LFCS
Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

The Edinburgh SML Library

by

Dave Berry

The Edinburgh SML Library

LFCS Report Series
ECS-LFCS-91-148
April 1991

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

The Edinburgh SML Library

Dave Berry

April 11, 1991

Abstract

The Edinburgh Standard ML Library provides more than 200 functions on a range of types. It is written in portable Standard ML. Optimised versions exist for Standard ML of New Jersey, Poly/ML and Poplog ML. The library provides a consistent framework into which new entries can be incorporated easily.

This report describes the basic framework of the library, and gives detailed documentation of all the current entries. It also gives some advice on how to write new entries.

Contents

Chapter 1

Introduction.

The Edinburgh SML Library currently provides over 200 functions on several basic types, including sets, hash tables and the built-in types. By using the library you can avoid having to write your own versions of these functions. What's more, you will be sure that they will run on any implementation of SML, sparing you or your users the need to port your program between different compilers. Using the library will also make your programs easier for other library users to understand.

The library provides a consistent framework that can be extended by the addition of new modules. Users of the library are invited to add their own code to the library, provided that they follow the guidelines described in this report. The framework ensures that the library entries present a consistent user interface. It also makes practical the development of more sophisticated tools, such as a hypertext browser.

Part of this framework is software. The `Make` system recompiles the part of your program that have changed, and no more. The coding styles used in the library are supported by a small number of types and functions, which the library makes available at top-level. Generic signatures let you write functors that can be applied to most library entries. Generic signatures also serve to structure the library.

The rest of the framework consists of conventions that library entries should follow. These are just as important as the software. They include conventions that ensure a consistent user interface, such as consistent use of upper-case and lower-case letters in identifiers, the preference of curried functions over tuple parameters, and several standard names. The conventions also include a standard format for comments. This makes each signature broadly equivalent to a UNIX on-line manual page.

The library also provides a standard interface for common extensions to the language, such as vectors and arrays. A portable version of the library defines the objects that it provides, and can be used with any implementation of the language. Implementers are encouraged to provide more efficient versions of library entries.

This notion of portability extends to features that can't be defined in pure SML, such as the `use` and `cd` functions provided by most compilers. Although the portable version of the library can't define these functions, it can specify their types; compiler-specific versions of the relevant library entries can implement them.

The Edinburgh SML Library has been developed over a number of years with input from several people. The current version will be distributed with most SML compilers. The library is free, and people are welcome to develop it as they see fit.

Chapter 2

Installing the Library.

2.1 Distribution.

The library is distributed with the following sub-directories:

`signatures`. The signatures that the user can see.

`portable`. The portable implementations of the functors and structures that make up the library.

`doc`. Copies of this documentation, in LaTeX format.

`nj-sml.dist`. System-specific implementations of some entries for Standard ML of New Jersey.

`poly.dist`. System-specific implementations of some entries for Poly/ML.

`poplog.dist`. System-specific implementations of some entries for Poplog ML.

2.2 Installation.

The `INSTALL` program does most of the work of installation. To use it, change directory to the directory containing the library, and call `INSTALL` with a command line like the following:

```
INSTALL -poly -nj-sml -poplog target
```

All the arguments to `INSTALL` are optional.

`INSTALL` copies the library to the `target` directory. If no target directory is given, `INSTALL` treats the current directory as the target.

If the `-nj-sml` flag is given, `INSTALL` creates a new sub-directory called `nj-sml`, and copies the compiler-specific implementations to that directory. Then it adds links to the portable implementations of the remaining library entries, so that the new sub-directory contains a complete version of the library. The `-poly` and `-poplog` flags have the same effect for the other compilers.

Finally, `INSTALL` creates the SML build files. You can then load the library into an SML session. There are three build files that you can choose from; normally you will want to use `build_make.sml` or `build_all.sml`. In addition, it's usually a good idea to save a core image or a persistent database containing the loaded library. How to do this depends on which compiler(s) you are using, and is described below.

2.3 Building The Library.

The basic method of building the library is simple, and varies only slightly with the choice of compiler. First, change directory to the subdirectory for the version that you want to build – either the portable version or the version specific to the compiler that you're using. Then run SML and `use` the file `poly.load`, `nj-sml.load` or `poplog.load`, as appropriate. This sets up a consistent set of load functions that is used by the rest of the library. Then `use` the build file that you want.

In New Jersey ML, the `use` function is available at top-level. In Poplog ML, it is in the `Compile` structure. In Poly/ML, it is in the `PolyML` structure.

If you are using Poly/ML and you want to store the compiled library in a child database, you have to create the database in one Poly/ML session, (using the function `PolyML.make_database`), and then run Poly/ML again with this database to actually build the library. When you leave Poly/ML, the database will contain everything that you evaluate in this session. It's usually a good idea to make the database read-only once you've built the library.

If you are using Standard ML of New Jersey, you can save the library in a core image using the command `exportML`. The saved image can be executed directly.

If you are using Poplog ML, you can save the library in a core image using either `PML.System.make` or `PML.System.save`. The core image can be run by giving the file name as an argument to the `pml` UNIX command, preceded by a `+` character. For example, on our systems, the file `/usr/local/bin/pml-library` contains the line:

`pml +/usr/local/lib/poplog/library.psv`.

Once you have created a saved-state that contains the library, you should add a user command to your system to start the version of SML that contains the library. You may wish to make this version the default.

2.4 Choice of Build File.

There are three ways that you can load the library, depending on which build file you use. You can load all the files at once with the file `build_all.sml`; load just the make system with the file `build_make.sml`, and let `Make` do the rest of the work when a new entry is needed; or use the file `build_core.sml`, which just loads a few functions that can find the library entries in the installation directory.

If you load all the entries, then users will be able to use them immediately. Also, you can move the source to another directory, since the system will never have to load it directly. However, the saved state will be fairly large (the library is around 4 megabytes).

If you load just the make system, then the saved state will be quite small. Users will load just the entries that they need. The make system will load all entries needed by those that the user wants. However, the user will have to wait for a short period while the entries are compiled, and the source must remain in the installation directory.

The third method uses virtually no space in the saved state. However, in addition to the disadvantages of the second option, it also requires the user to ensure that all entries are loaded in the correct order. This is not a good choice for general use. I would only recommend it for the final build of a stand-alone package, in which case the user should access the library directly instead of using a more generally useful version.

Chapter 3

Using the Library.

3.1 Getting Started.

To use the library, you need a version of SML with the library loaded. The person who installed the library on your system should have created such a version for you. If not, you'll have to read Chapter 2 and do it yourself.

Suppose that you want a function to convert integers to strings, and a function to prompt the user for input. On reading the rest of this chapter, you will find that you want the `Int` and `User` library entries, specifically the functions `Int.string` and `User.prompt`.

There are two ways that the library could be set up. Type `loadLibrary "User"`. If this returns the string `"All library entries have been loaded"`, then the entries are ready for use. If this is not the case, the your call to `loadLibrary` will compile the `User` entry for you. The `Make` system will ensure that all entries required by the `User` entry itself will be loaded as well.

Once you've loaded the `User` and `Int` entries, you can use the `Int.string` and `User.prompt` functions just like any other components of structures. Most library entries are implemented as structures instead of functors. This saves space and makes the library easier to use. [1]

3.2 Entries.

Most library entries are structures. Each structure is documented by a signature and a sample (possibly inefficient) implementation. Structure names are written in mixed case, with the corresponding signature names written in upper case. For example, the `Int` structure implements the `INT` signature.

The signature gives the usual type information, with a short description of each object (in English), and some introductory material. The introductory material includes details of the author, maintenance arrangements and so forth. Signatures may include Extended ML axioms (in comments), if the implementer provides them.

The implementation is a reference implementation of the objects declared in the signature. The actual implementation may be more complex than this, but it must

[1] The library can also be loaded without loading the `Make` system or any entries at all. This is intended for minimising space when building stand-alone applications rather than for general use.

produce the same results and side-effects. The only exception to this rule is that implementations may produce debugging or tracing output in addition to their normal effects.

Some entries are functors. Since SML doesn't have a notion of "functor signature", these entries are described by comments at the start of each functor.

The detailed descriptions of the current entries are given later in this report. This section gives an overview of the whole library, to give you some idea of what the library contains.

In addition to the entries described in this report, the library also includes space for contributions that don't fit the library format. To see these, find out where the library is installed on your system and browse the `contrib` sub-directory.

3.3 Structures.

The library currently contains the following structures:

`General`. Types, exceptions and functions that are widely used or that don't fit anywhere else. In particular, the types `Result`, `Option` and `Nat` are made global and are used in many other library entries.

`Bool, Instream, Int, List, Outstream, Real, Ref, String`. Basic operations on the pervasive types.

`BoolParse, IntParse, ListParse, StringParse`. Functions to parse the basic types from strings or instreams. The `ListParse` structure matches the `SEQ_PARSE` signature; the others match the `PARSE` signature.

`Ascii, StringType, StringListOps`. `Ascii` defines constants for the non-printing Ascii characters. `StringType` defines functions for testing whether the first character in a string is a letter, digit, control character, etc. `StringListOps` defines functions on strings that mimic those in the `List` entry.

`ListSort`. Functions to sort and permute lists.

`AsciiOrdString, LexOrdString, LexOrdList`. Orderings on sequences. `AsciiOrdString` compares strings when case difference is significant; by contrast `LexOrdString` ignores case.

`Array, ArrayParse, Byte, ByteParse, Vector, VectorParse`. Types and functions for arrays, bytes, and constant vectors.

`Pair, PairParse, ListPair, StreamPair`. Operations on pairs of objects. `StreamPair` also includes suggested functions for interacting with the host file system using pairs of one instream and one outstream.

`EqSet, Set`. Polymorphic sets. `EqSet` defines sets over equality types. `Set` defines sets over arbitrary types, but defines several functions that require and equality function to be passed as an argument.

`Hash`, `EqFinMap`. Hash tables and finite maps. The keys to finite maps must be equality types.

`User`. Functions to prompt the user for input. A window system should provide an appropriate replacement for this structure.

`Make`. The `Make` system mentioned in the `Getting Started` section. This will be described in more detail in Section 3.7.

`Const`. Creates unique copies of objects.

`System`. Some suggested functions for interacting with the host file system and the SML compiler. Many of these functions are not implemented in the portable version of the library.

`Combinator`. Simple combinator functions.

`Memo`. Memoising functions.

3.4 Signatures, Functors, and Monomorphic Types.

Most of the structures mentioned in the previous section have corresponding signatures with the same name (but in upper case). The exceptions are the Parse structures, which are match the `generic` signatures `PARSE` and `SEQ_PARSE`. (Exception: the `ArrayParse` structure matches the `ARRAY_PARSE` signature).

There are several other generic signatures. One example is the `EQUALITY` signature. This specifies a type `T` and an equality function `eq: T -> T -> bool`. By convention, when a library entry defines a new type, it gives it the name `T` in addition to its normal name. For example, the types `Int.int` and `Int.T` are identical (and are also identical to the built-in type `int`). This means that many of the structures in the library will match the `EQUALITY` signature.

Now suppose that you define a functor that takes an argument structure specified by the `EQUALITY` signature. For example, the functor `MonoSet` takes such an argument and returns a set of elements of type `T`. The library naming conventions mean that this functor can be applied to many of the types defined in the library.

The library provides several such generic signatures. `ORDERING` defines a type `T` and an ordering function. `PRINT` defines a type `T`, a function to convert values of that type to strings, and a function to print the string representation of a value on an outstream. `ORD_PRINT`, `EQ_ORD`, and `EQ_PRINT` are obvious combinations of these, while `OBJECT` combines all of them in a single signature. `EQTYPE_ORD` and `EQTYPE_PRINT` are versions of `ORDERING` and `PRINT` in which `T` is an equality type.

The functors `MonoSet`, `MonoList`, `MonoVector` and `MonoArray` use generic signatures. `MonoSet` uses `EQUALITY`, and the others use `EQ_PRINT`. They all return structures that are similar to their polymorphic equivalents, except that the objects they define contain elements of a fixed type. This is sometimes desirable.

One reason for creating monomorphic values in this way is that they can sometimes be implemented more efficiently then polymorphic ones. For example, a vector of booleans can be implemented as a bitset.

Another reason is that many functions over monomorphic collections have a simpler type, as they don't need a function parameter to apply to the elements of the vector. For example, the equality functions on polymorphic vectors and monomorphic vectors have the types `('a → 'a → bool) → 'a Vector → 'a Vector → bool` and `MonoVector → MonoVector → bool` respectively.

The library also contains the `ByteVector`, `BoolVector`, `ByteArray` and `BoolArray` structures. These can be created by applying the respective functors, or implemented directly. The library also contains the generic signatures `SEQUENCE` and `SEQ_ORD`, which are matched by these structures.

MonoVectors and MonoArrays can be extended to include parsing functions using the `MonoVectorParse` and `MonoArrayParse` functors. These take a MonoVector or MonoArray and a structure that matches the `PARSE` signature, and produce structures that match the generic `MONO_SEQ_PARSE` signature. The type `T` of the structure that matches the `PARSE` signature must be the type of the elements of the MonoVector or MonoArray. This is enforced by a sharing specification.

3.5 Library Conventions.

The previous section introduced the convention that types in library entries can be referred to by the name `T`, and showed how this is used to implement generic signatures. This section describes some more conventions that make library entries consistent and therefore easier to use.

Identifiers. We have already seen that structure and functor names are mixed case, while signatures names are all upper case. All words in alphanumeric identifiers are capitalised (as we have already seen for structure names), except for two cases:

1. Alphanumeric labels and value identifiers (not including constructors and exceptions) begin with a lower case letter. This is so that values and constructors can be distinguished easily.
2. As we have seen, signature identifiers are written in upper case, with multiple words separated by underscores. This is a widely used convention, and is useful when writing about structures and signatures in documentation or tutorial material.

Apart from signature identifiers, alphanumeric identifiers never include underscores. Symbolic identifiers are only used for functions or constructors, and then rarely.

Infix functions. Functions that are intended to be used infix are declared as such, with a suggested precedence, even though this status doesn't propagate when a structure is opened.

Currying. Nonfix functions are usually curried. Entries don't usually provide both curried and uncurried or infix versions of a function.

Imperative functions. Imperative functions usually return the unit value.

Exceptions. In the current entries, exceptions are only used to signal exceptional circumstances, such as a function called with unsuitable arguments. If a function sometimes returns a value and sometimes doesn't, then usually its result type will be an instance of the `Result` type defined in the `General` structure (and available globally). Using these types forces you to use a case expression instead of a handler to check the result, which means that the compiler will report missing cases.

However, this convention is not a hard and fast rule. For example, if someone wants to implement a version of the parsing functions that raises an exception on incorrect input, they may add this version to the library.

Exceptions usually return some information about the values that caused the exception, and the name of the function in which it occurred (if more than one function can raise that exception).

Equality. If an entry is defined over equality types, the name of that entry usually begins with `Eq`. Sometimes there will be two versions of an entry, one restricted to equality types, the other taking an explicit equality function. The `Set` and `EqSet` entries are an example of this.

If an entry uses an explicit equality function, then this function is passed to the functions that manipulate values of the type. It is not passed to functions that `create` values of the type. The latter approach can result in erroneous programs, where two values have the same type but different associated equality functions.

Monomorphic types. Often the library includes both monomorphic and polymorphic versions of an entry. In such cases the monomorphic version is implemented as a functor. The result signature of the functor is the same as that of the polymorphic structure, with the monomorphic type substituted for occurrences of the polymorphic type, and the elements type(s) substituted for the corresponding type variables.

Sharing. Most objects defined in a library entry are unique to that entry. A few shared types are defined in the `General` entry.

3.6 Standard Names.

Many library entries define a type and operations on that type. These entries use several names in common, so that frequently-used operations are easy to remember. (this was inspired by the SmallTalk and Eiffel libraries.) It also means that the entries can be used with the generic signatures, as we saw in Section 3.4.

`eq, ne, lt, le, gt, ge`. The comparison functions.

`fixedWidth`. True if the string representation of the type takes a known amount of space (e.g. for Bytes).

`string`. Converts an object to its string representation.

`print`. Prints the string representation of a value on an outstream.

`parse`. Converts the string representation of an object to that object.

`read`. Like parse, but takes an InStream.

`create`. Create an object from components.

`tabulate`. Generate a sequence by applying a function to the indices of the elements.

`size`. Return the number of elements in a sequence.

`empty`. Return true if a sequence is empty.

`sub, nth`. Return the nth element of a sequence.

`extract`. Return a subsequence of a sequence.

`map, apply, iterate, iterateApply`. Apply a function to each element of a sequence (four variations of the basic idea).

`foldL, foldR, foldL', foldR'`. Combine the elements of a sequence to produce a single value (four variations of the basic idea).

`pairwise`. Test that all adjacent pairs of elements in a sequence satisfy a binary predicate.

`rev`. Reverse a sequence.

`^`. Catenate two sequences.

In addition, there is a convention for naming conversion functions – functions that convert values of one type to values of another type. If the function converts values of type `X` to values of type `Y`, and is defined in the entry `X` that defines the type `X`, then the function is called `X.y`. Conversely, if the function is defined in the entry `Y` that defines type `Y`, then it is called `Y.fromX`. For example, the function to convert a vector to a list is called `Vector.list`, and the function to do the opposite conversion is called `Vector.fromList`.

The rationale behind this convention is that functions are usually called by their qualified name, and the name of the structure is the same as one of the types in the conversion. Therefore it's redundant to have both type names in the name of the function itself.

3.7 Using the Make System.

The `Make` system is used by the library to ensure that, when you load a new entry, all entries that the new entry requires are loaded first In addition, you can use `Make` to minimise the amount of recompilation you do when you change your program. `Make` works by keeping track of which pieces of your code depend on which others. You must give `Make` this information by annotating your code with special comments. Then, when you tell `Make` to compile a piece of code, it automatically loads any code that the new code requires that isn't already loaded.

The dependency information is given by preceding each piece of code by a tag declaration in a special comment. Keeping the dependency information with each piece of code, instead of in a separate file like the UNIX `make` command, makes it easier to keep the two in sync. Tag declaration comments are a subset of ordinary comments, so files containing them can be compiled without using make.

Tag declaration comments begin with the string `(*$` instead of the usual `(*`, and must start at the beginning of a line. Formatting characters are forbidden before and after the initial `(*` string, but are permitted between tags and dependencies. For example, the following tag declaration associates the tag `Foo` with the code that follows it:

```
(*$Foo *)
```

The following tag declaration associates the tag `Bar` with the code that follows it, and states that this code depends on the code associated with the tags `Foo1` and `Foo2`:

```
(*$Bar:  Foo1 Foo2 *)
```

The code associated with a tag is terminated by the next tag declaration or the end of a file.

The easiest way to consult the dependency information is to list the files containing the code in a file (called a tag file), and to call `loadFrom` with the name of that tag file. This loads all the dependency information into the make system. If you change the dependency information, use `consultDecl` or `consultFile` to update the stored dependencies.

Once the dependency information has been loaded, call `make tag` to compile the code associated with the tag and all the code that it depends on. The system will write all the relevant code into a temporary file (called `%Make.tmp%` by default) and call `use` on that file.

Once a piece of code has been read from a file, it is stored in a cache When the code associated with a tag is changed, call `touch tag` to tell the `Make` system that the current entry in the cache is obsolete. `consultDecl` and `consultFile` do this automatically for the tags that they read. If a piece of code fails to compile, then that entry in the cache is automatically touched as well. The system does not check the modification dates of files.

Once all the changed pieces of code have been touched, call `again ()` to re-make your program.

3.8 Infix Operators.

This is the list of infix operators provided by the pervasives and by the library:

```
infix 0 General.before
infix 1 Bool.or
infix 2 Bool.&
infix 3 o Bool.implies
     Combinator.oo Combinator.co Combinator.fby Combinator.over
infix 4 = <> < > <= >=
infix 5 Int.--
infixr 5 ::  @
infix 6 + - ^
```

`infix 7 / * div mod Int.divMod Int.quot Int.rem Int.quotRem`
`infix 8 Int.** Real.**`
`infix 9 X.sub`, for most `X`

Note that @ associates to the right in the library. Remember that infix status is local to each structure.

Equivalent curried functions for some of the above are as follows:

`=`	`X.eq`, for most `X`
`<>`	`X.eq`, for most `X`
`<`	`Int.lt, Real.lt`
`>`	`Int.gt, Real.gt`
`<=`	`Int.le, Real.le`
`>=`	`Int.ge, Real.ge`
`@`	`List.appendLast`
`(,)`	`Pair.create`

3.9 Future Entries.

The main aim of the library is to provide a framework for future development. The only way that the library can grow is by users contributing code. If you would like to add an entry to the library, please read Chapter 5. The following list suggests some entries that would be useful:

- Trees, of various kinds, and implementations of maps etc. based on trees.
- Regular expressions and substitutions over strings.
- Lazy lists.
- Complex numbers, rational numbers.
- Stacks, priority queues, etc.
- Tries.
- Scanner generators, parser generators.
- A portable, easy-to-use interface to host window systems.

Chapter 4

The Entries.

This chapter consists of the on-line documentation for the current entries in the Edinburgh SML library. As explained in Chapter 3, most entries are documented by an annotated signature. The others are described by generic signatures.

Each entry begins with a header section. This is a comment that includes a title, author, creation date, maintenance details, a description of the entry, and optional notes or references to related entries. In the actual files the header section also contains an RCS log.

Like UNIX manual pages, the header sections have a standard syntax. This is specified in Chapter 5.

The other objects defined in the entry are grouped under convenient sub-headings. These depend on the entry. The following list gives some sub-headings that I've used extensively; other authors may decide not to use these.

`PERVASIVES`. Most of the SML pervasives are provided in library entries, so that they can be accessed even if their identifiers are rebound at top level. Pervasive constructors, overloaded identifiers, and the equality function are not rebound in this way, because the language won't allow it.

`SYSTEM`. Some functions can't be defined in terms of the SML pervasives. These are usually grouped under the `SYSTEM` sub-heading.

`TYPES`. Any other type or types defined by an entry, including synonyms of the main type.

`CONSTANTS`. Constant values.

`CREATORS`. Functions that create composite values from elements.

`CONVERTORS`. Functions that convert values from one type to another.

`OBSERVERS`. Functions that return information about a value, such as membership of a class, number of elements, etc.

`SELECTORS`. Functions that return sub-components of a value.

`ITERATORS`. Functions that apply a parameter function to every element of a sequence.

`MANIPULATORS`. General functions.

Each object in an entry is documented by a comment that follows the definition. This comment may begin with an example call; if present, this should be separated from the main text by a semi-colon. For simple values, the comment may include the definition. I've only done this for one-liners.

4.1 Array

```
(*$ARRAY: GeneralTypes *)

signature ARRAY =
sig

(* ONE DIMENSIONAL ARRAYS

Created by: Dave Berry, LFCS, University of Edinburgh
Date: 30 Oct 1989

Maintenance: Author


DESCRIPTION

   An array is a single-dimensional polymorphic array with elements that
   be updated in place.  Different arrays are not equal, even if they hold
   the same elements.  Arrays always admit equality.  There is only one
   empty array.  The first element of any array has the index 0.

   Elements can be updated by the "update" function; the conventional
   "a sub i := v" notation doesn't work because the "a sub i" would have to
   return a reference, and this would either cause problems for the garbage
   collector or add an extra level of indirection.  If this extra level of
   indirection is acceptable, you can create constant vectors of references
   using the Vector type.

   To create an array of arrays, either use a tabulate function
   or first create an array of empty arrays and then update each
   element with the desired initial value.

   Unlike lists, this signature doesn't provide the  subLast  and  extractLast
   functions.  The "size" function should take constant time, so this is not a
   handicap.


SEE ALSO

   ARRAY_PARSE, VECTOR, MONO_ARRAY.


NOTES

   A possible implementation would be to view a random access file as an
   array.
```

```
*)


(* TYPES *)

  eqtype 'a Array

  eqtype 'a T
    sharing type T = Array


(* CREATORS *)

  exception Size of string * int
   (* Size (fn, n); raised by the creation functions when they are called
      with a negative size argument. *)

  val create: Nat -> '_a -> '_a Array
   (* create n i; create a Array of n locations, each containing i.
      Raise (Size ("create", n)) if n < 0. *)

  val tabulate: Nat * (int -> '_a) -> '_a Array
   (* tabulate (n, f); create an array v of n locations, (v sub 0) to
      (v sub (n-1)), with (v sub i) initialised to (f i).
      Raise (Size ("tabulate", n)) if n < 0. *)

  val tabulate': Nat * ('b -> '_a * 'b) * 'b -> '_a Array
   (* tabulate' (n, f, base); create an array of n locations, (v sub 0) to
      (v sub (n-1)) with (v sub 0) initialised to (# 1 (f base)) and
      (v sub i (i > 0)) initialised to (# 1 (f (# 2 f_i))), where f_i is
      the result of the i^th application of f.
      Raise (Size ("tabulate'", n)) if n < 0. *)


(* CONVERTERS *)

  val list: 'a Array -> 'a list
   (* list v; make a list containing (only) the elements of v, in
      the same order. *)

  val fromList: '_a list -> '_a Array
   (* fromList v; make an array containing (only) the elements of v, in
      the same order. *)
```

```
  val stringSep: string -> string -> string ->
                 ('a -> string) -> 'a Array -> string
   (* stringSep start finish sep p v; returns the string representation of v,
      beginning with start, ending with finish, and with the elements
      separated by sep. *)

  val string: ('a -> string) -> 'a Array -> string
   (* string p v; returns the canonical string representation of v. *)

  val printSep: outstream -> string -> string -> string ->
                (outstream -> 'a -> unit) -> 'a Array -> unit
   (* printSep os start finish sep p v; sends the string representation of v
      to the stream os, beginning with start, ending with finish, and with
      the elements separated by sep. *)

  val print: outstream -> (outstream -> 'a -> unit) -> 'a Array -> unit
   (* print os p v; sends the canonical string representation of v to
      the stream os. *)

(* OBSERVERS *)

  val isEmpty: 'a Array -> bool
   (* isEmpty v; returns true if v is empty. *)

  val size: 'a Array -> Nat
   (* size v; return the number of elements in v. *)

  val eq: ''a Array -> ''a Array -> bool
   (* eq x y; returns true if (x = y); returns false otherwise. *)

  val ne: ''a Array -> ''a Array -> bool
   (* ne x y; returns true if (x <> y); returns false otherwise. *)


(* SELECTORS *)

  exception Subscript of string * int
  (* infix 9 sub *)
  val sub: 'a Array * int -> 'a
   (* v sub n; return the n+1'th element of v.
      Raise (Subscript ("sub", n)) if not (0 <= n <= size v). *)

  val nth: int -> 'a Array -> 'a
   (* nth n v; return the n+1'th element of v.
      Raise (Subscript ("nth", n)) if not (0 <= n <= size v). *)
```

```
exception Extract of int * int
val extract: int -> int -> '_a Array -> '_a Array
 (* extract start finish v; returns the sub-array of v starting with
    (v sub start) and ending with (v sub (finish - 1)).
    Returns the empty array if (start = finish).
    Raise (Extract (start, finish)) if not (0 <= start,finish <= size v). *)


(* MANIPULATORS *)

 val rev: '_a Array -> '_a Array
  (* rev v; builds a new Array containing the elements of v in
     reverse order. *)

 (* infix 6 ^ *)
 val ^ : '_a Array * '_a Array -> '_a Array
  (* v ^ v'; builds an array containing the elements of v' appended to those
     of v. *)

 exception Update of int
 val update: 'a Array * int * 'a -> unit
  (* update (v, n, i); replace (v sub n) with i.
     Raise (Update n) if not (0 <= n <= size v). *)

 exception Copy of int * int * int
 val copy: '_a Array -> int -> int -> '_a Array -> int -> unit
  (* copy v start finish v' start'; copies the sub-array of v starting with
     (v sub start) and ending with (v sub (finish - 1)) to the Array v',
     starting with (v' sub start'). Has no effect if (start = finish).
     Raises (Copy (start, finish, start')) if
     not (0 <= start,finish <= size v) or if
     not (0 <= start',start'+finish-start <= size v'). *)

 exception UpdateRange of int * int
 val updateRange: 'a Array -> int -> int -> 'a -> unit
  (* updateRange v start finish i; update the elements of v starting with
     (v sub start) and ending with (v sub (finish - 1)) with i. Has no effect
     if (start = finish). Raises (UpdateRange (start, finish)) if
     not (0 <= start,finish <= size v). *)


(* REDUCERS *)

 val foldR: ('a -> 'b -> 'b) -> 'b -> 'a Array -> 'b
  (* foldR f base v; folds using f associating to the right over the
     base element.
     foldR f [a1,a2,...,an] base = f(a1,f(a2,...,f(an,base)...)). *)
```

```
  val foldL: ('a -> 'b -> 'b) -> 'b -> 'a Array -> 'b
   (* foldL f v base; folds using f associating to the left over the
      base element.
      foldL f [a1,a2,...,an] base = f(an,...,f(a2,f(a1,base))...). *)

  exception Empty of string
  val foldR': ('a -> 'a -> 'a) -> 'a Array -> 'a
   (* foldR' f v; folds using f associating to the right over the
      last element of v.  Raises (Empty "foldR'") if v is empty. *)

  val foldL': ('a -> 'a -> 'a) -> 'a Array -> 'a
   (* foldL' f v; folds using f associating to the left over the
      first element of v.  Raises (Empty "foldL'") if v is empty. *)

  val pairwise: ('a -> 'a -> bool) -> 'a Array -> bool
   (* pairwise f v; true if (f (v sub i) (v sub (i + 1))) is true for all
      0 <= i < size v, or if v is empty. *)


(* ITERATORS *)

  val map: ('a -> '_b) -> 'a Array -> '_b Array
   (* map f v; builds a new Array by applying f to each element of v. *)

  val apply: ('a -> unit) -> 'a Array -> unit
   (* apply f v; applies f to each element of v. *)

  val iterate: ('a * int -> '_b) -> 'a Array -> '_b Array
   (* iterate f v; builds a new Array by applying f to each element of v
      paired with its index. *)

  val iterateApply: ('a * int -> unit) -> 'a Array -> unit
   (* iterateApply f v; applies f to each element of v paired with its index. *
end
```

4.2 ArrayParse

```
(*$ARRAY_PARSE: GeneralTypes InstreamType *)

signature ARRAY_PARSE =
sig

(* FUNCTIONS TO READ AND PARSE ARRAYS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 7 Feb 1991

Maintenance: Author


DESCRIPTION

   Functions to read and parse arrays.  These functions differ from those
   in SEQ_PARSE in that they have imperative type variables.  See SEQ_PARSE
   for descriptions of each function.


SEE ALSO

   SEQ_PARSE, ARRAY.


NOTES

   These functions were originally in the main ARRAY signature.




*)


(* TYPES *)

  eqtype 'a T
    sharing type T = Array.Array


(* CONVERTERS *)
```

```
  exception Sep of string * string * string * string

  exception Size of string * int

  val parseSepN: string -> string -> string ->
           (string -> ('_a * string, 'b) Result) -> Nat -> string ->
    ('_a Array.Array * string, '_a Array.Array Option * string) Result

  val parseSep: string -> string -> string ->
           (string -> ('_a * string, 'b) Result) -> string ->
    ('_a Array.Array * string, '_a Array.Array Option * string) Result

  val parseN: (string -> ('_a * string, 'b) Result) -> Nat -> string ->
    ('_a Array.Array * string, '_a Array.Array Option * string) Result

  val parse: (string -> ('_a * string, 'b) Result) -> string ->
    ('_a Array.Array * string, '_a Array.Array Option * string) Result

  val readSep: string -> string -> string ->
           (instream -> ('_a, 'b) Result) -> instream ->
           ('_a Array.Array, '_a Array.Array Option) Result

  val readSepN: string -> string -> string ->
           (instream -> ('_a, 'b) Result) -> Nat -> instream ->
           ('_a Array.Array, '_a Array.Array Option) Result

  val read: (instream -> ('_a, 'b) Result) -> instream ->
    ('_a Array.Array, '_a Array.Array Option) Result

  val readN: (instream -> ('_a, 'b) Result) -> Nat -> instream ->
           ('_a Array.Array, '_a Array.Array Option) Result

  val fromFile: (instream -> ('_a, 'b) Result) -> string -> '_a Array.Array
   (* fromFile p name; read the contents of the file called name into a vector
      Stops reading from the file as soon as p returns Fail.
      Raises General.Io if something goes wrong. *)

  val file: ('a -> string) -> 'a Array.Array -> string -> unit
   (* file p v name; write the contents of v to the new file called name.
      Raises General.Io if something goes wrong. *)

end
```

4.3 Ascii

```
(*$ASCII *)

signature ASCII =
sig

(* ASCII CONTROL CODES

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date:         4 Feb 1991

Maintenance: Author


DESCRIPTION

   Bindings of the ASCII control characters to their standard names.


NOTES

   These definitions used to be in the STRING signature.



*)


(* CONSTANTS *)


  val nul: string
   (* nul = "\000" *)
  val soh: string
   (* soh = "\001" *)
  val stx: string
   (* stx = "\002" *)
  val etx: string
   (* etx = "\003" *)
  val eot: string
   (* eot = "\004" *)
  val enq: string
   (* enq = "\005" *)
  val ack: string
```

```
  (* ack = "\006" *)
val bel: string
  (* bel = "\007" *)
val bs: string
  (* bs = "\008" *)
val ht: string
  (* ht = "\009" *)
val nl: string
  (* nl = "\010" *)
val vt: string
  (* vt = "\011" *)
val np: string
  (* np = "\012" *)
val cr: string
  (* cr = "\013" *)
val so: string
  (* so = "\014" *)
val si: string
  (* si = "\015" *)
val dle: string
  (* dle = "\016" *)
val dc1: string
  (* dc1 = "\017" *)
val dc2: string
  (* dc2 = "\018" *)
val dc3: string
  (* dc3 = "\019" *)
val dc4: string
  (* dc4 = "\020" *)
val nak: string
  (* nak = "\021" *)
val syn: string
  (* syn = "\022" *)
val etb: string
  (* etb = "\023" *)
val can: string
  (* can = "\024" *)
val em: string
  (* em = "\025" *)
val sub: string
  (* sub = "\026" *)
val esc: string
  (* esc = "\027" *)
val fs: string
  (* fs = "\028" *)
val gs: string
  (* gs = "\029" *)
```

```
  val rs: string
   (* rs = "\030" *)
  val us: string
   (* us = "\031" *)
  val del: string
   (* del = "\127" *)

end
```

4.4 Bool

```
(*$BOOL *)

signature BOOL =
sig

(* BOOLEANS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 22 Sep 1989

Maintenance: Author

DESCRIPTION

   Standard functions on the built-in type "bool".


NOTES

   Possibly there should be read and write functions for binary form
   as well as ascii.  Portability issues would have to be addresses if
   this were so.



*)


 (* PERVASIVES *)

   eqtype bool

   val not: bool -> bool


 (* TYPES *)

   eqtype T
     sharing type T = bool


 (* CONVERTERS *)

   val string: bool -> string
```

```
  val print: outstream -> bool -> unit


(* OBSERVERS *)

  val eq: bool -> bool -> bool

  val ne: bool -> bool -> bool

  val fixedWidth: bool
   (* fixedWidth = false. *)


(* MANIPULATORS *)

  (* infix 1 or *)
  val or: bool * bool -> bool
   (* or (x, y); the standard logic function. *)

  (* infix 2 & *)
  val & : bool * bool -> bool
   (* x & y; the standard logic function. *)

  (* infix 3 implies *)
  val implies : bool * bool -> bool
   (* x implies y; the standard logic function. *)

end
```

4.5 BoolArray

See MonoArray.

4.6 BoolParse

See PARSE.

4.7 Byte

```
(*$BYTE *)

signature BYTE =
sig

(* BYTES

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date:       22 Sep 1989

Maintenance: Author

DESCRIPTION

   Functions on 8-bit bytes.


NOTES

   This exists mainly to support ByteVectors and ByteArrays.



*)


(* TYPES *)

  eqtype Byte

  eqtype T
    sharing type T = Byte


(* CONVERTERS *)

  val string: Byte -> string

  val print: outstream -> Byte -> unit


(* OBSERVERS *)
```

```
  val eq: Byte -> Byte -> bool

  val ne: Byte -> Byte -> bool

  val lt: Byte -> Byte -> bool

  val le: Byte -> Byte -> bool

  val gt: Byte -> Byte -> bool

  val ge: Byte -> Byte -> bool

  val fixedWidth: bool
   (* fixedWidth = true *)

end
```

4.8 ByteArray

See MonoArray.

4.9 ByteParse

See PARSE.

4.10 Combinator

```
(*$COMBINATOR *)

signature COMBINATOR =
sig

(* COMBINATOR FUNCTIONS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 15 Nov 1989

   Copying heavily from similar signatures by Gloria Quintinilla, Rod Burstall
   and Mick Francis.

Maintenance: Author


DESCRIPTION

   Combinators.


NOTES

   The single letter combinators have upper case names, by convention.

   I'm not sure that  can,  fby  and  over  really belong here.

   C', B' and S' are taken from Field and Harrison, "Functional Programming",
   Addison-Wesley, p287.


*)


(* PERVASIVES *)

  val o: ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)


(* GENERAL *)

  val curry: ('a * 'b -> 'c) -> 'a -> 'b -> 'c
   (* curry f x y = f (x, y). *)
```

```
val uncurry: ('a -> 'b -> 'c) -> 'a * 'b -> 'c
 (* uncurry f (x, y) = f x y. *)


(* NEW COMBINATORS *)

  val I: 'a -> 'a
   (* I x; the identity function -- returns x.  I x = x. *)

  val K: 'a -> 'b -> 'a
   (* K x; the constant function -- when applied to y, returns x.
      K x y = x. *)

  val C: ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
   (* C f; swaps the arguments of a curried function.  (C f) x y = f y x. *)

  val C': ('a -> 'b -> 'c) -> ('d -> 'a) -> 'b -> 'd -> 'c
   (* C' k f; like C, except that an extra argument before f is unaltered.
      C' k x y z = k (x z) y.  (C (B A E1) E2 = C' A E1 E2). *)

  val CK: 'a -> 'b -> 'b
   (* CK; returns the second of two arguments.  (CK) x y = K y x = y. *)

  val W: ('a -> 'a -> 'b) -> 'a -> 'b
   (* W f; returns a function of one argument that passes that argument to
      both arguments of f.  W f x = f x x.  *)

  val B: ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
   (* B f g; curried function composition.  B f g x = f (g x). *)

  val B': ('a -> 'b -> 'c) -> 'a -> ('d -> 'b) -> 'd -> 'c
   (* B' k f g; like B, except that an extra argument before f is unaltered.
      B' k f g x = k f (g x).  (B (A x) y = B' A x y). *)

  val S: ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
   (* S f g x = f x (g x). *)

  val S': ('a -> 'b -> 'c) -> ('d -> 'a) -> ('d -> 'b) -> 'd -> 'c
   (* S' k f g; like S, except that an extra argument before f is unaltered.
      S' k f g x = k (f x) (g x).  (S (B A x) y = S' A x y). *)

  val Y: (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
   (* Y f; the fixed point combinator. If F is an operation on functions
      that returns a function of the same type, then Y F is the (least)
      fixed point of F satisfying F(Y F) = (Y F). *)
```

```
val cond: bool -> 'a -> 'a -> 'a
 (* cond b x y; returns x if b is true and y if b is false. *)

(* infix 3 oo *)
val oo: ('c -> 'd) * ('a -> 'b -> 'c) -> 'a -> 'b -> 'd
 (* f oo g; composition of a unary and curried binary function.
    (f oo g) x y = f (g x y). *)

(* infix 3 co *)
val co: ('b -> 'c -> 'd) * ('a -> 'b) -> 'c -> 'a -> 'd
 (* f co g; composition of a curried binary and a unary function.
    (f co g) x y = f (g y) x. *)

val can: ('a -> 'b) -> 'a -> bool
 (* can f x; returns true if f x doesn't raise an exception, false
    otherwise. *)

(* infix 3 fby *)
val fby: ('a -> 'b) * ('a -> 'c) -> 'a -> 'c
 (* f fby g; an application of f to x followed by (fby for short) an
    application of g to x.  Any exception raised by f is caught and ignored.
    This function is often given the name "then", but that clashes with the
    SML keyword. *)

(* infix 3 over *)
val over: ('a -> 'b) * ('a -> 'b) -> 'a -> 'b
 (* (f over g) x; returns f x unless that raises an exception in which case
   it returns g x. *)

end;
```

4.11 Const

```
(*$CONST *)

signature CONST =
sig

(* TAGGED VALUES

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 12 Dec 1989

Maintenance: Author


DESCRIPTION

   The 'a Const type associates a value and a unique tag.  The tests for
   equality compare the tags of their arguments.


NOTES

   If 'a doesn't admit equality, then neither does 'a Const.  However,
   the functions eq and ne will still work.

   Ideally we would like the constructor Const to be visible, so that
   we could use it like ref.  We can't do that because it would show
   the internal representation.




*)


(* TYPES *)

  type 'a Const


(* CREATORS *)

  val create: 'a -> 'a Const
   (* create x; associate a unique tag with x. *)
```

```
(* OBSERVERS *)

  val eq: 'a Const -> 'a Const -> bool
   (* eq x y; returns true if x and y are values associated with the same tag.
      Returns false otherwise. *)

  val ne: 'a Const -> 'a Const -> bool
   (* ne x y; returns true if x and y are values associated with different
      tags.  Returns false otherwise. *)


(* SELECTORS *)

  val !! : 'a Const -> 'a
   (* !! x; returns the value part of x, discarding the tag. *)

end
```

4.13 EQTYPE_PRINT

```
(*$EQTYPE_PRINT *)

signature EQTYPE_PRINT =
sig

(* AN EQTYPE WITH A PRINT FUNCTION

Created by:     Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           10 Feb 1991

Maintenance:    Author


DESCRIPTION

   This signature defines an eqtype T and a function to produce a string
   representation of a value of that type.


SEE ALSO

   PRINT, EQ_PRINT, EQTYPE_ORD, OBJECT




*)


(* TYPES *)

  eqtype T


(* CONVERTERS *)

  val string: T -> string
   (* string x; returns the usual string representation of x. *)

  val print: outstream -> T -> unit
   (* print os x; send the usual string representation of x to
      the stream os. *)
```

4.12 EQTYPE_ORD

```
(*$EQTYPE_ORD *)

signature EQTYPE_ORD =
sig

(* AN EQUALITY TYPE WITH AN ORDERING FUNCTION

Created by:     Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           10 Feb 1991

Maintenance:    Author


DESCRIPTION

   This signature defines a eqtype T and an ordering function.


SEE ALSO

   ORDERING, EQ_ORD, EQTYPE_PRINT, OBJECT.




*)


(* TYPES *)

  eqtype T


(* OBSERVERS *)

  val lt: T -> T -> bool
   (* lt x y; returns true is x is less than y; returns false otherwise. *)

end;
```

```
(* OBSERVERS *)

  val fixedWidth: bool
   (* fixedWidth; is true if the usual string representation of type T uses
      a fixed number of characters for all values. *)
end;
```

4.14 EQUALITY

```
(*$EQUALITY *)

signature EQUALITY =
sig

(* A TYPE WITH AN EQUALITY FUNCTION

Created by:     Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           22 Jan 1991

Maintenance:    Author


DESCRIPTION

   This signature defines a type T and an equality function.


SEE ALSO

   ORDERING, PRINT, EQ_PRINT, EQ_ORD, OBJECT, MONO_SET.



*)


(* TYPES *)

  type T


(* OBSERVERS *)

  val eq: T -> T -> bool
   (* eq x y; returns true if x and y are equal; returns false otherwise. *)

end;
```

4.15 EqFinMap

```
(*$EQ_FIN_MAP *)

signature EQ_FIN_MAP =
sig

(* FINITE MAPS

Created by: Nick Rothwell, LFCS, University of Edinburgh
nick@lfcs.ed.ac.uk
Date: 18 Feb 91

Maintenance: Author


DESCRIPTION

   Finite maps from 'a to 'b; requires equality on the 'a's.




*)


(* TYPES *)

  type (''a, 'b) Map


(* VALUES *)

  val empty : (''a, 'b) Map
(* Empty map. *)


(* CREATORS *)

  val singleton : ''a * 'b -> (''a , 'b) Map
(* Map containing a single element. *)


(* OBSERVERS *)

  val isEmpty: (''a, 'b) Map -> bool
(* Empty test on maps. *)
```

```
  val lookup  : (''a, 'b) Map -> ''a -> 'b Option
(* Look up an element in a map - may fail, returning None. *)

  val dom        : (''a, 'b) Map -> ''a list
(* Domain of a map. Might contain duplicates. *)

  val range      : (''a, 'b) Map -> 'b list
(* Range of a map.  Might contain elements mapped to by
   duplicate keys. *)


(* MANIPULATORS *)

  val add  : (''a * 'b) -> (''a, 'b) Map -> (''a, 'b) Map
(* Add an element to a map, rendering any existing mapping from that
   value unavailable. *)

  val plus       : (''a, 'b) Map -> (''a, 'b) Map -> (''a, 'b) Map
(* Add two maps together. Entries in the second map override entries
   on the first one (cf. the various plus operations in the SML
   semantics). *)

  val composeMap : ('b -> 'c) -> (''a, 'b) Map -> (''a, 'c) Map
(* Is this an appropriate name? Apply a function to all elements of
   the range. *)

  val fold       : (('a * 'b) -> 'b) -> 'b -> (''d, 'a) Map -> 'b
(* Rather like the list fold operation - operates on the range of
   a map. Order of elements not guaranteed. Also, suffers from the
   duplicate problem above. *)

  val fold'       : (((''a * 'b) * 'c) -> 'c)-> 'c -> (''a, 'b) Map -> 'c
(* More complex fold, with a function from (dom, range) element pairs
   to some arbitrary type. *)

  val mergeMap:
    (('b * 'b) -> 'b) -> (''a, 'b) Map -> (''a, 'b) Map -> (''a, 'b) Map
(* Merges two finite maps, with a composition function to apply
   to the range elements of domain elements which clash.  In such
   a case the first argument to the compose function is the range
   element of the first map argument to mergeMap.  *)
end;
```

4.16 EQ_ORD

```
(*$EQ_ORD *)

signature EQ_ORD =
sig

(* A TYPE WITH EQUALITY AND ORDERING FUNCTIONS

Created by:     Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           5 Feb 1991

Maintenance:    Author


DESCRIPTION

   This signature defines a type T, an equality function and an
   ordering function.


SEE ALSO

   EQUALITY, ORDERING, EQ_PRINT, EQTYPE_ORD, OBJECT.




*)


(* TYPES *)

  type T


(* OBSERVERS *)

  val eq: T -> T -> bool
   (* eq x y; returns true if x and y are equal; returns false otherwise. *)

  val lt: T -> T -> bool
   (* lt x y; returns true if x is less than y; returns false otherwise. *)

end;
```

4.17 EQ_PRINT

```
(*$EQ_PRINT *)

signature EQ_PRINT =
sig

(* A TYPE WITH A PRINT FUNCTION AND AN EQUALITY FUNCTION

Created by:     Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           5 Feb 1991

Maintenance:    Author


DESCRIPTION

   This signature defines a type T and a function to produce a string
   representation of a value of that type.


SEE ALSO

   PRINT, EQUALITY, EQTYPE_PRINT, EQ_ORD, OBJECT.




*)


(* TYPES *)

  type T


(* CONVERTERS *)

  val string: T -> string
   (* string x; returns the usual string representation of x. *)

  val print: outstream -> T -> unit
   (* print os x; send the usual string representation of x to
      the stream os. *)
```

```
(* OBSERVERS *)

  val fixedWidth: bool
   (* fixedWidth; is true if the usual string representation of type T uses
      a fixed number of characters for all values. *)

  val eq: T -> T -> bool
   (* eq x y; returns true if x and y are equal; returns false otherwise. *)

end;
```

4.18 EqSet

```
(*$EQ_SET *)

signature EQ_SET =
sig

(* SETS OVER EQUALITY TYPES

Created by:     Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           22 Jan 1991

Maintenance:    Author


DESCRIPTION

   This is the simplest of the three structures that implement sets.
   It provides polymorphic functions that are restricted to equality types.


NOTES

   An alternative name would be EQTYPE_SET, but that might be more appropriate
   for the case where 'a Set was itself an equality type.

SEE ALSO

   MONO_SET, SET.



*)

(* TYPES *)

  type 'a Set


(* CONSTANTS *)

  val empty: 'a Set
   (* empty; the empty set. *)
```

```
(* CREATORS *)

  val singleton: 'a -> 'a Set
   (* singleton x; returns the set containing only x. *)


(* CONVERTERS *)

  val list: 'a Set -> 'a list
   (* list s; return a list of the elements of s. *)

  val fromList: ''a list -> ''a Set
   (* fromList l; return the set of elements of l, removing duplicates. *)


(* OBSERVERS *)

  val size: 'a Set -> int
   (* size s; the number of elements in s. *)

  val isEmpty: 'a Set -> bool
   (* isEmpty s; returns true if s is empty, false otherwise. *)

  val member: ''a -> ''a Set -> bool
   (* member x s; returns true is x is in s, false otherwise. *)

  val eq: ''a Set -> ''a Set -> bool
   (* eq s s'; returns true if s and s' have the same elements. *)


(* SELECTORS *)

  exception Empty of string
   (* Empty fn; raised if the function named fn is erronously applied to
      the empty set. *)

  val select: 'a Set -> ('a * 'a Set)
   (* select s; returns a pair consiting of an element of s and the set
      of the remaining elements. *)


(* MANIPULATORS *)

  val difference: ''a Set -> ''a Set -> ''a Set
   (* difference s s'; returns the set of those elements of s that aren't
      also in s'.  *)
```

```
  val insert: ''a -> ''a Set -> ''a Set
   (* insert x s; returns the union of s and {x}. *)

  val intersect: ''a Set -> ''a Set -> ''a Set
   (* intersect s s'; returns the set of those elements that are in
      both s and s'. *)

  val remove: ''a -> ''a Set -> ''a Set
   (* remove x s; returns the set of the elements of s with x removed. *)

  val partition: (''a -> bool) -> ''a Set -> (''a Set * ''a Set)
   (* partition p s; returns a pair of sets; the first containing the elements
      of s for which the predicate p is true, the second the elements of s
      for which p is false. *)

  val union: ''a Set -> ''a Set -> ''a Set
   (* union s s'; returns the set of elements that are in either or both s
      and s'. *)

  val closure: (''a -> ''a Set) -> ''a Set -> ''a Set
   (* closure f s; repeatedly applies f to elements of s and the elements
      of the results of such applications, until no further elements are
      generated. *)
end;
```

4.19 General

```
(*$GENERAL *)

signature GENERAL =
sig

(* GENERAL DEFINITIONS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 21 Sep 89

Maintenance: Author


DESCRIPTION

   Types, exceptions and functions that are widely used or that don't fit
   anywhere else.

NOTES

   This would be the place to define the equality function, but that can't be
   redefined.



*)


(* PERVASIVES *)

  exception Bind and Match and Interrupt

  type exn and unit

  val o: ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
  val <> : ''a * ''a -> bool

  (* The next two exceptions are used by the redefinition of the
     pervasive arithmetic operations. *)
  exception Overflow
  and OldDiv


(* TYPES *)
```

```
  datatype 'a Option = None | Some of 'a
   (* Option is used when a value is optional, or when an operation may or
      may not return a value. *)

  type Nat
    sharing type Nat = int
   (* Nat is used when a function expects a positive integer or zero.  Such
      a function should raise the Nat exception if it is passed a negative
      integer. *)

  datatype ('a, 'b) Result = OK of 'a | Fail of 'b
   (* Result is used when a function can either succeed, returning a value,
      or fail, returning an error value. *)


(* SYSTEM *)

  exception NotImplemented of string
   (* NotImplemented fn; raised if the function called fn isn't provided
      in this implementation of the library. *)

  exception Nat of string * int
   (* Nat (fn, n); raised if the function named fn is passed a negative
      integer n to an argument of type Nat. *)


(* FUNCTIONS *)

  val id: 'a -> 'a
   (* id x; returns x.  The identity function. *)

  val curry: ('a * 'b -> 'c) -> 'a -> 'b -> 'c
   (* curry f; (curry f) x y = f (x, y). *)

  val uncurry: ('a -> 'b -> 'c) -> ('a * 'b -> 'c)
   (* uncurry f; (uncurry f) (x, y) = f x y. *)

  (* infix 3 oo *)
  val oo: ('c -> 'd) * ('a -> 'b -> 'c) -> 'a -> 'b -> 'd
   (* f oo g; composition of a unary and curried binary function.
     (f oo g) x y = f (g x y). *)

  (* infix 0 before *)
  val before: 'a * 'b -> 'a
   (* x before y; evaluates x and y in order, and returns the value of x. *)
```

```
  val iterate: Nat -> ('a -> 'a) -> 'a -> 'a
   (* iterate n f;  performs self-composition of f, n times.
      In other words, iterate n f base = f (f ... (f base) ...),
      with n occurrences of f. *)

  val repeat: Nat -> ('a -> 'b) -> 'a -> unit
   (* repeat n f arg; applies f to arg n times, presumably for
      the side effects performed by f.  In other words,
      repeat n f arg = (f arg; ...; f arg; ()). *)

  val until: ('a -> bool) -> ('a -> 'a) -> 'a -> 'a
   (* until p f arg; returns f (f ...  (f arg) ...) for the smallest number
      of application of f such that p applioed to the result is true.
      f is applied at least once. *)

  val primRec: ('a -> Nat -> 'a) -> 'a -> Nat -> 'a
   (* primRec f init n; Primitive recursion.  Returns the nth application
      of f, starting from init.  Differs from iterate in the type of f. *)

end
```

4.20 GeneralTypes

```
(*$GENERAL_TYPES: General *)

signature GENERAL_TYPES =

(* GENERAL TYPES AND EXCEPTIONS

Created by: Dave Berry, LFCS, University of Edinburgh
                 db@lfcs.ed.ac.uk
Date:          21 Feb 1989

Maintenance: Author


DESCRIPTION

   The types, exceptions and infix operators defined in General; these
   will be made available globally.




*)
sig
  datatype 'a Option = None | Some of 'a
    sharing type Option = General.Option

  type Nat
    sharing type Nat = int

  datatype ('a, 'b) Result = OK of 'a | Fail of 'b
    sharing type Result = General.Result

  (* The next two exceptions are used by the redefinition of the
     pervasive arithmetic operations. *)
  exception Overflow
  and OldDiv

  val oo: ('c -> 'd) * ('a -> 'b -> 'c) -> 'a -> 'b -> 'd

  val before: 'a * 'b -> 'a
end
```

4.21 Hash

```
(*$HASH *)

signature HASH =
sig

(* HASH TABLE

Created by:  Gene Rollins, School of Computer Science
             Carnegie-Mellon Univ., Pittsburgh, PA 15213
             rollins@cs.cmu.edu
Date:        1 January 1991
Maintenance: None

DESCRIPTION

This hash table module defines operations that expect the caller to supply
both the key and the hash value (the result of applying the hash function to
the key). One can create a hash table to map any (domain) type to any (range)
type, if at table creation time, one supplies an equality operation on the
domain type.



*)

   type ('a,'b) table

   val create : '_a -> ('_a * '_a -> bool) -> int -> '_b -> ('_a, '_b) table
(* fun create (sample'key :'_a) (equality :'_a * '_a -> bool)
              (table'size :int) (sample'value :'_b) :('_a,'_b) table
Create a hash table mapping 'a to 'b. *)

   val defaultSize : int
(* val defaultSize = 97 *)

   val defaultEqual : string * string -> bool
(* fun defaultEqual ((x :string), (y :string)) :bool
Regular string equality (x=y). *)

   val createDefault : '_a -> (string, '_a) table
(* fun createDefault (sample'value :'_b) :(string,'_b) table
Create a hash table mapping string to 'b; use defaultSize and defaultEqual. *)

   val enter : ('a,'b) table -> 'a -> int -> 'b -> unit
(* fun enter (hash'table :('a,'b) table) (key :'a) (hash :int) (value :'b)
```

```
Alter hash'table to map key to value. *)

   val remove : ('a,'b) table -> 'a -> int -> unit
(* fun remove (hash'table :('a,'b) table) (key :'a) (hash :int)
Remove key from the mapping. *)

   val lookup : ('a,'b) table -> 'a -> int -> 'b Option
(* fun lookup (hash'table :('a,'b) table) key hash :'b Option
Return the value that hash'table maps key into. *)

   val string : ('a,'b) table -> ('a -> string) -> ('b -> string) -> string
(* fun string (hash'table :('a,'b) table)
          (string'key :'a -> unit) (string'value :'b -> unit)
Return a string representation of the hash'table mapping, including bucket
indexes, keys, hash values, and range values. *)

   val print : outstream -> ('a,'b) table -> (outstream -> 'a -> unit) ->
       (outstream -> 'b -> unit) -> unit
(* fun print (os: outstream) -> (hash'table :('a,'b) table)
          (print'key :outstream -> 'a -> unit)
   (print'value :outstream -> 'b -> unit)
Print the hash'table mapping, including bucket indexes, keys, hash values,
and range values, on the stream os. *)

   val scan : ('a,'b) table -> ('a -> int -> 'b -> unit) -> unit
(* fun scan (hash'table :('a,'b) table) (operation :'a -> int -> 'b -> unit)
Apply operation to every entry in the table.  Arguments to operation are
key, hash value, domain value. *)

   val fold : ('a,'b) table -> ('a -> int -> 'b -> 'g -> 'g) -> 'g -> 'g
(* fun fold (hash'table :('a, 'b) table)
        (operation :'a -> int -> 'b -> 'g -> 'g) (init :'g) :'g
Apply operation to every entry in the table and an accumulated value.  The
accumulated value is initially init.  The result of applying operation
to one table entry is passed as the accumulated value for the next
application of operation.  The result of the last application of
operation is returned as the result of fold. *)

   val scanUpdate : ('a,'b) table -> ('a -> int -> 'b -> 'b) -> unit
(* fun scanUpdate (hash'table:('a,'b) table) (operation:'a -> int -> 'b -> 'b)
Apply operation to every entry in hash'table, and replace the domain value
for the entry with the result of operation. *)

   val eliminate : ('a,'b) table -> ('a -> int -> 'b -> bool) -> unit
(* fun eliminate (hash'table:('a,'b) table) (predicate:'a->int->'b->bool)
Apply predicate to every entry in hash'table.  If the predicate is true,
then remove the entry from the table. *)
```

```
   val bucketLengths : ('a,'b) table -> int -> int Array.Array
(* fun bucketLengths (hash'table :('a,'b) table) (maxlen :int) :int Array.Array
Creates an array with domain 0 through maxlen.  Counts the number of buckets in
hash'table of every length 0 through maxlen-1, putting those counts into
appropriate slots in the result array.  Also counts the number of buckets whose
length is maxlen or greater and puts that count into the maxlen slot of the
result array. *)

end
```

4.22 Instream

```
(*$INSTREAM: GeneralTypes *)

signature INSTREAM =
sig

(* INPUT STREAMS

Created by: Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:          12 Nov 1989

Maintenance: Author


DESCRIPTION

   Functions on input streams.


SEE ALSO

   OUTSTREAM, STREAM_PAIR


NOTES

   The pervasive values std_in, open_in, close_in and end_of_stream have
   been given synonyms that fit the library conventions.  The original names
   are still available.

   The input1 function is included to handle the common case of reading a
   single character; this function will remain valid even if input is
   changed to be curried.



*)


(* PERVASIVES *)

  type instream

  exception Io of string
```

```
  val std_in: instream
  val open_in: string -> instream
  val close_in: instream -> unit
  val input: instream * Nat -> string
  val lookahead: instream -> string
  val end_of_stream: instream -> bool


(* SYSTEM *)

  val openString: string -> instream
   (* openString s; returns an instream.  The characters read from this
      stream will be those in s, in order, with the end of s being read as
      an end of file. *)

  val canInput: instream -> Nat -> bool
   (* canInput i n; returns true if n characters can be read from i
      without blocking. *)

  val reset: instream -> bool
   (* reset i; if i can be reset to the beginning, in some sense,
     this is done and true is returned.  Otherwise false is returned. *)

  val interactive: instream -> bool
   (* interactive i; returns true if i is associated with an interactive
      device. *)


(* MANIPULATORS *)


  val stdIn: instream
   (* stdIn = std_in *)

  val openIn: string -> instream
   (* openIn = open_in *)

  val closeIn: instream -> unit
   (* closeIn = close_in *)

  val eof: instream -> bool
   (* eof = end_of_stream *)

  val read: instream -> Nat -> string
   (* read i n; returns the empty string if there are no characters
      remaining in i before an end of file.  Otherwise it returns a string
```

```
    of at least 1 and not more than n characters, whatever input is
    available from i. *)

  val input1: instream -> string
   (* input1 i; returns the first character on the instream i, or the empty
    string if an end of file is read.  Blocks if there is no character
    or end of file to be read. *)

  val readString: instream -> string -> (unit, string) Result
   (* readString i s; returns OK () if reading from i gives the characters
    in s.  Returns (Fail s') as soon as the end of file is reached or a
    character is read that doesn't match the corresponding one in s, where
    s' is the characters read so far.  In either case all characters read
    from i are lost.  *)

  val skip: (string -> bool) -> instream -> unit
   (* skip p i; reads all characters from i that satisfy p.  Leaves the first
    character that doesn't satisfy p to be read.  *)

  val inputLine : instream -> string
   (* inputLine i; returns a strings consisting of characters read from i
    up to and including the next end of line character.  If the end of the
    file is reached first, all characters up to the end of file are returned
    (without a new line character). *)
end
```

4.23 InstreamType

```
(*$INSTREAM_TYPE: Instream *)

signature INSTREAM_TYPE =

(* INSTREAM TYPE

Created by: Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           21 Feb 1989

Maintenance: Author


DESCRIPTION

   The instream type and the functions defined in the initial basis.



*)

sig

  (* PERVASIVES *)

  type instream
    sharing type instream = Instream.instream

  val std_in: instream

  val open_in: string -> instream

  val input: instream * int -> string

  val lookahead: instream -> string

  val close_in: instream -> unit

  val end_of_stream: instream -> bool
end
```

4.24 Int

```
(*$INT *)

signature INT =
sig

(* INTEGERS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 22 Sep 1989

Maintenance: Author


DESCRIPTION

   Standard functions on the built-in type "int".


NOTES

   The arithmetic exceptions are defined to raise Overflow for all
   overflow operations and Div for attempts to divide by zero.

   Possibly there should be functions dv, md and dvMd (with exceptions Dv
   and Md) that select the fastest of rem, mod, etc. for positive numbers.

   Possibly the function -- should raise an exception if start > finish + 1.

   Possibly there should be functions to read and write integers in binary
   as well as ascii.  Their specification would have say something about
   portability.

   Possibly there should be functions to read and write integers (to/from
   ascii) in different bases.



*)


(* PERVASIVES *)

   eqtype int
```

```
  exception Overflow
  and Div

  val + : int * int -> int
  val - : int * int -> int
  val * : int * int -> int
  val div: int * int -> int
  val mod: int * int -> int
  val ~ : int -> int
  val abs: int -> int
  val real: int -> real


(* SYSTEM *)

  val minInt: int Option
   (* minInt; the smallest integer that can be stored on the system, or
      None if the system supports arbitrary length integers. *)

  val maxInt: int Option
   (* maxInt; the largest integer that can be stored on the system, or
      None if the system supports arbitrary length integers. *)


(* TYPES *)

  eqtype T
    sharing type T = int


(* CONVERTERS *)

  val string: int -> string

  val print: outstream -> int -> unit


(* OBSERVERS *)

  val eq: int -> int -> bool

  val ne: int -> int -> bool

  val lt: int -> int -> bool

  val le: int -> int -> bool
```

```
  val gt: int -> int -> bool

  val ge: int -> int -> bool

  val fixedWidth: bool
   (* fixedWidth = false. *)


(* MANIPULATORS *)

  (* infix 7 divMod *)
  val divMod: int * int -> int * int
   (* x divMod y = (x div y, x mod y). *)

  (* infix 7 quot rem quotRem *)

  val quot: int * int -> int
   (* x quot y; like x div y but rounding toward zero. *)

  val rem: int * int -> int
   (* x rem y; like x mod y but rounding toward zero. *)

  val quotRem: int * int -> int * int
   (* x quotRem y = (x quot y, x rem y). *)

  val max: int -> int -> int
   (* max x y; returns the greater of x and y.  *)

  val min: int -> int -> int
   (* min x y; returns the lesser of x and y.  *)

  val maxMin: int -> int -> int * int
   (* maxMin x y = (max (x, y), min (x, y)) .  *)

  (* infix 5 -- *)
  val -- : int * int -> int list
   (* x -- y; returns the list of integers between x and y inclusive.
      Returns nil if x > y.  *)

  (* infix 8 ** *)
  exception Power of int * int
  val ** : int * int -> int
   (* x ** y; x raised to the power y. *)

end
```

4.25 IntParse

See PARSE.

4.26 List

```
(*$LIST: GeneralTypes *)

signature LIST =
sig

(* BASIC FUNCTIONS ON LISTS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 3 Oct 1989

Maintenance: Author


DESCRIPTION

   Functions on the built-in type 'a list.

   Many of these functions use an orthogonal naming scheme.  They can be
   grouped as follows:

   f;
     if f is "last", return the last element.
     if f is "nth", return the nth element (counting from zero).
     if f is "first", return the first element that satisfies a predicate p.
     if f is "all", return the list of elements that satisfy p.
     if f is "prefix", return the prefix of elements that satisfy p.

   dropF ... l; l with (f l) removed.
   splitF ... l; a pair of lists:
 (elements before f l, elements from f l to last l (inc.)).
   removeF ... l; a pair of (f l, dropF l).
   updateF ... l; l with (f l) replaced with a value.
   changeF ... l; l with (f l) replaced with (g (f l)).
   spliceF ... l; l with (f l) replaced with the elements in another list.
   insertF ... l; l with the elements in another list inserted before (f l).
   appendF ... l; l with the elements in another list inserted after (f l).

   splitLast, splitAll, and splitPrefix are not provided because "all" and
   "prefix" return lists and because splitLast would be the same as removeLast.

   changePrefix, updatePrefix, insertPrefix, appendPrefix, and splicePrefix
   are not provided, because "prefix" returns a list.

   The "last" functions (except appendLast) raise "Empty" if l is empty.
```

The "nth" functions raise "Subscript" if not (0 <= n < size l).
The "first" functions (except dropFirst) raise "First" if there aren't any elements that satisfy p.

SEE ALSO

LIST_PARSE

NOTES

Possibly there should be a dropExtract (aka delete?) function.

```
*)

(* PERVASIVES *)

  eqtype 'a list

  val map: ('a -> 'b) -> 'a list -> 'b list
  val rev: 'a list -> 'a list
  (* infixr 5 @ *)
  val @ : 'a list * 'a list -> 'a list


(* TYPES *)

  eqtype 'a T
    sharing type T = list


(* CONSTANTS *)

  val empty: 'a list
   (* empty = []. *)


(* CREATORS *)

  exception Size of string * int
   (* Size (fn, i); raised by the creation functions when they are called
      with a negative size argument. *)
```

```
  val create: Nat -> 'a -> 'a list
   (* create n e; create a list of size n, each element being e. *)

  val tabulate: Nat * (int -> 'a) -> 'a list
  (* tabulate (n, f); create a list of size n, sych that the element with
     index i is initialised to (f i). *)

  val tabulate': Nat * ('b -> 'a * 'b) * 'b -> 'a list
  (* tabulate' (n, f, base); create a list l of size n, with (l sub 0)
     initialised to (# 1 (f base)) and (l sub i (i > 0)) initialised to
     (# 1 (f (# 2 f_i))), where f_i is the result of the i^th application
     of f. *)


(* CONVERTORS *)

  val stringSep: string -> string -> string ->
                 ('a -> string) -> 'a list -> string
   (* stringSep start finish sep p l; returns the string representation of l,
      beginning with start, ending with finish, and with the elements
      separated by sep. *)

  val string: ('a -> string) -> 'a list -> string
   (* string p l; returns the canonical string representation of l. *)

  val printSep: outstream -> string -> string -> string ->
                (outstream -> 'a -> unit) -> 'a list -> unit
   (* printSep os start finish sep p l; sends the string representation of l
      to the stream os, beginning with start, ending with finish, and with
      the elements separated by sep. *)

  val print: outstream -> (outstream -> 'a -> unit) -> 'a list -> unit
   (* print os p l; sends the canonical string representation of l to
      the stream os. *)


(* OBSERVERS *)

  exception Subscript of string * int

  val size: 'a list -> Nat
   (* size l; returns the number of elements in l. *)

  val isEmpty: 'a list -> bool
   (* isEmpty l; returns true if l is the empty list. *)

  val exists: ('a -> bool) -> 'a list -> bool
```

```
  (* exists p l; true if there exists an element of l satisfying p. *)

 val forAll: ('a -> bool) -> 'a list -> bool
  (* forAll p l; true if every element of l satisfies p. *)

 val member: ''a -> ''a list -> bool
  (* member a l; true if a is an element of l. *)

 val eq: ('a -> 'a -> bool) -> 'a list -> 'a list -> bool
  (* eq x y; returns true if (size x = size y) and for all i,
     0 <= i < size x, (p (x sub i) (y sub i)). *)

 val ne: ('a -> 'a -> bool) -> 'a list -> 'a list -> bool
  (* ne x y; returns true if (size x <> size y) or there exists
     an i such that 0 <= i < size x and (p (x sub i) (y sub i)). *)

 val eq': ''a list -> ''a list -> bool
  (* eq' x y; returns true if (x = y). *)

 val ne': ''a list -> ''a list -> bool
  (* ne' x y; returns true (x <> y). *)

 val index: ('a -> bool) -> 'a list -> (int, unit) Result
  (* index p l; returns the position in l of the first element
     of l satisfying p. *)

 val prefixes: ''a list -> ''a list -> bool
  (* prefixes l1 l2; returns true if l1 is a prefix of l2.
     Raises (Subscript ("prefixes", n)) if not (0 <= n < size s2). *)


(* MANIPULATING THE LAST ELEMENT *)

 exception Empty of string

 val last: 'a list -> 'a
  (* last l; returns the last element of l.  Raises (Empty "last") if l
     is empty. *)

 val dropLast: 'a list -> 'a list
  (* dropLast l; returns l without its last element.
     Raises (Empty "dropLast") if l is empty. *)

 val removeLast: 'a list -> ('a * 'a list)
  (* removeLast l = (lastl, dropLast l).
     Raises (Empty "removeLast") if l is empty. *)
```

```
val updateLast: 'a -> 'a list -> 'a list
 (* updateLast v l; returns l with its last element replaced by v.
    Raises (Empty "updateLast") if l is empty. *)

val changeLast: ('a -> 'a) -> 'a list -> 'a list
 (* changeLast f l; returns l with (last l) replaced by (f (last l)).
    Raises (Empty "changeLast") if l is empty. *)

val insertLast: 'a list -> 'a list -> 'a list
 (* insertLast l' l; returns l with the elements of l' inserted before
    (last l).  Raises (Empty "insertLast") if l is empty. *)

val appendLast: 'a list -> 'a list -> 'a list
 (* appendLast l' l; returns l with the elements of l' appended after
    (last l). *)

val spliceLast: 'a list -> 'a list -> 'a list
 (* spliceLast l' l; returns l with (last l) replaced by the elements of l'.
    Raises (Empty "spliceLast") if l is empty. *)


(* MANIPULATING THE NTH ELEMENT *)

  (* infix 9 sub *)
  val sub: 'a list * int -> 'a
   (* l sub n; returns the n-1'th element of l.
      Raises (Subscript ("sub", n)) if not (0 <= n < size l). *)

  val nth: int -> 'a list -> 'a
   (* nth n l; returns the n-1'th element of l.
      Raises (Subscript ("nth", n)) if not (0 <= n < size l). *)

  val removeNth: int -> 'a list -> ('a * 'a list)
   (* removeNth n l= (l sub n, dropNth n l).
      Raises (Subscript ("removeNth", n)) if not (0 <= n < size l). *)

  val splitNth: int -> 'a list -> ('a list * 'a list)
   (* splitNth n l; returns ([l sub 0, ... l sub (n-1)], [l sub n, ... last l]).
      Raises (Subscript ("splitNth", n)) if not (0 <= n <= size l - 1). *)

  val dropNth: int -> 'a list -> 'a list
   (* dropNth n l; returns l without (l sub n).
      Raises (Subscript ("dropNth", n)) if not (0 <= n < size l). *)

  val updateNth: int -> 'a -> 'a list -> 'a list
   (* updateNth n v l; returns l with (l sub n) replaced by v.
      Raises (Subscript ("updateNth", n)) if not (0 <= n < size l). *)
```

```
  val changeNth: int -> ('a -> 'a) -> 'a list -> 'a list
   (* changeNth n f l; returns l with (l sub n) replaced by (f (l sub n)).
      Raises (Subscript ("changeNth", n)) if not (0 <= n < size l). *)

  val insertNth: int -> 'a list -> 'a list -> 'a list
   (* insertNth n l' l; returns l with the elements of l' inserted before
      (l sub n).
      Raises (Subscript ("insertNth", n)) if not (0 <= n < size l). *)

  val appendNth: int -> 'a list -> 'a list -> 'a list
   (* appendNth n l' l; returns l with the elements of l' appended after
      (l sub n).
      Raises (Subscript ("insertNth", n)) if not (0 <= n < size l). *)

  val spliceNth: int -> 'a list -> 'a list -> 'a list
   (* spliceNth n l' l; returns l with (l sub n) replaced by the elements of
      l'.  Raises (Subscript ("spliceNth", n)) if not (0 <= n < size l). *)


(* ACCESSING A RANGE OF ELEMENTS *)

  exception ExtractLast of int
  val extractLast: int -> 'a list -> 'a list
   (* extractLast l start ; returns the elements of l from (l sub start) to
      last l.  Raises (ExtractLast start) if not (0 <= start < size l). *)

  exception Extract of int * int
  val extract: int -> int -> 'a list -> 'a list
   (* extract start finish l; returns the elements of l from (l sub start) to
      (l sub (finish - 1)).  Returns [] if (start = finish).  Raises
      (Extract (start, finish)) if not (0 <= start <= finish <= size l). *)


(* MANIPULATING THE FIRST ELEMENT THAT SATISFIES A PREDICATE *)

  exception First of string

  val first: ('a -> bool) -> 'a list -> 'a
   (* first p l; returns the first element in l satisfying p.
      Raises (First "first") if p doesn't hold for any element of l. *)

  val dropFirst: ('a -> bool) -> 'a list -> 'a list
   (* dropFirst p l; returns l without the first of its elements (if any)
      that satisfy p. *)

  val removeFirst: ('a -> bool) -> 'a list -> ('a * 'a list)
```

```
  (* removeFirst p l = (first l, dropFirst l).
     Raises (First "removeFirst") if p doesn't hold for any element of l. *)

 val splitFirst: ('a -> bool) -> 'a list -> ('a list * 'a list)
  (* splitFirst p l; returns (extract 0 n l, extractLast n l),
     where (l sub n) is the first element of l that satisfies p.
     Raises (First "splitFirst") if p doesn't hold for any element of l. *)

 val updateFirst: ('a -> bool) -> 'a -> 'a list -> 'a list
  (* updateFirst p v l; returns l with (first p l) replaced by v.
     Raises (First "updateFirst") if there is no (first p l). *)

 val changeFirst: ('a -> bool) -> ('a -> 'a) -> 'a list -> 'a list
  (* changeFirst p f l; returns l with (first p l) replaced by
     (f (first p l)).  Raises (First "changeFirst") if there is no
     (first p l). *)

 val insertFirst: ('a -> bool) -> 'a list -> 'a list -> 'a list
  (* insertFirst p l' l; returns l with the elements of l' inserted before
     (first p l).  Raises (First "insertFirst") if there is no (first p l). *)

 val appendFirst: ('a -> bool) -> 'a list -> 'a list -> 'a list
  (* appendFirst p l' l; returns l with the elements of l' appended after
     (first p l).  Raises (First "insertFirst") if there is no (first p l). *)

 val spliceFirst: ('a -> bool) -> 'a list -> 'a list -> 'a list
  (* spliceFirst p l' l; returns l with (first p l) replaced by the elements
     of l'.  Raises (First "spliceFirst") if there is no (first p l). *)


(* TAKING A PREFIX OF ELEMENTS THAT SATISFY A PREDICATE *)

 val prefix: ('a -> bool) -> 'a list -> 'a list
  (* prefix p l; returns the largest prefix of l each of whose
     elements satisfies p *)

 val dropPrefix: ('a -> bool) -> 'a list -> 'a list
  (* dropPrefix p l; returns l without the largest prefix of l
     each of whose elements satisfies p *)

 val removePrefix: ('a -> bool) -> 'a list -> ('a list * 'a list)
  (* removePrefix p l = (prefix p l, dropPrefix p l). *)


(* MANIPULATING ALL ELEMENTS THAT SATISFY A PREDICATE *)

 val all: ('a -> bool) -> 'a list -> 'a list
```

```
  (* all p l: returns a list of the elements of l that satisfy p. *)

 val dropAll: ('a -> bool) -> 'a list -> 'a list
  (* dropAll p l: returns a list of the elements of l that don't satisfy p. *)

 val removeAll: ('a -> bool) -> 'a list -> ('a list * 'a list)
  (* removeAll p l = (all p l, dropAll p l). *)

 val updateAll: ('a -> bool) -> 'a -> 'a list -> 'a list
  (* updateAll p v l; returns l with each element that satisfies p
     replaced by v. *)

 val changeAll: ('a -> bool) -> ('a -> 'a) -> 'a list -> 'a list
  (* changeAll p f l; returns l with each element e that satisfies p
     replaced by (f e). *)

 val insertAll: ('a -> bool) -> 'a list -> 'a list -> 'a list
  (* insertAll p l' l; returns l with the elements of l' inserted before each
     element of l that satisfies p. *)

 val appendAll: ('a -> bool) -> 'a list -> 'a list -> 'a list
  (* appendAll p l' l; returns l with the elements of l' appended after each
     element of l that satisfies p. *)

 val spliceAll: ('a -> bool) -> 'a list -> 'a list -> 'a list
  (* spliceAll p l' l; returns l with each element that satisfies p
     replaced by the elements of l'. *)


(* OTHER MANIPULATORS *)

 val appendIfAll: ('a -> bool) -> 'a list -> 'a list -> 'a list
  (* appendIfAll p l' l; appends l' at the end of l if every element of
     l satisfies p. *)


(* ITERATORS *)

(* map is pervasive
 val map: ('a -> 'b) -> 'a list -> 'b list
  (* map f l; builds a list by applying f to each element of l. *)
*)

 val mapAll: ('a -> bool) -> ('a -> 'b) -> 'a list -> 'b list
  (* mapAll p f l; builds a list by applying f to each element of l that
     satisfies p. *)
```

```
val iterate: ('a * int -> 'b) -> 'a list -> 'b list
 (* iterate f l; builds a list by applying f to each element of l paired
    with its index. *)

val apply: ('a -> unit) -> 'a list -> unit
 (* apply f l; applies f to each element of l. *)

val applyAll: ('a -> bool) -> ('a -> unit) -> 'a list -> unit
 (* applyAll p f l; applies f to each element of l that satisfies p. *)

val iterateApply: ('a * int -> unit) -> 'a list -> unit
 (* iterateApply f l; applies f to each element of l paired
    with its index. *)


(* REDUCERS *)

  val foldR: ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
   (* foldR f base l; folds using f associating to the right over the
      base element.
      foldR f [a1,a2,...,an] base = f a1 (f a2 ... (op an base)...). *)

  val foldL: ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
   (* foldL f l base; folds using f associating to the left over the
      base element.
      foldL f [a1,a2,...,an] base = f an ... (f a2 (f a1 base))... . *)

  val foldR': ('a -> 'a -> 'a) -> 'a list -> 'a
   (* foldR' f l; folds using f associating to the right over the
      last element of l.  Raises (Empty "foldR'") if l is empty. *)

  val foldL': ('a -> 'a -> 'a) -> 'a list -> 'a
   (* foldL' f l; folds using f associating to the left over the
      first element of l.  Raises (Empty "foldL'") if l is empty. *)

  val pairwise: ('a -> 'a -> bool) -> 'a list -> bool
   (* pairwise f l; true if (f (l sub i) (l sub (i + 1))) is true for all
      0 <= i < size l, or if l is empty. *)
end
```

4.27 ListPair

```
(*$LIST_PAIR *)

signature LIST_PAIR =
sig

(* PAIRS OF LISTS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date:          4 Oct 1989

Maintenance: Author


DESCRIPTION

   Functions on the built-in type ('a list * 'b list).


SEE ALSO

   LIST, PAIR.



*)

(* MANIPULATORS *)

   exception Zip
   val zip: 'a list * 'b list -> ('a * 'b) list
    (* zip (l1, l2); transforms two lists into a list of pairs. It
       raises Zip if the lists are of different length. *)

   val unzip: ('a * 'b) list -> ('a list * 'b list)
    (* unzip l; transforms a list of pairs (l) into a pair of lists. *)

   val unravel: 'a list -> ('a list * 'a list)
    (* unravel l; yields a pair of lists. The elements are taken from
       l alternating one element for the first list and the following
       element for the second. *)

   val interleave: 'a list * 'a list -> 'a list
    (* interleave l1 l2; yields a list created  alternating the elements
       of l1 with the elements of l2. *)
```

```
    val merge: ('a -> 'a -> bool) -> ('a list * 'a list) -> 'a list
     (* merge p l1 l2; As interleave, merge yields a list created
        alternating the elements of l1 with the elements of l2.
        The order of insertion of a particular pair is determined
        by the predicate p. *)
end
```

4.28 ListParse

See SEQ_PARSE.

4.29 ListSort

```
(*$LIST_SORT *)

signature LIST_SORT =
sig

(* FUNCTIONS TO SORT AND PERMUTE LISTS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 6 Feb 1991

Maintenance: Author


DESCRIPTION

   Functions to sort and permute elements of a list.


NOTES

   These functions were originally in the main list structure.


SEE ALSO

   LIST.





*)


(* MANIPULATORS *)

  val perms: 'a list -> 'a list list
   (* perms l; returns a list whose elements are all the permutations of l*)

  val sort: ('a -> 'a -> bool) -> 'a list -> 'a list
   (* sort p l; returns l sorted by p. *)


end
```

4.30 Make

```
(* There's no point putting a tag on this signature! *)

signature MAKE =
sig

(* A PORTABLE MAKE SYSTEM

Created by:     Nick Rothwell, LFCS, University of Edinburgh
                nick@lfcs.ed.ac.uk
Date:           30 Oct 1990

Maintenance:    Author


DESCRIPTION

   Each piece of code should be preceded by a tag declaration in a special
   comment.  Tag declaration comments begin with the string "(*$" instead of
   the usual "(*", and must start at the beginning of a line.  Formatting
   characters are forbidden before and after the initial "(*$" string, but
   are permitted between tags and dependencies.

   The following tag declaration associates the tag "Foo" with the code
   that follows it:

   (*$Foo *)

   The following tag declaration associates the tag "Bar" with the code that
   follows it, and states that this code depends on the code associated
   with the tags "Foo1" and "Foo2":

   (*$Bar: Foo1 Foo2 *)

   The code associated with a tag is terminated by the next tag declaration
   or the end of a file.  Keeping the dependency information with each
   piece of code makes it easier to keep the two in sync.  Tag declaration
   comments are a subset of ordinary comments, so files containing them can
   be compiled without using make.

   The easiest way to consult the dependency information is to list
   the files containing the code in a file (called a tag file), and to call
   loadFrom with the name of that tag file.  This loads all the dependency
   information into the make system.  If you change the dependency
   information, use consultDecl or consultFile to update the stored
   dependencies.
```

```
    about those tags with the information found in the file.  Useful when
    the dependency information changes for a number of items in a file. *)

val whereIs: string -> string
 (* whereIs tag; returns the name of the file that Make thinks tag is in. *)

val loadFrom: string -> unit
 (* loadFrom tagFile; reads in a list of filenames from the named tag file,
    and does a consultFile on each one.  *)

val reload: unit -> unit
 (* reload (); reloads from the last used tag file. *)

val wipeCache: unit -> unit
 (* wipeCache (); throw away the source text in the cache (to reclaim
    some heap). *)

val forgetAll: unit -> unit
 (* forgetAll (); forget the compilation state of all objects. *)

val touch: string -> unit
 (* touch tag; manually say that tag has been changed. *)

val compiling_: string -> unit
 (* compiling msg; puts out a compiling message, turns on TraceML, hides
    the object in the cache.  *)

val OK_: string -> unit
 (* OK_ tag; says that tag has been recompiled.  Make puts these
    OK calls actually within the temp file containing Foo, just
    after its occurrence.  Turns off TraceML.  *)

val allOK: string list -> unit
 (* allOK l; like OK_, but marks all dependents as O.K. a well. It also
    takes a list, since List.map is most useful. Slower than OK_, but
    thorough.  *)

val make: string -> unit
 (* make tag; gathers the tags of all the dependencies of tag
    which have changed, extracts the text into a temporary file,
    and does a 'use' on this.  Also puts occurrences of (OK_ ("..."))
    within the temp file to tell the Make system that various objects
    are becoming up-to-date.  (make "Foo") also assumes that Foo
    has been altered.  *)

val again: unit -> unit
 (* again (); performs a "make" of previous tag. *)
```

```
   Once the dependency information has been loaded, call (make tag) to compile
   the code associated with the tag and all the code that it depends on.
   The system will write all the relevant code into a temporary file (called
   "%Make.tmp%" by default) and call use on that file.

   Once a piece of code has been read from a file, it is stored in a cache.
   When the code associated with a tag is changed, call (touch tag) to
   tell the make system that the current entry in the cache is obselete.
   consultDecl and consultFile do this automatically for the tags that they
   read.  If a piece of code fails to compile, then that entry in the cache
   is automatically touched as well.
   The system does not check the modification dates of files.

   Once all the changed pieces of code have been touched, call (again ()) to
   re-make your program.



(The following end of comment symbols really are needed.) *) *) *)

*)

(* MANIPULATORS *)

  val resides: string * string -> unit
   (* resides (tag, filename) says that the item identified by tag lives in
      the named file.  Be sure to use the same filename string for each
      occurrence of a file.  *)

  val standsAlone: string -> unit
   (* standsAlone tag; declares tag to be independent of anything, i.e.
      clears any dependency information.  *)

  val depends: string * string -> unit
   (* depends (tag1, tag2); expresses a dependency.  Circularity not checked
      or catered for.  *)

  val consultDecl: string -> unit
   (* consultDecl tag; scans the tag declaration, extracting it from the
      appropriate file, and replaces the stored information with that found
      in the file. *)

  val consultFile: string -> unit
   (* consultFile filename; scans the named fildeclaration, examining all the t
      declarations, and replace the dependency information currently stored
```

4.31 Memo

```
(*$MEMO: GeneralTypes *)

signature MEMO =
sig

(* MEMO FUNCTION PACKAGE

Created by: K. Mitchell, LFCS, University of Edinburgh
Date: April 1989.

Maintenance: Author


DESCRIPTION

   If inj[xy] are injective functions, f contains no side-effects, and
   expfn is an increasing function, then

   fun memo (expfn: nat -> nat)  <-- determines how the memo table expands
            (inj: 'a -> nat)     <-- converts argument into an array index
            (f: ('a -> '_b) -> 'a -> '_b)     <-- function to be memoised
            : ('a -> '_b) * ('a -> '_b)
            = let fun f' n = f f' n in (f', f') end;

   memo2 is defined using memo, and memo functions of higher arity may be
   constructed in a similar way.

   The first component of the result is a memoised version of the supplied
   function.  The second component is a memoised version that retains the
   table between calls.  See the examples for further details of its use.


EXAMPLES

   fun I x = x;
   fun expfn n = n;

   local
      fun mf fib 0 = 1
        | mf fib 1 = 1
        | mf fib n = fib(n-1) + fib(n-2);
   in val (_,fib) = memo expfn I mf end;


   local
```

```
  val makeTask: string * string -> unit
   (* makeTask (Tag, file); extracts the sources as does make, but puts
      them into the named file, without the "compiling_" and "OK_" calls.  *)

  val setTempfile: string -> unit
   (* setTempfile file; use the named file as the temporary file. *)

  val uncompiled: unit -> unit
   (* Print any tags which have never been compiled (or which have
      subsequently been touched). *)
end;
```

4.32 MonoArray

Functor:

```
(*$MonoArray : MONO_ARRAY EQ_PRINT Array *)

functor MonoArray (
  structure Element: EQ_PRINT
): MONO_ARRAY
```

Signature:

```
(*$MONO_ARRAY: GeneralTypes *)

signature MONO_ARRAY =
sig

(* ARRAYS OF A NAMED TYPE

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 12 Feb 1990

Maintenance: Author


DESCRIPTION

   A MonoArray is a single-dimensional monomorphic array of objects that
   allows its elements to be updated in place.  The first element has the index
   0.  Different MonoArrays are not equal, even if they contain the same
   elements.  MonoArrays always admit equality.  There is one empty
   MonoArray for each application of the functor.

   To create a MonoArray of MonoArrays, use the "tabulate" functions.

   Example MonoArrays are ByteArrays and BoolArrays, which will often
   be implemented specially.  For example, a BoolArray can use one bit
   per element.


SEE ALSO

   ARRAY, MONO_VECTOR.


*)
```

```
    fun mf ack 0 y = y + 1
      | mf ack x 0 = ack (x-1) 1
      | mf ack x y = ack (x-1) (ack x (y-1));
  in val (_,ack) = memo2 expfn I I mf end;


  fun length [] = 0 | length (_::t) = 1 + length t;

  (* length is injective for the values encountered in the cc function  *)

  local
    fun mf cc 0 _ = 1
      | mf cc _ [] = 0
      | mf cc amount  (kinds as (h::t)) =
            if amount < 0 then 0
            else if amount - h < 0 then cc amount t
            else cc (amount-h) kinds + cc amount t;
  in val (_,cc) = memo2 expfn I length mf end;



*)


(* MEMO FUNCTIONS *)

  val memo: (Nat -> Nat) -> ('a -> Nat) -> (('a -> '_b) -> 'a -> '_b) ->
          ('a -> '_b) * ('a -> '_b)
   (* memo expfn inj f; memoises a unary function.  See DESCRIPTION. *)

  val memo2: (Nat -> Nat) -> ('a -> Nat) -> ('_b -> Nat) ->
           (('a -> '_b -> '_c) -> 'a -> '_b -> '_c) ->
           ('a -> '_b -> '_c) * ('a -> '_b -> '_c)
   (* memo2 expfn inj1 inj2 f; memoises a curried binary function.  Can be
      defined in terms of memo. *)

  val memo3: (Nat -> Nat) -> ('a -> Nat) -> ('_b -> Nat) -> ('_c -> Nat) ->
           (('a -> '_b -> '_c -> '_d) -> 'a -> '_b -> '_c -> '_d) ->
           ('a -> '_b -> '_c -> '_d) * ('a -> '_b -> '_c -> '_d)
   (* memo3 expfn inj1 inj2 inj3 f; memoises a curried trinary function.
      Can be defined in terms of memo. *)

end
```

```
(* TYPES *)

  type Element
  type MonoArray
  type T
    sharing type T = MonoArray

(* CONSTANTS *)

  val empty: MonoArray
   (* empty; the empty array of this type. *)


(* CREATORS *)

  exception Size of string * int
   (* Size (fn, i); raised by the creation functions when they are invoked
      with a negative size. *)

  val create: Nat -> Element -> MonoArray
   (* create n i; create a MonoArray of n locations, each containing i.
      Raise (Size ("create", n)) if n < 0. *)

  val tabulate: Nat * (int -> Element) -> MonoArray
   (* tabulate (n, f); create a MonoArray v of n locations, (v sub 0) to
      (v sub (n-1)) with (v sub i) initialised to (f i).
      Raise (Size ("tabulate", n)) if n < 0. *)

  val tabulate': Nat * ('b -> Element * 'b) * 'b -> MonoArray
   (* tabulate' (n, f, base); create a MonoArray of n locations, (v sub 0) to
      (v sub (n-1)) with (v sub 0) initialised to (# 1 (f base)) and
      (v sub i (i > 0)) initialised to (# 1 (f (# 2 f_i))), where f_i is
      the result of the i^th application of f.
      Raise (Size ("tabulate'", n)) if n < 0. *)


(* CONVERTERS *)

  val stringSep: string -> string -> string ->
MonoArray -> string
   (* stringSep start finish sep p v; returns the string representation of v,
      beginning with start, ending with finish, and with the elements
      separated by sep. *)

  val string: MonoArray -> string
   (* string p v; returns the canonical string representation of v. *)
```

```
val printSep: outstream -> string -> string -> string -> MonoArray -> unit
 (* printSep os start finish sep v; sends the string representation of v
    to the stream os, beginning with start, ending with finish, and with
    the elements separated by sep. *)

val print: outstream -> MonoArray -> unit
 (* print os v; sends the canonical string representation of v to
    the stream os. *)

val fromList: Element list -> MonoArray
 (* fromList l; make an array containing (only) the elements of l, in
    the same order. *)

val list: MonoArray -> Element list
 (* list v; make a list containing (only) the elements of v, in
    the same order. *)


(* OBSERVERS *)

val isEmpty: MonoArray -> bool
 (* isEmpty v; returns true if v is empty. *)

val size: MonoArray -> Nat
 (* size v; return the number of elements in v. *)

val eq: MonoArray -> MonoArray -> bool
 (* eq x y; returns true if x and y are the same MonoArray. *)

val ne: MonoArray -> MonoArray -> bool
 (* ne x y; returns true if x and y are not the same MonoArray. *)


(* SELECTORS *)

exception Subscript of string * int

(* infix 9 sub *)
val sub: MonoArray * int -> Element
 (* v sub n; return the n+1'th element of v.
    Raise (Subscript ("sub", n)) if not (0 <= n <= size v). *)

val nth: int -> MonoArray -> Element
 (* nth n v; return the n+1'th element of v.
    Raise (Subscript ("nth", n)) if not (0 <= n <= size v). *)
```

```
  exception Extract of int * int
  val extract: int -> int -> MonoArray -> MonoArray
   (* extract start finish v; returns the sub-vector of v starting with
      (v sub start) and ending with (v sub (finish - 1)).
      Returns the empty vector if (start = finish).
      Raise (Extract (start, finish)) if not (0 <= start,finish <= size v). *)


(* MANIPULATORS *)

  val rev: MonoArray -> MonoArray
   (* rev v; builds a MonoArray containing the elements of v in
      reverse order. *)

  (* infix 6 ^ *)
  val ^ : MonoArray * MonoArray -> MonoArray
   (* v ^ v'; builds a new MonoArray containing the elements of v' appended
      to those of v. *)

  exception Update of int
  val update: MonoArray * int * Element -> unit
   (* update (v, n, i); replace (v sub n) with i.
      Raise (Update n) if not (0 <= n <= size v). *)

  exception Copy of int * int * int
  val copy: MonoArray -> int -> int -> MonoArray -> int -> unit
   (* copy v start finish v' start'; copies the sub-vector of v starting with
      (v sub start) and ending with (v sub (finish - 1)) to the MonoArray
      v', starting with (v' sub start').  Has no effect if (start = finish).
      Raises (Copy (start, finish, start')) if
      not (0 <= start,finish <= size v) or if
      not (0 <= start',start'+finish-start <= size v'). *)

  exception UpdateRange of int * int
  val updateRange: MonoArray -> int -> int -> Element -> unit
   (* updateRange v start finish i; update the elements of v starting with
      (v sub start) and ending with (v sub (finish - 1)) with i.  Has no effect
      if (start = finish).  Raises (UpdateRange (start, finish)) if
      not (0 <= start,finish <= size v). *)


(* REDUCERS *)

  val foldR: (Element -> 'b -> 'b) -> 'b -> MonoArray -> 'b
   (* foldR f base v; folds using f associating to the right over the
      base element.
      foldR f [a1,a2,...,an] base = f(a1,f(a2,...,f(an,base)...)). *)
```

```
  val foldL: (Element -> 'b -> 'b) -> 'b -> MonoArray -> 'b
   (* foldL f v base; folds using f associating to the left over the
      base element.
      foldL f [a1,a2,...,an] base = f(an,...,f(a2,f(a1,base))...). *)

  exception Empty of string

  val foldR': (Element -> Element -> Element) -> MonoArray -> Element
   (* foldR' f v; folds using f associating to the right over the
      last element of v.  Raises (Empty "foldR'") if v is empty. *)

  val foldL': (Element -> Element -> Element) -> MonoArray -> Element
   (* foldL' f v; folds using f associating to the right over the
      last element of v.  Raises (Empty "foldL'") if v is empty. *)

  val pairwise: (Element -> Element -> bool) -> MonoArray -> bool
   (* pairwise f v; true if (f (v sub i) (v sub (i + 1))) is true for all
      0 <= i < size v, or if v is empty. *)


(* ITERATORS *)

  val map: (Element -> Element) -> MonoArray -> MonoArray
   (* map f v; builds a new vector by applying f to each element of v. *)

  val apply: (Element -> unit) -> MonoArray -> unit
   (* apply f v; applies f to each element of v. *)

  val iterate: (Element * int -> Element) -> MonoArray -> MonoArray
   (* iterate f v; builds a new vector by applying f to each element of v
      paired with its index. *)

  val iterateApply: (Element * int -> unit) -> MonoArray -> unit
   (* iterate f v; applies f to each element of v paired with its index. *)
end
```

4.33 MonoArrayParse

```
functor MonoArrayParse (
  structure MonoArray: MONO_ARRAY
  structure Parse: PARSE
    sharing type Parse.T = MonoArray.Element
): MONO_SEQ_PARSE
```

4.34 MonoList

Functor:

```
(*$MonoList : MONO_LIST EQ_PRINT List *)

functor MonoList (
  structure Element: EQ_PRINT
): MONO_LIST
```

Signature:

```
(*$MONO_LIST: GeneralTypes *)

signature MONO_LIST =
sig

(* MONOMORPHIC LISTS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 15 Feb 1991

Maintenance: Author


DESCRIPTION

   Monomorphic lists.

   Many of these functions use the orthogonal naming scheme described in the
   LIST signature.  They can be grouped as follows:

   f;
     if f is "last", return the last element.
     if f is "nth", return the nth element (counting from zero).
     if f is "first", return the first element that satisfies a predicate p.
     if f is "all", return the list of elements that satisfy p.
     if f is "prefix", return the prefix of elements that satisfy p.

   dropF ... l; l with (f l) removed.
   splitF ... l; a pair of lists:
 (elements before f l, elements from f l to last l (inc.)).
   removeF ... l; a pair of (f l, dropF l).
   updateF ... l; l with (f l) replaced with a value.
   changeF ... l; l with (f l) replaced with (g (f l)).
   spliceF ... l; l with (f l) replaced with the elements in another list.
   insertF ... l; l with the elements in another list inserted before (f l).
```

```
   appendF ... l; l with the elements in another list inserted after (f l).

   splitLast, splitAll, and splitPrefix are not provided because "all" and
   "prefix" return lists and because splitLast would be the same as removeLast

   changePrefix, updatePrefix, insertPrefix, appendPrefix, and splicePrefix
   are not provided, because "prefix" returns a list.

   The "last" functions raise "Empty" if l is empty.
   The "nth" functions raise "Subscript" if not (0 <= n < size l).
   The "first" functions (except dropFirst) raise "First" if there aren't
any elements that satisfy p.


SEE ALSO

   LIST, MONO_VECTOR, MONO_ARRAY.


NOTES

   Possibly there should be a dropExtract (aka delete?) function.



*)


(* TYPES *)

  type MonoList
  type Element
  type T
    sharing type T = MonoList


(* CONSTANTS *)

  val empty: MonoList
   (* empty; the empty list. *)


(* CREATORS *)

  exception Size of string * int
```

```
 (* Size (fn, i); raised by the creation functions when they are invoked
    with a negative size. *)

val create: Nat -> Element -> MonoList
 (* create n e; create a list of size n, each element being e. *)

val tabulate: Nat * (int -> Element) * MonoList
(* tabulate (n, f); create a list of size n, sych that the element
   with index i is initialised to (f i). *)

val tabulate': Nat * ('b -> Element * 'b) * 'b -> MonoList
(* tabulate' (n, f, base); create a list l of size n, with (l sub 0)
   initialised to (# 1 (f base)) and (l sub i (i > 0)) initialised to
   (# 1 (f (# 2 f_i))), where f_i is the result of the i^th application
   of f. *)


(* CONVERTORS *)

val stringSep: string -> string -> string -> MonoList -> string
 (* stringSep start finish sep l; returns the string representation of l,
    beginning with start, ending with finish, and with the elements
    separated by sep. *)

val string: MonoList -> string
 (* string l; returns the canonical string representation of l. *)

val printSep: outstream -> string -> string -> string ->
              (outstream -> 'a -> unit) -> MonoList -> unit
 (* printSep os start finish sep p l; sends the string representation of l
    to the stream os, beginning with start, ending with finish, and with
    the elements separated by sep. *)

val print: outstream -> (outstream -> 'a -> unit) -> MonoList -> unit
 (* print os p l; sends the canonical string representation of l to
    the stream os. *)


(* OBSERVERS *)

val size: MonoList -> Nat
 (* size l; returns the number of elements in l. *)

val isEmpty: MonoList -> bool
 (* isEmpty l; returns true if l is the empty list. *)

val exists: (Element -> bool) -> MonoList -> bool
```

```
  (* exists p l; true if there exists an element of l satisfying p. *)

val forAll: (Element -> bool) -> MonoList -> bool
  (* forAll p l; true if every element of l satisfies p. *)

val member: Element -> MonoList -> bool
  (* member a l; true if a is an element of l. *)

val eq: MonoList -> MonoList -> bool
  (* eq x y; returns true if (size x = size y) and for all i,
     0 <= i < size x, (p (x sub i) (y sub i)). *)

val ne: MonoList -> MonoList -> bool
  (* ne x y; returns true if (size x <> size y) or there exists
     an i such that 0 <= i < size x and (p (x sub i) (y sub i)). *)

val index: (Element -> bool) -> MonoList -> (int, unit) Result
  (* index p l; returns the position in l of the first element
     of l satisfying p. *)


(* MANIPULATING THE LAST ELEMENT *)

exception Empty of string

val last: MonoList -> Element
  (* last l; returns the last element of l.  Raises (Empty "last") if l
     is empty. *)

val dropLast: MonoList -> MonoList
  (* dropLast l; returns l without its last element.
     Raises (Empty "dropLast") if l is empty. *)

val removeLast: MonoList -> (Element * MonoList)
  (* removeLast l = (lastl, dropLast l).
     Raises (Empty "removeLast") if l is empty. *)

val updateLast: Element -> MonoList -> MonoList
  (* updateLast v l; returns l with its last element replaced by v.
     Raises (Empty "updateLast") if l is empty. *)

val changeLast: (Element -> Element) -> MonoList -> MonoList
  (* changeLast f l; returns l with (last l) replaced by (f (last l)).
     Raises (Empty "changeLast") if l is empty. *)

val insertLast: MonoList -> MonoList -> MonoList
  (* insertLast l' l; returns l with the elements of l' inserted before
```

```
    (last l).  Raises (Empty "insertLast") if l is empty. *)

  val appendLast: MonoList -> MonoList -> MonoList
   (* appendLast l' l; returns l with the elements of l' appended after
      (last l).  Raises (Empty "insertLast") if l is empty. *)

  val spliceLast: MonoList -> MonoList -> MonoList
   (* spliceLast l' l; returns l with (last l) replaced by the elements of l'.
      Raises (Empty "spliceLast") if l is empty. *)


(* MANIPULATING THE NTH ELEMENT *)

  exception Subscript of string * int

  (* infix 9 sub *)
  val sub: MonoList * int -> Element
   (* l sub n; returns the n-1'th element of l.
      Raises (Subscript ("sub", n)) if not (0 <= n < size l). *)

  val nth: int -> MonoList -> Element
   (* nth n l; returns the n-1'th element of l.
      Raises (Subscript ("nth", n)) if not (0 <= n < size l). *)

  val removeNth: int -> MonoList -> (Element * MonoList)
   (* removeNth n l= (l sub n, dropNth n l).
      Raises (Subscript ("removeNth", n)) if not (0 <= n < size l). *)

  val splitNth: int -> MonoList -> (MonoList * MonoList)
   (* splitNth n l; returns ([l sub 0, ... l sub (n-1)], [l sub n, ... last l]).
      Raises (Subscript ("splitNth", n)) if not (0 <= n <= size l - 1). *)

  val dropNth: int -> MonoList -> MonoList
   (* dropNth n l; returns l without (l sub n).
      Raises (Subscript ("dropNth", n)) if not (0 <= n < size l). *)

  val updateNth: int -> Element -> MonoList -> MonoList
   (* updateNth n v l; returns l with (l sub n) replaced by v.
      Raises (Subscript ("updateNth", n)) if not (0 <= n < size l). *)

  val changeNth: int -> (Element -> Element) -> MonoList -> MonoList
   (* changeNth n f l; returns l with (l sub n) replaced by (f (l sub n)).
      Raises (Subscript ("changeNth", n)) if not (0 <= n < size l). *)

  val insertNth: int -> MonoList -> MonoList -> MonoList
   (* insertNth n l' l; returns l with the elements of l' inserted before
      (l sub n).
```

```
    Raises (Subscript ("insertNth", n)) if not (0 <= n < size l). *)

  val appendNth: int -> MonoList -> MonoList -> MonoList
   (* appendNth n l' l; returns l with the elements of l' appended after
    (l sub n).
    Raises (Subscript ("insertNth", n)) if not (0 <= n < size l). *)

  val spliceNth: int -> MonoList -> MonoList -> MonoList
   (* spliceNth n l' l; returns l with (l sub n) replaced by the elements of
    l'.  Raises (Subscript ("spliceNth", n)) if not (0 <= n < size l). *)


(* ACCESSING A RANGE OF ELEMENTS *)

  exception ExtractLast of int
  val extractLast: int -> MonoList -> MonoList
   (* extractLast l start ; returns the elements of l from (l sub start) to
    last l.  Raises (ExtractLast start) if not (0 <= start < size l). *)

  exception Extract of int * int
  val extract: int -> int -> MonoList -> MonoList
   (* extract start finish l; returns the elements of l from (l sub start) to
    (l sub (finish - 1)).  Returns [] if (start = finish).  Raises
    (Extract (start, finish)) if not (0 <= start <= finish <= size l). *)


(* MANIPULATING THE FIRST ELEMENT THAT SATISFIES A PREDICATE *)

  exception First of string

  val first: (Element -> bool) -> MonoList -> Element
   (* first p l; returns the first element in l satisfying p.
    Raises (First "first") if p doesn't hold for any element of l. *)

  val dropFirst: (Element -> bool) -> MonoList -> MonoList
   (* dropFirst p l; returns l without the first of its elements (if any)
    that satisfy p. *)

  val removeFirst: (Element -> bool) -> MonoList -> (Element * MonoList)
   (* removeFirst p l = (first l, dropFirst l).
    Raises (First "removeFirst") if p doesn't hold for any element of l. *)

  val splitFirst: (Element -> bool) -> MonoList -> (MonoList * MonoList)
   (* splitFirst p l; returns (extract 0 n l, extractLast n l),
    where (l sub n) is the first element of l that satisfies p.
    Raises (First "splitFirst") if p doesn't hold for any element of l. *)
```

```
val updateFirst: (Element -> bool) -> Element -> MonoList -> MonoList
 (* updateFirst p v l; returns l with (first p l) replaced by v.
    Raises (First "updateFirst") if there is no (first p l). *)

val changeFirst: (Element -> bool) -> (Element -> Element) ->
 MonoList -> MonoList
 (* changeFirst p f l; returns l with (first p l) replaced by
    (f (first p l)).  Raises (First "changeFirst") if there is no
    (first p l). *)

val insertFirst: (Element -> bool) -> MonoList -> MonoList -> MonoList
 (* insertFirst p l' l; returns l with the elements of l' inserted before
    (first p l).  Raises (First "insertFirst") if there is no (first p l). *)

val appendFirst: (Element -> bool) -> MonoList -> MonoList -> MonoList
 (* appendFirst p l' l; returns l with the elements of l' appended after
    (first p l).  Raises (First "insertFirst") if there is no (first p l). *)

val spliceFirst: (Element -> bool) -> MonoList -> MonoList -> MonoList
 (* spliceFirst p l' l; returns l with (first p l) replaced by the elements
    of l'.  Raises (First "spliceFirst") if there is no (first p l). *)


(* TAKING A PREFIX OF ELEMENTS THAT SATISFY A PREDICATE *)

val prefix: (Element -> bool) -> MonoList -> MonoList
 (* prefix p l; returns the largest prefix of l each of whose
    elements satisfies p *)

val dropPrefix: (Element -> bool) -> MonoList -> MonoList
 (* dropPrefix p l; returns l without the largest prefix of l
    each of whose elements satisfies p *)

val removePrefix: (Element -> bool) -> MonoList -> (MonoList * MonoList)
 (* removePrefix p l = (prefix p l, dropPrefix p l). *)


(* MANIPULATING ALL ELEMENTS THAT SATISFY A PREDICATE *)

val all: (Element -> bool) -> MonoList -> MonoList
 (* all p l: returns a list of the elements of l that satisfy p. *)

val dropAll: (Element -> bool) -> MonoList -> MonoList
 (* dropAll p l: returns a list of the elements of l that don't satisfy p. *)

val removeAll: (Element -> bool) -> MonoList -> (MonoList * MonoList)
 (* removeAll p l = (all p l, dropAll p l). *)
```

```
  val updateAll: (Element -> bool) -> Element -> MonoList -> MonoList
   (* updateAll p v l; returns l with each element that satisfies p
      replaced by v. *)

  val changeAll: (Element -> bool) -> (Element -> Element) ->
MonoList -> MonoList
   (* changeAll p f l; returns l with each element e that satisfies p
      replaced by (f e). *)

  val insertAll: (Element -> bool) -> MonoList -> MonoList -> MonoList
   (* insertAll p l' l; returns l with the elements of l' inserted before each
      element of l that satisfies p. *)

  val appendAll: (Element -> bool) -> MonoList -> MonoList -> MonoList
   (* appendAll p l' l; returns l with the elements of l' appended after each
      element of l that satisfies p. *)

  val spliceAll: (Element -> bool) -> MonoList -> MonoList -> MonoList
   (* spliceAll p l' l; returns l with each element that satisfies p
      replaced by the elements of l'. *)


(* OTHER MANIPULATORS *)

  val appendIfAll: (Element -> bool) -> MonoList -> MonoList -> MonoList
   (* appendIfAll p l' l; appends l' at the end of l if every element of
      l satisfies p. *)

  val rev: MonoList -> MonoList
   (* rev l; builds a new MonoList containing the elements of l in reverse
      order. *)

  (* infixr 5 @ *)
  val @ : MonoList * MonoList -> MonoList
   (* l @ l'; builds a new MonoVector containing the elements of l' appended
      to those of l. *)


(* ITERATORS *)

  val map: (Element -> Element) -> MonoList -> MonoList
   (* map f l; builds a new MonoList by applying f to each element of l. *)

  val mapAll: (Element -> bool) -> (Element -> Element) -> MonoList -> MonoList
   (* mapAll p f l; builds a new MonoList by applying f to each element
      of l that satisfies p. *)
```

```
  val iterate: (Element * int -> Element) -> MonoList -> MonoList
   (* iterate f l; builds a new MonoList by applying f to each element
      of l paired with its index. *)

  val apply: (Element -> unit) -> MonoList -> unit
   (* apply f l; applies f to each element of l. *)

  val applyAll: (Element -> bool) -> (Element -> unit) -> MonoList -> unit
   (* applyAll p f l; applies f to each element of l that satisfies p. *)

  val iterateApply: (Element * int -> unit) -> MonoList -> unit
   (* iterateApply f l; applies f to each element of l paired
      with its index. *)


(* REDUCERS *)

  val foldR: (Element -> 'b -> 'b) -> 'b -> MonoList -> 'b
   (* foldR f base l; folds using f associating to the right over the
      base element.
      foldR f [a1,a2,...,an] base = f a1 (f a2 ... (op an base)...). *)

  val foldL: (Element -> 'b -> 'b) -> 'b -> MonoList -> 'b
   (* foldL f l base; folds using f associating to the left over the
      base element.
      foldL f [a1,a2,...,an] base = f an ... (f a2 (f a1 base))... . *)

  val foldR': (Element -> Element -> Element) -> MonoList -> Element
   (* foldR' f l; folds using f associating to the right over the
      last element of l.  Raises (Empty "foldR'") if l is empty. *)

  val foldL': (Element -> Element -> Element) -> MonoList -> Element
   (* foldL' f l; folds using f associating to the left over the
      first element of l.  Raises (Empty "foldL'") if l is empty. *)

  val pairwise: (Element -> Element -> bool) -> MonoList -> bool
   (* pairwise f l; true if (f (l sub i) (l sub (i + 1))) is true for all
      0 <= i < size l, or if l is empty. *)
end
```

4.35 MONO_SEQ_PARSE

```
(*$MONO_SEQ_PARSE: InstreamType GeneralTypes *)

signature MONO_SEQ_PARSE =

(* PARSE FUNCTIONS FOR MONOMORPHIC SEQUENCES

Created by: Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           21 Feb 1989

Maintenance: Author


DESCRIPTION

   Parse and read functions for monomorphic sequences and arrays,
   such as ByteVectors and BoolVectors.

   The design parallels that of the SEQ_PARSE signature, but replaces
   occurrences of 'a T with T.  See SEQ_PARSE for descrptions of the
   functions.


SEE ALSO

   SEQ_PARSE, MonoVectorParse, MonoArrayParse.



*)

sig

(* TYPES *)

  type T

(* CONVERTERS *)

  exception Sep of string * string * string * string

  exception Size of string * int

  val parseSepN: string -> string -> string ->
```

```
 Nat -> string -> (T * string, T Option * string) Result

  val parseSep: string -> string -> string ->
                string -> (T * string, T Option * string) Result

  val parseN: Nat -> string -> (T * string, T Option * string) Result

  val parse: string -> (T * string, T Option * string) Result

  val readSep: string -> string -> string -> instream -> (T, T Option) Result

  val readSepN: string -> string -> string ->
Nat -> instream -> (T, T Option) Result

  val readN: Nat -> instream -> (T, T Option) Result

  val read: instream -> (T, T Option) Result

  val fromFile: string -> T

  val file: T -> string -> unit

end
```

4.36 MonoSet

Functor:

```
(*$MonoSet: MONO_SET EQUALITY *)

functor MonoSet (
  structure Element: EQUALITY
): MONO_SET
```

Signature:

```
(*$MONO_SET *)

signature MONO_SET =
sig

(* SETS OF A GIVEN TYPE

Created by:     Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           22 Jan 1991

Maintenance:    Author


DESCRIPTION

   Monomorphic sets.


SEE ALSO

   EQ_SET, SET.




*)


(* TYPES *)

  type Element
  type Set
  type T
    sharing type T = Set
```

```
(* CONSTANTS *)

  val empty: Set
   (* empty; the empty set. *)


(* CREATORS *)

  val singleton: Element -> Set
   (* singleton x; returns the set containing only x. *)


(* CONVERTERS *)

  val list: Set -> Element list
   (* list s; return a list of the elements of s. *)

  val fromList: Element list -> Set
   (* fromList l; return the set of elements of l, removing duplicates. *)


(* OBSERVERS *)

  val size: Set -> int
   (* size s; the number of elements in s. *)

  val isEmpty: Set -> bool
   (* isEmpty s; returns true if s is empty, false otherwise. *)

  val member: Element -> (Set -> bool)
   (* member x s; returns true is x is in s, false otherwise. *)

  val eq: Set -> Set -> bool
   (* eq s s'; returns true if s and s' have the same elements. *)


(* SELECTORS *)

  exception Empty of string
   (* Empty fn; raised if the function named fn is erronously applied to
      the empty set. *)

  val select: Set -> (Element * Set)
   (* select s; returns a pair consiting of an element of s and the set
      of the remaining elements. *)
```

```
(* MANIPULATORS *)

  val difference: Set -> Set -> Set
   (* difference s s'; returns the set of those elements of s that aren't
      also in s'.  *)

  val insert: Element -> Set -> Set
   (* insert x s; returns the union of s and {x}. *)

  val intersect: Set -> Set -> Set
   (* intersect s s'; returns the set of those elements that are in
      both s and s'. *)

  val remove: Element -> Set -> Set
   (* remove x s; returns the set of the elements of s with x removed. *)

  val partition: (Element -> bool) -> (Set -> Set * Set)
   (* partition p s; returns a pair of sets; the first containing the elements
      of s for which the predicate p is true, the second the elements of s
      for which p is false. *)

  val union: Set -> Set -> Set
   (* union s s'; returns the set of elements that are in either or both s
      and s'. *)

  val closure: (Element -> Set) -> Set -> Set
   (* closure f s; repeatedly applies f to elements of s and the elements
      of the results of such applications, until no further elements are
      generated. *)
end;
```

4.37 MonoVector

Functor:

```
(*$MonoVector : MONO_VECTOR EQ_PRINT Vector *)

functor MonoVector (
  structure Element: EQ_PRINT
): MONO_VECTOR
```

Signature:

```
(*$MONO_VECTOR: GeneralTypes *)

signature MONO_VECTOR =
sig

(* CONSTANT VECTORS OF A NAMED TYPE

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 31 Oct 1989

Maintenance: Author


DESCRIPTION

   A MonoVector is a single-dimensional constant monomorphic array of objects.
   The first element has the index 0.  MonoVectors are equal if they contain
   the same elements and these elements admit equality.

   To create a MonoVector of MonoVectors, use the "tabulate" functions.

   Example MonoVectors are ByteVectors and BoolVectors, which will often
   be implemented specially.  For example, a BoolVector can use one bit
   per element.


SEE ALSO

   VECTOR, MONO_ARRAY.


*)
```

```
(* TYPES *)

  type Element
  type MonoVector
  type T
    sharing type T = MonoVector


(* CONSTANTS *)

  val empty: MonoVector
   (* empty; the empty vector of this type. *)


(* CREATORS *)

  exception Size of string * int
   (* Size (fn, i); raised by the creation functions when they are invoked
      with a negative size. *)

  val create: Nat -> Element -> MonoVector
   (* create n i; create a MonoVector of n locations, each containing i.
      Raise (Size ("create", n)) if n < 0. *)

  val tabulate: Nat * (int -> Element) -> MonoVector
   (* tabulate n f; create a MonoVector v of n locations, (v sub 0) to
      (v sub (n-1)) with (v sub i) initialised to (f i).
      Raise (Size ("tabulate", n)) if n < 0. *)

  val tabulate': Nat * ('b -> Element * 'b) * 'b -> MonoVector
   (* tabulate' n f base; create a MonoVector of n locations, (v sub 0) to
      (v sub (n-1)) with (v sub 0) initialised to (# 1 (f base)) and
      (v sub i (i > 0)) initialised to (# 1 (f (# 2 f_i))), where f_i is
      the result of the i^th application of f.
      Raise (Size ("tabulate'", n)) if n < 0. *)


(* CONVERTERS *)

  val stringSep: string -> string -> string -> MonoVector -> string
   (* stringSep start finish sep p v; returns the string representation
      of v, beginning with start, ending with finish, and with the elements
      separated by sep. *)

  val string: MonoVector -> string
   (* string v; returns the canonical string representation of v. *)
```

```
val printSep: outstream -> string -> string -> string -> MonoVector -> unit
 (* printSep os start finish sep v; sends the string representation of v
    to the stream os, beginning with start, ending with finish, and with
    the elements separated by sep. *)

val print: outstream -> MonoVector -> unit
 (* print os v; sends the canonical string representation of v to
    the stream os. *)

val list: MonoVector -> Element list
 (* list v; make a list containing (only) the elements of v, in
    the same order. *)

val fromList: Element list -> MonoVector
 (* fromList l; make a Vector containing (only) the elements of l, in
    the same order. *)


(* OBSERVERS *)

val isEmpty: MonoVector -> bool
 (* isEmpty v; returns true if v is empty. *)

val size: MonoVector -> Nat
 (* size v; return the number of elements in v. *)

val eq: MonoVector -> MonoVector -> bool
 (* eq x y; returns true if (size x = size y) and for all i,
    0 <= i <= size x, (Element.eq (x sub i) (y sub i)). *)

val ne: MonoVector -> MonoVector -> bool
 (* ne x y; returns true if (size x <> size y) and there exists an i
    such that 0 <= i <= size x and (Element.ne (x sub i) (y sub i)). *)


(* SELECTORS *)

exception Subscript of string * int

(* infix 9 sub *)
val sub: MonoVector * int -> Element
 (* v sub n; return the n+1'th element of v.
    Raise (Subscript ("sub", n)) if not (0 <= n <= size v). *)

val nth: int -> MonoVector -> Element
 (* nth n v; return the n+1'th element of v.
    Raise (Subscript ("nth", n)) if not (0 <= n <= size v). *)
```

```
exception Extract of int * int
val extract: int -> int -> MonoVector -> MonoVector
 (* extract start finish v; returns the sub-vector of v starting with
    (v sub start) and ending with (v sub (finish - 1)).
    Returns the empty vector if (start = finish).
    Raise (Extract (start, finish)) if not (0 <= start,finish <= size v). *


(* MANIPULATORS *)

  val rev: MonoVector -> MonoVector
   (* rev v; builds a new MonoVector containing the elements of v in
      reverse order. *)

  (* infix 6 ^ *)
  val ^ : MonoVector * MonoVector -> MonoVector
   (* v ^ v'; builds a new MonoVector containing the elements of v' appended
      to those of v. *)


(* REDUCERS *)

  val foldR: (Element -> 'b -> 'b) -> 'b -> MonoVector -> 'b
   (* foldR f base v; folds using f associating to the right over the
      base element.
      foldR f [a1,a2,...,an] base = f(a1,f(a2,...,f(an,base)...)). *)

  val foldL: (Element -> 'b -> 'b) -> 'b -> MonoVector -> 'b
   (* foldL f v base; folds using f associating to the left over the
      base element.
      foldL f [a1,a2,...,an] base = f(an,...,f(a2,f(a1,base))...). *)

  exception Empty of string

  val foldR': (Element -> Element -> Element) -> MonoVector -> Element
   (* foldR' f v; folds using f associating to the right over the
      last element of v.  Raises (Empty "foldR'") if v is empty. *)

  val foldL': (Element -> Element -> Element) -> MonoVector -> Element
   (* foldL' f v; folds using f associating to the right over the
      last element of v.  Raises (Empty "foldL'") if v is empty. *)

  val pairwise: (Element -> Element -> bool) -> MonoVector -> bool
   (* pairwise f v; true if (f (v sub i) (v sub (i + 1))) is true for all
      0 <= i < size v, or if v is empty. *)
```

```
(* ITERATORS *)

  val map: (Element -> Element) -> MonoVector -> MonoVector
   (* map f v; builds a new vector by applying f to each element of v. *)

  val apply: (Element -> unit) -> MonoVector -> unit
   (* apply f v; applies f to each element of v. *)

  val iterate: (Element * int -> Element) -> MonoVector -> MonoVector
   (* iterate f v; builds a new vector by applying f to each element of v
      paired with its index. *)

  val iterateApply: (Element * int -> unit) -> MonoVector -> unit
   (* iterate f v; applies f to each element of v paired with its index. *)
end
```

4.38 MonoVectorParse

```
functor MonoVectorParse (
  structure MonoVector: MONO_VECTOR
  structure Parse: PARSE
    sharing type Parse.T = MonoVector.Element
): MONO_SEQ_PARSE
```

4.39 OBJECT

```
(*$OBJECT *)

signature OBJECT =
sig

(* OBJECT DEFINITION

Created by: Dave Berry, LFCS, University of Edinburgh
                 db@lfcs.ed.ac.uk
Date:            24 Jan 1990

Maintenance: Author


DESCRIPTION

   An Object provides a type T, functions for comparing values of that
   type, and a function for producing a string representation of that
   type.  A full set of comparison functions is given, not just lt and eq.


SEE ALSO

   EQUALITY, ORDERING, PRINT, EQ_ORD, EQ_PRINT, EQTYPE_ORD, EQTYPE_PRINT,
   SEQUENCE.



*)

(* TYPES *)

  type T


(* CONVERTERS *)

  val string: T -> string
   (* string t; returns the string representation of t. *)

  val print: outstream -> T -> unit
   (* print os x; send the usual string representation of x to
      the stream os. *)
```

```
(* OBSERVERS *)

  val eq: T -> T -> bool
   (* eq x y; returns true if x and y are equal.  Usually this is x = y. *)

  val ne: T -> T -> bool
   (* ne x y; returns true if x and y are unequal.  Usually this is x <> y. *)

  val lt: T -> T -> bool
   (* lt x y; returns true if x is less than y. *)

  val le: T -> T -> bool
   (* le x y; returns true if x is less than or equal to y. *)

  val gt: T -> T -> bool
   (* gt x y; returns true if x is less than y. *)

  val ge: T -> T -> bool
   (* ge x y; returns true if x is less than or equal to y. *)

  val fixedWidth: bool
   (* fixedWidth; is true if the usual string representation of type T uses
      a fixed number of characters for all values. *)
end
```

4.40 ORDERING

```
(*$ORDERING *)

signature ORDERING =
sig

(* A TYPE WITH AN ORDERING FUNCTION

Created by:     Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           5 Feb 1991

Maintenance:    Author


DESCRIPTION

   This signature defines a type T and an ordering function.


SEE ALSO

   EQUALITY, PRINT, EQ_ORD, EQTYPE_ORD, OBJECT.




*)


(* TYPES *)

  type T


(* OBSERVERS *)

  val lt: T -> T -> bool
   (* lt x y; returns true if x is less than y; returns false otherwise. *)

end;
```

4.41 ORD_PRINT

```
(*$ORD_PRINT *)

signature ORD_PRINT =
sig

(* A TYPE WITH A PRINT FUNCTION AND AN ORDERING FUNCTION

Created by:     Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           10 Feb 1991

Maintenance:    Author


DESCRIPTION

   This signature defines a type T, a function to produce a string
   representation of a value of that type, and a function to compare
   two values of that type.


SEE ALSO

   PRINT, ORDERING, EQ_ORD, EQ_PRINT, EQTYPE_PRINT, OBJECT


*)


(* TYPES *)

  type T


(* CONVERTERS *)

  val string: T -> string
   (* string x; returns the usual string representation of x. *)

  val print: outstream -> T -> unit
   (* print os x; send the usual string representation of x to
      the stream os. *)
```

```
(* OBSERVERS *)

  val fixedWidth: bool
   (* fixedWidth; is true if the usual string representation of type T uses
      a fixed number of characters for all values. *)

  val lt: T -> T -> bool
   (* lt x y; returns true if x is less than y; returns false otherwise. *)

end;
```

4.42 Outstream

```
(*$OUTSTREAM *)

signature OUTSTREAM =
sig

(* OUTPUT STREAMS

Created by: Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:         12 Nov 1989

Maintenance: Author


DESCRIPTION

   Functions on output streams.


SEE ALSO

   INSTREAM, STREAM_PAIR


NOTES

   All counterparts of functions in the initial basis have been given both
   the original names and names that fit the library conventions.



*)

(* PERVASIVES *)

  type outstream

  exception Io of string

  val std_out: outstream
  val output: outstream * string -> unit
  val open_out: string -> outstream
  val close_out: outstream -> unit
```

```
(* MANIPULATORS *)

  val stdOut: outstream
   (* stdOut = std_out *)

  val openOut: string -> outstream
   (* openOut = open_out *)

  val closeOut: outstream -> unit
   (* closeOut = close_out *)

  val write: outstream -> string -> unit
   (* write os s = curry output *)


(* SYSTEM *)

  exception NotImplemented of string
   (* NotImplemented fn; raised if fn is not provided. *)

  val openAppend: string -> outstream
   (* openAppend file; returns a new outstream whose consumer is the file s.
      Output to this stream is appended to s.  If s doesn't exist, it is
      created, initially empty. *)

  val flush: outstream -> unit
   (* flush os; ensures that all characters output on os have been or can
      be received by the associated consumer without being delayed by the
      implementation.  For example, characters output to a stream connected
      to a terminal will appear on that screen, even if they are normally
      buffered. *)

  val eof: outstream -> unit
   (* eof os: signal an end of stream on os without closing os. *)

  val interactive: outstream -> bool
   (* interactive os; returns true if os is associated with a terminal. *)
end
```

4.43 OutstreamType

```
(*$OUTSTREAM_TYPE: Outstream *)

signature OUTSTREAM_TYPE =

(* OUTSTREAM TYPE

Created by: Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:          21 Feb 1989

Maintenance: Author


DESCRIPTION

   The outstream type defined in Outstream and made available globally.



*)

sig

  (* PERVASIVES *)

  type outstream
    sharing type outstream = Outstream.outstream

  val std_out: outstream

  val open_out: string -> outstream

  val output: outstream * string -> unit

  val close_out: outstream -> unit
end
```

4.44 Pair

```
(*$PAIR *)

signature PAIR =
sig

(* PAIRS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 4 Oct 1989

Maintenance: Author


DESCRIPTION

   Functions on the built-in type ('a * 'b).

   The usual comparison operators are omitted, because they're all the
   same as binary (or binaryPair).  For example,
     fun lt Int.lt (x, y) (x', y') = binary Int.lt (x, y) (x', y').


SEE ALSO

   LIST_PAIR, PAIR_PARSE.




*)


(* CREATORS *)

  val create: 'a -> 'b -> ('a * 'b)
   (* create x y; yields (x, y). *)


(* CONVERTORS *)

  val string: ('a -> string) -> ('b -> string) -> ('a * 'b) -> string

  val print: outstream -> (outstream -> 'a -> unit) ->
     (outstream -> 'b -> unit) -> ('a * 'b) -> unit
```

```
(* MANIPULATORS *)

  val swap: ('a * 'b) -> ('b * 'a)
   (* swap (x, y); returns (y, x). *)

  val apply: ('a -> 'b) -> ('a * 'a) -> ('b * 'b)
   (* apply f (x,y); yields (f x, f y). *)

  val applyPair: ('a -> 'b) * ('c -> 'd) -> ('a * 'c) -> ('b * 'd)
   (* applyPair (f, g) (x, y); yields (f x, g y). *)

  val binary: ('a -> 'b -> 'c) -> ('a * 'a) -> ('b * 'b) -> ('c * 'c)
   (* binary f (x, y) (x', y'); yields (f x x', f y y'). *)

  val binaryPair: ('a -> 'b -> 'c) * ('d -> 'e -> 'f) ->
  ('a * 'd) -> ('b * 'e) -> ('c * 'f)
   (* binaryPair (f, g) (x, y) (x', y'); yields (f x x', g y y'). *)

  val tee: ('a -> 'b) * ('a -> 'c) -> 'a -> ('b * 'c)
   (* tee f (x, y); yields (f x, f y). *)

end
```

4.45 PairParse

```
(*$PAIR_PARSE: InstreamType GeneralTypes *)

signature PAIR_PARSE =
sig

(* PARSE FUNCTIONS FOR PAIRS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 8 Feb 1991

Maintenance: Author


DESCRIPTION

   Parse and read functions on the built-in type ('a * 'b).


NOTES

   These functions were originally in the main PAIR signature.


SEE ALSO

   PAIR.


*)


(* CONVERTORS *)

  val parse: (string -> ('a * string, 'c Option * string) Result) ->
        (string -> ('b * string, 'd Option * string) Result) ->
     string -> (('a * 'b) * string, 'e Option * string) Result
   (* parse p1 p2 s; parses a pair from the beginning of s, using p1 and p2
      to parse the two elements. *)

  val read: (instream -> ('a, 'c) Result) ->
```

```
    (instream -> ('b, 'd) Result) ->
    instream -> ('a * 'b, unit) Result
   (* read p1 p2 i; reads a pair from i, using p1 and p2 to parse the
      two elements. *)

end
```

4.46 PARSE

```
(*$PARSE: InstreamType GeneralTypes *)

signature PARSE =
sig

(* PARSE AND READ FUNCTIONS FOR SIMPLE TYPES

Created by: Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:          4 Feb 1991

Maintenance: Author


DESCRIPTION

   This signature defines a type T and functions for reading and parsing
   values of that type.


SEE ALSO

   OBJECT, SEQ_PARSE, MONO_SEQ_PARSE.




*)


(* TYPES *)

  type T


(* CONVERTERS *)

  val parse:  string -> (T * string, T Option * string) Result
   (* parse s; parses value of type T from the beginning of s. *)

  val read: instream -> (T, T Option) Result
   (* read i; reads value of type T from i. *)


(* OBSERVERS *)
```

```
  val fixedWidth: bool
   (* fixedWidth; is true if the usual string representation of type T uses
      a fixed number of characters for all values. *)

end
```

4.47 PRINT

```
(*$PRINT *)

signature PRINT =
sig

(* A TYPE WITH A PRINT FUNCTION

Created by:     Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           5 Feb 1991

Maintenance:    Author


DESCRIPTION

   This signature defines a type T and a function to produce a string
   representation of a value of that type.


SEE ALSO

   EQUALITY, ORDERING, EQ_PRINT, EQTYPE_PRINT, ORD_PRINT, OBJECT.




*)


(* TYPES *)

  type T


(* CONVERTERS *)

  val string: T -> string
   (* string x; retruns the usual string representation of x. *)

  val print: outstream -> T -> unit
   (* print os x; send the usual string representation of x to
      the stream os. *)
```

```
(* OBSERVERS *)

  val fixedWidth: bool
   (* fixedWidth; is true if the usual string representation of type T uses
      a fixed number of characters for all values. *)
end;
```

4.48 Real

```
(*$REAL *)

signature REAL =
sig

(* REALS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 8 Nov 1989

Maintenance: Author


DESCRIPTION

   Standard functions on the built-in type "real".

   This signature is based on the draft ISO/ANSI Language Compatible
   Arithmetic Standard, Version 2.2, ANSI X3T2/89-179,
   ISO/IEC JTC1.SC22/WG11 N144


NOTES

   The arithmetic exceptions are defined to raise Overflow for all
   overflow operations and Div for attempts to divide by zero.

   The "trunc" function is intended for use when printing reals, but this might
   be an inefficient approach.  It does reduce the number of converters.

   Possible extra functions (from the Hope+ library):
   arccos, arccosh, arcsin, arcsinh, arctanh, cosh, sinh, tan, tanh, log10,
   log2, exp10, exp2


*)

(* PERVASIVES *)

  eqtype real

  exception Overflow
  and Div
```

```
  val + : real * real -> real
  val - : real * real -> real
  val * : real * real -> real
  val / : real * real -> real
  val ~ : real -> real
  val abs: real -> real
  val floor: real -> int
  val sin: real -> real
  val cos: real -> real
  val arctan: real -> real
  val exp: real -> real

  exception Ln
  val ln: real -> real

  exception Sqrt
  val sqrt: real -> real


(* SYSTEM *)

  val radix: int
   (* radix; the "base" of the implementation. *)

  val precBits: int Option
   (* precBits; the number of radix digits provided by the implementation.
      This is None if the system supports arbitrary precision real numbers. *)

  val minExp: int Option
   (* minExp; the smallest exponent value, in terms of the radix. *)

  val maxExp: int Option
   (* maxExp; the largest exponent value, in terms of the radix. *)

  val denorm: bool
   (* denorm; true if the implementation support denormalised values,
      false otherwise. *)

  val minReal: real Option
   (* minReal; the smallest real that can be stored on the system, or None
      if the system supports arbitrary length reals. *)

  val minNormReal: real Option
   (* minNormReal; the smallest normalised real that can be stored on the
      system, or None if the system supports arbitrary length reals.
      If denorm = false then minNormReal = minReal. *)
```

```
  val maxReal: real Option
   (* maxReal; the largest real that can be stored on the system, or None
      if the system supports arbitrary length reals. *)

  val precReal: int Option
   (* precReal; the largest (decimal) precision that a real can have on the
      system, or None if the system supports reals with arbitrary precision. *)

  val epsilon: real Option
   (* epsilon; the largest relative representation error for the set of
      normalised values provided by the implementation. *)

(* CONSTANTS *)

  val pi: real
   (* pi = 3.14159265 *)

  val e: real
   (* e = 2.71828183 *)


(* TYPES *)

  eqtype T
    sharing type T = real


(* OBSERVERS *)

  val lt: real -> real -> bool

  val le: real -> real -> bool

  val gt: real -> real -> bool

  val ge: real -> real -> bool

  val eq: real -> real -> bool

  val ne: real -> real -> bool

  val fixedWidth: bool
   (* fixedWidth = false *)


(* CONVERTERS *)
```

```
val string: real -> string
 (* string n; returns the string representation of n, in the most convenient
    form.  The result must be a real constant as defined in The Definition Of
    Standard ML. *)

val stringNoE: real -> string
(* stringNoE n; returns the string representation of n, without exponent.
   The results alway contains a decimal point and at least one digit after
   the decimal point. *)

val stringE: real -> string
(* stringE n; returns the string representation of n, with exponent.  If all
   digits after the decimal point are 0, they are omitted, as is the decimal
   point itself.  0.0 is printed "0.0". *)

val stringPadE: int -> real -> string
(* stringPadE w n; as stringE n except that the exponent must contain
   at least w characters. *)

val print: outstream -> real -> unit
 (* print os n; sends the string representation of n to the stream os,
    in the most convenient form.  The output must be a real constant as
    defined in The Definition Of Standard ML. *)

val printNoE: outstream -> real -> unit
(* printNoE os n; sends the string representation of n to the stream os,
   without exponent.  The results alway contains a decimal point and
   at least one digit after the decimal point. *)

val printE: outstream -> real -> unit
(* printE os n; sends the string representation of n to the stream os,
   with exponent.  If all digits after the decimal point are 0, they are
   omitted, as is the decimal point itself.  0.0 is printed "0E0". *)

val printPadE: outstream -> int -> real -> unit
(* printPadE os w n; as printE os n except that the exponent must contain
   at least w characters. *)

val trunc: int -> real -> real
(* trunc p n; returns n with (decimal) precision p.  If p is greater than
   the existing precision, trunc has no effect. *)

val round: real -> int
(* round n; returns the nearest integer to n, halves rounded up. *)

val int: real -> int
```

```
  (* int n; returns n truncated to an integer, truncated towards 0. *)

  val ceiling: real -> int
  (* ceiling n; returns n rounded up to the next integer. *)


(* MANIPULATORS *)

  val max: real -> real -> real
   (* max x y; returns the greater of x and y.  *)

  val min: real -> real -> real
   (* min x y; returns the lesser of x and y.  *)

  val maxMin: real -> real -> real * real
   (* maxMin x y = (max (x, y), min (x, y)).  *)

  val sinCos: real -> real * real
   (* sinCos x = (sin x, cos x). *)

  (* infix 8 ** *)
  exception Power of real * int
  val ** : real * int -> real
   (* x ** y; x raised to the power y. *)
end
```

4.49 Ref

```
(*$REF *)

signature REF =
sig

(* REFERENCES

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date:         3 Oct 1989

Maintenance: Author


DESCRIPTION

   Standard functions on the built-in type "ref".

   The usual comparison operators are omitted because it's just as easy to
   write (X.lt (!x) (!y)) as (lt X.lt x y), and for numbers it's even easier
   to write (!x < !y).

   The usual conversion functions are also omitted, partly because they
   won't be needed very often and partly because there's no way to preserve
   sharing.


*)


(* PERVASIVES *)

  (* imperative type variables here are a temporary hack for Poly *)
  val ! : '_a ref -> '_a
  val := : 'a ref * 'a -> unit


(* TYPES *)

  eqtype 'a T
    sharing type T = ref


(* OBSERVERS *)
```

```
  val eq: 'a ref -> 'a ref -> bool

  val ne: 'a ref -> 'a ref -> bool


(* MANIPULATORS *)

  val inc: int ref -> unit
   (* inc r; increment the contents of r. *)

  val dec: int ref -> unit
   (* dec r; decrement the contents of r. *)

  val mkRandom: int -> int -> int
   (* mkRandom seed; Given a seed, mkRandom returns a psuedo-random number
      generator which takes an integer argument of one more than the
      maximum return value required if it is positive, or one less if it
      is negative.  *)
end
```

4.50 SEQUENCE

```
(*$SEQUENCE *)

signature SEQUENCE =

(* GENERAL POLYMORPHIC CONSTANT SEQUENCES

Created by: Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           21 Feb 1989

Maintenance: Author


DESCRIPTION

   These functions form a general interface for one-dimensional constant
   sequences, such as lists and vectors.

   The stringSep function takes arbitrary strings as its start, finish
   and separating symbols.  The other functions are more limited.


SEE ALSO

   OBJECT, SEQ_ORD, SEQ_PARSE.




*)

sig

(* TYPES *)

  type 'a T


(* OBSERVERS *)

  val eq: ('a -> 'a -> bool) -> 'a T -> 'a T -> bool
   (* eq p x y; returns true if (size x = size y) and for all i,
      0 <= i < size x, (p (x sub i) (y sub i)). *)

  val ne: ('a -> 'a -> bool) -> 'a T -> 'a T -> bool
```

```
  (* ne p x y; returns true if (size x <> size y) or there exists
     an i such that 0 <= i < size x and (p (x sub i) (y sub i)). *)


(* CONVERTORS *)

 val stringSep: string -> string -> string ->
 ('a -> string) -> 'a T -> string
  (* stringSep start finish sep p s; returns the string representation of s,
     beginning with start, ending with finish, and with the elements
     separated by sep. *)

 val string: ('a -> string) -> 'a T -> string
  (* string p l; returns the canonical string representation of l. *)

 val printSep: outstream -> string -> string -> string ->
               (outstream -> 'a -> unit) -> 'a T -> unit
  (* printSep os start finish sep p l; sends the string representation of l
     to the stream os, beginning with start, ending with finish, and with
     the elements separated by sep. *)

 val print: outstream -> (outstream -> 'a -> unit) -> 'a T -> unit
  (* print os p l; sends the canonical string representation of l to
     the stream os. *)

end
```

4.51 SEQ_ORD

```
(*$SEQ_ORD *)

signature SEQ_ORD =
sig

(* A TYPE THAT TAKES ONE PARAMETER, WITH AN ORDERING FUNCTION

Created by:     Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           10 Feb 1991

Maintenance:    Author


DESCRIPTION

   This signature defines a type 'a T and an ordering function.


SEE ALSO

   SEQUENCE, ORDERING.


*)


(* TYPES *)

  type 'a T


(* OBSERVERS *)

  val lt: ('a -> 'a -> bool) -> 'a T -> 'a T -> bool
   (* lt p x y; returns true if x is less than y, using p to compare elements
      when necessary. *)

end;
```

4.52 SEQ_PARSE

```
(*$SEQ_PARSE: GeneralTypes InstreamType *)

signature SEQ_PARSE =

(* PARSE AND READ FUNCTIONS FOR TYPES WITH ONE PARAMETER

Created by: Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           7 Feb 1991

Maintenance: Author


DESCRIPTION

   These functions form a general interface for reading and parsing
   one-dimensional constant sequences, such as lists and vectors.

   The conversion functions include the simple parse, read and string
   functions of the PARSE signature.  In addition there are versions
   that allow you to specify the start, finish and separating symbols.
   This lets yopu parse simple text layouts.  To parse something that is
   too complicated to be handled by the functions given here, try using
   ML-YACC and ML-LEX.

   The start and finish symbols must be single non-blank characters or
   empty strings.  If they are not empty, then the input must contain
   the appropriate characters.  If they are empty they are ignored.

   The separating symbol may be a non-blank character, an empty string,
   or a blank.  If it is a blank, then the elements of the sequence may
   be separated by zero or more formatting characters.  If it is a single
   non-blank character, then elements must be separated by that character
   plus an arbitrary amount of whitespace.  If the separating symbol is
   the empty string, then nothing seperates elements - for example, if
   the elements are bytes then formatting characters will be read as the
   appropriate byte values.

   Other versions of the functions let you specify the length of the
   sequence to be read.  This is especially useful when there is no
   finish symbol or when the finish symbol is the same as the separating
   symbol.


NOTES
```

These functions were originally in the main SEQUENCE signature.

SEE ALSO

SEQUENCE, PARSE, MONO_SEQ_PARSE.

```
*)

sig

(* TYPES *)

  type 'a T


(* CONVERTORS *)

  exception Sep of string * string * string * string
   (* Sep (fn, start, finish, sep); raised if the function named fn is called
      with arguments start, finish and sep and one or more of these does not
      meet the requirements given above. *)

  exception Size of string * int
   (* Size (fn, i); raised when the function fn is invoked
      with a negative size. *)

  val parseSepN: string -> string -> string ->
 (string -> ('a * string, 'b) Result) -> Nat ->
 string -> ('a T * string, 'a T Option * string) Result
   (* parseSepN start finish sep p n s; reads a sequence of 'a of length n
      that begins with start, ends with finish, in which the elements are
      separated by sep, and which forms a prefix of s, if there is one.
      Raises (Sep ("parseSep", start, finish, sep)) unless start, finish and
      sep fit the requirements listed above. *)

  val parseSep: string -> string -> string ->
                (string -> ('a * string, 'b) Result) ->
                string -> ('a T * string, 'a T Option * string) Result
   (* parseSep start finish sep p s; reads a sequence of 'a that begins with
      start, ends with finish, in which the elements are separated by sep,
      and which forms a prefix of s, if there is one.
      Raises (Sep ("parseSep'", start, finish, sep))
```

```
   unless start, finish and sep fit the requirements listed above. *)

val parse: (string -> ('a * string, 'b) Result) ->
           string -> ('a T * string, 'a T Option * string) Result
 (* parse p s; reads a list of 'a that forms a prefix of s, if there
    is one, using default start, finish and separation symbols. *)

val parseN: (string -> ('a * string, 'b) Result) ->
            Nat -> string -> ('a T * string, 'a T Option * string) Result
 (* parse p n s; reads a list of 'a of length n that forms a prefix of s,
    if there is one, using default start, finish and separation symbols. *)

val readSep: string -> string -> string ->
             (instream -> ('a, 'b) Result) -> instream ->
             ('a T, 'a T Option) Result
 (* readSep start finish sep p i; reads a sequence of 'a that begins with
    start, ends with finish, in which the elements are separated by sep,
    from i, if it begins with one.
    Raises (Sep ("readSep", start, finish, sep))
    unless start, finish and sep fit the requirements listed above. *)

val readSepN: string -> string -> string ->
             (instream -> ('a, 'b) Result) -> Nat -> instream ->
             ('a T, 'a T Option) Result
 (* readSepN start finish sep p i; reads a sequence of 'a of length n that
    begins with start, ends with finish, in which the elements are separated
    by sep, from i, if it begins with one.
    Raises (Sep ("readSep", start, finish, sep))
    unless start, finish and sep fit the requirements listed above. *)

val read: (instream -> ('a, 'b) Result) -> instream ->
          ('a T, 'a T Option) Result
 (* read p i; reads a list of 'a from i, if it begins with one. *)

val readN: (instream -> ('a, 'b) Result) -> Nat -> instream ->
           ('a T, 'a T Option) Result
 (* read p n i; reads a list of 'a of length n from i, if it begins
    with one. *)

val fromFile: (instream -> ('a, 'b) Result) -> string -> 'a T
 (* fromFile p name; read the contents of the file called name into a
    sequence.  Stops reading from the file as soon as p returns Fail.
    Raises Instream.Io if something goes wrong. *)

val file: ('a -> string) -> 'a T -> string -> unit
 (* file p v name; write the contents of v to the new file called name.
    Raises Outstream.Io if something goes wrong. *)
```

```
end
```

4.53 Set

```
(*$SET *)

signature SET =
sig

(* POLYMORPHIC SETS

Created by:     Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           22 Jan 1991

Maintenance:    Author


DESCRIPTION

   This provides functions over polymorphic sets.  Several functions
   have to take an equality function.


NOTES

   An alternative implementation would be to give the equality function
   to only those functions that create sets - empty (which would be a
   function instead of a value), fromList and singleton.  This would
   cause ambiguities when two sets of the same type were created with
   different equality functions.

   EQ_SET and MONO_SET have simpler interfaces than SET, and are often
   preferable.


SEE ALSO

   MONO_SET, EQ_SET.




*)

(* TYPES *)

  type 'a Set
```

```
(* CONSTANTS *)

  val empty: 'a Set
   (* empty; the empty set. *)


(* CREATORS *)

  val singleton: 'a -> 'a Set
   (* singleton x; returns the set containing only x. *)


(* CONVERTERS *)

  val list: 'a Set -> 'a list
   (* list s; return a list of the elements of s. *)

  val fromList: ('a -> 'a -> bool) -> 'a list -> 'a Set
   (* fromList elemEq l; return the set of elements of l, removing
      duplicates. *)


(* OBSERVERS *)

  val size: 'a Set -> int
   (* size s; the number of elements in s. *)

  val isEmpty: 'a Set -> bool
   (* isEmpty s; returns true if s is empty, false otherwise. *)

  val member: ('a -> 'a -> bool) -> 'a -> 'a Set -> bool
   (* member elemEq x s; returns true is x is in s, false otherwise. *)

  val eq: ('a -> 'a -> bool) -> 'a Set -> 'a Set -> bool
   (* eq elemEq s s'; returns true if s and s' have the same elements. *)


(* SELECTORS *)

  exception Empty of string
   (* Empty fn; raised if the function named fn is erronously applied to
      the empty set. *)

  val select: 'a Set -> ('a * 'a Set)
   (* select s; returns a pair consiting of an element of s and the set
      of the remaining elements. *)
```

```
(* MANIPULATORS *)

  val difference: ('a -> 'a -> bool) -> 'a Set -> 'a Set -> 'a Set
   (* difference elemEq s s'; returns the set of those elements of s that
      aren't also in s'.  *)

  val insert: ('a -> 'a -> bool) -> 'a -> 'a Set -> 'a Set
   (* insert elemEq x s; returns the union of s and {x}. *)

  val intersect: ('a -> 'a -> bool) -> 'a Set -> 'a Set -> 'a Set
   (* intersect elemEq s s'; returns the set of those elements that are in
      both s and s'. *)

  val remove: ('a -> 'a -> bool) -> 'a -> 'a Set -> 'a Set
   (* remove elemEq x s; returns the set of the elements of s with x
      removed. *)

  val partition: ('a -> bool) -> 'a Set -> ('a Set * 'a Set)
   (* partition p s; returns a pair of sets; the first containing the elements
      of s for which the predicate p is true, the second the elements of s
      for which p is false. *)

  val union: ('a -> 'a -> bool) -> 'a Set -> 'a Set -> 'a Set
   (* union elemEq s s'; returns the set of elements that are in either or
      both s and s'. *)

  val closure: ('a -> 'a -> bool) -> ('a -> 'a Set) -> 'a Set -> 'a Set
   (* closure elemEq f s; repeatedly applies f to elements of s and the
      elements of the results of such applications, until no further elements
      are generated. *)
end;
```

4.54 StreamPair

```
(*$STREAM_PAIR: GeneralTypes InstreamType OutstreamType *)

signature STREAM_PAIR =
sig

(* PAIRS OF ONE INPUT STREAM AND ONE OUTPUT STREAM

Created by: Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:         12 Nov 1989

Maintenance: Author


DESCRIPTION

   A streampair is used to communicate with the user or with another
   process.  This package includes system functions to start other
   process and functions to ask the user for certain input.  It also
   provides most of the functions on outstreams and instreams, with
   the obvious inherited effect.


SEE ALSO

   INSTREAM, OUTSTREAM


NOTES

   System functions raise General.NotImplemented if the implementation
   doesn't support them.

   Possibly this module should provide functions for random access files.
   Alternatively such files could be viewed as (Ref)Vectors.

   Possibly the functions in this module should handle the Io exception
   and raise a local exception instead.


*)

(* TYPES *)
```

```
  type StreamPair

(* CONSTANTS *)

  val std: StreamPair
   (* std = (stdIn, stdOut); *)

(* CREATORS *)

  val create: instream * outstream -> StreamPair
   (* create (i, os); create a StreamPair from two streams. *)

(* CONVERTORS *)

  val streams: StreamPair -> instream * outstream
   (* streams io; split a StreamPair into two streams. *)

(* SYSTEM *)

  val openTemporary: unit -> StreamPair
   (* openTemporary (); returns a pair of streams that are connected to
      each other. *)

  val canInput: StreamPair -> Nat -> bool
   (* canInput io n; returns true if n characters can be read from io
      without blocking. *)

  val flush: StreamPair -> unit
   (* flush io; ensures that all characters output on io have been or can
      be received by the associated consumer without being delayed by the
      implementation.  For example, characters output to a stream connected
      to a terminal will appear on that screen, even if they are normally
      buffered. *)

  val reset: StreamPair -> bool
   (* reset io; if io can be reset to the beginning, in some sense,
     this is done and true is returned.  Otherwise false is returned. *)

  val interactive: StreamPair -> bool
   (* interactive io; returns true if io is associated with a terminal. *)

  val execute: string -> (StreamPair, string) Result
   (* execute proc; creates a pair of streams connected to a system process.
```

```
     In the case that the process is an SML, C/UNIX or Pascal program, the
     instream is connected to the stdOut stream, stdout stream or output
     file of the process and the outstream is connected to the process's
     stdIn stream, stdin stream or input file, respectively. *)

  val read: StreamPair -> Nat -> string
   (* input io s; reads n characters from io, if io has not been closed.
     Has the same behaviour as Instream.read. *)


(* MANIPULATORS *)

  val openPair: string -> (StreamPair, string) Result
   (* openPair s; returns a pair of streams whose producer and consumer are
     associated with the file named s.  It must be possible to open s
     for both reading and writing.  If s does not exist, it is created
     if this is possible. *)

  val closePair: StreamPair -> unit
   (* closePair io; terminates the streams io. *)

  val output: StreamPair * string -> unit
   (* output (io, s); writes the characters in s to io, if io has not been
     closed.  Raises (General.Io ("output", file, "")) in case of failure. *)

  val write: StreamPair -> string -> unit
   (* write = General.curry output *)

  val input: StreamPair * Nat -> string
   (* input (io, s); reads n characters from io, if io has not been closed.
     Has the same behaviour as Instream.input. *)

  val input1: StreamPair -> string
   (* input1 io; reads 1 charcater from io, if io has not been closed.
     Has the same behaviour as Instream.input1. *)

  val lookahead: StreamPair -> string
   (* lookahead io; returns a single character from io.  Has the same behaviour
     as the pervasive lookahead if there isn't a character available. *)

  val eof: StreamPair -> bool
   (* eof io; returns true if the last attempt to lookahead or input on the
     input stream of io returned the empty string, false otherwise. *)

  val readString: StreamPair -> string -> (unit, string) Result
   (* readString io s; returns true if reading from io gives the characters
     in s.  Returns false as soon as a character is read that doesn't
```

```
    match the corresponding one in s.  In either case all characters read
    from i are lost.  Raises (EOF "readString") if the end of file is
    reached before all the charcaters in s have been read. *)

val skip: (string -> bool) -> StreamPair -> unit
 (* skip p io; reads all characters from io that satisfy p.  Leaves the first
    character that doesn't satisfy p to be read.  Raises (EOF "skip") if
    the end of file is reached before a character is found that doesn't
    satisfy p. *)

val inputLine: StreamPair -> string
 (* inputLine io; returns a string consisting of characters read from io
    up to and including the next end of line character.  If the end of the
    file is reached first, all characters up to the end of file are returned
    (without a new line character). *)

val prompt: StreamPair -> string -> string
 (* prompt io s; writes s on io and reads an answer input line using
    inputLine. *)

val ask: StreamPair -> string -> (instream -> ('a, 'a Option) Result) -> 'a
 (* ask io s p; repeatedly prompts for an answer using s as the prompt;
    reads an answer using p until p succeeds; returns the value read by p. *)

val confirm: StreamPair -> string -> bool
 (* confirm s io; checks for confirmation with message s.  Returns true or
    false depending on the input.  If io is connected to an ASCII terminal,
    true might be indicated by typing "y" and false by typing "n", with
    other characters being ignored. *)

val menu: StreamPair -> string -> string list -> int
 (* menu io title entries; writes a menu to io using title as a title
    and the elements of entries as entries.  Returns an integer that
    corresponds to the position of the chosen element in the list
    (0 to (size entries - 1)). *)
end
```

4.55 String

```
(*$STRING: GeneralTypes *)

signature STRING =
sig

(* ASCII STRINGS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date:           4 Oct 1989

Maintenance: Author


DESCRIPTION

   Standard functions on the built-in type "string".

   Functions such as  index  and  search  take an integer
   offset.  This is because it's more efficient to index into
   a string than to take a substring.


NOTES

   Possibly the integer offset parameters and the revXXX constructs should
   be dropped, relying on the compiler to optimise away intermediate string
   constructions.

   Possibly there should be a dropExtract function to delete a range
   of characters from a string.


*)


(* PERVASIVES *)

  eqtype string

  val size: string -> int
  exception Ord
  val ord: string -> int
```

```
  exception Chr
  val chr: int -> string
  val explode: string -> string list
  val implode: string list -> string
  val ^ : string * string -> string


(* TYPES *)

  eqtype T
    sharing type T = string

  datatype Mode = IgnoreCase | MatchCase
   (* Search and replace functions can either ignore the case of letters
      or match letters only if they have the same case.  If replace functions
      ignore the case when matching, they substitute with the case of the
      matched characters. *)


(* CREATORS *)

  exception Size of string * int
   (* Size (fn, i); raised by the create function when it is invoked
      with a negative size. *)

  val create: Nat -> string -> string
   (* create n s; returns a string composed of n copies of s.
      Raises (Size ("create", n)) if n < 0. *)


(* CONVERTERS *)

  val string: string -> string
   (* string s; returns a version of s with all control characters
      expanded to the SML source form, e.g. if s contains a single
      newline character then (string s) consists of four characters, a
      double quote, a backslash, the letter n and another double quote. *)

  val print: outstream -> string -> unit
   (* print os s; sends a version of s to the stream os, with all control
      characters expanded to the SML source form, e.g. if s contains a
      single newline character then (print os s) outputs four characters,
      a double quote, a backslash, the letter n and another double quote. *)


(* OBSERVERS *)
```

```
val exists: Mode -> (string -> bool) -> string -> bool
 (* exists m p s; true if there exists a character in s satisfying p,
    ignoring case if m is IgnoreCase, and matching case if m is MatchCase. *)

val forAll: Mode -> (string -> bool) -> string -> bool
 (* forAll m p s; true if every character in s satisfies p, ignoring case
    if m is IgnoreCase, and matching case if m is MatchCase. *)

val eqMode: Mode -> string -> string -> bool
 (* eq m s s'; returns true if (s = s'). Returns false otherwise.
    Ignores case if m is IgnoreCase, and matches case if m is MatchCase. *)

val neMode: Mode -> string -> string -> bool
 (* ne m s s'; returns true if (s <> s'). Returns false otherwise.
    Ignores case if m is IgnoreCase, and matches case if m is MatchCase. *)

val eq: string -> string -> bool
 (* eq s s' = eqMode MatchCase. This function is included to match the
    generic signatures, such as EQ_PRINT. *)

val ne: string -> string -> bool
 (* ne m s s' = neMode MatchCase. This function is included to match the
    generic signatures, such as EQ_PRINT. *)

val fixedWidth: bool
 (* fixedWidth = false *)

exception Subscript of string * int
 (* Subscript (fn, n); raised if the function named fn is called with
    an out of range argument n. *)

val prefixes: Mode -> string -> string -> int -> bool
 (* prefixes m s1 s2 n; returns true if s1 is a prefix of the substring of
    s2 starting at (s2 sub n), ignoring case if m is IgnoreCase, and
    matching case if m is MatchCase.  Raises (Subscript ("prefixes", n))
    if not (0 <= n < size s2). *)

val postfixes: Mode -> string -> string -> int -> bool
 (* postfixes m s1 s2 n; returns true if s1 is a postfix of the substring of
    s2 ending at (s2 sub (n - 1)), ignoring case if m is IgnoreCase, and
    matching case if m is MatchCase.  Raises (Subscript ("postfixes", n))
    if not (0 <= n <= size s2). *)

val index: Mode -> (string -> bool) -> string -> int -> (int, unit) Result
 (* index m p s n; returns the position in s of the first character
    after (s sub n) (inclusive) that satisfies p ignoring case if m is
    IgnoreCase, and matching case if m is MatchCase.
```

```
    Raises (Subscript ("index", n)) if not (0 <= n < size s2). *)

  val revIndex: Mode -> (string -> bool) -> string -> int -> (int, unit) Result
   (* revIndex m p s n; returns the position in s of the last character
      before (s sub n) (exclusive) that satisfies p, ignoring case if m is
      IgnoreCase, and matching case if m is MatchCase.
      Raises (Subscript ("revIndex", n)) if not (0 <= n <= size s2). *)

  val search: Mode -> string -> string -> int -> (int, unit) Result
   (* search m s' s n; find the first occurrence of s' in s after (s sub n)
      (inclusive), ignoring case if m is IgnoreCase, and matching case if m
      is MatchCase.  Returns the index of the first character in that
      substring.  Raises (Subscript ("search", n)) if not (0 <= n < size s). *)

  val revSearch: Mode -> string -> string -> int -> (int, unit) Result
   (* revSearch m s' s n; find the last occurrence of s' in s before (s sub n)
      (exclusive),  ignoring case if m is IgnoreCase, and matching case if m is
      MatchCase.  Returns the index of the first character in that substring.
      Raises (Subscript ("revSearch", n)) if not (0 <= n <= size s). *)

  val occurs: Mode -> string -> string -> int -> bool
   (* occurs m s' s n; return true if s' occurs in s after (s sub n)
      (inclusive), ignoring case if m is IgnoreCase, and matching case if m is
      MatchCase.  Raises (Subscript ("occurs", n)) if not (0 <= n < size s). *)

  val revOccurs: Mode -> string -> string -> int -> bool
   (* revOccurs m s' s n; return true if s' occurs in s before (s sub n)
      (inclusive), ignoring case if m is IgnoreCase, and matching case if m
      is MatchCase.  Raises (Subscript ("revOccurs", n)) if
      not (0 <= n <= size s). *)


(* MANIPULATING STRING TYPE *)

  exception Empty of string
   (* Empty fn; raised if the function called fn is erroneously applied
      to the empty string. *)

  val upper: string -> string
   (* upper s; if the first character in s is a lower case letter,
      returns s with that letter in upper case.  Otherwise leaves s unchanged.
      Raises (Empty "upper") if s is empty. *)

  val lower: string -> string
   (* lower s; if the first character in s is an upper case letter,
      returns s with that letter in lower case.  Otherwise leaves s unchanged.
      Raises (Empty "lower") if s is empty. *)
```

```
val ascii: string -> string
 (* ascii s; if the first character in s is not an ascii character,
    returns s with the top bit stripped from that character.  Otherwise
    leaves s unchanged.  Raises (Empty "ascii") if s is empty. *)

val control: string -> string
 (* control s; if the first character in s is in the range @ .. _ , returns
    s with that character replaced with the corresponding control character.
    Otherwise leaves s unchanged.  Raises (Empty "ascii") if s is empty. *)


(* SELECTORS *)

  (* infix 9 sub *)
  val sub: string * int -> string
   (* s sub n; return the nth character of s.
      Raises (Subscript ("sub", n)) if not (0 <= n < size s). *)

  val nth: int -> string -> string
   (* nth n s; return the nth character of s.
      Raises (Subscript ("nth", n)) if not (0 <= n < size s). *)

  exception Char of string * string
   (* Char (fn, c); raised if the function called fn is called with a
      string c such that (size c <> 1).  The idea is that the update functions
      replace one character with another, rather than with an arbitrary length
      string.  *)

  exception Extract of int * int

  val extract: int -> int -> string -> string
   (* extract s start finish; 0 <= start,finish <= size (s); returns the
      substring starting with (s sub start) and ending with
      (s sub (finish - 1)).  Returns "" if (start = finish).  Raises
      (Extract (start, finish)) if not (0 <= start <= finish <= size s). *)


(* OTHER MANIPULATORS *)

  val skipSpaces: string -> string
   (* skipSpaces s; returns s with any leading invisible characters removed. *)

  val subst: Mode -> string -> string -> string -> string
   (* subst m c s' s; replaces all occurrences of c in s by s'.  If m is
      MatchCase, then c must exactly match the character to be replaced,
      and s' is substituted literally.  If m is IgnoreCase and c is a letter,
```

```
   then both upper case and lower case instances of c will be replaced,
   and all letters in the instances of s' that replace them are converted
   to the appropriate case.  Raises (Char ("subst", s')) if size c > 0. *)

val rev: string -> string
 (* rev s; returns the reflection of s. E.g. rev "abc" = "cba". *)

val showAscii: string -> string
 (* showAscii s; returns s with all characters expanded to the form they
    would have in a string literal.  E.g. showAscii "\n" = "\\n".
    This function differs from the string function only by not
    adding quotes at the beginning and end of the string and in having
    a more intuitive name. *)

val padL: string -> int -> string -> string
 (* padL c w s; pads the string s with character c on the
    left until its size is w.  Raises (Char ("padL", c)) if (size c <> 1). *)

val padR: string -> int -> string -> string
 (* padR c w s; pads the string s with character c on the
    right until its size is w.  Raises (Char ("padR", c)) if (size c <> 1). *)

val padC: string -> int -> string -> string
 (* padC c w s; pads the string s with character c to centre the existing
    satring in a new string of size w.
    Raises (Char ("padC", c)) if (size c <> 1). *)

val truncL: int -> string -> string
 (* truncL w s; truncates the string s by removing characters on the left
    until its size is w. *)

val truncR: int -> string -> string
 (* truncR w s; truncates the string s by removing characters on the right
    until its size is w. *)

val truncC: int -> string -> string
 (* truncC w s; truncates the string s by removing characters equally from
    both ends until its size is w. *)

val dropL: string -> string -> string
 (* dropL c s; removes occurrence of the character c on the left of s.
    Raises (Char ("dropL", c)) if (size c <> 1). *)

val dropR: string -> string -> string
 (* dropR c s; removes occurrence of the character c on the right of s.
    Raises (Char ("dropR", c)) if (size c <> 1). *)
```

```
(* ITERATORS *)

  val map: (string -> string) -> string -> string
   (* map f s; builds a string by applying f to each character in s. *)

  val apply: (string -> unit) -> string -> unit
   (* apply f s; applies f to each character in s. *)

  val mapAll: (string -> bool) -> (string -> string) -> string -> string
   (* map p f s; builds a string by applying f to each character in s
      that satisfies p. *)

  val applyAll: (string -> bool) -> (string -> unit) -> string -> unit
   (* applyAll p f s; applies f to each character in s that satisfies p. *)
end
```

4.56 StringListOps

```
(*$STRING_LIST_OPS *)

signature STRING_LIST_OPS =
sig

(* LIST-LIKE OPERATIONS ON ASCII STRINGS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date:          10 Feb 1991

Maintenance: Author


DESCRIPTION

   Functions on strings that parallel the functions defined on lists by
   the LIST signature.  They use an orthogonal naming scheme.  They can
   be grouped as follows:

   f;
     if f is "nth", return the nth character (counting from zero).
     if f is "first", return the first character that satisfies a predicate p.
     if f is "all", return the list of characters that satisfy p.
     if f is "prefix", return the prefix of characters that satisfy p.

   dropF ... s; s with (f s) removed.
   splitF ... s; a pair of strings:
 (characters before f s, characters from f s to the end of s).
   removeF ... s; a pair of (f s, dropF s).
   updateF ... s; l with (f s) replaced with a value.
   changeF ... s; l with (f s) replaced with (g (f s)).
   spliceF ... s; l with (f s) replaced with the elements in another string.
   insertF ... s; l with another string inserted before (f s).
   appendF ... s; l with another string inserted after (f s).

   splitAll, and splitPrefix are not provided because "all" and
   "prefix" return strings of arbitrary length.

   changePrefix, updatePrefix, insertPrefix, appendPrefix, and splicePrefix
   are not provided, because "prefix" returns a string of arbitrary length.

   The "nth" functions raise "Subscript" if not (0 <= n < size s).
   The "first" functions (except dropFirst) raise "First" if there aren't
        any elements that satisfy p.
```

NOTES

Possibly should add revFirst, dropRevFirst, ... and postfix, dropPostfix, ... sets of functions.

Possibly first, ... and prefix, ... should take an integer offset.

SEE ALSO

STRING, LIST.

NOTES

These functions used to be in the main STRING signature.
A few of these functions are duplicates of functions in the main STRING signature.

```
*)


(* MANIPULATING THE NTH ELEMENT *)

  (* infix 9 sub *)
  val sub: string * int -> string
   (* s sub n; return the nth character of s.
      Raises (Subscript ("sub", n)) if not (0 <= n < size s). *)

  val nth: int -> string -> string
   (* nth n s; return the nth character of s.
      Raises (Subscript ("nth", n)) if not (0 <= n < size s). *)

  val dropNth: int -> string -> string
   (* dropNth n s = returns s without (s sub n).
      Raises (Subscript ("dropNth", n)) if not (0 <= n < size s). *)

  val removeNth: int -> string -> (string * string)
   (* removeNth n s = (s sub n, dropNth n s).
      Raises (Subscript ("removeNth", n)) if not (0 <= n < size s). *)

  val splitNth: int -> string -> (string * string)
```

```
  (* splitNth n s; returns (extract 0 n s, extract n (size s) s).
     Raises (Subscript ("splitNth", n)) if not (0 <= n < size s). *)

 exception Char of string * string
  (* Char (fn, c); raised if the function called fn is called with a
     string c such that (size c <> 1).  The idea is that the update functions
     replace one character with another, rather than with an arbitrary length
     string.  *)

 val updateNth: int -> string -> string -> string
  (* updateNth n s' s; returns s with (s sub n) replaced by s'.
     Raises (Char ("updateNth", s')) if (size s' <> 1).
     Raises (Subscript ("updateNth", n)) if not (0 <= n < size s). *)

 val changeNth: int -> (string -> string) -> string -> string
  (* changeNth n f s; returns s with (s sub n) replaced by (f (s sub n)).
     Raises (Subscript ("changeNth", n)) if not (0 <= n < size s). *)

 val insertNth: int -> string -> string -> string
  (* insertNth n s' s; returns s with s' inserted before (s sub n).
     Raises (Subscript ("insertNth", n)) if not (0 <= n < size s). *)

 val appendNth: int -> string -> string -> string
  (* appendNth n s' s; returns s with s' appended after (s sub n).
     Raises (Subscript ("appendNth", n)) if not (0 <= n < size s). *)

 val spliceNth: int -> string -> string -> string
  (* spliceNth n s' s; returns s with (s sub n) replaced by s'.
     Unlike updateNth, s' may be any length.
     Raises (Subscript ("spliceNth", n)) if not (0 <= n < size s). *)


(* MANIPULATING THE FIRST ELEMENT THAT SATISFIES A PREDICATE *)

 exception First of string

 val first: (string -> bool) -> string -> string
  (* first p s; returns the first character in s satisfying p.
     Raises (First "first") if p doesn't hold for any character in s. *)

 val dropFirst: (string -> bool) -> string -> string
  (* dropFirst p s; returns s without the first of its characters (if any)
     that satisfy p. *)

 val removeFirst: (string -> bool) -> string -> (string * string)
  (* removeFirst p s = (first s, dropFirst s).
     Raises (First "removeFirst") if p doesn't hold for any character in s. *)
```

```
val splitFirst: (string -> bool) -> string -> (string * string)
 (* splitFirst p s; returns (extract 0 n s, extract n (size s - 1) s),
    where (s sub n) is the first character in s that satisfies p.
    Raises (First "splitFirst") if p doesn't hold for any character in s. *)

val updateFirst: (string -> bool) -> string -> string -> string
 (* updateFirst p s' s; returns s with (first p s) replaced by s'.
    Raises (First "updateFirst") if there is no (first p s).
    Raises (Char ("updateFirst", s')) if (size s' <> 1). *)

val changeFirst: (string -> bool) -> (string -> string) -> string -> string
 (* changeFirst p f s; returns s with (first p s) replaced by
    (f (first p s)).  Raises (First "changeFirst") if there is no
    (first p s).  *)

val insertFirst: (string -> bool) -> string -> string -> string
 (* insertFirst p s' s; returns s with s' inserted before
    (first p l).  Raises (First "insertFirst") if there is no (first p l). *)

val appendFirst: (string -> bool) -> string -> string -> string
 (* appendFirst p s' s; returns s with s' appended after
    (first p l).  Raises (First "insertFirst") if there is no (first p l). *)

val spliceFirst: (string -> bool) -> string -> string -> string
 (* spliceFirst p s' s; returns s with (first p s) replaced by s'.
    Raises (First "spliceFirst") if there is no (first p s).
    Unlike updateFirst, s' may be any length. *)


(* TAKING A PREFIX OF ELEMENTS THAT SATISFY A PREDICATE *)

val prefix: (string -> bool) -> string -> string
 (* getPrefix p s; returns the largest prefix of s in which each character
    satisfies p. *)

val dropPrefix: (string -> bool) -> string -> string
 (* dropPrefix p s; returns s without the largest prefix in which
    every character satisfies p. *)

val removePrefix: (string -> bool) -> string -> string * string
 (* removePrefix p s = (prefix p s, dropPrefix p s). *)


(* MANIPULATING ALL ELEMENTS THAT SATISFY A PREDICATE *)

val all: (string -> bool) -> string -> string
```

```
  (* all p s: returns a string formed from the characters in s that
     satisfy p. *)

 val dropAll: (string -> bool) -> string -> string
  (* dropAll p s: returns a string formed from the characters in s that don't
     satisfy p. *)

 val removeAll: (string -> bool) -> string -> (string * string)
  (* removeAll p s = (all p s, dropAll p s). *)

 val updateAll: (string -> bool) -> string -> string -> string
  (* updateAll p s' s; returns s with each character that satisfies p
     replaced by s'.  Raises (Char ("updateAll", s')) if (size s' <> 1). *)

 val changeAll: (string -> bool) -> (string -> string) -> string -> string
  (* changeAll p f s; returns s with each character c that satisfies p
     replaced by (f c). *)

 val insertAll: (string -> bool) -> string -> string -> string
  (* insertAll p s' s; returns s with s' inserted before each
     character in s that satisfies p. *)

 val appendAll: (string -> bool) -> string -> string -> string
  (* appendAll p s' s; returns s with s' appended after each
     character in s that satisfies p. *)

 val spliceAll: (string -> bool) -> string -> string -> string
  (* spliceAll p s' s; returns s with each character that satisfies p
     replaced by s'. *)

end
```

4.57 StringParse

```
(*$STRING_PARSE: InstreamType GeneralTypes *)

signature STRING_PARSE =
sig

(* STRING CONVERTERS

Created by: Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:         14 Feb 1990

Maintenance: Author


DESCRIPTION

   Standard conversion functions on the built-in type "string".
   These expect strings to be surrounded by quotes and with appropriate
   characters escaped.




*)

(* TYPES *)

  eqtype T
    sharing type T = string


(* CONVERTERS *)

  val parse:  string -> (string * string, string Option * string) Result

  val read: instream -> (string, string Option) Result

  val fromFile: string -> string
   (* fromFile name; read the contents of name into a string.
      Raises the pervasive exception Io if something goes wrong. *)

  val file: string -> string -> unit
   (* file name s; write s to the new file called name.
      Raises the pervasive exception Io if something goes wrong. *)
```

```
(* OBSERVERS *)

  val fixedWidth: bool
   (* fixedWidth = false. *)


(* MANIPULATORS *)

  val words: {groups: string, singles: string,
      preserveGroups: bool, preserveSingles: bool} ->
    string -> string list
   (* words {groups, singles, preserveGroups, preserveSingles} s;
      splits s into words.  Word boundaries are defined by each occurrence of
      a character in singles and by each consecutive sequence of characters
      in groups.  If preserveSingles or preserveGroups is true, then
      occurrences of the corresponding separator characters are included in
      the result.  If a character appears in both sep and singles, its
      presence in singles is ignored. *)

end
```

4.58 StringType

```
(*$STRING_TYPE *)

signature STRING_TYPE =
sig

(* CHARACTER CLASSES

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date:          4 Oct 1989

Maintenance: Author


DESCRIPTION

   Functions to find the types of the first character in a string.




*)


(* CONSTANTS *)

  val digits: string
   (* digits = "1234567890"; *)
  val hexes: string
   (* hexes = "1234567890abcdefABCDEF"; *)
  val formats: string
   (* formats; contains space, tab, newline, carriage return, backspace,
      and formfeed *)
  val uppers: string
   (* uppers = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; *)
  val lowers: string
   (* lowers = "abcdefghijklmnopqrstuvwxyz"; *)
  val letters: string
   (* letters = uppers ^ lowers; *)
  val alNums: string
   (* alNums = letters ^ digits; *)
  val ids: string
   (* ids = alNums ^ "_'"; *)
  val symbols: string
   (* symbols = "!%&$#+-/:<=>?@\\~`^|*"; *)
```

```
  val puncts: string
   (* puncts = symbols ^ "'_(),.;[]{}"; *)
  val visibles: string
   (* visibles = letters ^ digits ^ puncts *)
  val prints: string
   (* prints = visibles ^ formats *)
  val controls: string
   (* controls; the control characters (Ascii numbers 0 - 31 and 127) *)
  val asciis: string
   (* asciis = controls ^ visibles *)


(* OBSERVERS *)

  (* These functions test the first character of a string for membership
     of the above sets.  They raise Empty if the string is empty. *)

  exception Empty of string
   (* Empty fn; raised if the function called fn is applied to an empty
      string. *)

  val isDigit: string -> bool
  val isHex: string -> bool
  val isFormat: string -> bool
  val isPrint: string -> bool
  val isVisible: string -> bool
  val isLetter: string -> bool
  val isUpper: string -> bool
  val isLower: string -> bool
  val isPunct: string -> bool
  val isControl: string -> bool
  val isAlNum: string -> bool
  val isId: string -> bool
  val isSymbol: string -> bool
  val isAscii: string -> bool

end
```

4.59 System

```
(*$SYSTEM *)

signature SYSTEM =
sig

(* SYSTEM FUNCTIONS

Created by: Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk

Date:          14 Nov 1989

Maintenance: Author


DESCRIPTION

   Functions that interact with the operating system.

   These functions raise General.NotImplemented if the implementation doesn't
   provide them.  The values are extremely conservative; they should be
   replaced with the correct values in any implementation of ther library.



*)


(* ML SYSTEM *)

  val collect: unit -> unit
   (* collect (); an identity function that gives a hint to the system that
      now would be a good time to perform garbage collection.  This function
      never raises General.NotImplemented, since it can't affect the
      computation. *)

  val eq: 'a -> 'a -> bool
   (* eq x y; returns false if x and y are not located at the same hardware
      address.  May return either true or false otherwise. *)

  val hash: ''a -> int
   (* hash x; applies a has function to x and returns the result. *)

  val quit: unit -> unit (* never returns. *)
   (* quit (); exit ML. *)
```

```
(* INTERFACE TO OPERATING SYSTEM *)

  exception NoFile of string * string
   (* NoFile (fn, file); raised if the function called fn tries to access
      a file f that doesn't exist. *)

  exception Permission of string * string
   (* Permission (fn, file); raised if the function called fn tries to access
      a protected file f. *)

  val use: string -> unit
   (* use "f"; read f as if it was typed at top level.  This function is not
      defined unless it is called from top-level or from top-level in a file
      that is itself being read as a result of a call to "use".  The function
      should check for files that recursively "use" each other. *)

  val cd: string -> unit
   (* cd "d"; change working directory to d.  Raise (NoFile ("cd", "d"))
      if d doesn't exist.  Raise (Permission ("cd", "d")) if d exists
      but doesn't allow the user access.  *)

  val isDir: string -> bool
   (* isDir "d"; returns true if d is a directory. *)

  val pwd: unit -> string
   (* pwd (); return the full name of the working directory. *)

  val dir: string -> string list
   (* dir "d"; returns a list of the names of the files in directory d.
      Raise (NoFile ("dir", "d")) if d doesn't exist.
      Raise (Permission ("cd", "d")) if d exists and is unreadable. *)

  val delete: string -> unit
   (* delete "f"; delete the file f, if possible.
      Raise (NoFile ("delete", "f")) if f doesn't exist.
      Raise (Permission ("delete", "f")) if f exists but can't be deleted. *)

  val system: string -> string
   (* system "c"; run the system command c, with input from std_in, and
      return the output of c. *)

  val getenv: string -> string
   (* getenv var; return the string associated with var in the system
      environment. *)
end
```

4.60 User

```
(*$USER: InstreamType GeneralTypes *)

signature USER =
sig

(* USER INPUT FUNCTIONS

Created by: Dave Berry, LFCS, University of Edinburgh
                db@lfcs.ed.ac.uk
Date:           24 Mar 1990

Maintenance: Author


DESCRIPTION

   These functions prompt the user for some input, and return an
   appropriate value.


SEE ALSO

   STREAM_PAIR


NOTES

   When using an ASCII terminal, these routines are the same as those in
   StreamPair, but always use the standard streams.

   A window system should provide an implementation of this signature that
   uses real menus and dialog boxes.



*)

(* MANIPULATORS *)

  val prompt: string -> string
   (* prompt s; displays s to the user and reads an answer input line. *)

  val ask: string -> (instream -> ('a, 'a Option) Result) -> 'a
   (* ask s p; repeatedly prompts for an answer using s as the prompt;
```

```
    reads an answer using p until p succeeds; returns the value read by p. *)

  val confirm: string -> bool
   (* confirm s; checks for confirmation with message s.  Returns true or
      false depending on the input.  When using an ASCII terminal,
      true might be indicated by typing "y" and false by typing "n", with
      other characters being ignored. *)

  val menu: string -> string list -> int
   (* menu title entries; displays a menu using title as a title
      and the elements of entries as entries.  Returns an integer that
      corresponds to the position of the chosen element in the list
      (0 to (size entries - 1)). *)
end
```

4.61 Vector

```
(*$VECTOR *)

signature VECTOR =
sig

(* CONSTANT VECTORS

Created by: Dave Berry, LFCS, University of Edinburgh
db@lfcs.ed.ac.uk
Date: 31 Oct 1989

Maintenance: Author


DESCRIPTION

   A Vector is a single-dimensional constant polymorphic array.  The first
   element has the index 0.  Vectors are equal if they contain the same
   elements and these elements admit equality.

   To create a Vector of Vectors, use the "tabulate" functions.


SEE ALSO

   SEQ_PARSE, ARRAY.


NOTES

   A possible implementation would be a view of a random access file.
   Possibly there should be a ConstVector signature (combining the Const
   and Vector signatures) to make this easier.


*)

(* TYPES *)

  eqtype 'a Vector
  eqtype 'a T
    sharing type T = Vector
```

```
(* CONSTANTS *)

  val empty: 'a Vector
   (* empty; the empty Vector. *)


(* CREATORS *)

  exception Size of string * int
   (* Size (fn, i); raised by the creation functions when they are invoked
      with a negative size. *)

  val create: Nat -> 'a -> 'a Vector
   (* create n i; create a Vector of n locations, each containing i.
      Raise (Size ("create", n)) if n < 0. *)

  val tabulate: Nat * (int -> 'a) -> 'a Vector
   (* tabulate (n, f); create a Vector v of n locations, (v sub 0) to
      (v sub (n-1)) with (v sub i) initialised to (f i).
      Raise (Size ("tabulate", n)) if n < 0. *)

  val tabulate': Nat * ('b -> 'a * 'b) * 'b -> 'a Vector
   (* tabulate' (n, f, base); create a Vector of n locations, (v sub 0) to
      (v sub (n-1)) with (v sub 0) initialised to (# 1 (f base)) and
      (v sub i (i > 0)) initialised to (# 1 (f (# 2 f_i))), where f_i is
      the result of the i^th application of f.
      Raise (Size ("tabulate'", n)) if n < 0. *)


(* CONVERTERS *)

  val stringSep: string -> string -> string ->
 ('a -> string) -> 'a Vector -> string
   (* stringSep start finish sep p v; returns the string representation of v,
      beginning with start, ending with finish, and with the elements
      separated by sep. *)

  val string: ('a -> string) -> 'a Vector -> string
   (* string p v; returns the canonical string representation of v. *)

  val printSep: outstream -> string -> string -> string ->
                (outstream -> 'a -> unit) -> 'a Vector -> unit
   (* printSep os start finish sep p v; sends the string representation of v
      to the stream os, beginning with start, ending with finish, and with
      the elements separated by sep. *)
```

```
val print: outstream -> (outstream -> 'a -> unit) -> 'a Vector -> unit
 (* print os p v; sends the canonical string representation of v to
    the stream os. *)

val list: 'a Vector -> 'a list
 (* list v; make a list containing (only) the elements of v, in
    the same order. *)

val fromList: 'a list -> 'a Vector
 (* fromList l; make a Vector containing (only) the elements of l, in
    the same order. *)


(* OBSERVERS *)

val isEmpty: 'a Vector -> bool
 (* isEmpty v; returns true if v is empty. *)

val size: 'a Vector -> Nat
 (* size v; return the number of elements in v. *)

val eq: ('a -> 'a -> bool) -> 'a Vector -> 'a Vector -> bool

val ne: ('a -> 'a -> bool) -> 'a Vector -> 'a Vector -> bool


(* SELECTORS *)

exception Subscript of string * int
 (* Subscript (fn, n); raised when the function named fn is passed the
    arguments n and l such that (not (0 <= n <= length l)). *)

(* infix 9 sub *)
val sub: 'a Vector * int -> 'a
 (* v sub n; return the n+1'th element of v.
    Raise (Subscript ("sub", n)) if not (0 <= n <= size v). *)

val nth: int -> 'a Vector -> 'a
 (* v sub n; return the n+1'th element of v.
    Raise (Subscript ("nth", n)) if not (0 <= n <= size v). *)

exception Extract of int * int
val extract: int -> int -> 'a Vector -> 'a Vector
 (* extract start finish v; returns the sub-vector of v starting with
    (v sub start) and ending with (v sub (finish - 1)).
    Returns the empty vector if (start = finish).
    Raise (Extract (start, finish)) if not (0 <= start,finish <= size v). *)
```

```
(* MANIPULATORS *)

  val rev: 'a Vector -> 'a Vector
   (* rev v; builds a vector containing the elements of v in reverse order. *)

  (* infix 6 ^ *)
  val ^ : 'a Vector * 'a Vector -> 'a Vector
   (* v ^ v'; builds a Vector containing the elements of v' appended to those
      of v. *)


(* REDUCERS *)

  val foldR: ('a -> 'b -> 'b) -> 'b -> 'a Vector -> 'b
   (* foldR f base v; folds using f associating to the right over the
      base element.
      foldR f base [a1,a2,...,an] = op(a1,f(a2,...,f(an,base)...)). *)

  val foldL: ('a -> 'b -> 'b) -> 'b -> 'a Vector -> 'b
   (* foldL f base v; folds using op associating to the left over the
      base element.
      foldL f base [a1,a2,...,an] = f(an,...,f(a2,f(a1,base))...). *)

  exception Empty of string

  val foldR': ('a -> 'a -> 'a) -> 'a Vector -> 'a
   (* foldR' f v; folds using f associating to the right over the
      last element of v.  Raises (Empty "foldR'") if v is empty. *)

  val foldL': ('a -> 'a -> 'a) -> 'a Vector -> 'a
   (* foldL' f v; folds using f associating to the left over the
      first element of v.  Raises (Empty "foldL'") if v is empty. *)

  val pairwise: ('a -> 'a -> bool) -> 'a Vector -> bool
   (* pairwise f v; true if (f (v sub i) (v sub (i + 1))) is true for all
      0 <= i < size v, or if v is empty. *)

(* ITERATORS *)

  val map: ('a -> 'b) -> 'a Vector -> 'b Vector
   (* map f v; builds a new vector by applying f to each element of v. *)

  val apply: ('a -> unit) -> 'a Vector -> unit
   (* apply f v; applies f to each element of v. *)
```

```
  val iterate: ('a * int -> 'b) -> 'a Vector -> 'b Vector
   (* iterate f v; builds a new vector by applying f to each element of v
      paired with its index. *)

  val iterateApply: ('a * int -> unit) -> 'a Vector -> unit
   (* iterateApply f v; applies f to each element of v paired
      with its index. *)
end
```

4.62 VectorParse

See SEQ_PARSE.

Chapter 5

Writing Library Entries.

The library provides a consistent framework which can easily incorporate new entries. New entries should use the same format as the existing entries, to keep the framework consistent. For example, the standard names described in Section 3.6 should be used where appropriate. This chapter mentions some particular points that you should observe.

Look at the existing entries before writing your own; this may help focus your ideas. The library includes two skeleton files, one for signatures and one for functors.

Don't use the identifiers `import`, `abstraction` and `abstract`. The first two are keywords in Standard ML of New Jersey, and the third is proposed in an extension to the Definition of SML. Using them will make your code non-portable.

5.1 Format of Signatures.

The format of user-visible signatures is described in Chapter 4. New entries should be written in the same style, so that users can easily find the information they want.

Some aspects of the format may not be immediately obvious:

- The header section must include creation and maintenance details. If no-one is maintaining this entry, the maintenance line should be: `Maintenance:  None`.

- The header section must also include a general description of the entry. The combination of this description and the comments associated with each function should be suitable for on-line reading. If more detailed documentation is needed, it should be put in the `doc` directory.

- The only comments to begin in the left-most column are the header section and the sub-headings used to structure the entry.

- The comments associated with each object should immediately follow its specification.

5.2 Format of Structures and Functors.

A structure or functor that implements a user-visible signature should provide creation and maintenance details in the same format. It may also include a general description of the implementation, but this is not required.

Functors arguments should always be written in the form `structure foo: FOO`. The short form, `functor F (foo: FOO) = ...`, should not be used. Each functor that is intended to be visible to users should have argument and result signatures that meet the requirements of the previous section. The header section should come after the arguments to the functor.

5.3 Installing A New Entry.

Each entry should be described by a signature. This will usually be a new signature written for the entry, although some entries will be described by existing generic signatures. New user-visible signatures should be placed in the `signatures` sub-directory of the library. They should include dependency declarations for the `Make` system. Most signatures will only depend on `InstreamTypes` and `GeneralTypes`.

Many entries will be simple enough to be implemented by a single structure of functor. In this case the structure or functor should be placed in a file in the `portable` subdirectory (or a system-specific distribution sub-directory if appropriate). It should contain a dependency declaration for the `Make` system, and also an application of the `loadSig` function to a string containing the name of the user-visible signature. The latter is used by the `build_all.sml` and `build_core.sml` build files.

More complex entries (such as the `Make` system) will comprise several structures and functors, and more signatures than the user-visible one. Such entries should be given a sub-directory in the `portable` directory, containing all the files except the user-visible signature. In the existing entries, I've prefixed all local structure and signature identifiers by the name of the whole package, to make them less likely to clash with existing or user-defined names.

The local files of a complex package should be given dependency declarations for the `Make` system, as usual.[1] In addition, they should use the `loadLocalSig` and `loadEntry` functions to load files in the correct sequence when they are built without `Make`. You will need to change the `build_all.sml` build file to load the package properly. The function `setLoadPrefix` must be applied to the name of the sub-directory before the call to `loadEntry` that loads the package, and applied to the empty string afterwards. Look at the existing code for the `Make` and`Core` sub-directories.

You will also have to change the `INSTALL` file for every new sub-directory that you create. The first change is to generate the correct entries in the `ML_CONSULT` file used by the `build_make.sml` build file. The second change is to ensure that the new entry is linked into all versions of the library. Again, look at the existing code for the `Make` and `Core` subdirectories.

After you've added a new entry, you should run the `INSTALL` program to generate the new `ML_CONSULT` file for `build_make.sml`.

[1] The two existing sub-directories, `Make` and `Core`, don't have dependency declarations because they're loaded by `build_make.sml` itself.