Theme-D Standard Library Reference

Tommi Höynälänmaa

March 9, 2019

Contents

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-output` 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 `statprof` provides an interface to the guile `statprof` profiler.

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: `(:a-list <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: `:a-list`
Type: `<param-logical-type>`
Number of type parameters: 2
Description: An association list

Data type name: `:nonempty-a-list`
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` t_1 `...` t_n`)`.

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 jth elements of the `lst-k`'s for each $j = 1, \ldots, m$ (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, $k = 1, \ldots, n$. Procedure `proc` takes arguments with types t_k, $k = 1, \ldots, n$.

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 jth elements of the `lst-k`'s for each $j = 1, \ldots, m$ 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, $k = 1, \ldots, n$. 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 jth elements of the `lst-k`'s for each $j = 1, \ldots, m$ where m is the minimum of the list lengths
Result type: `(:uniform-list %result-type)`

Purity of the procedure: nonpure

The semantics or `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, $k = 1, \ldots, n$. 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, $k = 1, \ldots, n$. Procedure `proc` takes arguments with types t_k,

$k = 1, \ldots, n$.

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, $k = 1, \ldots, n$. Procedure `proc` takes arguments with types t_k, $k = 1, \ldots, n$. 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, $k = 1, \ldots, n$.

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, $k = 1, \ldots, n$.

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: `(:a-list %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: `(:a-list %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: `(:a-list %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: `(:a-list %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: `(:a-list %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?`.

a-list-delete

Syntax:

```
(a-list-delete key association-list my-eq?)
```

Type parameters: `%type1, %type2`

Arguments:

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

Name: `association-list`

Type: `(:a-list %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 `association-list`
Result type: `(:a-list %type1 %type2)`

Purity of the procedure: pure

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` $t_{k,1}$ ... $t_{k,m(k)}$`)`

Description: A tuple to be merged

Result value: A tuple constructed by appending the arguments
Result type: `(:tuple` $t_{1,1}$ `...` $t_{1,m(1)}$ `...` $t_{n,1}$ `...` $t_{n,m(n)}$`)`

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

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 characrter `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 indexth 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-start`
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 `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 `int1` $\geq$ `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 `int1` $\leq$ `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 $\texttt{i} = 0$, 1 if $\texttt{i} > 0$, and -1 if $\texttt{i} < 0$
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` $\leq$ `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` $\geq$ `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 $\mathtt{r} < \mathtt{i}$
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>) $\to$ <integer> pure = integer
: (<integer> <real>) $\to$ <real> pure = integer-real
: (<real> <integer>) $\to$ <real> pure = real-integer
: (<real> <real>) $\to$ <real> pure = real

/: (<integer> <real>) $\to$ <real> pure = integer-real/
/: (<real> <integer>) $\to$ <real> pure = real-integer/
/: (<real> <real>) $\to$ <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>) $\to$ <boolean> pure = integer<
<: (<integer> <real>) $\to$ <boolean> pure = integer-real<
<: (<real> <integer>) $\to$ <boolean> pure = real-integer<
<: (<real> <real>) $\to$ <boolean> pure = real<

<=: (<integer> <integer>) $\to$ <boolean> pure = integer<=
<=: (<integer> <real>) $\to$ <boolean> pure = integer-real<=
<=: (<real> <integer>) $\to$ <boolean> pure = real-integer<=
<=: (<real> <real>) $\to$ <boolean> pure = real<=

>: (<integer> <integer>) $\to$ <boolean> pure = integer>
>: (<integer> <real>) $\to$ <boolean> pure = integer-real>
>: (<real> <integer>) $\to$ <boolean> pure = real-integer>
>: (<real> <real>) $\to$ <boolean> pure = real>

>=: (<integer> <integer>) $\to$ <boolean> pure = integer>=
>=: (<integer> <real>) $\to$ <boolean> pure = integer-real>=
>=: (<real> <integer>) $\to$ <boolean> pure = real-integer>=
>=: (<real> <real>) $\to$ <boolean> pure = real>=

-: (<integer>) $\to$ <integer> pure = i-neg
-: (<real>) $\to$ <real> pure = r-neg
abs: (<integer>) $\to$ <integer> pure = i-abs
abs: (<real>) $\to$ <real> pure = r-abs
square: (<integer>) $\to$ <integer> pure = i-square
square: (<real>) $\to$ <real> pure = r-square
sign: (<integer>) $\to$ <integer> pure = i-sign
sign: (<real>) $\to$ <integer> pure = r-sign

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 ::= $clause_1$, ..., $clause_n$
$clause_k$ ::=($condition_k$ $expr_{k,1}$, ..., $expr_{k,m(k)}$)
else-clause ::=(**else** $else\text{-}expr_1$, ..., $else\text{-}expr_p$)

Each condition must have type `<boolean>`. The type of each $clause_k$ is the type of $expr_{k,m(k)}$ (the last expression in the clause). If *else-clause* is present its type is the type of $else\text{-}expr_p$ (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 $condition_k$ is evaluated in order until some of them returns `#t`. When some $condition_k$ returns `#t` the expressions $expr_{k,1}$, ..., $expr_{k,m(k)}$ are evaluated in order. If the result type of the **cond** expression is not `<none>` the value of the last expression $expr_{k,m(k)}$ is returned as the value of the **cond** expression. If none of the conditions return `#t` and *else-clause* is present the expressions $else\text{-}expr_1$, ..., $else\text{-}expr_p$ are evaluated in order. If the result type of the **cond** expression is not `<none>` the value of the last expression $else\text{-}expr_p$ is returned as the value of the **cond** expression.

and

Syntax:

(**and** arg_1 ...arg_n)

The type of each arg_k 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_1 ...arg_n)

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-list ::=$clause_1$, ..., $clause_n$
$clause_k$ ::=`(`$condition_k$ $expr_{k,1}$, ..., $expr_{k,m(k)}$ `)`| `(`$condition_k$ =>$handler_k$ `)`
else-clause ::=`(`**else** $else\text{-}expr_1$, ..., $else\text{-}expr_p$ `)`

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*$_1$ ... *var-spec*$_n$) *let-body-expressions*)

var-spec$_k$ ::=(*var-name*$_k$ [*var-type*$_k$] *value*$_k$)
var-name$_k$ ::=identifier
let-body-expressions ::=expression ...

The **let*** form is similar to **let** except that the expressions *value*$_k$ are evaluated in order and each expression may use the variables defined before it.

let*-mutable

Syntax:

(**let*-mutable** (*var-spec*$_1$ ...*var-spec*$_n$) *let-body-expressions*)

var-spec$_k$ ::=(*var-name*$_k$ *var-type*$_k$ *value*$_k$)
var-name$_k$ ::=identifier
let-body-expressions ::=expression ...

The **let*-mutable** form is similar to **let-mutable** except that the expressions *value*$_k$ are evaluated in order and each expression may use the variables defined before it.

let*-volatile

Syntax:

(**let*-volatile** (*var-spec*$_1$ ...*var-spec*$_n$) *let-body-expressions*)

var-spec$_k$ ::=(*var-name*$_k$ *var-type*$_k$ *value*$_k$)
var-name$_k$ ::=identifier
let-body-expressions ::=expression ...

The **let*-volatile** form is similar to **let-volatile** except that the expressions *value*$_k$ are evaluated in order and each expression may use the variables defined before it.

case

Syntax:

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

clause-list ::=$clause_1$, ..., $clause_n$
$clause_k$ ::=(($key_{k,1}$ ...$key_{k,p(k)}$) $expr_{k,1}$, ..., $expr_{k,m(k)}$)
else-clause ::=(**else** $else\text{-}expr_1$, ..., $else\text{-}expr_q$)

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 $expr_{k,j}$ 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 $else\text{-}expr_j$ 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** ($var\text{-}spec_1$ ... $var\text{-}spec_n$)
    (*condition* [*result-expression*])
    $body\text{-}expression_1$ ...$body\text{-}expression_n$)

$var\text{-}spec_k$ ::=($var\text{-}name_k$ $var\text{-}type_k$ $init\text{-}value_k$ $update\text{-}expr_k$)
$var\text{-}name_k$ ::=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 $var\text{-}name_k$ are assigned new values obtained by evaluating each $update\text{-}expr_k$ 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** (($var\text{-}name_1$ $var\text{-}type_1$ $init\text{-}value_1$ $update\text{-}expr_1$)...
    ($var\text{-}name_m$ $var\text{-}type_m$ $init\text{-}value_m$ $update\text{-}expr_m$))
    (*condition* [*result-expression*])
    $body\text{-}expression_1$ ...$body\text{-}expression_n$)

is equivalent to

(**let-mutable** ((*var-name*$_1$ *var-type*$_1$ *init-value*$_1$)...
(*var-name*$_m$ *var-type*$_m$ *init-value*$_m$))
(**until** (*condition* [*result-expression*])
body-expression$_1$ ...
body-expression$_n$
(**set!** *var-name*$_1$ *update-expr*$_1$)...
(**set!** *var-name*$_m$ *update-expr*$_m$)))

$let*
$letrec
$letrec*

Syntax:

({**$let*** | **$letrec** | **$letrec*** } (*var-spec*$_1$ ... *var-spec*$_n$) *let-body-expressions*)

var-spec$_k$ ::=(*var-name*$_k$ *value*$_k$)
var-name$_k$ ::=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? immutable? 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* ($type_1$ ... $type_n$)(*argument-list result-type attribute-list*) $body\text{-}expr_1$, ..., $body\text{-}expr_n$)

method-name ::=identifier
argument-list ::=([arg_1 ...arg_n])
arg_k ::=($arg\text{-}name_k$ $arg\text{-}type_k$)
$arg\text{-}name_k$ ::=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 $arg\text{-}type_k$ 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* ($type_1$ ... $type_n$) (*argument-list result-type attribute-list*) $body\text{-}expr_1$, ..., $body\text{-}expr_n$)

procedure-name ::=identifier
$type_k$ ::=identifier
argument-list ::=([arg_1 ...arg_n])
arg_k ::=($arg\text{-}name_k$ $arg\text{-}type_k$)
$arg\text{-}name_k$ ::=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 $arg\text{-}type_k$ 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*) $body\text{-}expr_1$, ..., $body\text{-}expr_n$)

method-name ::=identifier
argument-list ::=([arg_1 ...arg_n])
arg_k ::=($arg\text{-}name_k$ $arg\text{-}type_k$)
$arg\text{-}name_k$ ::=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 $arg\text{-}type_k$ 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*$_1$, ..., *body-expr*$_n$ `)`

procedure-name ::=identifier
argument-list ::=`(`[*arg*$_1$ ...*arg*$_n$] `)`
arg$_k$ ::=(*arg-name*$_k$ *arg-type*$_k$)
arg-name$_k$ ::=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*$_k$ 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*$_1$ ...*clause*$_n$ [*else-clause*] `)`
body-expr$_1$ ...*body-expr*$_n$ `)`
clause$_k$ ::=`(`*condition*$_k$ *expr*$_{k,1}$, ..., *expr*$_{k,m(k)}$ `)`
else-clause ::=`(`**else** *else-expr*$_1$, ..., *else-expr*$_p$ `)`

The syntax of *clause*$_k$ 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*$_k$ 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* $\textit{field-value}_1$ ...$\textit{field-value}_n$)

Keyword **make** creates an instance of *class* calling the constructor of *class* and passing the arguments $\textit{field-value}_k$. 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` in 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` in 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` in 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` in 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-output)

This module contains procedures to compute string output for different objects.

11.1 Simple Procedures

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

boolean-to-string

Syntax:

```
(boolean-to-string obj repr?)
```

Arguments:

Name: `obj`
Type: `<boolean>`
Description: A boolean value

Name: `repr?`
Type: `<boolean>`
Description: `#t` to give the source code representation (no effect in this procedure)

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

character-to-string

Syntax:

```
(character-to-string obj repr?)
```

Arguments:

Name: `obj`
Type: `<character>`

Description: A character value

Name: `repr?`
Type: `<boolean>`
Description: `#t` to give the source code representation

Result value: If `repr?` is `#t` return a string of the form #\c else return a string consisting of the given character
Result type: `<string>`

Purity of the procedure: pure

general-atom-to-string

Syntax:

```
(general-atom-to-string obj repr?)
```

Arguments:

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

Name: `repr?`
Type: `<boolean>`
Description: `#t` to give the source code representation (no effect in this procedure)

Result value: The name of the class of `obj` in brackets
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

integer-to-string

Syntax:

```
(integer-to-string obj repr?)
```

Arguments:

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

Name: `repr?`
Type: `<boolean>`
Description: `#t` to give the source code representation (no effect in this procedure)

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

null-to-string

Syntax:

```
(null-to-string obj repr?)
```

Arguments:

Name: `obj`
Type: `<null>`
Description: `null`

Name: `repr?`
Type: `<boolean>`
Description: `#t` to give the source code representation (no effect in this procedure)

Result value: `"null"`
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

real-to-string

Syntax:

```
(real-to-string obj repr?)
```

Arguments:

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

Name: `repr?`
Type: `<boolean>`
Description: `#t` to give the source code representation (no effect in this procedure)

Result value: Output string for the given value
Result type: `<string>`

Purity of the procedure: pure

string-to-display

Syntax:

```
(string-to-display obj)
```

Arguments:

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

Result value: Output string for the given object (as in Scheme `display`)
Result type: `<string>`

Purity of the procedure: pure

string-to-string

Syntax:

```
(string-to-string obj repr?)
```

Arguments:

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

Name: `repr?`
Type: `<boolean>`
Description: `#t` to give the source code representation

Result value: If `repr?` is `#t` return the string enclosed in double quotes else return the argument string without modifications
Result type: `<string>`

Purity of the procedure: pure

string-to-write

Syntax:

```
(string-to-display obj)
```

Arguments:

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

Result value: Output string for the given object in source code representation (as in Scheme `write`)
Result type: `<string>`

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

symbol-to-string

Syntax:

```
(symbol-to-string obj repr?)
```

Arguments:

Name: `obj`
Type: `<symbol>`
Description: A symbol

Name: `repr?`
Type: `<boolean>`
Description: `#t` to give the source code representation (no effect in this procedure)

Result value: Output string for the given value
Result type: `<string>`

Purity of the procedure: pure

to-string

Syntax:

```
(to-string obj repr?)
```

Arguments:

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

Name: `repr?`

Type: `<boolean>`
Description: `#t` to give the source code representation

Result value: Output string for the given object
Result type: `<string>`

Purity of the procedure: pure

11.2 Methods

`atom-to-string`: `(<object> <boolean>)` → `<string>` pure = `general-atom-to-string`
`atom-to-string`: `(<integer> <boolean>)` → `<string>` pure = `integer-to-string`
`atom-to-string`: `(<real> <boolean>)` → `<string>` pure = `real-to-string`
`string-to-string`: `(<string> <boolean>)` → `<string>` pure = `string-to-string`
`atom-to-string`: `(<symbol> <boolean>)` → `<string>` pure = `symbol-to-string`
`atom-to-string`: `(<null> <boolean>)` → `<string>` pure = `null-to-string`
`atom-to-string`: `(<character> <boolean>)` → `<string>` pure = `character-to-string`
`atom-to-string`: `(<boolean> <boolean>)` → `<string>` pure = `boolean-to-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>`

display

Syntax:

```
(display 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 procedure `atom-to-string` to obtain the string representation of the object and displays the string with procedure `display-string`.

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-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 procedure `atom-to-string` to obtain the string representation of the object and displays the string with procedure `display-string`. A newline is displayed after the object.

`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.

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 procedures opens an input file. If the operations 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 procedures opens an output file. If the operations 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 s 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.

write

Syntax:

```
(write 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 procedure `atom-to-string` to obtain the source code representation of the object and displays the string with procedure `display-string`.

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 procedure `atom-to-string` to obtain the source code representation of the object and displays the string with procedure `display-string`. A newline is written after the object.

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>) $\rightarrow$ <boolean> pure = rational=
=: (<rational> <integer>) $\rightarrow$ <boolean> pure = rational-integer=
=: (<integer> <rational>) $\rightarrow$ <boolean> pure = integer-rational=

<: (<rational-number> <rational-number>) $\rightarrow$ <boolean> pure abstract
<: (<rational> <rational>) $\rightarrow$ <boolean> pure = rational<
<: (<rational> <integer>) $\rightarrow$ <boolean> pure = rational-integer<
<: (<integer> <rational>) $\rightarrow$ <boolean> pure = integer-rational<

<=: (<rational-number> <rational-number>) $\rightarrow$ <boolean> pure abstract
<=: (<rational> <rational>) $\rightarrow$ <boolean> pure = rational<=
<=: (<rational> <integer>) $\rightarrow$ <boolean> pure = rational-integer<=
<=: (<integer> <rational>) $\rightarrow$ <boolean> pure = integer-rational<=

>: (<rational-number> <rational-number>) $\rightarrow$ <boolean> pure abstract
>: (<rational> <rational>) $\rightarrow$ <boolean> pure = rational>
>: (<rational> <integer>) $\rightarrow$ <boolean> pure = rational-integer>
>: (<integer> <rational>) $\rightarrow$ <boolean> pure = integer-rational>

>=: (<rational-number> <rational-number>) $\rightarrow$ <boolean> pure abstract
>=: (<rational> <rational>) $\rightarrow$ <boolean> pure = rational>=
>=: (<rational> <integer>) $\rightarrow$ <boolean> pure = rational-integer>=
>=: (<integer> <rational>) $\rightarrow$ <boolean> pure = integer-rational>=

+: (<rational-number> <rational-number>) $\rightarrow$ <rational-number> pure abstract
+: (<rational> <rational>) $\rightarrow$ <rational> pure = rational+
+: (<rational> <integer>) $\rightarrow$ <rational> pure = rational-integer+
+: (<integer> <rational>) $\rightarrow$ <rational> pure = integer-rational+

-: (<rational-number> <rational-number>) $\rightarrow$ <rational-number> pure abstract
-: (<rational> <rational>) $\rightarrow$ <rational> pure = rational-
-: (<rational> <integer>) $\rightarrow$ <rational> pure = rational-integer-
-: (<integer> <rational>) $\rightarrow$ <rational> pure = integer-rational-

*: (<rational-number> <rational-number>) $\rightarrow$ <rational-number> pure abstract
: (<rational> <rational>) $\rightarrow$ <rational> pure = rational
: (<rational> <integer>) $\rightarrow$ <rational> pure = rational-integer
: (<integer> <rational>) $\rightarrow$ <rational> pure = integer-rational

/: (<rational-number> <rational-number>) $\rightarrow$ <rational-number> pure abstract
/: (<rational> <rational>) $\rightarrow$ <rational> pure = rational/
/: (<rational> <integer>) $\rightarrow$ <rational> pure = rational-integer/
/: (<integer> <rational>) $\rightarrow$ <rational> pure = integer-rational/

-: (<rational-number>) $\rightarrow$ <rational-number> pure abstract
-: (<rational>) $\rightarrow$ <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 π
- 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:

```
(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:

```
(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:

```
(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 $\ln r = \ln |r| + \mathrm{i}\pi$.

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 $\log_{10} r = \log_{10}|r| + \mathrm{i}\pi/\ln 10$.

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:

```
(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:

```
(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:

```
(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 exact, 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>) $\to$ <number> pure abstract
log: (<integer>) $\to$ (:union <integer> <real> <complex>) pure
log: (<rational>) $\to$ (:union <integer> <real> <complex>) pure
log: (<real>) $\to$ (:union <real> <complex>) pure
log: (<complex>) $\to$ <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>) $\to$ <number> pure abstract
log10: (<integer>) $\to$ (:union <integer> <real> <complex>) pure
log10: (<rational>) $\to$ (:union <integer> <real> <complex>) pure
log10: (<real>) $\to$ (:union <real> <complex>) pure
log10: (<complex>) $\to$ <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>) $\to$ <number> pure abstract
sin: (<real-number>) $\to$ (:union <real> <integer>) pure abstract
sin: (<integer>) $\to$ (:union <real> <integer>) pure
sin: (<rational>) $\to$ (:union <real> <integer>) pure
sin: (<real>) $\to$ <real> pure = r-sin
sin: (<complex>) $\to$ <complex> pure = c-sin

These methods compute the sine of the argument. They return exact 0 when the argument is exact 0.

cos: (<number>) $\to$ <number> pure abstract
cos: (<real-number>) $\to$ (:union <real> <integer>) pure abstract
cos: (<integer>) $\to$ (:union <real> <integer>) pure
cos: (<rational>) $\to$ (:union <real> <integer>) pure
cos: (<real>) $\to$ <real> pure = r-cos
cos: (<complex>) $\to$ <complex> pure = c-cos

These methods compute the cosine of the argument. They return exact 1 when the argument is exact 0.

tan: (<number>) $\to$ <number> pure abstract
tan: (<real-number>) $\to$ (:union <real> <integer>) pure abstract
tan: (<integer>) $\to$ (:union <real> <integer>) pure
tan: (<rational>) $\to$ (:union <real> <integer>) pure
tan: (<real>) $\to$ <real> pure = r-tan
tan: (<complex>) $\to$ <complex> pure = c-tan

These methods compute the tangent of the argument. They return exact 0 when the argument is exact 0.

asin: (<number>) $\to$ <number> pure abstract
asin: (<integer>) $\to$ (:union <integer> <real> <complex>) pure
asin: (<rational>) $\to$ (:union <integer> <real> <complex>) pure
asin: (<real>) $\to$ (:union <real> <complex>) pure
asin: (<complex>) $\to$ <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>) $\to$ <number> pure abstract
acos: (<integer>) $\to$ (:union <integer> <real> <complex>) pure
acos: (<rational>) $\to$ (:union <integer> <real> <complex>) pure
acos: (<real>) $\to$ (:union <real> <complex>) pure
acos: (<complex>) $\to$ <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>) $\to$ <number> pure abstract
atan: (<real-number>) $\to$ <real-number> pure abstract
atan: (<integer>) $\to$ (:union <real> <integer>) pure
atan: (<rational>) $\to$ (:union <real> <integer>) pure
atan: (<real>) $\to$ <real> pure = r-atan
atan: (<complex>) $\to$ <complex> pure = c-atan

These methods compute the arctangent of the argument. They return exact 0 when the argument is exact 0.

sinh: (<number>) $\to$ <number> pure abstract
sinh: (<real-number>) $\to$ (:union <real> <integer>) pure abstract
sinh: (<integer>) $\to$ (:union <real> <integer>) pure
sinh: (<rational>) $\to$ (:union <real> <integer>) pure
sinh: (<real>) $\to$ <real> pure = r-sinh
sinh: (<complex>) $\to$ <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>) $\to$ <number> pure abstract
cosh: (<real-number>) $\to$ (:union <real> <integer>) pure abstract
cosh: (<integer>) $\to$ (:union <real> <integer>) pure
cosh: (<rational>) $\to$ (:union <real> <integer>) pure
cosh: (<real>) $\to$ <real> pure = r-cosh
cosh: (<complex>) $\to$ <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>) $\to$ <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.

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

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 $\log_2 r = \log_2 |r| + \mathrm{i}\pi / \ln 2$.

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 $\ln(1 + x)$.

cbrt: (<number>) → <number> pure abstract
cbrt: (<real-number>) → <real-number> pure abstract
cbrt: (<complex>) → <complex> pure
cbrt: (<real>) → <real> pure = r-cbrt
cbrt: (<rational>) → (:union <real> <rational> <integer>) pure
cbrt: (<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>)` $\to$ `(:union <real> <integer>)` pure abstract
`erfc`: `(<real>)` $\to$ `<real>` pure = `r-erfc`
`erfc`: `(<rational>)` $\to$ `(:union <real> <integer>)` pure
`erfc`: `(<integer>)` $\to$ `(:union <real> <integer>)` pure

These methods compute $\operatorname{erfc} x = 1 - \operatorname{erf} x$ (the complementary error function).

`lgamma`: `(<real-number>)` $\to$ `(:union <real> <integer>)` pure abstract
`lgamma`: `(<real>)` $\to$ `<real>` pure = `r-lgamma`
`lgamma`: `(<rational>)` $\to$ `(:union <real> <integer>)` pure
`lgamma`: `(<integer>>)` $\to$ `(:union <real> <integer>)` pure

These methods compute $\ln|\Gamma(x)|$.

`tgamma`: `(<real-number>)` $\to$ `(:union <real> <integer>)` pure abstract
`tgamma`: `(<real>)` $\to$ `<real>` pure = `r-tgamma`
`tgamma`: `(<rational>)` $\to$ `(:union <real> <integer>)` pure
`tgamma`: `(<integer>>)` $\to$ `(: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.

See e.g.

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>) $\rightarrow$ <real> pure
r-j1: (<real>) $\rightarrow$ <real> pure
r-jn: (<integer> <real>) $\rightarrow$ <real> pure

r-y0: (<real>) $\rightarrow$ <real> pure
r-y1: (<real>) $\rightarrow$ <real> pure
r-yn: (<integer> <real>) $\rightarrow$ <real> pure

20.2 Methods

j0: (<real-number>) $\rightarrow$ <real> pure abstract
j0: (<real>) $\rightarrow$ <real> pure = r-j0
j0: (<rational>) $\rightarrow$ <real> pure
j0: (<integer>) $\rightarrow$ <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 (standard1d-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)

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 hash tables or string 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 is compatible with the association procedure `object-assoc` with the following:

- symbols
- booleans
- characters
- strings
- user defined nonprimitive classes
- pairs
- vectors (all four kinds of vectors)

The equivalence predicate used by `object-assoc` is equivalent to `equal-objects?` for these classes. 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.

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

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-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 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: 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 size)
```

Type parameters: %value

Arguments:

Name: 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 size)
```

Type parameters: `%value`

Arguments:

Name: `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

object-assoc

Syntax:

```
(object-assoc object a-list)
```

Type parameters: `%value`

Arguments:

Name: `object`
Type: `<object>`
Description: The object to be searched

Name: `a-list`
Type: `(:a-list <object> %value)`
Description: The association list from which the object is searched

Result value: The association or `null` if none is found.
Result type: `(:maybe (:pair <object> %value))`

Purity of the procedure: pure

string-assoc

Syntax:

```
(string-assoc str a-list)
```

Type parameters: `%value`

Arguments:

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

Name: `a-list`
Type: `(:a-list <object> %value)`
Description: The association list from which the string is searched

Result value: The association or `null` if none is found.
Result type: `(:maybe (:pair <string> %value))`

Purity of the procedure: pure

Chapter 25

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.

25.1 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^7 Report on the Algorithmic Language Scheme. 2017. http://www.r7rs.org/.

[3] M. S. et al. Revised^6 Report on the Algorithmic Language Scheme. 2007. http://www.r6rs.org/.

Index