Standard ML Code

ÁNOQ of the Sun, Hardcore Processing *

February 12, 2001

1 Introduction

This page gives you access to lots of reusable pieces of `Standard ML` code. It is all distributed under GNU Lesser General Public License (`LGPL`) - except for one illustrative example which is under `GPL`. The code is released as small separate projects, but in a way that should make it easy to install several projects and use them together.

2 This document in other formats

This document is a available in the following different file formats:

- `http://www.HardcoreProcessing.com/Freeware/StandardMLCode.html`
- `http://www.HardcoreProcessing.com/Freeware/StandardMLCode.pdf`
- `http://www.HardcoreProcessing.com/Freeware/StandardMLCode.ps`

*©2000 ÁNOQ of the Sun (alias Johnny Andersen)

3 Available Projects

The following projects are available for download by clicking on the links:

- `CompatibilityLayer project` - gives various ML compilers a more uniform set of Standard ML Basis library modules, such as Word32, IntArray etc.
- `ErrorHandling project` - an attempt to provide a general framework for handling errors in Standard ML. This is required by some of the other projects.
- `Arith project` - some standard arithmetic signatures used many places in the projects below.
- `Geometry project` (requires the *Arith* project) - some basic geometry stuff like 3D point, 3D vector, 2D points with either real or integer coordinates, 2D rectangle etc.
- `Matrix project` (requires *Arith* and *ErrorHandling* projects) - 4x4 matrices, mostly for making coordinate transformations on 3D points.
- `Collections project` - implements a ternary search tree structure.
- `NumericRep project` (requires *ErrorHandling* project) - implements conversion of *binary IEEE Reals* to ML real values.
- `FunctionalIO project` - implements functional input streams, both text based and binary.
- `ParsingToolkit project` (requires *CompatibilityLayer* and *FunctionalIO* projects) - binary and text based *combinator parser* utilities using functional input streams.
- `test.rib.gz file` - a packed 4MB RenderMan RIB file for testing the *ParsingToolkit* and *FunctionalIO* projects on some more real data.

If you want to use several projects together at the same time, just unpack them all in the same directory. An appropriate directory structure is used for this. Most, if not all, of the projects should work with `SML/NJ`, `MLton` and `MLKit`. It has also been used with *MLWorks* once upon a time, but that is not supported any more. The projects will be documented in the following.

4 CompatibilityLayer

This project gives a compatibility layer between various Standard ML compilers. Including this module in your own project will give a more uniform set of Standard ML Basis library modules. In particular it should give you the following modules:

- IEEEReal
- LargeReal
- Word32
- IntVector
- StringVector
- BoolVector
- IntArray
- StringArray
- BoolArray

It is implemented for the following ML compilers:

- MLKit
- SML/NJ
- MLWorks

However there is no guarantee that I have had a use for all the listed modules for all the supported compilers - but then at least you should be able to add those modules quite easily by looking at the source code.

5 ErrorHandling

This project is an attempt to provide a general framework for handling errors in Standard ML. It provides both a simple signature for reporting errors and a framework of general exceptions.

5.1 ERROR_REPORTER signature and ErrorStdOut structure

The *ERROR_REPORTER* signature looks like the following:

```
signature ERROR_REPORTER =
  sig
    val bug : string -> unit
    val error : string -> unit
    val warning : string -> unit
  end
```

There is a structure called *ErrorStdOut* which implements this signature. In this structure the errors will just be shown as error messages on the standard output device. Further more the functions *bug* end *error* will also raise exceptions after the errormessage has been shown.

5.2 Excn structure

This structure attempts to be a general exception framework. The framework is probably best suited for a user interface toolkit where one can implement a general system for displaying errors as dialog boxes - but I'm not really sure if this is a good framework or not. The signature looks like this:

```
signature EXCN =
  sig
    datatype t =
      Generic
    | Bug
    | NotImplemented

    (* The excpetion data should be treated as being opaque
       - i.e.: DO NOT TOUCH! *)
    exception EGeneric of (t * string * string) list

    val eRaise : (t * string * string) -> 'a
    val eRaiseBug : string -> 'a
    val eRaiseGeneric : string * string -> 'a
    val eRaiseNotImplemented : string -> 'a

    val reRaise : exn -> (t * string * string) -> 'a

    val output : (string -> unit) -> exn -> unit

    val checkedExecOutF : (string -> unit) ->
```

```
        ('a -> 'b) -> 'a -> 'b -> 'b
  val checkedExec : ('a -> 'b) -> 'a -> 'b -> 'b
end
```

This framework classifies exceptions as either generic exceptions which could be anything, bugs which is errors that should never occur and an exception for when some part of a program is not implemented. It is intended that one does not raise exceptions other than through the supplied functions. All exceptions have 3 data elements:

- An exception class - either *Generic*, *Bug* or *NotImplemented*.
- Some descriptive category as a string.
- Some error or exception message as a string.

The *EGeneric* exception (that you're not supposed to raise yourself...) contains a list of exceptions. This is intended to be a stack of raised exceptions when the exception reaches the point where it will be shown to the user. The stack will be built when using the *reRaise* function which can re-raise an exception caught at a lower level to give a higher level exception. The function *eRaise* is the general function for raising an exception. *eRaiseBug*, *eRaiseGeneric* and *eRaiseNotImplemented* are the specific versions. The functions *eRaiseBug* and *eRaiseNotImplemented* will give empty descriptive categories.

The function *checkedExec* is for automatically handling exceptions in this framework. It takes as arguments a function to be executed - of type *'a -¿ 'b*, the argument to be supplied to the function and a default value to return in case an exception is raised. This function will display errors on the standard output device. The function *checkedExecOutF* is a more general version of this function. It takes as the first argument a function for displaying errors. So *checkedExec* is equivalent to *checkedExecOutF print*.

The function called *output* takes a function for displaying errors and an exception raised with this framework. The function will then show an appropriate error message. If the given exception is not the *EGeneric* exception used in this framework, then an error will be displayed indicating this.

6 Arith

The Arith project is a set of very useful arithmetic signatures. It is inspired by some of the *ARITH* signatures found in Larry Paulson's book ML for the Working Programmer. The following is some documentation for the signatures:

6.1 ADD_ARITH

The *ADD_ARITH* signature is a basic set of functions for adding, subtracting negating and comparing values of some given type *t*. The signature is listed here:

```
signature ADD_ARITH = (* Addition arithmetic *)
  sig
    type t

    (* When zeroValue is added to t, the result is t *)
    val zeroValue       : t
    val add             : (t * t) -> t
    val subtract        : (t * t) -> t

    (* Inverse number for add *)
    val negate          : t -> t

    (* Comparison *)
    val equal           : (t * t) -> bool
  end
```

The value called *zeroValue* is called *zeroValue* to be consistent with the value called *unitValue* in the *MULT_ARITH* signature.

6.2 SCALE_ARITH

SCALE_ARITH provides functions for scaling values of some given type *t* with scalar values. The signature is listed here:

```
signature SCALE_ARITH = (* Scaling arithmetic *)
  sig
    type t
    type scalar

    val scale        : (t * scalar) -> t
    val inverseScale : (t * scalar) -> t
  end
```

The function *scale* will scale a value of type *t* with a value of type *scalar*. *inverseScale* will scale a value of type *t* by the multiplicative inverse of a value of type *scalar*. For instance if the type *t* is a coordinate vector of real valued coordinates and *scalar* have type *real*, the function *scale* would scale a vector by a real number and the function *inverseScale* would scale a vector by the reciprocal value of a real number - i.e. vector * 1 / scalar. You should look at

the signature *MULT_ARITH* to see why the functions are not called *multiply* and *divide*.

6.3 MULT_ARITH

MULT_ARITH gives a signature for multiplying, dividing and inverting values of some given type *t*. The signature is listed here:

```
signature MULT_ARITH = (* Multiplication arithmetic *)
  sig
    type t

    (* It's called unitValue and not unit to avoid
       conflicts with the SML type unit...*)
    (* When unitValue is multiplied by t, the result is t *)
    val unitValue       : t
    val multiply        : (t * t) -> t
    val divide          : (t * t) -> t

    (* Inverse number for multiply *)
    val invert          : t -> t
  end
```

As can be seen from the comment in the code, it is not possible to call a value *unit* in Standard ML, because this is a builtin type.

6.4 ADD_SCALE_ARITH

This signature is the union of the *ADD_ARITH* and the *SCALE_ARITH* signatures. So it simply contains everthing that these two signatures contain.

6.5 ADD_MULT_ARITH

This signature is the union of the *ADD_ARITH* and the *MULT_ARITH* signatures. So it simply contains everthing that these two signatures contain.

6.6 ADD_SCALE_MULT_ARITH

This signature is the union of the *ADD_ARITH*, *SCALE_ARITH* and the *MULT_ARITH* signatures. So it simply contains everthing that these 3 signatures contain.

7 Geometry

The geometry project contains some basic geometry stuff. In particular it contains the following modules:

- *Point3D* - 3D Point structure with real valued coordinates.
- *Vector3D* - 3D Vector structure with real valued coordinates.
- *Point2D* - 2D Point structure with real valued coordinates.
- *IPoint2D* - 2D Point structure with integer valued coordinates.
- *IRect2D* - 2D Rectangle structure with integer valued coordinates.
- *ILine2D* - 2D lines structure with integer coordinates (very incomplete).
- *ISize2D* - 2D sizes structure with integer coordinates (very incomplete).

Each structure will be documented separately in the following.

7.1 Point3D

The *Point3D* structure implements 3D points with real valued coordinates. It implements the *ADD_SCALE_ARITH* signature and 3 additional functions for projecting points to 2D along the principal axes. The signature looks like this:

```
signature POINT3D = (* 3D Point with addition arithmetics *)
  sig
    type coord (* This has type real in the Point3D structure *)
    include ADD_SCALE_ARITH
              where type t = {x : coord, y : coord, z : coord}
                and type scalar = real
    type point2d = {x : coord, y : coord}

    val xyCoordsAs2D : t -> point2d
    val xzCoordsAs2D : t -> point2d
    val yzCoordsAs2D : t -> point2d
  end
```

7.2 Vector3D

The *Vector3D* structure implements 3D vectors with real valued coordinates. It implements the *ADD_SCALE_ARITH* signature and some additional functions. The signature looks like this:

```
signature VECTOR3D = (* 3D Vector with addition arithmetics *)
  sig
    type coord (* This has type real in the Vector3D structure *)
    include ADD_SCALE_ARITH
              where type t = {x : coord, y : coord, z : coord}
                and type scalar = real
    type vertex = t
```

```
    val fromVertices : (vertex * vertex) -> t
    val dotProduct   : (t * t) -> real
    val crossProduct : (t * t) -> t
    val length       : t -> real
    val normalize    : t -> t
  end
```

The function *fromVertices* creates a vector between 2 points in space, with the first point being the start of the vector and the second point being the end of the vector. *dotProduct* computes the dot product between 2 vectors and *crossProduct* computes the crossproduct between 2 vectors. *length* returns the length of a vector and *normalize* normalizes a given vector into a vector with the same direction and length 1.

7.3 Point2D

The *Point2D* structure implements 2D points with real valued coordinates. It implements the *ADD_SCALE_ARITH* signature and an additional function called *move*. The signature looks like this:

```
signature POINT2D =
  sig
    type coord (* This has type real in the
                   Point2D structure *)
    include ADD_SCALE_ARITH
              where type t = {x : coord, y : coord}
                and type scalar = real

    (* This is really the same as add, but it
       takes a pair of coords as the vector to move.
       So it's just for convenience. *)
    val move : (t * (coord * coord)) -> t
  end
```

7.4 IPoint2D

The *Point2D* structure implements 2D points with integer valued coordinates. It implements the *ADD_SCALE_ARITH* signature, the *move* function from the *POINT2D* signature and and 2 additional functions for converting to and from points with real valued coordinats. The signature looks like this:

```
signature IPOINT2D =
  sig
    include POINT2D where type coord = int

    (* For converting to and from real-valued points. *)
    val toPoint2d : t -> Point2D.t
    val fromPoint2d : Point2D.t -> t
  end
```

7.5 IRect2D

The *IRect2D* structure implements 2D rectangles with integer valued coordinates. The signature looks like this:

```
signature IRECT2D =
  sig
    (* Public types and values *)
    type t = {x : int, y : int, w : int, h : int}

    val zeroValue  : t

    (* Public functions *)
    (* FIXME: Rename to hasNoArea? or isEmpty?
              or isDegenerate? *)
    val isZero : t -> bool

    val equal  : t * t -> bool

    (* If the rectangle gets zero extent -
       zeroValue is returned *)
    val inset  : t * int -> t
    val move   : t * (int * int) -> t

    (* A point on the boudary of a rectangle is also
       considered to be contained in the rectangle. *)
    val containsPoint : t * IPoint2D.t -> bool

    (* Returns zeroValue for empty intersections
       i.e. intersections with zero or less
       height or width. *)
    val intersect     : t * t -> t
    val union         : t * t -> t
  end
```

The value *zeroValue* is a rectangle with (0, 0) as the corner vertex and width and hight 0. The function *isZero* checks if a rectangle does not have any extent or a negative extent (Notice: This function might be renamed in the future). *inset* shrinks the rectangle from all sides by some integer value. *move* will move the rectangle to another position. *containsPoint* returns true if the given point is inside or on the boundary of the given rectangle. *intersect* and *union* creates the intersection or the union, respectively, of two rectangles.

7.6 ILine2D

The structure *ILine2D* is very incomplete and only contains the a type for 2D lines with integer coordinates. This type is:

```
type t = {x0 : int, y0 : int, x1 : int, y1 : int}
```

7.7 ISize2D

The structure *ISize2D* is very incomplete and only contains the a type for 2D sizes with integer coordinates. This type is:

```
type t = {w : int, h : int}
```

8 Matrix

This project implements 4x4 matrices. It is mostly intended for doing coordinate transformations on 3D points. The structure implements the *ADD_SCALE_MULT_ARITH* signature as well as some other stuff. The signature looks like this:

```
signature MATRIX_4X4 =
  sig
    include ADD_SCALE_MULT_ARITH
              where type scalar = real
    type tuples = Matrix4x4Type.t

    val fromTuples : tuples -> t
    val toTuples : t -> tuples
    val determinant : t -> real
    val transpose : t -> t
    val adjoint : t -> t

    structure Const : MATRIX_4X4_COMMON
                        where type angle = real
    structure Trans : MATRIX_4X4_TRANSFORMATION
                        where type t = t

    sharing type t = Const.t
  end
```

It should be emphasized that matrix multiplication is not symmetric. So the result of multiplying two matrices with the function *multiply* will in general depend on which matrix is the left operand and which is the right operand. The *Matrix4x4* structure assumes the mathematical standard representation of matrices - meaning that you will most likely want to multiply matrices on to the *left* of another matrix. Please read the section below called *About Matrices* for futher details on these issues.

The type *tuples* is a 4-tuple of 4-tuples of reals where the outer tuple is a tuple of rows and each of the inner tuples is a row. The functions *fromTuples* and *toTuples* converts to and from this representation. This is also the representation used internally in the implementation - but that could easily change without changing the signature.

There are also 2 nested structures. The structure *Const* is a collection of functions for creating some handy 4x4 matrices. *Trans* is for transforming 3D points and 3D vectors. Finally there are also 3 functions which are particular to matrices. These are *determinant*, *transpose* and *adjoint* which work as you would expect.

8.1 MATRIX_4X4_COMMON

The structure *Const* has the following signature:

```
signature MATRIX_4X4_COMMON =
  sig
    type t (* This is the type of a matrix in the
```

```
            structure Const *)
    type angle (* This is type real in the structure
                  Const *)

    val rotateSinCosX : real * real -> t
    val rotateSinCosY : real * real -> t
    val rotateSinCosZ : real * real -> t
    val rotateX : angle -> t
    val rotateY : angle -> t
    val rotateZ : angle -> t

    (* FIXME: rotateXYZ not tested! *)
    val rotateXYZ : angle * real * real * real -> t
    val scaleXYZ : real * real * real -> t
    val translateXYZ : real * real * real -> t
    val shearXY : real * real -> t
    val shearXZ : real * real -> t
    val shearYZ : real * real -> t
    val perspective : real -> t (* FIXME: Not tested! *)
    val parallel : t (* FIXME: Not tested! *)
    val swapLRCoordsys : t (* FIXME: Not tested! *)
  end
```

If you know about 3D coordinate transformations this should be almost self explanatory. Otherwise you might want to read up on it anyway :) Notice that *angle* is expected to be in radians. The functions *rotateSinCosX*, *rotateSinCosY* and *rotateSinCosZ* can be used if you have the sine and cosine of a desired rotation angle and they are more efficient than using *rotateX*, *rotateY* and *rotateZ*.

The reason why this signature is called *MATRIX_4X4_COMMON* is that I have had plans for implementing a structure called *Oper* which has the same signature - except that the type t would be *Matrix4x4.t -¿ Matrix4x4.t*. The purpose of this would be to create optimized functions for multiplying a matrix with each of the common transformation matrices listed in the signature.

8.2 MATRIX_4X4_TRANSFORMATION

The signature for transforming 3D points and 3D vectors looks like this:

```
signature MATRIX_4X4_TRANSFORMATION =
  sig
    type t (* Matrix type *)

    val transformPoint : t -> {x : real, y : real, z : real} ->
                              {x : real, y : real, z : real}

    val transformVector : t -> {x : real, y : real, z : real} ->
                               {x : real, y : real, z : real}
  end
```

These points and vectors must have the type *x : real, y : real, z : real* which is also the type used in the *Geometry* project described above.

The functions will transform points and vectors by considering them as columns multiplied on to the *right* of the matrix - or put another way by multiplying a matrix on to the *left* of the point. This conforms to the mathematical standard way of transforming points.

8.3 About Matrices

First off it should be noted that some computer graphics textbooks use the mathematical standard way of transforming points while other books use a transposed notation for matrices.

Some books that follow the mathematical standard when describing coordinate transformations are listed here:

- *Computer Graphics: Principles and Practice* by Foley, van Dam, Feiner and Hughes.
- *3D Game Engine Design* by David H. Eberly.
- *Real-Time Rendering* by Thomas Moller end Eric Haines.
- *Warping and Morphing of Graphical Objects* by Gomes, Darsa, Costa and Velho.

Some books that use the transposed notation are listed here:

- *Advanced Animation and Rendering Techniques* by Mark Watt and Alan Watt.
- *Advanced RenderMan: Creating CGI for Motion Pictures* by Larry Gritz and Anthony A. Apodaca.
- *Graphics Gems* edited by Andrew S. Glassner.

.

You would typically transform a point by a series of matrices like this:

```
Ptransformed = (M5 * M4 * M3 * M2 * M1) * Poriginal
```

First you would mutiply the matrices to obtain a single matrix and then transform your bzillion points by multiplying with the resulting matrix. In ML you can multiply a list of matrices together with the following code:

```
val m = foldr Matrix4x4.multiply Matrix4x4.unitValue matrices
```

Or equivalently, since matrix multiplication is associative:

```
fun swap (a, b) = (b, a)

val m = foldl (Matrix4x4.multiply o swap)
              Matrix4x4.unitValue matrices
```

However using *foldr* is probably slower and might use memory proportional to the length of the list *matrices*.

In case you want to generate the inverse of a list of matrix transformations, you could invert each matrix and multiply the matrices together in the opposite order. This can be done with the following code:

```
val mInverse = foldl Matrix4x4.multiply
                     Matrix4x4.unitValue
                     (map myInvert matrices)
```

9 Collections

This project implements a ternary search tree structure which is used in the Standard ML bindings for RenderMan - `RI::ML`. In my own CVS repository, the *Collections* project also contains other structures, such as dictionary, set implementations, hash table etc. However I'm still not content with the signatures for those structures, so it is not released here yet. However you can find some older versions of these structures as part of the `AbstactUI::ML` project. Actually I would like to implement some very general signatures usable for many kinds of collection structures - even many of those found in the Standard ML Basis library! But, alas - time is in short supply.

But, returning to the ternary tree structure, the signature looks like this:

```
signature TERNARY_TREE =
  sig
    type 'a t

    val empty : 'a t
    val lookup : 'a t -> string -> 'a option

    (* insert overwrites any previous data. *)
    val insert : 'a t -> string -> 'a -> 'a t
  end
```

It's like a functional dictionary structure with a string as the key value. Lookups and inserts are all done in practical constant time... actually linear time in the length of the key string, but never mind. It might even be more efficient than using a hash table! *empty* is an empty search tree, the *lookup* function will look up a string value in the search tree and *insert* will insert a new value with a key string into the tree - possibly overwriting existing data for that key. And, that's it! :)

10 NumericRep

This project currently only implements conversion of binary single precision IEEE floating point values into ML real values. The project has two structures *IEEESingleFloat* and *IEEEDoubleFloat*. However only *IEEESingleFloat* is implemented. The signature for both structures looks like this:

```
signature IEEE_FLOAT =
  sig
    type binary (* This is Word32.word in IEEESingleFloat and
                           Word64.word in IEEEDoubleFloat  *)

    val toLargeReal : binary -> LargeReal.real
    val fromLargeReal : LargeReal.real -> binary
  end
```

Only the function *toLargeReal* is implemented in IEEESingleFloat. Converting the other way seems more troublesome, and I have not had any need for it. The *IEEESingleFloat* structure is used in my commercial application called `CeX3D Converter` for reading binary float values in the LightWave 3D object file format, so it is production proven :)

10.1 Proposal for the Standard ML Basis library

I believe it would be very natural if a few functions for converting to and from binary IEEE float values were added to the REAL signature in the Standard ML Basis library. This is mostly because that these modules already use the IEEE binary representations internally.

It could be implemented by adding functions *toBinary* and *fromBinary* and implementing single precision for *Real32* and double precision for *Real64*. But of course this might make the *Real* module inconsistent. Alternatively, all the *Real* modules could have 4 functions *toBinarySingle*, *toBinaryDouble*, *fromBinarySingle* and *fromBinaryDouble*, but this is less orthogonal, so I'm not sure what would be best.

11 FunctionalIO

This project implements a way of reading binary and text files in a functional way. It tries to match a subset of the the *TextIO* and *BinIO* modules of the Standard ML Basis library as closely as possible. It contains the structures *FuncBinIO* and *FuncTextIO*. The basic signature looks like this:

```
signature FUNCTIONAL_IO =
  sig
    include IO

    type vector (* This is Word8Vector.vector in FuncBinIO
                        and CharVector.vector in FuncTextIO. *)
    type elem (* This is Word8.word in FuncBinIO
                        and Char.char in FuncTextIO. *)
    type instream
    type outstream

    val input : instream -> vector * instream
    val input1 : instream -> (elem * instream) option
    val inputN : (instream * int) -> vector * instream

    (* Closes file for further input.
        New end of file becomes the furthest
        position in the file that has been read
        internally. *)
    val closeIn : instream -> unit
  end
```

However the *FuncBinIO* and *FuncTextIO* structures also implement the *openIn* function as can be seen in this signature:

```
signature FUNC_BIN_IO =
  sig
    include FUNCTIONAL_IO

    val openIn : string -> instream
  end
```

The functions *input* and *inputN* will return vectors of length zero if no more characters can be read from the stream. An important thing to notice is that *closeIn* will only close the underlying file, meaning that all the functional versions of the stream can continue to read as much data as has already been read into memory from the file.

12 ParsingToolkit

This project implements combinator parsers as described in *ML for the Working Programmer* by Larry Paulson. However this implementation uses the *FunctionalIO* streams described above. The implementation is based on a combinator parser implementation by Fritz Henglein.

The main structures of this project are *TextIOParserCombinators* and *BinIOParserCombinators*. The signature for these looks like this:

```
signature PARSER_COMBINATORS =
  sig
    type instream (* This is FuncTextIO.instream in
                             TextIOParserCombinators
                         and FuncBinIO.instream in
                             BinIOParserCombinators *)
    type vec (* This is string in TextIOParserCombinators
                    and Word8Vector.vector in
                        BinIOParserCombinators *)
    type elem (* This is char in TextIOParserCombinators
                     and Word8.word in
                         BinIOParserCombinators *)

    (* This is the type of a parser. A parser is a
       function taking a functional instream as
       argument. It returns a value that has been
       created by the parser during parsing, and
       a new functional instream with the stream
       position updated to where the parser stopped
       reading. *)
    type 'a parser = instream -> ('a * instream)

    (* SyntaxError is raised when a parser fails to parse. *)
    exception SyntaxError of string * instream

    (* Functions given to the >> combinator are expected
       to raise ValidityError on invalid arguments. *)
    exception ValidityError of string

    (* The combinators *)
    (* The purpose of this combinator is to try parsing
       with 2 parser functions and return the result
       of the first function that succeeds. *)
    val || : ('a parser) * ('a parser) -> ('a parser)

    (* This combinator will execute 2 parsers in sequence
       and return a pair of the results of the parsers. *)
    val -- : ('a parser) * ('b parser) -> (('a * 'b) parser)

    (* This combinator executes 2 parsers in sequence and
       ignores the result of the first parser. *)
```

```
  val $-- : ('a parser) * ('b parser) -> ('b parser)

  (* This combinator executes 2 parsers in sequence and
     ignores the result of the second parser. *)
  val --$ : ('a parser) * ('b parser) -> ('a parser)

  (* Execute a parser and run the result through a function. *)
  val >> : ('a parser) * ('a -> 'b) -> ('b parser)

  (* This combinator is for reading an verifying an expected
     keyword. *)
  val $$ : vec -> (vec parser)

  (* Some handy built-in parsers. *)

  (* Doesn't parse anything, just returns nil. *)
  val empty : ('a list) parser

  (* Given a predicate, returns a parser that will read an
     element from the stream if the predicate is true. *)
  val getIf : (elem -> bool) -> (elem parser)

  (* Given a parser, returns a parser that will read a
     list of values with the given parser. Parses as
     many values as possible. *)
  val repeat : ('a parser) -> (('a list) parser)

  (* Given a predicate, returns a parser that will
     read a list of elements, until the predicate
     is false. *)
  val repeatIf : (elem -> bool) -> ((elem list) parser)

  (* Given a number n and a parser, returns a parser
     that parses a list of n values with the given
     parser. *)
  val repeatN : int -> ('a parser) -> ('a list parser)

  (* Same as repeatIf, except that this will read at
     least one value - or fail. *)
  val repeatOneIf : (elem -> bool) -> ((elem list) parser)
end
```

A parser (of type *'a parser*) is a function which reads data from a given input stream and returns a value created or read during parsing and the input stream updated with a new stream position. If a parser fails to read what it is supposed to read, it raises the exception *SyntaxError*.

Parsers operate on input streams of the type *instream*, which is from the text based or the binary *FunctionalIO* modules, depending on whether is is the binary parsers or the string based parsers. The data read from input streams are vectors of elements, where each element has type *elem* and the vector of

elements has type *vec*.

Now we will go through all the combinators for constructing new parser functions from existing parser functions:

- `||` The purpose of this combinator is to try parsing with 2 parser functions and return the result of the first function that succeeds. So, when given two parsers, it returns a new parser which does the following: First try the first parser given, by starting at the position of the given input stream, if it fails try the second parser at the position given input stream. Returns the result of the parser that succeedes first and the corresponding updated input stream. Notice that this is the only combinator which does any backtracing during parsing, so it is the only combinator which requires reading the input stream more than once - and thus slowing down parsing.

- `--` This combinator will execute 2 parsers in sequence and return a pair of the results of the parsers. So, when given two parsers, it returns a new parser which does this: First execute first parser at the position of the given input stream, then second parser on the resulting position of the input stream. Return a tuple of the results of both parsers and the updated input stream.

- `$--` This combinator executes 2 parsers in sequence and ignores the result of the first parser.

- `--$` This combinator executes 2 parsers in sequence and ignores the result of the second parser.

- `>>` Execute a parser and run the result through a function. The function may raise the *ValidityError* exception on invalid input.

- `$$` This combinator is for reading an verifying an expected keyword. So, when given a value of type vec, it returns a parser which does this: Try to read from the given input stream. If the given vec value can be read from the stream, return the given vec value and the input stream with the new position.

Some handy parsers are also supplied in these modules. It should be more or less clear what they do from the comments in the signature above. It should be noted that if the repeat parsers are implemented naively, they will use huge amounts of memory when reading a list of data values. However, the not-so-naive implementation supplied here works very well in practice.

There is also a structure called *TextIOParserCombExtra*. It gives a few more handy parser functions for use with text based parsing. Also, it can be see as a quick and very dirty example of writing parser functions. The signature looks like the following:

```
signature TEXT_IO_PARSER_COMB_EXTRA =
  sig
    type instream = TextIOParserCombinators.instream
    type elem = TextIOParserCombinators.elem
    type 'a parser = 'a TextIOParserCombinators.parser
```

```
  val isWhitespaceChar : char -> bool
  val isLetterChar : char -> bool
  val isDigitChar : char -> bool

  val whitespaceForce : elem list parser
  val whitespace : elem list parser

  val getReal : instream -> real * instream
  val getRealWS : real parser
end
```

The functions *whitespace* and *whitespaceForce* are parsers that will read as many whitespace characters as possible. *whitespaceForce* will read at least one whitespace - or fail. The function *getReal* is a parser that tries to read a floating point value, as they look in many programming languages, and return it as an ML real value. *getRealWS* will also read a floating point value as a real, but will also read and skip any whitespaces that may be following the floating point value.

12.1 An example of a more real parser

A small example is included for reading a small subset of a `RenderMan` RIB file. There is no documentation for this, and it is included only for instructive purposes of how to write a parser. This particular example is released under `GPL` and not `LGPL` as the rest of the code. There is a 4MB RenderMan RIB file called *test.rib* available for download (it is packed with gzip), so that you can see that the parser combinators and the functional IO modules work well in practice even with relatively big files.