
libaldor User Guide and Reference Manual

Manuel Bronstein

Version 1.0.0 – March 8, 2002

Contents

1	Introduction	5
2	User Guide	6
	Arithmetic	8
	Data structures	8
	Input/Output	11
	Compatibility with C types	13
	Using GMP	14
	Profiling	14
	Exceptions	15
	Debugging	15
3	Reference Manual	17
	Arithmetic	
	AldorInteger	18
	AdditiveType	19
	ArithmeticType	24
	BinaryPowering	30
	Boolean	32
	BooleanArithmeticType	34
	Complex	36
	DoubleFloat	43
	FloatType	46
	GMPFloat	50
	GMPInteger	55
	IntegerSegment	59
	IntegerType	65
	LinearCombinationType	77
	MachineInteger	81
	OrderedArithmeticType	84
	PartiallyOrderedType	87
	PackableType	89
	PrimitiveType	92
	RandomNumberGenerator	94
	SingleFloat	102
	TotallyOrderedType	105
	Input/Output	
	BinaryReader	107
	BinaryWriter	111
	Byte	116
	Character	120
	File	125

FileException	132
FileExceptionType	133
InputType	134
OutputType	136
SerializableType	138
SyntaxException	140
SyntaxExceptionType	141
TextReader	142
TextWriter	147
WriterManipulator	152

Data Structures

Array	156
ArrayException	157
ArrayExceptionType	158
ArrayType	159
BoundedFiniteDataStructureType	166
BoundedFiniteLinearStructureType	171
CheckingArray	174
CheckingList	175
CheckingMemoryBlock	176
DataStructureType	177
DynamicDataStructureType	180
FiniteLinearStructureType	183
HashTable	187
KeyEntry	188
HashType	192
LinearStructureType	194
List	200
ListException	201
ListExceptionType	202
ListType	203
MemoryBlock	215
PackedPrimitiveArray	216
PrimitiveArray	217
PrimitiveArrayType	218
PrimitiveMemoryBlock	224
Set	226
SortedAssociationSet	231
SortedList	232
SortedSet	233
Stream	234
String	242
TableType	249

Utilities

AldorLibraryInformation	256
VersionInformationType	257
BinarySearch	261
CommandLine	263
CopyableType	267
Generator	269
GeneratorException	271
GeneratorExceptionType	272
Partial	273
Pointer	277
Timer	279

1 Introduction

What is libaldor?

`libaldor` (the standard aldor library) is a compact, general-purpose library for ALDOR programs. It provides a low-level interface between ALDOR programmers and the abstract machine, allowing its users to start programming with a minimal set of basic types and data structures already built into the language.

How do I get and install libaldor?

You can download `libaldor` by anonymous ftp from the CAFÉ server at `ftp-sop.inria.fr` in `cafe/software/aldorlib`, or from the URL:

`http://www.inria.fr/cafe/Manuel.Bronstein/aldorlib/`

After downloading the file `aldorlib.tar.gz`, issue “`gzip -dc aldorlib.tar.gz | tar -xvf -`” in order to unpack it. This will create the following directories:

- `aldorlib/doc`: this user guide,
- `aldorlib/lib`: the library,
- `aldorlib/include`: the required include files,
- `aldorlib/src`: the sources,
- `aldorlib/test`: some test files,
- `aldorlib/tutorial`: an ALDOR and `libaldor` tutorial.

Once the above file is unpacked, do the following:

- add the option `-csys=XXX` to your `ALDORARGS` environment variable, where `XXX` depends on your hardware and operating system. Common values for `XXX` are `axposf1v4` for OSF1 V4.0 on a DEC Alpha, `linux-486` for linux on a 486 or above PC, and `sun4os55g-v8` for SunOS 5.5 on a SPARC v8 machine. See the file `$ALDORROOT/include/aldor.conf` for other values;
- go to the `aldorlib/lib` directory and execute either `source makealdor-csh` or `source makealdor-bash` depending on your shell;
- if you want to build the debug version of the library, execute `source makealdord-csh` or `source makealdord-bash` depending on your shell; See the subsection on debugging for more information on using the debug library.

How do I use libaldor in my programs?

Once `libaldor` is properly built, you need to set the following environment variables before using it:

- the environment variable `ALDORLIBROOT` should be set to the main `aldorlib` directory;
- `$ALDORLIBROOT/include` should be appended to your `INCPATH` variable;
- `$ALDORLIBROOT/lib` should be appended to your `LIBPATH` variable;

To use `libaldor`, just add the line `#include "aldor"` in your ALDOR sources (if you were previously an `axllib` user, do *not* include `"axllib"` anymore). When building your final executable, add the options

```
-laldor -y$ALDORLIBROOT/lib
```

to your compiler command line, or

```
-laldord -dDEBUG -y$ALDORLIBROOT/lib
```

to link to the debug version of `libaldor`. Check the subsection on using GMP for the options required if you want to use the GMP package.

If you are running `libaldor` inside the compiler interactive loop, then type the line

```
#include "aldorinterp"
```

immediately after `#include "aldor"`, which will import various things for interactive use and make the interpreter loop print values automatically. Note that `GMPInteger` and `GMPFloat` are not available in the interactive loop. As with any ALDOR program, do not forget the `-q` option in order to optimize your programs, specially if performance is an issue.

Before using `libaldor` for the first time, please check your installation by running `make` in the `aldorlib/test` directory, followed by running `testall`.

Please report any installation problem or bugs you encounter to salli@sophia.inria.fr.

2 User Guide

This guide introduces the common types and categories provided by `libaldor`, and presupposes some familiarity with programming in ALDOR. If you are unfamiliar with ALDOR, then after installing `libaldor`, go through the tutorial `aldorlib/tutorial/tutorial.ps`, which will familiarize you both with ALDOR and `libaldor`. `libaldor` provides basic categories and types in 3 areas: arithmetic, data structures and input/output. Figure 1 shows the general category hierarchy in those areas and a few of the types provided, while Figure 2 shows the category hierarchy for data structures. Note that all the category names end with the word `Type`.

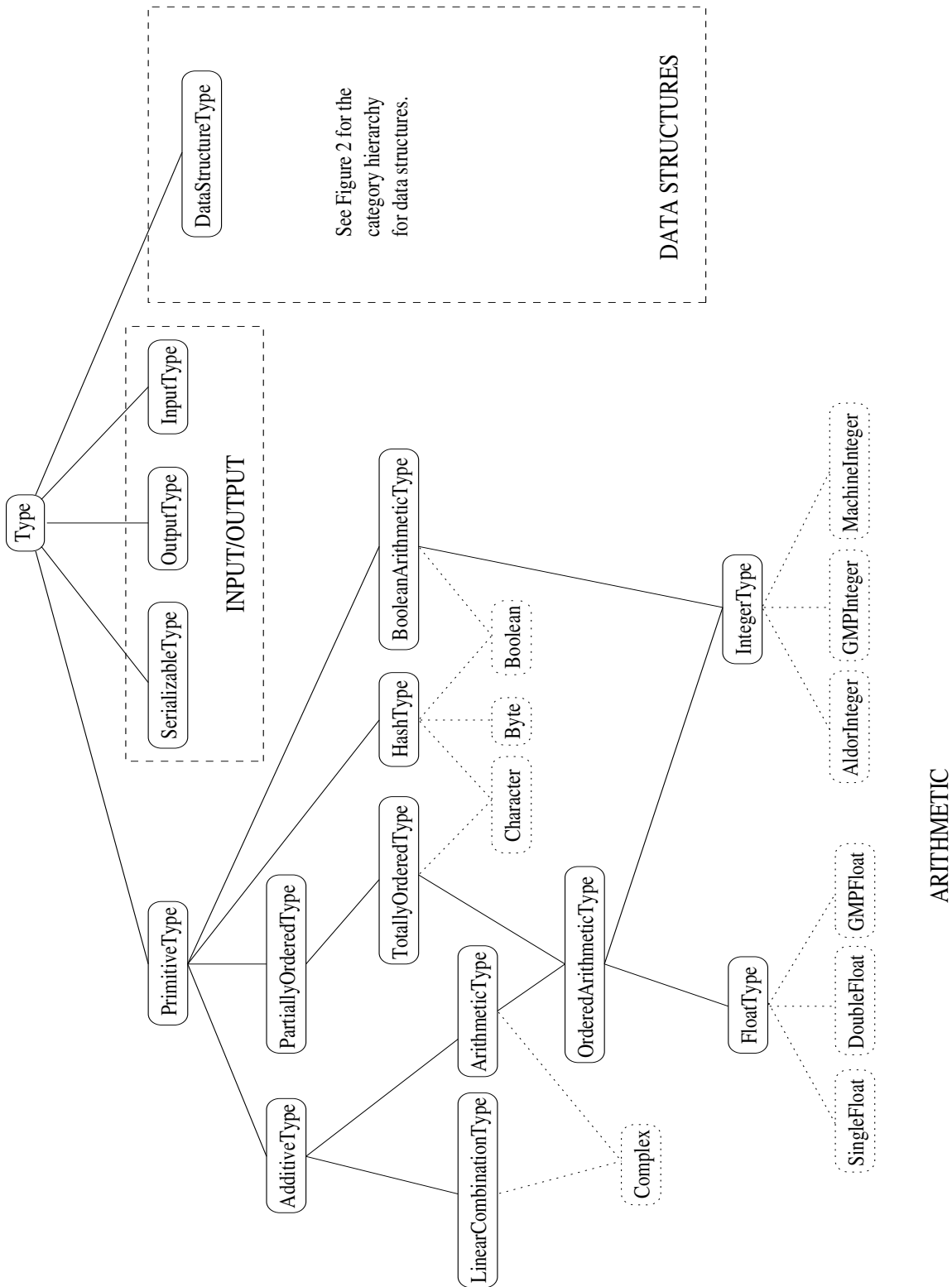


Figure 1: The libaldor general category hierarchy

Arithmetic

`libaldor` contains several integer types: `MachineInteger` provides full-word signed machine integers, while `Integer`¹ provides “infinite” precision software integers. Both of them are of category `IntegerType`, so the same functions can be applied to them. In addition, `MachineInteger` exports 3 useful constants: `min` and `max`, whose values are the smallest and greatest machine integers, and `bytes`, whose values is the machine wordsize in bytes. The `bytes` constant allows you to write wordsize-dependent code, for example when generating half-word-size prime numbers. A `MachineInteger` value `n` can be coerced to an `Integer` via `n::Integer`, and the reverse conversion is provided via the `machine` function, *i.e.* `n := machine m`. This last conversion can of course lose precision if `m` does not fit within a machine word. The general form for iterating over a range of integers is

`for variable in from..to by step — condition repeat { ... }`

where the *to* parameter and the *by* and “— *condition*” parts are optional.

Besides the integers, `libaldor` also provides machine floating point numbers, both in single precision (`SingleFloat`) and double precision (`DoubleFloat`). Finally, if you are linking with the GMP library, `libaldor` provides `GMPFloat`, which is an interface to the “infinite” precision software floats of GMP. The above 3 type share the category `FloatType`.

Data structures

`libaldor` provides both linear data structures (e.g. arrays and lists) and non-linear ones (e.g. hash tables). The category hierarchy for data structures is shown in Figure 2.

`libaldor` provides some standard linear data structures having very similar functionalities: `List T` and `CheckingList` both provide linked lists whose entries are of type `T`, while `Array T` and `CheckingArray T` both provide arrays whose entries are of type `T`. Lower-level linear data structures are `PrimitiveArray T` and `PackedPrimitiveArray T`, which provide simpler arrays whose entries are of type `T`. All of those data structures can be created by explicitly listing a finite number of elements, for example

`l:List String := ["hello", "world"]`

or by bracketing a generator, for example

`a:PrimitiveArray MachineInteger := [n for n in 1..100 | odd? n].`

In addition, the function `new` also allows structures to be created, and the constant `empty` returns an empty structure. The individual elements can be accessed via `s.n` where `s` is a data structure and `n` an index. The indexing scheme and the bound-checking scheme both depend on the structure: `List` and `CheckingList` are 1-indexed, while `Array`, `CheckingArray`, `PrimitiveArray` and `PackedPrimitiveArray` are 0-indexed. If you need to know the indexing scheme at runtime, or want to write index-independent code, the `firstIndex` constant returns the index of the first element of a structure.

The general form for iterating efficiently over a `BoundedFiniteLinearStructureType` is

`for variable in structure — condition repeat { ... }`

where “— *condition*” is optional. This is available for all high-level linear data structures, but not

¹`Integer` is actually a macro that you should use if you do not care about the specific big integer package that will be used at runtime. If integer efficiency is an issue, then read the section on `using GMP` to learn about the actual integer types available and how to select one.

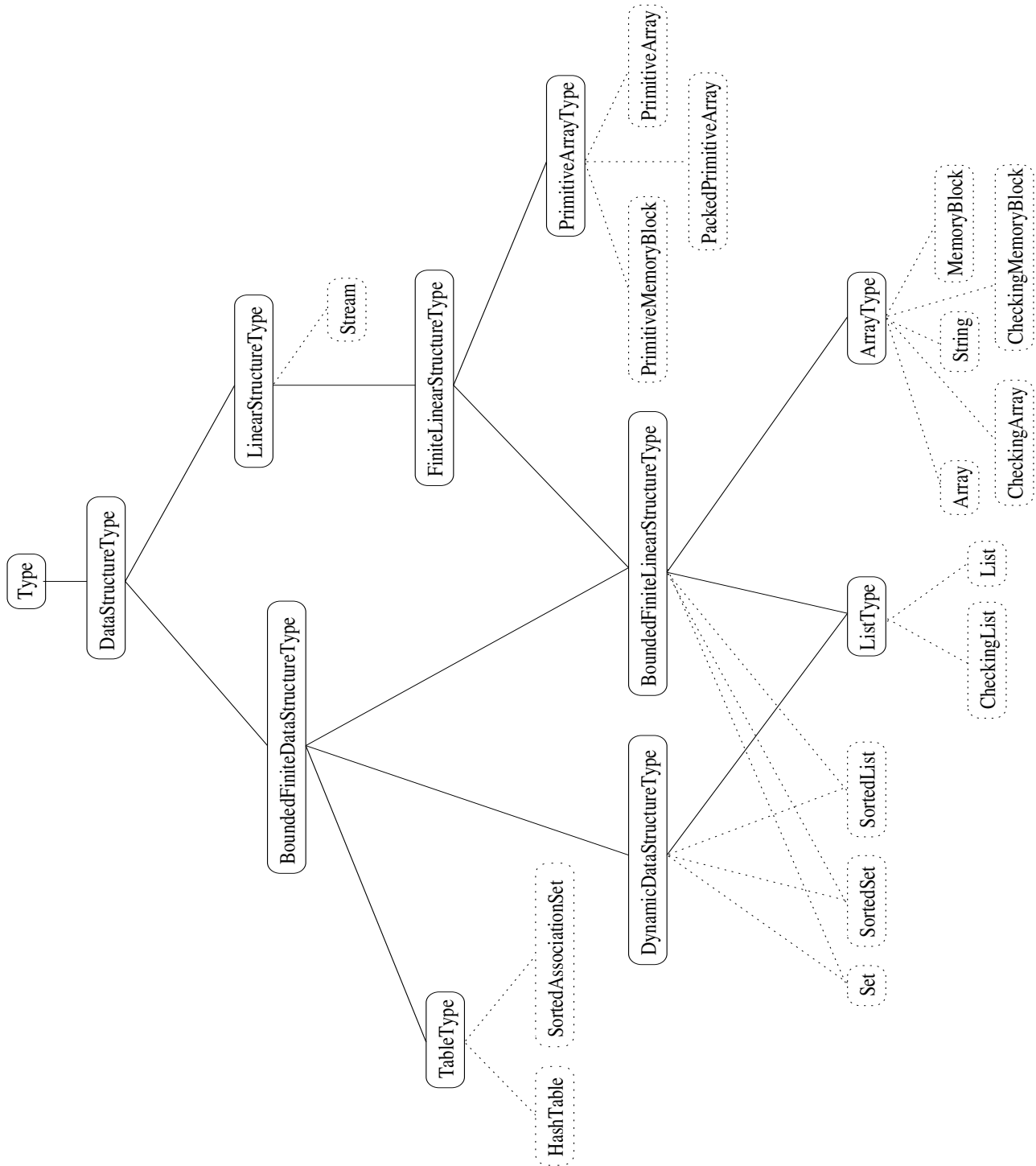


Figure 2: The libaldor category hierarchy for data structures

for those of category `PrimitiveArrayType`.

There are classical tradeoffs between the various array types: the only difference between `CheckingArray` and `Array` is that `CheckingArray` checks whether the index is within the range

of the array, while `Array` does not. Similarly, `CheckingList` checks whether a list is empty before calling `first` and `rest`, while `List` does not. Since both pairs of types offer the same exports, you can use one during the development and testing phase, and then switch to the other for releasing your code. An advantage of the lower-level arrays of category `PrimitiveArrayType` is that they are compatible with C pointers and generate more efficient code when accessing their elements. In order to benefit from the advantages of all those types, `libaldor` provides the `data` function, which returns the data of an `Array` or `CheckingArray` as a `PrimitiveArray` without copying or allocating memory. It is thus possible to use `Array` or `CheckingArray` in your code, making sure to apply `data` to it before accessing elements in a loop. For example, the following function efficiently computes the sum of all the elements with even indices of an array of machine integers:

```
evenSum(a:Array MachineInteger):MachineInteger == {
  import from MachineInteger;           -- for the index i
  import from PrimitiveArray MachineInteger; -- for accessing the elements of p
  p := data a;                          -- for efficiency (optimized code)
  sum:MachineInteger := 0;
  for i in 0..#a by 2 repeat sum := sum + p.i;
  sum;
}
```

Conversely, the function `array` creates an `Array` or `CheckingArray` from a `PrimitiveArray` without copying. Note that the generator functions in `Array` and `CheckingArray` use the `data` function, so iterating an `Array` or `CheckingArray` `a` via

```
for x in a repeat { ... }
```

is as efficient as using a `PrimitiveArray`. Note that the debug version² of `libaldor` performs bound checking on accesses into all types of arrays, including `String` and all the low-level array types.

Since `String` has the category `ArrayType Character`, all the array functionalities are also applicable to strings. For example,

```
for c in "hello" repeat { ... }
```

assigns successively the characters 'h', 'e', 'l', 'l' and 'o' to `c`, and strings can be created from `Generator Character`. Note that `String` and `PrimitiveArray Character` are not interchangeable, since the former is packed and not the latter. `String` is however interchangeable with `PackedPrimitiveArray Character`. Similarly, chunks of memory viewed as byte arrays are provided by either one of `PrimitiveMemoryBlock`, `MemoryBlock` or `CheckingMemoryBlock`. Those types are to be used for buffers rather than `PrimitiveArray Byte`, which is not interchangeable with `PrimitiveMemoryBlock`.

For types whose elements are not word-size, such as `Byte`, `Character`, `SingleFloat` and `DoubleFloat`, you can use `PackedPrimitiveArray` as a packed alternative to `PrimitiveArray`, which is then compatible with the corresponding C-arrays (see Table 1).

The type `Stream T`, of category `LinearStructureType T`, provides infinite linear structures. There are several ways to create streams, the easier ones being via an unbounded iterator, or via a function that computes its n^{th} element. For example,

```
import from MachineInteger, Stream MachineInteger;
sqr1 := [n^2 for n in 0..];
sqr2 := stream(0, (n:MachineInteger):MachineInteger +-> n^2);
```

²See the subsection on `debugging` for more information on using the debug library.

are two different ways to create the infinite stream $[0^2, 1^2, 2^2, \dots]$. Streams can be iterated, yielding loops whose duration cannot be predicted in advance, so using a parallel finite iterator or a termination condition inside the loop is advisable. Finally, streams are lazy in that $s.0, \dots, s.n$ are computed only when $s.n$ has been specifically requested, and elements are never recomputed a second time.

`libaldor` also provides different table structures: hash tables are provided by the `HashTable` type and are created via the `table` function, as in

```
t:HashTable(String, MachineInteger) := table();
```

The hash function defaults to the one provided by the type of the keys if it has `HashType`, but can be overridden by providing your own as last argument to the hash table type constructor, as in

```
t:HashTable(SingleFloat, MachineInteger, h) := table();
```

where `h` any function producing machine integers from machine floats. Providing the hash function is required when the type of the keys does not have `HashType`. Sorted tables are provided by the `SortedAssociationSet` type, those do not use hashing however.

Input/Output

`libaldor` provides an I/O model inspired from the stream model of C++. Data is written in text format on objects of type `TextWriter`, in binary format on objects of type `BinaryWriter`, read in text format from objects of type `TextReader` and read in binary format from objects of type `BinaryReader`. The standard streams `stdin`, `stdout` and `stderr` are constants exported by `TextReader` and `TextWriter` respectively, while `bin`, `bout` and `berr` are their binary counterparts. In addition, any `File` can be coerced into either text or binary readers or writers, any `String` can be coerced into text readers or writers, any `PrimitiveMemoryBlock` can be coerced into binary readers or writers, and you can create custom streams via the `textReader`, `textWriter`, `binaryReader` and `binaryWriter` functions.

The function `<<` is used for both input and output: its binary version “*writer << data*” is for output, and its unary version “*<< reader*” is for input, in which case the return type must be specified, either via an assignment to a variable, e.g. `n:MachineInteger := << stdin`, or via the `@` construct, or via the context. Whether you are reading/writing text or serializing data depends on the reader/writer type (text or binary). For example, a `libaldor` version of the “Hello world” program is

```
#include "aldor"
```

```
import from TextWriter, String, Character; -- Character needed for 'newline'
stdout << "Hello world!" << newline;
```

Text and binary writers can be flushed, either via the `flush!` function, or by sending the constant `flush`, exported by `WriterManipulator`. Thus, the two lines

```
flush!(stdout << "Hello world" << newline);
```

and

```
stdout << "Hello world" << newline << flush;
```

are equivalent. The manipulator `endl` sends first a `newline` and then flushes the stream, so another alternative for the above is

```
stdout << "Hello world" << endl;
```

When coercing strings and buffers to readers or writers, you should assign the resulting stream

to a variable if you intend to read or write more than one value from the stream, since the coercion resets the stream to the beginning of the string. For example, if `s` is the `String` “12 56”, then

```
import from MachineInteger;
a:MachineInteger := << s::TextReader;      -- assigns 12 to a
b:MachineInteger := << s::TextReader;      -- assigns again 12 to b
```

while

```
import from MachineInteger;
p := s::TextReader;
a:MachineInteger := << p;                  -- assigns 12 to a
b:MachineInteger := << p;                  -- assigns 56 to b
```

Similarly, if the file “mydata” contains “[1,2,3] [4,5,6]”, then the way to read both structures is

```
import from List MachineInteger, Array MachineInteger, File, String;
f := open("mydata", fileRead);
s := f::TextReader;
l:List MachineInteger := << s;             -- assigns [1,2,3] to l
v:Array MachineInteger := << s;            -- assigns [4,5,6] to v
close! f;
```

When coercing a `String` to a text writer or a `PrimitiveMemoryBlock` to a binary writer, you must ensure that the string or buffer is long enough to contain all the data that will be written to it, since `libaldor` does not protect you against writing past the end of the string or buffer, which is not extended by the write operation. You can either use `new` to create a large enough buffer, or write into an existing string or buffer. When writing to an existing string, you may have to send `null` (a constant exported by `Character`) to your string in order to terminate it after your data. Note that the debug version of `libaldor` performs bound checking on accesses into `String` and `PrimitiveMemoryBlock`, so it can verify whether you are writing past the end of a string or buffer. See the subsection on debugging for more information on using the debug library. Coercing a `File` to a reader or writer allocates memory, so it is advisable to assign the resulting stream to a variable. Unlike the ones for strings or buffers, those coercions do not reset the file to its beginning.

`libaldor` provides 2 categories for text input/output: `InputType` is for types whose elements can be read from a `TextReader`, and `OutputType` is for types whose elements can be written to a `TextWriter`. In addition, the single category `SerializableType` is for types whose elements can be serialized in binary format from a `BinaryReader` and to a `BinaryWriter`. All the arithmetic types provided by `libaldor`, as well as `Byte`, `Character`, `String`, `MemoryBlock` and `CheckingMemoryBlock` are `InputType`, `OutputType` and `SerializableType`, allowing you to read, write and serialize their elements. The data structures `List T`, `Array T`, `PrimitiveArray T` and `HashTable(K,V)` inherit whatever input and output capabilities that their parameters have.

Programs that perform input or output tend to repeatedly import the various stream types and accessories (manipulators, characters and strings). As an alternative to those imports, you can use `#include "aldorio"` in addition to `#include "aldor"`, which does a global import of all the following: `Character`, `String`, `File`, `TextReader`, `TextWriter`, `BinaryReader`, `BinaryWriter` and `WriterManipulator`. So an alternative “Hello world” program would be:

```
#include "aldor"          -- performs no import
#include "aldorio"         -- imports all the I/O types

stdout << "Hello world!" << endl;
```

Compatibility with C types

Several of `libaldor`'s types are compatible with their C counterparts and can be passed as arguments to C functions. Table 1 lists those types that can safely be exchanged with C functions. Since ALDOR does not provide a type that is guaranteed to be compatible with the C `int` type

libaldor	C
<code>Boolean</code>	<code>long</code>
<code>MachineInteger</code>	<code>long</code>
<code>SingleFloat</code>	<code>float</code>
<code>Character</code>	<code>char</code>
<code>String</code>	<code>char*</code>
<code>PrimitiveMemoryBlock</code>	<code>char*</code>
<code>PackedPrimitiveArray Byte</code>	<code>char*</code>
<code>PackedPrimitiveArray Character</code>	<code>char*</code>
<code>PackedPrimitiveArray SingleFloat</code>	<code>float*</code>
<code>PackedPrimitiveArray DoubleFloat</code>	<code>double*</code>
<code>Pointer</code>	<code>void*</code>
<code>PrimitiveArray</code>	<code>void**</code>
<code>PrimitiveArray MachineInteger</code>	<code>long*</code>
<code>File</code>	<code>FILE*</code>
<code>GMPFloat</code>	<code>mpf_t</code>
<code>GMPInteger</code>	<code>mpz_t</code>

Table 1: Compatibility between `libaldor` and C types

on all platforms, in order to use C functions having `int` in their parameters, you must first write a C wrapper that communicates only through the type `long`. Note that even though some type `T` can be compatible with a C type `TC`, it is not always the case that `PrimitiveArray T` is compatible with `TC*`, for example `PrimitiveArray Character` is not compatible with `char*` (nor is it compatible with `String` in `libaldor`). It is always the case however that `PackedPrimitiveArray T` is compatible with `TC*`. Note that `DoubleFloat` is not compatible with the C type `double`, but with `DFlo$Machine` instead (there are coercions between `DFlo$Machine` and `DoubleFloat`). On the other hand, `PackedPrimitiveArray DoubleFloat` is compatible with `double*`. When exchanging objects of type `PrimitiveArray`, `PrimitiveMemoryBlock` or `String` with C functions, always use the `array`, `pointer`, `string` and `pointer` functions. While those have no effect and no cost in the release version, they are necessary when linking with the debug version in order to use the bound-checking features. When doing this, you must use `Pointer` in the prototypes for the C functions rather than `PrimitiveArray`, `PrimitiveMemoryBlock` or `String`. For example, a function that uses the C-function `unlink` to remove a file could be written in the following way:

```
#include "aldor"
remove(filename:String):() == {
  import { unlink: Pointer -> MachineInteger } from Foreign C;
  unlink pointer filename;
}
```

While a function that uses the C-function `mktemp` to get a temporary file name could be written in the following way:

```
#include "aldor"

temporaryName():() == {
    import { mktemp: Pointer -> Pointer } from Foreign C;
    string mktemp pointer("/tmp/XXXXXX");
}
```

Note the use of `string` even though the buffer was allocated within `libaldor` and not within `mktemp`.

Using GMP

The type `Integer` described in this document is actually a macro, which defaults to `AldorInteger`, the software integers provided by the ALDOR runtime. For efficiency or other reasons, you may prefer to use the GMP³ library, which is supported by `libaldor`. The easiest way to use GMP is to compile all your source files with the options

```
-dGMP -cruntime=foam-gmp,gmp
```

The option `-dGMP` makes `Integer` point to the type `GMPInteger`, while the other option, which is necessary only when linking an executable, ensure that GMP makes use of the ALDOR garbage collector. All you need is GMP 3.0 or later installed in a file called `libgmp.a` to produce executables. Using GMP generally produces more efficient programs, but programs calling GMP cannot be interpreted by the ALDOR compiler, nor can they run inside its interactive loop.

Using the `-dGMP` option allows you to compile the same sources either with or without GMP, which can be appreciable, but you must ensure that you do mix files compiled with and without `-dGMP` since the macro `Integer` would then be expanded into two different types.

An additional advantage of using GMP, is that `GMPInteger` exports and uses internally several of the in-place or higher-level operations of GMP, which are not available with `AldorInteger`. In addition, variables of type `GMPInteger` are compatible with the C type `mpz_t` from GMP, so you can directly call C programs that use GMP from your ALDOR code.

Profiling

`libaldor` provides a couple of tools to help you determine how much time is spent in various sections of your programs. The simplest way is to use the `TIMESTART` and `TIME(...)` macros in your code. Those macros have absolutely no effect when compiling normally, but if you add the option `-dTIME` in the compiler command line when making the `.ao` file, then a CPU stopwatch is started when `TIMESTART` is encountered, and read at each `TIME(...)` statement. Those readings are then written to `stderr` together with whatever message appears inside the `TIME` macro. For example, if a computation has 3 major parts, then the following coding allows you to determine how much time is spent in each part:

³GMP (the GNU Multiple Precision library) is a free library for software integers, rationals and floats, available from <http://www.swox.com/gmp/>.

```

myComputation(...):... == {
    TIMESTART;                -- has no effect on normal compilation
    ... part 1 of the computation ...
    TIME("myComputation: part one done at");
    ... part 2 of the computation ...
    TIME("myComputation: part two done at");
    ... part 3 of the computation ...
    TIME("myComputation: part three done at");
    ...                      -- do not forget to return the result here
}

```

When profiling sections of a multi-file library, simply recompile the desired `.as` files with the `-dTIME -fo -q3` options, then link your executable with the local `.o` files, which takes precedence over the ones in the library.

Finer profiling, including obtaining information on the garbage collection time, is possible via the use of the type `Timer`, whose elements are stopwatches that can be started and stopped at will. See the reference section for more information on the use of timers.

Exceptions

`libaldor` can throw the following exceptions:

- `open` throws the exception `FileNotFoundException` if the file cannot be opened in the desired mode.
- `<<` from a `TextReader` throws the exception `SyntaxException` if the input does not correspond to an element of the expected type.
- Accessing a `CheckingArray` or `CheckingMemoryBlock` out of bounds throws the exception `ArrayException`.
- Accessing a `CheckingList` out of bounds throws the exception `ListException`.
- Calling `next!` on an empty generator throws the exception `GeneratorException`.

No function in `libaldor` makes a call to `error`.

Debugging

`libaldor` comes with a debug version, which makes various assertions about the arguments of the functions called. Its most important debugging feature is probably the ability to bound check all accesses into `PrimitiveArray`, `PackedPrimitiveArray`, `PrimitiveMemoryBlock`, `List` and `String` at runtime. To use the debug version of the library, just compile your application with the

```
-laldord -dDEBUG
```

options rather than `-laldor`. It is also preferable when debugging to add the `-q1` option in order to prevent inlining.

`libaldor` also provides a couple of macros to help you debug your applications:

`TRACE(message,value)`: has no effect on normal compilation, but sends *message* followed by *value* and a newline to `stderr` when compiled with the `-dTRACE` option.

`AGAT(stream,value)`: has no effect on normal compilation, but sends *value* to the Agat stream *stream* when compiled with the `-dAGAT` option. This option is only available if you have Agat installed (see <http://www-sop.inria.fr/cafe/Olivier.Arsac/agat/agat.html>).

3 Reference Manual

AldorInteger

Usage

```
import from AldorInteger
```

Description

AldorInteger provides an interface to the software (“infinite” precision) integers provided by the ALDOR virtual machine.

Exports

```
IntegerType
```

AdditiveType

Usage

AdditiveType: Category

Description

AdditiveType is the category of types with addition/substraction operations.

Exports

PrimitiveType

0 :	%	zero
+:	(%, %) → %	addition
- :	% → %	opposite
- :	(%, %) → %	substraction
add!:	(%, %) → %	In-place addition
minus!:	% → %	In-place opposite
minus!:	(%, %) → %	In-place subtraction
zero?:	% → Boolean	test for 0

Usage

0

Signature

0: %

Returns

Return the 0 constant of the type.

Usage $x + y$ $x - y$ $-x$ **Signatures** $-: \quad \% \rightarrow \%$ $+, -: \quad (\%, \%) \rightarrow \%$

Parameter	Type	Description
x, y	$\%$	elements of the type

Returns

$x + y, x - y$ return respectively the sum and difference x with y , while $-x$ returns the opposite of x .

See also

add!, minus!

Usage

```
add!(x, y)
minus!(x, y)
minus! x
```

Signatures

```
minus!::      % → %
add!, minus!:: (%, %) → %
```

Parameter	Type	Description
x, y	$\%$	Elements of the type

Returns

`add!(x, y)` and `minus!(x, y)` returns respectively $x + y$ and $x - y$, while `minus! x` returns the opposite of x . In all cases, the storage used by x is allowed to be destroyed or reused, so x is lost after this call.

Remarks

Those functions may cause x to be destroyed, so do not use them unless x has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

`+, -`

Usage

zero? x

Signature

zero?: % \rightarrow Boolean

Parameter	Type	Description
x	%	an element of the type

Returns

Returns the result of $x = 0$ using the semantics of $=$ of the type.

ArithmeticType

Usage

ArithmeticType: Category

Description

ArithmeticType is the category of types with standard arithmetic operations.

Exports

AdditiveType

1:	%	one
*	(%, %) → %	product
^:	(%, MachineInteger) → %	exponentiation
commutative?:	Boolean	check whether * is commutative
one?:	% → Boolean	test for 1
times!:	(%, %) → %	In-place product

Usage

1

Signature

1: %

Returns

Return the 1 constant of the type.

Usage

$x * y$
 $x ^ n$

Signatures

$*$: $(\%, \%) \rightarrow \%$
 $^$: $(\%, \text{MachineInteger}) \rightarrow \%$

Parameter	Type	Description
x, y	$\%$	elements of the type
n	<code>MachineInteger</code>	an exponent

Returns

$x * y$ returns the product of x with y , while $x ^ n$ returns x to the power n .

See also

`times!`

Usage

commutative?

Signature

commutative?: Boolean

Returns

Returns *true* if *** is commutative, *false* otherwise.

Usage

one? x

Signature

one?: % \rightarrow Boolean

Parameter	Type	Description
x	%	an element of the type

Returns

Returns the result of $x = 1$ using the semantics of $=$ of the type.

Usage

times!(x, y)

Signature

times!: (% , %) → %

Parameter	Type	Description
x, y	%	Elements of the type

Returns

Return xy , where the storage used by x is allowed to be destroyed or reused, so x is lost after this call.

Remarks

This function may cause x to be destroyed, so do not use it unless x has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

*

BinaryPowering

Usage

import from BinaryPowering(T , Z)

Parameter	Type	Description
T	ArithmeticType	An arithmetic system
Z	IntegerType	An integer-like type

Description

BinaryPowering provides binary exponentiation of elements of T with exponents in Z .

Exports

binaryExponentiation: $(T, Z) \rightarrow T$ Binary powering
binaryExponentiation!: $(T, Z) \rightarrow T$ In-place binary powering

Usage

binaryExponentiation(a, n)
binaryExponentiation!(a, n)

Signature

binaryExponentiation: $(T, Z) \rightarrow T$

Parameter	Type	Description
a	T	The element to exponentiate
n	Z	The exponent

Returns

Returns a^n . The exponent n must be nonnegative. When using `binaryExponentiation!(a, n)`, the storage used by `a` and `n` is allowed to be destroyed or reused, so `a` and `n` are lost after this call.

Remarks

A call to `binaryExponentiation!(a, n)` may cause `a` and `n` to be destroyed, so do not use it unless `a` and `n` have been locally allocated, and are guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Boolean

Usage

import from Boolean

Description

Boolean implements the boolean values *true* and *false*.

Exports

BooleanArithmeticType

HashType

false: % *false*

true: % *true*

Usage

true
false

Signature

true,false: %

Returns

true and false return the boolean values *true* and *false* respectively.

BooleanArithmeticType

Usage

BooleanArithmeticType: Category

Description

BooleanArithmeticType is the category of types allowing boolean arithmetic.

Exports

PrimitiveType

\sim : $\% \rightarrow \%$ negation

\wedge : $(\%, \%) \rightarrow \%$ and

\vee : $(\%, \%) \rightarrow \%$ or

xor: $(\%, \%) \rightarrow \%$ exclusive or

Usage

$\sim a$
 $a \wedge b$
 $a \vee b$
 $\text{xor}(a, b)$

Signatures

$\sim : \quad \% \rightarrow \%$
 $\wedge, \vee, \text{xor} : (\%, \%) \rightarrow \%$

Parameter	Type	Description
a, b	$\%$	elements of the type

Returns

$\sim a$ returns $\text{not}(a)$, while $a \vee b$ returns (a or b), $a \wedge b$ returns (a and b), and $\text{xor}(a, b)$ returns

$$(a \wedge \sim b) \vee (\sim a \wedge b).$$

The semantics of *not*, *or* and *and* can be logical or bitwise, depending on the actual type.

Remarks

For the type `Boolean`, the difference between $a \vee b$ and `a or b` is that $a \vee b$ guarantees that both expressions a and b are evaluated while `a or b` may evaluate only a and return *true* if a evaluates to *true*. There is a similar difference between $a \wedge b$ and `a and b`.

Example

If a and b are the `MachineInteger` 5 and 7, then $\sim a = -6$, $a \vee b = 5$, $a \wedge b = 7$ and $\text{xor}(a, b) = 2$.

Complex

Usage

import from Complex R

Parameter	Type	Description
R	ArithmeticType	Type to be extended

Description

Complex R implements the algebraic extension of R generated by a root of $X^2 + 1 = 0$. The coefficient type R must satisfy 2 additional mathematical properties for the arithmetic of Complex R to be correct, namely:

- $X^2 + 1$ is irreducible over R .
- The element α such that $\alpha^2 + 1 = 0$ must commute with all the elements of R (which is obviously satisfied when the multiplication of R is commutative).

Exports

ArithmeticType

CopyableType

LinearCombinationType R

coerce: $R \rightarrow \%$ Natural embedding

complex: $(R, R) \rightarrow \%$ create a complex

conjugate: $\% \rightarrow \%$ conjugation

conjugate!: $\% \rightarrow \%$ in-place conjugation

copy!: $(\%, R, R) \rightarrow \%$ in-place copy

imag: $\% \rightarrow R$ imaginary part

real: $\% \rightarrow R$ real part

if R has FloatType then

$/:$ $(\%, \%) \rightarrow \%$ Division

if R has InputType then

InputType

if R has OutputType then

OutputType

if R has SerializableType then

SerializableType

Usage`coerce x``x::%`**Signature**`coerce: R \rightarrow %`

Parameter	Type	Description
x	R	an element of the base type

Returns

Returns the complex $x + 0\sqrt{-1}$.

Usage`complex(x,y)`**Signature**`complex: (R, R) → %`

Parameter	Type	Description
x, y	R	real and imaginary parts

ReturnsReturns the complex $x + y\sqrt{-1}$.

Usage

conjugate z
conjugate! z

Signature

conjugate: % \rightarrow %

Parameter	Type	Description
z	%	a complex

Description

Returns $x - y\sqrt{-1}$ where $z = x + y\sqrt{-1}$. When using conjugate!, the storage used by z is allowed to be destroyed or reused, so z is lost after this call.

Usage

copy!(z,x,y)

Signature

copy!: (% , R, R) → %

Parameter	Type	Description
z	%	a complex
x, y	R	real and imaginary parts

Description

Replaces z by $x + y\sqrt{-1}$ and return z , where the storage used by z is allowed to be destroyed or reused, so z is lost after this call.

Remarks

This call may cause z to be destroyed, so do not use it unless z has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

copy!

Usage

imag,real z

Signature

imag,real: % \rightarrow R

Parameter	Type	Description
z	%	a complex

Returns

imag(z) and real(z) return respectively y and x where $z = x + y\sqrt{-1}$.

Usage

`norm z`

Signature

`norm: % → %`

Parameter	Type	Description
z	<code>%</code>	a complex

Returns

Returns $z\bar{z}$ where \bar{z} is the conjugate of z .

Remarks

When R is commutative, the imaginary part of $z\bar{z}$ is 0, and the usual complex norm can be computed via `real norm z`. The imaginary part of $z\bar{z}$ is not necessarily 0 when R is not commutative.

DoubleFloat

Usage

import from DoubleFloat

Description

DoubleFloat implements the single-precision signed machine floats.

Exports

CopyableType

FloatType

PackableType

$\hat{\cdot}$ (% , %) \rightarrow % exponentiation

max: % largest single-precision machine float

min: % smallest single-precision machine float

Usage $x \wedge y$ **Signatures** $\wedge: (\%,\%) \rightarrow \%$

Parameter	Type	Description
x,y	$\%$	floats

Returns

Returns x to the power y .

Usage

max

min

Signature

max,min: %

Returns

max and min return respectively the largest and the smallest double-precision machine floats.

FloatType

Usage

FloatType: Category

Description

FloatType is the category of types representing floats.

Exports

InputType

OrderedArithmeticType

OutputType

SerializableType

/: ($\%$, $\%$) \rightarrow $\%$ division

coerce: MachineInteger \rightarrow $\%$ conversion to a float

fraction: $\%$ \rightarrow $\%$ fractional part

truncate: $\%$ \rightarrow AldorInteger truncation

Usage x/y **Signature** $/: (\%,\%) \rightarrow \%$

Parameter	Type	Description
x,y	$\%$	floats

Returns

Returns the quotient of x by y .

Usage

`n::%`

Signature

`coerce: MachineInteger → %`

Parameter	Type	Description
<i>n</i>	MachineInteger	a machine integer

Returns

Returns `n` converted to a float.

Usage

fraction x
truncate x

Signatures

fraction: % \rightarrow %
truncate: % \rightarrow AldorInteger

Parameter	Type	Description
x	%	a float

Returns

truncate(x) returns n such that $nx \geq 0$ and $|n| \leq |x| < |n| + 1$, while fraction(x) returns $x - \text{truncate}(x)$.

GMPLFloat

Usage

import from GMPLFloat

Description

GMPLFloat implements arbitrary precision floats with the float arithmetic provided by GMP.

Exports

CopyableType

FloatType

coerce:	DoubleFloat \rightarrow %	conversion
limbs:	% \rightarrow Generator MachineInteger	iteration
new:	() \rightarrow %	initialization (for calling GMP)
precision:	% \rightarrow MachineInteger	precision
setPrecision!:	(%, MachineInteger) \rightarrow %	adjust precision

Usage`x::%`**Signature**`coerce: DoubleFloat \rightarrow %`

Parameter	Type	Description
x	DoubleFloat	a machine float

ReturnsConverts x to a GMP float.

Usage

```
precision x
setPrecision!(x, n)
```

Signatures

```
precision:      % → MachineInteger
setPrecision!:  (% , MachineInteger) → %
```

Parameter	Type	Description
x	%	a float
n	MachineInteger	a precision

Description

precision(x) returns the precision actually used for x , while setPrecision!(x , n) sets the precision for x to be at least n bits.

Remarks

Those functions are wrapper to the `mpf_get_prec` and `mpf_set_prec` GMP functions, so setPrecision! can involve a call to `realloc`.

Usage

for d in limbs x repeat { ... }

Signature

limbs: % \rightarrow Generator MachineInteger

Parameter	Type	Description
x	%	a float

Description

This function allows the individual limbs of a GMP float to be iterated independently of the machine size. This generator yields the limbs from the least to the most significant.

Remarks

The limbs of x describe $|x|$, so combine it with **sign** if you need a complete description of x .

Signature

new: $() \rightarrow \%$

Returns

Returns an initialized GMP float but with no value stored into it. Results from this function can be used only as parameters to explicit calls to `mpf_` functions.

GMPIInteger

Usage

import from GMPIInteger

Description

GMPIInteger implements arbitrary precision integers with the integer arithmetic provided by GMP.

Exports

CopyableType

IntegerType

coerce: AldorInteger \rightarrow % conversion

coerce: % \rightarrow AldorInteger conversion

limbs: % \rightarrow Generator MachineInteger iteration

new: () \rightarrow % initialization (for calling GMP)

Usage`m::%``n::AldorInteger`**Signatures**`coerce: AldorInteger → %``coerce: % → AldorInteger`

Parameter	Type	Description
m	<code>%</code>	an ALDOR integer
n	<code>AldorInteger</code>	a gmp integer

Description

Those functions convert between ALDOR and GMP integers.

Remarks

The conversion to an `AldorInteger` can be quadratic in the number of bits of the integer, which is expensive.

Usage

for d in limbs n repeat { ... }

Signature

limbs: % \rightarrow Generator MachineInteger

Parameter	Type	Description
n	%	an integer

Description

This function allows the individual limbs of a GMP integer to be iterated independently of the machine size. This generator yields the limbs from the least to the most significant.

Remarks

The limbs of n describe $|n|$, so combine it with **sign** if you need a complete description of n .

Signature

new: $() \rightarrow \%$

Returns

Returns an initialized GMP integer but with no value stored into it. Results from this function can be used only as parameters to explicit calls to `mpz_` functions.

IntegerSegment

Usage

import from IntegerSegment Z

Parameter	Type	Description
Z	IntegerType	an integer type

Description

IntegerSegment(Z) implements open and closed segments of Z , *i.e.* a selection of equally spaced integers in a range of the form $[a, b]$ or $[a, +\infty)$.

Exports

PrimitiveType

InputType

OutputType

SerializableType

...	$Z \rightarrow \%$	creation of a segment
	$(Z, Z) \rightarrow \%$	
by:	$(\%, Z) \rightarrow \%$	change the spacing
generator:	$\% \rightarrow \text{Generator } Z$	iterate over a segment
high:	$\% \rightarrow Z$	upper bound
low:	$\% \rightarrow Z$	lower bound
open?:	$\% \rightarrow \text{Boolean}$	check whether a segment is open
step:	$\% \rightarrow Z$	spacing

a..
a..b

$$\begin{array}{l} \dots \quad Z \rightarrow \% \\ \dots \quad (Z, Z) \rightarrow \% \end{array}$$

Parameter	Type	Description
a, b	Z	integers

a.. returns the open range $[a, +\infty)$ while a..b returns the closed range $[a, b]$. Every integer in the range belongs to the resulting segment.

by

Usage

s by n
step s

Signatures

by: $(\%, \mathbb{Z}) \rightarrow \%$
step: $\% \rightarrow \mathbb{Z}$

Parameter	Type	Description
s	$\%$	a segment
n	\mathbb{Z}	a step

Returns

s by n changes s to become the segment consisting of every n^{th} integer in its range, *i.e.* the integers $a, a + n, a + 2n, \dots$ that are within the range $[a, b]$ or $[a, +\infty)$ of s, while step(s) returns n such that s is the segment consisting of every n^{th} integer in its range.

Remarks

The function s by n does not create a new segment but side-effects s, whose former step is lost.

Usage

for x in s repeat { ... }
 for x in generator s repeat { ... }

Signature

generator: % \rightarrow Generator Z

Parameter	Type	Description
<i>s</i>	%	a segment

Description

This functions allows a segment to be iterated. This generator yields the integers of s in succession.

Example

The following code computes the sum of all the positive even machine integers that are smaller than *n*:

```
evenSum(n:MachineInteger):MachineInteger == {
  s := 0;
  for x in 2..prev(n) by 2 repeat s := s + x;
  s;
}
```

Usage

high s

low s

Signaturehigh,low: $\% \rightarrow \mathbb{Z}$

Parameter	Type	Description
s	$\%$	a segment

Returns

high(s) and low(s) return respectively the upper and lower bound of the range of s. The result of high(s) is undefined if s is an open segment.

Usage

open? s

Signature

open?: % \rightarrow Boolean

Parameter	Type	Description
<i>s</i>	%	a segment

Returns

Returns *true* if the range of *s* is infinite, *false* otherwise.

IntegerType

Usage

IntegerType: Category

Description

IntegerType is the category of types representing integers.

Exports

BooleanArithmeticType

HashType

InputType

OrderedArithmeticType

OutputType

SerializableType

bit?: (% , MachineInteger) → Boolean

check a bit

clear: (% , MachineInteger) → %

clear a bit

coerce: MachineInteger → %

conversion from machine integer

divide: (% , %) → (% , %)

Euclidean division

even?: % → Boolean

test whether a number is even

factorial: % → %

factorial

gcd: (% , %) → %

greatest common divisor

lcm: (% , %) → %

least common multiple

length: % → MachineInteger

number of bits

machine: % → MachineInteger

conversion to a machine integer

mod: (% , %) → %

remainder

(% , MachineInteger) → MachineInteger

next: % → %

next greater integer

nthRoot: (% , %) → (Boolean, %)

n^{th} -root

odd?: % → Boolean

test whether a number is odd

prev: % → %

next smaller integer

quo: (% , %) → %

quotient

random: () → %

random integer

MachineInteger → %

rem: (% , %) → %

remainder

set: (% , MachineInteger) → %

set a bit

shift: (% , MachineInteger) → %

shift

shift!: (% , MachineInteger) → %

in-place shift

Usage

```

bit?(a, n)
clear(a, n)
set(a, n)

```

Signatures

```

bit?:      (% , MachineInteger) → Boolean
clear, set: (% , MachineInteger) → %

```

Parameter	Type	Description
a	<code>%</code>	an integer
n	<code>MachineInteger</code>	a nonnegative machine integer

Returns

`bit?(a, n)` returns *true* if the n^{th} bit of `a` is 1, *false* if it is 0, while `clear(a, n)` and `set(a, n)` return copies of `a` where the n^{th} bit is set respectively to 0 and 1. For all 3 functions, the rightmost bit of `a` is the 0^{th} bit and so on.

Usage`n::%``machine a`**Signatures**`coerce: MachineInteger \rightarrow %``machine: % \rightarrow MachineInteger`

Parameter	Type	Description
<i>a</i>	%	an integer
<i>n</i>	MachineInteger	a machine integer

Returns

`n::%` returns `n` converted to the current type, while `machine(a)` returns the low machine word of `a` converted to a **MachineInteger**. That operation can cause a loss of precision if `a` is greater than a machine word.

Usage

divide(a, b)
 a mod n
 a mod b
 a quo b
 a rem b

Signatures

divide: $(\%, \%) \rightarrow (\%, \%)$
 mod,quo,rem: $(\%, \%) \rightarrow \%$
 mod: $(\%, \text{MachineInteger}) \rightarrow \text{MachineInteger}$

Parameter	Type	Description
a, b	$\%$	integers, $b \neq 0$
n	MachineInteger	a nonzero machine integer

Returns

$a \bmod b$ (resp. $a \bmod n$) returns m such that $0 \leq m < |b|$ (resp. $0 \leq m < |n|$) and $a \equiv m \pmod{b}$ (resp n), while $a \bmod b$ returns r such that $-|b| < r < |b|$ and $a \equiv r \pmod{b}$, $a \text{ quo } b$ returns $(a - (a \bmod b))/b$, and $\text{divide}(a, b)$ returns the pair $(a \text{ quo } b, a \bmod b)$.

Remarks

mod returns a unique remainder modulo b, but is more expensive to compute than rem, and is not guaranteed to be compatible with the result of quo. The version whose second argument is a **MachineInteger** allows for more efficient implementations.

Usage

even? a
odd? a

Signature

even?,odd?: % \rightarrow Boolean

Parameter	Type	Description
<i>a</i>	%	an integer

Returns

even?(a) and odd?(a) return *true* when a is even, respectively odd, *false* otherwise.

Usage

factorial a

Signature

factorial: % \rightarrow %

Parameter	Type	Description
a	%	a nonnegative integer

Returns

Returns $a! = \prod_{i=1}^a i$.

Usage

gcd(a, b)
lcm(a, b)

Signature

gcd,lcm: (%,%) \rightarrow %

Parameter	Type	Description
a, b	%	integers

Returns

gcd(a, b) and lcm(a, b) return respectively a greatest common divisor and a least common multiple of a and b.

Usage

length a

Signature

length: % \rightarrow MachineInteger

Parameter	Type	Description
a	%	an integer

Returns

Returns the number of binary bits of a, *i.e.* n such that `bit?(a, n - 1)` is *true* and `bit?(a, m)` is *false* for $m \geq n$.

Usage

next a

prev a

Signaturenext,prev: % \rightarrow %

Parameter	Type	Description
a	%	an integer

Returnsnext(a) and prev(a) return $a + 1$ and $a - 1$ respectively.

Usage

`nthRoot(a, b)`

Signature

`nthRoot: (%,%) → (Boolean,%)`

Parameter	Type	Description
<i>a</i>	%	an integer
<i>b</i>	%	a positive integer

Returns

Returns (found?, n) such that $a = n^b$ if found? is *true*. Otherwise, found? is *false* and

$$n^b < a < (n + 1)^b.$$

Usage

random()
random n

Signatures

random: () → %
random: MachineInteger → %

Parameter	Type	Description
n	MachineInteger	a positive size

Returns

random() returns a random integer, while random(n) returns a random integer with n limbs.

Usage

shift(*a*, *n*)
shift!(*a*, *n*)

Signature

shift: (*%*, MachineInteger) → *%*

Parameter	Type	Description
<i>a</i>	<i>%</i>	an integer
<i>n</i>	MachineInteger	a machine integer

Returns

Returns *a* shifted left *n* times if $n \geq 0$, shifted right $-n$ times if $n \leq 0$.

Remarks

shift! does not make a copy of *x*, which is therefore modified after the call. It is unsafe to use the variable *x* after the call, unless it has been assigned to the result of the call, as in `x := shift!(x, n)`.

LinearCombinationType

Usage

LinearCombinationType R:Category

Parameter	Type	Description
<i>R</i>	AdditiveType	The coefficient domain

Description

LinearCombinationType R is the category of types containing linear combinations of their elements with coefficients in R.

Exports

AdditiveType

`*`: (R, %) → % Left-multiplication by a scalar

`add!`: (% , R, %) → % In-place product and sum

`times!`: (R, %) → % In-place product by a scalar

Usage $r * p$ **Signature** $*: (R, \%) \rightarrow \%$

Parameter	Type	Description
r	R	A scalar
p	$\%$	An element of the type

ReturnsReturns the product rp .**See also**`times!`

Usage

add!(p, r, q)

Signature

add!: ($\%$, R, $\%$) \rightarrow $\%$

Parameter	Type	Description
p	$\%$	An element of the type (to be destroyed)
r	R	A scalar
q	$\%$	An element of the type

Returns

add!(p, r, q) returns $p + rq$.

Remarks

The storage used by p is allowed to be destroyed or reused, so p is lost after this call. This function may cause p to be destroyed, so do not use it unless p has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

times!

Usage

times!(r, p)

Signature

times!: (R, %) → %

Parameter	Type	Description
r	R	A scalar to be multiplied by p
p	%	An element of the type (to be destroyed)

Returns

Returns the product rp .

Remarks

The storage used by p is allowed to be destroyed or reused, so p is lost after this call. This may cause p to be destroyed, so do not use this unless p has been locally allocated, and is thus guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

*,add!

MachineInteger

Usage

```
import from MachineInteger
```

Description

MachineInteger implements the full-word signed machine integers.

Exports

CopyableType

IntegerType

bytes:	%	machine word-size
max:	%	largest machine integer
min:	%	smallest machine integer
mod_+:	(%, %, %) → %	modular addition
mod_-:	(%, %, %) → %	modular subtraction
mod_*:	(%, %, %) → %	modular multiplication
mod_/:	(%, %, %) → %	modular division
mod^:	(%, %, %) → %	modular exponentiation
modInverse:	(%, %) → %	modular inverse

Usage

bytes
max
min

Signature

bytes,max,min: %

Returns

bytes, max and min return respectively the size in bytes of a machine integer, the largest and the smallest machine integers.

Usage

mod_X(a, b, n)
 modInverse(a, n)

Parameter	Type	Description
a, b, n	%	machine integers

Signatures

mod_X: (% , % , %) \rightarrow %
 modInverse: (% , %) \rightarrow %

Returns

mod_X(a, b, n) returns $(aXb) \pmod n$ where X is one of $+, -, *, /, ^$, while modInverse(a, b) returns the inverse of a modulo n .

Remarks

Those operations require that $0 \leq a, b < n$.

OrderedArithmeticType

Usage

OrderedArithmeticType: Category

Description

OrderedArithmeticType is the category of ordered types with the standard arithmetic operations.

Exports

ArithmeticType

TotallyOrderedType

abs: $\% \rightarrow \%$ norm

sign: $\% \rightarrow \text{MachineInteger}$ sign

Usage

abs x

Signature

abs: % \rightarrow %

Parameter	Type	Description
x	%	an element of the type

Returns

Returns the norm $|x|$ of x.

Usage

sign x

Signature

sign: % \rightarrow MachineInteger

Parameter	Type	Description
x	%	an element of the type

Returns

Returns 1 if $x > 0$, 0 if $x = 0$ and -1 if $x < 0$.

PartiallyOrderedType

Usage

PartiallyOrderedType: Category

Description

PartiallyOrderedType is the category of partially ordered types. Hence, a domain of this category is endowed with a transitive binary relation $>$ such that either $x > y$ or $y > x$ does not hold for every x and y . If both do not hold then x and y are called not comparable for $>$. In particular if $x = y$ holds then x and y are not comparable for $>$.

Exports

PrimitiveType

$<:$ $(\%, \%) \rightarrow \text{Boolean}$ stricly less than
 $>:$ $(\%, \%) \rightarrow \text{Boolean}$ stricly greater than
 $\leq:$ $(\%, \%) \rightarrow \text{Boolean}$ less than or equal to
 $\geq:$ $(\%, \%) \rightarrow \text{Boolean}$ greater than or equal to

Usage

$$a < b$$

$$a > b$$

$$a \leq b$$

$$a \geq b$$
Signature

$$<,>: (\%,\%) \rightarrow \text{Boolean}$$

Parameter	Type	Description
a, b	$\%$	elements of the type

Returns

$a < b$, $a > b$, $a \leq b$, $a \geq b$ return *true* when a is respectively stricly smaller than, stricly greater than, less than or equal to, greater than or equal to b . Observe that if neither $a > b$ nor $a = b$ hold, this does not imply that $a < b$ holds, since a and b may not be comparable, except if the type has `TotallyOrderedType`.

PackableType

Usage

PackableType: Category

Description

PackableType is the category of types providing their own primitive packed arrays over themselves. You can use `PackedPrimitiveArray(T)` as a packed alternative to `PrimitiveArray(T)` whenever `T` provides such functions.

Exports

<code>getPackedArray:</code>	<code>(Pointer, MachineInteger) → %</code>	access an array element
<code>newPackedArray:</code>	<code>MachineInteger → Pointer</code>	create an array
<code>setPackedArray!:</code>	<code>(Pointer, MachineInteger, %) → ()</code>	changes an array element

Usage

getPackedArray(*a*, *n*)
setPackedArray!(*a*, *n*, *x*)

Signatures

getPackedArray: (Pointer, MachineInteger) → %
setPackedArray!: (Pointer, MachineInteger, %) → ()

Parameter	Type	Description
<i>a</i>	Pointer	an array
<i>n</i>	MachineInteger	an index
<i>x</i>	%	a value

Description

getPackedArray(*a*, *n*) returns the element at position *n* in *a*, while setPackedArray!(*a*, *n*, *x*) sets the element at position *n* in *a*. Note that both functions are 0-indexed.

Usage

newPackedArray n

Signature

newPackedArray: $\text{MachineInteger} \rightarrow \text{Pointer}$

Parameter	Type	Description
n	MachineInteger	a size

Returns

Returns an array for n elements

PrimitiveType

Usage

PrimitiveType: Category

Description

PrimitiveType is the category of the most basic types.

Exports

`=:` $(\%, \%) \rightarrow \text{Boolean}$ equality test
`~=:` $(\%, \%) \rightarrow \text{Boolean}$ inequality test

Usage

$a = b$
 $a \sim = b$

Signatures

$=:$ $(\%,\%) \rightarrow \text{Boolean}$
 $\sim =:$ $(\%,\%) \rightarrow \text{Boolean}$

Parameter	Type	Description
a, b	$\%$	elements of the type

Returns

If $a = b$ returns *true*, then a and b are guaranteed to represent the same element of the type. The behavior if $a = b$ returns *false* depends on the type, since a full equality test might not be available. At least, it is guaranteed that a and b do not share the same memory location in that case. The semantics of $a \sim = b$ is the boolean negation of $a = b$.

RandomNumberGenerator

Usage

import from RandomNumberGenerator

Description

RandomNumberGenerator provides independent pseudo-random number generators.

Exports

apply:	$\% \rightarrow Z$	generate a random number
generator:	$\% \rightarrow \text{Generator } Z$	generate random numbers
max:	$\% \rightarrow Z$	largest number that can be generated
min:	$\% \rightarrow Z$	smallest number that can be generated
numberOfGenerators:	Z	number of predefined generators
randomGenerator:	$() \rightarrow \%$	get a generator
randomGenerator:	$Z \rightarrow \%$	get a generator
randomGenerator:	$(Z, Z) \rightarrow \%$	get a bounded generator
randomGenerator:	$(Z, Z, Z) \rightarrow \%$	get a bounded generator
randomInteger:	$() \rightarrow Z$	generate a random number
seed:	$(\%, Z) \rightarrow Z$	set the seed

where

$Z == \text{MachineInteger}$

Usage

apply r
r()

Signature

apply: % \rightarrow MachineInteger

Parameter	Type	Description
<i>r</i>	%	a pseudo-random number generator

Returns

Returns a pseudo-random integer.

Usage

```
for n in r repeat { ... }
for n in generator r repeat { ... }
```

Signature

```
generator: % → Generator MachineInteger
```

Parameter	Type	Description
<i>r</i>	%	a pseudo-random number generator

Description

This functions allows a for-loop that generates infinitely many pseudo-random numbers.

Example

The following code computes the number of tries it takes for a random generator to generate a multiple of 10:

```
multiple10():MachineInteger == {
  r := randomGenerator();
  for n in r for tries in 1.. repeat {
    zero?(n rem 10) => return tries;
  }
  never;
}
```


Usage

max r
min r

Signature

max,min: % \rightarrow MachineInteger

Parameter	Type	Description
<i>r</i>	%	a pseudo-random number generator

Returns

max(r) and min(r) return respectively the largest and smallest random integers that r can generate.

Usage

numberOfGenerators

Signature

numberOfGenerators: `MachineInteger`

Returns

Returns the number of independent generators provided.

Usage

```

randomGenerator()
randomGenerator n
randomGenerator(a, b)
randomGenerator(a, b, n)

```

Signatures

```

randomGenerator: () → %
randomGenerator: MachineInteger → %
randomGenerator: (MachineInteger, MachineInteger) → %
randomGenerator: (MachineInteger, MachineInteger, MachineInteger) → %

```

Parameter	Type	Description
<i>a, b</i>	MachineInteger	bounds for the generator
<i>n</i>	MachineInteger	identifier for the generator (optional)

Returns

randomGenerator() returns a pseudo-random generator, while randomGenerator(a, b) returns a pseudo-random number generator that generates numbers between a and b inclusive. If the optional argument n is given, then the n^{th} independent generator is returned, which allows for several independent sources of random numbers.

See also

```

numberOfGenerators

```

Usage

randomInteger()

Signature

randomInteger: $() \rightarrow \text{MachineInteger}$

Returns

Returns a pseudo-random integer.

Usage`seed(r, s)`**Signature**`seed: (% , MachineInteger) → %`

Parameter	Type	Description
<i>r</i>	%	a pseudo-random number generator
<i>s</i>	MachineInteger	a nonzero seed

Description

Sets the seed of *r* to *s* and returns *s*. This is useful when a reproducible pseudo-random sequence is desired. If an unseeded generator is called, then it seeds itself from the system clock the first time it is used, so the sequence is not reproducible. Note that setting the seed to 0 causes the generator to generate an infinite sequence of 0.

SingleFloat

Usage

import from SingleFloat

Description

SingleFloat implements the single-precision signed machine floats.

Exports

CopyableType

FloatType

PackableType

$\hat{\cdot}$ (% , %) \rightarrow % exponentiation

max: % largest single-precision machine float

min: % smallest single-precision machine float

Usage

$x \wedge y$

Signatures

$\wedge: (\%,\%) \rightarrow \%$

Parameter	Type	Description
x,y	$\%$	floats

Returns

Returns x to the power y .

Usage

max

min

Signature

max,min: %

Returns

max and min return respectively the largest and the smallest single-precision machine floats.

TotallyOrderedType

Usage

TotallyOrderedType: Category

Description

TotallyOrderedType is the category of totally ordered types, *i.e.* partially ordered types where every pair $x \neq y$ is comparable.

Exports

PartiallyOrderedType

max: $(\%, \%) \rightarrow \%$ greater element

min: $(\%, \%) \rightarrow \%$ smaller element

Usage`max(x,y)``min(x,y)`**Signature**`max,min: (%,%) → %`

Parameter	Type	Description
<i>a, b</i>	<code>%</code>	elements of the type

Returns

`max(a,b)` and `min(a,b)` respectively the largest and the smallest among `a` and `b`.

BinaryReader

Usage

import from BinaryReader

Description

BinaryReader provides various binary input streams.

Exports

<code>bin:</code>	<code>%</code>	standard input stream
<code>binaryReader:</code>	<code>() → Byte → %</code>	create a stream
<code>read!:</code>	<code>% → Byte</code>	read from a stream

Usage

bin

Signature

bin: %

Returns

bin is the standard input stream.

Usage

binaryReader f

Signature

binaryReader: $() \rightarrow \text{Byte} \rightarrow \%$

Parameter	Type	Description
f	$() \rightarrow \text{Byte}$	the single-byte read function

Returns

Returns the input stream for which $f()$ reads a byte.

Usage

read! s

Signature

read!: % \rightarrow Byte

Parameter	Type	Description
<i>s</i>	%	a stream

Returns

Reads a byte from *s* and returns it. Some streams may return the byte `eof` if the end of file is reached.

BinaryWriter

Usage

import from BinaryWriter

Description

BinaryWriter provides various binary output streams.

Exports

berr:	%	the binary standard error stream
bout:	%	the binary standard output stream
binaryWriter:	(Byte → ()) → %	create a stream
binaryWriter:	(Byte → (), () → ()) → %	create a stream
flush!:	% → %	flush a stream
write!:	(Byte, %) → ()	write to a stream

Usage

berr
bout

Signature

berr,bout: %

Returns

berr and bout are the binary standard error and standard output streams respectively.

Usage

binaryWriter f
binaryWriter(f, g)

Signatures

binaryWriter: (Byte → ()) → %
binaryWriter: (Byte → (), () → ()) → %

Parameter	Type	Description
<i>f</i>	Byte → ()	the single-byte write function
<i>g</i>	() → ()	the flush function (optional)

Returns

Returns the output stream for which $f(c)$ writes the byte c and such that $g()$ flushes the stream.
If g is not given, then flushing the resulting stream has no effect.

Usage

flush! s

Signature

flush!: % → %

Parameter	Type	Description
<i>s</i>	%	a stream

Description

flush!(s) causes all previous values inserted into s to be really written and returns the stream. Has no effect on unbuffered streams, such as `berr`.

Usage

write!(c, s)

Signature

write!: (Byte, %) → ()

Parameter	Type	Description
<i>c</i>	Byte	byte to write
<i>s</i>	%	a stream

Returns

Writes the byte *c* to the stream *s* and returns *c*.

Byte

Usage

import from Byte

Description

Byte implements machine bytes.

Exports

HashType

InputType

OutputType

PackableType

SerializableType

coerce: % → MachineInteger conversion to an integer

coerce: % → Character conversion to a character

coerce: Character → % conversion from a character

eof: % end-of-file marker

lowByte: MachineInteger → % low-byte of an integer

Usage`b::MachineInteger``b::Character``c::%`**Signatures**`coerce: % → MachineInteger``coerce: % → Character``coerce: Character → %`

Parameter	Type	Description
<i>b</i>	%	a byte
<i>c</i>	Character	a character

Returns

`b::MachineInteger` and `b::Character` return `b` converted to an integer and a character respectively, while `c::%` returns `c` converted to a byte.

Byte

eof

Usage
eof

Signature
eof: %

Returns
eof is the end-of-file marker.

Usage

lowByte n

Signature

lowByte: `MachineInteger` \rightarrow %

Parameter	Type	Description
<i>n</i>	<code>MachineInteger</code>	an integer

Returns

Returns the low-byte of n.

Character

Usage

import from Character

Description

Character implements machine characters.

Exports

HashType

InputType

OutputType

PackableType

SerializableType

TotallyOrderedType

char:	MachineInteger → %	create a character
digit?:	% → Boolean	test for a decimal digit
eof:	%	end-of-file character
letter?:	% → Boolean	test for a letter
lower:	% → %	convert to lower case
newline:	%	newline character
null:	%	null character
ord:	% → MachineInteger	character code
space?:	% → Boolean	test for a blank space
tab:	%	tab character
upper:	% → %	convert to upper case

Usage

char *n*
ord *c*

Signatures

char: `MachineInteger` \rightarrow %
ord: % \rightarrow `MachineInteger`

Parameter	Type	Description
<i>n</i>	<code>MachineInteger</code>	a character code
<i>c</i>	%	a character

Returns

char(*n*) returns the character whose code is *n*, while ord(*c*) returns the code corresponding to the character *c*.

Usage

digit? c

letter? c

space? c

Signaturedigit?,letter?,space?: % \rightarrow Boolean

Parameter	Type	Description
<i>c</i>	%	a character

Returns

digit?(c) returns *true* if c is in the range '0'–'9', *false* otherwise, while letter?(c) returns *true* if c is in the range 'a'–'z' or the range 'A'–'Z', *false* otherwise and space?(c) returns *true* if c is a blank space, *i.e.* a space or a tab, *false* otherwise.

Usage

eof
newline
null
tab

Signature

eof,newline,null,tab: %

Returns

eof is the end-of-file character, newline is the newline character, null is the 0-character (used to terminate strings) and tab is the tab character.

Usage

lower c
upper c

Signature

lower,upper: % \rightarrow %

Parameter	Type	Description
<i>c</i>	%	a character

Returns

lower(c) and upper(c) return c converted to lower, respectively upper, case.

File

Usage

import from File

Description

File is a type whose elements are operating system files.

Exports

<code>close!:</code>	<code>% → ()</code>	close a file
<code>coerce:</code>	<code>% → BinaryReader</code>	conversion to a binary input stream
<code>coerce:</code>	<code>% → BinaryWriter</code>	conversion to a binary output stream
<code>coerce:</code>	<code>% → TextReader</code>	conversion to a text input stream
<code>coerce:</code>	<code>% → TextWriter</code>	conversion to a text output stream
<code>fileAppend:</code>	<code>MachineInteger</code>	mode for <code>open</code>
<code>fileBinary:</code>	<code>MachineInteger</code>	mode for <code>open</code>
<code>fileRead:</code>	<code>MachineInteger</code>	mode for <code>open</code>
<code>fileText:</code>	<code>MachineInteger</code>	mode for <code>open</code>
<code>fileWrite:</code>	<code>MachineInteger</code>	mode for <code>open</code>
<code>open:</code>	<code>(String, MachineInteger) → %</code>	open a file
<code>remove:</code>	<code>String → ()</code>	removes a file
<code>uniqueName:</code>	<code>String → String</code>	get a unique filename

Usage

close! f

Signature

close!: % \rightarrow ()

Parameter	Type	Description
<i>f</i>	%	a file

Description

Closes the file f.

Usage

```
f::BinaryReader
f::BinaryWriter
f::TextReader
f::TextWriter
```

Signatures

```
coerce: % → BinaryReader
coerce: % → BinaryWriter
coerce: % → TextReader
coerce: % → TextWriter
```

Parameter	Type	Description
<i>f</i>	%	a file

Description

Converts the file *f* to to a binary or text reader or writer. This is necessary before reading from or writing to the file. The file must have been opened in an appropriate mode for reading or writing respectively.

Remarks

Coercing a file to a reader or writer allocates memory, so it is advisable to assign the resulting stream to a variable. Unlike the ones for `String`, those coercions do not reset the file to its beginning.

Usage

fileAppend
fileBinary
fileRead
fileText
fileWrite

Signature

fileAppend,fileBinary,fileRead,fileText,fileWrite: `MachineInteger`

Description

Those constants are for use in the mode parameter of the `open` function.

Usage

`open(s,m)`

Signature

`open: (String, MachineInteger) → %`

Parameter	Type	Description
<i>s</i>	String	a filename
<i>m</i>	MachineInteger	a mode (optional)

Description

Opens the file with the name `s` in the mode `m`, and returns the opened file. The mode is any combination of the constants `fileAppend`, `fileRead`, and `fileWrite`, together with one of `fileBinary` or `fileText`, grouped together with `+` or `\`. The default is `fileRead + fileText`.

Remarks

`open` returns `nil` and throws the exception `FileException` if the file cannot be opened for any reason.

Usage

remove s

Signature

remove: `String` \rightarrow `()`

Parameter	Type	Description
<i>s</i>	<code>String</code>	A file name

Description

Removes the file with name *s* in the file system.

Usage

uniqueName s

Signature

uniqueName: `String` \rightarrow `String`

Parameter	Type	Description
<i>s</i>	<code>String</code>	A root string

Returns

Returns a unique name with prefix *s* in the file system.

FileException

Usage

```
throw FileException  
try ... catch E in { E has FileExceptionType =; ... }
```

Description

FileException is an exception type thrown by file operations.

FileExceptionType

Usage

FileExceptionType: Category

Description

FileExceptionType is the category of exceptions thrown by file operations.

InputType

Usage

InputType: Category

Description

InputType is the category of types whose objects can be read in in text format.

Exports

<<: TextReader \rightarrow % read using text encoding

Usage

<< s

Signature

<<: TextReader → %

Parameter	Type	Description
<i>s</i>	TextReader	an input stream

Returns

<< s reads an element of the current type in text format from the stream s and returns the element read.

OutputType

Usage

OutputType: Category

Description

OutputType is the category of types whose objects can be written onto text writers.

Exports

`<<: (TextWriter, %) → TextWriter` write using text encoding

Usage

`s << x`

Signature

`<<: (TextWriter, %) → TextWriter`

Parameter	Type	Description
<i>s</i>	TextWriter	an output stream
<i>x</i>	%	an object of the type

Returns

writes `x` in text format to the stream `s` and returns `s` after the write.

Example

```
import from TextWriter, MachineInteger, Character;
stdout << 65 << space;
writes "65 " to the standard output stream.
```

SerializableType

Usage

SerializableType: Category

Description

SerializableType is the category of types whose objects can be read in and written out in binary mode.

Exports

`<<: BinaryReader → %` read using binary encoding
`<<: (BinaryWriter, %) → BinaryWriter` write using binary encoding

Usage

```
out << x
<< in
```

Signatures

```
<<: (BinaryWriter, %) → BinaryWriter
<<: BinaryReader → %
```

Parameter	Type	Description
<i>in</i>	BinaryReader	an input stream
<i>out</i>	BinaryWriter	an output stream
<i>x</i>	%	an object of the type

Returns

out << x writes x in binary format to the stream out and returns the stream after the write, while << in reads an element of the type in binary format from the stream in and returns the element read.

SyntaxException

Usage

throw SyntaxException

try ... catch E in { E has SyntaxExceptionType =; ... }

Description

SyntaxException is an exception type thrown by read operations from a `TextReader`.

SyntaxExceptionType

Usage

SyntaxExceptionType: Category

Description

SyntaxExceptionType is the category of exceptions thrown by read operations from a **TextReader**.

TextReader

Usage

import from TextReader

Description

TextReader provides various input streams.

Exports

<code>push!:</code>	<code>(Character, %) → ()</code>	push back a character
<code>read!:</code>	<code>% → Character</code>	read from a stream
<code>stdin:</code>	<code>%</code>	standard input stream
<code>textReader:</code>	<code>((() → Character, Character → ())) → %</code>	create a stream

Usage

push!(c, s)

Signature

push!: (Character,%) → ()

Parameter	Type	Description
<i>c</i>	Character	a character
<i>s</i>	%	a stream

Returns

Pushes the character *c* back onto *s*. One character of pushback is guaranteed, this function might however fail if it is called too many times on the same stream without intervening read or positioning calls.

Usage

read! s

Signature

read!: % → Character

Parameter	Type	Description
<i>s</i>	%	a stream

Returns

Reads a character from *s* and returns it. Returns `eof` if the end of file is reached.

Usage

stdin

Signature

stdin: %

Returns

stdin is the standard input stream.

Usage

textReader(*f*, *g*)

Signature

textReader: $() \rightarrow \text{Character}, \text{Character} \rightarrow () \rightarrow \%$

Parameter	Type	Description
<i>f</i>	$() \rightarrow \text{Character}$	the single-character read function
<i>g</i>	$\text{Character} \rightarrow ()$	the single-character pushback function

Returns

Returns the input stream for which $f()$ reads a character and $g(c)$ pushes back the character c .

TextWriter

Usage

import from TextWriter

Description

TextWriter provides various output streams.

Exports

<code>flush!:</code>	<code>% → %</code>	flush a stream
<code>stderr:</code>	<code>%</code>	the standard error stream
<code>stdout:</code>	<code>%</code>	the standard output stream
<code>textWriter:</code>	<code>(Character → ()) → %</code>	create a stream
<code>textWriter:</code>	<code>(Character → (), () → ()) → %</code>	create a stream
<code>write!:</code>	<code>(Character, %) → ()</code>	write to a stream

Usage

flush! s

Signature

flush!: % → %

Parameter	Type	Description
<i>s</i>	%	a stream

Description

flush!(s) causes all previous values inserted into s to be really written and returns the stream. Has no effect on unbuffered streams, such as `stderr`.

Usage

stderr
stdout

Signature

stderr,stdout: %

Returns

stderr and stdout are the standard error and standard output streams respectively.

Usage

TextWriter f
TextWriter(f, g)

Signatures

TextWriter: (Character → ()) → %
TextWriter: (Character → (), () → ()) → %

Parameter	Type	Description
<i>f</i>	Character → ()	the single-character write function
<i>g</i>	() → ()	the flush function (optional)

Returns

Returns the output stream for which $f(c)$ writes the character c and such that $g()$ flushes the stream. If g is not given, then flushing the resulting stream has no effect.

Usage

write!(c, s)

Signature

write!: (Character, %) → ()

Parameter	Type	Description
<i>c</i>	Character	character to write
<i>s</i>	%	a stream

Returns

Writes the character *c* to the stream *s* and returns *c*.

WriterManipulator

Usage

import from WriterManipulator

Description

WriterManipulator provides manipulators for text or binary writers.

Exports

<code><<</code>	<code>(BinaryWriter, %)</code>	<code>→ BinaryWriter</code>	manipulate a binary writer
<code>endl</code>	<code>%</code>		send a newline and flush the stream
<code>flush</code>	<code>%</code>		flush the stream

Usage

out << x

Signature

<<: (BinaryWriter, %) → BinaryWriter

Parameter	Type	Description
<i>out</i>	BinaryWriter	an output stream
<i>x</i>	%	a manipulator

Returns

out << x takes the action given by x on the stream out and returns the stream after the action.

Usage

endl

Signature

endl: %

Description

Sending endl to a text or binary writer causes a **newline** to be sent to the stream and then the stream to be flushed, so `s << endl` is equivalent to `flush!(s << newline)`.

Usage

flush

Signature

flush: %

Description

Sending flush to a text or binary writer causes the stream to be flushed, so `s << flush` is equivalent to `flush!(s)`. Has no effect on unbuffered streams, such as `stderr`.

Array

Usage

import from Array T

Parameter	Type	Description
T	Type	the type of the array entries

Description

Array provides arrays of entries of type T , 0-indexed and without bound checking.

Exports

ArrayType(T , PrimitiveArray T)

ArrayException

Usage

throw ArrayException

try ... catch E in { E has ArrayExceptionType = ... }

Description

ArrayException is an exception type thrown by array access.

ArrayExceptionType

Usage

ArrayExceptionType: Category

Description

ArrayExceptionType is the category of exceptions thrown by array access.

ArrayType

Usage

ArrayType(T , P): Category

Parameter	Type	Description
T	Type	the type of the array entries
P	PrimitiveArrayType T	the type of the underlying data

Description

ArrayType is the category of arrays whose entries are of type T and whose underlying data is of type P .

Exports

BoundedFiniteLinearStructureType T

array: $(P, \text{MachineInteger}) \rightarrow \%$ construction of an array

data: $\% \rightarrow P$ access to raw data

new: $\text{MachineInteger} \rightarrow \%$ creation of an array

resize!: $(\%, \text{MachineInteger}) \rightarrow \%$ resize an array

sort!: $(\%, (T, T) \rightarrow \text{Boolean}) \rightarrow \%$ sort an array

if T has TotallyOrderedType then

TotallyOrderedType

binarySearch: $(T, \%) \rightarrow (\text{Boolean}, \text{MachineInteger})$ binary search

sort!: $\% \rightarrow \%$ sort an array

Usage

array(p, n)

Signature

array: (P, MachineInteger) → %

Parameter	Type	Description
p	P	a primitive array structure
n	MachineInteger	a number of elements

Returns

Returns an array containing the first n entries of p. No copying is made.

Usage

binarySearch(*t*, *a*)

Signature

binarySearch: (T, %) → (Boolean, MachineInteger)

Parameter	Type	Description
<i>t</i>	T	the value to search for
<i>a</i>	%	an array

Returns

Returns (found?, *i*) such that $0 \leq i < \#a$ and $t = a.i$ if found? is *true*. Otherwise, found? is *false* and:

- if $i < 0$ then $t < a.0$;
- if $i \geq \#a - 1$ then $t > a(\#a - 1)$;
- if $0 \leq i < \#a - 1$, then $a.i < t < a(i + 1)$.

The array *a* must be sorted in increasing order. If *a* is sorted with respect to a different order, it is still possible to use binary search, but from `BinarySearch`.

See also

linearSearch

Usage

data a

Signature

data: $\% \rightarrow P$

Parameter	Type	Description
<i>a</i>	$\%$	an array

Returns

Returns the raw data of the array *a*. No copying is made. This function can be used for efficiency before accessing the elements of *a* inside a loop, or in order to pass the elements of *a* to a C function.

Usage

new n

Signature

new: MachineInteger \rightarrow %

Parameter	Type	Description
n	MachineInteger	a nonnegative size

Returns

Returns an array of n uninitialized entries.

Usage

resize!(a, n)

Signature

resize!: ($\%$, MachineInteger) \rightarrow $\%$

Parameter	Type	Description
a	$\%$	an array
n	MachineInteger	a nonnegative size

Returns

Returns an array of n entries, whose first $\#a$ entries are the first $\#a$ entries of a and whose remaining entries are uninitialized.

Usage

```
sort! a
sort!(a, f)
```

Signature

```
sort!::  (%,(T,T) → Boolean) → %
```

Parameter	Type	Description
a	$\%$	a primitive array
f	$(T, T) \rightarrow \text{Boolean}$	a comparison function

Description

Sorts the array a using the ordering $x < y \iff f(x, y)$. The comparison function f is optional if T has `TotallyOrderedType`, in which case the order function of T is taken.

BoundedFiniteDataStructureType

Usage

BoundedFiniteDataStructureType T: Category

Parameter	Type	Description
T	Type	the type of the entries

Description

BoundedFiniteDataStructureType is the category of finite general structures whose entries are of type T and whose size is always known.

Exports

CopyableType

DataStructureType

#: $\% \rightarrow \text{MachineInteger}$ number of entries

generator: $\% \rightarrow \text{Generator } T$ iteration over a structure

if T has PrimitiveType then

findAll: $(T, \%) \rightarrow \text{Generator Cross}(\text{MachineInteger}, T)$ linear search

member?: $(T, \%) \rightarrow \text{Boolean}$ look for a value

if T has HashType then

HashType

if T has OutputType then

OutputType

Usage

a

Signatures

#: % → MachineInteger

Parameter	Type	Description
<i>a</i>	%	a finite data structure

Returns

Returns the number of entries in the structure *a*.

Usage

for pair in findAll(*t*, *a*) repeat { (*pos*, *val*) := pair; ... }

Signature

findAll: (*T*, %) → Generator Cross(MachineInteger, *T*)

Parameter	Type	Description
<i>t</i>	<i>T</i>	the value to search for
<i>a</i>	%	a finite data structure

Description

Iterates through all pairs (i, x) such that $t = x$ (using the equality of the type T). The index i is the position of x in the iteration of t by the function **generator**: $i = 1$ means x is the first element generated, $i = 2$ means x is the second element generated, etc.

See also

member?

Usage

for x in a repeat { ... }
 for x in generator a repeat { ... }

Signature

generator: % \rightarrow Generator T

Parameter	Type	Description
<i>a</i>	%	a finite data structure

Description

This function allows a structure to be iterated independently of its representation. This generator yields the elements of *a* in some order, which is determined by the actual type.

Example

The following code computes the sum of all the elements of an array of machine integers:

```
sum(a:Array MachineInteger, n:MachineInteger):MachineInteger == {
  s:MachineInteger := 0;
  for x in a repeat s := s + x;
  s;
}
```

Usage

member?(t, a)

Signature

member?: (T, %) → Boolean

Parameter	Type	Description
<i>t</i>	T	the value to search for
<i>a</i>	%	a finite data structure

Returns

Returns *true* if t is a member of a, *false* otherwise.

BoundedFiniteLinearStructureType

Usage

BoundedFiniteLinearStructureType T: Category

Parameter	Type	Description
T	Type	the type of the entries

Description

BoundedFiniteLinearStructureType is the category of finite linear structures whose entries are of type T and whose size is always known.

Exports

BoundedFiniteDataStructureType T

FiniteLinearStructureType T

map: $(T \rightarrow T) \rightarrow \% \rightarrow \%$ lift a mapping

map!: $(T \rightarrow T) \rightarrow \% \rightarrow \%$ lift a mapping

if T has PrimitiveType then

PrimitiveType

linearSearch: $(T, \%) \rightarrow (\text{Boolean}, \text{MachineInteger}, T)$

linear search

$(T, \%, \text{MachineInteger}) \rightarrow (\text{Boolean}, \text{MachineInteger}, T)$

if T has InputType then

InputType

if T has SerializableType then

SerializableType

Usage

```
linearSearch(t, a)
linearSearch(t, a, n)
```

Signature

```
linearSearch: (T, %, MachineInteger) → Boolean, MachineInteger, T
```

Parameter	Type	Description
t	T	the value to search for
a	%	a finite linear structure
n	MachineInteger	an initial index (optional)

Returns

Returns (found?, i, a.i) such that $t = a.i$ if found? is *true*, t is not in a otherwise. If the optional argument n is present, then the search starts at the entry $a.n$ and ignores the previous ones.

See also

```
findAll
```

Usage

```
map f
map! f
map(f)(a)
map!(f)(a)
```

Signature

```
map: (T → T) → % → %
```

Parameter	Type	Description
f	$T \rightarrow T$	a map
a	$\%$	a finite linear structure

Returns

$\text{map}(f)(a)$ returns the new structure `[f(x) for x in a]`, while $\text{map}(f)$ returns the mapping $a \rightarrow [\text{f}(x) \text{ for } x \text{ in } a]$. In both cases, map! does not make a copy of the structure a but modifies it in place.

CheckingArray

Usage

import from CheckingArray T

Parameter	Type	Description
T	Type	the type of the array entries

Description

CheckingArray provides arrays of entries of type T , 0-indexed and with bound checking.

Exports

ArrayType(T , PrimitiveArray T)

Remarks

The functions `apply` and `set!` throw the exception `ArrayException` when attempting to access an array out of its bounds.

CheckingList

Usage

import from CheckingList T

Parameter	Type	Description
T	Type	the type of the list entries

Description

CheckingList provides lists of entries of type T , 1-indexed and with bound checking.

Exports

ListType T

CheckingMemoryBlock

Usage

import from CheckingMemoryBlock

Description

CheckingMemoryBlock provides packed arrays of bytes, 0-indexed and with bound checking.

Exports

ArrayType(Byte, PrimitiveMemoryBlock)

Remarks

The functions `apply` and `set!` throw the exception `ArrayException` when attempting to access a memory block out of its bounds.

DataStructureType

Usage

DataStructureType: Category

Description

DataStructureType is the category of general data structures, not necessarily finite or linear.

Exports

<code>empty?:</code>	<code>% → Boolean</code>	test whether a structure is empty
<code>free!:</code>	<code>% → ()</code>	memory disposal

Usage

empty? a

Signature

empty?: % → Boolean

Parameter	Type	Description
<i>a</i>	%	a data structure

Returns

Returns *true* if *a* contains no element, *false* otherwise.

Usage

free! a

Signature

free!: % → ()

Parameter	Type	Description
<i>a</i>	%	a data structure

Description

Releases the memory occupied by *a*.

DynamicDataStructureType

Usage

DynamicDataStructureType T: Category

Parameter	Type	Description
T	Type	the type of the entries

Description

DynamicDataStructureType is the category of finite general structures in which entries of type T can be inserted or removed dynamically.

Exports

BoundedFiniteDataStructureType T

insert: (T, %) → % add an element

insert!: (T, %) → % add an element

if T has PrimitiveType then

remove: (T, %) → % remove an element

remove!: (T, %) → % remove an element

removeAll: (T, %) → % remove all occurrences of an element

removeAll!: (T, %) → % remove all occurrences of an element

Usage

insert(*t*, *a*)
insert!(*t*, *a*)

Signature

insert: (T, %) → %

Parameter	Type	Description
<i>t</i>	T	an element to add
<i>a</i>	%	a dynamic data structure

Description

Adds *t* to *a* and returns the new structure. insert creates a new structure while insert! modifies *a* itself.

Usage

remove(*t*, *a*)
remove!(*t*, *a*)
removeAll(*t*, *a*)
removeAll!(*t*, *a*)

Parameter	Type	Description
<i>t</i>	T	an element to remove
<i>a</i>	%	a dynamic data structure

Description

remove(*t*, *a*) and remove!(*t*, *a*) remove the first occurrence of *t* in *a* and return the new structure, while removeAll(*t*, *a*) and removeAll!(*t*, *a*) remove all the occurrences of *t* in *a*. remove and removeAll create a new structure, while remove! and removeAll! modify *a* itself.

FiniteLinearStructureType

Usage

FiniteLinearStructureType T: Category

Parameter	Type	Description
<i>T</i>	Type	the type of the entries

Description

FiniteLinearStructureType is the category of finite linear structures whose entries are of type T.

Exports

LinearStructureType T		
<code>[]:</code>	<code>Tuple T → %</code>	construction of a structure
<code>empty:</code>	<code>%</code>	empty structure
<code>new:</code>	<code>(MachineInteger, T) → %</code>	creation of a structure

Usage

$[t_1, \dots, t_n]$

Signature

$[]: \text{ Tuple } T \rightarrow \%$

Parameter	Type	Description
t_1, \dots, t_n	T	elements of T

Returns

Returns the structure $[t_1, \dots, t_n]$.

Usage

empty

Signature

empty: empty

Returns

Returns an empty structure.

See also

empty?

Usage

`new(n, x)`

Signature

`new: (MachineInteger, T) → %`

Parameter	Type	Description
n	MachineInteger	a nonnegative size
x	T	an entry

Returns

Returns a structure of n entries, all of them set to x .

HashTable

Usage

```
import from HashTable(K, V)
import from HashTable(K, V, h)
```

Parameter	Type	Description
K	PrimitiveType HashType	the type of the keys
V	Type	the type of the entries
h	$K \rightarrow \text{MachineInteger}$	the hash function to use

Description

HashTable provides hash tables with keys of type K , entries of type V and that uses the hash-function h . If K has `HashType`, then the parameter h is optional as the function `hash` is used by default in that case.

Exports

```
TableType(K, V)
```

KeyEntry

Usage

import from KeyEntry(K, V)

Parameter	Type	Description
K	Type	the type of the keys
V	Type	the type of the entries

Description

KeyEntry(K , V) provides pairs consisting of a key from K , and a entry from V . When K is a `PrimitiveType`, then two pairs are equal if they share the same key. When K is a `TotallyOrderedType`, then the pair x is greater than the pair y if $key(x) > key(y)$. Hence key-entry pairs are useful for building tables of entries where each slot is given by a unique key.

Exports

`CopyableType`

`[]`: (K , V) \rightarrow % construction of a key-entry pair

`entry`: % \rightarrow V get the entry

`explode`: % \rightarrow (K , V) get the key and the entry

`key`: % \rightarrow K get the key

`free!`: % \rightarrow () memory disposal

`setEntry!`: (% , V) \rightarrow V change the entry

`setKey!`: (% , K) \rightarrow K change the key

if K has `PrimitiveType` then

`PrimitiveType`

if K has `TotallyOrderedType` then

`TotallyOrderedType`

Usage
 $[k, v]$

Signature
 $[]: (K, V) \rightarrow \%$

Parameter	Type	Description
k	K	a key
v	V	an entry

Returns
Returns the key-entry pair $[k, v]$.

Usage

entry p
(k, v) := explode p
key p

Signatures

entry: % \rightarrow V
explode: % \rightarrow (K, V)
key: % \rightarrow K

Parameter	Type	Description
p	%	a key-entry pair

Returns

entry(p) and key(p) return respectively the entry and key of p , while explode(p) returns the pair (key p , entry p).

Usage

setEntry!(p, v)

setKey!(p, k)

Signatures

setEntry!: ($\%$, V) \rightarrow V

setKey!: ($\%$, K) \rightarrow K

Parameter	Type	Description
p	$\%$	a key-entry pair
k	K	a key
v	V	an entry

Description

setEntry!(p, v) (resp. setKey!(p, k)) changes the entry (resp. key) of p to v (resp. k) and returns v (resp. k).

HashType

Usage

HashType: Category

Description

HashType is the category of types whose objects can be hashed into machine integers.

Exports

PrimitiveType

hash: % \rightarrow MachineInteger hash function

Usage

hash x

Signature

hash: % \rightarrow MachineInteger

Parameter	Type	Description
<i>x</i>	%	an element of the type

Returns

Returns a hash-code for x.

LinearStructureType

Usage

LinearStructureType T: Category

Parameter	Type	Description
T	Type	the type of the entries

Description

LinearStructureType is the category of linear structures whose entries are of type T.

Exports

DataStructureType

`[]`: Generator T \rightarrow % construction of a structure

`apply`: (% MachineInteger) \rightarrow T extraction of an entry

`firstIndex`: MachineInteger index of first element

`set!`: (% MachineInteger, T) \rightarrow T modification of an entry

if T has PrimitiveType then

`equal?`: (% %, MachineInteger) \rightarrow Boolean compare the first n elements

Usage

[t for t in g]

Signature

[]: Generator T → %

Parameter	Type	Description
<i>g</i>	Generator T	an iterator producing elements of T

Returns

Returns the structure composed of all the elements generated by *g* in the order in which they are generated.

Usage

apply(a, n)
a.n

Signature

apply: ($\%$, MachineInteger) \rightarrow T

Parameter	Type	Description
a	$\%$	a linear structure
n	MachineInteger	an index

Returns

Returns the element of a with index n . The position of that element depends on the indexing scheme of each actual type, for example if it is 0-indexed, then $a.n$ returns the $(n + 1)^{\text{st}}$ element of a , while if it is 1-indexed, then $a.n$ returns the n^{th} element of a .

Remarks

Bound checking depends on the actual type, some perform it and some do not.

Usage

firstIndex

Signature

firstIndex: MachineInteger

Returns

Returns the index of the first element of a structure.

Usage

```
set!(a, n, x)
a.n := x;
```

Signature

```
set!:  (% , MachineInteger, T) → T
```

Parameter	Type	Description
a	%	a linear structure
n	MachineInteger	an index
x	T	an entry

Description

Sets the element of a with index n to x and returns x . The position of that element depends on the indexing scheme of each actual type, for example if it is 0-indexed, then $a.n := x$ sets the $(n + 1)^{\text{st}}$ element of a , while if it is 1-indexed, then $a.n := x$ sets the n^{th} element of a .

Remarks

Bound checking depends on the actual type, some perform it and some do not.

Usage

equal?(a, b, n)

Signature

equal?: (% , % , MachineInteger) → Boolean

Parameter	Type	Description
a, b	%	linear structures
n	MachineInteger	a nonnegative integer

Returns

Returns *true* if the first n elements of a and b are all equal, *false* otherwise.

Remarks

a and b must both have at least n elements.

List

Usage

import from List T

Parameter	Type	Description
T	Type	the type of the list entries

Description

List provides lists of entries of type T , 1-indexed and without bound checking.

Exports

ListType T

ListException

Usage

```
throw ListException  
try ... catch E in { E has ListExceptionType = j ... }
```

Description

ListException is an exception type thrown by list access.

ListExceptionType

Usage

ListExceptionType: Category

Description

ListExceptionType is the category of exceptions thrown by list access.

ListType

Usage

ListType T: Category

Parameter	Type	Description
T	Type	the type of the list entries

Description

ListType is the category of lists of entries of type T .

Exports

BoundedFiniteLinearStructureType T

DynamicDataStructureType T

+	(%, MachineInteger) → %	translate the base
append!:	(%, T) → %	adds an entry at the end
append!:	(%, %) → %	adds a list at the end
cons:	(T, %) → %	adds an entry at the front
delete!:	(%, MachineInteger) → %	remove an entry
first:	% → T	first entry
rest:	% → %	all entries after the first
reverse:	% → %	reverse a list
reverse!:	% → %	reverse a list in-place
setFirst!:	(%, T) → T	changes the first element of a list
setRest!:	(%, %) → %	changes the rest of a list
sort!:	(%, (T, T) → Boolean) → %	sort a list

if T has PrimitiveType then

find: (T, %) → (%, MachineInteger) linear search

if T has TotallyOrderedType then

TotallyOrderedType

sort!: % → % sort a list

Usage $l + n$ **Signature** $+: (\%, \text{MachineInteger}) \rightarrow \%$

Parameter	Type	Description
l	$\%$	a list
n	<code>MachineInteger</code>	an index

Returns

$l+n$ returns the sublist of l starting at the $(n+1)^{\text{st}}$ element of l , *i.e.* $l+0$ returns l , $l+1$ returns the sublist starting at the second element of l , etc. . . . No copy of l is made.

Example

If l is the list of `MachineInteger` $[1, 2, 3, 4, 5]$, then $l+2$ is the list $[3, 4, 5]$.

Usage

append!(l, x)
append!(l, s)

Signatures

append!: ($\%$, T) \rightarrow $\%$
append!: ($\%$, $\%$) \rightarrow $\%$

Parameter	Type	Description
l, s	$\%$	lists
x	T	an entry

Returns

append!(l,x) and append!(l,s) return the lists $[l, x]$ and $[l, s]$ respectively.

Remarks

append! does not make a copy of l , which is therefore modified after the call. It is unsafe to use the variable l after the call, unless it has been assigned to the result of the call, as in `l := append!(l, x)`.

See also

cons

Usage

`cons(x, l)`

Signature

`cons: (T, %) → %`

Parameter	Type	Description
x	T	an entry
l	%	a list

Returns

Returns the list $[x, l]$. Does not make a copy of l .

See also

`append!`

Usage

delete!(l, n)

Signature

delete!: (% , MachineInteger) → %

Parameter	Type	Description
<i>l</i>	%	a list
<i>n</i>	MachineInteger	an index

Returns

Returns the list *l* with *l.n* removed. Does not make a copy of *l*.

Usage

find(*t*, *l*)

Signature

find: (T, %) → (% , MachineInteger)

Parameter	Type	Description
<i>t</i>	T	the value to search for
<i>l</i>	%	a list

Returns

Returns (*b*, *i*) such that:

- if *b* is not **empty**, then $l.i = t$ is the first occurrence of *t* in *l*, and *b* is the sublist of *l* starting at $l.i$,
- if *b* is **empty**, then *i* is undefined and *t* does not occur in *l*.

The list *a* does not need to be sorted, and the complexity is expected to be linear in its size.

Remarks

No copy of *l* is made: if *b* is not **empty**, then its data is shared with *l*. This function allows all the occurrences of *t* to be found successively in a list, as in the example below.

Example

If *l1* is the list of MachineInteger [1,6,2,5,3,7,2,4], then (*l2*, *n*) := find(2, *l1*) sets *l2* to [2,5,3,7,2,4] and *n* to 3, the further call (*l3*, *n*) := find(2, rest *l2*) sets *l3* to [2,4] and *n* to 4, and the further call (*l4*, *n*) := find(2, rest *l3*) sets *l4* to **empty**.

Usage

first *l*

Signature

first: % \rightarrow T

Parameter	Type	Description
<i>l</i>	%	a nonempty list

Returns

Returns the first entry of *l*.

Usage

rest l

Signature

rest: % \rightarrow %

Parameter	Type	Description
<i>l</i>	%	a nonempty list

Returns

Returns *l* with the first element removed. Does not make a copy of *l*.

See also

setRest!

Usage

reverse l
reverse! l

Signature

reverse: $\% \rightarrow \%$

Parameter	Type	Description
l	$\%$	a list

Returns

reverse(l) returns a copy of l with the elements in reverse order, while reverse!(l) reverses l without copying it.

Remarks

reverse! does not make a copy of l , which is therefore modified after the call. It is unsafe to use the variable l after the call, unless it has been assigned to the result of the call, as in `l := reverse! l`.

Usage

setFirst!(l,t)

Signature

setFirst!: (%T) → T

Parameter	Type	Description
<i>l</i>	%	a nonempty list
<i>t</i>	%	an element

Description

Replaces the first element of *l* by *t* and returns *t*.

Remarks

setFirst!(l,t) does not make a copy of *l*, which is therefore modified after the call.

See also

setRest!

Usage

setRest!(l,t)

Signature

setRest!: ($\%$, $\%$) \rightarrow $\%$

Parameter	Type	Description
l	$\%$	a nonempty list
t	$\%$	a list

Description

Replaces l by the list `[first(l),t]` and returns t .

Remarks

setRest!(l,t) does not make a copy of l , which is therefore modified after the call.

See also

setFirst!

Usage

```
sort! l
sort!(l, f)
```

Signature

```
sort!:: (%,(T,T) → Boolean) → %
```

Parameter	Type	Description
l	$\%$	a list
f	$(T, T) \rightarrow \text{Boolean}$	a comparison function

Description

Sorts the list a using the ordering $x < y \iff f(x, y)$. The comparison function f is optional if T has `TotallyOrderedType`, in which case the order function of T is taken.

Remarks

`sort!` does not make a copy of l , which is therefore modified after the call. It is unsafe to use the variable l after the call, unless it has been assigned to the result of the call, as in `l := sort! l`.

MemoryBlock

Usage

```
import from MemoryBlock
```

Description

MemoryBlock provides packed arrays of bytes, 0-indexed and without bound checking (the debug version of `libaldor` provides bound-checking for memory blocks).

Exports

```
ArrayType(Byte, PrimitiveMemoryBlock)
```

PackedPrimitiveArray

Usage

import from PackedPrimitiveArray T

Parameter	Type	Description
T	PackableType	the type of the array entries

Description

PackedPrimitiveArray provides packed arrays of entries of type T , 0-indexed and without bound checking (the debug version of libaldor provides bound-checking for packed primitive arrays).

Exports

PrimitiveArrayType T

PrimitiveArray

Usage

import from PrimitiveArray T

Parameter	Type	Description
T	Type	the type of the array entries

Description

PrimitiveArray provides arrays of entries of type T , 0-indexed and without bound checking (the debug version of `libaldor` provides bound-checking for primitive arrays).

Exports

PrimitiveArrayType T

PrimitiveArrayType

Usage

import from PrimitiveArrayType T

Parameter	Type	Description
T	Type	the type of the array entries

Description

PrimitiveArrayType is the category of primitive arrays whose entries are of type T .

Exports

FiniteLinearStructureType T

array: (Pointer, Z) → % conversion from a C-array

new: Z → % creation of an array

pointer: % → Pointer conversion to a C-array

resize!: (% , Z, Z) → % resize an array

sort!: (% , Z, Z, (T, T) → Boolean) → % sort an array

if T has SerializableType then

read: (BinaryReader, Z) → % read using binary encoding

write: (BinaryWriter, write using binary encoding

if T has TotallyOrderedType then

sort!: (% , Z, Z) → % sort an array

where

Z == MachineInteger

Usage

array(*p*, *n*)
 pointer *a*

Signatures

array: (Pointer, MachineInteger) → %
 pointer: % → Pointer

Parameter	Type	Description
<i>p</i>	Pointer	a C-array
<i>n</i>	MachineInteger	a nonnegative size
<i>a</i>	%	A primitive array

Description

Use those functions to safely convert between pointers returned or needed by C-functions and primitive arrays. Those function have no effect in the release version of `libaldor`, but they are necessary when using the debug version, so it is recommended to use them in all cases.

Usage

new n

Signature

new: MachineInteger \rightarrow %

Parameter	Type	Description
<i>n</i>	MachineInteger	a nonnegative size

Returns

Returns a primitive array of *n* uninitialized entries.

Usage

read(*in*, *n*)
write(*out*, *a*, *n*)

Signatures

read: (BinaryReader, MachineInteger) → %
write: (BinaryWriter, %, MachineInteger) → BinaryWriter

Parameter	Type	Description
<i>in</i>	BinaryReader	an input stream
<i>out</i>	BinaryWriter	an output stream
<i>n</i>	MachineInteger	a nonnegative size
<i>a</i>	%	A primitive array

Returns

read(*in*, *n*) reads a primitive array of *n* elements in binary format from the stream *in* and returns the array read, while write(*out*, *a*, *n*) writes the first *n* elements of *a* to the stream *out* and returns the stream after the write.

Usage

```
resize!(a, n, m)
```

Signature

```
resize!:  (%MachineInteger,MachineInteger) → %
```

Parameter	Type	Description
<i>a</i>	%	a primitive array
<i>n, m</i>	MachineInteger	nonnegative sizes

Returns

Returns a primitive array of *m* entries, whose first *n* entries are the first *n* entries of *a* and whose remaining entries are uninitialized.

Remarks

resize! may free the space previously used by *a*, so it is unsafe to use the variable *a* after the call, unless it has been assigned to the result of the call, as in `a := resize!(a, n, m)`.

Usage

```
sort!(a, n, m)
sort!(a, n, m, f)
```

Signature

```
sort!:  (%MachineInteger,MachineInteger, (T,T) → Boolean) → %
```

Parameter	Type	Description
a	%	a primitive array
n, m	MachineInteger	indices
f	$(T, T) \rightarrow \text{Boolean}$	a comparison function

Description

Sorts the subarray $[a.n, \dots, a.m]$ using the ordering $x < y \iff f(x, y)$. The comparison function f is optional if T has `TotallyOrderedType`, in which case the order function of T is taken.

PrimitiveMemoryBlock

Usage

import from PrimitiveMemoryBlock

Description

PrimitiveMemoryBlock provides packed arrays of bytes, 0-indexed and without bound checking (the debug version of libaldor provides bound-checking for primitive memory blocks).

Exports

PrimitiveArrayType Byte

coerce: % → BinaryReader conversion to a binary input stream

coerce: % → BinaryWriter conversion to a binary output stream

Usage`a::BinaryReader``a::BinaryWriter`**Signatures**`coerce: % → BinaryReader``coerce: % → BinaryWriter`

Parameter	Type	Description
<i>a</i>	%	a primitive memory block

Description

`a::T` where `T` is an I/O stream type converts the block `s` to a binary reader or writer, allowing one to read data or write data to it.

Remarks

When writing to a memory block, you must ensure that the block is large enough for all the data that will be written to it, since the block will not be extended and this function does not protect you against overwriting. When reading from or writing to a memory block, each coercion to a reader or writer resets the stream to the beginning of the block, and the block is not side-affected by the subsequent read or write operations, while the stream is side-affected. Thus, when reading several values from the same block, you must assign the reader to a variable and read the values from that variable.

See also`coerce`

.

Set

Usage

import from Set T

Parameter	Type	Description
T	PrimitiveType	the type of the set entries

Description

Set provides sets of entries of type T , 1-indexed and without bound checking.

Exports

BoundedFiniteLinearStructureType T

DynamicDataStructureType T

<code>-:</code>	<code>(%, %) → %</code>	set difference
<code>intersection:</code>	<code>(%, %) → %</code>	intersect two sets
<code>intersection!:</code>	<code>(%, %) → %</code>	intersect two sets
<code>minus!:</code>	<code>(%, %) → %</code>	set difference
<code>union:</code>	<code>(%, T) → %</code>	add an element
	<code>(%, %) → %</code>	add a set of elements
<code>union!:</code>	<code>(%, T) → %</code>	add an element
	<code>(%, %) → %</code>	add a set of elements

Usage $x - y$ **Signature** $-: (\%, \%) \rightarrow \%$

Parameter	Type	Description
x, y	$\%$	sets

Returns

Return $x - y = \{a \in x \text{ such that } a \notin y\}$.

See also

`minus!`

Usage

intersection(x,y)
intersection!(x,y)

Signature

intersection: (% , %) → %

Parameter	Type	Description
x, y	%	sets

Returns

Return $x \cap y = \{a \text{ such that } a \in x \text{ and } a \in y\}$.

Remarks

intersection! does not make a copy of x , which is therefore modified after the call. It is unsafe to use the variable x after the call, unless it has been assigned to the result of the call, as in `x := intersection!(x, y)`.

Usage

minus!(x,y)

Signature

minus!: (% , %) → %

Parameter	Type	Description
x, y	%	sets

Description

Return $x - y = \{a \in x \text{ such that } a \notin y\}$.

Remarks

minus! does not make a copy of x , which is therefore modified after the call. It is unsafe to use the variable x after the call, unless it has been assigned to the result of the call, as in `x := minus!(x, y)`.

See also

—

Usage

```
union(x,t)
union(x,y)
union!(x,t)
union!(x,y)
```

Signature

```
union:  (% , %) → %
```

Parameter	Type	Description
x, y	%	sets
t	T	an element

Returns

`union(x,y)` and `union!(x,y)` both return $x \cup y = \{a \text{ such that } a \in x \text{ or } a \in y\}$, while `union(x,t)` and `union!(x,t)` both return $x \cup \{t\}$.

Remarks

`union!` does not make a copy of x , which is therefore modified after the call. It is unsafe to use the variable x after the call, unless it has been assigned to the result of the call, as in `x := union!(x, y)`.

SortedAssociationSet

Usage

import from SortedAssociationSet(K , V)

Parameter	Type	Description
K	TotallyOrderedType	the type of the keys
V	Type	the type of the entries

Description

SortedAssociationSet(K , V) implements sorted sets of key-entry pairs. The keys come from K with a total ordering and the entries come from T which is any type. A sorted association set can be viewed as a hash-table with the identity as hash-function.

Exports

TableType(K , V)

SortedList

Usage

import from SortedList T

Parameter	Type	Description
<i>T</i>	PartiallyOrderedType	the type of the entries

Description

SortedList(T) provides sorted lists whose entries belong to a partially ordered type. Duplicate entries are allowed.

Exports

BoundedFiniteLinearStructureType T
DynamicDataStructureType T

SortedSet

Usage

import from SortedSet T

Parameter	Type	Description
<i>T</i>	PartiallyOrderedType	the type of the entries

Description

SortedSet(*T*) provides sorted sets whose entries belong to a partially ordered type. Duplicate entries are not allowed.

Exports

BoundedFiniteLinearStructureType *T*
DynamicDataStructureType *T*

Stream

Usage

import from Stream T

Parameter	Type	Description
T	Type	the type of the stream entries

Description

Stream provides streams of entries of type T , 0-indexed and with bound checking.

Exports

LinearStructureType T

#:	$\% \rightarrow Z$	number of computed elements
constant:	$\% \rightarrow (Z, \text{Partial } T)$	check for an eventually constant stream
generator :	$\% \rightarrow \text{Generator } T$	iteration over a stream
map!:	$(T \rightarrow T) \rightarrow \% \rightarrow \%$	lift a mapping
orbit:	$(T, T \rightarrow T) \rightarrow \%$	creation of a stream
stream:	$T \rightarrow \%$	creation of a stream
	$(() \rightarrow T) \rightarrow \%$	
	$((Z, T) \rightarrow T) \rightarrow \%$	
	$(Z, Z \rightarrow T) \rightarrow \%$	
	$(Z, Z \rightarrow T, Z, T) \rightarrow \%$	
	$(\text{Generator } T, T) \rightarrow \%$	
	$(\text{Generator } T, Z, T) \rightarrow \%$	

where

$Z == \text{MachineInteger}$

if T has OutputType then

OutputType

Usage
s

Signatures
#: % → MachineInteger

Parameter	Type	Description
<i>s</i>	%	a stream

Returns
Returns the number of elements of *s* that have been computed.

Usage

constant s

Signature

constant: $\% \rightarrow (\text{MachineInteger}, \text{Partial } T)$

Parameter	Type	Description
s	$\%$	a stream

Returns

Returns $(-1, \text{failed})$ if s is not known to be eventually constant, otherwise returns $(n \geq 0, [t])$ such that $s.k = s.n = t$ for $k \geq n$.

Usage

```
for x in s repeat { ... }  
for x in generator s repeat { ... }
```

Signature

```
generator:  %  $\rightarrow$  Generator T
```

Parameter	Type	Description
<i>s</i>	%	a stream

Description

This function allows the elements of a stream to be iterated.

Remarks

Since those generators are infinite, you should have a termination condition either inside the loop or in parallel with the generator in order to guarantee termination.

Example

The following code creates the stream of all the squares for $n \geq 0$ and prints those of them that are smaller than 1000:

```
import from MachineInteger, Stream MachineInteger;  
s := stream(0, (n:MachineInteger):MachineInteger +-> n^2);  
for x in s while x < 1000 repeat { stdout << x << newline; }
```

Usage

map! f
map!(f)(s)

Signature

map!: $(T \rightarrow T) \rightarrow \% \rightarrow \%$

Parameter	Type	Description
f	$T \rightarrow T$	a map
s	$\%$	a stream

Returns

map!(f)(s) returns the stream `[f(x) for x in s]`, while map!(f) returns the mapping $s \rightarrow [f(x) \text{ for } x \text{ in } s]$. In both cases, the stream `s` is modified in place and no copy is made.

Usage

orbit(*t*, *f*)

Signature

orbit: $(T, T \rightarrow T) \rightarrow \%$

Parameter	Type	Description
<i>t</i>	T	an element of T
<i>f</i>	$T \rightarrow T$	a function

Returns

Returns the stream $[t, f(t), f^2(t), \dots]$.

See also

stream

Usage

```
stream t
stream f
stream h
stream(n, g)
stream(n, g, m, t)
stream(gen, t)
stream(gen, m, t)
```

Signatures

```
stream: T → %
stream: (() → T) → %
stream: ((MachineInteger, %) → T) → %
stream: (MachineInteger, MachineInteger → T) → %
stream: (MachineInteger, MachineInteger → T, MachineInteger, T) → %
stream: (Generator T, T) → %
stream: (Generator T, MachineInteger, T) → %
```

Parameter	Type	Description
n, m	<code>MachineInteger</code>	machine integers
t	<code>T</code>	an element
f	<code>() → T</code>	a function
g	<code>MachineInteger → T</code>	a function
h	<code>(MachineInteger, %) → T</code>	a function
gen	<code>Generator T</code>	a generator

Description

`stream(t)` returns the constant stream $[t, t, t, \dots]$, while `stream(f)` returns the stream obtained by successive calls to `f()`, `stream(n, g)` returns the stream $[g(n), g(n+1), g(n+2), \dots]$, `stream(n, g, m, t)` returns the eventually constant stream $[g(n), g(n+1), g(n+2), \dots, g(m-1), t, t, t, \dots]$ and `stream(h)` returns the stream $[s_0, s_1, s_2, \dots]$ where $s_k = h(k, s_0, \dots, s_{i-k})$ for any $k \geq 0$. When using generators, `stream(gen, t)` returns the stream $[x \text{ for } x \text{ in } gen]$ followed by $[t, t, t, \dots]$ if `gen` is finite, while `stream(gen, m, t)` returns the first m elements of the stream $[x \text{ for } x \text{ in } gen]$ followed by $[t, t, t, \dots]$.

See also

`orbit`

Remarks

The function g must be defined for all $n \leq k$ (resp. $n \leq k < m$) and h must be defined for all $n \geq 0$, even if you intend to use `set!` to set specific values of the stream later. Note that `stream(f)` does not necessarily returns a constant stream since each call to `f()` can side-effect its environment. Constant or eventually constant streams do not repeatedly store their constant value and are therefore preferable. For example, `stream(0)` stores only one value no matter how many values are requested, while `stream(() : MachineInteger + - > 0)` stores n times 0 when its n^{th} element is requested.

Example

The following code creates the stream of the Fibonacci numbers and prints the first 10 of them:

```
import from MachineInteger, AldorInteger, Stream AldorInteger;
fib(n:MachineInteger, f:Stream AldorInteger):AldorInteger == {
    n = 0 or n = 1 => 1;
    f(n-1) + f(n-2);
}
sfib := stream fib;
a := sfib.9;    -- computes sfib.0 to sfib.9
stdout << a << newline;
```

String

Usage

import from String

Description

String provides basic strings, null-terminated, 0-indexed and without bound checking.

Exports

ArrayType(Character, PackedPrimitiveArray Character)

+: (% , %) → % concatenation

char: % → Character first character

coerce: Character → % conversion to a string

coerce: % → TextReader conversion to a text input stream

coerce: % → TextWriter conversion to a text output stream

error: % → Exit error exit

new: MachineInteger → % buffer allocation

pointer: % → Pointer conversion to a null-terminated C-string

string: Pointer → % conversion from a null-terminated C-string

Usage $s + t$ **Signature** $+: (\%, \%) \rightarrow \%$

Parameter	Type	Description
s, t	$\%$	strings

Returns

$s+t$ returns the string st . Copies all the characters of s and t , so s is unchanged after the call, and s and t do not share characters with $s + t$.

Remarks

$s + \text{empty}$ and $\text{empty} + s$ both return a copy of s .

Example

If s is the string “abcde”, then $s + (s + 2)$ is the string “abcdecde”.

Usage

char s

Signature

char: % \rightarrow Character

Parameter	Type	Description
s	%	a nonempty string

Description

Returns the first character of s.

Usage

```
c::%
s::TextReader
s::TextWriter
```

Signatures

```
coerce: Character → %
coerce: % → TextReader
coerce: % → TextWriter
```

Parameter	Type	Description
<i>c</i>	Character	a character
<i>s</i>	%	a string

Description

`c::String` converts the character `c` to the string “`c`”, while `s::T` where `T` is an I/O stream type converts the string `s` to a text reader or writer, allowing one to read data or write data to it.

Remarks

When writing to a string, you must ensure that the string is large enough for all the data that will be written to it, since the string will not be extended and this function does not protect you against overwriting. When reading from or writing to a string, each coercion to a reader or writer resets the stream to the beginning of the string, and the string is not side-affected by the subsequent read or write operations, while the stream is side-affected. Thus, when reading several values from the same string, you must assign the reader to a variable and read the values from that variable, as in the example below.

Example

```
import from MachineInteger, String;
s := " 12 56";
a:MachineInteger := << s::TextReader;
b:MachineInteger := << s::TextReader;
assigns the value 12 to both a and b, while
```

```
import from MachineInteger, String;
s := " 12 56";
p := s::TextReader;
a:MachineInteger := << p;
b:MachineInteger := << p;
assigns 12 to a and 56 to b.
```

See also

PrimitiveMemoryBlock

coerce

Usage

error s

Signature

error: % \rightarrow Exit

Parameter	Type	Description
<i>s</i>	%	a string

Description

Writes the message *s* to **stderr** and exits.

Usage

new n

Signature

new: `MachineInteger` \rightarrow %

Parameter	Type	Description
n	<code>MachineInteger</code>	a nonnegative size

Returns

Returns a string of $n + 1$ null characters. This is useful when creating a memory buffer of a specified number of bytes, or when creating a string to be used as an `TextWriter`.

Usage

pointer *s*
string *p*

Signatures

pointer: $\% \rightarrow \text{Pointer}$
string: $\text{Pointer} \rightarrow \%$

Parameter	Type	Description
<i>s</i>	$\%$	A <code>libaldor</code> string
<i>p</i>	<code>Pointer</code>	A null-terminated C-string

Description

Use those functions to safely convert between null-terminated returned or needed by C-functions and `libaldor` strings. Those functions have no effect in the release version of `libaldor`, but they are necessary when using the debug version, so it is recommended to use them in all cases. The C-type `char*` should be replaced by `Pointer` in the prototypes when using C-functions in `libaldor` clients.

TableType

Usage

TableType(K, V): Category

Parameter	Type	Description
K	PrimitiveType	the type of the keys
V	Type	the type of the entries

Description

TableType(K , V) is the category of tables, *i.e.* discrete many-to-one mappings from keys to entries. More precisely, every element of a domain of this category is a table whose slots contain elements from V and such that every slot is given by a unique key from K .

Exports

BoundedFiniteDataStructureType Cross(K , V)

apply:	(%, K) $\rightarrow V$	extraction of an entry
entries:	% \rightarrow Generator V	iterate through the entries
find:	(K , %) \rightarrow Partial V	search for an entry
keys:	% \rightarrow Generator K	iterate through the keys
numberOfEntries:	% \rightarrow MachineInteger	number of entries
set!:	(%, K , V) $\rightarrow V$	modification of an entry
table:	() \rightarrow %	creation of a table
	MachineInteger \rightarrow %	

if K has InputType and V has InputType then
InputType

if K has OutputType and V has OutputType then
OutputType

if K has SerializableType and V has SerializableType then
SerializableType

Usage

apply(*t*, *k*)
t.k

Signature

apply: (*%*, K) → V

Parameter	Type	Description
<i>t</i>	<i>%</i>	a table
<i>k</i>	K	a key

Returns

Returns the element of *t* with key *k*, which must be present in the table.

Remarks

Produces an error if *k* is not in *t*, use **find** if it is not known whether *k* is present in table.

Usage

for v in entries t repeat {...}
for k in keys t repeat {...}

Signatures

entries: % \rightarrow Generator V
keys: % \rightarrow Generator K

Parameter	Type	Description
<i>t</i>	%	a table

Description

These generators yield respectively all the entries, or keys in the table *t*.

Usage

find(*k*, *t*)

Signature

find: (K, %) → Partial V

Parameter	Type	Description
<i>k</i>	K	a key
<i>t</i>	%	a table

Returns

Returns *failed* if there is no element with key *k* in *t*, the element of *t* with key *k* otherwise.

See also

apply

Usage
numberOfEntries t

Parameter	Type	Description
<i>t</i>	%	a table

Returns
Returns the actual number of entries in the table *t*. That number can be different from the size of the table.

Usage

```
set!(t, k, v)
t.k := v;
```

Signature

```
set!:: (%K, K, V) -> V
```

Parameter	Type	Description
<i>t</i>	%	a table
<i>k</i>	K	a key
<i>v</i>	V	an entry

Returns

Sets the element of *t* with key *k* to *v* and returns *v*.

Usage

table()
table n

Signature

table: MachineInteger \rightarrow %

Parameter	Type	Description
<i>n</i>	MachineInteger	a starting size (optional)

Returns

Returns an empty table with initial space for *n* entries. That space grows when needed as elements are inserted in the table.

AldorLibraryInformation

Usage

import from AldorLibraryInformation

Description

AldorLibraryInformation provides `libaldor` version information.

Exports

VersionInformationType

VersionInformationType

Usage

VersionInformationType: Category

Description

VersionInformationType is the category of types providing version information.

Exports

name:	String	library name
credits:	List String	various credits
version:	String	version information

Signature

credits: List String

Returns

Returns a list of lines, crediting the various authors of a library.

Usage

name

Signature

name: **String**

Returns

Returns the name of the library.

Usage

version

Signature

version: **String**

Returns

Returns a string describing the current version of a library.

BinarySearch

Usage

import from BinarySearch(R , S)

Description

BinarySearch(R , S) provides a general version of binary search.

Parameter	Type	Description
R	IntegerType	The space being searched
S	TotallyOrderedType	The target values being searched for

Exports

binarySearch: $(S, R \rightarrow S, R, R) \rightarrow (\text{Boolean}, R)$ binary search

Usage

binarySearch(s, f, a, b)

Signature

binarySearch: $(S, R \rightarrow S, R, R) \rightarrow (\text{Boolean}, R)$

Parameter	Type	Description
s	S	The value to search for
f	$R \rightarrow S$	A monotonic increasing function
a	R	The left end of the interval to search
b	R	The right end of the interval to search

Returns

Returns (found?, r) such that $s = f(r)$ if found? is *true*. Otherwise, found? is *false* and:

- if $r < a$ then $s < f(a)$ or $b < a$;
- if $r \geq b$ then $s > f(b)$;
- if $a \leq r < b$, then $f(r) < s < f(r + 1)$;

CommandLine

Usage

import from CommandLine

Description

CommandLine provides utilities for command-line processing.

Exports

arguments:	Array String	command line arguments
command:	String	command line command
flag:	Character → List String	value of a command line flag
flag?:	Character → Boolean	test for a command line flag

Usage

arguments

Signature

arguments: **Array String**

Returns

Returns an array containing all the arguments of the command line, the command itself is not included.

Signature

command: String

Returns

Returns the command, *i.e.* the first word of the command line.

Usage

```
flag c
flag(c, s)
flag? c
flag?(c, s)
```

Signatures

```
flag:   Character → List String
flag:   (Character,String)→List String
flag?:  Character → Boolean
flag?:  (Character, String) → Boolean
```

Parameter	Type	Description
<i>c</i>	Character	flag code to look for
<i>s</i>	String	special flag codes (optional)

Returns

flag(c) returns all the values of the flag c each time it is present in the command line, an empty list otherwise.

flag?(c) returns *true* if the flag c is present in the command line, *false* otherwise.

In both functions, the optional argument s contains a list of flag codes which cause the rest of the argument to be skipped.

Example

If the command line to the program was `myprog -lsalli -l gmp -v`, then `flag?(char "a")` and `flag?(char "v")` both return *true*, while `flag?(char "b")` and `flag?(char "a", "l")` both return *false*. In addition, `flag(char "l")` returns the list `['salli', 'gmp']`.

CopyableType

Usage

CopyableType: Category

Description

CopyableType is the category of types whose objects can be copied.

Exports

<code>copy:</code>	<code>% → %</code>	Make a copy
<code>copy!:</code>	<code>(%, %) → %</code>	In-place copy

Usage

copy y
copy!(x, y)

Signatures

copy: % \rightarrow %
copy!: (% , %) \rightarrow %

Parameter	Type	Description
x, y	%	Element of the type

Returns

copy(y) returns a copy of y, while copy!(x, y) returns a copy of y, where the storage used by x is allowed to be destroyed or reused, so x is lost after this call.

Remarks

Use copy before making in-place operations on a parameter. The call copy!(x, y) may cause x to be destroyed, so do not use it unless x has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Generator

Usage

import from Generator T

Parameter	Type	Description
-----------	------	-------------

<i>T</i>	Type	the type of the elements generated
----------	------	------------------------------------

Description

Generator T is a type which allows values of type T to be obtained serially in a ‘repeat’ or ‘collect’ form.

Exports

`next!:` $\% \rightarrow T$ get the next element

Usage

next! *g*

Signature

next!: % \rightarrow T

Parameter	Type	Description
<i>g</i>	%	a generator

Returns

Returns the next element produced by *g*, updating *g*. If *g* is empty, then next!(*g*) throws the exception `GeneratorException`.

GeneratorException

Usage

throw GeneratorException

try ... catch E in { E has GeneratorExceptionType = ... }

Description

GeneratorException is an exception type thrown by stepping through an empty generator.

GeneratorExceptionType

Usage

GeneratorExceptionType: Category

Description

GeneratorExceptionType is the category of exceptions thrown by generators.

Partial

Usage

import from Partial T

Parameter	Type	Description
<i>T</i>	Type	a type

Description

Partial(T) implements a union of T and *failed*.

Exports

<code>[]:</code>	<code>T → %</code>	create an element
<code>failed:</code>	<code>%</code>	the element <i>failed</i>
<code>failed?:</code>	<code>% → Boolean</code>	check for the element <i>failed</i>
<code>retract:</code>	<code>% → T</code>	convert to an element of T

if *T* has `PrimitiveType` then
 `PrimitiveType`

if *T* has `HashType` then
 `HashType`

if *T* has `InputType` then
 `InputType`

if *T* has `OutputType` then
 `OutputType`

if *T* has `SerializableType` then
 `SerializableType`

Usage
[t]

Signature
[]: T → %

Parameter	Type	Description
<i>t</i>	T	an element

Returns
Returns the element *t* converted to an element of %.

Usage

failed
failed? x

Signatures

failed: %
failed?: % \rightarrow Boolean

Parameter	Type	Description
<i>x</i>	%	a partial element

Returns

failed returns the special element *failed*, while failed?(x) returns *true* if x is *failed*, *false* otherwise.

Usage

retract x

Signature

retract: % \rightarrow T

Parameter	Type	Description
x	%	a partial element

Returns

Returns the element x converted to an element of T, provided that x is not *failed*.

Pointer

Usage

import from Pointer

Description

Pointer implements machine pointers.

Exports

HashType

InputType

OutputType

SerializableType

coerce: % → MachineInteger conversion to an integer

coerce: MachineInteger → % conversion from an integer

nil: % the nil pointer

nil?: % → Boolean test for the nil pointer

Usage

nil
nil? p

Signatures

nil: $\rightarrow \%$
nil?: $\% \rightarrow \text{Boolean}$

Parameter	Type	Description
p	$\%$	a pointer

Returns

nil returns the nil pointer, while nil?(p) returns *true* if p is the nil pointer, *false* otherwise.

Timer

Usage

import from Timer

Description

Timer is a type whose elements are stopwatch timers, which can be used to time precisely various sections of code, including garbage collection. The precision can be up to 1 millisecond but depends on the operating system. The times returned are CPU times (user + gc) used by the process that created the timer.

Exports

<code>gc:</code>	<code>% → MachineInteger</code>	read a timer
<code>read:</code>	<code>% → MachineInteger</code>	read a timer
<code>reset!:</code>	<code>% → %</code>	reset a timer to 0
<code>start!:</code>	<code>% → MachineInteger</code>	start or restart a timer
<code>stop!:</code>	<code>% → MachineInteger</code>	stop a timer
<code>timer:</code>	<code>() → %</code>	create a new timer

Signature

gc: % \rightarrow MachineInteger

Parameter	Type	Description
<i>t</i>	%	The timer to read

Description

Reads the timer *t* without stopping it.

Returns

Returns the total accumulated garbage collection time in milliseconds by all the start/stop cycles since *t* was created or last reset. If *t* is running, the garbage collection time since the last start is added in, and *t* is not stopped or affected.

See also

read

Usage

read t

Signature

read: % \rightarrow MachineInteger

Parameter	Type	Description
<i>t</i>	%	The timer to read

Description

Reads the timer t without stopping it.

Returns

Returns the total accumulated time in milliseconds by all the start/stop cycles since t was created or last reset. This times includes any eventual garbage collection time (see `gc` to extract this information). If t is running, the time since the last start is added in, and t is not stopped or affected.

Example

The following function takes a positive `MachineInteger` *n* as input, computes and prints a machine approximation of $\sum_{i=1}^n 1/i$, and returns the CPU time needed to compute it, but not the time needed to print it.

```
timeHarmonic(n:MachineInteger):MachineInteger == {
    import from MachineInteger, SingleFloat, Timer, Character, TextWriter;
    t := timer();
    m:SingleFloat := 1;
    start! t;
    for i in 2..n repeat m := m + 1 / (i::SingleFloat);
    stop! t;
    stdout << "H" << n << " = " << m << newline;
    read t;
}
```

See also

`gc`, `start!`, `stop!`

Usage

reset! t

Signature

reset!: % \rightarrow %

Parameter	Type	Description
<i>t</i>	%	The timer to reset

Description

Resets the timer *t* to 0 and stops it if it is running.

Returns

Returns the timer *t* after it is reset.

Usage

start! t

Signature

start!: % \rightarrow MachineInteger

Parameter	Type	Description
<i>t</i>	%	The timer to start

Description

Starts or restarts t, without resetting it to 0, It has no effect on t if it is already running.

Returns

Returns 0 if t was already running, the absolute time at which the start/restart was done otherwise.

See also

read, stop!

Usage

stop! t

Signature

stop!: % → MachineInteger

Parameter	Type	Description
<i>t</i>	%	The timer to stop

Description

Stops t without resetting it to 0. It has no effect on t if it is not running.

Returns

Returns the elapsed time in milliseconds since the last time t was restarted, 0 if t was not running.

See also

read, start!

Usage

timer()

Signature

timer: () → %

Description

Creates a timer, set to 0 and stopped.

Returns

Returns the timer that has been created.

See also

reset!

Index

- *
 - LinearCombinationType, 78
- **
 - DoubleFloat, 44
 - SingleFloat, 103
- +
- ListType, 204
 - String, 243
-
- Set, 227
- /
 - FloatType, 47
- <,>
 - PartiallyOrderedType, 88
- <<
 - InputType, 135
 - OutputType, 137
 - SerializableType, 139
 - WriterManipulator, 153
- =
 - PrimitiveType, 93
- *,**
 - ArithmeticType, 26
- +,-
 - AdditiveType, 21
- ..
 - IntegerSegment, 60
- []
 - FiniteLinearStructureType, 184
 - KeyEntry, 189
 - LinearStructureType, 195
 - Partial, 274
- 0
 - AdditiveType, 20
- 1
 - ArithmeticType, 25
- abs
 - OrderedArithmeticType, 85
- AdditiveType, 19
- AldorInteger, 18
- AldorLibraryInformation, 256
- and,or,not,xor
 - BooleanArithmeticType, 35
- apply
 - LinearStructureType, 196
 - RandomNumberGenerator, 95
 - TableType, 250
- arguments
 - CommandLine, 264
- ArithmeticType, 24
- Array, 156
- array
 - ArrayType, 160
- array,pointer
 - PrimitiveArrayType, 219
- ArrayException, 157
- ArrayExceptionType, 158
- ArrayType, 159
- berr,bout
 - BinaryWriter, 112
- bin
 - BinaryReader, 108
- binaryExponentiation
 - BinaryPowering, 31
- BinaryPowering, 30
- BinaryReader, 107
- binaryReader
 - BinaryReader, 109
- BinarySearch, 261
- binarySearch
 - ArrayType, 161
 - BinarySearch, 262
- BinaryWriter, 111
- binaryWriter
 - BinaryWriter, 113
- bit?,clear,set
 - IntegerType, 66
- Boolean, 32
- BooleanArithmeticType, 34
- BoundedFiniteDataStructureType, 166
- BoundedFiniteLinearStructureType, 171
- by,step

- IntegerSegment, 61
- Byte, 116
- bytes,max,min
 - MachineInteger, 82
- char
 - String, 244
- char,ord
 - Character, 121
- Character, 120
- CheckingArray, 174
- CheckingList, 175
- CheckingMemoryBlock, 176
- coerce
 - Byte, 117
 - Complex, 37
 - File, 127
 - FloatType, 48
 - GMPFloat, 51
 - GMPInteger, 56
 - PrimitiveMemoryBlock, 225
 - String, 245
- coerce,machine
 - IntegerType, 67
- command
 - CommandLine, 265
- CommandLine, 263
- commutative?
 - ArithmeticType, 27
- Complex, 36
- complex
 - Complex, 38
- conjugate
 - Complex, 39
- cons
 - ListType, 206
- constant
 - Stream, 236
- copy
 - CopyableType, 268
- CopyableType, 267
- credits
 - VersionInformationType, 258
- data

- ArrayType, 162
- DataStructureType, 177
- digit?,letter?,space?
 - Character, 122
- divide,mod,quo,rem
 - IntegerType, 68
- DoubleFloat, 43
- DynamicDataStructureType, 180
- empty
 - FiniteLinearStructureType, 185
- empty?
 - DataStructureType, 178
- endl
 - WriterManipulator, 154
- entries, keys
 - TableType, 251
- entry,explode,key
 - KeyEntry, 190
- eof
 - Byte, 118
- eof,newline,null,tab
 - Character, 123
- equal?
 - LinearStructureType, 199
- error
 - String, 246
- even?,odd?
 - IntegerType, 69
- factorial
 - IntegerType, 70
- failed
 - Partial, 275
- File, 125
- fileAppend,fileBinary,fileRead,fileText,fileWrite
 - File, 128
- FileException, 132
- FileExceptionType, 133
- find
 - ListType, 208
 - TableType, 252
- findAll
 - BoundedFiniteDataStructureType, 168
- FiniteLinearStructureType, 183

- first
 - ListType, 209
- firstIndex
 - LinearStructureType, 197
- flag
 - CommandLine, 266
- FloatType, 46
- flush
 - WriterManipulator, 155
- fraction,truncate
 - FloatType, 49
- gc
 - Timer, 280
- gcd,lcm
 - IntegerType, 71
- Generator, 269
- generator
 - BoundedFiniteDataStructureType, 169
 - IntegerSegment, 62
 - RandomNumberGenerator, 96
 - Stream, 237
- GeneratorException, 271
- GeneratorExceptionType, 272
- GMPFloat, 50
- GMPInteger, 55
- hash
 - HashType, 193
- HashTable, 187
- HashType, 192
- high,low
 - IntegerSegment, 63
- imag,real
 - Complex, 41
- InputType, 134
- insert
 - DynamicDataStructureType, 181
- IntegerSegment, 59
- IntegerType, 65
- intersection
 - Set, 228
- KeyEntry, 188
- length
 - IntegerType, 72
- limbs
 - GMPFloat, 53
 - GMPInteger, 57
- LinearCombinationType, 77
- linearSearch
 - BoundedFiniteLinearStructureType, 172
- LinearStructureType, 194
- List, 200
- ListException, 201
- ListExceptionType, 202
- ListType, 203
- lowByte
 - Byte, 119
- lower,upper
 - Character, 124
- MachineInteger, 81
- map
 - BoundedFiniteLinearStructureType, 173
- max,min
 - DoubleFloat, 45
 - RandomNumberGenerator, 97
 - SingleFloat, 104
 - TotallyOrderedType, 106
- member?
 - BoundedFiniteDataStructureType, 170
- MemoryBlock, 215
- modX,modInverse
 - MachineInteger, 83
- name
 - VersionInformationType, 259
- new
 - ArrayType, 163
 - FiniteLinearStructureType, 186
 - GMPFloat, 54
 - GMPInteger, 58
 - PrimitiveArrayType, 220
 - String, 247
- newPackedArray
 - PackableType, 91
- next,prev
 - IntegerType, 73
- nil

- Pointer, 278
- norm
 - Complex, 42
- nthRoot
 - IntegerType, 74
- numberOfEntries
 - TableType, 253
- numberOfGenerators
 - RandomNumberGenerator, 98
- one?
 - ArithmeticType, 28
- open
 - File, 129
- open?
 - IntegerSegment, 64
- orbit
 - Stream, 239
- OrderedArithmeticType, 84
- OutputType, 136
- PackableType, 89
- PackedPrimitiveArray, 216
- Partial, 273
- PartiallyOrderedType, 87
- Pointer, 277
- pointer,string
 - String, 248
- PrimitiveArray, 217
- PrimitiveArrayType, 218
- PrimitiveMemoryBlock, 224
- PrimitiveType, 92
- random
 - IntegerType, 75
- randomGenerator
 - RandomNumberGenerator, 99
- randomInteger
 - RandomNumberGenerator, 100
- RandomNumberGenerator, 94
- read
 - Timer, 281
- read,write
 - PrimitiveArrayType, 221
- remove
 - File, 130

- remove,removeAll
 - DynamicDataStructureType, 182
- rest
 - ListType, 210
- retract
 - Partial, 276
- reverse
 - ListType, 211
- seed
 - RandomNumberGenerator, 101
- SerializableType, 138
- Set, 226
- shift
 - IntegerType, 76
- sign
 - OrderedArithmeticType, 86
- SingleFloat, 102
- size
 - BoundedFiniteDataStructureType, 167
 - Stream, 235
- SortedAssociationSet, 231
- SortedList, 232
- SortedSet, 233
- stderr,stdout
 - TextWriter, 149
- stdin
 - TextReader, 145
- Stream, 234
- stream
 - Stream, 240
- String, 242
- SyntaxException, 140
- SyntaxExceptionType, 141
- table
 - TableType, 255
- TableType, 249
- TextReader, 142
- textReader
 - TextReader, 146
- TextWriter, 147
- textWriter
 - TextWriter, 150
- Timer, 279

timer

Timer, 285

TotallyOrderedType, 105

true,false

Boolean, 33

union

Set, 230

uniqueName

File, 131

version

VersionInformationType, 260

VersionInformationType, 257

WriterManipulator, 152

zero?

AdditiveType, 23