# Theme-D Standard Library Reference

Tommi Höynälänmaa

March 7, 2018

# 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 `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 `math` implements scientific functions. Basic numerical operations are defined in `core`.
- Module `complex` implements complex numbers.
- 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.

# Chapter 2

# Module (standard-library core)

## 2.1 Control Structures

### 2.1.1 Data Types

*Data type name:* `<scheme-condition>`
*Type:* `<class>`
*Description:* The data type of Scheme exceptions

### 2.1.2 Simple Procedures

#### 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** forms, see section 3.1. The semantics of **guard** and `raise` are similar to their

semantics in Scheme.

### 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.

## 2.2 Command Line

### 2.2.1 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

## 2.3 Equality Predicates

### 2.3.1 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

## 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

## 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

## 2.4 Class Membership Predicates

### 2.4.1 Data Types

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

### 2.4.2 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

## 2.5 Lists, Tuples, and Pairs

### 2.5.1 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

### 2.5.2 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

### 2.5.3 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 $j$th 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 $j$th 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 $j$th 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-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-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

## 2.6 Logical Operations

### 2.6.1 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

## 2.7 Strings

### 2.7.1 Data Types

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

*Type:* `:union`

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

### 2.7.2 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.

## 2.8 Vectors

### 2.8.1 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

## 2.9 Arithmetic Operations

### 2.9.1 Simple Procedures

#### `ceiling`

*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 infinity.

## 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.

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

*Purity of the procedure:* pure

Note that this procedure always returns an integer.

## 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

## integer-abs

*Syntax:*

```
(integer-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

## integer-neg

*Syntax:*

```
(integer-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-square

*Syntax:*

```
(integer-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

## 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-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-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

## 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.

## 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.

## 2.9.2 Methods

#### +

*Syntax:*

```
(+ nr1 nr2)
```

*Arguments:*

Name: `nr1`
Type: `<integer>` or `<real>`
Description: A number

Name: `nr2`
Type: `<integer>` or `<real>`
Description: A number

*Result value*: Sum of the arguments
*Result type:* `<integer>` or `<real>`

*Purity of the procedure:* pure

All combinations of argument types `<integer>` and `<real>` are supported.

## -

*Syntax:*

```
(- nr)
```

*Arguments:*

Name: `nr`
Type: `<integer>` or `<real>`
Description: A number

*Result value*: The opposite number of the argument
*Result type:* `<integer>` or `<real>`

*Purity of the procedure:* pure

## -

*Syntax:*

```
(- nr1 nr2)
```

*Arguments:*

Name: `nr1`
Type: `<integer>` or `<real>`
Description: A number

Name: `nr2`
Type: `<integer>` or `<real>`
Description: A number

*Result value*: Difference of the arguments
*Result type:* `<integer>` or `<real>`

*Purity of the procedure:* pure

All combinations of argument types `<integer>` and `<real>` are supported.

### *

*Syntax:*

```
(* nr1 nr2)
```

*Arguments:*

Name: `nr1`
Type: `<integer>` or `<real>`
Description: A number

Name: `nr2`
Type: `<integer>` or `<real>`
Description: A number

*Result value*: Product of the arguments
*Result type:* `<integer>` or `<real>`

*Purity of the procedure:* pure

All combinations of argument types `<integer>` and `<real>` are supported.

### /

*Syntax:*

```
(/ nr1 nr2)
```

*Arguments:*

Name: `nr1`
Type: `<integer>` or `<real>`
Description: A number

Name: `nr2`
Type: `<integer>` or `<real>`
Description: A number

*Result value*: Quotient of the arguments
*Result type:* `<integer>` or `<real>`

*Purity of the procedure:* pure

All combinations of argument types `<integer>` and `<real>` are supported.

## <

*Syntax:*

```
(< nr1 nr2)
```

*Arguments:*

Name: `nr1`
Type: `<integer>` or `<real>`
Description: A number

Name: `nr2`
Type: `<integer>` or `<real>`
Description: A number

*Result value*: `#t` if `nr1 < nr2`
*Result type:* `<boolean>`

*Purity of the procedure:* pure

All combinations of argument types `<integer>` and `<real>` are supported.

## <=

*Syntax:*

```
(<= nr1 nr2)
```

*Arguments:*

Name: `nr1`
Type: `<integer>` or `<real>`
Description: A number

Name: `nr2`
Type: `<integer>` or `<real>`
Description: A number

*Result value*: `#t` if `nr1 <= nr2`
*Result type:* `<boolean>`

*Purity of the procedure:* pure

All combinations of argument types `<integer>` and `<real>` are supported.

## >

*Syntax:*

```
(> nr1 nr2)
```

*Arguments:*

Name: `nr1`
Type: `<integer>` or `<real>`
Description: A number

Name: `nr2`
Type: `<integer>` or `<real>`
Description: A number

*Result value*: `#t` if `nr1 > nr2`
*Result type:* `<boolean>`

*Purity of the procedure:* pure

All combinations of argument types `<integer>` and `<real>` are supported.

## >=

*Syntax:*

```
(>= nr1 nr2)
```

*Arguments:*

Name: `nr1`
Type: `<integer>` or `<real>`
Description: A number

Name: `nr2`
Type: `<integer>` or `<real>`
Description: A number

*Result value*: `#t` if `nr1 >= nr2`
*Result type:* `<boolean>`

*Purity of the procedure:* pure

All combinations of argument types `<integer>` and `<real>` are supported.

## abs

*Syntax:*

```
(abs nr)
```

*Arguments:*

Name: `nr`
Type: `<integer>` or `<real>`
Description: A number

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

*Purity of the procedure:* pure

## square

*Syntax:*

```
(square nr)
```

*Arguments:*

Name: `nr`
Type: `<integer>` or `<real>`

Description: A number

*Result value*: Square of the argument
*Result type:* `<integer>` or `<real>`

*Purity of the procedure:* pure

# Chapter 3

# Module (standard-library core-forms)

## 3.1 Macros

### with-syntax

See [2].

### syntax-rules

See [2].

### identifier-syntax

See [2].

### quasiquote

See [2].

### quasisyntax

See [2].

## 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-expr*$_1$, ..., *else-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-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-expr*$_1$, ..., *else-expr*$_p$ are evaluated in order. If the result type of the **cond** expression is not `<none>` the value of the last expression *else-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-spec*$_1$ ... *var-spec*$_n$ )
(*condition* [*result-expression*] )
*body-expression*$_1$ ...*body-expression*$_n$ )

$var\text{-}spec_k$ ::=(*var-name*$_k$ *var-type*$_k$ *init-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-name*$_1$ *var-type*$_1$ *init-value*$_1$ $update\text{-}expr_1$ )...
(*var-name*$_m$ *var-type*$_m$ *init-value*$_m$ $update\text{-}expr_m$ ))
(*condition* [*result-expression*] )
*body-expression*$_1$ ...*body-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-expr*$_1$, ..., *body-expr*$_n$ )

*method-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`

The **define-param-method** defines a parametrized method. 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.

## define-param-proc

*Syntax:*

(**define-param-proc** *procedure-name* (*type*$_1$ ... *type*$_n$ ) (*argument-list result-type attribute-list* ) *body-expr*$_1$, ..., *body-expr*$_n$ )

*procedure-name* ::=identifier
*type*$_k$ ::=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-param-proc** defines constant *procedure-name* with a parametrized 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 type of the last body expression is not a subtype of *result-type*. If *result-type* is not `<none>` the result value of the procedure is the value of the last body expression.

## define-simple-method

*Syntax:*

(**define-simple-method** *method-name* (*argument-list result-type attribute-list* ) *body-expr*$_1$, ..., *body-expr*$_n$ )

*method-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-method** defines a simple method. 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.

## 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*$_n$ `)`

The syntax of *clause*$_k$ and *else-clause* is similar to the same syntax elements in **cond** form, see section 3.1. The semantics of **guard** is similar to the semantics in Scheme.

## make

*Syntax:*

`(`**make** *class field-value*$_1$ ...*field-value*$_n$ `)`

Keyword **make** creates an instance of *class* calling the constructor of *class* and passing the arguments *field-value*$_k$. Expression *class* has to be a static type expression and its value has to be a class.

# Chapter 4

# Module (standard-library promise)

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

## 4.1 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

## 4.2 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.

## 4.3 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 5

# 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.

## 5.1 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

## 5.2 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

## 5.3 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 6

# Module (standard-library iterator)

This module implements purely functional iterators, see [1].

## 6.1 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

## 6.2 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 7

# 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.

## 7.1 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

## 7.2 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 8

# Module (standard-library object-string-output)

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

## 8.1 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

## 8.2 Methods

This module defines the generic procedure `atom-to-string`. Procedures `xxx-to-string` defined in this module are added into `atom-to-string`.

# Chapter 9

# Module (standard-library text-file-io)

## 9.1 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)

## 9.2 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>`

## 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.

## 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.

## 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.

## 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>`

## 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>`

## 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>`

The Theme-D runtime environment checks that the result object does not contain any data types unknown to Theme-D.

## 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>)`

## 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 10

# Module (standard-library console-io)

This module implements input and output for the standard input and standard output.

## 10.1 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 11

# Module (standard-library system)

## 11.1 Simple Procedures

### file-exists?

*Syntax:*

```
(file-exists?  str-filename)
```

*Arguments:*

Name: `str-filename`
Type: `<string>`
Description: The name of the file

*Result value*: `#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 12

# Module (standard-library math)

## 12.1 Simple Procedures

### atan2

*Syntax:*

```
(atan2 y x)
```

*Arguments:*

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

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

*Result value*: The arctangent of `y` / `x`
*Result type:* `<real>`

*Purity of the procedure:* pure

### 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

## 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

## 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

## 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-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-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

## 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-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-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-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

## 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

## 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

## 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-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-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

### 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-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

## 12.2 Methods

### acos

*Syntax:*

```
(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

## acosh

*Syntax:*

```
(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

## asin

*Syntax:*

```
(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

## asinh

*Syntax:*

```
(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

## atan

*Syntax:*

```
(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

## atanh

*Syntax:*

```
(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

### cos

*Syntax:*

```
(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

### cosh

*Syntax:*

```
(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

## exp

*Syntax:*

```
(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).

## expt

*Syntax:*

```
(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

## log

*Syntax:*

```
(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

## log10

*Syntax:*

```
(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

## sin

*Syntax:*

```
(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

## sinh

*Syntax:*

```
(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

## sqrt

*Syntax:*

```
(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

## tan

*Syntax:*

```
(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

## tanh

*Syntax:*

```
(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

# Chapter 13

# Module (standard-library complex)

## 13.1 Data Types

*Data type name:* `<complex>`
*Type:* `<class>`
*Description:* A complex number
Class `<complex>` is immutable, equal by value, and not inheritable.

## 13.2 Simple Procedures

### 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-acos

*Syntax:*

```
(c-acos c)
```

*Arguments:*

Name: `c`
Type: `<complex>`
Description: A complex number

*Result value*: The arccosine 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-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

## 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

## 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-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-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

## 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-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-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-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-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-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-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-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-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-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

## 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-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

## complex+

*Syntax:*

```
(complex+ c1 c2)
```

*Arguments:*

Name: `c1`
Type: `<complex>`
Description: A complex value

Name: `c2`
Type: `<complex>`
Description: A complex value

*Result value*: The sum of the arguments
*Result type:* `<complex>`

*Purity of the procedure:* pure

## complex-

*Syntax:*

```
(complex- c1 c2)
```

*Arguments:*

Name: `c1`
Type: `<complex>`
Description: A complex value

Name: `c2`
Type: `<complex>`
Description: A complex value

*Result value*: The difference of the arguments
*Result type:* `<complex>`

*Purity of the procedure:* pure

## complex*

*Syntax:*

```
(complex* c1 c2)
```

*Arguments:*

Name: `c1`
Type: `<complex>`
Description: A complex value

Name: `c2`
Type: `<complex>`
Description: A complex value

*Result value*: The product of the arguments
*Result type:* `<complex>`

*Purity of the procedure:* pure

## complex/

*Syntax:*

```
(complex/ c1 c2)
```

*Arguments:*

Name: `c1`
Type: `<complex>`
Description: A complex value

Name: `c2`
Type: `<complex>`
Description: A complex value

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

*Purity of the procedure:* pure

## 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

## 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

## 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

## 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

## 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

### 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

## 13.3 Methods

### +

*Syntax:*

```
(+ nr1 nr2)
```

*Arguments:*

Name: `nr1`
Type: `<complex>`, `<real>`, or `<integer>`
Description: A number

Name: `nr2`
Type: `<complex>`, `<real>`, or `<integer>`
Description: A number

*Result value*: Sum of the arguments
*Result type:* `<complex>`

*Purity of the procedure:* pure

All combinations of argument types `<complex>`, `<real>`, and `<integer>` where either of the argument types is `<complex>` are supported.

### -

*Syntax:*

```
(- c)
```

*Arguments:*

Name: `c`
Type: `<complex>`
Description: A complex number

*Result value*: The opposite number of the argument
*Result type:* `<complex>`

*Purity of the procedure:* pure

### -

*Syntax:*

```
(- nr1 nr2)
```

*Arguments:*

Name: `nr1`
Type: `<complex>`, `<real>`, or `<integer>`
Description: A number

Name: `nr2`
Type: `<complex>`, `<real>`, or `<integer>`
Description: A number

*Result value*: Difference of the arguments
*Result type:* `<complex>`

*Purity of the procedure:* pure

All combinations of argument types `<complex>`, `<real>`, and `<integer>` where either of the argument types is `<complex>` are supported.

### *

*Syntax:*

```
(* nr1 nr2)
```

*Arguments:*

Name: `nr1`
Type: `<complex>`, `<real>`, or `<integer>`
Description: A number

Name: `nr2`
Type: `<complex>`, `<real>`, or `<integer>`
Description: A number

*Result value*: Product of the arguments
*Result type:* `<complex>`

*Purity of the procedure:* pure

All combinations of argument types `<complex>`, `<real>`, and `<integer>` where either of the argument types is `<complex>` are supported.

### /

*Syntax:*

```
(/ nr1 nr2)
```

*Arguments:*

Name: `nr1`
Type: `<complex>`, `<real>`, or `<integer>`
Description: A number

Name: `nr2`
Type: `<complex>`, `<real>`, or `<integer>`
Description: A number

*Result value*: Quotient of the arguments
*Result type:* `<complex>`

*Purity of the procedure:* pure

All combinations of argument types `<complex>`, `<real>`, and `<integer>` where either of the argument types is `<complex>` are supported.

## abs

*Syntax:*

```
(abs c)
```

*Arguments:*

Name: `c`
Type: `<complex>`
Description: A number

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

*Purity of the procedure:* pure

## acos

*Syntax:*

```
(acos c)
```

*Arguments:*

Name: `c`
Type: `<complex>`
Description: A complex number

*Result value*: The arccosine of the argument
*Result type:* `<complex>`

*Purity of the procedure:* pure

## acosh

*Syntax:*

```
(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

## asin

*Syntax:*

```
(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

## asinh

*Syntax:*

```
(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

## atan

*Syntax:*

```
(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

## atanh

*Syntax:*

```
(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

## cos

*Syntax:*

```
(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

## cosh

*Syntax:*

```
(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

## exp

*Syntax:*

```
(exp c)
```

*Arguments:*

Name: `c`

Type: `<complex>`
Description: A complex number

*Result value*: $e$ to the power of `c`
*Result type:* `<complex>`

*Purity of the procedure:* pure

Number $e$ is the base of natural logarithms (approx. 2.718).

## expt

*Syntax:*

```
(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

## atom-to-string

*Syntax:*

```
(atom-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

This generic procedure contains `complex-to-string` as a method.

## log

*Syntax:*

```
(log c)
```

*Arguments:*

Name: `c`
Type: `<complex>`
Description: A complex number

*Result value*: The natural logarithm of `c`
*Result type:* `<complex>`

*Purity of the procedure:* pure

## log10

*Syntax:*

```
(log10 c)
```

*Arguments:*

Name: `c`
Type: `<complex>`
Description: A complex number

*Result value*: The base 10 logarithm of `c`
*Result type:* `<complex>`

*Purity of the procedure:* pure

## sin

*Syntax:*

```
(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

## sinh

*Syntax:*

```
(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

## sqrt

*Syntax:*

```
(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

## square

*Syntax:*

```
(square c)
```

*Arguments:*

Name: `c`
Type: `<complex>`
Description: A number

*Result value*: Square of the argument
*Result type:* `<complex>`

*Purity of the procedure:* pure

## tan

*Syntax:*

```
(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

## tanh

*Syntax:*

```
(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

# Chapter 14

# Module (standard-library matrix)

## 14.1 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.

## 14.2 Parametrized Procedures

### 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

## 14.3 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*: 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 15

# Module (standard-library dynamic-list)

## 15.1 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 2.5.3. 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 2.5.3.

## 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 2.5.3.

## 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 16

# Module (standard-library singleton)

## 16.1 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.

## 16.2 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 17

# 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)
```

## 17.1 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.

## 17.2 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

## 17.3 Parametrized Procedures

### `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-ref

*Syntax:*

```
(hash-ref 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*: The value associated with the given key in the hash table. Returns `null` if the key is not found.
*Result type:* `(:maybe %value)`

*Purity of the procedure:* pure

## 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 18

# 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.

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

# Index