A PROPOSAL FOR STANDARD ML  

1. Introduction

The language proposed here—called here “Standard ML” but a better name may be found—is not supposed to be novel. Its aims are

(i) to remove some redundancies and bad choices in the original design of ML;
(ii) to “round out” ML in one particular respect—namely, the use of patterns in parameter passing, not only for the standard data constructors (pairing, lists) but also for constructors which are user defined;
(iii) to make sure that just enough input/output is standardised to allow serious work to be done (the user should be able to define enough higher-level I/O functions within the language that he feels no pressing need to extend it);
(iv) thereby to determine as clearly as possible—for the benefit of the user community—what is guaranteed in ML.

This standardisation is conservative in many respects: little attention paid to wide programming environment issues (editing etc.); no lazy evaluation; no attempt to generalise escape trapping to allow other than tokens as escape values; no use of a more general typechecking scheme to allow polymorphic assignments (or indeed to allow other-than-token escape values); no use of
record and variant types à la Cardelli; no attempt to introduce the idea of working in 'persistent' (and possibly undeletable) environments.

None of these things is considered wrong: far from it. But a well-rounded language seems to emerge without them; this means that future extensions can be examined against a firm background. It should then be easier to see what limitations are, or are not, inherent in the ML kind of functional language.

The main inspirations towards this standardisation are from Luca Cardelli — many of whose ideas are adopted — and from HOPE (Burstell et al) which, in my view, besides the most natural rounding-out of ML w.r.t. data types and data parameters.

A word about environment operators: as Standard ML develops, it emerges that Luca's important environment operator "enc" (such that in "decl enc decl2" decl exports into decl2 and both decl1 and decl2 export from the whole) gets entirely conflated with the semantic — which has always had the meaning of enc when it occurs between top-level declarations. With one other minor shift, it then turns out that every top-level command sequence is just a single declaration! This has a pleasant unifying effect — for example, external ML files can be imported either as global or as local declarations; the latter has important uses.
2. **Bare ML**

We first give a bare version of Standard ML which omits (i) convenient alternative syntactic forms, (ii) infixes, (iii) references and assignment, and — quite importantly — (iv) all standard types.

A strong point in favour of the data declarations (inherited from HOPE) is that all our standard types — unit, bool, int and token — can in principle be defined. Of course, they will probably be implemented in a special way. But, by omitting them to begin with, we show how the language is largely independent of them. For example, it is only later (via typechecking constraints) that escape values are required to be tokens; a different extension of Bare ML could make a different choice.

Possibly the only innovation in Bare ML is the function abstraction form

```ml
fun v1, exp1 | ... | vn, expn
```

which generalises the familiar lambda abstraction "\( \lambda \nu. \exp \)". When applied, this "function" matches its argument to each variable \( \nu \) in left-to-right order until a match succeeds (which binds any variables in \( \nu \) to components of the argument value) then evaluates the corresponding \( \exp_i \). If no match is found, the application escapes with a determined value (which, in the standard language, will be the token `failmatch`). This construct — which is something like a lunatic proposal by Alan Mycroft (though he may disagree!) — cannot nicely be defined in terms of elementary lambda-abstraction by iterated escape trapping.
since it is vital to distinguish - say - a failure to match \( n \) from an escape generated by evaluation of \( \text{exp1} \). (This latter will escape from the entire application.) Note that \( n = 1 \) gives an ordinary function abstraction, though still with varstruct matching.

The elementary syntax classes are given in Table 1. For identifiers, we follow Luca Cardelli more or less. But it seems robust to exclude reserved words (including certain symbol sequences) from being identifiers. Since data constructors can appear in varstructs, they cannot be re-used as bound variables within the scope of the data declaration which introduced them. We seize the opportunity of discarding \( *, ** \), etc as type variables, and propose \( a, b, ... \) pronounced as Greek letters.

The syntax of Bare ML is in Table 2. Note that no construct is mentioned which (like conditionals from some host) presupposes any standard type. We discuss the construction class by class.

1) Expressions: In application \( \text{exp1 \ exp2} \) no order of evaluation is assumed.

In original ML, \( \text{exp1} \) was evaluated (to a function) before \( \text{exp2} \). In Cardelli's VAX ML, apparently \( \text{exp2} \) is evaluated first — and this seems to work better in his abstract machine, making function application very efficient (if I understand him right). Of course the order of evaluation chosen in an implementation can be detected by escape-trapping — and this could be exploited in multiple applications like \( f \ \text{exp1} \ ... \ \text{expn} \); but it still seems worth giving the implementor freedom. Likewise in \( \text{exp1} \ \text{exp2} \) no order of evaluation is assumed.

In "escape \( \text{exp} \)" \( \text{exp} \) will (in the standard language) be token-valued. "Escape" seems better than "fail". In \( \text{exp \ trap \ u1,exp1, low,expn} \) the \( u1 \) will (in standard ML) be of type token. We could have chosen \( \text{exp1} \ \text{exp1} \ \text{exp2} \),
where \( \text{exp2} \) is of type "token"; but then it is urksome to have to specify how escapes in evaluating \( \text{exp2} \) are treated. As it is, no order of evaluation question arises — and one can always get the effect (mostly) of "\( \text{exp1 \_trap \_exp2} \)" by writing "\( \text{exp1 \_trap \_t.} (\text{exp2 \_t}) \)"

In "let \( \text{dec \_in \_exp} \)" dec is evaluated first. Note that this is now the only use of "let"; top-level declarations will (usually) start with "var", "data" or "abstract". This differs both from DEC10 ML and from VAX ML, but I think it is justified; see the later discussion on declarations.

We have already discussed "fun match". The only extra point to be added is that — although we cannot readily define the new form in terms of simple "\( \lambda v. \text{exp} \)" — we can probably implement it well enough in terms of the closures of Luca's abstract machine FHM, using traps.

(2) Varstructs: The wild card "any" — matching anything — is not strictly necessary, since "()" can play this role in varstructs without losing any power. But this double use can be mildly confusing, while any is more or less self-explanatory.

Note that the use of user-defined constructors in varstructs gives crucial extra power to the language (Luca added varstructs for his records and variants, and here the constructors play the same role).

A constant \( c \) is really a constructor of unit (see Table 4 defining the data type unit, which Luca called void); but we don't want the pedantry of writing \( c() \) instead of \( c \) — in varstructs or in expressions.
A constructor is formally of one argument, but for a 'binary'
constructor \texttt{cons} (say) the forms \texttt{"cons v"}, \texttt{"cons(v,v)"}, \texttt{"cons any"}
and \texttt{"cons (any, any)"} are all admissible variants — but not \texttt{"cons"} by
itself.

(3) \textbf{Types} : The omission of disjoint sum is discussed in \texttt{Table 4}.
As discussed under infixes later, we propose that all type constructors (except
\# and \texttt{\rightarrow}) will be postfixed. This avoids the need for separate infix
notations for identification in types and in expressions, which seems of
slight value.

(4) \textbf{Declarations} : I have aimed to give all the power of Luca's various
environment operators, but with considerably more restriction on the
possible forms — which I believe will lead to easier understanding. This
is probably the most debatable part of \texttt{BareML}; I think the choices
I have made will lead to convenient and clearly understood programming
style, but then so would other choices! Here are several points

(i) \texttt{"local dec in dec"} plays the role of Luca's \texttt{"inside"}. The prefixed
keyword \texttt{"local"} should help in reading. It corresponds precisely to \texttt{"let"}
in expressions (see above), but the use of a different word tells a
reader that this is a declaration, not an expression.

(ii) Semicolon \texttt{";"} used for sequencing plays the role of Luca's
\texttt{"enc"}. I chose it because (a) it underlines the fact that \texttt{"enc"}
is associative (unlike \texttt{"inside"}); (b) it has the same effect as \texttt{";";
separating top-level declarations commands, and its weak binding power
ensures that at top level \texttt{"dec;dec"} is treated as two separate
commands; (c) it seems the nicest way of writing a sequence of
local declarations each dependent on the last — for example
We have the three forms

<table>
<thead>
<tr>
<th>DEC10 ML</th>
<th>VAN ML</th>
<th>STANDARD ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>let z = x+y</td>
<td>let z = x+y</td>
<td>let var z &lt;- x+y</td>
</tr>
<tr>
<td>in</td>
<td>end w = z*z</td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>in</td>
<td>var w &lt;- z*z</td>
</tr>
</tbody>
</table>

(iii) The restriction of "rec" to qualify only simultaneous bindings seems quite adequate — and avoids the need for Luca's restriction "no inside within a rec". The restriction of "rec" to qualify only whole simultaneous bindings avoids the (notably useless) possibility of "rec" within "rec", and this stratified approach probably makes an implementation easier. Sufficient use of "local in ..." will get round any need for "rec" to qualify part of a simultaneous binding.

(iv) I can see no real need for simultaneous bindings of different kinds, as in "data in ... and var in ...", so have precluded them.

(v) The keyword "var" seems appropriate to match "data" and "abstract".

(vi) I believe that the only use of an abstract binding (isomorphism) is to provide operations defined via the isomorphism, so have forced abstract type declarations to have a with part. Note that, for each type constructor "tycon" introduced in such a binding, the constructor "abstycon" is available in the with part both in expressions and in vstructured; this means that "reptycon" is no longer necessary.

Finally, the locality of data and abstype declaration should be enforced (as in previous ML) by the typechecker ensuring that no value is exported from their scope whose type involves the declared type constructors. This export could be the result of an expression, or by assignment to a reference (but it needs checking whether this latter export is prevented; I believe it is).
(5) **Variable Bindings**: I think it's time to get rid of "=" in variable bindings, so "\(<\)" is used instead. Perhaps we can introduce "be", "is", "are" as synonyms.

(6) **Data Bindings**: It seems worth having the grammatical form e.g. "cons of 'a # 'a list" rather than "cons ('a # 'a list)" as in HOPE, or "cons : 'a # 'a list". Both the latter forms are mild puns, and the last is misleading. The choice of "\(|\)" to separate alternative forms is supposed to reverberate with BNF, and with the syntactic form match used in expressions. To choose "\(,\)" instead would be overloading the comma with too many meanings.

There is no restriction on the types occurring after of (except that all type variables used must occur on the left of "\(<\)""). Thus data types don't have to be "data" all the way down — only of the top level of construction.

(7) **Abstract Bindings**: These are as in DEC10 ML (except for "\(<\)=\)" in place of "\(" =\)", following Luca).

---

*We could have use "\(<\)=\)" as in HOPE, but prefer it to mean "less than or equal to" to ease the transition for PASCAL programmers.*
3. Adding Infixes to BareML

First, the commands

\[
\text{Com} ::= \ldots \quad | \text{infix id}_1 \ldots \text{id}_n \{ \text{r}_1, \ldots, \text{r}_k, \text{ass}_1, \ldots, \text{ass}_n \} \quad | \text{nonfix id}_1 \ldots \text{id}_n
\]

where

- \( r_i ::= 1 \mid 2 \mid \ldots \) (precedence)
- \( \text{ass} ::= \text{left} \mid \text{right} \) (association)

are added. These commands establish the infix status of identifiers in expressions only, not within types. Default values for \( k, \text{ass} \) are 1, left.

The pairing operator "\( \langle \rangle \)" has precedence 1; thus, it binds more loosely than every predefined infix. Infixes bind more loosely than application or type constraints, more tightly than all other expression constructs. Examples:

\[
\begin{align*}
& f \ a + b \text{ means } (f a) + b \\
& a + b \text{ trap } m \text{ means } (a + b) \text{ trap } m
\end{align*}
\]

Second, every non-infixed occurrence of an identifier with infix status must be preceded by "\text{infix}". Infixed occurrences are only allowed in expressions and varstucts, not in data binding constructions.

Third, all infixes must stand for functions of pairs, so

\[
\begin{align*}
& a + b \text{ means } \text{infix} + (a, b) \\
& \text{not } \text{infix} + a b
\end{align*}
\]

See Table 6 for predefined infixes.
Although Luis Damas produced a subtle form of typechecking to allow polymorphic assignment to references, I propose their standard language sticks to monomorphic assignment. The main reason is that the original typechecking discipline has enough pedigree (Curry, Hindley) to deserve the name "Standard" better than any other; it seems wise to keep the language purist this far and no further. Certainly some implementations will want to be more permissive in typechecking (not only for references; another example is in recursively defined functions), and they can declare this explicitly in their documentation. But the example below shows that the effect of polymorphic assignment can be got, in the standard language, at the price of passing polymorphic "assignment functions" as parameters. Also, this is in harmony with the proposed treatment of equality as (mainly) monomorphic; see the end of this section.

Thus, the additions to Bare ML for references are:

1. The type constructor `&ref`.
2. The function `ref : μ → μ ref` at all monotypes `μ`, for creating new references. In variants, however, `ref : d → d ref` may be used polymorphically.
3. The function `infix := : μ ref # μ → unit` at all monotypes, for assignment.

Besides this, the standard contents function `@ : d ref → d`, defined by "`bar @ (ref x) ← x`", is provided.
Example. Here is how to define a row, the type of parametric, one-dimensional arrays. Its operations are parameterised on parametric functions:

\[
\text{new : } d \rightarrow \text{dref}
\]

\[
\text{assign : (dref \times d) \rightarrow unit}
\]

They are:

\[
\text{newrow new : } d \rightarrow \text{int} \rightarrow \text{row}
\]

(“newrow new v k” returns a row of length k with value v in each cell)

\[
\text{assignrow assigni : } d \rightarrow \text{row} \rightarrow \text{int} \rightarrow \text{unit}
\]

(“assignrow assigni v R k” places value v in k\text{th cell of row }R)

\[
\text{controw : } \text{row} \rightarrow \text{int} \rightarrow d
\]

(“controw R k” returns value of k\text{th cell in }R)

\[
\text{abstype } d\text{ row } \leftrightarrow d\text{ref list}
\]

\[
\text{with } \text{newrow ref v k } \leftarrow \text{absrow}(\text{newlist k}) \text{ where } \text{ref newlist} \leftarrow
\]

\[
\text{fun O, nil | k, ref v :: newlist (k-1)}
\]

and \[
\text{assignrow (infix :=) v (absrow L) k } \leftarrow \text{assignlist L k where } \text{ref assignlist} \leftarrow
\]

\[
\text{fun nil, escape "outside" }
\]

\[
| c :: L. \text{fun (l, c := v | k, assignlist L (k-1))}
\]

and \[
\text{controw (absrow L) k } \leftarrow \text{contrlist L k where } \text{ref contrlist} \leftarrow
\]

\[
\text{fun nil, escape "outside" }
\]

\[
| c :: L. \text{fun (l @ c | k, contrlist L (k-1))}
\]

Note that only the underlined parameters are extra to what one could write if polymorphic assignment were allowed. (Just as an abstype may have to be supplied with a polymorphic equality predicate as parameter).
From this, monomorphic arrays are easily set up:

```
abstract introw <--> int row
with newintrow (n:int) <- absintrow o (newrow ref n)
and assignintrow (n:int)(absintrow R) <- assignrow (index :=) n R
and contintrow (absintrow R) <- controw R
```

Note: that here the standard (overloaded) monomorphic "ref" and "=" are supplied.

An example of a freely polymorphic function which uses "@" but not "ref" or "=" is

```
var freeze : (a reflist -> a list) <- map @
```

Equality: Following Castelli's approach, we take the view that

(i) we know what equality on data types (monomorphic) must mean,
(ii) we know that equality of references must mean identity,
(iii) otherwise, we should not determine its meaning.

Hence we allow the infixed predicates =, \( \leftrightarrow \) : \( \Gamma \# \Gamma \rightarrow \text{bool} \)

where \( \Gamma \) is any type built from polymorphic reference types to ref
by data type constructor.

Thus for example, "=" is allowed at type 'a ref list' but not at type 'a list'.

Procedural programming: It seems best to throw out all the weird forms of
looping in DEXO ML, and just extend expressions by

```
exp :: = \ldots | exp1; exp2 (sequencing)
   | while exp1 do exp2 (iteration).
```

Too bad!
5. External Files in Standard ML

By virtue of the abbreviation (see Table 5)

\[ \text{exp} \quad \rightarrow \quad \text{var} \mid \text{my} \leftrightarrow \text{exp} \]

every command sequence is — except for \text{infix}, \text{prefix} commands — a sequence of declarations. In fact, since "\text{is}" is a declaration combinator, \text{it} is just a single declaration. It is therefore appropriate to add the new declaration form

\[ \text{dec} ::= \ldots \mid \text{use} \text{Token} \]

where the \text{Token} is a file-name, to extend the environment by declarations on an external file. There seems no need to restrict \text{use} to top level; non-top-level occurrences may be valuable, e.g.

\[ \text{local use `TREES.ML'} \]

\[ \text{in abstract --- % the abstract type of balanced trees, say %} \]

(Again use is to rename identifiers declared in the file, so as to avoid conflict with the current environment).

Any \text{use} file will be parsed with standard \text{infix} status, any \text{fix} commands which it contain will only affect the file itself (i.e. \text{infix} status is saved during \text{use} and restored afterwards). A \text{use} file can contain further \text{use} declarations.

All this seems to admit a later extension such as

\[ \text{dec} ::= \ldots \mid \text{engage `filename'} \]

for loading and using precompiled environments (and even updating them); but I think Standard ML should stop short of this.
6. Input and output in Standard ML

The essence of these proposals is:
(i) I/O functions take filenames (tokens) as parameters.
(ii) As in PASCAL, files are opened either for reading or for writing — which cannot be mixed.
(iii) All data types \( \Delta \), i.e. monotypes built with data type constructors, may be input or output — using current infix statues for constructors.

Then we can do with just six functions:

1. `openread : token \rightarrow \text{unit}`
   
   "openread \( f \)" rewinds \( f \) to allow reading from the start; escapes with \"NO FILE\" if \( f \) does not exist.

2. `openwrite : token \rightarrow \text{unit}`
   
   "openwrite \( f \)" creates a new empty file \( f \) to allow writing from the start.

3. `read : token \rightarrow \Delta`

How is the type \( \Delta \) conveyed to the operation? in what form? The type \( \Delta \) is a monotype, and its value is a construction (expression) of type \( \Delta \).

Escapes with

- \"FILE READ\" if syntax is wrong;
- \"EOF\" on end-of-file;
- \"NO OPENREAD\" if not in read mode.

Note: `read ""` refers to keyboard.

`read `-'` refers to current file (within a use declaration)"
(4) Write : token -> \Delta -> unit

"Write `f` `exp` " writes the value of \( \exp \) on \( f \).

Escapes with "No openWRITE" if not in Write mode.

Note: "Write " refers to screen.

(5) readchar : token -> token

"Readchar `f` " reads a single character from \( f \).

Escapes and conventions as for "read".

This allows layout characters to be read; things like PASCAL "readen" can thus be defined.

(6) writechar : token -> token -> unit

"Writechar `f` `t` " writes the first character of \( t \) on \( f \).

Escapes and conventions as for "Write".

Remarks. For read/write dialogue at the Terminal, the implementation could usefully

(a) Indent the dialogue by say 3 spaces

(b) Prompt for input by the name of the type - e.g.

```
int list :
```

Note that programs can self-document the type of input values by

```
exphat type expression, e.g. "read " next : int list "
```
7. Miscellaneous

(1) Sections (DEC/10 ML): I propose that sections be dropped. The feature by which they exported values (i.e., the value of "it" on exit from the section is that of the last expression in the section) seems ad-hoc in the new context. It seems entirely adequate to replace "begin .... exp end" by "local ..... in var result = exp." The only thing we will lack is the naming of sections — but I don't think this is particularly useful.

(2) Conversion to and from Tokens: To match the input/output functions read, write (Section 6), the two functions
\[ \text{parse : Token} \to \Delta \] \[ \text{unparse : } \Delta \to \text{token} \] are proposed. Since \( \Delta \) can be "int", these subsume the old set of \text{int read}, \text{int write} functions (and they are the identity when \( \Delta = \text{token} \)).

(3) Literals in Tokens and Token Lists: I propose we follow Cardelli's use of "'" to quote funny characters in Tokens and Token Lists, and his choice of representations for them.

(4) Comments: \%

(5) Boolean operations: the infixes "\&", "\|" (and, or) evaluate both arguments — i.e., they are true functions (following Luca).
TABLE 1: ELEMENTARY SYNTAX CLASSES OF BARE ML

1. Identifiers

\[ \text{Id ::= Any sequence of letters and digits, starting with a letter,} \]
\[ \text{followed possibly by one or more primes (') } \]
\[ \text{Any sequence of one or more of the symbols} \]
\[ ! \# \$ \& \* \+ \- \./<=>@\^\_\~|\*,. \]

Exceptions: (i) Any reserved word (underlined in the syntax definition) of Standard ML

(ii) The symbol sequences (there are also reserved words)

| . | ← | : | ← |

2. Constructors, Variables

\[ \text{Con ::= any identifier declared by a Data binding as a constructor } \]
\[ \text{or constant, within the scope of that binding; also the } \]
\[ \text{identifier abscid, within the with part of an Abstract-like binding } \]
\[ \text{which introduces id as a type constructor.} \]

\[ \text{Var = Id \ Con (thus, constructors cannot be re-declared as variables within their scope).} \]

3. Type variables

\[ \text{Tyvar ::= 'a|'b|...|'n} \quad (\text{pronounced alpha, beta,... ?}) \]

4. Type constructors

\[ \text{Tycon ::= any type constructor or constant declared by a Data or} \]
\[ \text{Abstract type binding, within the scope of that binding.} \]

Notes (i) Infixed identifiers are introduced later - but not for type constructors.

(ii) An identifier used as a type constructor is considered distinct from the same used as a constructor or variable.
<table>
<thead>
<tr>
<th>EXPRESSIONS</th>
<th>DECLARATIONS</th>
<th>VARIABLE BINDINGS</th>
<th>DATA BINDINGS</th>
<th>ABSTRACT BINDINGS</th>
<th>COMMANDS</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exp ::= var</code></td>
<td><code>dec ::= </code>{rec</td>
<td>var</td>
<td>vib`</td>
<td><code>ut ::= ut &lt;-&gt; exp</code></td>
<td><code>dt ::= </code>{tyvar_seq? id &lt;-&gt; const`</td>
</tr>
<tr>
<td><code>con</code></td>
<td>``(variable)`</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>exp1, exp2</code></td>
<td>``(constant, constructor)`</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>exp : ty</code></td>
<td>``(application)`</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>exp1, exp2</code></td>
<td>``(constraint)`</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>escape exp</code></td>
<td>``(pairing)`</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>exp1? match</code></td>
<td>``(escape trapping)`</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>let dec in exp</code></td>
<td>``(local declaration)`</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fun match</code></td>
<td>``(function abstraction)`</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>`match ::= v1, exp1 ...</td>
<td>un.exp`</td>
<td>``(matching)`</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Conventions:**
(i) `{...}` means optional
(ii) For any syntax class S, S_seq ::= S | (S1, ..., Sn)

(iii) Alternatives are in order of decreasing binding power
(iv) Parentheses may enclose any named syntax class
(v) L, R mean left-, right-associative

Commands are separated by ";"
Table 3: Example of a composite declaration

Setting up the free monoid over 'a

```
local
  rec data 'a seq ← nil | cons of 'a # 'a seq
in
abstype 'a monoid ←→ 'a seq
with
  local rec var ap ← fun nil, m . m
       | cons(x,l), m . cons(x, ap(l, m))
in
var empty ← absmonoid nil
and singleton(x) ← absmonoid (cons(x, nil))
and concat (absmonoid l, absmonoid m) ← absmonoid (ap(l, m))
```

Notes:
(i) The local data declaration qualifies both the abstract finding
('a monoid ←→ ...) and the with part.
(ii) Typechecking will ensure that no object with type involving "seq"
will be exported by the whole declaration.
(iii) The constructor "absmonoid" has been conveniently used in
a varstruc; the need for "repsmonoid" is naturally avoided.
The type constructors `unit`, `bool`, `token`, `int` and `list` can be considered as predefined by the following declarations. Standard functions, also definable from these declarations, are not given here.

1. data unit ← unity ;

   **CONVENTION**: unity is represented instead by "()"

2. data bool ← true | false ;

3. rec data token ← empty | c₁ of token | ... | cₙ of token ;
   (where \{c₁, ..., cₙ\} is the character set)

   **CONVENTION**: c₁(...(cᵢₑₗ₉ empty)...cᵦₗ₉) is represented instead by "c₁, ..., cₙ" 

4. local rec data posint ← one | succ of posint
   in data int ← zero | pos of posint | neg of posint ;

   **CONVENTION**: zero, pos (succ⁻¹one), neg (succ⁻²one) are represented instead by the numerals 0, k, ~k (k > 0)

5. infix :: 30 right ;
   rec data 'a list ← nil | :: of 'a # 'a list ;

   (Note: the qualifier "infix" is not required in data bindings)

Remark: Disjoint sum is not included as a standard type constructor, though it could easily be given by "data ('a,'b) sum ← inl of 'a | inr of 'b". It is likely that users will (or should) prefer their own definitions, with meaningful identifiers as constructors. On the other hand, for technical reasons it seems natural (perhaps necessary) to include Product type (pairing) in the Raw language instead of attempting to define it by data ('a,'b) prod ← pair of ('a,'b)". Note how the product "#" is needed in defining list, for example.
<table>
<thead>
<tr>
<th>Table 5: Standard Abbreviations and Alternative Forms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note: the abbreviations are not to suggest implementation, but merely to show that there is a semantically equivalent &quot;raw ML&quot; form.</td>
</tr>
</tbody>
</table>

1. Expressions

- `quit` → `escape "quit"`
- `exp1 :or exp2` → `exp1 trap any, exp2`
- `exp1 orf "t1 .. tn" exp2` → `exp1 trap (\(\text{t1}, \text{exp2}\) | ... | \(\text{tn}, \text{exp2}\) | \(\text{x}, \text{exp2}\))`
- `exp where dec` → `let dec in exp`
- `case exp of match` → `case exp of (true, exp1 | false, exp2)`
- `if exp then exp1 else exp2` → `case exp of (true, exp1 | false, exp2)`
- `fun v1 ... vn, exp` → `fun v1, ... , (fun vn, exp), ...` (n≥1)
- `[exp1, ..., expn]` → `exp1, ..., expn, nil` (n≥0)
- "ti .. tn" → `[\(\text{ti}, \text{tn}\)]` (n≥0)
- `exp1 ; exp2` → `let var any ← exp1 in exp2`
- `while exp1 do exp2` → `if () where we f() ←
if exp1 then (exp2 ; f()) else ()`

2. Variants

- `[v1, ..., vn]` → `v1, ..., vn, nil` (n≥0)
- "ti .. tn" → `[\(\text{ti}, \text{tn}\)]` (n≥0)

3. Variable Bindings

- `id v1 ... on { :ty? ← exp}` → `id ← fun v1 ... on { :ty?, exp}` (n≥1)
- `v1 id v2 v3 ... on { :ty? ← exp }` → `infix id v1 ... on { :ty? ← exp}` (n≥2)
- `(v1 id v2) v3 ... on { :ty? ← exp }` (when id has infix status)

4. Declarations

- `exp` → `var any ← exp`

Note: By this means expressions become a subclass of declarations. This means that any command sequence is (apart from fix commands) just a declaration! This fact is exploited in parsing external ML files as declarations; see Section 5.
### Table 6: Predefined Identifiers in Standard ML

#### Nonfixes

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>int → int</td>
<td>minus.</td>
</tr>
<tr>
<td>@</td>
<td>α ref → α</td>
<td>contents</td>
</tr>
<tr>
<td>fst</td>
<td>α &amp; β → α</td>
<td>unpairing</td>
</tr>
<tr>
<td>snd</td>
<td>α &amp; β → β</td>
<td></td>
</tr>
<tr>
<td>hd</td>
<td>α list → α</td>
<td></td>
</tr>
<tr>
<td>tl</td>
<td>α list → α list</td>
<td></td>
</tr>
<tr>
<td>map</td>
<td>(α → β) &amp; α list &amp; β list</td>
<td>Lists</td>
</tr>
<tr>
<td>rev</td>
<td>α list → α list</td>
<td></td>
</tr>
<tr>
<td>explode</td>
<td>token → token list</td>
<td>Tokens</td>
</tr>
<tr>
<td>implode</td>
<td>token list → token</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>bool → bool</td>
<td>negation</td>
</tr>
<tr>
<td>unparse</td>
<td>Δ → token</td>
<td>unpairing</td>
</tr>
<tr>
<td>parse</td>
<td>token → Δ</td>
<td>parsing</td>
</tr>
<tr>
<td>ref</td>
<td>⋆ → ⋆ ref</td>
<td>new reference (in expressions)</td>
</tr>
<tr>
<td></td>
<td>α → α ref</td>
<td>reference (in versions)</td>
</tr>
<tr>
<td>it</td>
<td></td>
<td>value of last expression</td>
</tr>
</tbody>
</table>

#### Infixes

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Precedence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>int &amp; int → int</td>
<td>L</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>div</td>
<td>int &amp; int → int</td>
<td>L</td>
<td>50</td>
</tr>
<tr>
<td>mod</td>
<td>bool &amp; bool → bool</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>bool &amp; bool → bool</td>
<td>L</td>
<td>Conjunction</td>
</tr>
<tr>
<td>+</td>
<td>int &amp; int → int</td>
<td>L</td>
<td>40</td>
</tr>
<tr>
<td>-</td>
<td>int &amp; int → int</td>
<td>L</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>∨</td>
<td>bool &amp; bool → bool</td>
<td>L</td>
<td>Disjunction</td>
</tr>
<tr>
<td>::</td>
<td>α &amp; α list → α list</td>
<td>R</td>
<td>30</td>
</tr>
<tr>
<td>++</td>
<td>α list &amp; α list → α list</td>
<td>R</td>
<td>List append</td>
</tr>
<tr>
<td>=</td>
<td>Γ &amp; Γ → bool</td>
<td>L</td>
<td>Comparison</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>int &amp; int → bool</td>
<td>L</td>
<td>20</td>
</tr>
<tr>
<td>&lt;</td>
<td>int &amp; int → bool</td>
<td>L</td>
<td>Integer order</td>
</tr>
<tr>
<td>&lt;=</td>
<td>int &amp; int → bool</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>(β → γ) # (α → β)</td>
<td>L</td>
<td>Composition</td>
</tr>
<tr>
<td>&amp;</td>
<td>(α &amp; β) # (β → γ)</td>
<td>L</td>
<td>10</td>
</tr>
<tr>
<td>#</td>
<td>(α → β) # (α → γ)</td>
<td>R</td>
<td>Pairing of functions</td>
</tr>
<tr>
<td>1</td>
<td>α &amp; β → α &amp; β</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>:=</td>
<td>μ ref &amp; μ → unit</td>
<td>L</td>
<td>0</td>
</tr>
</tbody>
</table>

### Note:

- μ is any monotype
- δ is any mono-data-type
- Γ is any type built from references types by data-type constructors.
| exp ::= | (var  ···)  |
|         | (constant, constructor) |
|         | (application) |
|         | (type constraint) |
| var     | (infix application) |
| con     | (escape with 'let') |
| exp 1 exp 2 | (escape with 'quit') |
| exp : ty | (conditional) |
| exp 1 infix exp 2 | (case analysis) |
| escape exp | (iteration) |
| quit     | (escape trapping) |
| if exp 1 then exp 2 else exp 3 | (universal escape trapping) |
| case exp of match | (selective escape trapping) |
| while exp 1 do exp 2 | (sequence) |
| exp trap match | (list; n ≥ 0) |
| exp 1 or exp 2 | (token list; n ≥ 0) |
| exp 1 or exp 2 "t1 ... tn" exp 2 | (local declaration) |
| exp 1 ; exp 2 | (local declaration) |
| [exp 1 ; ... ; exp n] | (function abstraction) |
| "t1 ... tn" | (Curried abstraction; n ≥ 1) |
| exp where dec | (Matching; n ≥ 1) |
| let dec in exp | |
| fun match | |
| fun v1 ... vn. exp | |

match ::= v1. exp 1 | ... | vn. exp n
### Table 8: Declarations and Bindings in Standard ML

#### Declarations:

<table>
<thead>
<tr>
<th>DEC ::=</th>
<th>EXP</th>
<th>(vacuous declaration)</th>
</tr>
</thead>
<tbody>
<tr>
<td>use 'filename'</td>
<td>(external ML file)</td>
<td></td>
</tr>
<tr>
<td>[rec] var vB</td>
<td>(variable declaration)</td>
<td></td>
</tr>
<tr>
<td>[rec] data dB</td>
<td>(data declaration)</td>
<td></td>
</tr>
<tr>
<td>[rec] abstype ab with dec</td>
<td>(abstract type declaration)</td>
<td></td>
</tr>
<tr>
<td>local decl in dec2</td>
<td>(local declaration)</td>
<td></td>
</tr>
<tr>
<td>decl : decl2</td>
<td>(declaration sequence)</td>
<td></td>
</tr>
</tbody>
</table>

#### Variable Bindings:

<table>
<thead>
<tr>
<th>UB ::=</th>
<th>U &lt;- EXP</th>
<th>(simple binding)</th>
</tr>
</thead>
<tbody>
<tr>
<td>id v1 ... un ? : ty ? &lt;- exp</td>
<td>(function binding: n ≥ 1)</td>
<td></td>
</tr>
<tr>
<td>? v1 infix v2 ; v3 ... un ? : ty ? &lt;- exp</td>
<td>(infix function binding: n ≥ 2)</td>
<td></td>
</tr>
<tr>
<td>vB1 and vB2</td>
<td>(simultaneous binding)</td>
<td></td>
</tr>
</tbody>
</table>

#### Data Bindings:

<table>
<thead>
<tr>
<th>DB ::=</th>
<th>{tyvar-seq; id &lt;- constrs}</th>
<th>(simple)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>(simultaneous)</td>
<td></td>
</tr>
<tr>
<td>dB1 and dB2</td>
<td>(simultaneous)</td>
<td></td>
</tr>
</tbody>
</table>

#### Constrs:

| CONSTRS ::= | id1 : ty1 ; ... ; idn : ty n | (n ≥ 1) |

#### Abstract Type Bindings:

<table>
<thead>
<tr>
<th>AB ::=</th>
<th>{tyvar-seq; id &lt;- ty}</th>
<th>(simple)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>(simultaneous)</td>
<td></td>
</tr>
<tr>
<td>ab1 and ab2</td>
<td>(simultaneous)</td>
<td></td>
</tr>
</tbody>
</table>
### Table 9: Varstructs, Types and Commands in Standard ML

#### Varstructs:

\[
\begin{align*}
U &::= \text{any} \quad \text{(wild card)} \\
&\quad \text{var} \quad \text{(variable)} \\
&\quad \text{con}\{U\}_{\text{seq}} \quad \text{(construction)} \\
&\quad \text{ref}\ U \quad \text{(reference)} \\
&\quad \text{abst}\ U \quad \text{(abstraction)} \\
&\quad U : ty \quad \text{(type constraint)} \\
&\quad U1 \ \text{infix} \ U2 \quad \text{(infixed construction)}
\end{align*}
\]

#### Types:

\[
\begin{align*}
Ty &::= Ty\\text{var} \quad \text{(type variable)} \\
&\quad \{Ty\}_{\text{seq}} \Ty\text{con} \quad \text{(type construction)} \\
&\quad Ty1 \ \# \ Ty2 \quad \text{(binary type)} \\
&\quad Ty1 \ \rightarrow \ Ty2 \quad \text{(function type)}
\end{align*}
\]

#### Commands:

\[
\begin{align*}
\text{Com} &::= \text{dec} \quad \text{(declaration)} \\
&\quad \text{exp} \quad \text{(expression)} \\
&\quad \text{infix id1 ... idn \{prec\} \{ass\}} \quad \text{(infix status)} \\
&\quad \text{nonfix id1 ... idn} \quad \text{(nonfix status)} \\
\text{prec} &::= 1 \ 2 \ 3 \\
\text{ass} &::= \text{left} \ | \ \text{right}
\end{align*}
\]