A Proposal for Standard ML

Robin Milner, November 1983

1. Introduction

1.1 How this proposal evolved

ML is a strongly typed functional programming language, which has been used by a number of people for serious work during the last few years [1]. At the same time HOPE, designed by Rod Burstall and his group, has been similarly used [2]. The original DEC-10 ML was incomplete in some ways, redundant in others. Some of these inadequacies were remedied by Cardelli in his VAX version; others can be put right by importing ideas from HOPE.

In April '83, prompted by Bernard Sufrin, I wrote a tentative proposal to consolidate ML, and while doing so became convinced that this consolidation was possible while still keeping its character. The main strengthening came from generalising the "varstructs" of ML - the patterns of formal parameters - to the patterns of HOPE, which are extendible by the declaration of new data types.

Many people immediately discussed the initial proposal. It was extremely lucky that we managed to have several separate discussions, in large and small groups, in the few succeeding months; we could not have chosen a better time to do the job. Also, Luca Cardelli very generously offered to freeze his detailed draft ML manual [3] until this proposal was worked out.

The proposal went through a second draft, on which there were further discussions. The results of these discussions were of two kinds. First, it became clear that two areas were still contentious: input/output and facilities for separate compilation. Second, many points were brought up about the remaining core of the language, and these were almost all questions of fine detail. The conclusion was rather clear; it was obviously better to present at first a definition of the core language without the two contentious areas. This course is further justified by the fact that the core language appears to be almost completely unaffected by the choice of input/output primitives and of separate compilation constructs. Also, there are already strong and carefully considered proposals, from Cardelli and MacQueen respectively, on how to design these two vital facilities. These proposals will appear very soon, and together with this document they will form a complete language definition which can be adopted in its entirety, while still leaving open the possibility of adopting only parts of it. But the strong hope is that the whole will be very widely accepted.

The main contributors to the proposed language, through their design work on ML and on HOPE, are:

Rod Burstall, Luca Cardelli, Michael Gordon, David MacQueen,
Robin Milner, Lockwood Morris, Malcolm Newey, Christopher Wadsworth.

The final proposal also owes much to criticisms and suggestions from many other people: Guy Cousineau, Gerard Huet, Robert Milne, Kevin Mitchell, Brian Monahan, Peter Mosses, Alan Mycroft, Larry Paulson, David Rydeheard, Don Sannella, David Schmidt, John Scott, Stefan Sokolowski, Bernard Sufrin, Philip Wadler. Most of them have expressed strong support for most of the design; any inadequacies which remain are my fault, but I have tried to represent the consensus.

[1] M.Gordon, R.Milner and C.Wadsworth, Edinburgh LCF; Springer-Verlag, Lecture Notes in Computer Science, Vol 78, 1979.
[2] R.Burstall, D.MacQueen and D.Sannella, HOPE: An Experimental Applicative Language; Report CSR-62-80, Computer Science Dept, Ediburgh University, 1980.
[3] L.Cardelli, ML under UNIX; Bell Laboratories, Murray Hill, New Jersey, 1982.

1.2 Design principles

The proposed ML is not intended to be the functional language. There are too many degrees of freedom for such a thing to exist: lazy or eager evaluation, presence or absence of references and assignment, whether and how to handle exceptions, types-as-parameters or polymorphic type-checking, and so on. Nor is the language or its implementation meant to be a commercial product. It aims to be a means for propagating the craft of functional programming and a vehicle for further research into the design of functional languages.

The over-riding design principle is to restrict the core language to ideas which are simple and well-understood, and also well-tried - either in previous versions of ML or in other functional languages (the main other source being HOPE, mainly for its argument-matching constructs). One effect of this principle has been the omission of polymorphic references and assignment. There is indeed an elegant and sound scheme for polymorphic assignment worked out by Luis Damas; unfortunately it is not yet documented, and we will do better to wait for a clear exposition either from Damas or - as promised - from David MacQueen. In the proposed language much can be done to get the polymorphic effect by passing assignment functions as parameters; it is worthwhile experimenting with this method, and there is further advantage in keeping to the simple polymorphic type-checking discipline which derives from Curry's Combinatory Logic via Hindley.

A second design principle is to generalise well-tried ideas where the generalisation is apparently natural. This has been applied in generalising ML varstructs to HOPE patterns, in broadening the structure of declarations (following Cardelli's declaration connectives which go back to Robert Milne's Ph.D. Thesis) and in allowing exceptions which carry values of arbitrary polymorphic type. It should be pointed out here that a difficult decision had to be made concerning HOPE's treatment of data types - present only in embryonic form in the original ML - and the labelled records and variants which Cardelli introduced in his VAX version. The latter have definite advantages which the former lack; on the other hand, the HOPE treatment is well-rounded in its own terms. Though a combination of these features is possible, it seemed (at least to me, but some disagreed!) to entail too rich a language for the present proposal. Thus the HOPE treatment is fully adopted here.

A third principle is to specify the language completely, so that programs will port between correct implementations with minimum fuss. This entails, first, precise concrete syntax (abstract syntax is in some senses more important - but we do not all have structure editors yet, and humans still communicate among themselves in concrete syntax!); second, it entails exact evaluation rules (e.g. we must specify the order of evaluation of two expressions, one applied to the other, just because of the exception mechanism). Third, the principle implies that the present document is not a full language definition; it only becomes so when the proposals for input/output and for separate compilation are added.

1.3 An example

The following declaration illustrates some constructs of the proposed language. A longer expository paper should contain many more examples; here, we hope only to draw attention to some of the less familiar ideas.

The example sets up the abstract type 'a dictionary , in which each entry associates an item (of arbitrary type 'a) with a key (an integer). Besides the null dictionary, the operations provided are for looking up a key, and for adding a new entry which overrides any old entry with the same key. A natural representation is by a list of key-item pairs, ordered by key.

```
abstype 'a dictionary = dict of (int * 'a) list
with                                              {dict is the abstraction}
                                                  {        constructor.   }
   val nulldict = dict nil
                                                  {The function lookup may}
   exception lookup : unit                        {    raise an exception.}

   val lookup (key:int)                           {'a is the result type. }
            (dict entrylist) :'a  =
      let val rec search nil = escape lookup       {An auxiliary clausal   }
                | search ((k, item)::entries) =    {  function declaration.}
                    if key=k then item
                    else if key<k then escape lookup
                    else search entries
      in search entrylist
      end

   val enter (newentry as (key,item))             {A layered varstruct.   }
           (dict entrylist) :'a dictionary  =
      let val rec update nil = [ newentry ]       {A singleton list.      }
                | update ((entry as (k,_))::entries) =
                    if key=k then newentry::entries
                    else if key<k then newentry::entry::entries
                    else entry::update entries
      in dict(update entrylist)
      end

end   {of dictionary}
```

After the declaration is evaluated, five identifier bindings are reported, and recorded in the top-level environment. They consist of the type binding of dictionary, the exception binding of lookup, and three value bindings with their types:

```
nulldict : 'a dictionary
lookup : int -> 'a dictionary -> 'a
enter : int * 'a -> 'a dictionary -> 'a dictionary
```

The layered pattern construct "as" was first introduced in HOPE, and yields both brevity and efficiency. The discerning reader may be able to find one further use for it in the declaration.

2. The bare language

2.1 Discussion

It is convenient to present the language first in a bare form, containing enough on which to base the semantic description given in Section 3. Things omitted from the bare language description are:

(1) Derived syntactic forms, whose meaning derives from their equivalent forms in the bare language (Section 6);

(2) Directives for introducing infix identifier status (Section 4);

(3) Standard types (Section 5);

(4) References and equality (Section 7);

(5) Type checking (Section 9).

The principal syntactic objects are expressions and declarations. The composite expression forms are application, type constraint, tupling, raising and handling exceptions, local declaration (using let) and function abstraction.

An important subclass of expressions are the varstructs; they are essentially expressions containing only variables and value constructors, and are used as patterns to create value bindings. Declarations may declare value variables (using value bindings), types with associated constructors or operations (using type bindings), and exceptions (using exception bindings). Apart from this, one declaration may be local to another (using local), and a sequence of declarations is allowed as a single declaration.

An ML program is a series of declarations, called top-level declarations,

dec1 ; .. decn ;

each terminated by a semicolon (where each deci is not itself of the form "dec ; dec'"). In evaluating a program, the bindings created by dec1 are reported before dec2 is evaluated, and so on. In the complete language, an expression occurring in place of any deci is an abbreviated form (see Section 6.2) for a declaration binding the expression value to the variable "it"; such expressions are called top-level expressions.

The bare syntax is in Section 2.8 below; first we consider lexical matters.

2.2 Reserved words

The following are the reserved words used in the complete language. They may not (except =) be used as identifiers. In this document the alphabetic reserved words are always underlined.

abstype and andalso as case do else end
escape exception fun handle if in infix
infixr let local nonfix of op orelse raise
rec then trap type val with while

( ) [ ] , : ; . | = _ ?

2.3 Special constants

The unique object of type unit is denoted by the special constant ().

An integer constant is any non-empty sequence of digits, possibly preceded by ~ representing negation.

A string constant is a sequence of zero or more printable characters or spaces enclosed between quotes ("), but within which any quote symbol is preceded by the escape character \ . Use of \ in strings also has meaning as follows:

\1..\9	One to nine spaces	\E	Escape
\0	Ten spaces	\N	Null (Ascii 0)
\C	Carriage return	\D	Del (Ascii 127)
\L	Line feed	\^c	Ascii control character c
\T	Tabulation	\c	c (any other character)
\B	Backspace		

2.4 Identifiers

Identifiers are used to stand for five different syntax classes which, if we had a large enough character set, would be disjoint:

value variables	(var)
value constructors	(con)
type variables	(tyvar)
type constructors	(tycon)
exception identifiers	(exid)

An identifier is either alphanumeric: any sequence of letters, digits, primes (') and underbars (_) starting with a letter or prime, or symbolic: any sequence of the following symbols

! # % & $ + - / : < = > ? @ \ ~ ` ^ | *

In either case, however, reserved words are excluded. This means that for example ? and | are not identifiers, but ?? and |=| are identifiers. The only exception to this rule is that the symbol =, which is a reserved word, is also allowed as an identifier to stand for the equality predicate (see Section 7.2). The identifier = may not be rebound; this precludes any syntactic ambiguity.

A type variable (tyvar) may be any alphanumeric identifier starting with a prime. The other four classes (var, con, tycon, exid) are represented by identifiers not starting with a prime. Thus type variables are disjoint from the other four classes. Otherwise, the syntax class of an occurrence of identifier id is determined thus:

(1) In types, id is a type constructor, and must be within the scope of the type binding which introduced it.

(2) Following exception, raise or handle id is an exception identifier.

(3) Elsewhere, id is a value constructor if it occurs in the scope of a type binding which introduced it as such, otherwise it is a value variable.

It follows from (3) that no value binding can make a hole in the scope of a value constructor by introducing the same identifier as a variable, since this identifier must stand for the constructor in any varstruct which lies in the scope of the type declaration by which this constructor was introduced. In fact, by means of a syntactic restriction (see Section 10(8)), we ensure that the scopes of a type constructor and of its associated value constructors are identical.

The syntax-classes var, con, tycon and exid all depend on which bindings are in force, but only the classes var and con are necessarily disjoint. The context determines (as described above) to which class each identifier occurrence belongs.

In the complete language, an identifier may be given infix status by the infix or infixr directive; this status only pertains to its use as a var or a con. If id has infix status, then "exp1 id exp2" (resp. "vs1 id vs2") may occur wherever the application "id(exp1,exp2)" (resp. "id(vs1,vs1)") would otherwise occur. On the other hand, non-infixed occurrences of id must be prefixed by the keyword "op". Infix status is cancelled by the nonfix directive.

2.5 Comments

A comment is any character sequence within curly brackets {} in which curly brackets are properly nested. An unmatched occurrence of } is faulted by the compiler.

2.6 Lexical analysis

Each item of lexical analysis is either a reserved word or a special constant or an identifier; comments and non-visible characters separate items (except spaces within string constants) and are otherwise ignored. At each stage the longest next item is taken.

As a consequence of this simple approach, spaces - or parentheses - are needed sometimes to separate identifiers and reserved words. Two examples are

```
   a:= !b      or      a:=(!b)      but not     a:=!b
                                  (assigning contents of b to a)
 ~ :int->int   or   (~):int->int   but not     ~:int->int
                                  (unary minus qualified by its type)
```

Rules which allow omission of spaces in such examples, such as adopted by Cardelli in VAX ML, also forbid certain symbol sequences as identifiers and - more importantly - are hard to remember; it seems better to keep a simple scheme and tolerate a few extra spaces or parentheses.

2.7 Delimiters

Not all constructs have a terminating reserved word; this would be verbose. But a compromise has been adopted; end terminates any construct which declares bindings with local scope. This involves only the let, local and abstype constructs.

2.8 The bare syntax

Conventions: {..} means optional.
For any syntax class s, define s_seq ::= s
(s1,..,sn) (n≥1)
Alternatives are in order of decreasing precedence.
L (resp. R) means left (resp. right) association.
Parentheses may enclose phrases of any named syntax class
defined below or in Appendices 1 and 2.

EXPRESSIONS exp

```
aexp ::=
  var                       (variable)
  con                       (constructor)
  ( exp )

exp ::=
  aexp                      (atomic)
  exp aexp                 L(application)
  exp : ty                 L(constraint) **
  exp1 , .. , expn          (tuple, n≥2)
  raise exid exp            (raise exc'n)
  let dec in exp end        (local dec'n)
  exp handle exid match     (handle exc'n) *
  exp1 ? exp2               (handle exc'ns) *
  fun match                 (function)

match ::=                   (n≥1)
  vs1.exp1 | .. | vsn.expn
```

VARSTRUCTS vs

```
avs ::=
  _                         (wildcard)
  var                       (variable)
  con                       (constant)
  ( vs )

vs ::=
  avs                       (atomic)
  con avs                  L(construction)
  vs : ty                  L(constraint) **
  var{:ty} as vs            (layered)
  vs1 , .. , vsn            (tuple, n≥2)
```

VALUE BINDINGS vb

```
vb ::=
  vs = exp                  (simple)
  vb1 and .. and vbn        (multiple,n≥2)
  rec vb                    (recursive)
```

DECLARATIONS dec

```
dec ::=
  val vb                    (values)
  type tb                   (types)
  abstype tb
      with dec end          (abs. types)
  exception eb              (exceptions)
  local dec1
      in dec2 end           (local dec'n)
  dec1 {;} .. decn {;}      (sequence, n≥0)
```

TYPE BINDINGS tb

```
tb ::=
  {tyvar_seq} tycon
        = constrs           (simple)
  tb1 and .. and tbn        (multiple,n≥2)
  rec tb                    (recursive)

constrs ::=                 (n≥1)
  con1{of ty1} | .. | conn{of tyn}
```

TYPES ty

```
ty ::=
  tyvar                     (type variable)
  {ty_seq} tycon            (type construction)
  ty1 * .. * tyn            (tuple type, n≥2)
  ty1 -> ty2               R(function type)
```

EXCEPTION BINDINGS eb

```
eb ::=
  exid {: ty}               (simple)
  eb1 and .. and ebn        (multiple,n≥2)
```

PROGRAMS: dec1 ; .. decn ;

** The syntax of types binds more tightly than that of expressions, so these forms should be parenthesized if not followed by a reserved word.
* These two forms are of equal precedence and left associative.

3. Evaluation

3.1 Environments and Values

Evaluation of phrases takes place in the presence of an ENVIRONMENT and a STORE. An ENVIRONMENT E has two components: a value environment VE associating values to variables and to value constructors, and an exception environment EE associating exceptions to exception identifiers. A STORE S associates values to references, which are themselves values. (A third component of an environment, a type environment TE, is ignored here since it is relevant only to typechecking and compilation, not to evaluation.)

An exception e, associated to an exception identifier exid in any EE, is an object from which exid may be recovered. A packet p=(e,v) is an exception e paired with a value v, called the excepted value. Neither exceptions nor packets are values. Besides possibly changing S (by assignment), evaluation of a phrase returns a result as follows:

Phrase	Result
Expression	v or p
Value binding	VE or p
Type binding	VE
Exception binding	EE
Declaration	E or p

For every phrase except a handle or "?" expression, whenever its evaluation demands the evaluation of an immediate subphrase which returns a packet p as result, no further evaluation of subphrases occurs and p is also the result of the phrase. This rule should be remembered while reading the evaluation rules below.

A function value f is a partial function which, given a value, may return a value or a packet; it may also change the store as a side-effect. Every other value is either a constant (a nullary constructor), a construction (a constructor with a value), a tuple or a reference.

3.2 Environment manipulation

We may write <(id1,v1)...(idn,vn)> for a value environment, where the idi are distinct. Then <> is the empty VE, and VE1 + VE2 means the VE in which the associations of VE2 supersede those of VE1. Similarly for exception environments. If E=(VE,EE) and E'=(VE',EE'), then E+E' means (VE+VE',EE+EE'), E+VE' means E+(VE',<>), etc. This implies that an identifier may be associated both in VE and in EE without conflict.

3.3 Matching varstructs

The matching of a varstruct vs to a value v either fails or yields a VE. Failure is distinct from returning a packet, but will result in this when all varstructs fail in applying a match to a value (see Section 3.4). In the following rules, if any component varstruct fails to match then the whole varstruct fails to match.

The following is the effect of matching cases of varstruct vs to value v:

_	: the empty VE is returned.
var	: the VE <(var,v)> is returned.
con{vs}	: if v = con{v'} then vs is matched to v', else failure.
var{:ty} as vs	: vs is matched to v returning VE; then <(var,v)> + VE is returned.
vs1,..,vsn	: if v=(v1,...,vn) then vsi is matched to vi returning VEi, for each i; then VE1+...+VEn is returned.
vs:ty	: vs is matched to v.

3.4 Applying a match

Assume environment E. Applying match m = vs1.exp1|..|vsn.expn to value v returns a value or packet as follows:

Each vsi is matched to v in turn, from left to right, until one succeeds returning VEi; then expi is evaluated in E + VEi. If none succeeds, then the packet (ematch,()) is returned, where ematch is the standard exception bound by predeclaration to the exception identifier "match". But matches which may fail are to be detected by the compiler and flagged with a warning; see Section 10(2).

Thus, for each E, a match m denotes a function value.

3.5 Evaluation of expressions

Assume environment E=(VE,EE). Evaluating an expression exp returns a value or packet as follows, by cases of exp:

var	: returns value VE(var).
con	: returns value VE(con).
exp aexp	: exp is evaluated, returning function value f; then aexp is evaluated, returning value v; then f(v) is returned.
exp1,...,expn	: the expi are evaluated in sequence, from left to right, returning vi respectively; then (v1,...,vn) is returned.
raise exid exp	: exp is evaluated, returning value v; then packet (e,v) is returned, where e = EE(exid).
exp handle exid match	: exp is evaluated; if exp returns a value v, then v is returned; if exp returns p = (e,v) then: (1) if e = EE(exid) then match is applied to v, (2) otherwise p is returned.

exp1 ? exp2 : exp1 is evaluated; if exp1 returns a value v, then v is returned; if exp1 returns any packet, then exp2 is evaluated.

let dec in exp end : dec is evaluated, returning E'; then exp is evaluated in E + E'.

fun match : f is returned, where f is the function of v gained by applying match to v in environment E.

exp:ty : exp is evaluated.

3.6 Evaluation of value bindings

Assume environment E = (VE,EE). Evaluating a value binding vb returns a value environment VE' or a packet as follows, by cases of vb:

vs = exp : exp is evaluated in E, returning value v; then vs is matched to v; if this returns VE', then VE' is returned, and if it fails then the packet (ebind,()) is returned, where ebind is the standard exception bound by predeclaration to the exception identifier "bind".

vb1 and..and vbn : vb1, .. ,vbn are evaluated in E from left to right, returning VE1, .. ,VEn; then VE1 + .. + VEn is returned.

rec vb : vb is evaluated in E', returning VE', where E' = (VE+VE',EE). Because the values bound by evaluating vb must be function values (Section 10(4)), E' is well defined by "tying knots" (Landin).

3.7 Evaluation of type bindings

The components VE and EE of the current environment do not affect the evaluation of type bindings (TE affects their type checking and compilation). Evaluating a type binding tb returns a value environment VE' (it cannot return a packet) as follows, by cases of tb:

{tyvar_seq} tycon = con1 {of ty1} | .. | conn {of tyn} : the value environment VE' = <(con1,v1),...,(conn,vn)> is returned, where vi is either the constant value coni (if "of tyi" is absent) or else the function which maps v to coni(v). Note that all other effect of this type binding is handled by the compiler or type-checker, not by evaluation.

tb1 and..and tbn : tb1, .. ,tbn are evaluated from left to right, returning VE1, .. ,VEn; then VE' = VE1 + .. + VEn is returned.

rec tb : tb is evaluated. Note again that the recursion is handled by typechecking only.

3.8 Evaluation of exception bindings

The evaluation of an exception binding eb returns an exception environment EE' as follows, by cases of eb:

exid {:ty} : a new exception e is generated (an object from which the exception identifier exid may be retrieved), and EE' = <(exid,e)> is returned.

eb1 and..and ebn : eb1, .. ,ebn are evaluated from left to right, returning EE1, .. ,EEn; then EE' = EE1 + .. + EEn is returned.

3.9 Evaluation of declarations

Assume environment E = (VE,EE). Evaluating a declaration dec returns an environment E' or a packet as follows, by cases of dec:

val vb : vb is evaluated, returning VE'; then E' = (VE',<>) is returned.

type tb : tb is evaluated, returning VE'; then E' = (VE',<>) is returned.

abstype tb with dec end :
tb is evaluated, returning VE'; then dec is evaluated in E + VE', returning E'; then E' is returned.

exception eb : eb is evaluated, returning EE'; then E' = (<>,EE') is returned.

local dec1 in dec2 end :
dec1 is evaluated, returning E1, then dec2 is evaluated in E + E1, returning E2; then E' = E2 is returned.

dec1 {;} .. decn {;} :
each deci is evaluated in E+E1+ .. +E(i-1), returning Ei, for i = 1,2, .. ,n; then (<>,<>)+E1+ .. +En is returned. Thus when n=0 the empty environment is returned.

Each declaration is defined to return only the new environment which it makes, but the effect of a declaration sequence is to accumulate environments.

3.10 Evaluation of programs

The evaluation of a program

dec1 ; .. decn ;

takes place in the initial presence of the standard top-level environment ENV0 containing all the standard bindings (see Section 5). The top-level environment ENVi, present after the evaluation of deci in the program, is defined recursively as follows: deci is evaluated in ENV(i-1) returning environment Ei, and then ENVi = ENV(i-1)+Ei.

4. Directives

Directives are included in ML as (syntactically) a subclass of declarations. They possess scope, as do all declarations.

There is only one kind of directive in the standard language, namely those concerning the infix status of value variables and constructors. Others, perhaps also concerned with syntactic conventions, may be included in extensions of the language. The directives concerning infix status are:

```
infix{r} {p} id1 ... idn
nonfix id1 ... idn
```

where p is a non-negative integer. The infix and infixr directives introduce infix status for each idi (as a value variable or constructor), and the nonfix directive cancels it. The integer p (default 0) determines the precedence, and an infixed identifier associates to the left if introduced by infix, to the right if by infixr. Different infixed identifiers of equal precedence associate to the left.

While id has infix status, each occurrence of it (as a value variable or constructor) must be infixed or else preceded by op; note that this includes such occurrences within varstructs, even within the varstructs of a match.

Several standard functions and constructors have infix status (see Appendix 3) with precedence; these are all left associative except "::".

It may be thought better that the infix status of a variable or constructor should be established in some way within its binding occurrence, rather than by a separate directive. However, the use of directives avoids problems in parsing.

The use of local directives (introduce by let or local) imposes on the parser the burden of determining their textual scope. A quite superficial analysis is enough for this purpose, due to the use of end to delimit local scopes.

5. Standard bindings

The bindings of this section constitute the standard top-level environment ENV0.

5.1 Standard type constructors

The bare language provides the function-type constructor, ->, and for each n $\geq$ 2 a tuple-type constructor *n. Type constructors are in general postfixed in ML, but -> is infixed, and the n-ary tuple-type constructed from ty1, .., tyn is written "ty1 * .. * tyn". Besides these type constructors, the following are standard:

Type constants (nullary constructors)	: unit,bool,int,string
Unary type constructors	: list,ref

The constructors unit, bool and list are fully defined by the following assumed declaration

```
infixr 30 ::
type unit = () and bool = true | false
and rec 'a list = nil | op :: of 'a * 'a list
```

The word "unit" is chosen since the type contains just one value; this is why it is preferred to the word "void" of ALGOL 68. Note that it is also (up to isomorphism) a unit for type tupling, though we do not exploit this isomorphism by allowing a coercion between the types ty and ty * unit .

The type constant int is equipped with constants 0, 1, ~1, 2, .. . The type constant string is equipped with constants as described in Section 2.3. The type constructor ref is for constructing reference types; see Section 7.

Real numbers will be defined as a standard type in a later extension.

5.2 Standard functions and constants

All standard functions and constants are listed in Appendix 3. There is not a lavish number; we envisage function libraries provided by each implementation, together with the equivalent ML declaration of each function (though the implementation may be more efficient). In time, some such library functions may accrue to the standard; a likely candidate for this is a group of array-handling functions, grouped in a standard declaration of the unary type constructor "array".

Most of the standard functions and constants are familiar, so we need mention only a few critical points:

(1) explode yields a list of strings of size 1; implode is iterated string concatenation (^). ord yields the Ascii code number of the first character of a string; chr yields the Ascii character (as a string of size 1) corresponding to an integer.

(2) ref is a monomorphic function, but in varstructs it may be used polymorphically, with type 'a ->'a ref .

(3) The list destructors hd and tl, the character functions ord and chr, and the five arithmetic operators *, div, mod, + and - , may generate standard exceptions (see Section 5.3) whose identifier in each case is the same as that of the function. This occurs for hd and tl when the argument is nil; for ord when the string is empty; for chr when the character is undefined; and for the others when the result is out of range or, for div and mod, when the second argument is zero.

(4) div and mod are defined as in PASCAL.

5.3 Standard exceptions

All predeclared exception identifiers are of type unit. There are two special ones, match and bind; these exceptions are raised on failure of matching or binding, as explained in Sections 3.4 and 3.6 respectively. Note, however, that these exceptions cannot occur unless the compiler has given a warning, as detailed in Section 10(2),(3), except in the case of a top-level declaration as indicated in 10(3).

The only other predeclared exception identifiers are

```
hd  tl  ord  chr  *  div  mod  +  -
```

These are the identifiers of standard functions which are ill-defined for certain arguments, as detailed in Section 5.2. For example, using the derived form trap explained in Section 8.2, the expression

```
3 div x trap div 10000
```

will return 10000 when x = 0. Also the function hd, together with its exception identifier, could have been defined in ML as follows (using some derived forms introduced in Section 6):

```
exception hd : unit
val hd(nil) = escape hd
  | hd(x::l) = x
```

6. Standard Derived Forms

6.1 Expressions and Varstructs

DERIVED FORM	EQUIVALENT FORM
Expressions :	
escape exid	raise exid ()
exp1 trap exid exp2	exp1 handle exid _.exp2
case exp of match	(fun match) exp
if exp then exp1 else exp2	case exp of true.exp1 | false.exp2
exp1 orelse exp2	if exp1 then true else exp2
exp1 andalso exp2	if exp1 then exp2 else false
exp1 ; exp2	case exp1 of _.exp2
while exp1 do exp2	let val rec f = fun (). if exp1 then (exp2;f()) else () in f() end
[exp1 ; .. ; expn]	exp1:: .. ::expn::nil (n≥0)
Varstructs :	
[vs1 ; .. ; vsn]	vs1:: .. ::vsn::nil (n≥0)

The derived form may be implemented more efficiently than its equivalent form, but must be precisely equivalent to it semantically. The type-checking of each derived form is also defined by that of its equivalent form.

The binding power of all bare and derived forms is shown in Appendix 1. Note particularly that a semicolon, whether used in declaration sequencing, in expression sequencing or in the derived list form, always has weakest binding power.

The escape form is only admissible with exceptions of type unit, and the trap form is useful both in this case and when the excepted value is immaterial.

6.2 Bindings and Declarations

DERIVED FORM	EQUIVALENT FORM

Value bindings :

```
var avs11 .. avs1n {:ty} = exp1
  | ..
  | var avsm1 .. avsmn {:ty} = expm
```

```
var = fun x1. .. fun xn.
   case (x1, .., xn) of
   ( avs11, .., avs1n . exp1 {:ty}
   | ..
   | avsm1, .., avsmn . expm {:ty} )
```

.. where the xi are new, and m,n≥1.

Declarations :

exp	val it = exp
type tb	type tb' dec'
abstype tb with dec end	abstype tb' with local dec' in dec end end

.. where tb',dec' are described below.

The derived value binding allows function definitions, possibly Curried, with several clauses. The first derived declaration is only allowed at top-level, for treating top-level expressions as degenerate declarations; "it" is just a normal value variable.

The last two derived declarations are concerned with a generalisation of type bindings. In place of the bare syntax for constructions

```
constrs ::= con1 {of ty1} | .. | conn {of tyn}
```

a more general form is allowed in the full language, as follows:

```
constrs ::= con1 {of compt_seq1} | .. | conn {of compt_seqn}
compt ::= {var :} ty
```

The effect is to allow selector functions to be declared in the same phrase as constructor functions. (They can always be declared later, as illustrated for the selector hd in Section 5.3, but the present derived form makes the declaration simpler to write and easier to implement efficiently.) Before seeing the derivation of the general form from the bare language, it is helpful to look at the simple example of defining the type constructor list complete with its selectors hd and tl, which together form the inverse of :: . The derived declaration is

```
type rec 'a list = nil | op :: of (hd:'a, tl:'a list)
```

This is equivalent, by the general rule below, to the following declaration sequence:

```
type rec 'a list = nil | op :: of 'a * 'a list
exception hd:unit   val hd(x::l) = x | hd(_) = escape hd
exception tl:unit   val tl(x::l) = l | tl(_) = escape tl
```

Unlike other derived forms, the derived type binding is not by itself equivalent to a phrase of the bare language, but any declaration containing the derived binding is equivalent to an expanded declaration in the bare language. To obtain this we first define, for a derived type binding tb, a bare type binding tb' and an auxiliary declaration sequence dec'. Consider any construction

```
con of ({var1:}ty1, .. ,{varn:}tyn)
```

occurring within the right side of any simple binding within tb (the parentheses could be absent if n=1). Then in the bare type binding tb' the corresponding construction is

```
con of ty1 * .. * tyn
```

Also, for each vari present in the construction, the auxiliary declaration dec' will contain the subsequence

```
exception vari:unit
val vari(con(x1,..,xn)) = xi | vari(_) = escape vari
```

Note that the selectors thus introduced may raise exceptions identified by their own names. However, these exceptions cannot occur when con is the unique constructor of a simple binding; in this case the exception declaration and the second clause of the val declaration (which would be redundant) are omitted from dec'.

Having defined tb' and dec' for any derived type binding tb, we can now expand any type or abstype declaration of tb into a bare language declaration, as shown in the table above.

7. References and equality

7.1 References and assignment

Following Cardelli, references are provided by the type constructor "ref". Since we are sticking to monomorphic references, there are two overloaded functions available at all monotypes mty:

(1) ref : mty -> mty ref, which associates (in the store) a new reference with its argument value. "ref" is a constructor, and may be used polymorphically in varstructs, with type 'a -> 'a ref .

(2) op := : mty ref * mty -> unit , which associates its second (value) argument with its first (reference) argument in the store, and returns () as result.

The polymorphic contents function "!" is provided, and is equivalent to the declaration "val !(ref x) = x" .

7.2 Equality

The overloaded equality function op = : ety * ety -> bool is available at all types ety which admit equality, according to the definition below. The effect of this definition is that equality will only be applied to values which are built up from references (to arbitrary values) by value constructors, including of course value constants. On references, equality means identity; on objects of other types ety, it is defined recursively in the natural way.

The types ety which admit equality are therefore defined as follows:

(1) A type ty admits equality iff it is built from arbitrary reference types by type constructors which admit equality.

(2) The standard type constructors *n, unit, int, bool, string and list all admit equality.

Thus for example, the type (int * 'a ref)list admits equality, but (int * 'a)list and (int -> bool)list do not.

A user-defined type constructor tycon, declared by a type binding tb whose bare form is

{tyvar_seq} tycon = con1 {of ty1} | .. | conn {of tyn}

admits equality within its scope (but, if declared by abstype, only within the with part of its declaration) iff it satisfies the following condition:

(3) Each construction type tyi in this binding is built from arbitrary reference types and type variables, either by type constructors which already admit equality or (if tb is within a rec) by tycon or any other type constructor declared by mutual recursion with tycon, provided these other type constructors also satisfy the present condition.

The first paragraph of this section should be enough for an intuitive understanding of the types which admit equality, but the precise definition is given in a form which is readily incorporated in the type-checking mechanism.

8. Exceptions

8.1 Discussion

Some discussion of the exception mechanism is needed, as it goes a little beyond what exists in other functional languages. It was proposed by Alan Mycroft, as a means to gain the convenience of dynamic exception trapping without risking violation of the type discipline (and indeed still allowing polymorphic exception-raising expressions). Brian Monahan also put forward a similar idea.

The rough and ready rule for understanding it is as follows. If an exception is raised by a raise expression

raise exid exp

which lies in the textual scope of a declaration of the exception identifier exid, then it may be handled by a handle expression

handle exid match

only if this expression is in the textual scope of the same declaration. If it is not so handled, then it may only be caught by the universal handler "?".

This rule is perfectly adequate for exceptions declared at top level; some examples in 8.3 below illustrate what may occur in other cases.

8.2 Derived forms

A handle expression is a double filter for exception packets. First, it handles only those packets (e,v) for which e is the exception bound to its exception identifier; second, its match may discriminate upon v, the excepted value.

A case which is likely to be frequent is when no discrimination on v is required; in this case, the derived expression

exp1 trap exid exp2

is appropriate for handling. Further, exceptions of type unit may be raised by the derived expression

escape exid

since the only possible excepted value is ().

These two derived forms differ from those in earlier drafts of this proposal, which rested upon the assumptions of a single predeclared exception identifier of type string. Don Sannella convinced me that the present forms are more elegant and useful; they also harmonise well with the predeclared exceptions of Section 5.3.

8.3 Some examples

The evaluation rules for raise and handle expressions are designed to ensure (with the help of type-checking) that any exception packet which is handled by the expression

handle exid match

will contain an excepted value of the type expected by the match. Consider a simple example:

```
exception exid : bool;
val f(x) =
     let exception exid:int in
          if x > 100 then raise exid x else x+1
     end;
f(200) handle exid (true.500 | false.1000);
```

The program is well-typed, but useless. The exception bound to the outer exid is distinct from that bound to the inner exid; thus the exception raised by f(200), with excepted value 200, could only be handled by a handle expression within the scope of the inner exception declaration - it will not be handled by the handle expression in the program, which expects a boolean value. So this exception will just explode at top level. This would apply even if the outer exception declaration were also of type int; the two exceptions bound to exid would still be distinct.

On the other hand, if the last line of the program is changed to

```
f(200) ? 500 ;
```

then the exception will be caught, and the value 500 returned. The universal exception handler "?" catches any exception packet, even one exported from the scope of the declaration of the associated exception identifier, but cannot examine the excepted value in the packet, since the type of this value cannot be statically determined.

Even a single textual exception binding - if for example it is declared within a recursively defined function - may bind distinct exceptions to the same identifier. Consider another useless program:

```
val rec f(x) =
     let exception exid in
          if p(x) then a(x) else
          if q(x) then f(b(x)) trap exid c(x) else
          raise exid d(x)
     end;
f(v);
```

Now if p(v) is false but q(v) is true, the recursive call will evaluate f(b(v)). Then, if both p(b(v)) and q(b(v)) are false, this evaluation will raise an exid exception with excepted value d(b(v)). But this packet will not be handled, since the exception of the packet is that which is bound to exid by the inner - not outer - evaluation of the exception declaration.

These pathological examples should not leave the impression that exceptions are hard to use or to understand. The rough and ready rule of Section 8.1 will almost always give the correct understanding.

9. Type-checking

The type-checking discipline is exactly as in original ML, and therefore need only be described with respect to new phrases.

In a match "vs1.exp1 | .. | vsn.expn", the types of all vsi must be the same (ty say), and if variable var occurs in vsi then all free occurrences of var in expi must have the same type as its occurrence in vsi. In addition, the types of all the expi must be the same (ty' say). Then ty->ty' is the type of the match.

The type of "fun match" is the type of the match. The type of "exp handle exid match" is ty', where exp has type ty', match has type ty->ty', and exid has type ty. The type of "raise exid exp" is arbitrary, but exp and exid must have the same type. Thus the type of an exception may be polymorphic; exid is only required to have the same type at all occurrences within the scope of its declaration (and this must be an instance of any type qualifying the declaration).

A type variable is only explicitly bound (in the sense of variable-binding in lambda-calculus) by its occurrence in the tyvar_seq in the type binding "{tyvar_seq} tycon = constrs", and then its scope is "constrs". This means for example that bound uses of 'a in both tb1 and tb2 in the type binding "tb1 and tb2" bear no relation to each other.

Otherwise, repeated occurrences of a (free) type variable may serve to link explicit type constraints. The scope of such a type variable is the top-level declaration or expression in which it occurs.

The type-checker refers to the type environment (TE) component of the environment, and records its findings there. Details of TE are not given in this report; they are compatible with what is done in current ML implementations, except that value constructors (and their types) are associated with the type constructors to which they belong.

Type-discipline

The type-checking discipline is exactly as in original ML, and therefore need only be described with respect to new phrases.

In a match "val.exp | ... | van.exp", the types of all val must be the same ("t say)} and if variable van occurs in val then all free occurrences of van in exp must have the same type as its occurrences in val. In addition, the type of all the exp must be the same ("t' say). Then "t->t'" is the type of the match.

The type of "fun match" is the type of the match. The type of "exp where rec.sub" is "t', where sub has type "t->t', match has type "t->t', and exp has type t'. The type of "value exp" is arbitrary, but exp and exp must have the same type. Thus the type of an exception may be polymorphic; exn is only required to have the same type at all occurrences within the scope of its declaration (and this must be an instance of any type qualifying the declaration).

A type variable is only explicitly bound (in the sense of variable-binding in lambda-calculus) by its occurrence in the tyvar_seq in the type binding "{tyvar_seq} tycon = constr". and then its scope is "constr". This means for example that bound uses of 't in both tp1 and tp2 in the type binding "tp1 and tp2" bear no relation to each other.

Otherwise, repeated occurrences of a (free) type variable may serve to link explicit type constraints. The scope of such a type variable is the top-level expression or expression in which it occurs.

The type-checker refers to the type environment (TE) together with the environment and records its findings there. Details of TE are not given in current ML phrases; they are comparible with what is done in current ML implementations, except that value constructors (and their types) are associated with the type constructors to which they belong.

10. Syntactic restrictions

(1) No varstruct may contain two occurrences of the same variable.

(2) In a match "vs1.exp1 | .. | vsn.expn", the varstruct sequence vs1, .., vsn should be irredundant and exhaustive. That is, each vsj must match some value (of the right type) which is not matched by vsi for any i<j, and every value (of the right type) must be matched by some vsi. The compiler must give warning on violation of this restriction, but should still compile the match. The restriction is inherited by derived forms; in particular, this means that in the Curried function binding "var avs1 .. avsn {:ty} = exp" (consisting of one clause only), each separate avsi should be exhaustive by itself.

(3) For each value binding "vs = exp" the compiler must issue a report (but still compile) if either vs is not exhaustive or vs contains no variable. This will (on both counts) detect a mistaken declaration like "val nil = exp" in which the user expects to declare a new variable nil (whereas the language dictates that nil is here a constant varstruct, so no variable gets declared). Cardelli points out this danger.

However, these warnings should not be given when the binding is a component of a top-level declaration val vb ; e.g. "val x::l = exp1 and y = exp2" is not faulted by the compiler at top level, but may of course generate an escape with exception identifier "match" (see Section 3.4).

(4) For each value binding "vs = exp" within rec, exp must be of the form "fun match". (The derived form of value binding given in Section 6.2 necessarily obeys this restriction.)

(5) In every instance of "{tyvar_seq} tycon" the tyvar_seq must contain no type variable more than once. The right hand side of a simple type binding may contain only the type variables mentioned on the left.

(6) In "let dec in exp end" and "local dec in dec' end" no type constructor exported by dec may occur in the type of exp or in the type of any variable or value constructor exported by dec'.

(7) Every global exception binding - that is, not localised either by let or by local - must be explicitly constrained by a monotype.

(8) If a type constructor tycon' is declared within the scope of a type constructor tycon, then (a) if tycon and tycon' are distinct identifiers, then their value constructors must be disjoint; (b) if tycon and tycon' are the same identifier, then the value constructors of the outer declaration are not accessible in the scope of the inner declaration, whether or not the inner and outer declarations employ the same identifier(s) as a value constructor(s). These constraints ensure that the scope of a type constructor is identical with the scopes of its associated value constructors.

11. Conclusion

Many points in this design were ardently debated, and it might be thought worthwhile to end with a detailed discussion of the reasons for each particular choice. But it seems rather better - and certainly simpler - to let the result speak for itself, and to allow infelicities (which presumably exist) to emerge during the next year or so in the course of experience with implementation and use.

It would be reasonable after such a period to collect the reactions and to publish a list of corrections - just those which can be agreed among several seriously concerned implementers and users.

Besides these corrections there will clearly be extensions - design ideas which use the present language as a platform. It will be important to keep these two developments separate as far as possible. Corrections should be few and preferably done at most once; extensions may be many, but need not impair the identity of the present language.

APPENDIX 1

SYNTAX : EXPRESSIONS and VARSTRUCTS

(See Section 2.8 for conventions)

```
aexp ::=
       {op} var                              (variable)
       {op} con                              (constructor)
       [ exp1 ; .. ; expn ]                  (list)
       ( exp )

exp ::=
       aexp                                  (atomic)
       exp aexp                             L(application)
       exp : ty                             L(constraint) **
       exp1 id exp2                          (infixed application)
       exp1 andalso exp2                     (conjunction)
       exp1 orelse exp2                      (disjunction)
       exp1 , .. , expn                      (tuple, n≥2)
       raise exid exp                        (raise exception)
       escape exid                           (raise unit exception)
       if exp then exp1 else exp2            (conditional)
       while exp1 do exp2                    (iteration)
       let dec in exp end                    (local declaration)
       case exp of match                     (case expression)
       exp handle exid match                 (handle exception) *
       exp1 trap exid exp2                   (handle exception,discarding value) *
       exp1 ? exp2                           (handle all exceptions) *
       fun match                             (function)
       exp1 ; exp2                           (sequence)

match ::=                                    (n≥1)
       vs1.exp1 | .. | vsn.expn


avs ::=
       _                                     (wildcard)
       {op} var                              (variable)
       con                                   (constant)
       [ vs1 ; .. ; vsn ]                    (list, n≥0)
       ( vs )

vs ::=
       avs                                   (atomic)
       {op} con avs                         L(construction)
       vs : ty                              L(constraint) **
       vs1 con vs2                           (infixed construction)
       var {: ty}  as vs                     (layered)
       vs1 , .. , vsn                        (tuple, n≥0)
```

* These three forms are of equal precedence and left associative.

** The syntax of types binds more tightly than that of expressions, so these forms should be parenthesized if not followed by a reserved word.

APPENDIX 2

SYNTAX : TYPES, BINDINGS, DECLARATIONS and PROGRAMS

(See Section 2.8 for conventions)

```
ty ::=
       tyvar                                          (type variable)
       {ty_seq} tycon                                 (type construction)
       ty1 * .. * tyn                                 (tuple type, n≥2)
       ty1 -> ty2                                    R(function type)

vb ::=
       vs = exp                                       (simple)
       {op} var avs11...avs1n {:ty} = exp1            (clausal function) *
         |  ..
         | {op} var avsm1 .. avsmn {:ty} = expn       (m,n≥1)
       vb1 and .. and vbn                             (multiple, n≥2)
       rec vb                                         (recursive)

tb ::=
       {tyvar_seq} tycon = constrs                    (simple)
       tb1 and .. and tbn                             (multiple, n≥2)
       rec tb                                         (recursive)

constrs ::=                                           (n≥1)
       con1 {of compt_seq1} | .. | conn {of compt_seqn}

compt ::=
       {var :} ty

eb ::=
       exid {:ty}                                     (simple)
       eb1 and .. and ebn                             (multiple, n≥2)

dec ::=
       val vb                                         (value declaration)
       type tb                                        (type declaration)
       abstype tb with dec end                        (abstract type declaration)
       exception eb                                   (exception declaration)
       local dec1 in dec2 end                         (local declaration)
       exp                                            (top-level only)
       dir                                            (directive)
       dec1 {;} .. decn {;}                           (declaration sequence, n≥0)

dir ::=
       infix{r} {p} id1 .. idn                        (declare infix status, p≥0)
       nonfix id1 .. idn                              (cancel infix status)
```

PROGRAMS: dec1 ; .. decn ;

* If var has infix status then op is required in this form; alternatively var may be infixed in any clause. Thus at the start of any clause:

  op var (avs,avs') may be written : (avs var avs')

  where the parentheses may also be dropped if "=" follows immediately.

APPENDIX 3

PREDECLARED VARIABLES and CONSTRUCTORS

nonfix

```
nil         : 'a list
hd          : 'a list -> 'a
tl          : 'a list -> 'a list
map         : ('a->'b) -> 'a list
                    -> 'b list
rev         : 'a list -> 'a list

true        : bool
false       : bool
not         : bool -> bool
~           : int -> int

size        : string -> int
chr         : int -> string
ord         : string -> int
explode     : string -> string list
implode     : string list -> string

ref         : mty -> mty ref
!           : 'a ref -> 'a
```

Special constants:

```
()          : unit
0,1,~1,2,.. : int
"---"       : string
```

infix

Precedence 50:

```
*   : int * int -> int
div:  "     "     "
mod:  "     "     "
```

Precedence 40:

```
+   :  "     "     "
-   :  "     "     "
^   : string * string -> string
```

Precedence 30:

```
::  : 'a * 'a list -> 'a list
@   : 'a list * 'a list
                    -> 'a list
```

Precedence 20:

```
=   : ety * ety -> bool
<>  :  "     "     "
<   : int * int -> bool
>   :  "     "     "
<=  :  "     "     "
>=  :  "     "     "
```

Precedence 10:

```
o   : ('b->'c) * ('a->'b)
                  -> ('a->'c)
:=  : mty ref * mty -> unit
```

Notes:

(1) The following are constructors, and thus may appear in varstructs:

nil true false ref ::
.. and all special constants.

(2) mty denotes any monotype, and ety (as explained in Section 7.2) denotes any type admitting equality.

(3) Infixes of higher precedence bind tighter. "::" associates to the right; otherwise infixes of equal precedence associate to the left.

(4) The meanings of these predeclared bindings are discussed in Section 5.2.