The Oz Base Environment

Denys Duchier
Leif Kornstaedt
Christian Schulte

Version 1.3.2
June 15, 2006

mozart

Abstract

Oz is a concurrent language providing for functional, object-oriented, and constraint programming. The Oz Base Environment is part of the Oz language specification and contains procedures that are made generally available to the programmer. A thorough knowledge of the Oz Base Environment is highly recommended for effective programming in Oz.

The Oz Base Environment provides the basic operations on the values of the Oz universe and a set of procedures whose use makes for more elegant and readable programs. It provides high-level access to programming with threads, to real-time programming and to data structures such as arrays and dictionaries.

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

1 Introduction

This document describes the base environment of the Oz programming language. The environment is a mapping of identifiers to values. The values are organized in modules (modeled as records) and are available via field selection from their respective module; some values are available directly (without the need for selection) for convenience.

Your actual environment may be a superset of the one described by this document, depending on the Oz implementation you are running. For instance, the Mozart system defines more procedures when working in the interactive development system as described in *“The Oz Programming Interface”*.

Acknowledgements

Credit has to go to the following people:

- Martin Henz, Martin Müller, and Christian Schulte for writing ‘The Oz Standard Modules’, part of the Oz Documentation Series for DFKI Oz 2.0, from which this documented originated;
- Daniel Simon, for translating the original LaTeX sources into SGML;
- Leif Kornstaedt, for updating many parts of the document to Oz 3.0;
- Andreas Schoch, for the cover illustration.

Note that the persons responsible for the document as-is are Christian Schulte and Leif Kornstaedt. Please address any remarks to them.

2

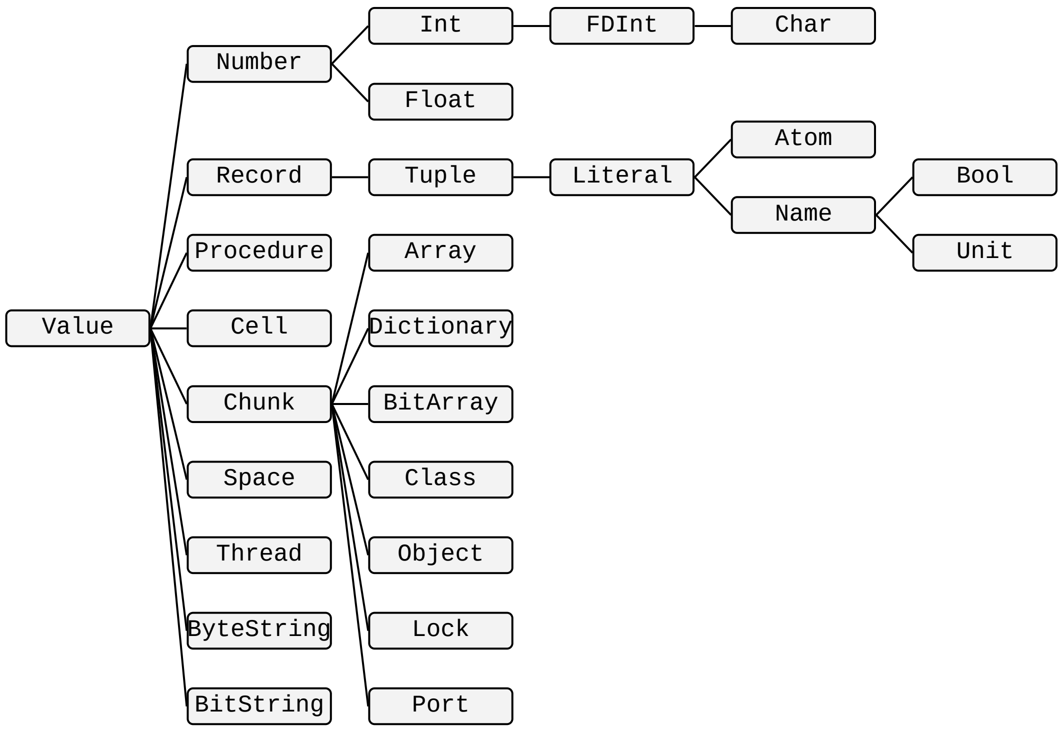
Type Structure and Description Format

2.1 Type Structure

Types are sets of values of the Oz universe which share a common structure and common operations. Types are divided into primary types and secondary types.

Primary Types

Figure 2.1 Standard Primary Types in Oz.



The primary types of Oz are depicted in Figure 2.1. Primary types provide a classification of values in the Oz Universe such that any two different subtypes of some primary type are disjoint. To check for a value to belong to a primary type, only its top-level constructor needs to be tested. Note that implementations of Oz are free to introduce more primary types (so called extensions) as immediate subtypes of either ‘value’ or ‘chunk’.

Numbers Numbers are either integers or floats.

Literals Literals are either atoms or names.

Tuples Tuples are special records whose features are the integers from 1 to *n* for some integer *n*, *n* `>= 0`.

Procedures Procedures are classified according to their arity. We speak about *n*-ary procedures.

Chunks Chunks serve to represent abstract data structures. They are defined similarly to records but provide only a restricted set of operations. This makes it possible to hide some or all of their features. Typical chunks are objects and classes, locks and ports, and arrays and dictionaries. There are chunks which do not belong to these types.

Secondary Types

Secondary types are additional types for which Oz provides standard procedures or modules.

Features Features are either integers or literals.

Pairs A pair is a record matching `'#'(_ _)`.

Lists A list is either the atom `nil` or a record matching `'|'(X Y)` (or, equivalently, `X|Y`) such that `Y` is again a list. Note that Oz allows cyclic lists which have an infinite unrolling.

Property Lists A property list is a list of pairs whose first component is a feature, i.e., a literal or an integer.

Virtual Strings Virtual strings are used to encode strings. Virtual strings contain atoms, strings, byte strings, integers, and floats, and are closed under tuple construction with the label `'#'`. For more details see Chapter 7.

2.2 Variable Status

Each node *X* in the store has exactly one of the following statuses: free, determined, future, or kinded.

Free Variable A variable `X` is free if the constraint store does not know anything about `X` apart from variable equalities `X = Y`.

Determined Variable A variable `X` is determined if the constraint store entails `X = N` for some number `N`, or if it entails `X = f(a1: _ ... an: _)` for some label `f` and the arity `[a1 ... an]`, or if it entails `X =` *Y* for some byte string, procedure, cell, chunk, space or thread `Y`.

Future A variable `X` is future if the constraint store entails `X = F` for some future `F`.

Kinded Variable A variable `X` is kinded if it is neither free nor determined nor future.

2.3 Description Format

Every standard procedure has an associated signature of the form

```
{Map +Xs +P ?Ys}
```

which specifies its arity, as well as types and modes of its arguments.

Types

The type of an argument is indicated by its name, using the abbreviations summarized in the following table:

Abbreviation	Type
`A`	atom
`B`	bool
`C`	chunk
`F`	float
`I`	integer
`K`	class
`L`	literal
`N`	name
`O`	object
`P`	procedure
`R`	record
`S`	string
`T`	tuple
`U`	unit
`V`	virtual string
`X Y Z`	value
`FI`	number
`LI`	feature
`AFI`	atom, float, or int
`PO`	unary procedure or object
`Xs`	lists of elements of type `X`

We use indices such as `R1` or `R2` to disambiguate several occurrences of arguments of the same type. We combine these abbreviations as in `FI` meaning ‘float or integer’ (i.e., number) or `LI` meaning ‘literal or integer’ (i.e., feature). We use the plural-‘s’ suffix to indicate lists of values of a certain type. For instance, `Is` stands for a list of integers. This suffix can be repeated to indicate lists of lists etc. Additionally, these arguments can be prefixed as in `LowI`, which indicates that the integer represents a lower bound.

Modes

Modes $+$, ?, none The arguments of procedures can have one of four modes which are indicated by a symbol ($+$, ?, none) attached to the arguments in the signature.

Modes indicate the synchronisation behaviour of a procedure. The application of a procedure `P` waits until its inputs ($+$) are determined. If the input arguments are well-typed, `P` will return outputs (?) of the specified types. Ill-typed input arguments will raise a type error exception. Types may be incompletely checked, especially those of of output arguments: This happens when a value needs not be completely processed to perform an operation, e.g., in `List.dropWhile`.

Occasionally, signatures of the base language will use the input mode $*$. Unless one uses any primitives from the constraint extensions, this is identical with $+$.

Naming Conventions

Aliases Some of the standard values are so frequent that a special name is provided for them. For example, `List.map` is also available as `Map`. The signature of `Map` occurring in the description of module `List` (see Section 6.3) says that the procedure `List.map` is available via the abbreviation `Map`.

Procedure Names Given the signature

{*procname* ...}

in the description of the module *module*, the procedure is available as:

- *procname*, provided *procname* is just a variable;
- *module*.*procname'*, where *procname'* can be obtained from *procname* by downcasing the first letter and deleting the string *module*.

For example, the test for lists is available as `IsList` and as `List.is`.

Infix Notation For very frequent procedures like arithmetic operations, there exists a convenient infix notation (see Chapter 11). By convention, the procedure names as they appear in the modules are the infix operators as atoms (i.e., wrapped in quotes). For instance, `Number.'+'` and `Value.'<'` have an infix notation using the operators `+` and `<`.

A Note on the Examples

Several examples used in this document assume a unary procedure `Browse` in the environment, which is supposed to display its argument value in a tool called browser.

3

Values

3.1 Values in General

The module `Value` contains procedures that can operate on many kinds of values.

`=`

`{Value.'=' X Y}`

unifies the values of the variables X and Y.

`==`

`{Value.'==' X Y ?B}`

tests whether X is equal to Y. The test may suspend.

`\=`

`{Value.'\\=' X Y ?B}`

tests whether X is not equal to Y. The test may suspend.

`toVirtualString`

`{Value.toVirtualString X +DepthI +WidthI ?VS}`

returns a virtual string describing the value of X. Note that this does not block on X. The values of `DepthI` and `WidthI` are used to limit output of records in depth and width respectively.

3.2 Variable Status

The following procedures allow to inspect a variable's status.

`IsFree`

`{Value.isFree X ?B}`

tests whether X is currently free.

`IsDet`

`{Value.isDet X ?B}`

tests whether X is determined.

`IsFuture`

`{Value.isFuture X ?B}`

tests whether X is a future.

`IsFailed`

`{Value.isFailed X ?B}`

tests whether X is a failed value.

`IsKinded`

`{Value.isKinded X ?B}`

tests whether X is currently kinded, i.e., is constrained but not yet determined. For example, `foo(a:12 ...)` is kinded because it is constrained to be a record, yet its arity is not yet known. Also, a non-determined finite domain variable is kinded: its type is known to be integer, but its value is not yet determined. Similarly for finite set variables.

`status`

`{Value.status X ?T}`

returns status and type information on X. If X is free, the atom `free` is returned. If X is a future, the atom `future` is returned. If X is a failed value, the atom `failed` is returned. If X is kinded, the tuple `kinded(`*Y*`)` is returned, where *Y* is bound to either the atoms `int`, `fset` or `record`, depending on the type of X. If X is determined, the tuple `det(`*Y*`)` is returned, where *Y* is bound to the atom as returned by `{Value.type X}`.

`type`

`{Value.type +X ?A}`

returns an atom describing the type of X. If X is of one of the standard primary types depicted in Figure 2.1 (except 'value'), then A is constrained to the most specific of `int`, `float`, `record`, `tuple`, `atom`, `name`, `procedure`, `cell`, `byteString`, `bitString`, `chunk`, `array`, `dictionary`, `bitArray`, `'class'`, `object`, `'lock'`, `port`, `space`, or `'thread'`. If any other atom is returned, this means that X is of no standard primary type, but an implementation-dependent extension.

3.3 Comparisons

This section collects procedures to compare integers with integers, floats with floats, and atoms with atoms. Atoms are compared lexically. Comparison between values of different types is not allowed and an attempt to do so will raise a run-time error.

`=<`

`{Value.'=<' +AFI1 +AFI2 ?B}`

tests whether AFI1 is less than or equal to AFI2.

`<`

`{Value.'<' +AFI1 +AFI2 ?B}`

tests whether AFI1 is less than AFI2.

`>=`

`{Value.'>=' +AFI1 +AFI2 ?B}`

tests whether AFI1 is greater than or equal to AFI2.

>

```
{Value.'>' +AFI1 +AFI2 ?B}
```

tests whether AFI1 is greater than AFI2.

Max

```
{Value.max +AFI1 +AFI2 ?AFI3}
```

returns the maximum of AFI1 and AFI2.

Min

```
{Value.min +AFI1 +AFI2 ?AFI3}
```

returns the minimum of AFI1 and AFI2.

4

Numbers

4.1 Numbers in General

The module `Number` contains procedures operating on numbers. Numbers in Oz are either integers or floats. The following arithmetic procedures are defined both on integers and on floats; however, there is no implicit conversion. If one input argument is a float and the other an integer, a type error is raised.

IsNumber

```
{Number.is +X ?B}
```

tests whether `X` is a number.

+

```
{Number.'+' +FI1 +FI2 ?FI3}
```

returns the sum of `FI1` and `FI2`.

-

```
{Number.'-' +FI1 +FI2 ?FI3}
```

returns the difference of `FI1` and `FI2`.

*****

```
{Number.'*' +FI1 +FI2 ?FI3}
```

returns the product of `FI1` and `FI2`.

~

```
{Number.'~' +FI1 ?FI2}
```

returns the negation of `FI1`.

Pow

```
{Number.pow +FI1 +FI2 ?FI3}
```

returns `FI1` to the power of `FI2`.

Abs

```
{Number.abs +FI1 ?FI2}
```

returns the absolute value of `FI1`.

4.2 Integers

The module `Int` contains procedures operating on integers.

`IsInt`

```
{Int.is +X ?B}
```

tests whether X is an integer.

`IsNat`

```
{Int.isNat +I ?B}
```

tests whether `I` is a natural number, i.e., an integer greater than or equal to `0`.

`IsOdd`

```
{Int.isOdd +I ?B}
```

tests whether `I` is an odd integer.

`IsEven`

```
{Int.isEven +I ?B}
```

tests whether `I` is an even integer.

`div`

```
{Int.'div' +I1 +I2 ?I3}
```

returns `I1` integer-divided by `I2`, rounding towards `0`. `Int.'div'` can be defined as follows:

```
local
   fun {Div I1 I2}
      case I1 < I2 then 0 else 1 + {Div I1 - I2 I2} end
   end
in
   fun {Int.'div' I1 I2}
      {Div {Abs I1} {Abs I2}} *
      case I1 * I2 >= 0 then 1 else ~1 end
   end
end
```

`mod`

```
{Int.'mod' +I1 +I2 ?I3}
```

returns `I1` modulo `I2`. `Int.'mod'` can be defined as follows:

```
fun {Int.'mod' I1 I2}
   I1 - I2 * (I1 div I2)
end
```

`IntToFloat`

```
{Int.toFloat +I ?F}
```

returns the float closest to the integer `I`.

`IntToString`

```
{Int.toString +I ?S}
```

returns the string describing the integer `I` in Oz concrete syntax.

4.3 Floats

The module `Float` contains procedures operating on floating point numbers.

`IsFloat`

```
{Float.is +X ?B}
```

tests whether `X` is a float.

`/`

```
{Float.'/' +F1 +F2 ?F3}
```

returns `F1` divided by `F2`.

`Exp`

```
{Float.exp +F1 ?F2}
```

returns `F1` to the power of *e*.

`Log`

```
{Float.log +F1 ?F2}
```

returns the logarithm to the base *e* of `F1`.

`Sqrt`

```
{Float.sqrt +F1 ?F2}
```

returns the square root of `F1`.

`Ceil`

```
{Float.ceil +F1 ?F2}
```

returns the ceiling of `F1` (rounding towards positive infinity).

`Floor`

```
{Float.floor +F1 ?F2}
```

returns the floor of `F1` (rounding towards negative infinity).

`Round`

```
{Float.round +F1 ?F2}
```

returns the integral value closest to `F1`. If there are two candidates, `F1` is rounded to the closest even integral value, e.g., `{Float.round 1.5}` and `{Float.round 2.5}` both return `2.0`.

`Sin`

```
{Float.sin +F1 ?F2}
```

returns the sine of `F1`.

`Cos`

```
{Float.cos +F1 ?F2}
```

returns the cosine of `F1`.

`Tan`

```
{Float.tan +F1 ?F2}
```

returns the tangent of `F1`.

Asin

`{Float.asin +F1 ?F2}`

returns the arc sine of `F1`.

Acos

`{Float.acos +F1 ?F2}`

returns the arc cosine of `F1`.

Atan

`{Float.atan +F1 ?F2}`

returns the arc tangent of `F1`.

Atan2

`{Float.atan2 +F1 +F2 ?F3}`

returns the principal value of the arc tangent of `F1 / F2`, using the signs of both arguments to determine the quadrant of the return value. An error exception may (but needs not) be raised if both arguments are zero.

Float.sinh

`{Float.sinh +F1 ?F2}`

returns the hyperbolic sine of `F1`.

Float.cosh

`{Float.cosh +F1 ?F2}`

returns the hyperbolic cosine of `F1`.

Float.tanh

`{Float.tanh +F1 ?F2}`

returns the hyperbolic tangent of `F1`.

Float.asinh

`{Float.asinh +F1 ?F2}`

returns the inverse hyperbolic sine of `F1`.

Float.acosh

`{Float.acosh +F1 ?F2}`

returns the inverse hyperbolic cosine of `F1`.

Float.atanh

`{Float.atanh +F1 ?F2}`

returns the inverse hyperbolic tangent of `F1`.

FloatToInt

`{Float.toInt +F ?I}`

returns the integer closest to float `F`. If there are two candidates, `F` is rounded to the closest even integer, e.g., `{FloatToInt 1.5}` and `{FloatToInt 2.5}` both return `2`.

In the current implementation, the value is converted through a signed 32-bit integer, so this function is bogus when applied to large floats.

`FloatToString`

```
{Float.toString +F ?S}
```

returns the string describing the float `F` in Oz concrete syntax.

5

Literals

5.1 Literals in General

The module `Literal` contains procedures operating on literals, i.e., atoms and names.

`IsLiteral`

```
{Literal.is +X ?B}
```

tests whether `X` is a literal.

5.2 Atoms

The module `Atom` contains procedures operating on atoms.

`IsAtom`

```
{Atom.is +X ?B}
```

tests whether `X` is atom.

`AtomToString`

```
{Atom.toString +A ?S}
```

binds `S` to the string (list of characters) representing atom `A` according to ISO 8859-1. See also `String.toAtom`.

For example,

```
{AtomToString 'abc'}
```

yields as output `[97 98 99]`.

5.3 Names

The module `Name` contains procedures operating on names.

`IsName`

```
{Name.is +X ?B}
```

tests whether `X` is a name.

`NewName`

```
{Name.new ?N}
```

Creates a new name and binds `N` to it.

5.4 Truth Values

The module `Bool` contains procedures operating on the truth values **`true`** and **`false`**, which denote names.

`IsBool`

```
{Bool.is +X ?B}
```

tests whether `X` is **`true`** or **`false`**.

`Not`

```
{Bool.'not' +B1 ?B2}
```

returns the negation of truth value `B1`.

`And`

```
{Bool.and +B1 +B2 ?B3}
```

returns the conjunction of truth values `B1` and `B2`. Note that `And` is different from conditional conjunction available via the keyword **`andthen`** in that it always evaluates its second argument.

For instance, **`false andthen`** `P` reduces without reducing application of `P`, whereas reduction of `{And` **`false`** `P}` always applies `P`.

`Or`

```
{Bool.'or' +B1 +B2 ?B3}
```

returns the disjunction of truth values `B1` and `B2`. Note that `Or` is different from conditional disjunction available via the keyword **`orelse`** in that it always evaluates its second argument.

For instance, **`true orelse`** `P` reduces without reducing application of `P`, whereas reduction of `{Or` **`true`** `P}` always applies `P`.

5.5 The Value Unit

The module `Unit` contains procedures operating on the value available as **`unit`**, which denotes a name.

`IsUnit`

```
{Unit.is +X ?B}
```

tests whether `X` is **`unit`**.

6

Records, Tuples, and Lists

This chapter describes procedures to be used with records in general and special kinds of records, namely tuples and lists.

6.1 Records in General

The module `Record` contains procedures operating on records.

Procedures that iterate over the subtrees of a record operate in ascending order as specified for `Arity`.

.

```
{Value.'.' +RC +LI X}
```

returns the field `X` of `RC` at feature `LI`.

HasFeature

```
{Value.hasFeature +RC +LI ?B}
```

tests whether `RC` has feature `LI`.

CondSelect

```
{Value.condSelect +RC +LI X Y}
```

returns the field of `RC` at feature `LI`, if `RC` has feature `LI`. Otherwise, return `X`.

IsRecord

```
{Record.is +X ?B}
```

tests whether `X` is a record.

MakeRecord

```
{Record.make +L +LIs ?R}
```

returns a new record with label `L`, features `LIs`, and fresh variables at every field. All elements of `LIs` must be pairwise distinct, else an error exception is raised.

For example, `{MakeRecord L A R}` waits until `L` is bound to a literal, say `b`, and `A` is bound to a list of literals and integers, say `[c d 1]`, and then binds `R` to `b(_ c: _ d: _)`.

clone

```
{Record.clone +R1 ?R2}
```

returns a record `R2` with the same label and features as `R1` and fresh variables at every field.

`Label`

```
{Record.label +R ?L}
```

returns the label of `R` in `L`.

`Width`

```
{Record.width +R ?I}
```

returns the width of `R` in `I`.

`Arity`

```
{Record.arity +R ?LIs}
```

returns the arity `LIs` of `R`. The arity of `R` is the list of its features, beginning with all integer features in ascending order, followed by the literal features. The atomic literals occur in ascending order interspersed with names.

For example, `{Arity a(nil 7 c: 1 b: c)}` yields `[1 2 b c]` as output.

`Adjoin`

```
{Record.adjoin +R1 +R2 ?R3}
```

returns the result of adjoining `R2` to `R1`. Note that features and label in R2 take precedence over R1.

For example,

```
{Adjoin a(a b c: 1) b(4 b: 3 c: 2)}
```

yields the record `b(4 b b: 3 c: 2)` as output.

`AdjoinAt`

```
{Record.adjoinAt +R1 +LI X ?R2}
```

binds `R2` to the result of adjoining the field `X` to `R1` at feature `LI`.

For example,

```
{AdjoinAt a(a c: 1) 2 b}
```

yields `a(a b c: 1)` as output, whereas

```
{AdjoinAt a(a c: 1) c b}
```

yields `a(a c: b)` as output.

`AdjoinList`

```
{Record.adjoinList +R1 +Ts ?R2}
```

binds `R2` to the result of adjoining to `R1` all entries of `Ts`, a finite list of pairs whose first components are literals or integers, representing features. Features further to the right overwrite features further to the left.

For example,

```
{AdjoinList a(b:1 c:2) [d#3 c#4 d#5]}
```

yields `a(b: 1 c: 4 d: 5)` as output.

subtract

```
{Record.subtract +R1 +LI ?R2}
```

If `R1` has feature `LI`, returns record `R1` with feature `LI` removed. Otherwise, returns `R1`.

subtractList

```
{Record.subtractList +R1 +LIs ?R2}
```

Returns record `R1` with all features in `LIs` removed.

For example,

```
{Record.subtractList f(jim: 1 jack: 2 jesse: 4) [jesse jim]}
```

returns the record `f(jack: 2)`.

zip

```
{Record.zip +R1 +R2 +P ?R3}
```

Given two records `R1` and `R2` and a ternary procedure `P`, `R3` is bound to a record with the same label as `R1` and those features which are common to `R1` and `R2`. Features appearing only in one of the records are silently dropped. Each fields `X` of `R3` is computed by applying `{P R1.X R2.X R3.X}`.

For example,

```
{Record.zip
 f(jim: 1 jack: 2 jesse: 4)
 g(jim: a jack: b joe: c)
 fun {$ X Y} X#Y end}
```

yields as output the record `f(jim: 1#a jack: 2#b)`.

toList

```
{Record.toList +R ?Xs}
```

binds `Xs` to list of all fields of `R` in the order as given by `Arity` (which see).

For example,

```
{Record.toList f(a a: 2 b: 3)}
```

yields `[a 2 3]` as output.

toListInd

```
{Record.toListInd +R ?Ts}
```

binds `Ts` to the property list that contains the feature-field pairs of `R` in the order as given by `Arity` (which see).

For example,

```
{Record.toListInd f(a a: 2 b: 3)}
```

yields `[1#a a#2 b#3]` as output.

toDictionary

```
{Record.toDictionary +R ?Dictionary}
```

returns a dictionary `Dictionary` whose keys and their entries correspond to the features and their fields of `R`.

All of the following procedures are provided in two versions. The so-called *index* version passes to the procedures an additional index as first actual argument. The index is an integer or a literal giving the feature of the field currently processed.

`map`

```
{Record.map +R1 +P ?R2}
```

returns a record with same label and arity as `R1`, whose fields are computed by applying the binary procedure `P` to all fields of `R1`.

For example,

```
{Record.map a(12 b: 13 c: 1) IntToFloat}
```

yields the record `a(12.0 b: 13.0 c: 1.0)` as output.

`mapInd`

```
{Record.mapInd +R1 +P ?R2}
```

is similar to `Record.map`, but the ternary procedure `P` is applied with the index as first actual argument.

For example,

```
{Record.mapInd a(1: d 3: a f: e) fun {$ I A} A(I) end}
```

yields the record `a(1: d(1) 3: a(3) f: e(f))` as output.

`foldL`

```
{Record.foldL +R +P X ?Y}
```

`foldR`

```
{Record.foldR +R +P X ?Y}
```

Used for folding the fields of `R` by applying the ternary procedure `P`.

Suppose that `R` has the arity `[F1 ... Fn]`. Applying the left folding procedure `{Record.foldL R P Z Out}` reduces to

```
{P ... {P {P Z R.F1} R.F2 ... R.Fn Out}
```

The first actual argument of `P` is the accumulator in which the result of the previous application or the start value `Z` is passed. The second actual argument is a field of `R`.

Besides the left folding procedure there exists a right folding variant. The application `{Record.foldR R P Z Out}` reduces to

```
{P R.F1 {P R.F2 ... {P R.Fn Z} ... Out}
```

The first actual argument of `P` is a field of `R`; the second actual argument is the accumulator in which the result of the previous application or the start value `Z` is passed.

For example,

```
{Record.foldL a(3 a: 7 b: 4) fun {$ Xr X} X|Xr end nil}
```

yields the output `[4 7 3]`, whereas

```
{Record.foldR a(3 a: 7 b: 4) fun {$ X Xr} X|Xr end nil}
```

yields the output `[3 7 4]`.

foldLInd

```
{Record.foldLInd +R +P X ?Y}
```

foldRInd

```
{Record.foldRInd +R +P X ?Y}
```

are similar to `Record.foldL` and `Record.foldR`, but the 4-ary procedure `P` is applied with the current index as first actual argument.

forAll

```
{Record.forAll +R +PO}
```

applies the unary procedure or object `PO` to each field of `R`.

Suppose that the arity of `R` is `[F1 ... Fn]`. The application `{Record.forAll R P}` reduces to the sequence of statements

```
{P R.F1} ... {P R.Fn}
```

For example,

```
{Record.forAll O1#O2#O3 proc {$ O} {O do()} end}
```

sends the message `do()` to the objects `O1`, `O2`, and `O3`.

forAllInd

```
{Record.forAllInd +R +P}
```

is similar to `Record.forAll`, but the binary procedure `P` is applied with the current index as first actual argument.

For example, assuming `O1`, `O2`, and `O3` are objects,

```
{Record.forAllInd a(do: O1 stop: O2 run: O3)
 proc {$ M O} {O M} end}
```

sends the message `do` to the object `O1`, the message `stop` to `O2`, and the message `run` to `O3`.

all

```
{Record.all +R +P ?B}
```

some

```
{Record.some +R +P ?B}
```

tests whether the unary boolean function `P` yields `true` when applied to all fields resp. some field of `R`. Stops at the first field for which `P` yields `false` resp. `true`. The fields are tested in the order given by `Arity` (which see).

allInd

```
{Record.allInd +R +P ?B}
```

someInd

```
{Record.someInd +R +P ?B}
```

is similar to `Record.all` resp. `Record.some`, but the binary boolean function `P` is applied with the current index as first actual argument.

filter

```
{Record.filter +R1 +P ?R2}
```

partition

```
{Record.partition +R1 +P ?R2 ?R3}
```

`Record.filter` computes a record `R2` which contains all the features and fields of the record `R1` for which the unary boolean procedure `P` applied to the field yields **`true`**. `Record.partition` works similarly, but returns in `R3` a record with all remaining fields of `R1`.

For example, the application

```
{Record.partition a(1 4 7 a: 3 b: 6 c: 5) IsOdd ?R2 ?R3}
```

returns `a(1: 1 3: 7 a: 3 c: 5)` in `R2` and `a(2: 4 b: 6)` in `R3`.

filterInd

```
{Record.filterInd +R1 +P ?R2}
```

partitionInd

```
{Record.partitionInd +R1 +P ?R2 ?R3}
```

are similar to `Record.filter` and `Record.partition`, but the binary boolean function `P` is applied with the current index as first actual argument.

takeWhile

```
{Record.takeWhile +R1 +P ?R2}
```

dropWhile

```
{Record.dropWhile +R1 +P ?R3}
```

takeDropWhile

```
{Record.takeDropWhile +R2 +P ?R2 ?R3}
```

While `Record.filter` selects all fields and features of a record which satisfy a certain condition, the procedure `Record.takeWhile` selects only the starting sequence of features and fields which fulfill this condition. The procedure `Record.dropWhile` is dual: It computes a record with the remaining features and fields. `Record.takeWhileDrop` computes both records.

For example,

```
{Record.takeWhile a(1 4 7 a: 3 b: 6 c: 5) IsOdd}
```

yields as output `a(1)`, whereas

```
{Record.dropWhile a(1 4 7 a: 3 b: 6 c: 5) IsOdd}
```

yields `a(2: 4 3: 7 a: 3 b: 6 c: 5)` as output. Both records can be computed simultaneously by

```
{Record.takeDropWhile  a(1 4 7 a: 3 b: 6 c: 5) IsOdd ?R2 ?R3}
```

takeWhileInd

```
{Record.takeWhileInd +R1 +P ?R2}
```

dropWhileInd

```
{Record.dropWhileInd +R1 +P ?R3}
```

takeDropWhileInd

```
{Record.takeDropWhileInd +R1 +P ?R2 ?R3}
```

are similar to `Record.takeWhile`, `Record.dropWhile` and `Record.takeDropWhile` but the binary boolean function `P` is applied with the current index as first actual argument.

waitOr

```
{Record.waitOr +R ?LI}
```

blocks until at least one field of +`R` is determined. Returns the feature `LI` of a determined field. Raises an exception if `R` is not a proper record, that is, if `R` is a literal.

For example,

```
{Record.waitOr a(_ b: 1)}
```

returns `b` while

```
{Record.waitOr a(2 b: _)}
```

returns `1`, and

```
{Record.waitOr a(_ b: _)}
```

blocks.

6.2 Tuples

The module `Tuple` contains procedures operating on tuples.

IsTuple

```
{Tuple.is +X ?B}
```

tests whether `X` is a tuple.

MakeTuple

```
{Tuple.make +L +I ?T}
```

binds `T` to new tuple with label `L` and fresh variables at features `1` through `I`. `I` must be non-negative, else an error exception is raised.

For example, `{MakeTuple L N T}` waits until `L` is bound to a literal, say `b`, and `N` is bound to a number, say `3`, whereupon `T` is bound to `b(_ _ _)`.

toArray

```
{Tuple.toArray +T ?A}
```

returns an array with bounds between `1` and `{Width T}`, where the elements of the array are the subtrees of `T`.

`append`

```
{Tuple.append +T1 +T2 ?T3}
```

returns a tuple with same label as `T2`. Given that `T1` has width *i* and `T2` has width *j*, `T3` will have width *i* `+` *j*, and the first *i* fields of `T3` will be the same as the fields of `T1` in their original order, and the fields *i* `+ 1` through *i* `+` *j* will be the same as the fields of `T2` in their original order.

6.3 Lists

The module `List` contains procedures operating on lists.

`IsList`

```
{List.is +X ?B}
```

tests whether `X` is a list. Diverges if `X` is an infinite list.

`MakeList`

```
{List.make +I ?Xs}
```

returns a list of length `I`. All elements are fresh variables.

`Append`

```
{List.append +Xs Y ?Zs}
```

binds `Zs` to the result of appending `Y` to `Xs`. `Y` needs not be a list. However, `Zs` is only a proper list, if also `Y` is a proper list.

For example,

```
{Append [1 2] [3 4]}
```

returns the list `[1 2 3 4]`, whereas

```
{Append 1|2|nil 3|4}
```

returns `1|2|(3|4)` which is not a proper list, since `3|4` is not a proper list.

`Member`

```
{List.member X +Ys ?B}
```

tests whether `X` is equal (in the sense of `==`) to some element of `Ys`. Note: all other procedures of the `List` module that operate on a list take it as their first argument. `Member` is the only exception (for historical reasons).

`Length`

```
{List.length +Xs ?I}
```

returns the length of `Xs`.

`Nth`

```
{List.nth +Xs +I ?Y}
```

returns the `I`th element of `Xs` (counting from `1`).

subtract

```
{List.subtract +Xs Y ?Zs}
```

binds `Zs` to `Xs` without the leftmost occurrence of `Y` if there is one.

sub

```
{List.sub +Xs +Ys ?B}
```

tests whether `Xs` is a sublist of `Ys`, i.e., whether it contains all elements of `Xs` in the same order as `Xs` but not necessarily in succession.

For example, `[a b]` is a sublist of both `[1 a b 2]` and `[1 a 2 b 3]`, but not of `[b a]`.

Reverse

```
{List.reverse +Xs ?Ys}
```

returns the elements of `Xs` in reverse order.

Sort

```
{List.sort +Xs +P ?Ys}
```

binds `Ys` to the result of sorting `Xs` using the ordering `P`. `Sort` is implemented using the mergesort algorithm.

For example,

```
{Sort [c d b d a] Value.'<'}
```

returns the list `[a b c d d]`.

Merge

```
{List.merge +Xs +Ys +P ?Zs}
```

binds `Zs` to the result of merging `Xs` and `Ys` using the ordering `P`. The lists `Xs` and `Ys` must be sorted.

Flatten

```
{List.flatten +Xs ?Ys}
```

binds `Ys` to the result of flattening `Xs`, i.e., of concatenating all sublists of `Xs` recursively.

withTail

```
{List.withTail +I Y ?Xs}
```

returns a list with at least `I` elements whose rest is `Y` (which needs not be a list). The first `I` elements are fresh variables.

For example, `{List.withTail 2 [a b]}` returns `[_ _ a b]`.

number

```
{List.number +FromI +ToI +StepI ?Xs}
```

returns a list with elements from `FromI` to `ToI` with step `StepI`.

For example, `{List.number 1 5 2}` returns `[1 3 5]`, `{List.number 5 1 2}` yields the list `nil`, and `{List.number 5 0 -2}` yields the list `[5 3 1]`.

take

```
{List.take +Xs +I ?Ys}
```

returns the list that contains the first `I` elements of `Xs`, or `Xs` if it is shorter.

`drop`

```
{List.drop +Xs +I ?Ys}
```

returns the list `Xs` with the first `I` elements removed, or to `nil` if it is shorter.

`takeDrop`

```
{List.takeDrop +Xs +I ?Ys ?Zs}
```

binds `Ys` to `{List.take Xs I}` and `Zs` to `{List.drop Xs I}`.

`last`

```
{List.last +Xs ?Y}
```

returns the last element of `Xs`. Raises an error exception if `Xs` is `nil`.

`toTuple`

```
{List.toTuple +L +Xs ?T}
```

binds `T` to a tuple with label `L` that contains the elements of `Xs` as subtrees in the given order.

For example,

```
{List.toTuple '#' [a b c]}
```

returns `a#b#c`.

`toRecord`

```
{List.toRecord +L +Ts ?R}
```

binds `R` to a record with label `L` whose subtrees are given by the property list `Ts`: For every element `L`*i*`#X`*i* of `Xs`, `R` has a field `X`*i* at feature `L`*i*. The features in the property list must be pairwise distinct, else an error exception is raised.

For example,

```
{List.toRecord f [a#1 b#2 c#3]}
```

returns `f(a: 1 b: 2 c: 3)`.

`zip`

```
{List.zip +Xs +Ys +P ?Zs}
```

returns the list of all elements `Z`*i* computed by applying `{P X`*i* `Y`*i*`}`, where `X`*i* is the *i*th element of `Xs` and `Y`*i* the *i*th element of `Ys`. The two input lists must be of equal length, else an error exception is raised.

For example,

```
{List.zip [1 6 3] [4 5 6] Max}
```

returns the list `[4 6 6]`.

`isPrefix`

```
{List.isPrefix +Xs +Ys ?B}
```

tests whether `Xs` is a prefix of `Ys`. Given that `Xs` has length *i*, it is a prefix of `Ys` if `Ys` has at least length *i* and the first *i* elements of `Ys` are equal to the corresponding elements of `Xs`.

All of the following procedures exist in two versions. The so-called *index* version passes to the procedures an additional index as first actual argument. The index is an integer giving the position of the list element currently processed (counting from `1`).

`Map`

```
{List.map +Xs +P ?Ys}
```

returns the list obtained by applying `P` to each element of `Xs`.

For example,

```
{Map [12 13 1] IntToFloat}
```

returns `[12.0 13.0 1.0]`.

`mapInd`

```
{List.mapInd +Xs +P ?Ys}
```

is similar to `Map`, but the ternary procedure `P` is applied with the index as first actual argument.

For example,

```
{List.mapInd [d a e] fun {$ I A} I#A end}
```

yields the list `[1#d 2#a 3#e]` as output.

`FoldL`

```
{List.foldL +Xs +P X ?Y}
```

`FoldR`

```
{List.foldR +Xs +P X ?Y}
```

Used for folding the elements of `Xs` by applying a ternary procedure `P`.

Application of the left folding procedure `{FoldL [X1 ... Xn] P Z Out}` reduces to

```
{P ... {P {P Z X1} X2} ... Xn Out}
```

The first actual argument of `P` is the accumulator in which the result of the previous application or the start value `Z` is passed. The second actual argument is an element of `Xs`.

Besides the left folding procedure there exists a right folding variant. The application `{FoldR [X1 ... Xn] P Z Out}` reduces to

```
{P X1 {P X2 ... {P Xn Z} ...} Out}
```

The first actual argument of `P` is an element of `Xs`. The second actual argument of `P` is the accumulator in which the result of the previous application or the start value `Z` is passed.

For example,

```
{FoldL [b c d] fun {$ X Y} f(X Y) end a}
```

returns `f(f(f(a b) c) d)`, whereas

```
{FoldR [b c d] fun {$ X Y} f(X Y) end a}
```

returns `f(b f(c f(d a)))`.

`foldLInd`

```
{List.foldLInd +Xs +P X ?Y}
```

`foldRInd`

```
{List.foldRInd +Xs +P X ?Y}
```

are similar to `FoldL` and `FoldR`, but the `4`-ary procedure `P` is applied with the current index as first actual argument.

`FoldLTail`

```
{List.foldLTail +Xs +P X ?Y}
```

`FoldRTail`

```
{List.foldRTail +Xs +P X ?Y}
```

Used for folding all non-`nil` tails of `Xs` by applying a ternary procedure `P`, i.e., application of the left folding procedure

```
{FoldLTail [X1 ... Xn] P Z Out}
```

reduces to

```
{P ... {P {P Z [X1 ... Xn]} [X2 ... Xn]} ... Xn] Out}
```

The right folding procedure is analogous.

`foldLTailInd`

```
{List.foldLTailInd +Xs +P X ?Y}
```

`foldRTailInd`

```
{List.foldRTailInd +Xs +P X ?Y}
```

are similar to `FoldLTail` and `FoldRTail`, but the `4`-ary procedure `P` is applied with the current index as first actual argument.

`ForAll`

```
{List.forAll +Xs +PO}
```

applies the unary procedure or object `PO` to each element of `Xs`, i.e., the application

```
{ForAll [X1 ... Xn] P}
```

reduces to the sequence of statements

```
{P X1} ... {P Xn}
```

For example,

```
{ForAll [O1 O2 O3] proc {$ O} {O do()} end}
```

sends the message `do()` to the objects `O1`, `O2`, and `O3`.

`forAllInd`

```
{List.forAllInd +Xs +P}
```

is similar to `ForAll`, but the binary procedure `P` is applied with the current index as first actual argument.

For example, assuming `O1`, `O2`, and `O3` are objects, the following statement sends the message `do(1)` to the object `O1`, the message `do(2)` to `O2`, and the message `do(3)` to `O3`:

```
{List.forAllInd [O1 O2 O3]
 proc {$ I O} {O do(I)} end}
```

`ForAllTail`

```
{List.forAllTail +Xs +PO}
```

applies the unary procedure or object `PO` to each non-`nil` tail of `Xs`, i.e., the application

```
{ForAllTail [X1 ... Xn] P}
```

reduces to the sequence of statements

```
{P [X1 ... Xn]} {P [X2 ... Xn]} ... {P [Xn]}
```

`forAllTailInd`

```
{List.forAllTailInd +Xs +P}
```

is similar to `ForAllTail`, but the binary procedure `P` is applied with the current index as first actual argument.

`All`

```
{List.all +Xs +P ?B}
```

`Some`

```
{List.some +Xs +P ?B}
```

tests whether the unary boolean function `P` yields **`true`** when applied to all elements resp. some element of `Xs`. Stops at the first element for which `P` yields **`false`** resp. **`true`**.

`allInd`

```
{List.allInd +Xs +P ?B}
```

`someInd`

```
{List.someInd +Xs +P ?B}
```

are similar to `All` and `Some`, but the binary boolean function `P` is applied with the current index as first actual argument.

`Filter`

```
{List.filter +Xs +P ?Ys}
```

`partition`

```
{List.partition +Xs +P ?Ys ?Zs}
```

`Filter` returns a list of the elements of `Xs` for which the application of the unary boolean function `P` yields **`true`**, where the ordering is preserved. `List.partition` works similarly, but additionally returns in Zs a list of all remaining elements of Xs, where the ordering is preserved as well.

For example, the application

```
{List.partition [1 4 2 3 6 5] IsOdd Ys Zs}
```

returns `[1 3 5]` in `Ys` and `[4 2 6]` in `Zs`.

`filterInd`

```
{List.filterInd +Xs +P ?Ys}
```

`partitionInd`

```
{List.partitionInd +Xs +P ?Ys ?Zs}
```

are similar to `Filter` and `List.partition`, but the binary boolean function P is applied with the current index as first actual argument.

`takeWhile`

```
{List.takeWhile +Xs +P ?Ys}
```

`dropWhile`

```
{List.dropWhile +Xs +P ?Ys}
```

`takeDropWhile`

```
{List.takeDropWhile +Xs +P ?Ys ?Zs}
```

While `Filter` selects all elements of a list which satisfy a certain condition, the procedure `List.takeWhile` selects only the starting sequence of elements which fulfill this condition. The procedure `List.dropWhile` is dual: It returns the remainder of the list. For convenience, `List.takeDropWhile` combines the functionality from both `List.takeWhile` and `List.dropWhile`.

For example, the application

```
{List.takeWhile [1 4 2 3 6 5] IsOdd Ys}
```

returns `[1]` in `Ys`, whereas

```
{List.dropWhile [1 4 2 3 6 5] IsOdd Zs}
```

returns `[4 2 3 6 5]` in `Ys`.

```
{List.takeDropWhile [1 4 2 3 6 5] IsOdd Ys Zs}
```

combines both.

`takeWhileInd`

```
{List.takeWhileInd +Xs +P ?Ys}
```

`dropWhileInd`

```
{List.dropWhileInd +Xs +P ?Ys}
```

`takeDropWhileInd`

```
{List.takeDropWhileInd +Xs +P ?Ys ?Zs}
```

are similar to `List.takeWhile`, `List.dropWhile` and `List.takeDropWhile` but the binary boolean function P is applied with the current index as first actual argument.

Text

This chapter describes modules for handling data encoding textual information. Characters are encoded as integers. Strings are lists of characters. Virtual Strings are atoms, strings, byte strings, integers, and floats closed under virtual concatenation encoded by tuples with label `'#'`.

For example,

```
"Contains "#also#" numbers: "#(1#' '#2.045)
```

is a virtual string representing the string

```
"Contains also numbers: 1 2.045"
```

7.1 Characters

The module `Char` contains procedures operating on characters. Characters are integers between `0` and `255`, used for building strings. For the encoding of characters by integers, we use the ISO 8859-1 standard [1]. The functionality provided by this module is similar to the `ctype.h` module of ANSI C, see for instance [2].

The procedures described herein can be used to compute with strings by using the generic procedures of the list module (see Section 6.3). For example,

```
{Filter "r E!m O\nv7E" Char.isAlpha}
```

keeps only the letters of the given string and returns the string `"rEmOvE"`.

IsChar

```
{Char.is +X ?B}
```

tests whether `X` is a character, i.e., an integer between `0` and `255` inclusively.

isLower

```
{Char.isLower +Char ?B}
```

tests whether `Char` encodes a lower-case letter.

isUpper

```
{Char.isUpper +Char ?B}
```

tests whether `Char` encodes an upper-case letter.

`isDigit`

```
{Char.isDigit +Char ?B}
```

tests whether `Char` encodes a digit.

`isSpace`

```
{Char.isSpace +Char ?B}
```

tests whether `Char` encodes a white space character, i.e., either a space, a form feed (`&\f`), a newline (`&\n`), a carriage return (`&\r`), a tab (`&\t`), a vertical tab (`&\v`) or a non-breaking space (`&\240`).

`isPunct`

```
{Char.isPunct +Char ?B}
```

tests whether `Char` encodes a punctuation character, i.e., a visible character, which is not a space, a digit, or a letter.

`isCntrl`

```
{Char.isCntrl +Char ?B}
```

tests whether `Char` encodes a control character.

`isAlpha`

```
{Char.isAlpha +Char ?B}
```

tests whether `Char` encodes a letter.

`isAlNum`

```
{Char.isAlNum +Char ?B}
```

tests whether `Char` encodes a letter or a digit.

`isGraph`

```
{Char.isGraph +Char ?B}
```

tests whether `Char` encodes a visible character.

`isPrint`

```
{Char.isPrint +Char ?B}
```

tests whether `Char` is a visible character or either the space or non-breaking space character.

`isXDigit`

```
{Char.isXDigit +Char ?B}
```

tests whether `Char` is a hexadecimal digit.

`type`

```
{Char.type +Char ?A}
```

maps `Char` to its simple type `A`, i.e., one of the atoms `lower`, `upper`, `digit`, `space`, `punct`, or `other`.

`toLower`

```
{Char.toLower +Char1 ?Char2}
```

returns the corresponding lower-case letter if `Char1` is an upper-case letter, otherwise `Char1` itself.

`toUpper`

```
{Char.toUpper +Char1 ?Char2}
```

returns the corresponding upper-case letter if `Char1` is a lower-case letter, otherwise `Char1` itself.

`toAtom`

```
{Char.toAtom +Char ?A}
```

maps `Char` to the corresponding atom `A`. If `Char` is zero, `A` will be the empty atom `''`.

7.2 Strings

The module `String` contains procedures operating on strings. Strings are lists whose elements are characters (see Section 7.1).

`IsString`

```
{String.is +X ?B}
```

tests whether `X` is string.

`StringToAtom`

```
{String.toAtom +S ?A}
```

converts a string `S` to an atom `A`. `S` must not contain NUL characters. This is the inverse of `Atom.toString` (which see).

`isAtom`

```
{String.isAtom +S ?B}
```

tests whether the string `S` can be converted to an atom.

`StringToInt`

```
{String.toInt +S ?I}
```

converts a string `S` to an integer `I`, according to Oz concrete syntax. See also `IntToString`.

`isInt`

```
{String.isInt +S ?B}
```

tests whether the string `S` can be converted to an integer.

`StringToFloat`

```
{String.toFloat +S ?F}
```

converts a string `S` to a float `F`, according to Oz concrete syntax. See also `FloatToString`.

`isFloat`

```
{String.isFloat +S ?B}
```

tests whether the string `S` can be converted to a float.

`token`

```
{String.token +S1 X ?S2 ?S3}
```

splits the string S1 into two substrings S2 and S3. S2 will contain all characters before the first occurence of X, S3 all remaining characters with X excluded. If X does not occur in S1, then S2 will be equal to S1 and S3 will be the empty string.

For example,

```
{String.token "a:b:c" &: S1 S2}
```

binds `S1` to `"a"` and `S2` to `"b:c"`.

tokens

```
{String.tokens +S X ?Ss}
```

splits the string S into substrings Ss delimited by occurrences of X in S. Note that the final empty string will be omitted if the last element of S is an X.

For example,

```
{String.tokens "a:bb:cc:d:" &:}
```

returns `["a" "bb" "cc" "d"]`.

7.3 Byte Strings

Module `ByteString` provides an interface to a more economical representation for textual data: a simple array of bytes. In terms of memory, the economy is at least a factor of 8, which may improve local processing, IO, pickle sizes, and remote communications (distributed processing). However strings, i.e., lists, are often more convenient for all forms of recursive processing (e.g., `Map` or `Filter`). Typically, you will be processing textual data in list form, but saving or communicating it in byte string form.

IsByteString

```
{ByteString.is +X ?B}
```

tests whether X is a byte string.

make

```
{ByteString.make +V ?ByteString}
```

returns a new byte string created from the virtual string V.

get

```
{ByteString.get +ByteString +I ?C}
```

retrieves the Ith character of byte string ByteString. The first index is `0`.

append

```
{ByteString.append +ByteString1 +ByteString2 ?ByteString3}
```

returns the new byte string ByteString3 which is the concatenation of ByteString1 and ByteString2.

slice

```
{ByteString.slice +ByteString1 +FromI +ToI ?ByteString2}
```

returns a new byte string for the bytes in ByteString1 starting at index FromI and extending up to, but not including, index ToI.

width
length

```
{ByteString.width +ByteString ?I}
{ByteString.length +ByteString ?I}
```

returns the width `I` of `ByteString`.

toString

```
{ByteString.toString +ByteString ?S}
```

converts `ByteString` to a string `S`.

toStringWithTail

```
{ByteString.toStringWithTail +ByteString X ?S}
```

converts `ByteString` to a string `S` ending with `X`. This is useful for subsequently instantiating `X`, e.g., with another call to `ByteString.toStringWithTail`.

strchr

```
{ByteString.strchr +ByteString +OffsetI +Char ?PosBI}
```

returns the position `PosI` of the first occurrence of `Char` in `ByteString` starting at offset `OffsetI`. If none is found **`false`** is returned instead.

7.4 Virtual Strings

The module `VirtualString` contains procedures operating on virtual strings. Virtual strings are designed as a convenient way to combine strings, byte strings, atoms, integers and floats to compound strings without explicit concatenation and conversion.

IsVirtualString

```
{VirtualString.is +X ?B}
```

tests whether `X` is a virtual string. Virtual strings are defined recursively as the set of all integers, floats, atoms, strings, byte strings, and tuples with label `'#'` whose subtrees are virtual strings.

toString

```
{VirtualString.toString +V ?S}
```

converts a virtual string `V` to a string `S`.

The transformation is straightforward: Atoms (except `nil` and `'#'`), integers, floats and byte strings are transformed into strings using `Atom.toString`, `Int.toString`, `Float.toString`, and `ByteString.toString` respectively, where in numbers `-` is used instead of `~`. A tuple with label `'#'` is transformed by concatenation of the transformed subtrees. Note that both `nil` and `'#'` denote the empty string.

The following relation holds for all virtual strings `V1` and `V2`:

```
{VirtualString.toString V1#V2}
= {Append
   {VirtualString.toString V1}
   {VirtualString.toString V2}}
```

Thus, `VirtualString.toString` maps `#` homomorphically to `Append`.

toAtom

```
{VirtualString.toAtom +V ?A}
```

converts a virtual string V to an atom A.

This procedure can be defined as:

```
fun {VirtualString.toAtom V}
   {String.toAtom {VirtualString.toString V}}
end
```

toByteString

```
{VirtualString.toByteString +V ?ByteString}
```

converts a virtual string V to a byte string `ByteString`.

This procedure is a synonym of `ByteString.make` (which see).

length

```
{VirtualString.length +V ?I}
```

returns the length of a virtual string in characters. Can be defined as:

```
{Length {VirtualString.toString V} I}
```

changeSign

```
{VirtualString.changeSign +V1 X ?V2}
```

returns a virtual string derived from V1 where all occurrences of the unary minus sign for integers and floats are replaced by X.

8

Procedures and Cells

8.1 Procedures

The module `Procedure` specifies operations on procedures.

`IsProcedure`

```
{Procedure.is +X ?B}
```

tests whether `X` is a procedure.

`ProcedureArity`

```
{Procedure.arity +P ?I}
```

returns the procedure arity of `P`, i.e., the number of arguments which `P` takes.

`apply`

```
{Procedure.apply +P +Xs}
```

applies the procedure `P` to the arguments given by the elements of the list `Xs`, provided that

```
{Procedure.arity P} == {Length Xs}
```

8.2 Cells

The module `Cell` contains procedures operating on cells.

`IsCell`

```
{Cell.is +X ?B}
```

tests whether `X` is a cell.

`NewCell`

```
{Cell.new X ?Cell}
```

returns a new cell with initial content `X`.

`Exchange`

```
{Cell.exchange +Cell X Y}
```

returns the current content of `Cell` in `X`, and sets the content of `Cell` to `Y`.

Access

```
{Cell.access +Cell X}
```

returns the current content of `Cell` in `X`.

Assign

```
{Cell.assign +Cell X}
```

sets the content of `Cell` to `X`.

9

Chunks

9.1 Chunks in General

The module `Chunk` contains procedures operating on chunks.

`.`

`{Value.'.' +RC +LI X}`

returns the field `X` of `RC` at feature `LI`.

`HasFeature`

`{Value.hasFeature +RC +LI ?B}`

tests whether `RC` has feature `LI`.

`CondSelect`

`{Value.condSelect +RC +LI X Y}`

returns the field `Y` of `RC` at `LI`, if `RC` has feature `LI`. Otherwise, returns `X`.

`IsChunk`

`{Chunk.is +X ?B}`

tests whether `X` is a chunk.

`NewChunk`

`{Chunk.new +R ?C}`

returns a new chunk with the same features and fields as `R`.

9.2 Arrays

The module `Array` contains procedures operating on arrays. Whenever an array access is indexed with an illegal key, an error exception is raised.

`IsArray`

`{Array.is +X ?B}`

tests whether `X` is an array.

`NewArray`

`{Array.new +LowI +HighI InitX ?Array}`

returns a new array with key range from `LowI` to `HighI` including both. All items are initialized to `InitX`.

`Put`

```
{Array.put +Array +I X}
```

sets the item of `Array` under key `I` to `X`.

`Get`

```
{Array.get +Array +I X}
```

returns the item of `Array` under key `I`.

`exchange`

```
{Array.exchange +Array +I OldVal NewVal}
```

returns the current value of `Array` under key `I` as item `OldVal` and updates the value of `Array` under key `I` to be `NewVal`.

`low`

```
{Array.low +Array ?LowI}
```

returns the lower bound of the key range of `Array`.

`high`

```
{Array.high +Array ?HighI}
```

returns the upper bound of the key range of `Array`.

`clone`

```
{Array.clone +A1 ?A2}
```

returns a new array with the same bounds and contents as `A1`.

`toRecord`

```
{Array.toRecord +L +A ?R}
```

returns a record with label L that contains as features the integers between `{Array.low A}` and `{Array.high A}` and with the corresponding fields.

9.3 Dictionaries

The module `Dictionary` contains procedures operating on dictionaries. If a dictionary contains an item under some key `LI`, we say `LI` is a valid key. Whenever a dictionary access is indexed with an ill-typed key, a type error is raised. For a missing but well-typed key, a system exception is raised.

`IsDictionary`

```
{Dictionary.is +X ?B}
```

tests whether `X` is a dictionary.

`NewDictionary`

```
{Dictionary.new ?Dictionary}
```

returns a new empty dictionary.

`put`

`{Dictionary.put +Dictionary +LI X}`

sets the item in `Dictionary` under key `LI` to `X`.

`get`

`{Dictionary.get +Dictionary +LI X}`

returns the item `X` of `Dictionary` under key `LI`.

`condGet`

`{Dictionary.condGet +Dictionary +LI DefVal X}`

returns the item `X` of `Dictionary` under key `LI`, if `LI` is a valid key of `Dictionary`. Otherwise, returns `DefVal`.

`exchange`

`{Dictionary.exchange +Dictionary +LI OldVal NewVal}`

returns the current value of `Dictionary` under key `LI` as item `OldVal` and updates the value of `Dictionary` under key `LI` to be `NewVal`.

`condExchange`

`{Dictionary.condExchange +Dictionary +LI DefVal OldVal NewV`

If `LI` is a valid key of `Dictionary` then returns the current value of `Dictionary` under key `LI` as item `OldVal` otherwise, returns `DefVal` as item `OldVal`. Sets the value of `Dictionary` under key `LI` to be `NewVal`.

`keys`

`{Dictionary.keys +Dictionary ?LIs}`

returns a list of all currently valid keys of `Dictionary`.

`entries`

`{Dictionary.entries +Dictionary ?Ts}`

returns the list of current entries of `Dictionary`. An entry is a pair `LI#X`, where `LI` is a valid key of `Dictionary` and `X` the corresponding item.

`items`

`{Dictionary.items +Dictionary ?Xs}`

returns the list of all items currently in `Dictionary`.

`isEmpty`

`{Dictionary.isEmpty +Dictionary ?B}`

tests whether `Dictionary` currently contains an entry.

`remove`

`{Dictionary.remove +Dictionary +LI}`

removes the item under key `LI` from `Dictionary` if `LI` is a valid key. Otherwise, does nothing.

`removeAll`

`{Dictionary.removeAll +Dictionary}`

removes all entries currently in `Dictionary`.

`member`

`{Dictionary.member +Dictionary +LI ?B}`

tests whether `LI` is a valid key of `Dictionary`.

`clone`

`{Dictionary.clone +Dictionary1 ?Dictionary2}`

returns a new dictionary `Dictionary2` containing the currently valid keys and corresponding items of `Dictionary1`.

`toRecord`

`{Dictionary.toRecord +L +Dictionary ?R}`

returns a record `R` with label `L` whose features and their fields correspond to the keys and their entries of `Dictionary`.

`weak`

another way to access module `WeakDictionary` (see Section 9.4).

9.4 Weak Dictionaries

The module `WeakDictionary` contains procedures operating on weak dictionaries. A weak dictionary is much like an ordinary dictionary and supports the same API. The main difference is that an entry is kept only as long as its item (i.e. the value recorded under the key) has not become garbage. If the item is only reachable through one or more weak dictionaries, the corresponding entries will automatically be dropped from all weak dictionaries at the next garbage collection.

Finalization Stream Each weak dictionary is associated with a *finalization stream*. When an item `X` (indexed under `Key`) becomes garbage, the entry is automatically removed from the weak dictionary at the next garbage collection and the pair `Key#X` is sent on to the finalization stream (as if the weak dictionary were associated with a port and the pair was sent to it using `Port.send`). This means that the item, which was garbage, becomes again non-garbage when it is sent to the finalization stream. If subsequently, this last remaining reference disappears, then the item really becomes garbage since it won't be referenced even through a weak dictionary.

The finalization stream is created at the same time as the weak dictionary; both are output arguments of `NewWeakDictionary`. If you are not interested in the finalization stream, you can explicitly close it using `WeakDictionary.close`.

Module `WeakDictionary` can also be accessed as `Dictionary.weak`.

`IsWeakDictionary`

`{WeakDictionary.is +X ?B}`

tests whether `X` is a weak dictionary.

`NewWeakDictionary`

`{WeakDictionary.new ?L ?Weak}`

has the same arguments as `Port.new` (page 49). Returns a new empty weak dictionary associated with a new finalization stream `L`.

close

`{WeakDictionary.close +Weak}`

drops the finalization stream (if any). After this, any entry that becomes garbage is simply dropped instead of being sent to the finalization stream. Note that the `close`, `put`, `remove`, and `removeAll` operations cannot be used in a (subordinated) space other than where the weak dictionary has been constructed.

put

`{WeakDictionary.put +Weak +LI X}`

sets the item in `Weak` under key `LI` to `X`.

get

`{WeakDictionary.get +Weak +LI X}`

returns the item `X` of `Weak` under key `LI`.

condGet

`{WeakDictionary.condGet +Weak +LI X Y}`

returns the item `Y` of `Weak` under key `LI`, if `LI` is a valid key of `Weak`. Otherwise, returns `X`.

exchange

`{WeakDictionary.exchange +Weak +LI OldVal NewVal}`

returns the current value of `Weak` under key `LI` as item `OldVal` and updates the value of `Weak` under key `LI` to be `NewVal`.

condExchange

`{WeakDictionary.condExchange +Weak +LI DefVal          OldVal NewVal}`

If `LI` is a valid key of `Weak` then returns the current value of `Weak` under key `LI` as item `OldVal` otherwise, returns `DefVal` as item `OldVal`. Sets the value of `Weak` under key `LI` to be `NewVal`.

keys

`{WeakDictionary.keys +Weak ?LIs}`

returns a list of all currently valid keys of `Weak`.

entries

`{WeakDictionary.entries +Weak ?Ts}`

returns the list of current entries of `Weak`. An entry is a pair `LI#X`, where `LI` is a valid key of `Weak` and `X` the corresponding item.

items

`{WeakDictionary.items +Weak ?Xs}`

returns the list of all items currently in `Weak`.

isEmpty

`{WeakDictionary.isEmpty +Weak ?B}`

tests whether `Weak` currently contains an entry.

remove

`{WeakDictionary.remove +Weak +LI}`

removes the item under key `LI` from `Weak` if `LI` is a valid key. Otherwise, does nothing.

`removeAll`

`{WeakDictionary.removeAll +Weak}`

removes all entries currently in `Weak`.

`member`

`{WeakDictionary.member +Weak +LI ?B}`

tests whether `LI` is a valid key of `Weak`.

`toRecord`

`{WeakDictionary.toRecord +L +Weak ?R}`

returns a record `R` with label `L` whose features and their fields correspond to the keys and their entries of `Weak`.

9.5 Bit Arrays

The module `BitArray` contains procedures operating on arrays of bits (i.e., units of information each being either set or reset).

`IsBitArray`

`{BitArray.is +X ?B}`

tests whether `X` is a bit array.

`new`

`{BitArray.new +LowI +HighI ?BitArray}`

creates an new `BitArray` with lower bound `LowI` and upper bound `HighI`, and all bits initially cleared. This interface is identical to that of general Oz arrays.

`set`

`{BitArray.set +BitArray +I}`

sets bit `I` of `BitArray`.

`clear`

`{BitArray.clear +BitArray +I}`

clears bit `I` of `BitArray`.

`test`

`{BitArray.test +BitArray +I ?B}`

tests whether bit `I` of `BitArray` is set.

`low`

`{BitArray.low +BitArray ?LowI}`

returns the lower bound `LowI` of `BitArray`.

`high`

`{BitArray.high +BitArray ?HighI}`

returns the upper bound `HighI` of `BitArray`.

`clone`

`{BitArray.clone +BitArray1 ?BitArray2}`

returns a new bit array that is a copy of its first argument.

`disj`

`{BitArray.disj +BitArray1 +BitArray2}`

side-effects its first argument with the bitwise ‘or’ of the two arguments.

`conj`

`{BitArray.conj +BitArray1 +BitArray2}`

side-effects its first argument with the bitwise ‘and’ of the two arguments.

`nimpl`

`{BitArray.nimpl +BitArray1 +BitArray2}`

side-effects its first argument with the bitwise ‘and’ of the the first argument and the negation of the second argument (i.e., negated implication).

`disjoint`

`{BitArray.disjoint +BitArray1 +BitArray2 ?B}`

tests whether the bit arrays have no set bits in common.

`card`

`{BitArray.card +BitArray ?I}`

returns the number of set bits.

`toList`

`{BitArray.toList +BitArray ?L}`

returns the list of indices for all set bits in `BitArray`.

`complementToList`

`{BitArray.complementToList +BitArray ?L}`

returns the list of indices for all cleared bits in `BitArray`.

9.6 Ports

The module `Port` contains procedures operating on ports.

`IsPort`

`{Port.is +X ?B}`

tests whether `X` is a port.

`NewPort`

`{Port.new ?Xs ?Port}`

returns a new port, together with its associated stream `Xs`.

Send

```
{Port.send +Port X}
```

sends `X` to the port `Port`: The stream pointed to by `Port` is unified with `X|_` (in a newly created thread), and the pointer advances to the stream's new tail.

SendRecv

```
{Port.sendRecv +Port X Y}
```

sends the pair `X#Y` to the port `Port`: The stream pointed to by `Port` is unified with `X#Y|_` (in a newly created thread), and the pointer advances to the stream's new tail.

The argument `X` is commonly used as message to be sent, while `Y` serves as reply to that message.

9.7 Locks

The module `Lock` contains procedures for locks.

IsLock

```
{Lock.is +X ?B}
```

tests whether `X` is a lock.

NewLock

```
{Lock.new ?LockC}
```

creates and returns a new lock.

9.8 Classes

The module `Class` contains procedures operating on classes.

IsClass

```
{Class.is +X ?B}
```

tests whether `X` is a class.

new

```
{Class.new +ParentKs +AttrR +FeatR +PropAs ?K}
```

creates a new class by inheriting from `ParentKs` with new attributes `AttrR` and new features `FeatR`. The fields with integer features in `AttrR` define the free attributes. The fields with literal features define attributes with initial values, where the feature is the attribute name and the field its initial value. The semantics for `FeatR` is accordingly. The properties of the class to be created are defined by `PropAs` (a list of atoms, valid elements are `sited`, `final`, and `locking`).

For example, the statement

```
C={Class.new [D E] a(a:1 b) f(f:2 g) [final]}
```

is equivalent to

```
class C from D E
   prop final
   attr a:1 b
   feat f:2 g
end
```

getAttr

```
{Class.getAttr +K +LI ?X}
```

Returns the initival value X for attribute LI as defined by the class K.

For example, the statement

```
{Class.getAttr class attr a:4 end a}
```

returns 4.

9.9 Objects

The module Object contains procedures operating on objects.

The system procedures that define the behaviour of Oz objects and classes are also given in this section.

IsObject

```
{Object.is +X ?B}
```

tests whether X is an object.

New

```
{Object.new +K +InitMessageR ?O}
```

Creates a new object from class K with initial message InitMessageR.

The BaseObject Class

The class BaseObject defines the following method.

noop

```
noop()
```

does nothing. It is defined as `meth noop() skip end`.

9.10 Functors

The module Functor contains procedures operating on functors.

is

```
{Functor.is +X ?B}
```

tests whether X is a functor.

new

```
{Functor.new ⟨import spec⟩ ⟨export spec⟩ +P ?Functor}
```

returns a new functor with imports as described by the ⟨import spec⟩, exports as described by the ⟨export spec⟩, and body as performed by P.

The ⟨import spec⟩ is a record mapping the name of each imported module to a record giving information about it:

```
⟨import spec⟩   ::=   'import'(⟨module name⟩: ⟨import info⟩ ...
                               ⟨module name⟩: ⟨import info⟩)

⟨module name⟩   ::=   ⟨atom⟩
```

The optional `'from'` field gives the value of this import's **`at`** clause, if given:

```
⟨import info⟩   ::=   info(type: ⟨type⟩ ['from': ⟨atom⟩])
```

The `type` field is the expected type of the module. This can be any of the atoms returned by `Value.type`, plus some more implementation-specific ones, or a record with label `record`:

```
⟨type⟩   ::=   int | atom | ...                                     % see above
           |   record(⟨feature⟩: ⟨type⟩ ... ⟨feature⟩: ⟨type⟩)
           |   nil                                                  % no information kno
```

The ⟨export spec⟩ is a record mapping each feature of the module resulting from applications of this functor to the type of the corresponding value:

```
⟨export spec⟩   ::=   'export'(⟨feature⟩: ⟨type⟩ ... ⟨feature⟩: ⟨type⟩)
```

The body is a binary procedure `{P ⟨import⟩ ⟨export⟩}` where:

```
⟨import⟩   ::=   'IMPORT'(⟨module name⟩: ⟨value⟩ ... ⟨module name⟩: ⟨value⟩)

⟨export⟩   ::=   ⟨value⟩
```

10

Control

This chapter contains control procedures which allow to block, suspend or terminate threads, and provide functionality dealing with loops, real-time programming and threads.

10.1 General

The module `Value` contains the following general control procedures.

`Wait`

```
{Value.wait +X}
```

blocks until X is determined. This statement makes X needed, causing all by-need computations on X to be triggered. If X is or becomes bound to a failed value, then its encapsulated exception is raised.

`WaitOr`

```
{Value.waitOr X Y}
```

blocks until at least one of X or Y is determined.

`waitQuiet`

```
{Value.waitQuiet +X}
```

blocks until X is determined or failed. Contrary to `Wait`, `Value.waitQuiet` does not make X needed. Also, if X is or becomes bound to a failed value, no exception is raised.

`!!`

```
{Value.'!!' X Y}
```

returns a future Y for X, i.e., a read-only placeholder for X. If Y becomes needed, X is made needed too.

`WaitNeeded`

```
{Value.waitNeeded X}
```

blocks until X is needed. This operation is the by-need synchronization.

`IsNeeded`

```
{Value.isNeeded X ?B}
```

tests whether X is needed.

makeNeeded

```
{Value.makeNeeded X}
```

makes X needed. This statement is useful for triggering by-need computations on X with having to suspend on X.

ByNeed

```
{Value.byNeed P X}
```

concurrently evaluates {P X} as soon as X becomes needed. It can be defined as follows:

```
proc {ByNeed P X}
   thread {WaitNeeded X} {P X} end
end
```

ByNeedFuture

```
{Value.byNeedFuture P X}
```

creates a by-need computation that evaluates the expression {P}, and returns a future X of its result. If the call to P raises an exception E, then X is bound to a failed value (see below) that encapsulates E. It can be defined as follows:

```
fun {ByNeedFuture P}
   !!{ByNeed fun {$}
                try {P} catch E then {Value.failed E} end
             end}
end
```

failed

```
{Value.failed E X}
```

creates a failed value X encapsulating exception E. Whenever a statement needs X, in particular, whenever a thread synchronizes on X, exception term E is raised. This is convenient in concurrent designs: if a concurrent generator encounters a problem while computing a value, it may catch the corresponding exception, package it as a failed value and return the latter instead. Thus each consumer will be able to synchronously handle the exception when it attempts to use the 'failed' value. For example, the module manager returns failed futures for modules that cannot be found or linked.

The failed value X can only unify to itself. An attempt to unify X to any other value results in exception E to be raised. Unifying two distinct failed values causes one of them to raise its exception, the choice of which being nondeterministic.

10.2 Loops

The module Loop contains procedures that represent recursive versions of common iteration schemes with integers. However, for most common iteration patterns, the **for** loop offers a nicer alternative (see *"Loop Support"*).

For

```
{Loop.'for' +I1 +I2 +I3 +P}
```

applies the unary procedure P to integers from I1 to I2 proceeding in steps of size I3. For example,

```
{For 1 11 3 Browse}
```

displays the numbers 1, 4, 7, and 10 in the browser window, whereas

```
{For 11 1 ~3 Browse}
```

displays the numbers 11, 8, 5, and 2.

ForThread

```
{Loop.forThread +I1 +I2 +I3 +P X ?Y}
```

applies the ternary procedure P to integers from I1 to I2 proceeding in steps of size I3 while threading an additional accumulator argument through the iteration. The procedure P takes the accumulator argument (initially set to X) and the loop index and returns an updated accumulator.

For example,

```
{ForThread 1 5 1 fun {$ Is I} I*I|Is end nil}
```

yields the list [25 16 9 4 1] as output, whereas

```
{ForThread 5 1 ~1 fun {$ Is I} I*I|Is end nil}
```

yields [1 4 9 16 25] as output.

Note that ForThread is similar to FoldL (see Section 6.3).

multiFor

```
{Loop.multiFor +Xs +P}
```

generalizes For (see above) to the case of multiple nested loops.

Xs is a list containing tuples of the form I1#I2#I3 specifying a loop by its start value I1, upper limit I2 and step size I3.

For example,

```
{Loop.multiFor [1#5#1 10#20#2] Browse}
```

displays the lists [1 10], [1 12], ..., [5 20] in the browser.

multiForThread

```
{Loop.multiForThread +Xs +P X ?Y}
```

generalizes ForThread (see above) to the case of multiple nested loops.

Xs is a list containing tuples of the form I1#I2#I3 specifying a loop by its start value I1, upper limit I2 and step size I3.

For example,

```
{Loop.multiForThread [1#2#1 5#4#~1]
 fun {$ Is [I J]}
    I#J|Is
 end nil}
```

yields the list [2#4 2#5 1#4 1#5] as output.

10.3 Time

The module `Time` contains procedures for real-time applications.

`Alarm`

```
{Time.alarm +I ?U}
```

returns **`unit`** after `I` milliseconds. This is done asynchronously in that it is evaluated on its own thread.

`Delay`

```
{Time.delay +I}
```

reduces to **`skip`** after `I` milliseconds. Whenever `I =< 0`, `{Delay I}` reduces immediately.

`time`

```
{Time.time ?T}
```

binds `T` to the number of seconds elapsed since January, 1st of the current year.

The Repeater Class

`repeat`

```
Time.repeat
```

is a class which allows to

- repeat an action infinitely often, or a fixed number of times and perform some final action thereafter,
- with a fixed delay between iterations (or, alternatively, a delay specified by a unary procedure),
- stop and resume the iteration.

There are default values for any of the iteration parameters. These are set on creation of an object inheriting from `Time.repeat` and can be changed by inheritance. The functionality is controlled by the following methods.

`setRepAll`

```
setRepAll(action:   +ActionPR  <= dummyRep
          final:    +FinalPR   <= finalRep
          delay:    +DelayI    <= 1000
          delayFun: +DelayFunP <= fun {$} 1000 end
          number:   +NumI      <= ~1)
```

initializes the loop with the action `ActionPR` to iterate (default: message `dummyRep`), the action `FinalPR` to finalize a finite iteration (default: message `finalRep`), the delay `DelayI` between iterations (default: one second), the function `DelayFunP` yielding the delay between iterations (default: constant `1000`), and the maximal number `NumI` of iterations (default: infinitely many).

The methods `dummyRep` and `finalRep` do nothing. Only one of the `delay` and `delayFun` parameters can be given. The default actions `ActionPR` and `FinalPR` can be changed by inheritance.

The loop is started on the calling thread.

For example, try the following:

```
local
   O = {New Time.repeat
          setRepAll(action: proc {$} {OS.system 'fortune' _} end
                    number: 10)}
in
   {O go()}
end
```

getRep

```
getRep(action:   ?ActionPR  <= _
       final:    ?FinalPR   <= _
       delay:    ?DelayI    <= _
       delayFun: ?DelayFunP <= _
       number:   ?LimitI    <= _
       actual:   ?NumI      <= _)
```

returns the current loop parameters: `LimitI` returns the current limit of the iteration, and `NumI` the number of iterations left to be done. If the delay was specified via `DelayFunP` (which need not be constant), then `DelayI` returns the *last* delay used. If `DelayI` is requested before the start of the iteration, then `~1` is returned. The other values correspond to the fields of the method `setRepAll`.

For example try:

```
local
   class Counter from Time.repeat
      attr a: 0
      meth inc()      a := @a + 1 end
      meth get(?A)    A = @a      end
      meth finalRep() a := 0      end
   end
   C = {New Counter setRepAll(action: inc number: 1000)}
in
   thread {C go()} end
   {C getRep(final: {Browse}
             action: {Browse}
             actual: {Browse})}
   {C get({Browse})}
end
```

This will show the atoms `'finalRep'` and `'inc'` in the Browser, as well as a number between `1` and `1000`. After termination of the loop, the value of `@a` will be reset to `0`.

setRepDelay

```
setRepDelay(+DelayI <= 1000}
```

setRepNum

```
setRepNum(+NumI <= ~1)}
```

setRepAction

```
setRepAction(+ActionPR <= dummyRep)
```

setRepFinal

```
setRepFinal(+FinalPR <= finalRep)}
```

setRepDelayFun

```
setRepDelayFun(+DelayFunP <= fun {$} 1000 end)
```

allow to set the numeric parameters of the iteration.

`DelayI` and `NumI` must be integers. The iteration limit `NumI` is stored and subsequent loop instances (triggered by `go`) also obey it, unless the limit is reset to `~1`.

`ActionPR` and `FinalPR` may be nullary procedures or records. If they are procedures they are called as is. If they are records, they are interpreted as messages to be sent to `self`.

`DelayFunP` must be a unary procedure which returns an integer value on application.

go

```
go()
```

starts the loop if it is not currently running.

stop

```
stop()
```

halts the loop and resets the iteration index. The loop may be restarted with `go`.

10.4 Exceptions

The module `Exception` provides procedures to construct exceptions and raise them.

Special Exceptions Any value may be raised as exception, although commonly only records are used. Some of these serve special purposes: error exceptions are records with label `error`. These are raised when a programming error occurs; it is not recommended to catch these. System exceptions are records with label `system`. These are raised when an unforeseeable runtime condition occurs; a file operations library might raise system exceptions when a file cannot be opened. It is recommended to always handle such exceptions. Failure exceptions are records with label `failure`; these are raised when a tell operation fails.

Dispatch Fields Both error and system exceptions have a dispatch field. This is the subtree at feature `1` of the exception record. This is usually a record further describing the exact condition that occurred.

Debug Information If an exception is a record and has a feature `debug` with value `unit`, then the implementation may replace (depending on the value of the property `ozconf.errorDebug`) the corresponding subtree by implementation-dependent debugging information. This is to be printed out in the case of uncaught exceptions.

All procedures in the Base Environment only ever raise special-purpose exceptions as described above.

error

```
{Exception.error X ?Y}
```

returns an error exception record with dispatch field `X`.

system

```
{Exception.system X ?Y}
```

returns a system exception record with dispatch field X.

failure

```
{Exception.failure X ?Y}
```

returns a failure exception. The value X may give a hint on why failure occurred; implementations may store this inside the constructed exception's `debug` field.

Raise

```
{Exception.'raise' X}
```

raises X as an exception.

raiseError

```
{Exception.raiseError X}
```

wraps X into an error exception and raises this. This procedure can be defined as follows, except that it always add debug informations:

```
proc {Exception.raiseError X}
   {Exception.'raise' {AdjoinAt {Exception.error X} debug unit}}
end
```

10.5 Threads

The module `Thread` provides operations on first class threads.

Threads may be in one of three states, namely runnable, blocked, or terminated. Orthogonally, a thread may be suspended.

Runnable and non-suspended threads are scheduled according to their priorities, which may be `low`, `medium`, or `high`. The default priority is `medium`. The priority of a thread may influence its time share for execution, where threads with `medium` priority obtain at least as long a time share as threads with `low` priority and at most as long as threads with `high` priority. Implementations may also choose not to schedule a thread at all if a thread with higher priority is runnable.

A newly created thread inherits its priority from its parent if the latter has either `medium` or `low` priority. Otherwise, the new thread gets default (i.e., `medium`) priority.

IsThread

```
{Thread.is +X ?B}
```

test whether X is a thread.

this

```
{Thread.this ?Thread}
```

returns the current thread.

state

```
{Thread.state +Thread ?A}
```

returns one of the atoms `runnable`, `blocked`, `terminated` according to the current state of `Thread`.

`resume`

```
{Thread.resume +Thread}
```

resumes `Thread`. Resumption undoes suspension.

`suspend`

```
{Thread.suspend +Thread}
```

suspends `Thread` such that it cannot be further reduced.

`isSuspended`

```
{Thread.isSuspended +Thread ?B}
```

tests whether `Thread` is currently suspended.

`injectException`

```
{Thread.injectException +Thread +X}
```

raises `X` as exception on `Thread`. If `Thread` is terminated, an error exception is raised in the current thread.

`terminate`

```
{Thread.terminate +Thread}
```

raises an exception `kernel(terminate ...)` on `Thread`.

`getPriority`

```
{Thread.getPriority +Thread ?A}
```

returns one of them atoms `low`, `medium`, or `high` according to the current priority of `Thread`.

`setPriority`

```
{Thread.setPriority +Thread +A}
```

sets priority of thread `Thread` to the priority described by atom `A`. `A` must be one of `low`, `medium`, or `high`.

`getThisPriority`

```
{Thread.getThisPriority ?A}
```

returns one of them atoms `low`, `medium`, or `high` according to the priority of the current thread.

`setThisPriority`

```
{Thread.setThisPriority +A}
```

sets priority of the current thread to the priority described by atom `A`. `A` must be one of `low`, `medium`, or `high`.

`preempt`

```
{Thread.preempt +Thread}
```

preempts the current thread, i.e., immediately schedules another runnable thread (if there is one). `Thread` stays runnable.

11 Infix Notations

Oz supports infix and prefix notation for very common procedures (see Section *Operator Associativity and Precedence, (The Oz Notation)*).

In the following table, we give the prefix and infix notations and the corresponding expansions. The operators are grouped together according to their precedence. Members of the same group have the same precedence, groups further up have lower precedence than groups further down. ‘Having higher precedence’ means ‘binding tighter’; e.g., the term `X.Y + Z` is equal to `(X.Y) + Z`. Ambiguities within each group are resolved by the associativity given before each group (e.g., `X - Y + Z` is equivalent to `(X - Y) + Z`).

Infix	Normal
right-associative	
`X = Y`	`{Value.'=' X Y}`
right-associative	
`+X := Y`	Cell, Attribute, Dictionary or Array element assignment
`Z = +X := Y`	Cell, Attribute, Dictionary or Array element exchange
`+C.+LI := X`	Dictionary or Array element assignment
`Y = +C.+LI := X`	Dictionary or Array element exchange
`+LI <- X`	`{Object.'<-' self LI X} %% Object.'<-' is internal`
`Y = +LI <- X`	`{Object.exchange self LI X Y} %% Object.exchange is int`
non-associative	
`?B = X == Y`	`{Value.'==' X Y B}`
`?B = X \= Y`	`{Value.'\\=' X Y B}`
`?B = +AFI1 < +AFI2`	`{Value.'<' AFI1 AFI2 B}`
`?B = +AFI1 =< +AFI2`	`{Value.'=<' AFI1 AFI2 B}`
`?B = +AFI1 > +AFI2`	`{Value.'>' AFI1 AFI2 B}`
`?B = +AFI1 >= +AFI2`	`{Value.'>=' AFI1 AFI2 B}`
left-associative	
`?FI3 = +FI1 + +FI2`	`{Number.'+' FI1 FI2 FI3}`
`?FI3 = +FI1 - +FI2`	`{Number.'-' FI1 FI2 FI3}`
left-associative	
`?FI3 = +FI1 * +FI2`	`{Number.'*' FI1 FI2 FI3}`
`?F3 = +F1 / +F2`	`{Float.'/' F1 F2 F3}`
`?I3 = +I1 div +I2`	`{Int.'div' I1 I2 I3}`
`?I3 = +I1 mod +I2`	`{Int.'mod' I1 I2 I3}`
right-associative	
`+K, +R`	`{Object.',' K R} %% Object.',' is internal`
prefix	
`?FI1 = ~+FI2`	`{Number.'~' FI2 FI1}`
left-associative	
`X = +Y.+LI`	get content of Record, Dictionary, or Array element
prefix	
`X = @+Y`	get content of Cell, Attribute, Dictionary or Array element
`X = !!Y`	`{Value.'!!' X Y}`

The expansion of the state-manipulation operators (`.`, `:=`, and `@`) depends on the type of the expressions involved. The expansions are simplified, suitable error messages are returned if the type of the expressions are not valid.

`E1.E2`

`E1.E2` expands to

```
if {Record.is E1} then {Value.'.' E1 E2} else @(E1#E2) end
```

returns the content of a record, dictionary, or array element.

`E1.E2 := E3`

`E1.E2 := E3` expands to

```
@(E1#E2) := E3
```

`'. :='` is a *ternary* operator for updating dictionary and array elements.

E1 := E2

E1 := E2 expands to

```
case E1
of (D#K) andthen {Dictionary.is D} then {Dictionary.put D K E2}
[] (A#I) then {Array.put A I E2}
elseif {IsCell E1} then
   {Cell.assign E1 E2}
else
   {Object.assign self E1 E2}   %% E1 <- E2 (Object.assign is internal
end
```

':=' updates dictionaries, arrays, cells, and attributes. Note, `Object.assign` is a dummy routine and not actually visible to the library user.

@E

@E expands to

```
case E
of (D#K) andthen {Dictionary.is D} then {Dictionary.get D K}
[] (A#I) then {Array.get A I}
elseif {IsCell E} then
    {Cell.access E}
else
    {Object.access self E}   %% @E (Object.access is internal)
end
```

'@' returns the current value stored in dictionaries, arrays, cells, and attributes. Note, `Object.access` is a dummy routine and not actually visible to the library user.

X = E1 := E2

X = E1 := E2 expands to

```
'atomic'
   X = @E1
   E1 := E2
'end'
```

In an expression context ':=' performs an atomic exchange with the current value stored in the dictionary, array, cell, or attribute. Note 'atomic' ... 'end' is pseudo code to indicate that the exchange is an atomic action.

12

Miscellaneous

12.1 Bit Strings

Module `BitString` provides an interface to a fast and economical representation for immutable bit sequences.

`IsBitString`

```
{BitString.is +X ?B}
```

tests whether `X` is a bit string.

`make`

```
{BitString.make +I +L ?BitString}
```

creates a bit string of width `I` with precisely those bits set that are specified in `L`, a list of indices.

`conj`

```
{BitString.conj +BitString1 +BitString2 ?BitString3}
```

returns the bitwise 'and' of its first and second arguments, which must be of indentical widths.

`disj`

```
{BitString.disj +BitString1 +BitString2 ?BitString3}
```

returns the bitwise 'or' of its first and second arguments, which must be of indentical widths.

`nega`

```
{BitString.nega +BitString1 ?BitString2}
```

returns the bitwise negation of its first argument.

`get`

```
{BitString.get +BitString +I ?B}
```

tests whether bit `I` of `BitString` is set.

`put`

```
{BitString.put +BitString1 +I +B ?BitString2}
```

returns a new bit string which is identical to `BitString1` except that bit `I` is set iff `B` is **`true`**.

width

`{BitString.width +BitString ?I}`

returns the width of the `BitString`.

toList

`{BitString.toList +BitString ?L}`

returns the list of indices of all set bits in `BitString`.

12.2 Foreign Pointers

Module `ForeignPointer` provides an interface to encapsulated *raw* pointers to foreign data. This is useful for implementors of foreign libraries: any C pointer can be encapsulated as a ForeignPointer and passed around as an Oz value. However, you should consider subclassing the `Oz_Extension` class instead, to encapsulate your C++ data structures into new Oz (abstract) datatypes.

IsForeignPointer

`{ForeignPointer.is +X ?B}`

tests whether X is a foreign pointer

toInt

`{ForeignPointer.toInt +X ?I}`

converts a foreign pointer to an integer. Two foreign pointers convert to the same integer iff they point to the same location.

Bibliography

[1] Information processing – 8-bit single-byte coded graphic character sets – part 1: Latin, alphabet no. 1. Technical Report ISO 8859-1:1987, Technical committee: JTC 1/SC 2, International Organization for Standardization, 1987.

[2] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall International, second edition, 1988.

Index