A PROPOSAL FOR
STANDARD ML
(second draft)
A Proposal for Standard ML

Robin Milner, June '83

1. Introduction
   1.1 How the proposal evolved; 1.2 The purpose of the proposed language; 1.3 Principles followed in this proposal.

2. The bare language
   2.1 Discussion; 2.2 Reserved Words; 2.3 Numerals; 2.4 String constants; 2.5 Identifiers; 2.6 Comments; 2.7 Lexical Analysis; 2.8 The bare syntax.

3. Evaluation
   3.1 Environments and values; 3.2 Environment manipulation; 3.3 Matching variables; 3.4 Applying a match; 3.5 Evaluation of expressions; 3.6 Evaluation of value bindings; 3.7 Evaluation of type bindings; 3.8 Evaluation of declarations.

4. Directives
   4.1 Infix directives; 4.2 Type abbreviation directives; 4.3 Schematic directives.

5. Standard type constructors

6. Standard derived forms
   6.1 Expressions and variables; 6.2 Bindings and declarations.

7. References and equality
   7.1 References and assignment; 7.2 Equality.

8. Input and Output
   8.1 Input/output in Programs; 8.2 External programs.

9. Typechecking

10. Syntaxic restrictions

11. Record of discussions

Appendices: 1 Expressions; 2 Variables and Variable Bindings; 3 Types, Type Bindings and Declarations; 4 Directives, Commands and Programs; 5 Predelclared variables and constants.
1. **Introduction**

1.1 How the proposal evolved

Over the past few years ML has been used by several people for serious work; in parallel, HOPE has been used similarly. The original ML (on DEC-10) lacked was incomplete in some ways, redundant in others. Some of these faults were remedied, and valuable extensions made, by Cardelli in his VAX version; other faults could be mended by borrowing ideas from HOPE.

In April '83, prompted by Bertrand Smythe, I wrote a tentative proposal to consolidate ML, and while doing so became convinced that this consolidation was possible while still keeping its character. Many people immediately discussed the proposal (see below for a list of people; I hope it's complete); there was a wide occurrence of views, but also much consensus -- and of course several bad things in the proposal were brought to light.

This proposal tries to represent the consensus. One point of consensus was that a good language doesn't get changed by a committee; so I have tried to see it as a whole -- despite the fact that most of the good ideas that make it different from previous ML are not my ideas. The people who contributed were: Rod Burstall, Luca Cardelli, Guy Cousineau, Anne Girdon, Gérard Huet, David MacQueen, Robin Milner, Kevin Muffett, Brian Monahan, Peter Mosses, Alan Mycroft, Larry Paulson, David Rugby, Don Sannella, Daint Schmidt, John Scott, Stefan Sokolowski.
It was extremely lucky that we managed to have several separate talks between April and June; both Dave MacQueen and Luca Cardelli just happened to be in Europe, and Dave gave a lot of coherence by visiting Edinburgh, Cambridge, and INRIA. Guy Cousineau also happened to visit Edinburgh from INRIA, and Mike Guzdial and Lenny Paulson came from Cambridge to Edinburgh. Altogether, we couldn't have chosen a better time to do the job. Also, Luca Cardelli has a detailed draft of an ML manual, at present describing his VAX version; he very generously offered to freeze it until this proposal is worked out.

With all this good fortune, I hope that people will be able to accept the various ways in which this language falls short of their expectations. I am ready to act as the focus for corrections to the proposal; I think it is converging, and that a corrected version should soon appear in "Polyphony" during the next few months; the experience of implementation may induce a few (I hope minor) amendments.

This is a good place to recall that three people—nor mentioned above—worked on the design and implementation of ML originally, and the language owes a lot to them: Lockwood Morris, Malcolm Newey and Chris Wadsworth.
The purpose of the proposed language

This 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, types-as-parameters or polymorphic typechecking, ...). Nor is it meant to be a commercial product. Mainly, it aims to be a vehicle for research in functional language design, and a means for propagating the functional programming craft and for developing functional styles.

The first of these aims, research, demands that it should be clearly delineated, with well-defined omissions. One such omission is the notion of module (as in e.g. HOPE or MODULA 2); another is polymorphic references and assignment. Since many people want these — particularly modules — a justification is needed.

For modules, it seemed preferable to me to avoid inserting a tentative and modest form as standard; rather, it is better to leave the way clear for stronger proposals. Such a proposal, perhaps from those with experience (e.g., modules in HOPE), is likely to come up soon; it could be called MML (Modular ML), and is likely to be compatible with almost all the present proposal (in fact, all except the abstract method and the struct directives). However,
This proposal includes one thing which - though not the whole essence of the module idea - is always meant to be provided by modules, namely the possibility of precompilation of parts of programs. Several people, including MacQueen, Cardelli, Mitchell and Paulson, argue convincingly for this; it seems to be provided without much fuss by the `spec' directive (Section 4.3).

For polymorphic references and assignment, we do have an elegant and sound scheme worked out by Luis Damas, but it is not documented and we would do better to wait for a clear exposition, either from Damas or - as promised - from 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, with the advantage of keeping the well-understood type-checking discipline, which derives from Curry's combinatory logic via Hindley.

The second aim, teaching and propagating, will benefit from a language which is well-rounded and not too large, and which doesn't depart far from what we know works well. It is quite encouraging that the syntax of a bare form of the proposed language - without derived forms - fits onto a single page (Section 2.8), and that it is easily recognized as ML with injections of HOPE.

In Section 11, are some reports of points made in various discussions between the tentative proposal (April) and now,
1.3 Principles followed in this proposal

The overriding principle of this design is that the language should be restricted to ideas which are well-tried, either in previous versions of ML or in other functional languages (in fact, the main other source is HOPE, mainly for its pattern-matching of arguments). A second principle is that well-tried ideas should be generalised, where the generalisation is apparently natural; this has been applied mainly in generalising ML varstubs to HOPE patterns, in broadening the structure of declarations (following Cardelli’s declaration connectives which go back to Peter Milne’s Thesis), and in allowing exceptions which allow escape with values of arbitrary type (generalising the original failure construct of ML).

A third principle is to specify the language completely, so that programs will port from one concrete implementation to another with minimum fuss. This entails, first, precision in concrete syntax (I agree with Cardelli that abstract syntax is in some respects more important — but we do not all have structure editors yet, and humans still communicate with each other in concrete syntax!); second, it entails exact evaluation rules (e.g., we have to specify the order of evaluation of two expressions, one applied to the other, just because of the escape mechanism).
2. The bare language

2.1 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 infixes and type abbreviations, and specifying type environments (Section 4).
3. Standard types (Section 5).
4. Input and Output, based on the standard type "stream", and external ML files (Section 8).
5. References and Equality (Section 7).
6. Type checking (Section 9).

The composite expression forms are application, type constraint, tupling, raising and handling exceptions, local declaration (using let) and function abstraction. Variables are (except for the wildcard $) a subclass of expressions used in value bindings. Declarations may declare value variables (using value bindings), types with associated constructors or operations (using type bindings), and exceptions; apart from this, one declaration may be local to another (using local) and sequences of declarations are allowed. An ML program is just a sequence of declarations or directives; thus — omitting the directives which have purely syntactic effect — it is just a single declaration. 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 be used as identifiers. In this document the alphabetic reserved words are always underlined.

abtype and case do else end escape exception fun handle if in infix left let local longtype nonfix of op or raise rec
shortype spec then type use val with while

( ) [ ] { } ; : ; . \ # $
& \ ? \ = \ \rightarrow \ \leftarrow

2.3 Numerals

A numeral is any non-empty sequence of digits.

2.4 String constants

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

\1 .. \9 One to nine spaces \E Escape
\0 Ten spaces \N Null (Ascii 0)
\R Carriage return \D Del (Ascii 127)
\L Line feed \^c Ascii control character c
\T Tabulation \c c (any other character)
\B Backspace
2.5 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 \((\text{var})\)
- value constructors \((\text{con})\)
- type variables \((\text{tyvar})\)
- type constructors \((\text{tycon})\)
- exception identifiers \((\text{exid})\)

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

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

In either case, however, reserved words are excluded.

A type variable \((\text{tyvar})\) may be any alphabetical identifier starting with a prime. The other four classes \((\text{var}, \text{con}, \text{tycon}, \text{exid})\) are separated 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 (2) 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.

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.

An identifier may be given infix status by the infix command; this status only pertains to its use as a var or a con. If id has infix status, then "exp1 id exp2" (resp. "vst1 id vst2") may occur wherever the application "id (exp1, exp2)" (resp. "id (vst1, vst2)") 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 command. On ML files only standard infixes (e.g. "+", ":") are assumed; after reading the file, the previous infix or nonfix status of every identifier is resumed (but see 4.1 for a refinement of this rule).
2.6 **Comments**

A comment is any character sequence within curly brackets `{ }` in which curly brackets are properly nested.

2.7 **Lexical analysis**

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

**Note:** As a consequence of this simple approach, spaces are needed sometimes to separate identifiers and reserved words. Two examples are:

\[ a := b \quad \text{not} \quad a := ! b \quad \text{assigning contents of} \; b \; \text{to} \; a \]

\[ ~: \text{int} \rightarrow \text{int} \quad \text{not} \quad ~: \text{int} \rightarrow \text{int} \quad \text{unary minus qualified by its type} \]

Rules which allow omission of spaces in such examples, such as adopted by Cacciola 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.
2.8 The bare syntax

Conventions: (1) ... means optional. (2) For any syntax class S, $S_{-seq} ::= S \mid (S_1, \ldots, S_n)$. (3) Alternatives are in order of decreasing precedence, and L(R) means left(right) association.

### Expressions exp

$$\text{aexp ::= } \begin{align*}
\text{var} & \quad \text{(variable)} \\
\text{con} & \quad \text{(constructor)} \\
(\text{exp}) & \\
\end{align*}
$$

$$\text{exp ::= } \begin{align*}
\text{aexp} & \\
\text{exp aexp} & \quad \text{(application)} \\
\text{exp : tyj} & \quad \text{(constraint)} \\
\text{exp1, \ldots, expn} & \quad \text{(tuple)} \\
\text{raise exid exp} & \quad \text{(raise exception)} \\
\text{Let \ dec \ in \ exp \ end} & \quad \text{(local decl)} \\
\text{exp handle exid match} & \quad \text{(hand exception)} \\
\text{fun match} & \quad \text{(function)} \\
\text{match ::= } \begin{align*}
\text{us1. exp1 | \ldots | usn.expn} & \\
\end{align*}
$$

### Declarations dec

$$\text{dec ::= } \begin{align*}
\text{val \ ut} & \quad \text{(values)} \\
\text{type \ t6} & \quad \text{(types)} \\
\text{abstract \ t6 \ with \ dec \ end} & \quad \text{(abstract types)} \\
\text{exception \ id1; \ldots; \idn} & \quad \text{(exceptions)} \\
\text{local \ decl \ in \ dec \ end} & \quad \text{(local decl)} \\
\text{decl ; dec2} & \quad \text{R (sequence)} \\
\end{align*}
$$

### Commands com

$$\text{com ::= } \begin{align*}
\text{dec ;} & \quad \text{(declaration command)} \\
\text{directive ;} & \quad \text{(directive command)} \\
\end{align*}
$$

### Value bindings ut

$$\text{ut ::= } \begin{align*}
\text{ut \ { : t3} == \ exp} & \quad \text{(simple)} \\
\text{ut1 \ and \ ut2} & \quad \text{(simultaneous)} \\
\text{rec \ ut} & \quad \text{(recursive)} \\
\end{align*}
$$

### Type bindings t6

$$\text{t6 ::= } \begin{align*}
\text{\{ tvar - seq \} \ t6; \ tcon == \ constrs} & \quad \text{(simple)} \\
\text{t61 \ and \ t62} & \quad \text{(simultaneous)} \\
\text{rec \ t6} & \quad \text{(recursive)} \\
\end{align*}
$$

### Constructors

$$\text{con ::= } \begin{align*}
\text{con \ \{ e1. tij \} \ | \ \ldots \ | \ con \ \{ e1. tyn \} } \\
\end{align*}
$$

### Types ty

$$\text{ty ::= } \begin{align*}
\text{tvar} & \quad \text{(type variable)} \\
\text{\{ t - seq \} tij; tcon} & \quad \text{(type constructor)} \\
\text{tj1 \# \ ... \# \ tyn} & \quad \text{(tuple, type)} \\
\text{tj1 \ -> \ tj2} & \quad \text{R (function type)} \\
\end{align*}
$$

---

A program is a series of commands.
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 a value constructor, 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 type checking and compilation, not to evaluation.)

An exception $e$, associated to an exception identifier, exists in any $EE$, is an object from which an $e$ may be recovered. A packet $p = (e, v)$ is an exception paired with a value. Packets are not values. Besides possibly changing $S$ (by assignment), evaluation of a phrase returns a result as follows:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>$v$ or $p$</td>
</tr>
<tr>
<td>Declaration</td>
<td>$E$ or $p$</td>
</tr>
<tr>
<td>Value binding</td>
<td>$VE$ or $p$</td>
</tr>
<tr>
<td>Type binding</td>
<td>$VE \prec p$</td>
</tr>
</tbody>
</table>

For every phrase except a handle expression, whenever its evaluation demands, the evaluation of an immediate subphrase which returns a packet $p$ as result, $p$ is also the result of the phrase.

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 \( \langle (\text{var}_1, v_1) \ldots (\text{var}_n, v_n) \rangle \) for a value environment, where the \( \text{var}_i \) are distinct. Then \( \langle \rangle \) is the empty VE, and \( \text{VE}_1 \circ \text{VE}_2 \) means the VE in which the associations of \( \text{VE}_2 \) supersede those of \( \text{VE}_1 \). Similarly for exception environments. If \( E = \langle \text{VE}_3, \text{EE} \rangle \) and \( E' = \langle \text{VE}', \text{EE}' \rangle \), then \( E + E' \) means \( \langle \text{VE}_3 + \text{VE}', \text{EE} + \text{EE}' \rangle \), \( E + \text{VE}' \) means \( E + \langle \text{VE}', \langle \rangle \rangle \), etc. This implies that an identifier may be associated both in VE and in EE without conflict.

"better to write \( \text{VE}_1 \circ \text{VE}_2 \)"
3.3 Matching varstructs

The matching of a varstruct \( \mathbf{vs} \) 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 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 \( \mathbf{us} \) to \( v \):

\[ $: \text{the empty VE is returned.} \]
\[ \text{var}: \text{the VE } \langle (\text{var}, v) \rangle \text{ is returned} \]
\[ \text{const}: \text{if } v = \text{const}, \text{then } \mathbf{us} \text{ is matched to } v; \text{ else failure.} \]
\[ \text{vs1, \ldots, vsn}: \text{if } v = (v_1, \ldots, v_n) \text{ then } \mathbf{us} \text{ is matched to } v \text{ returning } \text{VE1, \ldots, VEn} \text{ is returned.} \]

\[ \text{us1 : ty}: \text{US is matched to } v. \]

[What about requirement that varstructs be linear, so that \( \text{VE1} \) are all over disjoint sets of ids?]

3.4 Applying a match

Assume environment \( E \). Applying match \( m = \text{vs1.exp1} \mid \ldots \mid \text{vsn.expn} \) to value \( v \) returns a value or packet as follows:

Each \( \text{vs1} \) is matched to \( v \) in turn, from left to right, until one succeeds returning \( \text{VE1} \); then \( \text{exp1} \) is evaluated in \( E + \text{VE1} \). If none succeeds, then the packet \( (\text{E.string}, \text{"FAILED TO MATCH VALUE } \mathbf{v} \text{"}) \) is returned, where \( \text{E.string} \) is the standard exception always associated in \( E \) with the standard exception identifier \text{STRING} of type \text{string}, and \( \mathbf{v} \) is the print form of value \( v \).

Thus, for each \( E \), a match \( m \) denotes a function value.
3.5 Evaluation of expressions

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

- var: returns value $VE(\text{var})$
- con: returns value $VE(\text{con})$
- $\text{exp}_{aexp}$: $\text{exp}$ is evaluated, returning function value $f$; then $\text{aexp}$ is evaluated, returning value $v$; then $f(v)$ is returned.
- $\text{exp}_1, \ldots, \text{exp}_n$: the $\text{exp}_i$ are evaluated in sequence, from left to right, returning $v_i$ respectively; then $(v_1, \ldots, v_n)$ is returned.
- raise_exid $\text{exp}$: $\text{exp}$ is evaluated, returning value $v$; then packet $(e, v)$ is returned, where $e = EE(\text{exid})$.
- $\text{exp}_{\text{handle exid}} \text{match}$: $\text{exp}$ is evaluated; if $\text{exp}$ returns $v$ then $v$ in returned; if $\text{exp}$ returns $p = (e, v)$ then (1) if $e = EE(\text{exid})$ then match is applied to $v$, (2) if $e \neq EE(\text{exid})$ then $p$ is returned.
- let dec in $\text{exp}$ end: $\text{dec}$ is evaluated, returning $E'$; then $\text{exp}$ 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 $E$.
- $\text{exp}_{\text{ty}}$: $\text{exp}$ is evaluated.
3.6 Evaluation of value bindings

Assume environment \( E = (V, E) \). Evaluating a value binding \( \text{vb} \) returns a value environment \( V'E \) or a packet as follows, by cases of \( \text{vb} \):

\( \text{vb} = \text{exp} : \) \( \text{exp} \) is evaluated in \( E \), returning value \( v \); then \( v \) is matched to \( v \); if this returns \( V'E \), then \( V'E \) is returned, and if it fails then the packet \((e \text{string}, \text{"FAILED TO BIND VALUE } v\text{"})\) is returned, where \( e \text{string} \) and \( v \) are as described in 3.4.

\( \text{vb}_1 \text{ and } \text{vb}_2 : \) \( \text{vb}_1 \) is evaluated in \( E \), returning \( V'E_1 \); then \( \text{vb}_2 \) is evaluated in \( E \), returning \( V'E_2 \); then \( V'E_1 + V'E_2 \) is returned.

\( \text{rec } \text{vb} : \) \( \text{vb} \) is evaluated in \( E' \), returning \( V'E' \), where \( E' = (V'E + V'E', E) \). Because the values bound by evaluating \( \text{vb} \) must be function values, \( E' \) is well defined by "tying knots" (Lamind).
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 \( \mathbf{tb} \) returns a value environment \( \mathbf{VE}' \) (if it cannot return a packet) as follows, by cases of \( \mathbf{tb} \):

\[
\begin{align*}
\text{tycon} & = \text{const} \mid \text{ty} \mid \ldots \mid \text{conn}, \text{of ty} \mid \text{sub} \mid \text{of ty} \mid \text{sub} \mid \text{of ty} \mid \text{sub} \\
\mathbf{VE}' & = \left< \left( \text{const}, v_1 \right), \ldots, \left( \text{conn}, \text{of ty} \right) \right> \text{ is returned, where} \\
v_i & \text{ is either the constant value } c_{\text{coni}} \text{ (if "of ty" is absent) or else the function value } \text{of ty} \rightarrow c_{\text{coni}} \text{. Note that all other effects of this type binding is handled by the compiler and type checker, not by evaluation.}
\end{align*}
\]

\( \mathbf{tb}_1 \) and \( \mathbf{tb}_2 \) : \( \mathbf{tb}_1 \) and \( \mathbf{tb}_2 \) are evaluated, returning \( \mathbf{VE}_1 \) and \( \mathbf{VE}_2 \) respectively; then \( \mathbf{VE}' = \mathbf{VE}_1 + \mathbf{VE}_2 \) is returned.

\( \mathbf{rec \ tb} \) : \( \mathbf{tb} \) is evaluated. Note again that the recursion is handled by type checking only.
3.8 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 $v$**: $v$ is evaluated, returning $VE'$; then $E' = (VE', <>)$ is returned.

- **type $t$**: $t$ is evaluated, returning $VE'$; then $E' = (VE', <>)$ is returned.

- **abstract $t$ with dec end**: $t$ is evaluated, returning $VE'$; then $dec$ is evaluated in $E + VE'$, returning $E'$; then $E'$ is returned.

- **exception $e$**: a new exception $e$ is generated (from which the exception identifier $e$ may be recovered), and $E' = (<>, ((e, e)))$ is returned.

- **local $dec_1$ in $dec_2$ end**: $dec_1$ is evaluated, returning $E_1$; then $dec_2$ is evaluated in $E + E_1$, returning $E_2$; then $E' = E_2$ is returned.

- **dec 1; dec 2**: $dec_1$ is evaluated, returning $E_1$; then $dec_2$ is evaluated in $E + E_1$, returning $E_2$; then $E' = E_1 + E_2$ is returned.

Note that each declaration is defined to return only the new environment which it makes, but the effect of declarations composed by ";;" is to accumulate environments.
4. Directive

There are three kinds of directive: they are for establishing infix status of value variables and constructors, for introducing and cancelling type abbreviations, and for specifying an assumed type environment (mainly in external programs, which allows their precompilation).

4.1 Infix directives

\[
\text{infix id \ldots idn} \\
\text{nonfix id \ldots idn}
\]

The infix directive influences infix status for each id \(i\) (as a value variable or constructor), and the nonfix directive cancels it. While id has infix status, each occurrence of it must be infixed or else preceded by op. Several standard functions and constructors have infix status (see Appendix 5) with precedence and left or right association; user-defined infixes are all left associative with precedence 0 (this is to simplify the spec directives — see 4.3 below).

An external program assumes initially only standard infix statuses. After it is imported (by the use declaration) previous infix or nonfix statuses are resumed, except that any id declared with uncancelled infix status within the external program, whose declaration contains the external program, retains its infix status. This is naturally achieved by recording infix status in the type environment \(TE\).

4.2 Type abbreviation directives

\[
\text{shorttype \{ttype-seq\} tycmn = tycj \ldots \text{ttype-seq} tycmn = \text{tyn}} \\
\text{language tycm1 \ldots tycmn}
\]

The shorttype directive has no semantic significance; it merely allows any instance (by substitution for type variables) of tycj to be replaced — both in programs and on output — by the corresponding instance of
In abbreviating types on output, most recent short-type directives are matched first, and each type is matched before its sub-types.

The long-type directive cancels the short-type status of tyconi, which revokes its previous role (if any) as a type constructor.

There are no standard type abbreviations. An external program initially assumes no type abbreviations, and any which it introduces are later forgotten.

4.3. Specification directives

The purpose of specification directives is to specify type constructors, value variables and continuas (with their infix status), and exception identifiers, which are used but not declared in a program. This is mainly to make external programs syntactically self-contained and to allow their precompilation. The directives would normally occur at the head of these programs, but are only necessary if precompilation is done—and then need only occur before the specified items are mentioned.

```
spec val [op] id:tyl and ... and [op] id:tyun
spec type [tyvar-seq] tycon1 and ... and [tyvar-seq] tyconn
spec type ty
spec exception id:tyl and ... and id:tym
```

Note that type contracts are required, not optional as in declarations.

The `spec val` directive specifies the generic types of value variables which are to be assumed (and infix status, if any), when the program is loaded—precompiled or not—by use, the current type environment TE.
must record each idi as a variable (or value constructor) having

type tyj, of which tyj is an instance, and infix status iff

specifies. The first form of spec type directive indicates:
type constructors which must be in force (declared by type or subtype) on
loading. The second form indicates types — with constructors which
must be in force on loading. This form is needed for precompilation
of a program which uses the specified constructors in varstructs, or
uses equality (=) on the specified types; the full form (using a ty)
is adopted because precompilation must be able to check the conditions
to be satisfied by the varstructs in a match; see Section 10
under (2).

The spec exceptin directive specifies exception identifiers which
must be in force on loading.

As an example, if a program assumes the existence of a type
'a TREE, with three constructors (one infixed), it may specify:

spec type 'a TREE == multtree | nilp of 'a | op constr of 'a TREE & 'a TREE;
and then the type environment TE on loading must record TREE as a
unary type constructor with these three value constructors at the same (or more
general) types. But if the program does not use the constructors in
varstructs, nor uses equality on trees, then it may merely specify:

spec type 'a TREE;
spec val multtree : 'a TREE and nilp : 'a -> 'a TREE
and op constr : 'b TREE & 'b TREE -> 'b TREE;

In this case, the loading environment may be as above, or may
merely record the unary type constructor TREE and three operations
(constructors or not) with appropriate type,
5. **Standard type constructors**

The base language provides the function-type constructor \(\to\), and for each \(n \geq 1\) a tuple-type constructor \(\times_n\). \((t_1, \ldots, t_n)\times_n\) is written \(t_1 \times \cdots \times t_n\). Beside these, the following are standard:

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

The constructor unit, bool, and list are fully defined by the following assumed declaration (note that "::" is a standard infix):

```
type unit := ()
and bool := true | false
and rec 'a list := nil | 'a :: 'a list;
```

The type constant int (integer) is equipped with constants \(0, 1, 2, 3, \ldots\). The type constant string is equipped with constants as described in 2.4. The type constant stream is for Input/Output; see Section 8. The type constructor ref is for constructing reference types; see Section 7.

The standard functions over all these types are listed in Appendix 5. There are not a lavish number of them; we envisage libraries of functions provided by each implementation, together with their ML declarations (though they may be implemented more efficiently).
### Derived Form vs Equivalent Form

<table>
<thead>
<tr>
<th>Derived Form</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressions</strong></td>
<td></td>
</tr>
<tr>
<td><code>escape exp</code></td>
<td><code>raise STRING exp</code></td>
</tr>
<tr>
<td><code>exp1 ? exp2</code></td>
<td><code>exp1 Trap ($, exp2)</code></td>
</tr>
<tr>
<td><code>case exp of match</code></td>
<td><code>(fun match) exp</code></td>
</tr>
<tr>
<td><code>if exp then exp1 else exp2</code></td>
<td>`case exp of (true, exp1</td>
</tr>
<tr>
<td><code>exp1 or exp2</code></td>
<td><code>if exp1 then true else exp2</code></td>
</tr>
<tr>
<td><code>exp1 &amp; exp2</code></td>
<td><code>if exp1 then exp2 else false</code></td>
</tr>
<tr>
<td><code>exp where dec end</code></td>
<td><code>let dec in exp end</code></td>
</tr>
<tr>
<td><code>exp1 ; exp2</code></td>
<td><code>let val $ = exp1 in exp2 end</code></td>
</tr>
<tr>
<td><code>while exp1 do exp2</code></td>
<td><code>let val rec f = fun(). if exp1 then (exp2 ; f) else () in f() end</code></td>
</tr>
<tr>
<td><code>[exp1 ; ... ; expn]</code></td>
<td><code>exp1 ; ... ; expn :: nil</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[v1 ; ... ; vn]</code></td>
<td><code>v1 ; ... ; vn :: nil</code></td>
</tr>
</tbody>
</table>

The derived form may be implemented more efficiently than its equivalent form.

The type-checking of each derived form is defined by that of its equivalent form, except in the case expression which is treated more like the `let` expression. (See Section 9.)

The binding power of all base and derived forms is shown in Appendix 1. Important: ";" has weakest binding power in both expressions and declarations.

The `escape` and `trap` forms refer to a predefined exception identifier "STRING" of type `string`; this models exactly the old ML failure forms.
6.2 Bindings and declarations

**VALUE BINDINGS**

<table>
<thead>
<tr>
<th>DERIVED FORM</th>
<th>EQUIVALENT FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>var (\text{a}1) \ldots \text{a}n \text{e} \text{ty} \Rightarrow \text{exp}</td>
<td>var \Rightarrow \text{fun} \text{a}1\ldots \text{fun} \text{a}n\text{exp} \text{ty}</td>
</tr>
<tr>
<td>var \text{a}1 \text{e} \text{ty} \Rightarrow \text{exp}</td>
<td>var \Rightarrow \text{fun} \text{a}1 \text{exp} \text{ty}</td>
</tr>
</tbody>
</table>

**TYPE BINDING**

\[ \text{ty} \text{var-seq} \text{tycon} \Rightarrow \text{ty} \]

\[ \text{tyvar-seq} \text{tycon} \Rightarrow \text{mk-tycon of ty} \]

**DECLARATIONS**

\[ \text{exp} \]

\[ \text{val it} \Rightarrow \text{exp} \]

**Notes:** The first derived value binding allows Curried function definitions; the second allows separate equations in defining non-Curried functions with several patterns. Separate equations for Curried function definitions are forbidden.

The derived type binding is for isomorphic types, introducing an explicit "abstraction" constructor consisting of "mk-" prefixed to the type constructor.

The derived declaration is principally for treating expressions at top-level as degenerate declarations. The variable "it" is just a normal variable, but is useful for referring to the value returned by the last top-level expression.
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 \( \mu \):

(1) \( \text{ref} : \mu \to \mu \text{ref} \), which associates (in the store) a new reference with
its argument value. In variables, \( \text{ref} : \alpha \to \alpha \text{ref} \) may
be used polymorphically.

(2) \( \text{of} := : \mu \text{ref} \# \mu \to \text{unit} \), which associates its second
(value) argument with its first (reference) argument in the store,
and returns \( () \) as result.

The polymorphic contents function "!" is provided, but may be defined
as follows: "\text{val} \! (\text{ref} \, x) = x".

7.2 Equality

The overloaded equality function \( \text{of} = : \gamma \# \gamma \to \text{bool} \)
is available at all types \( \gamma \) built from references by tuple type
and type constructor (declared by type (including "list"), not by abstype :

\[ \gamma ::= \text{ty ref } | \gamma \# \gamma \# \ldots \# \gamma | \{ \gamma \} \]

On references, equality means identity. On objects of other types \( \gamma \)
it is recursively defined on the structure of \( \gamma \) in the natural way.
The inequality function \( \text{of} < > \) is also provided.
Following Carreille's suggestion, Input/Output is done using
by streams (of characters). Filenames are strings. A stream is created from a file
by the function "getstream : string → stream", and a file is created
(or re-created) from a stream by the function "putstream : string → stream → unit".
These are the only operations which refer to files. Getstream doesn't change
the file (and treats nonexistent files as empty); putstream doesn't change the stream.

"stream" may be regarded as a pre-declared abstract type; it
provides an operations the two functions getstream and putstream, and
four stream operations:

- newstream : unit → stream (Creates an empty stream)
- nextstring : stream → int → string (Returns a string of given length
  from the front of a stream, leaving the stream unchanged)
- instring : stream → int → string (Removes a string of given length
  from the front of a stream).
- outstring : stream → string → unit (Appends a string to a stream).

Terminal I/O is represented by the standard stream variables input/output.
The variable "input" denotes the (infinite!) stream of characters from the
keyboard; this stream may be terminated by e.g. "CTRL Z" whereupon input
will again stand for the stream of characters which will follow. The variable
"output" denotes the (initially empty) stream of characters sent to the screen.

The functions getstream and putstream will escape with "getstream", 
"putstream" in case of inadmissible file names, protection status etc.
(This is implementation dependent, and not reflected in the description given below).
The functions nextstring, instring escape with "nextstring", "instring" if their
integer argument is negative or too large. When necessary, nextstring and instring
must prompt for a further segment of the input stream. Applying nextstring
or instring to the output stream, or instring to the input stream, generates an escape.
The following abstract type definition of stream is to be regarded as evaluated once when a new file area is created. (The description is not an implementation suggestion! It is only for clarification.) It establishes a directory: a map from strings (filenames) to string references (representing files). All uses of getstream and putstream by ML refer to the directory, and other file-handling operations outside ML are thought of as working through these two functions. To shorten the description, we assume a function "split: string # int -> string # string" such that \( \text{split}(s, n) = (s^1, s^2) \) where \( s^1 \cdot s^2 = s \) and \( s^1 \) contains \( n \) characters; split escapes if \( n \) is negative or too large.

```plaintext
abstract stream <= string ref with
  | first initialise the file directory with all files empty;
val directory : string -> string ref == fun s => ref 
  | next define the file operations; error conditions are not represented here;
val getstream (filename : string): stream == mk_stream (ref (! (directory filename)))
and putstream (filename : string) (mk_stream (ref s): stream): unit ==
  directory filename := s;
  | finally define the stream operations;
val newstream (): stream == mkstream (ref "")
and nextstring (mk_stream r : stream) (n: int): string ==
  let val (s, $ = split (! r, n) in s end ? escape "nextstring" and insting (mk_stream r : stream) (n: int): string ==
    let val (s, s' = split (! r, n) in r := s; s end ? escape "insting" and
    outstring (mk_stream r : stream) (s: string): unit == r := ! r ^ s
    end { of stream }; |
Note that "nextstring 1" escapes if \( s \) is empty, allowing a test for the empty stream to be defined. The function nextstring is needed also for "peeking" at streams without changing them.
8.2 External programs

An ML program on a file may be evaluated using the `use` declaration; since a program (ignoring directives) is just a declaration, the meaning of this is well-defined. Note that (Section 10.7) a `use` declaration may not be inside a `match`. This means it cannot be evaluated more than once. It may, however, be in a toplevel context such as

```ml
local use "myprog" in dec end;
```

An external program assumes only standard `infix` status, and standard (i.e. no `!`) type abbreviations; if it introduces any of these, they last only for the program itself. Thus the effect of directives in the main program is not disturbed by a `use` declaration. An external program may contain further `use` declarations.

By use of `spec` directives, an external program may specify bindings which it requires in any environment to which it is loaded. This allows external programs to be precompiled. The `use` declaration may refer both to source and to precompiled external programs.

† There is one exception: `infix` functions declared in the external program retain their `infix` status (see 4.1).
9. Type-checking

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

In a match $m = \text{usi} . \text{exp1} \ldots \text{usin} . \text{expn}$, the types of all usi must be the same (ty'), and if variable var occurs in usi then all free occurrences of var must have the same type as its occurrence in usi. There is one relaxation of this rule—the case expression; see below. In addition, the types of all the expi must be the same (ty'). Then $\text{ty} \rightarrow \text{ty'}$ is the type of $m$.

The type of "fun match" is the type of the match.

The type of "\text{exp handle exid match}" is \text{ty'}, where \text{exp} has type \text{ty'}, \text{match} has type $\text{ty} \rightarrow \text{ty'}$, and \text{exid} has type \text{ty}. The type of "\text{raise exid exp}" is arbitrary, but \text{exp} and \text{exid} must have the same type. Thus, the type of an exception may be polymorphic; \text{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).

The type of "\text{case exp of match}" is \text{ty'}, where \text{exp} has type \text{ty} and \text{match} has type $\text{ty} \rightarrow \text{ty'}$. However, in the match, for each var occurring in a usi, each occurrence of var in expi is only required to have as type a generic instance of its type in usi; in this respect, case is similar to let.
A type variable is only explicitly bound (in the sense of variable-binding in \( \lambda \)-calculus) by its occurrence in \( \text{tyvar-seq} \) in the type binding \( \{ \text{tyvar-seq} ; \text{tycon} \rightarrow \text{consts} \} \), and then its scope is \( \text{consts} \). This means that bound uses of \( 'a \) in both \( t1 \) and \( t2 \) in the type binding \( t1 \text{ and } t2 \) have 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 smallest top-level command 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.
10. Syntactic restrictions

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

(2) In a match \( \text{"v1.exp1} \ldots \text{v} m . \text{expm"} \), the varstruct sequence \( v_1, \ldots, v_m \) should be non-decreasing and exhaustive. That is, for \( i < j \), \( v_i \) must not match all the values which \( v_j \) matches, and every value (of the right type) must be matched by some \( v_i \). The compiler must report a violation of this restriction, but should still compile the match. The restriction applies to all derived forms; in particular, this means that in the Curried function binding \( \text{var} \times v_1 \ldots v_m . \text{exp} \), each separate \( \text{\textit{vs}i} \) should be exhaustive by itself.

(3) For each value binding \( \text{\"vs i: \text{ty} i \text{= exp"} } \), the compiler must issue a report (but still compile) if either \( \text{\textit{vs}} \) is not exhaustive or \( \text{\textit{vs}} \) contains no variable. This will (on both counts) detect errors like \( \text{\"val x: \text{ty} \text{= exp"} } \), in which the user expects to declare a new variable \( \text{\textit{ni}} \), whereas the language dictates that \( \text{\textit{ni}} \) 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; e.g. \( \text{\"val x: e \text{= exp"} } \) is not faulted by the compiler at top level, but may of course generate an escape "FAILED TO MATCH ..." (see Sect. 3.4).

(5) In every instance of \( \{ \text{\textit{ty}var \ldots \text{\textit{ty}j}} \text{\textit{ty}k} \} \) \( \text{\textit{ty}var-\textit{seq}} \), the \( \text{\textit{ty}var-\textit{seq}} \) must mention 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.

(4) For each value binding \( \text{\"vs i: \text{ty} i \text{= exp"} } \) within \( \text{\textit{rec}} \), and for each derived form of such a binding, \( \text{\textit{exp}} \) must be of the form \( \text{\"fun match"} \).
(6) In "let dec in exp and" and "local dec in dec' end" no type constructor excepted by dec may occur in the type of exp or in the type of any variable or value constructor expected by dec'. The "where" form inherits a similar restriction.

(7) The declaration form use "filename" must not occur within a match, nor within a derived form if it would occur within a match in the equivalent form.

(8) Every top-level exception declaration must be explicitly constrained by a monotype.
11. Record of discussions

In this section I try to record some of the views which people expressed about my tentative proposal and alternatives to it. There were two very helpful group discussions, with eight or ten people present, and several smaller meetings at INEA, Cambridge and Edinburgh. Dave MacQueen was most active in these, gathering views together as much as possible. A lot of progress was made; even so, I sensed that more group discussions would have wasted people’s time in proportion to their benefit; the best thing was to collect views in smaller discussions and try to find a coherent proposal which gained from them, while avoiding weak compromises.

More things were discussed than are reported below, but I hope to have recorded the more important points of contention.
11.1 Abstract types and modules

In original ML - and in my Tentative proposal - the type isomorphism declaration "abs_type \{lycan-seq,lycan \} ty with ... " was a basic form. In this proposal, thanks to Red Burdall and others, it becomes a derived form which is a special case of something more powerful.

The present form seems to represent the natural effect of introducing HODE data types into a consolidated ML, without further language development. Note that the data-type-with-constructor idea is itself a natural rounding off, generalising the fixed variants of old ML to allow user-defined variants.

On the other hand, the strongest pressure for developing the language was in the case of modules. The argument - from Cardelli, Mitchell, Paulson (at least) - is certainly strong. First, modules allow separate compilation; secondly, they allow specification to be separated from implementation. In neither version, they can be parametric, and can even be values of a new kind.

This pressure was balanced by others who favoured the conservative cause of consolidation.

There are virtues and dangers in both courses of action. Having committed myself to propose something definite (or ah...
less to monitor such a proposal) I am acutely aware of the need to act quickly - otherwise impetus is lost, people's viewpoints change, and the chances of a coherent (if not perfect) design recede dramatically! So it seemed better to leave modules our of this proposal because:

(1) The original aim was consolidation

(2) It isn't clear to me - even after reading a nice, intuitively modest module proposal from Luca Cardelli - where to draw the line in what modules provide.

(3) A proposal for a language with modules should come from those who have experience of them, which I do not.

However, the practical need for separate compilation doesn't demand modules. To satisfy this need, the spec directives (Section 4.3) seem rather simple and can be seen as consolidation, not extension. Note that they do not change the meaning of programs at all; the unifying idea, that a program is (ignoring the directives) just a single declaration, is preserved.

This does not prevent a revision of ML with modules being adopted, say within the next year or two, as a development of the present proposal.
11.2 Escapes and Traps

The exception mechanism in this proposal came from Alan Mycroft; it also owes something to an idea from Brian Monahan. The views expressed on escaping and trapping ranged from contentment with the existing ML mechanism, in which escapes can only carry tokens as values, to approval of the present proposal in which exceptions can carry arbitrary values, and can even be polymorphic. (To be fair, there was also the view that both escapes and references are impure and should be banned; this was a minority view.)

The proposal is easy to implement, easy to describe semantically, and specialises smoothly to the existing mechanism. Thus, although some could predict its effect on programming style, it seems safe to adopt it. No other scheme was put forward which fits so well with the polymorphic type discipline.
11.3 Data types and parameter matching

The proposal to import the HOPE data constructs, with use of constructors in varstructs, met with approval from most people. It could have been combined with Cardelli’s records and variants, allowing named fields in records, but to do so would have led to an embarrassment of riches. Luca Cardelli was kind enough to accept it.

There was a lot of discussion about the constraints that should be placed upon the varstructs in a match, and the order in which these should be matched. The main point is that, if the set of varstructs in a match is required to be closed under unification, and the order of matching is from more specific to less specific (which still leaves freedom) the implementation can be very efficient. On the other hand, left-to-right order is easy for users to understand.

It came out in discussion with Dave MacQueen (who has studied this question in detail) that in fact the above requirement is not essential to gain the efficient implementation. Roughly, it appears that a compiler can process an arbitrary match to attain the unification-closed property, and can then adopt an efficient order of matching which is semantically equivalent to left-to-right. I have therefore proposed left-to-right order, to gain a simple universal rule in the language, and the compiler will issue a warning if a more specific varstruct follows a less specific one in a match, or if the varstructs in a match are not exhaustive.
11.4 Input/Output

There was not much general discussion of I/O, but Luca Cardelli proposed the use of streams (in place of my Tentative proposal) and this seems very clean.

The proposed standard functions are obviously too few for convenient use. But library functions should be defined from them. Cardelli proposed a function \texttt{copystream : stream \rightarrow stream}; I left this out because Kenn Mitchell suggested that it might demand a special choice of implementation. In any case, it can be defined if we are prepared to use an auxiliary file \texttt{COPY}:

\begin{verbatim}
val copystream (s) = (putstream "copy": s ; getstream "COPY")
\end{verbatim}

In my Tentative proposal I suggested overloaded functions \texttt{read} and \texttt{write}, which would read and write data of arbitrary (mono)type, in the form of ML constructors. This seems a bad idea; it would involve delicate interaction with the ML compiler (perhaps not only the parser) to take account of the currently defined value constructors.
11.5 Clausal forms for function declaration

There was some discussion of how rich a form of clausal function declaration to allow. The present proposal goes as far as allowing e.g.

\[
\begin{align*}
\text{val rec } \text{member} \ (x, \text{nil}) &= \text{false} \\
& \quad \text{member} \ (x, y::l) = x = y \lor \text{member} (x, l)
\end{align*}
\]

but forbidding the form

\[
\begin{align*}
\text{val rec } \text{member} \ x \ \text{nil} &= \text{false} \\
& \quad \text{member} \ x \ (y::l) = x = y \lor \text{member} x \ l
\end{align*}
\]

It was agreed that the second form requires careful explanation of when pattern matching of arguments fails. It was not fully agreed whether the form is important enough to justify the explanation. With this doubt, the decision to forbid the curried form seems justified, acknowledging that to mix currying with alternative patterns risks confusion.

11.6 Tuples vs. pairs

Most people with whom I discussed it prefer the tuple type \(t_1 \times \cdots \times t_n\) to the simple pair type \(t_1 \times t_2\). This is the scheme adopted in HOPE. I now prefer it too. Thus \(t_1 \times \cdots \times t_n\) abbreviates \((t_1, \ldots, t_n)\), where \(t_n\) is a different type constructor for each \(n \geq 2\). It follows that \((x, y, z)\) is distinct from \((x, (y, z))\), unlike original ML. I guess that the choice of a simple pair type in original ML dates from the time when we had only a small finite collection of standard type constructors (\(\rightarrow\#\#\)) and no user-defined ones.
11.7 Syntactic matters

(1) **end**: Everyone has different dos and don'ts (or ods and tnocks) about terminating keywords. Old ML gave people headaches with things like `where`. There is no perfect compromise between ambiguity and verbosity. In the absence of inspiration, at least the present proposal can probably be remembered:

```
end delimits only let, where, local and abstract
```

Mnemonic: end is needed in any construct which slider a declaration to other things with alphabetic keywords!

See also (10) below

(2) Atomic variables: This wasn't discussed, but has certainly caused trouble in the past. We don't want

```
val f : x:int , y:bool = ...
```

in place of "val f (x:int, y:bool) = " . In fact, we want

```
val f : (x:int, y:bool):string = ...
```

to declare a function of type `int # bool → string`. This points to having argument variables atomic (closed) in these sugared forms of declaration. On the other hand, we do wish to allow non-atomic variables in function abstraction, e.g.

```
fun x : int, y : bool . ---- , or fun x, y . ----
```

To avoid possible confusion, it seems wise to omit Curried fun forms like

```
fun (x : int) (y : bool) . ....
```

because the variables here should be atomic, and this would blur the simple mnemonic that parameter variables need only be atomic in derived function declaration forms. Besides, Curried fun forms are not too frequent in most practical programming styles.
(3) Distices; again, there wasn't much general discussion of the need for these. Don Samuelson gave helpful advice about how to manage them. In the end, I couldn't find enough enthusiasm in anyone (particularly in myself) to justify the careful explanation that they need — surprisingly tricky when you try it — so they are omitted.

(4) **Abstract vs concrete syntax**: Luca Cardelli points out that eventually (soon for some people) structure editors will reduce or remove our worries about precedence, delimiters, lexical analysis etc. Hence we should be sure to define the abstract syntax of the language, and only recommend the concrete syntax. It is a relief to adopt this view, particularly as human beings degenerate into animals when they argue about concrete syntax, just as they do when they drive motor cars! But the recommendation needs to be quite strong, because people communicate with other people in concrete syntax. This proposal hasn't dismissed abstract syntax; it can be deduced quite easily from the recommended concrete forms.
(5) **Type abbreviations:** There was a preponderance (among those who have written fairly large programs) of the need for type abbreviations. The consensus — not that everyone expressed a view — seems to be that it is a dirty but expedient device, and that it should not have semantic significance.

(6) **let:** There was some doubt as to whether `let` should be (a) on a par with `type` as a way of characterising a binding, or (b) an indication that a binding is of limited scope. Possibly this doubt arose from the unfortunate way in which old ML confused the two, which people had come to accept. The majority favoured nonconfusion, if it could be achieved without pain. The present `val` (on a par with `type`) was accepted by at least some people — certainly as preferable to `var` which was used in this role in my tentative proposal.

(7) **Underbar:** Most people want it in identifiers. Some people are happy to have `\` as a wildcard variable as well. I don't like this; the two uses can lead to some odd uses, e.g., `\-a\-b\-` which in this proposal is written `\$a-b\$`, meaning the same as `op a-b(\$,\$)`.

(8) `\` and `\=`: No one liked my use of `\` in bindings, in my tentative proposal, to avoid overloading the symbol `\=`. Luca Cardelli is still happy to overload `\=` but a fair compromise seems to be `\=` in bindings,
(9) **Sequence connectors**: there are many different places where sequences of like forms occur (identifiers in some directives, bindings in declaration, tuples). The temptation is to use a comma as the connective rather often. The proposal avoids this; commas are only used for tupleung — including tuples of type arguments of type constructors. "and" is used for bindings in declaration and for the degenerate bindings which occur in spec directives. When a sequence of bare identifiers is required, a space is the connective.

One connective, the semi-colon, is a bit overloaded; this is tolerated in deference to the ML tradition of explicit list formation and the general programming tradition of statement (or command) sequencing.

(10) **Delimiters again**: The low precedence of semi-colon will avoid some tendencies to error in syntactic grouping. Beyond this, the habit of always enclosing a match in parentheses will remove other such tendencies. This habit could even be made a requirement; on balance I think I prefer to leave it optional.
\textbf{Appendix 1: Expressions}

\[ \text{aexp} ::= \]
\[ \{ \text{exp?, var} \} \]
\[ \{ \text{exp?, con} \} \]
\[ [\text{exp}_1, \ldots, \text{exp}_n] \]
\[ (\text{exp}) \]

\[ \text{exp} ::= \]
\[ \text{aexp} \]
\[ \text{exp} \ \text{aexp} \]
\[ \text{exp : ty} \]
\[ \text{exp}_1 \ \text{id} \ \text{exp}_2 \]
\[ \text{exp}_1 \ & \ \text{exp}_2 \]
\[ \text{exp}_1 \ \text{or} \ \text{exp}_2 \]
\[ \text{exp}_1, \ldots, \text{exp}_n \]
\[ \text{raise exp id exp} \]
\[ \text{escape exp} \]
\[ \text{if exp then exp}_1 \ \text{else exp}_2 \]
\[ \text{while exp}_1 \ \text{do exp}_2 \]
\[ \text{let \ dec in exp \ and exp \ where \ dec \ end} \]
\[ \text{case exp of match} \]
\[ \text{exp handle exp id match} \]
\[ \text{exp trap match} \]
\[ \{ \text{exp}_1 ? \ \text{exp}_2 \]
\[ \text{fun match} \]
\[ \text{exp}_1 ; \ \text{exp}_2 \]

\[ \text{match} ::= \usl. \text{exp}_1 | \ldots | \usn. \text{exp}_n \]

\[ \dagger \text{These three forms are of equal precedence and left-associative} \]
Appendix 2: VARSTRUCTS and VARIABLE BINDINGS

aus ::=

$               \quad \text{(Wildcard)}$

{obj} var

con

[us1; ... ; usn]

(us)

us ::=

aus

con aus

us : ty

us1 con us2

us1, ..., usn

ut ::=

us {: ty} == exp

\dagger \left\{ \begin{array}{l}
{obj} var aus1 {: ty} == exp1 | ... | {obj} var ausn {: ty} == expn \\
{exp} var aus1 ... ausn {: ty} == exp
\end{array} \right. \quad \text{(clausal function)}

ut1 and ut2

rec ut

\dagger \text{In case "obj var" is admissible (because var has infix status) then}

"aus var aus'" may replace "obj var (aus, aus')"

in these forms.
APPENDIX 3: TYPES, TYPE BINDINGS AND DECLARATIONS

\[ \text{ty} ::= \]
\[ \text{ty var} \quad \text{(type variable)} \]
\[ \text{ty-seq}\text{ty-con} \quad \text{(type construction)} \]
\[ \text{ty}_1 \# \ldots \# \text{ty}_n \quad \text{(tuple type)} \]
\[ \text{ty}_1 \to \text{ty}_2 \quad \text{(function type)} \]

\[ \text{tb} ::= \]
\[ \{\text{ty var-seq}\text{ty-con} = \text{con}_1 \{\text{ty}_1\} \ldots \text{con}_n \{\text{ty}_n\} \quad \text{(simple)} \]
\[ \text{ty var-seq}\text{ty-con} \leftrightarrow \text{ty} \quad \text{(isomorphism)} \]
\[ \text{tb}_1 \text{ and } \text{tb}_2 \quad \text{(simultaneous)} \]
\[ \text{rec tb} \quad \text{(recursive)} \]

\[ \text{dec ::=} \]
\[ \text{val tb} \quad \text{(value declaration)} \]
\[ \text{type tb} \quad \text{(type declaration)} \]
\[ \text{abstract tb with dec end} \quad \text{(abstract type declaration)} \]
\[ \text{exception end1}\{\text{ty}_1\} \ldots \text{endn}\{\text{ty}_n\} \quad \text{(exception declaration)} \]
\[ \text{local dec1 in dec2 end} \quad \text{(local declaration)} \]
\[ \text{use "filename"} \quad \text{(use file as declaration)} \]
\[ \text{exp} \quad \text{(declaration of "it")} \]
\[ \text{dec1 ; dec2} \quad \text{(declaration sequence)} \]
DIRECTIVES, COMMANDS and PROGRAMS

**Directive ::=**

- **infix id1 ... idn** (give infix status)
- **nonfix id1 ... idn** (cancel infix status)
- **shorttype \{tvar-seq1\} tyc1 == tyl and ... and \{tvar-seqn\} tycn == tyn** (type abbreviation)
- **longtype tyc1 ... tycn** (cancel type abbreviation)
- **spec var \{obj{id1 : tyl} and ... and obj{idn : tyn}\** (specify value variables)
- **spec type tl** (specify types with value constructors)
- **spec type \{tvar-seq1\} tyc1 and ... and tvar-seqn tycn** (specify type constructors)
- **spec exception id1 : tyl and ... and idn : tyn** (specify exception identifiers)

**Command ::=**

- **dec ;**
- **directive ;**

A PROGRAM is a sequence of commands.
## APPENDIX 5: PREDECLARED VARIABLES AND CONSTANTS

### Nonfixes

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nil</code></td>
<td>'a list</td>
<td></td>
</tr>
<tr>
<td><code>head</code></td>
<td>'a list -&gt; 'a</td>
<td></td>
</tr>
<tr>
<td><code>tail</code></td>
<td>'a list -&gt; 'a list</td>
<td></td>
</tr>
<tr>
<td><code>map</code></td>
<td>('a -&gt; 'b) -&gt; 'a list -&gt; 'b list</td>
<td></td>
</tr>
<tr>
<td><code>rev</code></td>
<td>'a list -&gt; 'a list</td>
<td></td>
</tr>
<tr>
<td><code>true</code></td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td><code>false</code></td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td><code>not</code></td>
<td>bool -&gt; bool</td>
<td></td>
</tr>
<tr>
<td><code>intofstr</code></td>
<td>'int -&gt; 'string</td>
<td></td>
</tr>
<tr>
<td><code>stringofint</code></td>
<td>'int -&gt; 'string</td>
<td></td>
</tr>
<tr>
<td><code>explode</code></td>
<td>'string -&gt; 'string list</td>
<td></td>
</tr>
<tr>
<td><code>implode</code></td>
<td>'string list -&gt; 'string</td>
<td></td>
</tr>
<tr>
<td><code>~</code></td>
<td>'int -&gt; 'int</td>
<td></td>
</tr>
<tr>
<td><code>!</code></td>
<td>'a ref -&gt; 'a</td>
<td></td>
</tr>
<tr>
<td><code>ref</code></td>
<td>'a -&gt; 'a ref</td>
<td></td>
</tr>
</tbody>
</table>

### infixes

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>*</code></td>
<td>int # int -&gt; int</td>
<td>L</td>
</tr>
<tr>
<td><code>div</code></td>
<td></td>
<td>L</td>
</tr>
<tr>
<td><code>mod</code></td>
<td></td>
<td>L</td>
</tr>
<tr>
<td><code>+</code></td>
<td>int # int -&gt; int</td>
<td>L</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>'string # 'string -&gt; 'string</td>
<td>L</td>
</tr>
<tr>
<td><code>@</code></td>
<td>'a list # 'a list -&gt; 'a list</td>
<td>R</td>
</tr>
<tr>
<td><code>;</code></td>
<td>'a list # 'a list -&gt; 'a list</td>
<td>R</td>
</tr>
<tr>
<td><code>=</code></td>
<td>'y # 'y -&gt; bool</td>
<td>R</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>int # int -&gt; bool</td>
<td>R</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>int # int -&gt; bool</td>
<td>R</td>
</tr>
<tr>
<td><code>==</code></td>
<td></td>
<td>R</td>
</tr>
<tr>
<td><code>&lt;&gt;</code></td>
<td></td>
<td>R</td>
</tr>
<tr>
<td><code>:=</code></td>
<td>'a ref # 'a -&gt; unit</td>
<td>R</td>
</tr>
</tbody>
</table>

### Special Constants

- `(unit)`
- `0, 1, 2, 3, ...` : int
- `"" ...""` : string

### Precedence

- **Precedence 0**: `:=` : `mu ref # mu -> unit`
- **Precedence 1**: `0 : ('b -> 'c) # ('a -> 'b)`
- **Precedence 2**: `@ : 'a list # 'a list -> 'a list`
- **Precedence 3**: `+ : int # int -> int`
- **Precedence 4**: `< : 'string # 'string -> 'string`
- **Precedence 5**: `* : int # int -> int`

### Notes

- All marked (+) are constants or constructors, so may appear in varstructs.
- "ref" is polymorphic, with type 'a -> 'a ref, in varstructs only.
- "explode" yields a list of strings of length 1; "implode" is iterated concatenation (*
- `μ` stands for any monotype, and `γ` is explained in Section 7.2.
- All function escape with their names in unpacked arguments.