The Standard ML of New Jersey Library Reference Manual (Version 0.1)

AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ, 07974

February 1, 1993

Contents

1 Introduction

This document describes the first release of the Standard ML of New Jersey (SML/NJ) Library. The Library interface is presented as a collection of UNIX style manual pages.

1.1 Stylistic conventions

In order that the Library provide a cohesive interface, we use the following notational conventions in naming modules and their components:

- Signature names are given in all upper-case letters, possibly separated by underscores (e.g., `A_SIGNATURE_NAME`).
- Structure and functor names begin with a upper-case letter, and use upper-case letters (not underscores) to mark words (e.g., `AStructureName`).
- Type names are all lower-case letter, possibly separated by underscores (e.g., `a_type_name`).
- Value identifiers begin with an initial lower-case letter, and use upper-case letters (not underscores) to mark words (e.g., `aValueName`).
- Exception and datatype constructors always start with an initial upper-case letter.

We encourage future contributors to follow these conventions.

In addition to the above notational conventions, we have adopted the following programming conventions:

- Uncurried forms are favored, except where partial application is likely.
- Modules generally do *not* provide operations that do input or output (e.g., `print` functions). While such functions may be useful in interactive use, they add an overhead on stand-alone applications.

1.2 Contributors

This library has been organized by Emden Gansner and John Reppy, of AT&T Bell Laboratories. The library was initially developed as part of eXene, but most of the source code was adapted from other sources (such as the SML/NJ compiler sources and the SourceGroups library). The follow is a list of known contributors and their affiliations at the time of their contributions:

Stephan Adams	University of Southampton
William Aitken	Cornell University
Andrew Appel	Princeton University
Pierre Crégut	AT&T Bell Laboratories
Dave Berry	University of Edinburgh
Emden Gansner	AT&T Bell Laboratories
Lal George	AT&T Bell Laboratories
David MacQueen	AT&T Bell Laboratories
Greg Morrisett	Carnegie Mellon University
Larry Paulson	University of Cambridge
John Reppy	Cornell University, AT&T Bell Laboratories
Gene Rollins	Carnegie Mellon University

2 Installation

The file `load-all` is an SML script that will load all of the library modules into the top-level environment (using `use`). This image can then be exported (using `exportML`) to produce a version of SML/NJ with the library pre-loaded. For people who use SourceGroups, the file `library.dep` defines the inter-module dependencies in SourceGroups format.

3 Bug reports

Please send bug reports to `sml-bugs@research.att.com`, using the fomat given in `lib-bug-form`.

4 Manpages

The remainder of this document consists of UNIX style manual pages. Section 1 manual pages cover utilities that extend the interactive top-level programming environment. Section 3 manual pages cover library modules that are useful for application building.

NAME

LibBase — base definitions for the SML/NJ library.

SYNOPSIS

signature LIB_BASE
structure LibBase : LIB_BASE

SIGNATURE

```
exception Unimplemented of string
exception Impossible of string
exception BadArg of string

val badArg      : {module : string, func : string, msg : string} -> 'a

datatype relation = Equal | Less | Greater

val version     : {major : int, minor : int, date : string}
val versionName : string
```

DESCRIPTION

This module provides a set of common types, exceptions, and utility functions for the library. The exception `Unimplemented` is raised to report unimplemented features. The exception `Impossible` is raised to report internal errors. The exception `BadArg` is raised to report semantically incorrect arguments. For consistency, the string should include the library module and function names. The function `badArg` is provided for this purpose. The `relation` type is returned by various collating functions. The record value `version` gives information about the version of the library, and the string value `versionName` is the name of the version.

FILES

$SMLLIB/lib-base-sig.sml	contains the `LIB_BASE` signature.
$SMLLIB/lib-base.sml	contains the `LibBase` structure.

NAME

Array2 — two-dimensional arrays.

SYNOPSIS

signature ARRAY2
structure Array2 : ARRAY2

SIGNATURE

```
type 'a array2

exception Size
exception Subscript

val array      : (int * int * '1a) -> '1a array2
val tabulate   : (int * int * ((int * int) -> '1a)) -> '1a array2
val sub        : ('a array2 * int * int) -> 'a
val update     : ('a array2 * int * int * 'a) -> unit
val dimensions : 'a array2 -> (int * int)
val row        : ('1a array2 * int) -> '1a Array.array
val column     : ('1a array2 * int) -> '1a Array.array
```

DESCRIPTION

The `Array2` structure provides two-dimensional arrays organized in row-major order.

array (*m*, *n*, *v*)
: creates an $m \times n$ array with each element initialized to *v*. If *m* or *n* is less than zero, then the exception `Size` is raised.

tabulate (*m*, *n*, *f*)
: creates an $m \times n$ array, where the *(i,j)* element is initialized to *f(i,j)*. If *m* or *n* is less than zero, then the exception `Size` is raised.

sub (*a*, *i*, *j*)
: returns the *(i,j)* element of the array *a*. If either *i*, or *j* is out of range, then the `Subscript` exception is raised.

update (*a*, *i*, *j*, *x*)
: sets the *(i,j)* element of *a* to *x*. If either *i*, or *j* is out of range, then the `Subscript` exception is raised.

dimensions *a*
: returns the number of rows and columns in the array *a*.

row (*a*, *i*)
: projects the *i*th row of *a*. If *i* is out of range, then the `Subscript` exception is raised.

column (*a*, *j*)
: projects the *j*th column of *a*. If *j* is out of range, then the `Subscript` exception is raised.

Note that the exceptions `Size` and `Subscript` are different than the one-dimensional array exceptions.

FILES

$SMLLIB/array2-sig.sml	contains the `ARRAY2` signature.
$SMLLIB/array2.sml	contains the `Array2` structure.

NAME

ARRAY_SORT — signature for in-place sorting of arrays

SYNOPSIS

signature ARRAY_SORT

SIGNATURE

```
val sort   : (('a * 'a) -> LibBase.relation) -> 'a Array.array -> 'a Array.array
val sorted : (('a * 'a) -> LibBase.relation) -> 'a Array.array -> bool
```

DESCRIPTION

sort *cmp a*
: sorts the array *a* in non-decreasing order using the comparison relationship defined by *cmp*.

sorted *cmp a*
: returns true if the array *a* is sorted in non-decreasing order under the comparison predicate *cmp*.

FILES

$SMLLIB/array-sort-sig.sml contains the `ARRAY_SORT` signature.

SEE ALSO

ArrayQSort(LIB)

NAME

ArrayQSort — in-place sorting of arrays

SYNOPSIS

signature ARRAY_SORT
structure ArrayQSort : ARRAY_SORT

SIGNATURE

```
val sort   : (('a * 'a) -> LibBase.relation) -> 'a Array.array -> 'a Array.array
val sorted : (('a * 'a) -> LibBase.relation) -> 'a Array.array -> bool
```

DESCRIPTION

The structure `ArrayQSort` provides functions for the in-place sorting of arrays. It uses a tuned version of quicksort due to Bentley and McIlroy.

sort *cmp a*
sorts the array *a* in non-decreasing order using the comparison relationship defined by *cmp*.

sorted *cmp a*
returns true if the array *a* is sorted in non-decreasing order under the comparison predicate *cmp*.

FILES

$SMLLIB/array-qsort.sml	contains the `ArrayQSort` structure.

SEE ALSO

ARRAY_SORT(LIB)

NAME

BinaryDict — binary tree dictionary functor

SYNOPSIS

functor BinaryDict (K : ORD_KEY) : DICT

SIGNATURE ORD_KEY

```
type ord_key

val cmpKey : (ord_key * ord_key) -> LibBase.relation
```

SIGNATURE DICT

```
structure Key : ORD_KEY

type 'a dict

exception NotFound

val mkDict    : unit -> '1a dict
val insert    : ('1a dict * Key.ord_key * '1a) -> '1a dict
val find      : ('a dict * Key.ord_key) -> 'a
val peek      : ('a dict * Key.ord_key) -> 'a option
val remove    : ('1a dict * Key.ord_key) -> ('1a dict * '1a)
val numItems  : 'a dict -> int
val listItems : 'a dict -> (Key.ord_key * 'a) list
val app       : ((Key.ord_key * 'a) -> 'b) -> 'a dict -> unit
val revapp    : ((Key.ord_key * 'a) -> 'b) -> 'a dict -> unit
val fold      : ((Key.ord_key * 'a * 'b) -> 'b) -> 'a dict -> 'b -> 'b
val revfold   : ((Key.ord_key * 'a * 'b) -> 'b) -> 'a dict -> 'b -> 'b
val map       : ((Key.ord_key * 'a) -> '2b) -> 'a dict -> '2b dict
val transform : ('a -> '2b) -> 'a dict -> '2b dict
```

DESCRIPTION

The `BinaryDict` functor implements applicative-style maps given a structure defining an ordered key type. It is based on Stephen Adams' integer set code, which uses binary trees of bounded balance. The semantics expected of the functions are obvious; see the `DICT` man pages for a more complete description. The `listItems` function produces a list in increasing order of the key. The `fold` and `revapp` functions traverse the dictionary in decreasing order of the keys. The `revfold`, `map` and `app` functions traverse the dictionary in increasing order of the keys. A key/value pair inserted into a dictionary will replace a pair with an equal key.

FILES

$SMLLIB/binary-dict.sml contains the `BinaryDict` functor.

SEE ALSO

DICT(LIB), ORD_KEY(LIB), SplayDict(LIB)

NAME

BinarySet — binary tree set functor

SYNOPSIS

functor BinarySet (K : ORD_KEY) : ORD_SET

SIGNATURE ORD_KEY

```
type ord_key

val cmpKey : (ord_key * ord_key) -> LibBase.relation
```

SIGNATURE ORD_SET

```
type item
type set

exception NotFound

val empty        : set
val singleton    : item -> set
val add          : (set * item) -> set
val find         : (set * item) -> item
val peek         : (set * item) -> item option
val member       : (set * item) -> bool
val delete       : (set * item) -> set
val numItems     : set ->  int
val union        : (set * set) -> set
val intersection : (set * set) -> set
val difference   : (set * set) -> set
val listItems    : set -> item list
val app          : (item -> 'b) -> set -> unit
val revapp       : (item -> 'b) -> set -> unit
val fold         : ((item * 'b) -> 'b) -> set -> 'b -> 'b
val revfold      : ((item * 'b) -> 'b) -> set -> 'b -> 'b
```

DESCRIPTION

The `BinarySet` functor implements applicative sets on an ordered type. The resulting structure will satisfy `ord_key = item`. It is based on Stephen Adams' integer set code, which uses binary trees of bounded balance. The semantics expected of the functions are obvious; see the `ORD_SET` man pages for a more complete description. The `listItems` function produces a list in increasing order of the key. The `fold` and `revapp` functions traverse the set in decreasing order of the items. The `revfold` and `app` functions traverse the set in increasing order of the items. An item inserted into a set will replace an equal item.

FILES

$SMLLIB/binary-set.sml contains the `BinarySet` functor.

SEE ALSO

IntSet(LIB), ORD_SET(LIB), ORD_KEY(LIB), SplaySet(LIB)

NAME

CharSet — sets of characters

SYNOPSIS

signature CHAR_SET
structure CharSet : CHAR_SET

SIGNATURE

```
type char_set

val mkCharSet     : string -> char_set
val charSetOfList : int list -> char_set

val charsOfSet    : char_set -> string
val inSetOrd      : char_set -> int -> bool
val inSet         : char_set -> (string * int) -> bool
```

DESCRIPTION

Fast, read-only, character sets. These are meant to be used to construct predicates for the functions in the `Strings` structure.

mkCharSet *s*
makes a character set consisting of the characters in *s*.

charSetOfList *l*
make a character set consisting of the characters whose ordinals are given by the integer list *l*. The exception `LibBase.BadArg` is raised if any of the ordinals is out of range.

charsOfSet *cset*
returns a string representation from *cset*.

inSetOrd *cset i*
returns true if the character with the ordinal *i* is in *cset*.

inSet *cset* (*s*, *i*)
returns true if the *i*th character of *s* is in *cset*.

FILES

$SMLLIB/charset-sig.sml	contains the `CHAR_SET` signature.
$SMLLIB/charset.sml	contains the `CharSet` structure.

SEE ALSO

CType(LIB), Strings(LIB)

NAME

CType — character classification and conversion functions

SYNOPSIS

signature CTYPE
structure CType : CTYPE

SIGNATURE CTYPE

```
val isAlpha    : (string * int) -> bool
val isUpper    : (string * int) -> bool
val isLower    : (string * int) -> bool
val isDigit    : (string * int) -> bool
val isXDigit   : (string * int) -> bool
val isAlphaNum : (string * int) -> bool
val isSpace    : (string * int) -> bool
val isPunct    : (string * int) -> bool
val isGraph    : (string * int) -> bool
val isPrint    : (string * int) -> bool
val isCntrl    : (string * int) -> bool
val isAscii    : (string * int) -> bool

val isAlphaOrd    : int -> bool
val isUpperOrd    : int -> bool
val isLowerOrd    : int -> bool
val isDigitOrd    : int -> bool
val isXDigitOrd   : int -> bool
val isAlphaNumOrd : int -> bool
val isSpaceOrd    : int -> bool
val isPunctOrd    : int -> bool
val isGraphOrd    : int -> bool
val isPrintOrd    : int -> bool
val isCntrlOrd    : int -> bool
val isAsciiOrd    : int -> bool

val toAscii    : string -> string
val toUpper    : string -> string
val toLower    : string -> string
val toAsciiOrd : int -> int
val toUpperOrd : int -> int
val toLowerOrd : int -> int
```

DESCRIPTION

The predicate functions come in two flavors: test of a character in a string, and test of an integer. The expression `isFoo(s,i)` is equivalent to `isFooOrd(ordof(s,i))`, and returns true, if the specified character is in the "*Foo*" character class. The character classes are defined as follows:

Character class	Description
Alpha	letters (`A-Z`, `a-z`)
Upper	upper-case letters (`A-Z`)
Lower	lower-case letters (`a-z`)
Digit	digits (`0-9`)
XDigit	hexadecimal digits (`0-9`, `A-F`, `a-f`)
AlphaNum	alphanumeric characters (`0-9`, `A-Z`, `a-z`)
Space	white-space characters (space, tab, return, newline, formfeed, or vertical tab)
Punct	punctuation characters (neither control nor alphanumeric)
Graph	visible graphic characters (not including space)
Print	printing characters (includes space)
Cntrl	control characters (including delete)
Ascii	characters with ordinals between `0` and `127`.

In addition to the predicates, there are conversion functions, which also have two flavors. The expression `toFoo` *s* is equivalent to the expression `implode(map toFooOrd (explode` *s*`))`. The "Ord" version of the functions are defined as follows:

toAsciiOrd *c*
: if the integer (character) *c* is outside the range `0..127`, then *c* mod `128` is returned; otherwise *c* is returned.

toUpperOrd *c*
: if `(isLowerOrd` *c*`)` is true, then the upper-case version of *c* is returned; otherwise *c* is returned.

toLowerOrd *c*
: if `(isUpperOrd` *c*`)` is true, then the upper-case version of *c* is returned; otherwise *c* is returned.

FILES

$SMLLIB/ctype-sig.sml	contains `CTYPE` signature
$SMLLIB/ctype.sml	contains `CType` structure

SEE ALSO

CharSet(LIB)

CAVEATS

Currently the `isFoo` functions raise the `Ord` exception on out-of-bound indices; it might be more useful to return false.

NAME

DICT — signature for applicative dictionaries

SYNOPSIS

signature DICT

SIGNATURE

```
structure Key : ORD_KEY

type 'a dict

exception NotFound

val mkDict    : unit -> '1a dict
val insert    : ('1a dict * Key.ord_key * '1a) -> '1a dict
val find      : ('a dict * Key.ord_key) -> 'a
val peek      : ('a dict * Key.ord_key) -> 'a option
val remove    : ('1a dict * Key.ord_key) -> ('1a dict * '1a)
val numItems  : 'a dict -> int
val listItems : 'a dict -> (Key.ord_key * 'a) list
val app       : ((Key.ord_key * 'a) -> 'b) -> 'a dict -> unit
val revapp    : ((Key.ord_key * 'a) -> 'b) -> 'a dict -> unit
val fold      : ((Key.ord_key * 'a * 'b) -> 'b) -> 'a dict -> 'b -> 'b
val revfold   : ((Key.ord_key * 'a * 'b) -> 'b) -> 'a dict -> 'b -> 'b
val map       : ((Key.ord_key * 'a) -> '2b) -> 'a dict -> '2b dict
val transform : ('a -> '2b) -> 'a dict -> '2b dict
```

DESCRIPTION

The `DICT` signature provides an abstract description of applicative dictionaries/tables/associative arrays.

mkDict ()
Produces a new empty dictionary.

insert (*dict*, *key*, *v*)
Insert the key/value pair (*key*,*v*), producing a new dictionary.

find (*dict*, *key*)
Find the value corresponding to the given key. Raises the exception `NotFound` if no such value exists.

peek (*dict*, *key*)
Find the value corresponding to the given key. Returns `NONE` if no such value exists.

remove (*dict*, *key*)
Remove the key/value pair corresponding to the given key. Raises the exception `NotFound` if no such pair exists.

numItems *dict*
Return the number of pairs in the dictionary.

listItems *dict*
Return a list of the pairs in the dictionary.

transform *mapfn dict*
Creates a new dictionary whose entries are (*key*, *mapfn value*) for all entries (*key*, *value*) in *dict*.

If we assume a dictionary is implemented as a list of key/value pairs, stored in increasing order of the keys, the functions `app`, `revapp`, `fold`, `revfold` and `map` should behave like the analogous `List` functions.

FILES

$SMLLIB/dict-sig.sml	contains `DICT` signature

SEE ALSO

BinaryDict(LIB), IntMap(LIB), ORD_KEY(LIB), SplayDict(LIB)

NAME

DynamicArray — dynamic array functor

SYNOPSIS

signature STATIC_ARRAY
signature DYNAMIC_ARRAY
functor DynamicArray (A : STATIC_ARRAY) : DYNAMIC_ARRAY

SIGNATURE STATIC_ARRAY

```
type elem
type array

exception Subscript
exception Size

val array       : (int * elem) -> array
val sub         : (array * int) -> elem
val update      : (array * int * elem) -> unit
val length      : array -> int
val tabulate    : (int * (int -> elem)) -> array
val arrayoflist : elem list -> array
```

SIGNATURE DYNAMIC_ARRAY

```
type elem
type array

exception Subscript
exception Size

val array       : (int * elem) -> array
val subArray    : (array * int * int) -> array
val arrayoflist : (elem list * elem) -> array
val tabulate    : (int * (int -> elem) * elem) -> array
val sub         : (array * int) -> elem
val update      : (array * int * elem) -> unit
val default     : array -> elem
val bound       : array -> int
```

DESCRIPTION

The `DynamicArray` functor takes an array structure as an argument. It produces a structure implementing dynamic arrays. These act like arrays of unbounded length. The type `elem` is shared with the argument structure, as are the exception values.

array (*sz*, *v*)
: Creates an unbounded array, all of whose elements are initialized to *v*. The parameter *sz* is used as a hint of the upper bound on non-default elements. Raises the exception `Size` if *sz* `< 0`.

subArray (*array*, *lo*, *hi*)
: Creates a new array with the same default value as *array*, and whose values in the range `[0,`*hi-lo*`]` equal the values in *array* in the range `[`*lo*`,`*hi*`]`. Raises the exception `Size` if *hi* `<` *lo*.

default *array*
: Returns the default value of the dynamic array.

bound *array*
: Returns an upper bound on the indices of non-default values.

The functions `arrayoflist`, `sub`, `tabulate` and `update` are analogues to the similar functions in the pervasive `Array` structure, raising similar exceptions. The final parameter to `arrayoflist` and `tabulate` specifies the default value.

FILES

$SMLLIB/static-array-sig.sml	contains the `STATIC_ARRAY` signature.
$SMLLIB/dynamic-array-sig.sml	contains the `DYNAMIC_ARRAY` signature.
$SMLLIB/dynamic-array.sml	contains the `DynamicArray` functor.

NAME

Fifo — applicative queue

SYNOPSIS

signature FIFO
structure Fifo : FIFO

SIGNATURE

```
type 'a fifo

exception Dequeue

val empty    : 'a fifo
val isEmpty  : 'a fifo -> bool
val enqueue  : ('a fifo * 'a) -> 'a fifo
val dequeue  : 'a fifo -> ('a fifo * 'a)
val head     : 'a fifo -> 'a
val length   : 'a fifo -> int
val contents : 'a fifo -> 'a list
val app      : ('a -> 'b) -> 'a fifo -> unit
val revapp   : ('a -> 'b) -> 'a fifo -> unit
val map      : ('a -> 'b) -> 'a fifo -> 'b fifo
val fold     : (('a * 'b) -> 'b) -> 'b -> 'a fifo -> 'b
val revfold  : (('a * 'b) -> 'b) -> 'b -> 'a fifo -> 'b
```

DESCRIPTION

The `Fifo` structure implements applicative queues.

empty

An empty queue.

isEmpty *fifo*

Returns true if and only if *fifo* is empty.

enqueue (*fifo***,** *value***)**

Create a new queue by appending *value* to *fifo*.

dequeue *fifo*

Return head of *fifo*, plus the remainder of *fifo*. Raise exception `Dequeue` if the queue is empty.

head *fifo*

Returns the head of the queue. Raise exception `Dequeue` if the queue is empty.

length *fifo*

Returns the number of elements in the queue.

contents *fifo*

Returns a list of the elements in the queue in queue order.

The functions `app`, `revapp`, `map`, `fold` and `revfold` are analogues of the functions in the pervasive structure `List`.

FILES

$SMLLIB/fifo-sig.sml	contains `FIFO` signature
$SMLLIB/fifo.sml	contains `Fifo` structure

SEE ALSO

Queue(LIB)

NAME

Finalizer — object finalization functor

SYNOPSIS

signature FINALIZED_OBJ
signature FINALIZER
functor Finalizer (Obj : FINALIZED_OBJ) : FINALIZER

SIGNATURE FINALIZED_OBJ

```
type object
type objinfo

val finalize : objinfo -> unit
```

SIGNATURE FINALIZER

```
structure Obj : FINALIZED_OBJ

val registerObj : (Obj.object * Obj.objinfo) -> unit
val getDead     : unit -> Obj.objinfo list
val finalize    : unit -> unit
```

DESCRIPTION

The `Finalizer` functor provides a mechanism for building a finalization registry for a class of objects. The registry associates a finalization operation to be performed on an object once the garbage collector determines that it is unreachable.

registerObj (*obj*, *info*)
Register *obj* for finalization. It is important that *info* not contain any reference to *obj*, otherwise *obj* will never become free.

getDead ()
Return a list of registered dead objects, and remove them from the registry.

finalize ()
Finalize all registered dead objects and remove them from the registry.

Note that these operations do not force garbage collection.

FILES

$SMLLIB/finalize-sig.sml	contains the `FINALIZED_OBJ` and `FINALIZER` signatures.
$SMLLIB/finalizer.sml	contains the `Finalizer` functor.

SEE ALSO

System.Control.Unsafe.CInterface(2)

CAVEATS

On each garbage collection, the garbage collector must check for dead finalized objects; thus having large numbers of finalized objects can slow a system down.

It is important to realize that because of garbage collection, the finalization of objects may be significantly delayed from the time that they actually become dead (this is especially true for generational collectors).

NAME

Format — create formatted strings

SYNOPSIS

signature FORMAT
structure Format : FORMAT

SIGNATURE

```
datatype fmt_item
  = INT of int
  | BOOL of bool
  | STR of string
  | REAL of real
  | LEFT of (int * fmt_item)
  | RIGHT of (int * fmt_item)

exception BadFormat
exception BadArgList
exception BadInput of fmt_item list

val format  : string -> fmt_item list -> string
val formatf : string -> (string -> unit) -> fmt_item list -> unit

val scan    : string -> string -> fmt_item list
val scani   : string -> (string * int) -> (fmt_item list * int)
```

DESCRIPTION

The `Format` structure provides string formatting and scanning along the lines of the ANSI C **sprintf**, and **sscanf** functions. The `fmt_item` type is a tagged union of the basic types, plus two padding specifiers (`LEFT` and `RIGHT`), which are only used by the formatting functions. These functions are curried, since there is precompilation done of the format specifier strings; if a specifier string is ill-formed, then the exception `BadFormat` is raised.

The formatting operations are as follows:

format *fmt items*
formats the list of *items* using the format specified by *fmt*, returning the formatted string.

formatf *fmt sink items*
formats the list of *items* using the format specified by *fmt*, feeding the result to *sink*.

A format specification is a string, which contains two types of objects: plain characters, which are copied to the output, and conversion specifications, which direct the conversion of values from the *items* list into strings. The kind of the item constructor and the conversion specifier must match (as described below); a type mismatch results in the exception `BadArgList` being raised. If the item is `LEFT(`*w*`,` *item*`)`, the formatted *item* is left justified in a field of minimum width *w* (likewise for the `RIGHT` constructor). Note that if the formated *item* is wider than *w*, then there is no effect. Each conversion specification is introduced by the "`%`" character, after which the following appear in sequence:

- Zero or more flags, which modify the meaning of the conversion specification.
- An optional minimal field width, specified as a decimal number.
- An optional *precision* that specifies the number of digits to the right of the decimal point for the `e`, `E`, and `f` conversions, and maximum number of significant digits for the `g` and `G` conversions. The precision is specified as a decimal point ("`.`") followed by a decimal number.

- A character that specifies the conversion to be applied.

The sequence "`%%`" is used to generate a "`%`" character. The flag characters, and their meanings are:

`-` The result of the conversion will be left-justified within the field (the default is right justification).

`~` Use "`~`" as the negation character (otherwise "`-`" is used).

`+` The result of a numeric conversion will always begin with a plus or minus sign.

space If the first character of a numeric conversion is not a sign, then a space will be prefixed to the result (if both "`+`" and *space* are specified, then the `BadFormat` exception will be raised).

`#` The result will be converted to alternate form. For `o` conversions, it forces the first digit to be zero; for `x` (or `X`) conversions, a "`0x`" (or "`0X`") is prefixed; for `e`, `E`, and `f` conversions, the result will always contain a decimal, even if no digits follow; and for `g` and `G` conversions, trailing zeros are not removed. There is no effect on other conversions.

`0` Right-justified numeric conversions are zero-padded on the left (after any leading sign and base).

The conversion specifiers, and their meanings are:

`d` The argument `INT` *n* is converted to signed decimal.

`x,X` The argument `INT` *n* is converted to signed hexidecimal; the letters `abcdef` are used for `x` conversions, and the letters `ABCDEF` are used for `X` conversions..

`o` The argument `INT` *n* is converted to signed octal.

`c` The argument `INT` *n* is converted to a single character; the exception `Chr` is raised if *n* is out of range.

`s` The argument `STR` *s* is copied to the output.

`b` The argument `BOOL` *b* is converted to string representation (`"true"` or `"false"`).

`f` The argument `REAL` *r* is converted to the style "[`-`]*ddd*`.`*ddd*," where the number of digits to the right of the decimal point is given by the precision specification. If the precision is missing, it is taken to be 6; if the precision is zero and the `#` flag is not specified, then no decimal point appears. If there is a decimal point, then at least one digit appears before it.

`e,E` The argument `REAL` *r* is converted to the style "[`-`]*d*`.`*ddd*`e`[`-`]*dd*," where there is one digit before the decimal point (which is non-zero, if *r* is non=zero), and the number of digits after the decimal point is equal to the precision specification. If the precision is missing, it is taken to be 6; if the precision is zero and the `#` flag is not specified, then no decimal point appears. The `E` specification produces an "`E`" character instead of "`e`."

`g,G` The argument `REAL` *r* is converted in style `f` or `e`.

The scanning operations are as follows:

scan *fmt input*
scans the string *input* using the format specifier *fmt*, and returns a pair of the list of scanned items and the index of the next character to be scanned.

scani *fmt* **(***input***,** *i***)**
: scans the string *input*[*i*..] using the format specifier *fmt*, and returns a pair of the list of scanned items and the index of the next character to be scanned.

Note that the scanning functions never produce `LEFT` or `RIGHT` values. The interpretation of conversion specifiers by the scanning routines is less complicated. While conversion specifier syntax is the same as for the formatting operations, only the conversion character is meaningful (except in the case of `%s`).

`d` a decimal integer *n* is scanned and returned as `INT` *n*.

`x,X` a hexidecimal integer *n* is scanned and returned as `INT` *n*.

`o` an octal integer *n* is scanned and returned as `INT` *n*.

`c` a single character is scanned, and its ordinal is returned as `INT` *n*.

`s` a white-space delimited string is scanned; if a width *w* is specified, then no more than *w* characters are scanned.

`b` a boolean value *b* is scanned and returned as `BOOL` *b*.

`f,e,E,g,G` a floating-point number *r* (in either floating-point or scientific notation) is scanned and returned as `REAL` *r*.

If the input is badly formed, the exception `BadInput` is raised with a list of the items scanned upto the input error.

FILES

$SMLLIB/format-sig.sml	contains the `FORMAT` signature
$SMLLIB/format.sml	contains the `Format` structure

SEE ALSO

Makestring(LIB), StringCvt(LIB)

NAME

HashString — computing hash keys on strings

SYNOPSIS

structure HashString : sig ... end

SIGNATURE

```
val hashString : string -> int
```

DESCRIPTION

The `hashString` function computes an integer hash key for the given string argument.

FILES

$SMLLIB/hash-string.sml	contains `HashString` structure.

SEE ALSO

HashTable(LIB)

NAME

HashTable — hash table functor

SYNOPSIS

signature HASH_KEY
signature HASH_TABLE
functor HashTable (Key : HASH_KEY) : HASH_TABLE

SIGNATURE HASH_KEY

```
type hash_key

val hashVal : hash_key -> int
val sameKey : (hash_key * hash_key) -> bool
```

SIGNATURE HASH_TABLE

```
structure Key : HASH_KEY

type 'a hash_table

val mkTable     : (int * exn) -> '1a hash_table
val numItems    : 'a hash_table ->  int
val listItems   : 'a hash_table -> (Key.hash_key * 'a) list
val insert      : '2a hash_table -> (Key.hash_key * '2a) -> unit
val find        : 'a hash_table -> Key.hash_key -> 'a
val peek        : 'a hash_table -> Key.hash_key -> 'a option
val remove      : 'a hash_table -> Key.hash_key -> 'a
val apply       : ((Key.hash_key * 'a) -> 'b) -> 'a hash_table -> unit
val filter      : ((Key.hash_key * 'a) -> bool) -> 'a hash_table -> unit
val map         : ((Key.hash_key * 'a) -> '2b) -> 'a hash_table -> '2b hash_table
val transform   : ('a -> '2b) -> 'a hash_table -> '2b hash_table
val copy        : '1a hash_table -> '1a hash_table
val bucketSizes : 'a hash_table -> int list
```

DESCRIPTION

The `HashTable` functor takes a hash key structure as an argument. Hash keys are abstract objects with a hashing function and an equality function.

hashVal *key*

Compute an integer key from a hash key.

sameKey (*key1***,** *key2***)**

Return true if two keys are the same. **Note:** if `sameKey(`*h1*`, `*h2*`)`, then it must be the case that `(hashVal `*h1*` = hashVal `*h2*`)`.

The hash table operations have the following behavior:

mkTable (*sizeHint***,** *exn***)**

creates a new table; the int is a size hint and the exception is to be raised by `find`.

numItems *tbl*

Return the number of items in the table.

listItems *tbl*

Return a list of the items (and their keys) in the table.

insert *tbl* **(***key***,** *v***)**
Insert an item. If the key already has an item associated with it, then the old item is discarded.

find *tbl key*
Find an item, the table's exception is raised if the item doesn't exist.

peek *tbl key*
Look for an item, return NONE if the item doesn't exist.

remove *tbl key*
Remove an item, returning the item. The table's exception is raised if the item doesn't exist.

apply *f tbl*
Apply a function to the entries of the table.

filter *pred tbl*
Remove all items from *tbl* that do not satisfy the predicate *pred*.

map *f tbl*
Map a table to a new table that has the same keys.

transform *f tbl*
Map a table to a new table that has the same keys.

copy *tbl*
Create a copy of a hash table.

bucketSizes *tbl*
Return a list of the sizes of the various buckets. This allows users to gauge the quality of the hashing function.

FILES

$SMLLIB/hash-key-sig.sml	contains the `HASH_KEY` signature.
$SMLLIB/hash-table-sig.sml	contains the `HASH_TABLE` signature.
$SMLLIB/hash-table.sml	contains the `HashTable` functor.

SEE ALSO

HashString(LIB), PolyHashTable(LIB)

NAME

IntMap — applicative integer map

SYNOPSIS

signature INTMAP
structure IntMap : INTMAP

SIGNATURE

```
type 'a intmap

exception NotFound

val empty      : unit -> 'a intmap
val insert     : ('a intmap * int * 'a) -> 'a intmap
val find       : ('a intmap * int) -> 'a
val peek       : ('a intmap * int) -> 'a option
val remove     : ('a intmap * int) -> 'a intmap
val numItems   : 'a intmap ->  int
val listItems  : 'a intmap -> (int * 'a) list
val app        : ((int * 'a) -> 'b) -> 'a intmap -> unit
val revapp     : ((int * 'a) -> 'b) -> 'a intmap -> unit
val fold       : ((int * 'a * 'b) -> 'b) -> 'a intmap -> 'b -> 'b
val revfold    : ((int * 'a * 'b) -> 'b) -> 'a intmap -> 'b -> 'b
val map        : ((int * 'a) -> 'b) -> 'a intmap -> 'b intmap
val transform  : ('a -> 'b) -> 'a intmap -> 'b intmap
```

DESCRIPTION

The `IntMap` structure implements applicative integer maps. It is based on Stephen Adams' integer set code, which uses binary trees of bounded balance. The interface is compatible with that specified in the `DICT` signature, with `Key.ord_key = int`. See the `DICT` man pages for a more complete description. The `listItems` produces a list in increasing order of the integer component. The `fold` and `revapp` functions traverse the dictionary in decreasing order of the keys. The `revfold`, `map` and `app` functions traverse the dictionary in increasing order of the keys. A key/value pair inserted into a map will replace a pair with an equal key.

FILES

$SMLLIB/intmap-sig.sml	contains `INTMAP` signature
$SMLLIB/intmap.sml	contains `Intmap` structure

SEE ALSO

DICT(LIB), BinaryDict(LIB), SplayDict(LIB)

NAME

IntSet — applicative integer set

SYNOPSIS

signature INTSET
structure IntSet : INTSET

SIGNATURE

```
type intset

exception NotFound

val empty        : intset
val singleton    : int -> intset
val add          : (intset * int) -> intset
val member       : (intset * int) -> bool
val delete       : (intset * int) -> intset
val numItems     : intset ->  int
val union        : (intset * intset) -> intset
val intersection : (intset * intset) -> intset
val difference   : (intset * intset) -> intset
val listItems    : intset -> int list
val app          : (int -> 'b) -> intset -> unit
val revapp       : (int -> 'b) -> intset -> unit
val fold         : ((int * 'b) -> 'b) -> intset -> 'b -> 'b
val revfold      : ((int * 'b) -> 'b) -> intset -> 'b -> 'b
```

DESCRIPTION

The `IntSet` structure implements applicative integer sets. It is based on Stephen Adams' integer set code, which uses binary trees of bounded balance. The interface is compatible with that specified in the `ORD_SET` signature, with `Key.ord_key = int`, except there is neither a `find` nor a `peek` function. The semantics of the functions are obvious; see the `ORD_SET` man pages for a more complete description. The `listItems` function produces a list in increasing order. The `fold` and `revapp` functions traverse the set in decreasing order of the items. The `revfold` and `app` functions traverse the set in increasing order of the items.

FILES

$SMLLIB/intset-sig.sml	contains `INTSET` signature
$SMLLIB/intset.sml	contains `IntSet` structure

SEE ALSO

ORD_SET(LIB), BinarySet(LIB), SplaySet(LIB)

NAME

Iterate — general purpose iteration

SYNOPSIS

signature ITERATE
structure Iterate : ITERATE

SIGNATURE

```
val iterate : ('a -> 'a) -> int -> 'a -> 'a
val repeat  : (int * 'a -> 'a) -> int -> 'a -> 'a
```

DESCRIPTION

The `Iterate` structure implements various general purpose iterators involving higher-order functions.

iterate *f cnt*

Creates a function that is the composition of *cnt* copies of *f*. In particular, if *cnt* is 0, it is the identity function. Raises the exception `LibBase.BadArg` if *cnt* is negative.

repeat *f cnt v*

Repeatedly apply the function *f* to a pair whose first value if the iteration count, starting at 0, and whose second value is the result of the previous application, starting with the given value *v*, i.e., *f*(*cnt*`-1,`*f*`(...`*f*`(1,`*f*`(0,`*v*`))...))`. This is equivalent to `#2(iterate (fn (i,x) => (i+1,`*f*`(i,x))) `*cnt*` (0,`*v*`))`.

FILES

$SMLLIB/iterate-sig.sml	contains `ITERATE` signature
$SMLLIB/iterate.sml	contains `Iterate` structure

NAME

ListFormat — formatting and scanning of lists

SYNOPSIS

signature LIST_FORMAT
structure ListFormat : LIST_FORMAT

SIGNATURE

```
val formatList : {init : string, sep : string, final : string, fmt : 'a -> string}
      -> 'a list -> string

exception ScanList of int

val scanList :
      {init : string, sep : string, final : string, scan : (string * int) -> ('a * int)}
        -> (string * int)
          -> ('a list * int)
```

DESCRIPTION

formatList {*init*, *sep*, *final*, *fmt*} *l*

formats the list *l* using the given initial, separator and final strings. If *l* has the form `[`*a*`, ... ,`*b*`]`, the string *init*`^(`*fmt a*`)^`*sep*`...^`*sep*`^(`*fmt b*`)^`*final* will be returned.

scanList {*init*, *sep*, *final*, *scan*} (*s*, *i*)

scans the string *s* for a list of items, which begin with *init*, are separated by *sep*, and are terminated by *final*. The function *scan* is used to scan the individual items. Initial white space is ignored. If the string *s* is ill-formed, then the exception `ScanList` is raised with the position of the first error.

FILES

$SMLLIB/list-format-sig.sml	contains the signature `LIST_FORMAT`.
$SMLLIB/list-format.sml	contains the structure `ListFormat`.

SEE ALSO

Format(LIB), Makestring(LIB), StringCvt(LIB)

CAVEATS

When scanning a string, `scanList` checks for the separator before the terminator, thus if the separator is a prefix of the terminator, the function will not work.

NAME

LIST_SORT — generic interface for list sorting modules.

SYNOPSIS

signature LIST_SORT

SIGNATURE

```
val sort       : (('a * 'a) -> bool) -> 'a list -> 'a list
val uniqueSort : (('a * 'a) -> LibBase.relation) -> 'a list -> 'a list
val sorted     : (('a * 'a) -> bool) -> 'a list -> bool
```

DESCRIPTION

sort *gt l*
: sorts the list *l* in ascending order using the "greater-than" relationship defined by *gt*.

uniqueSort *cmp list*
: Returns a list of the items in *list* sorted in increasing order as determined by the relation function *cmp*. Only the first of equal items is retained.

sorted *gt l*
: returns true if the list *l* is sorted in ascending order under the "greater-than" predicate *gt*.

FILES

$SMLLIB/listsort-sig.sml contains the `LIST_SORT` signature.

SEE ALSO

ListMergeSort(LIB)

NAME

ListMergeSort — applicative list sorting using merge sort

SYNOPSIS

structure ListMergeSort : LIST_SORT

SIGNATURE

```
val sort       : (('a * 'a) -> bool) -> 'a list -> 'a list
val uniqueSort : (('a * 'a) -> LibBase.relation) -> 'a list -> 'a list
val sorted     : (('a * 'a) -> bool) -> 'a list -> bool
```

DESCRIPTION

The `ListMergeSort` structure implements applicative list sorting. The sort algorithms are implemented using a smooth merge sort, based on the implementation given in Paulson's book, pp. 99-100.

sort *gtr list*
Returns a list of the items in *list* sorted in nondecreasing order as determined by the "greater than" function *gtr*.

uniqueSort *cmp list*
Returns a list of the items in *list* sorted in increasing order as determined by the relation function *cmp*. Only the first of equal items is retained.

sort *gtr list*
Returns `true` if and only if the list is sorted in nondecreasing order as determined by the "greater than" function *gtr*.

FILES

$SMLLIB/list-mergesort.sml	contains `ListMergeSort` structure

SEE ALSO

LIST_SORT(LIB)

CAVEATS

The sorting algorithm is not stable.

NAME

ListUtil — list utility functions

SYNOPSIS

signature LIST_UTIL
structure ListUtil : LIST_UTIL

SIGNATURE

```
exception Zip
exception Split

val find      : ('a -> bool) -> 'a list -> 'a list
val findOne   : ('a -> bool) -> 'a list -> 'a option
val remove    : ('a -> bool) -> 'a list -> 'a list
val removeOne : ('a -> bool) -> 'a list -> 'a list
val filter    : ('a -> 'b option) -> 'a list -> 'b list
val splitp    : ('a -> bool) -> 'a list -> ('a list * 'a list)
val prefix    : ('a -> bool) -> 'a list -> 'a list
val suffix    : ('a -> bool) -> 'a list -> 'a list
val split     : int -> 'a list -> ('a list * 'a list)
val flatten   : 'a list list -> 'a list
val zip       : ('a list * 'b list) -> ('a * 'b) list
val unzip     : ('a * 'b) list -> ('a list * 'b list)
val fromTo    : (int * int) -> int list
val genList   : (int * (int -> 'a)) -> 'a list
```

DESCRIPTION

The `ListUtil` structure implements various general list functions. All of these functions are stable, in that elements in sublists maintain their respective ordering from an initial list.

find *pred l*

Returns a list of all items in *l* satisfying the predicate *pred*.

findOne *pred l*

Returns the first item in *l* satisfying the predicate *pred*; otherwise it returns `NONE`.

remove *pred l*

Returns a list of all items in *l* not satisfying the predicate *pred*. This is equivalent to `find (not o` *pred*`)` *l*.

removeOne *pred l*

Returns a list of all items in *l* except the first one satisfying the predicate *pred*.

filter *f l*

Maps the function *f* across the list *l*, discarding the `NONE`s.

splitp *pred l*

Splits the list *l* into two lists, the first being all items before the first item satisfying the predicate *pred*, the second consisting of the remaining items.

split *n l*

Splits the list *l* into two lists, the first being of length *n*. Raises the exception `LibBase.BadArg` if *n* `< 0` and the exception `Split` if `length` *l* `<` *n*.

prefix *pred l*

Returns the list of all items in *l* before the first item accepted by *pred*. This is equivalent to `#1(splitp` *pred l*`)`.

suffix *pred l*
: Returns the list of all items in *l* after the first item accepted by *pred*. This is equivalent to `(tl(#2(splitp` *pred l*`))) handle _ => []`.

flatten *ll*
: Flattens the list of lists *ll*, returning the catenation of the items in *ll*.

zip (*l1*, *l2*)
: Combines the two lists into a list of pairs, matching the first element in *l1* with the first element in *l2*, the second with the second, and so on. Raises the exception Zip if the two lists do not have equal length.

unzip *l*
: The inverse of `zip`.

genList (*len*, *genfn*)
: Generates a list of length *len* whose first item is *genfn* `0`, the second item is *genfn* `1`, etc. Raises `LibBase.BadArg` if *len* `< 0`.

fromTo (*lo*, *hi*)
: Generates a list of integers from *lo* to *hi*. Raises `LibBase.BadArg` if *lo* is greater than *hi*.

FILES

$SMLLIB/list-util-sig.sml	contains `LIST_UTIL` signature
$SMLLIB/list-util.sml	contains `ListUtil` structure

NAME

Makestring — convert primitive types to their string representation.

SYNOPSIS

signature MAKESTRING
structure Makestring : MAKESTRING

SIGNATURE

```
val padLeft      : (string * int) -> string
val padRight     : (string * int) -> string

val boolToStr    : bool -> string

val intToBin     : int -> string
val intToOct     : int -> string
val intToStr     : int -> string
val intToHex     : int -> string

exception BadPrecision

val realToFloStr : (real * int) -> string
val realToSciStr : (real * int) -> string

val realFFormat  : (real * int) -> {sign : bool, mantissa : string}
val realEFormat  : (real * int) -> {sign : bool, mantissa : string, exp : int}
val realGFormat  : (real * int)
      -> {sign : bool, whole : string, frac : string, exp : int option}
```

DESCRIPTION

padLeft (*s*, *n*)
: pads the string *s* with blanks on the left to length *n*. If the length of *s* is greater than *n*, then *s* is returned.

padRight (*s*, *n*)
: pads the string *s* with blanks on the right to length *n*. If the length of *s* is greater than *n*, then *s* is returned.

boolToStr *b*
: convert the boolean *b* to either `"true"` or `"false"`.

intToBin *i*
: convert the integer *i* to binary representation; if *i* is negative, then $-i$ is converted, and a leading “`~`” is added. Note that no leading “`0b`” is produced.

intToOct *i*
: convert the integer *i* to octal representation; if *i* is negative, then $-i$ is converted, and a leading “`~`” is added.

intToStr *i*
: convert the integer *i* to decimal representation; if *i* is negative, then $-i$ is converted, and a leading “`~`” is added.

intToHex *i*
: convert the integer *i* to hexidecimal representation; if *i* is negative, then $-i$ is converted, and a leading “`~`” is added. Note that no leading “`0x`” is produced.

realToFloStr (*r*, *prec*)
converts the real number *r* to a string of the form "`[~]`*ddd*`.`*ddd*," where the precision (number of fractional digits) is *prec*. The exception `BadPrecision` is raised if *prec* is less than zero.

realToSciStr (*r*, *prec*)
converts the real number *r* to a string of the form "`[~]`*d*`.`*ddd*`E[~]`*dd*," where the precision (number of fractional digits) is *prec*. The exception `BadPrecision` is raised if *prec* is less than zero.

In addition, there are three low-level real to string conversion functions provided.

realFFormat (*r*, *prec*)
returns the sign (true is negative) and the string representation of the real number *r*. The number of digits to the right of the decimal point is given by *prec*.

realEFormat (*r*, *prec*)
returns the sign (true is negative), the string representation of the mantissa, and the exponent of the real number *r*. The number of digits to the right of the decimal point in the mantissa is given by *prec*.

realGFormat (*r*, *prec*)

FILES

$SMLLIB/makestring-sig.sml	contains the `MAKESTRING` signature.
$SMLLIB/makestring.sml	contains the `Makestring` structure.

SEE ALSO

CvtString(LIB), Format(LIB)

NAME

Name — unique strings

SYNOPSIS

signature NAME
structure Name : NAME

SIGNATURE

```
type name

val mkName    : string -> name
val sameName  : (name * name) -> bool
val stringOf  : name -> string

type 'a name_tbl

val mkNameTbl : (int * exn) -> '1a name_tbl
val numItems  : 'a name_tbl -> int
val listItems : 'a name_tbl -> (name * 'a) list
val insert    : '2a name_tbl -> (name * '2a) -> unit
val remove    : 'a name_tbl -> name -> 'a
val find      : 'a name_tbl -> name -> 'a
val peek      : 'a name_tbl -> name -> 'a option
val apply     : ((name * 'a) -> 'b) -> 'a name_tbl -> unit
val filter    : ((name * 'a) -> bool) -> 'a name_tbl -> unit
val map       : ((name * 'a) -> '2b) -> 'a name_tbl -> '2b name_tbl
val transform : ('a -> '2b) -> 'a name_tbl -> '2b name_tbl
val copy      : '1a name_tbl -> '1a name_tbl
```

DESCRIPTION

The `Name` structure provides strings with constant-time equality/inequality testing.

mkName *s*
Map a string to the corresponding unique name.

sameName (*n1***,** *n2***)**
return true if the names are the same

stringOf *n*
return the string representation of the name

In addition to names, the `Name` structure also provides hash tables keyed by names.

mkNameTbl (*sizeHint***,** *exn***)**
creates a new table; the int is a size hint and the exception is to be raised by `find`.

numItems *tbl*
return the number of items in the table

listItems *tbl*
return the keyed list of table entries

insert *tbl* **(***n***,** *v***)**
insert a new entry; this will replace any previous entry with the same key.

find *tbl n*
find an entry; raise the table's exception if there is no entry with the given key.

peek *tbl n*
Look for an entry; return NONE if there is no entry with the given key.

remove *tbl n*
remove an entry, returning the item. If there is no entry with the given name, then raise the table's exception.

apply *f tbl*
apply a function to the entries of the table

filter *pred tbl*
removes any item in *tbl* that does not satisfy the predicate *pred*.

transform *f tbl*
map a table to a new table

copy *tbl*
creates a copy of *tbl*.

FILES

$SMLLIB/name-sig.sml	contains the `NAME` signature.
$SMLLIB/name.sml	contains the `Name` structure.

SEE ALSO

HashTable(LIB)

CAVEATS

The fast equality testing of names is broken if they are blasted out, and the read back in (inequality testing will still be fast).

NAME

ORD_KEY — signature for ordered keys

SYNOPSIS

signature ORD_KEY

SIGNATURE

```
type ord_key

val cmpKey : (ord_key * ord_key) -> LibBase.relation
```

DESCRIPTION

The `ORD_KEY` provides an abstract description of ordered keys. It specifies a type plus a trivalent comparison function.

FILES

$SMLLIB/ord-key-sig.sml	contains `ORD_KEY` signature

SEE ALSO

BinaryDict(LIB), BinarySet(LIB), DICT(LIB), ORD_SET(LIB), SplayDict(LIB), SplaySet(LIB)

NAME

ORD_SET — signature for applicative sets on ordered types

SYNOPSIS

signature ORD_SET

SIGNATURE

```
type item
type set

exception NotFound

val empty        : set
val singleton    : item -> set
val add          : (set * item) -> set
val find         : (set * item) -> item
val peek         : (set * item) -> item option
val member       : (set * item) -> bool
val delete       : (set * item) -> set
val numItems     : set ->  int
val union        : (set * set) -> set
val intersection : (set * set) -> set
val difference   : (set * set) -> set
val listItems    : set -> item list
val app          : (item -> 'b) -> set -> unit
val revapp       : (item -> 'b) -> set -> unit
val fold         : ((item * 'b) -> 'b) -> set -> 'b -> 'b
val revfold      : ((item * 'b) -> 'b) -> set -> 'b -> 'b
```

DESCRIPTION

The `ORD_SET` provides an abstract description of applicative-style sets on an ordered type.

empty
The empty set.

singleton *v*
Create a set containing only the item *v*.

add (*set*, *v*)
Create a set that is the union of *set* and {*v*}.

find (*set*, *v*)
Find the value *v* in the given set. Raises the exception `NotFound` if no such value exists.

peek (*set*, *v*)
Find the value *v* in the given set. Returns `NONE` if no such value exists.

delete (*set*, *v*)
Create a set that is the difference of *set* and {*v*}. Raises the exception `NotFound` if no such value exists.

numItems *set*
Return the number of values in the set.

union (*s*, *s'*)
Create a set that is the union of *s* and *s'*.

intersection (*s*, *s'*)
Create a set that is the intersection of *s* and *s'*.

difference (*s*, *s'*)
Create a set that is the difference of *s* and *s'*.

listItems *set*
Return a list of the items in the set.

If we assume a set is implemented as a list of values, stored in increasing order, the functions `app`, `revapp`, `fold` and `revfold` should behave like the analogous `List` functions. The `fold` and `revapp` functions traverse the set in decreasing order of the items. The `revfold` and `app` functions traverse the set in increasing order of the items.

FILES

$SMLLIB/ord-set-sig.sml contains `ORD_SET` signature

SEE ALSO

BinarySet(LIB), IntSet(LIB), SplaySet(LIB)

NAME

Pathname — support for pathname decomposition and search path lists.

SYNOPSIS

signature PATHNAME
structure Pathname : PATHNAME

SIGNATURE

```
val makePath       : {dir : string, name : string, ext : string} -> string
val splitPath      : string -> {dir : string, name : string, ext : string}

val dirOfPath      : string -> string
val nameOfPath     : string -> string
val extOfPath      : string -> string
val baseOfPath     : (string * string) -> string

val defaultExt     : {path : string, ext : string} -> string

val pathExplode    : string -> string list
val pathImplode    : string list -> string

val isAbsolutePath : string -> bool

val mkAbsolutePath : (string * string) -> string
val mkRelativePath : (string * string) -> string

type path_list

val mkSearchPath   : string -> path_list
val pathsOfList    : path_list -> string list
val appendPath     : (path_list * string) -> path_list
val prependPath    : (string * path_list) -> path_list

val findFile       : (path_list * (string -> bool)) -> string -> string
val findFiles      : (path_list * (string -> bool)) -> string -> string list
exception NoSuchFile
```

DESCRIPTION

The `Pathname` module provides routines for manipulating UNIX paths, and mostly follows the conventions of the Modula-3 Pathname module. A path is composed of four parts: the directory, the name, the separator, and the extension. A path can also be viewed as a sequence of *simple paths* separated by "`/`" characters. The directory part of a path is the prefix that ends with the rightmost "`/`" character. If the path contains no "`/`" characters, then the directory part is empty. The name, separator and extension comprise the rightmost simple path. The extension is the suffix that starts after the rightmost "`.`" character (which is the separator), providing that it is neither the first or last character of the simple path and that it is not immediately proceeded by a "`.`" character. If there is no such suffix, then the extension and separator are empty. The name part of a simple path is everything to the left of the separator. Since the separator occurs if, and only if, there is an extension, we consider it implicit. The following table illustrates the decomposition of paths:

Path	Directory	Name	Extension
`foo`		`foo`	
`old/foo.bar`	`old/`	`foo`	`bar`
`.tmp.sh`		`.tmp`	`sh`
`/`	`/`		
`/usr/local`	`/usr/`	`local`	
`x.`		`x.`	
`x..`		`x..`	
`x..z`		`x..z`	
`lang.grm.sml`		`lang.grm`	`sml`
`x.y..z`		`x`	`y..z`
`foo//.`	`foo//`	`.`	
`/home/joe/.login`	`/home/joe`	`.login`	
`~/bin`	`~/`	`bin`	
`$HOME/foo.o`	`$HOME/`	`foo`	`o`

The operations on paths are defined as follows:

makePath {*dir*, *name*, *ext*}
constructs a pathname out of its constituent parts.

splitPath *path*
splits *path* into its directory, name and extension parts.

dirOfPath *path*
returns the directory part of *path*.

nameOfPath *path*
returns the name part of *path*.

extOfPath *path*
returns the extension part of *path*.

baseOfPath (*path*, *ext*)
returns the non-directory part of *path* with the extension *ext* removed.

defaultExt {*path*, *ext*}
if *path* has a non-null name and does not have an extension, then add `".`*ext*`"`.

pathExplode *path*
splits *path* into a list of simple paths; if *path* begins with “`/`,” then the head of the result will be `""`, and if *path* ends with “`/`,” the tail of the result will be `""`.

pathImplode *l*
constructs a path from a list *l* of simple paths. If the head of *l* is `""`, then the result will have a leading “`/`.” Likewise, if the tail of *l* is `""`, then the result will end in “`/`.”

isAbsolutePath *path*
returns true if *path* is an absolute (or full) path (i.e., begins with “`/`”).

mkAbsolutePath (*path1*, *path2*)
returns *path1* if it is absolute; otherwise, it returns an absolute path that is formed by concatenating *path2* and *path1* (i.e., the absolute path corresponding to *path1* with respect to *path2*). If *path2* is not absolute, and even if *path1* is, the exception `LibBase.BadArg` is raised.

mkRelativePath (*path1***,** *path2***)**
returns *path1* if it is absolute; otherwise it returns an equivalent path relative to *path2*. If *path2* is not absolute, and even if *path1* is, the exception `LibBase.BadArg` is raised.

The `Pathname` module also supports search path lists (e.g., such as the defined by the shell variable **PATH**). The abstract type `path_list` is used to represent these, and the following operations are provided:

mkSearchPath *s*
make a search path list from a colon separated sequence of paths (i.e., a string of the form `"`*p1*`:`*p2*`:`*...*`:`*pn*`"`)

appendPath (*pathList***,** *path***)**
append a path onto a search path list

prependPath (*path***,** *pathList***)**
prepend a path onto a search path list.

pathsOfList *pathList*
returns the list of paths that comprise *pathList*.

findFile (*pathList***,** *pred***)** *name*
searches for a path that satisfies the predicate *pred* as follows. If *path* is absolute, then `findFile` returns *path* if it satisfies *pred*. If *path* is relative, then each directory in *pathList* is checked by appending *path* to see if it specifies a path that satisfies *pred*; the first such path found is returned. If no such file is found, then the exception `NoSuchFile` is raised.

findFiles (*pathList***,** *pred***)** *name*
searches for paths that satisfy the predicate *pred* as follows. If *path* is absolute, then `findFile` returns the singleton list `[`*path*`]` if *path* satisfies *pred*. If *path* is relative, then each directory in *pathList* is checked by appending *path* to see if it satisfies *pred*; a list of all such paths found is returned. If no such files are found, then the empty list is returned.

FILES

$SMLLIB/pathname-sig.sml	contains the `PATHNAME` signature.
$SMLLIB/pathname.sml	contains the `Pathname` structure.

SEE ALSO

UnixPath(LIB)

NAME

PolyHashTable — polymorphic hash tables

SYNOPSIS

signature POLY_HASH_TABLE
structure PolyHashTable : POLY_HASH_TABLE

SIGNATURE

```
type ('a, 'b) hash_table

val mkTable  : (('2a -> int) * (('2a * '2a) -> bool)) -> (int * exn)
      -> ('2a, '2b) hash_table
val numItems : ('a, 'b) hash_table ->  int
val listItems : ('a, 'b) hash_table -> ('a * 'b) list
val insert   : ('2a, '2b) hash_table -> ('2a * '2b) -> unit
val find     : ('a, 'b) hash_table -> 'a -> 'b
val peek     : ('a, 'b) hash_table -> 'a -> 'b option
val remove   : ('a, 'b) hash_table -> 'a -> 'b
val apply     : (('a * 'b) -> 'c) -> ('a, 'b) hash_table -> unit
val filter    : (('a * 'b) -> bool) -> ('a, 'b) hash_table -> unit
val map       : (('2a * 'b) -> '2c) -> ('2a, 'b) hash_table -> ('2a, '2c) hash_table
val transform : ('b -> '2c) -> ('2a, 'b) hash_table -> ('2a, '2c) hash_table
val copy      : ('1a, '1b) hash_table -> ('1a, '1b) hash_table
val bucketSizes : ('a, 'b) hash_table -> int list
```

DESCRIPTION

The `PolyHashTable` structure provides polymorphic hash tables. The hash table operations have the following behavior:

mkTable (*hash***,** *eq***) (***sizeHint***,** *exn***)**
: creates a new table using the hashing function *hash* and the equality predicate *eq*. The integer *sizeHint* a hint as to the initial size of the table, and the exception *exn* is raised by `find`.

numItems *tbl*
: Return the number of items in the table.

listItems *tbl*
: Return a list of the items (and their keys) in the table.

insert *tbl* **(***key***,** *v***)**
: Insert an item. If the key already has an item associated with it, then the old item is discarded.

find *tbl key*
: Find an item, the table's exception is raised if the item doesn't exist.

peek *tbl key*
: Look for an item, return NONE if the item doesn't exist.

remove *tbl key*
: Remove an item, returning the item. The table's exception is raised if the item doesn't exist.

apply *f tbl*
: Apply a function to the entries of the table.

filter *pred tbl*
: Remove all items from *tbl* that do not satisfy the predicate *pred*.

map *f tbl*
Map a table to a new table that has the same keys.

transform *f tbl*
Map a table to a new table that has the same keys.

copy *tbl*
Create a copy of a hash table.

bucketSizes *tbl*
Return a list of the sizes of the various buckets. This allows users to gauge the quality of the hashing function.

FILES

$SMLLIB/poly-hash-table-sig.sml	contains the `POLY_HASH_TABLE` signature.
$SMLLIB/poly-hash-table.sml	contains the `PolyHashTable` structure.

SEE ALSO

HashString(LIB), HashTable(LIB)

NAME

Queue — imperative queue

SYNOPSIS

signature QUEUE
structure Queue : QUEUE

SIGNATURE

```
type '1a queue

exception Dequeue

val mkQueue  : unit -> '1a queue
val isEmpty  : 'a queue -> bool
val enqueue  : ('a queue * 'a) -> unit
val dequeue  : 'a queue -> 'a
val head     : 'a queue -> 'a
val length   : 'a queue -> int
val contents : 'a queue -> 'a list
val app      : ('a -> 'b) -> 'a queue -> unit
val revapp   : ('a -> 'b) -> 'a queue -> unit
val fold     : (('a * 'b) -> 'b) -> 'b -> 'a queue -> 'b
val revfold  : (('a * 'b) -> 'b) -> 'b -> 'a queue -> 'b
val map      : ('a -> 'b) -> 'a queue -> 'b queue
```

DESCRIPTION

The `Queue` structure implements imperative queues.

mkQueue ()
Create an empty queue.

isEmpty *queue*
Returns true if and only if *queue* is empty.

enqueue (*queue***,** *value***)**
Append *value* to *queue*.

dequeue *queue*
Remove head of *queue*. Raise exception `Dequeue` if the queue is empty.

head *queue*
Returns the head of the queue, leaving the queue unchanged. Raise exception `Dequeue` if the queue is empty.

length *queue*
Returns the number of elements in the queue.

contents *queue*
Returns a list of the elements in the queue in queue order.

The functions `app`, `revapp`, `map`, `fold` and `revfold` are analogues of the functions in the pervasive structure `List`.

FILES

$SMLLIB/queue-sig.sml	contains `QUEUE` signature

$SMLLIB/queue.sml	contains `Queue` structure

SEE ALSO

Fifo(LIB)

NAME

Random — simple random number generator

SYNOPSIS

signature RANDOM
structure Random : RANDOM

SIGNATURE

```
val random   : real -> real
val mkRandom : real -> unit -> real
val norm     : real -> real
val range    : (int * int) -> real -> int
```

DESCRIPTION

The `Random` structure implements a simple random number generator with a uniform distribution. It is based on the one described in Paulson's book (pp. 170-171), which is derived from the article "Random number generators: good ones are hard to find," by Stephen K. Park and Keith W. Miller, *CACM* 31 (1988), pp. 1192-1201.

random *seed*

Generates a nonnegative real based on *seed*. Iteratively using the value returned by `random` as the next seed will produce a sequence of pseudo-random numbers.

mkRandom *seed*

Produces a function generating a sequence of pseudo-random numbers based on *seed*.

norm *rand*

Normalizes the values returned by `random` and `mkRandom` to the range [0.0,1.0).

range (*lo*,*hi*)

Produces a function that maps values produced by `random` and `mkRandom` into integers in the range [*lo*,*hi*]. Raises the exception `LibBase.BadArg` if *hi* < *lo*.

FILES

$SMLLIB/random-sig.sml	contains `RANDOM` signature
$SMLLIB/random.sml	contains `Random` structure

NAME

SplayDict — splay tree dictionary functor

SYNOPSIS

functor SplayDict (K : ORD_KEY) : DICT

SIGNATURE ORD_KEY

```
type ord_key

val cmpKey : (ord_key * ord_key) -> LibBase.relation
```

SIGNATURE DICT

```
structure Key : ORD_KEY

type 'a dict

exception NotFound

val mkDict    : unit -> '1a dict
val insert    : ('1a dict * Key.ord_key * '1a) -> '1a dict
val find      : ('a dict * Key.ord_key) -> 'a
val peek      : ('a dict * Key.ord_key) -> 'a option
val remove    : ('1a dict * Key.ord_key) -> ('1a dict * '1a)
val numItems  : 'a dict -> int
val listItems : 'a dict -> (Key.ord_key * 'a) list
val app       : ((Key.ord_key * 'a) -> 'b) -> 'a dict -> unit
val revapp    : ((Key.ord_key * 'a) -> 'b) -> 'a dict -> unit
val fold      : ((Key.ord_key * 'a * 'b) -> 'b) -> 'a dict -> 'b -> 'b
val revfold   : ((Key.ord_key * 'a * 'b) -> 'b) -> 'a dict -> 'b -> 'b
val map       : ((Key.ord_key * 'a) -> '2b) -> 'a dict -> '2b dict
val transform : ('a -> '2b) -> 'a dict -> '2b dict
```

DESCRIPTION

The `SplayDict` functor implements applicative maps given a structure defining an ordered key type. It is based on Sleator-Tarjan splay trees. The semantics expected of the functions are obvious; see the `DICT` man pages for a more complete description. The `listItems` produces a list in increasing order of the key. The `fold` and `revapp` functions traverse the dictionary in decreasing order of the keys. The `revfold`, `map` and `app` functions traverse the dictionary in increasing order of the keys. A key/value pair inserted into a dictionary will replace a pair with an equal key.

FILES

$SMLLIB/splay-dict.sml contains the `SplayDict` functor.

SEE ALSO

DICT(LIB), ORD_KEY(LIB), BinaryDict(LIB), SplayTree(LIB)

NAME

SplaySet — splay tree set functor

SYNOPSIS

functor SplaySet (K : ORD_KEY) : ORD_SET

SIGNATURE ORD_KEY

```
type ord_key

val cmpKey : (ord_key * ord_key) -> LibBase.relation
```

SIGNATURE ORD_SET

```
type item
type set

exception NotFound

val empty        : set
val singleton    : item -> set
val add          : (set * item) -> set
val find         : (set * item) -> item
val peek         : (set * item) -> item option
val member       : (set * item) -> bool
val delete       : (set * item) -> set
val numItems     : set ->  int
val union        : (set * set) -> set
val intersection : (set * set) -> set
val difference   : (set * set) -> set
val listItems    : set -> item list
val app          : (item -> 'b) -> set -> unit
val revapp       : (item -> 'b) -> set -> unit
val fold         : ((item * 'b) -> 'b) -> set -> 'b -> 'b
val revfold      : ((item * 'b) -> 'b) -> set -> 'b -> 'b
```

DESCRIPTION

The `SplaySet` functor implements applicative sets on an ordered type. It is based on Sleator-Tarjan splay trees. The semantics expected of the functions are obvious; see the `ORD_SET` man pages for a more complete description. The `listItems` function produces a list in increasing order of the key. The `fold` and `revapp` functions traverse the set in decreasing order of the items. The `revfold` and `app` functions traverse the set in increasing order of the items. An item inserted into a set will replace an equal item.

FILES

$SMLLIB/splay-set.sml	contains the `SplaySet` functor.

SEE ALSO

IntSet(LIB), ORD_SET(LIB), ORD_KEY(LIB), BinarySet(LIB), SplayDict(LIB)

NAME

SplayTree — splay tree data structure

SYNOPSIS

signature SPLAY_TREE
structure SplayTree : SPLAY_TREE

SIGNATURE

```
datatype 'a splay
  = SplayObj of {
        value : 'a,
        right : 'a splay,
        left : 'a splay
      }
  | SplayNil

val splay : (('a -> LibBase.relation) * 'a splay) -> (B.relation * 'a splay)
val join  : ('a splay * 'a splay) -> 'a splay
```

DESCRIPTION

The `SplayTree` structure provides the datatype and two basic functions necessary for applicative Sleator-Tarjan splay trees.

splay (*cmp*, *t*)

Returns `(`*r*`,` *t'*`)` where *t'* is *t* adjusted using the comparison function *cmp*. Usually, *cmp v* will compare *v* against some fixed value. For example, if *cmpfn* `: 'a * 'a -> relation` defines an ordered relation on the type `'a` and we wish to search for a value *u*, we can pass the function `fn` *v* `=>` *cmpfn*`(`*v*`,`*u*`)` to the `splay` function. If *t'* `= OBJ{`*value*`,...}`, then *r* `=` *cmp value*. *t'* is `NIL` if and only if *t* is `NIL`, in which case *r* is undefined.

join (*t*, *t'*)

Returns a new splay tree joining *t* and *t'*.

FILES

$SMLLIB/splaytree-sig.sml	contains `SPLAYTREE` signature
$SMLLIB/splaytree.sml	contains `SplayTree` structure

SEE ALSO

SplayDict(LIB), SplaySet(LIB)

CAVEATS

Not only is the data structure concrete, but the `splay` function takes a comparison function as an argument, allowing the semantics of the splay tree to be changed on the fly. It is assumed that this structure will only be used within another structure that will guarantee the consistency of the trees.

NAME

StringCvt — convert strings into primitive values.

SYNOPSIS

signature STRING_CVT
structure StringCvt : STRING_CVT

SIGNATURE

```
exception Convert

datatype radix = Bin | Oct | Dec | Hex

val strToInt  : radix -> (string * int) -> (int * int)
val batoi     : string -> int
val oatoi     : string -> int
val atoi      : string -> int
val xatoi     : string -> int

val strToReal : (string * int) -> (real * int)
val atof      : string -> real

val strToBool : (string * int) -> (bool * int)
val atob      : string -> bool
```

DESCRIPTION

The `StringCvt` module provides low-level functions for converting strings to primitive values. All of these functions ignore leading whitespace and recognize either "`~`" or "`-`" as negation signs. If incorrect input is encountered, the exception `Convert` is raised.

strToInt *radix* **(***s***,** *i***)**
: scans the string *s*[*i*..] for a base-*radix* integer. The scanned integer and the index of the next character are returned.

batoi *s*
: scans the string *s* for a base-2 integer.

oatoi *s*
: scans the string *s* for a base-8 integer.

atoi *s*
: scans the string *s* for a base-10 integer.

xatoi *s*
: scans the string *s* for a base-16 integer.

strToReal (*s***,** *i***)**
: scans the string *s*[*i*..] for a real number, which can be in either floating-point or scientific notation. The scanned real and the index of the next character are returned.

atof *s*
: scans the string *s* for a real number.

strToBool (*s***,** *i***)**
: scans the string *s*[*i*..] for a boolean literal. This is case insensitive; for example, the string `"tRuE"` will map to `true`. The scanned boolean and the index of the next character are returned.

atob *s*

scans the string *s* for a boolean literal.

FILES

$SMLLIB/string-cvt-sig.sml	contains `STRING_CVT` signature.
$SMLLIB/string-cvt.sml	contains `StringCvt` structure.

SEE ALSO

Format(LIB), Makestring(LIB)

NAME

StringUtil — string operations

SYNOPSIS

signature STRING_UTIL
structure StringUtil : STRING_UTIL

SIGNATURE

```
exception NotFound

val index       : string -> (string * int) -> int
val indexp      : (int -> bool) -> (string * int) -> int
val revindex    : string -> (string * int) -> int
val revindexp   : (int -> bool) -> (string * int) -> int

val spanp       : (int -> bool) -> (string * int) -> int
val span        : string -> (string * int) -> int

val cspanp      : (int -> bool) -> (string * int) -> int
val cspan       : string -> (string * int) -> int

val tokenizep   : (int -> bool) -> (string * int) -> string list
val tokenize    : string -> (string * int) -> string list

val findstr     : (string * int * string) -> int
val revfindstr  : (string * int * string) -> int

val strcmp      : (string * string) -> LibBase.relation

val isPrefix    : (string * string * int) -> bool

unequalAt       : (string * int * string * int) -> (int * int)

val prefixCmp   : (string * int * string * int) -> (bool * bool)

val suffix      : (string * int) -> string

val stringTrans : (string * string) -> string -> string
val stringMap   : (int -> string) -> string -> string

val compressStr : string -> string
val expandStr   : string -> string
```

DESCRIPTION

These string operations roughly correspond to a merging of the BSD `strings.h` and System V `string.h` interfaces.

The first set of operations are defined in terms of sets of characters. These character sets are specified as either a predicate on ordinals, or as a string (where the set consists of those characters in the string).

indexp *pred* **(***s***,** *i***)**

returns the index of the leftmost character in *s*[*i*..] that satisfies *pred*. If such a character does not occur, then the exception `NotFound` is raised.

index *s1* **(***s2***,** *i***)**

returns the index of the leftmost character in *s2*[*i*..] that is in *s1*. If such a character does not occur, then the exception `NotFound` is raised.

revindexp *pred* (*s*, *i*)
: returns the index of the rightmost character in $s[0..i-1]$ that satisfies *pred*. If such a character does not occur, then the exception `NotFound` is raised.

revindex *s1* (*s2*, *i*)
: returns the index of the rightmost character in $s2[0..i-1]$ that is in *s1*. If such a character does not occur, then the exception `NotFound` is raised.

spanp *pred* (*s*, *i*)
: returns the length of the initial segment of *s*[*i*..] satisfying the predicate *pred*.

span *s1* (*s2*, *i*)
: returns the length of the initial segment of *s2*[*i*..] consisting wholly of characters in *s1*.

cspanp *pred*
: is equivalent to the expression `span (not o` *pred*`)`.

cspan *s1* (*s2*, *i*)
: returns the length of the initial segment of *s2* consisting wholly of characters not in *s1*.

tokenizep *pred* (*s*, *i*)
: splits the string *s*[*i*..] into tokens, where *pred* defines the separator characters.

tokenize *s1* (*s2*, *i*)
: splits the string *s2*[*i*..] into tokens, where the characters in *s1* are the separators.

A number of string comparison operations are also provided:

strcmp (*s1*, *s2*)
: lexically compares *s1* and *s2*, and returns their relation.

isPrefix (*s1*, *s2*, *i*)
: returns true, if *s1* is a prefix of *s2*[*i*..].

unequalAt (*s1*, *i1*, *s2*, *i2*)
: compare the strings *s1*[*i1*..] and *s2*[*i2*..], returning the indices of the first characters at which they differ.

prefixCmp (*s1*, *i1*, *s2*, *i2*)
: tests whether *s1*[*i1*..] is a prefix of *s2*[*i2*..], and *vice versa*. The return result of `prefixCmp` is as follows:

`(true, true)`	if *s1*[*i1*..] is equal to *s2*[*i2*..]
`(false, true)`	if *s1*[*i1*..] is a prefix of *s2*[*i2*..]
`(true, false)`	if *s2*[*i2*..] is a prefix of *s1*[*i1*..]
`(false, false)`	otherwise

In addition, there are a few other string operations provided:

suffix (*s*, *i*)
: returns the suffix of *s* starting at the *i*th character. Note: if *i* is equal or greater than the length of *s*, then the empty string is returned.

stringTrans (*s1*, *s2*) *src*
: translates the string *src* according to the map defined by *s1* and *s2*; i.e., each occurrence of a character in *s1* is mapped to the corresponding character in *s2*. If *s1* and *s2* have different lengths, then the exception `LibBase.BadArg` is raised.

stringMap *trans src*
translates the string *src* according to the translation function *trans*.

compressStr *src*
compresses ML-style escape sequences in *src* to single characters.

expandStr *src*
expands any non-printing characters (plus “`"`” and “\”) in *src* to their ML-style escape sequences.

FILES

$SMLLIB/string-util-sig.sml	contains the signature `STRING_UTIL`.
$SMLLIB/string-util.sml	contains the structure `StringUtil`.

SEE ALSO

LibBase(LIB), CharSet(LIB), CType(LIB)

NAME

TimeLimit — limit the amount of time spent evaluating a function application.

SYNOPSIS

structure TimeLimit : sig ... end

SIGNATURE

```
exception TimeOut

val timeLimit : System.Timer.time -> ('a -> 'b) -> 'a -> 'b
```

DESCRIPTION

The expression `(timeLimit` *t f x*`)` evaluates the function *f* applied to *x* using at most time *t*. If evaluation takes longer than *t*, then the exception `TimeOut` is raised.

FILES

$SMLLIB/time-limit.sml	contains the `TimeLimit` structure.

CAVEATS

This function uses the interval timer and the `SIGALRM` signal, and so is not compatible with other applications that use these features (e.g., CML).

NAME

UnixEnv — manage name-value environments

SYNOPSIS

signature UNIX_ENV
structure UnixEnv : UNIX_ENV

SIGNATURE

```
val getFromEnv    : (string * string list) -> string option
val getValue      : {name : string, default : string, env : string list} -> string
val removeFromEnv : (string * string list) -> string list
val addToEnv      : (string * string list) -> string list

val environ       : unit -> string list

val getEnv        : string -> string option
```

DESCRIPTION

The `UnixEnv` structure provides functions for managing a Unix-like environment, which consists of list of strings of the form *name*=*value*.

getFromEnv (*name*, *env*)
: searchs *env* for an item of the form *name*`^"="^`*value*. If found, it returns `SOME` *value*; otherwise, it returns `NONE`.

getValue {*name*, *default*, *env*}
: searchs *env* for an item of the form *name*`^"="^`*value*. If found, it returns *value*; otherwise, it returns *default*.

removeFromEnv (*nameValue*, *env*)
: If *nameValue* has the form *name*, delete the first corresponding name-value pair from *env*.

addToEnv (*nameValue*, *env*)
: If *nameValue* has the form *name*`^"="^`*value*, use this to replace first item in *env* of the form *name*`^"="^`*value'*. If no such item exists, append *nameValue* to *env*.

environ ()
: return the user's environment.

getEnv *name*
: return the binding of *name* in the user's environment.

FILES

$SMLLIB/unix-env-sig.sml	contains `UNIX_ENV` signature
$SMLLIB/unix-env.sml	contains `UnixEnv` structure

SEE ALSO

CAVEATS

Note that environments are implemented concretely, and there is no validation of the data.

NAME

UnixPath — support for UNIX search path lists.

SYNOPSIS

signature UNIX_PATH
structure UnixPath : UNIX_PATH

SIGNATURE

```
type path_list
  sharing type path_list = Pathname.path_list

val getWD           : unit -> string

val getPath         : unit -> path_list

val mkAbsolutePath  : string -> string
val mkRelativePath  : string -> string

datatype access = A_READ | A_WRITE | A_EXEC
  sharing type System.Unsafe.SysIO.access = access
datatype file_type = F_REGULAR | F_DIR | F_SYMLINK | F_SOCK | F_CHR | F_BLK
  sharing type System.Unsafe.SysIO.file_type = file_type

exception NoSuchFile

val findFile        : (path_list * access list) -> string -> string
val findFiles       : (path_list * access list) -> string -> string list

val findFileOfType  : (path_list * file_type * access list) -> string -> string
val findFilesOfType : (path_list * file_type * access list) -> string -> string list
```

DESCRIPTION

getWD ()
return the path to the current working directory (this is a physical path).

getPath ()
return the `path_list` specified by the user's `PATH` variable.

mkAbsolutePath *path*
returns *path* if it is absolute; otherwise, it returns an absolute path that is formed by concatenating the current working directory and *path*.

mkRelativePath *path*
returns *path* if it is absolute; otherwise it returns an equivalent path relative to the current working directory.

The `UnixPath` module also supports additional operations on search path lists (see **Pathname**(LIB)):

findFile (*pathList***,** *mode***)** *path*
searches for a file with the access permissions defined by *mode* (the empty list means just check for existence) as follows. If *path* is absolute, then `findFile` returns *path* if it specifies such a file. If *path* is relative, then each directory in *pathList* is checked by appending *path* to see if it specifies a file with the given access permissions; the full pathname of the first such file found is returned. If no such file is found, then the exception `NoSuchFile` is raised.

findFiles (*pathList***,** *mode***)** *path*

searches for files with the access permissions defined by *mode* (the empty list means just check for existence) as follows. If *path* is absolute, then `findFile` returns the singleton list `[`*path*`]` if it specifies such a file. If *path* is relative, then each directory in *pathList* is checked by appending *path* to see if it specifies a file with the given access permissions; the full pathnames of all such files is is returned. If no such files are found, then the empty list is returned.

findFileOfType (*pathList***,** *ftype***,** *mode***)** *path*

searches for a file of type *ftype* and with the access permissions defined by *mode* (the empty list means just check for existence) as follows. If *path* is absolute, then `findFile` returns *path* if it specifies such a file. If *path* is relative, then each directory in *pathList* is checked by appending *path* to see if it specifies a file of the given type and access permissions; the full pathname of the first such file found is returned. If no such file is found, then the exception `NoSuchFile` is raised.

findFilesOfType (*pathList***,** *ftype***,** *mode***)** *path*

searches for files of type *ftype* and with the access permissions defined by *mode* (the empty list means just check for existence) as follows. If *path* is absolute, then `findFile` returns the singleton list `[`*path*`]` if it specifies such a file. If *path* is relative, then each directory in *pathList* is checked by appending *path* to see if it specifies a file of the given type and access permissions; the full pathnames of all such files is is returned. If no such files are found, then the empty list is returned.

FILES

$SMLLIB/unix-path-sig.sml	contains the `UNIX_PATH` signature.
$SMLLIB/unix-path.sml	contains the `UnixPath` structure.

SEE ALSO

PathName(LIB), UnixEnv(LIB)

Index