: A procedure to delete an element from an association list

*Result value:* A hash table

*Result type:* `(:hash-table %key %value)`

*Purity of the procedure:* pure

## make-hash-table-with-size

*Syntax:*

```
(make-hash-table proc-hash proc-assoc proc-delete i-size)
```

*Type parameters:* %key, %value

*Arguments:*

Name: `proc-hash`  
Type: `(:hash-proc %key)`  
Description: A procedure to compute hash values

Name: `proc-assoc`  
Type: `(:assoc-proc %key %value)`  
Description: A procedure to associate keys and values

Name: `proc-delete`  
Type: `(:alist-delete-proc %key %value)`  
Description: A procedure to delete an element from an association list

Name: `i-size`  
Type: `<integer>`  
Description: The size of the hash table

*Result value:* A hash table with the given size

*Result type:* `(:hash-table %key %value)`

*Purity of the procedure:* pure

## make-object-hash-table

*Syntax:*

```
(make-object-hash-table)
```

*Type parameters:* %value

No arguments.

*Result value:* An object hash table

*Result type:* `(:object-hash-table %value)`

*Purity of the procedure:* pure

## make-object-hash-table-with-size

*Syntax:*

```
(make-object-hash-table-with-size i-size)
```

*Type parameters:* %value

*Arguments:*

Name: i-size  
Type: <integer>  
Description: Size of the hash table

*Result value:* An object hash table with given size

*Result type:* (:object-hash-table %value)

*Purity of the procedure:* pure

## make-string-hash-table

*Syntax:*

```
(make-string-hash-table)
```

*Type parameters:* %value

No arguments.

*Result value:* A string hash table

*Result type:* (:string-hash-table %value)

*Purity of the procedure:* pure

## make-string-hash-table-with-size

*Syntax:*

```
(make-string-hash-table-with-size i-size)
```

*Type parameters:* %value

*Arguments:*

Name: `i-size`  
Type: `<integer>`  
Description: Size of the hash table

*Result value:* A string hash table with given size

*Result type:* `(:string-hash-table %value)`

*Purity of the procedure:* pure

## `make-symbol-hash-table`

*Syntax:*

`(make-symbol-hash-table)`

*Type parameters:* `%value`

No arguments.

*Result value:* A symbol hash table

*Result type:* `(:symbol-hash-table %value)`

*Purity of the procedure:* pure

## `make-symbol-hash-table-with-size`

*Syntax:*

`(make-symbol-hash-table-with-size i-size)`

*Type parameters:* `%value`

*Arguments:*

Name: `i-size`  
Type: `<integer>`  
Description: Size of the hash table

*Result value:* A symbol hash table with given size

*Result type:* `(:symbol-hash-table %value)`



*Purity of the procedure:* pure

## hash-clear!

*Syntax:*

```
(hash-clear! hashtable)
```

*Type parameters:* %key, %value

*Arguments:*

Name: hashtable  
Type: (:hash-table %key %value)  
Description: A hash table

No result value.

*Purity of the procedure:* nonpure

This procedure removes all the element from the argument hash table.

## hash-exists?

*Syntax:*

```
(hash-exists? hashtable key)
```

*Type parameters:* %key, %value

*Arguments:*

Name: hashtable  
Type: (:hash-table %key %value)  
Description: A hash table

Name: key  
Type: %key  
Description: Key to be searched

*Result value:* Returns #t iff the given key is found from the hash table

*Result type:* <boolean>

*Purity of the procedure:* pure

## hash-for-each

*Syntax:*

```
(hash-for-each proc hashtable)
```

*Type parameters:* %key, %value

*Arguments:*

Name: `proc`  
Type: `(:procedure (%key %value) <none> nonpure)`  
Description: The procedure to be called

Name: `hashtable`  
Type: `(:hash-table %key %value)`  
Description: A hash table

No result value.

*Purity of the procedure:* nonpure

This procedure calls the given procedure for all elements in the hash table.

## hash-ref

*Syntax:*

```
(hash-ref hashtable key default)
```

*Type parameters:* %key, %value, %default

*Arguments:*

Name: `hashtable`  
Type: `(:hash-table %key %value)`  
Description: A hash table

Name: `key`  
Type: %key  
Description: Key to be searched

Name: `default`  
Type: %default  
Description: The default value

*Result value:* The value associated with the given key in the hash table. Returns `default` if the key is not found.

*Result type:* `(:union %value %default)`

*Purity of the procedure:* pure

## hash-remove!

*Syntax:*

```
(hash-remove! hashtable key)
```

*Type parameters:* `%key`, `%value`

*Arguments:*

Name: `hashtable`  
Type: `(:hash-table %key %value)`  
Description: A hash table

Name: `key`  
Type: `%key`  
Description: Key to be defined

No result value.

*Purity of the procedure:* nonpure

This procedure removes the given key from the hash table. If the key is not found the procedure does nothing.

## hash-set!

*Syntax:*

```
(hash-set! hashtable key value)
```

*Type parameters:* `%key`, `%value`

*Arguments:*

Name: `hashtable`  
Type: `(:hash-table %key %value)`  
Description: A hash table

Name: `key`  
Type: `%key`  
Description: Key to be defined

Name: `value`  
Type: `%value`  
Description: Value to be associated

No result value.

*Purity of the procedure:* nonpure

This procedure associates the given key with the given value in the hash table.

## hash-count

*Syntax:*

```
(hash-count pred hashtable)
```

*Type parameters:* `%key`, `%value`

*Arguments:*

Name: `pred`  
Type: `(:binary-predicate %key %value)`  
Description: A binary predicate

Name: `hashtable`  
Type: `(:hash-table %key %value)`  
Description: A hash table

*Result value:* The number of elements satisfying the predicate in the hash table

*Result type:* `<integer>`

*Purity of the procedure:* pure

## hash-count-elements

*Syntax:*

```
(hash-count-elements hashtable)
```

*Type parameters:* %key, %value

*Arguments:*

Name: hashtable  
 Type: (:hash-table %key %value)  
 Description: A hash table

*Result value:* The number of elements in the hash table

*Result type:* <integer>

*Purity of the procedure:* pure

## hash-search-first

*Syntax:*

```
(hash-search-first pred ht x-default)
```

*Type parameters:* %key, %value, %default

*Arguments:*

Name: pred  
 Type: (:binary-predicate %key %value)  
 Description: A binary predicate

Name: ht  
 Type: (:hash-table %key %value)  
 Description: A hash table

Name: x-default  
 Type: %default  
 Description: The default return value

*Result value:* The first element satisfying the predicate in the hash table

*Result type:* (:union %value %default)

*Purity of the procedure:* pure

If no element satisfies the predicate value `x-default` is returned.

## assoc1

*Syntax:*

```
(assoc1 key lst)
```

*Type parameters:* %type1, %type2

*Arguments:*

Name: `key`  
 Type: %type1  
 Description: The key to be searched

Name: `lst`  
 Type: (:alist %key %value)  
 Description: The association list from which the object is searched

*Result value:* The association or #f if none is found.

*Result type:* (:union (:pair %type2 %type2) <boolean>)

*Purity of the procedure:* pure

This procedure uses the equivalence predicate `equal?`.

## assoc-values1

*Syntax:*

```
(assoc-values1 key lst)
```

*Type parameters:* %type1, %type2

*Arguments:*

Name: `key`  
 Type: %type1  
 Description: The key to be searched

Name: `lst`  
 Type: (:alist %key %value)  
 Description: The association list from which the object is searched

*Result value:* The association or #f if none is found.

*Result type:* (:union (:pair %type2 %type2) <boolean>)

*Purity of the procedure:* pure

This procedure uses the equivalence predicate `equal-values?`.

## `assoc-objects1`

*Syntax:*

```
(assoc-objects1 key lst)
```

*Type parameters:* `%type1`, `%type2`

*Arguments:*

Name: `key`  
Type: `%type1`  
Description: The key to be searched

Name: `lst`  
Type: `(:alist %key %value)`  
Description: The association list from which the object is searched

*Result value:* The association or `#f` if none is found.

*Result type:* `(:union (:pair %type2 %type2) <boolean>)`

*Purity of the procedure:* pure

This procedure uses the equivalence predicate `equal-objects?`.

## `assoc-contents1`

*Syntax:*

```
(assoc-contents1 key lst)
```

*Type parameters:* `%type1`, `%type2`

*Arguments:*

Name: `key`  
Type: `%type1`  
Description: The key to be searched

Name: `lst`  
Type: `(:alist %key %value)`  
Description: The association list from which the object is searched

*Result value:* The association or #f if none is found.

*Result type:* (:union (:pair %type2 %type2) <boolean>)

*Purity of the procedure:* pure

This procedure uses the equivalence predicate `equal-contents?`.



## Chapter 26

# Module (standard-library statprof)

This is a wrapper module for guile profiler `statprof`. Note that all features of `statprof` are not supported. A simple use of `statprof` would look like this:

```
(statprof-reset 0 50000 #f)
(statprof-start)
(do-something)
(statprof-stop)
(statprof-display)
```

See guile 2.0 documentation for further information.

### 26.1 Simple Procedures

#### `statprof-start`

*Syntax:*

```
(statprof-start)
```

No arguments.

No result value.

*Purity of the procedure:* nonpure

Start profiling.

## statprof-stop

*Syntax:*

```
(statprof-stop)
```

No arguments.

No result value.

*Purity of the procedure:* nonpure

Stop profiling.

## statprof-reset

*Syntax:*

```
(statprof-reset sample-seconds sample-microseconds count-calls?)
```

*Arguments:*

Name: `sample-seconds`

Type: `<integer>`

Description: Seconds for the sampler interval

Name: `sample-microseconds`

Type: `<integer>`

Description: Microseconds for the sampler interval

Name: `count-calls?`

Type: `<boolean>`

Description: `#t` to count procedure calls

No result value.

*Purity of the procedure:* nonpure

Reset the profiler.

## statprof-display

*Syntax:*

`(statprof-display)`

No arguments.

No result value.

*Purity of the procedure:* nonpure

Display a summary of the statistics collected.



# Bibliography

- [1] H. G. Baker. Iterators: signs of weakness in object oriented languages. *ACM OOPS Messenger*, 4(3):18–25, 1993. <http://www.pipeline.com/~hbaker1/Iterator.html>.
- [2] A. S. et al. Revised<sup>7</sup> Report on the Algorithmic Language Scheme. 2017. <http://www.r7rs.org/>.
- [3] M. S. et al. Revised<sup>6</sup> Report on the Algorithmic Language Scheme. 2007. <http://www.r6rs.org/>.

# Index

`*`, 102, 199, 220, 255, 262, 287, 288  
`+`, 102, 199, 220, 255, 262, 290  
`-`, 102, 199, 220, 255, 262, 290, 291  
`/`, 102, 199, 220, 255, 262, 289  
`:alist-delete-proc`, 323  
`:alist`, 25  
`:assoc-proc`, 307, 323  
`:binary-predicate`, 103  
`:consumer`, 133  
`:diagonal-matrix`, 275  
`:hash-proc`, 307, 323  
`:hash-table`, 308, 323  
`:iterator-inst`, 133  
`:iterator`, 133  
`:matrix`, 275  
`:maybe`, 25  
`:nonempty-alist`, 25  
`:nonempty-nonpure-stream`, 123  
`:nonempty-stream`, 123  
`:nonempty-uniform-list`, 25  
`:nonpure-consumer`, 141  
`:nonpure-iterator-inst`, 141  
`:nonpure-iterator`, 141  
`:nonpure-promise`, 119  
`:nonpure-stream`, 123  
`:object-hash-table`, 308, 323  
`:promise`, 119  
`:singleton`, 303  
`:stream`, 123  
`:string-hash-table`, 308, 324  
`:symbol-hash-table`, 308, 324  
`:unary-predicate`, 103  
`<=`, 102, 199, 220  
`<complex>`, 221  
`<condition>`, 7  
`<input-port>`, 157  
`<list>`, 25  
`<nonempty-list>`, 25  
`<number>`, 257  
`<output-port>`, 157  
`<pair>`, 25  
`<rational-number>`, 175  
`<rational>`, 175  
`<raw-hash-table>`, 307  
`<real-number>`, 201  
`<rte-exception-info>`, 7  
`<rte-exception-kind>`, 7  
`<string-match-result>`, 56  
`<type-predicate>`, 20  
`<`, 102, 199, 220  
`=`, 15, 20, 198, 220, 255, 287  
`>=`, 102, 199, 220  
`>`, 102, 199, 220  
`%debug-print`, 13  
`abc`, 200  
`abs`, 102, 220, 255, 262  
`acosh`, 265  
`acos`, 264  
`alist-delete`, 45  
`and-map-nonpure1?`, 37  
`and-map-nonpure?`, 36  
`and-map1?`, 36  
`and-map?`, 35  
`append-tuples`, 51  
`append`, 50  
`ash`, 117  
`asinh`, 265  
`asin`, 264  
`assoc-contents1`, 320, 337  
`assoc-contents`, 44  
`assoc-general`, 41  
`assoc-objects1`, 320, 337  
`assoc-objects`, 43  
`assoc-values1`, 319, 336  
`assoc-values`, 42  
`assoc1`, 319, 335  
`assoc`, 42  
`atanh`, 265  
`atan`, 264  
`atom-to-string`, 200, 255

- bitwise-and, 115
- bitwise-arithmetic-shift-left, 118
- bitwise-arithmetic-shift-right, 117
- bitwise-arithmetic-shift, 117
- bitwise-ior, 116
- bitwise-not, 115
- bitwise-xor, 116
- boolean->string, 151
- boolean=?, 15
- boolean?, 20
- c-abs, 240
- c-acosh, 253
- c-acos, 251
- c-angle, 242
- c-asinh, 253
- c-asin, 250
- c-atanh, 253
- c-atan, 251
- c-cosh, 252
- c-cos, 249
- c-exp2, 245
- c-expt, 247
- c-exp, 248
- c-int-expt, 246
- c-log10, 249
- c-log, 248
- c-neg, 240
- c-nonneg-int-expt, 246
- c-sinh, 251
- c-sin, 249
- c-sqrt, 247
- c-square, 241
- c-tanh, 252
- c-tan, 250
- call-with-input-string, 165
- call-with-output-string, 166
- car, 26
- cbrt, 270
- cdr, 26
- ceiling, 72
- character->string, 152
- character-ready?, 157
- character=?, 15
- character?, 21
- close-input-port, 158
- close-output-port, 158
- column-vector, 277
- command-line-arguments, 14
- complex\*, 233
- complex+, 226
- complex->exact, 260
- complex-integer\*, 234
- complex-integer+, 227
- complex-integer-, 230
- complex-integer/, 237
- complex-integer=, 223
- complex-rational\*, 235
- complex-rational+, 229
- complex-rational-, 232
- complex-rational/, 239
- complex-rational=, 225
- complex-real\*, 234
- complex-real+, 228
- complex-real-expt, 244
- complex-real-, 231
- complex-real/, 238
- complex-real=, 224
- complex-to-string, 254
- complex-, 230
- complex/, 236
- complex=?, 222
- complex=, 223
- complex, 254
- console-character-ready?, 169
- console-display-character, 170
- console-display-line, 170
- console-display-string, 170
- console-display, 169
- console-newline, 171
- console-read-character, 171
- console-read, 171
- console-write-line, 172
- console-write, 172
- cons, 28
- content-alist-delete, 47
- cosh, 264
- cos, 263
- count, 54
- current-input-port, 158
- current-output-port, 159
- d-append, 297
- d-car, 297
- d-cdr, 298
- d-for-each1, 299
- d-for-each, 298
- d-list-ref, 300
- d-list, 299
- d-map-nonpure1, 302

- d-map-nonpure, 301
- d-map1, 301
- d-map, 300
- delete-file, 173
- denominator, 176
- diagonal-matrix\*, 278
- diagonal-matrix+, 279
- diagonal-matrix-copy, 280
- diagonal-matrix-matrix=, 277
- diagonal-matrix-ref, 280
- diagonal-matrix-set!, 281
- diagonal-matrix-, 279
- diagonal-matrix=, 276
- diagonal-matrix, 278
- disable-rte-exception-info, 9
- display-character, 160
- display-line, 161
- display-string, 161
- display, 166
- distinct-elements?, 52
- drop-right, 29
- drop, 29
- enable-rte-exception-info, 9
- end-iter, 133
- eof?, 21
- equal?, 20, 198, 255
- erfc, 271
- erf, 270
- exact->inexact, 261
- exact?, 259
- exit, 7
- exp2, 269
- expm1, 269
- expt, 262
- exp, 262
- factorial, 73
- fdim, 267
- file-exists?, 173
- filter, 52
- find, 53
- finite?, 73
- floor, 74
- fmax, 267
- fmin, 267
- fmod, 267
- for-each1, 32
- for-each, 31
- force-nonpure, 120
- force, 120
- fpclassify, 268
- frexp, 267
- gcd, 74
- gen-car, 27
- gen-cdr, 27
- gen-generator, 148
- gen-list-nonpure, 142
- gen-list, 134
- gen-mutable-vector-nonpure, 143
- gen-mutable-vector, 135
- general-alist-delete, 44
- general-object->string, 151
- generator->iterator, 149
- get-list-iterator, 135
- get-list-nonpure-iterator, 143
- get-mutable-vector-iterator, 136
- get-mutable-vector-nonpure-iterator, 144
- get-rte-exception-info0, 11
- get-rte-exception-info, 12
- get-rte-exception-kind0, 10
- get-rte-exception-kind, 11
- getenv, 174
- hash-clear!, 311, 331
- hash-contents, 311, 326
- hash-count-elements, 312, 334
- hash-count, 334
- hash-exists?, 312, 331
- hash-for-each, 313, 332
- hash-ref, 313, 332
- hash-remove!, 314, 333
- hash-search-first, 335
- hash-set!, 314, 333
- hashq, 310, 325
- hashv, 310, 326
- hypot, 270
- i-abs, 80
- i-cbrt, 268
- i-expt, 196
- i-log10-exact, 74
- i-log2-exact, 75
- i-neg, 82
- i-nonneg-expt, 80
- i-sign, 80
- i-sqrt, 219
- i-square, 81
- ilogb, 267
- imag-part, 241
- inexact->exact, 261
- inexact?, 259
- infinite?, 75



- inf, 81
- integer\*, 77
- integer+, 76
- integer->complex, 222
- integer->rational, 186
- integer->real, 79
- integer->string, 152
- integer-complex\*, 234
- integer-complex+, 227
- integer-complex-, 231
- integer-complex/, 237
- integer-complex=, 224
- integer-float?, 82
- integer-rational\*, 192
- integer-rational+, 189
- integer-rational-, 190
- integer-rational/, 193
- integer-rational<=, 181
- integer-rational<, 180
- integer-rational=, 178
- integer-rational>=, 184
- integer-rational>, 182
- integer-real\*, 83
- integer-real+, 82
- integer-real-, 83
- integer-real/, 84
- integer-real<=, 85
- integer-real<, 84
- integer-real=, 17
- integer-real>=, 86
- integer-real>, 85
- integer-valued?, 257
- integer-, 76
- integer<=, 79
- integer<, 77
- integer=?, 16
- integer=, 16
- integer>=, 78
- integer>, 78
- integer?, 22
- iter-every1, 137
- iter-every2, 138
- iter-map1, 136
- iter-map2, 137
- j0, 273
- j1, 273
- jn, 274
- join-strings-with-sep, 57
- last, 31
- ldexp, 267
- length, 25
- lgamma, 271
- list->stream, 126
- list, 28
- log10, 263
- log1p, 270
- log2, 270
- logb, 267
- log, 263
- make-column-vector, 281
- make-diagonal-matrix, 282
- make-hash-table-with-size, 316, 328
- make-hash-table, 315, 327
- make-input-expr-stream, 124
- make-matrix, 283
- make-nonpure-promise, 121
- make-numerical-overflow, 13
- make-object-hash-table-with-size, 317, 329
- make-object-hash-table, 316, 328
- make-polar, 242
- make-promise, 121
- make-row-vector, 283
- make-rte-exception, 9
- make-simple-exception, 12
- make-singleton, 303
- make-string-hash-table-with-size, 317, 329
- make-string-hash-table, 317, 329
- make-symbol-hash-table-with-size, 318, 330
- make-symbol-hash-table, 318, 330
- map-car, 40
- map-cdr, 40
- map-nonpure1, 34
- map-nonpure, 34
- map1, 33
- map, 33
- matrix\*, 284
- matrix+, 285
- matrix-copy, 286
- matrix-diagonal-matrix=, 276
- matrix-ref, 291, 292
- matrix-set!, 292, 293
- matrix-, 285
- matrix=, 275
- matrix, 284
- member-contents?, 49
- member-general?, 47

- member-objects?, 49
- member-values?, 48
- member?, 48
- modf, 268
- mutable-value-vector-length, 67
- mutable-value-vector-ref, 68
- mutable-value-vector-set!, 68
- mutable-vector-length, 69
- mutable-vector-ref, 69
- mutable-vector-set!, 70
- nan?, 87
- nan, 86
- neg-inf, 87
- newline, 162
- nonpure-end-iter, 142
- nonpure-iter-every1, 146
- nonpure-iter-every2, 146
- nonpure-iter-for-each1, 147
- nonpure-iter-for-each2, 147
- nonpure-iter-map1, 144
- nonpure-iter-map2, 145
- nonpure-stream->list, 128
- nonpure-stream-empty?, 127
- nonpure-stream-for-each, 131
- nonpure-stream-map, 130
- nonpure-stream-next, 127
- nonpure-stream-value, 126
- not-null?, 22
- not-object, 55
- not, 54
- null->string, 153
- null?, 22
- number-of-columns, 294
- number-of-rows, 295
- numerator, 176
- object->string, 155
- object-alist-delete, 46
- object-hash, 309, 324
- open-input-file, 162
- open-output-file, 162
- or-map-nonpure1?, 39
- or-map-nonpure?, 39
- or-map1?, 38
- or-map?, 37
- pair->string, 154
- pair?, 23
- peek-character, 163
- quotient, 87
- r-abs, 88
- r-acosh, 217
- r-acos, 215
- r-asinh, 217
- r-asin, 214
- r-atan2, 218
- r-atanh, 218
- r-atan, 215
- r-cbrt, 267
- r-ceiling, 88
- r-complex-expt, 244
- r-complex-log, 243
- r-copysign, 268
- r-cosh, 216
- r-cos, 214
- r-erfc, 267
- r-erf, 267
- r-exp2, 267
- r-expm1, 267
- r-expt, 211
- r-exp, 212
- r-floor, 89
- r-fma, 267
- r-hypot, 267
- r-int-expt, 89
- r-isnormal?, 268
- r-j0, 273
- r-j1, 273
- r-jn, 273
- r-lgamma, 267
- r-log-neg, 243
- r-log10-neg, 244
- r-log10, 213
- r-log1p, 267
- r-log2-neg, 268
- r-log2, 267
- r-log, 212
- r-nearbyint, 267
- r-neg, 90
- r-nextafter, 268
- r-nonneg-int-expt, 90
- r-remainder, 267
- r-round, 90
- r-signbit, 268
- r-sign, 91
- r-sinh, 216
- r-sin, 213
- r-sqrt, 211
- r-square, 91
- r-tanh, 217
- r-tan, 214
- r-tgamma, 267

- r-truncate, 92
- r-y0, 273
- r-y1, 273
- r-yn, 273
- raise-numerical-overflow, 14
- raise-simple, 12
- raise, 8
- rat-abs, 194
- rat-cbrt, 269
- rat-int-expt, 196
- rat-integer-valued?, 185
- rat-inverse, 195
- rat-log10-exact, 197
- rat-log2-exact, 197
- rat-neg, 194
- rat-nonneg-int-expt, 195
- rat-one?, 188
- rat-one, 187
- rat-sign, 185
- rat-sqrt, 219
- rat-square, 195
- rat-zero?, 187
- rat-zero, 186
- rational\*, 191
- rational+, 188
- rational->complex, 222
- rational->integer, 186
- rational->real, 202
- rational-complex\*, 236
- rational-complex+, 229
- rational-complex-, 233
- rational-complex/, 239
- rational-complex=, 226
- rational-integer\*, 191
- rational-integer+, 188
- rational-integer-, 190
- rational-integer/, 193
- rational-integer<=, 180
- rational-integer<, 179
- rational-integer=, 178
- rational-integer>=, 183
- rational-integer>, 182
- rational-real\*, 209
- rational-real+, 207
- rational-real-, 208
- rational-real/, 210
- rational-real<=, 205
- rational-real<, 204
- rational-real=, 203
- rational-real>=, 207
- rational-real>, 206
- rational-to-string, 198
- rational-valued?, 258
- rational-, 189
- rational/, 192
- rational<=, 180
- rational<, 179
- rational=?, 177
- rational=, 177
- rational>=, 183
- rational>, 181
- rational, 175
- raw-exit, 8
- read-all, 164
- read-character, 164
- read-line, 165
- read-string, 165
- read, 163
- real\*, 93
- real+, 92
- real->complex, 221
- real->exact, 260, 261
- real->integer, 96
- real->string, 153
- real-complex\*, 235
- real-complex+, 228
- real-complex-expt, 245
- real-complex-, 232
- real-complex/, 238
- real-complex=, 225
- real-integer\*, 97
- real-integer+, 96
- real-integer-, 97
- real-integer/, 98
- real-integer<=, 99
- real-integer<, 98
- real-integer=, 18
- real-integer>=, 100
- real-integer>, 99
- real-part, 241
- real-rational\*, 209
- real-rational+, 207
- real-rational-, 208
- real-rational/, 210
- real-rational<=, 204
- real-rational<, 203
- real-rational=, 202
- real-rational>=, 206
- real-rational>, 205
- real-valued?, 258

- real-, 93
- real/, 93
- real<=, 95
- real<, 94
- real=?, 17
- real=, 18
- real>=, 95
- real>, 94
- real?, 23
- remainder, 100
- replace-char-with-string, 56
- replace-char, 56
- reverse, 51
- rint, 267
- round, 101
- row-vector, 286
- rte-exception?, 10
- search-substring-from-end, 58
- search-substring, 57
- sign, 102, 200, 220
- simplify-rational, 184
- singleton-get-element, 304
- singleton-set-element!, 304
- sinh, 264
- sin, 263
- split-string, 58
- sqrt, 262
- square, 102, 200, 220, 255, 262
- statprof-display, 340
- statprof-reset, 340
- statprof-start, 339
- statprof-stop, 340
- stream->list, 126
- stream-empty?, 125
- stream-for-each, 129
- stream-map-nonpure, 129
- stream-map, 128
- stream-next, 125
- stream-value, 124
- string->string, 153
- string->symbol, 59
- string-append, 60
- string-char-index-right, 61
- string-char-index, 60
- string-contains-char?, 61
- string-drop-right, 62
- string-drop, 62
- string-empty?, 63
- string-exact-match?, 63
- string-hash, 309, 325
- string-last-char, 64
- string-length, 64
- string-match, 64
- string-ref, 65
- string-take-right, 66
- string-take, 65
- string=?, 19
- string?, 24
- string, 59
- substring, 66
- symbol->string, 154
- symbol=?, 19
- symbol?, 24
- take-right, 30
- take, 30
- tanh, 264
- tan, 263
- tgamma, 271
- truncate, 101
- uniform-list-ref, 51
- value-alist-delete, 45
- value-vector-length, 70
- value-vector-ref, 71
- vector-length, 71
- vector-ref, 72
- write-character, 159
- write-line, 160
- write-string, 159
- write, 166
- xor, 55
- y0, 274
- y1, 274
- yn, 274
- \$and**, 110
- \$let\***, 110
- \$letrec\***, 110
- \$letrec**, 110
- \$or**, 110
- and-object**, 107
- and**, 106
- case**, 109
- cond-object**, 107
- cond**, 106
- define-normal-goops-class**, 111
- define-param-method**, 111
- define-param-proc**, 111
- define-simple-method**, 112
- define-simple-proc**, 112
- delay-nonpure**, 120
- delay**, 119

**do**, 109  
**guard**, 113  
**identifier-syntax**, 105  
**let\*-mutable**, 108  
**let\*-volatile**, 108  
**let\***, 108  
**make**, 113  
**or-object**, 107  
**or**, 106  
**quasiquote**, 105  
**quasisyntax**, 105  
**syntax-rules**, 105  
**with-syntax**, 105