A PROPOSAL FOR STANDARD ML (TENTATIVE)

Robin Milner, April '83.

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 constructions (pairing, lists) but also for constructions 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 updateable) 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 (Burstall et al) which, in my view, provides the most natural rounding-out of ML w.r.t. data types and data parameters.

A word about environment operations : as Standard ML develops, it emerges that Luca's important environment operator "enc" (such that in "dec1 enc dec2" dec1 exports into dec2 and both dec1 and dec2 export from the whole) gets entirely conflated with the semicolon — 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

$$\underline{\text{fun}}\ v1.exp1 \mid \ldots \mid vn.expn$$

which generalises the familiar lambda abstraction "$\lambda v.exp$". When applied, this "function" matches its argument to each varstruct vi in left-to-right order until a match succeeds (which binds any variables in vi to components of the argument value) then evaluates the corresponding $expi$. 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 lunchtime 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 v1 from an escape generated by evaluation of exp1. (This latter will escape from the entire application.) Note that n=1 gives 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 presuppose bool) presupposes any standard type. We discuss the constructions class by class.

(1) Expressions: In application "exp1 exp2" no order of evaluation is assumed. In original ML, exp1 was evaluated (to a function) before exp2. In Cardelli's VAX ML, apparently 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 exp1 ... expn"; but it still seems worth giving the implementor freedom. Likewise in "exp1, exp2" no order of evaluation is assumed.

In "escape exp" exp will (in the standard language) be token-valued. "escape" seems better than "fail". In "exp trap v1.exp1|..|vn.expn" the vi will (in Standard ML) be of type token. We could have chosen "exp1 trap exp2", where exp2 is of type "token → ..." ; but then it is irksome to have to specify how escapes in evaluating exp2 are treated. As it is, no 'order of evaluation' question arises — and one can always get the effect (mostly) of "exp1 trap exp2" by writing "exp1 trap t.(exp2 t)"

In "let 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 "abstype". 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 "λv.exp" — we can probably implement it well enough in terms of the closures of Luca's abstract machine FAM, using traps.

(2) Varstructs: The wild card "any" — matching anything — is not strictly necessary, since "()" can play this rôle 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 rôle).

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 cons (say) the forms "cons v", "cons(v, v')", "cons any" and "cons (any, any)" are all admissible varstructs — but not "cons" by itself.

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

(4) 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 Bare ML; 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) "local dec in dec" plays the rôle of Luca's "inside". The prefixed keyword "local" should help in reading. It corresponds precisely to "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 ";" used for sequencing plays the rôle of Luca's "enc". I chose it because (a) it underlines the fact that "enc" is associative (unlike "inside"!); (b) it has the same effect as ";" separating top-level declaration commands, and its weak binding power ensures that at top level "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

DEC10 ML	VAX ML	STANDARD ML
<u>let</u> z = x+y <u>in</u> <u>let</u> w = z*z <u>in</u>	<u>let</u> z = x+y <u>enc</u> w = z*z <u>in</u>	<u>let</u> <u>var</u> z ← x+y ; <u>var</u> w ← z*z <u>in</u> ------

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

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

(v) The keyword "<u>var</u>" seems appropriate to match "<u>data</u>" and "<u>abstype</u>"

(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 <u>with</u> part. Note that, for each type constructor "tycon" introduced in such a binding, the constructor "abstycon" is available in the <u>with</u> part both in expressions and in varstructs; this means that "reptycon" is no longer necessary.

Finally, the locality of <u>data</u> and <u>abstype</u> declarations 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 at 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

Com ::= .... | infix id1 .... idn {k} {ass} | nonfix id1 ... idn

where
k ::= 1 | 2 | ··· (precedence)
ass ::= left | right (association)

are added. These commands establish the infix status of identifiers in expressions only, not within types. Default values for k, ass are 1, left. The pairing operator "," has precedence 1; thus it binds more loosely than every predefined infix except ":=". Infixes bind more loosely than application or type constraint, more tightly than all other expression constructs. Examples:

f a + b means (f a) + b

a + b trap m means (a + b) trap m

Second, every non-infixed occurrence of an identifier with infix status must be preceded by "infix". Infixed occurrences are only allowed in expressions and varstructs, not in Data binding constructions.

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

a + b means infix + (a, b)
not infix + a b

See Table 6 for predefined infixes

4. References and Equality in Standard ML

Although Luis Damas produced a subtle form of typechecking to allow polymorphic assignment to references, I propose that the 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 commit the language just 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 construction α ref.

(2) The function ref : $\mu \to \mu$ ref at all monotypes μ, for creating new references. In varstructs, however, ref : $\alpha \to \alpha$ ref may be used polymorphically.

(3) The function infix := : μ ref # $\mu \to$ unit at all monotypes, for assignment.

Besides this, the standard contents function @ : α ref $\to \alpha$, defined by "bar @(ref x) ← x", is provided.

Example. Here is how to define α row, the type of polymorphic one dimensional arrays. Its operations are parameterised on polymorphic functions

new : α → α ref
assign : (α ref # α) → unit

They are:

newrow new : α → int → α row
("newrow new v k" returns a row of length k with value v in each cell)

assignrow assign : α → α row → int → unit
("assignrow assign v R k" places value v in k^{th} cell of row R)

controw : α row → int → α
("controw R k" returns value of k^{th} cell in R)

```
abstype α row <=> α ref list
with newrow ref v k <- absrow(newlist k) where rec newlist <-
                  fun 0. nil | k. ref v :: newlist (k-1)
and assignrow (infix :=) v (absrow L) k <- assignlist L k where rec assignlist <-
                  fun nil. escape `outofrange`
                    | c::L. fun(1. c := v  | k. assignlist L (k-1))
and controw (absrow L) k <- contlist L k where rec contlist <-
                  fun nil. escape `outofrange`
                    | c::L. fun (1. @c | k. contlist 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:

```
abstype introw <=> int row
with newintrow (n:int) <- absintrow o (newrow ref n)
and assignintrow (n:int)(absintrow R) <- assignrow (infix :=) 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 :(α ref list -> α list) <- map @
```

<u>Equality</u>: Following Cardelli'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 <u>should not determine</u> its meaning.

Hence we allow the infixed predicates $=, <> : \Gamma \# \Gamma \rightarrow bool$ where Γ is any type built from <u>polymorphic reference types ty ref</u> by <u>data</u> type constructors.

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

<u>Procedural programming</u>: it seems best to throw out all the wild forms of looping in DEC10 ML, and just extend expressions by

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

Too bad!

5. External Files in Standard ML

By virtue of the abbreviation (see Table 5)

exp ⟼ var any ← exp

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

dec ::= .... | use Token

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

local use `TREES.ML`
in abstype ... % the abstract type of balanced trees, say % ...

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

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

All this seems to admit a later extension such as

dec ::= .... | 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 Δ, ie. monotypes built with data type constructors, may be input or output — using current infix status for constructors.

Then we can do with just six functions

(1) openread : token ⟶ unit

"openread \`f\`" rewinds f to allow reading from the start; escapes with \`NO FILE\` if f does not exist.

(2) openwrite : token ⟶ unit

"openwrite \`f\`" creates a new empty file f to allow writing from the start.

(3) read : token ⟶ Δ

How is the type Δ conveyed to the operation? in what form?

"read \`f\`" reads the shortest value (construction) of type Δ.
Escapes with
- \`FAILED 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 → Δ → 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 "readln" 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".

<u>Remarks</u>. 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 types of input values by explicit type expressions, e.g "read``next : int list"

7. Miscellaneous

(1) Sections (DEC10 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 adhoc 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

parse : Token → Δ
unparse : Δ → Token (Δ any mono-data type)

are proposed. Since Δ can be "int", these subsume the old intoftok, tokofint functions (and they are the identity when Δ = 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

Id ::= Any sequence of letters and digits, starting with a letter, followed possibly by one or more primes (')

Any sequence of one or more of the symbols

! # $ & + - / : < = > ? @ \ ↑ _ ~ | * . ,

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

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

| . ← : ⟺

2. Constructors, Variables

Con ::= any identifier declared by a Data binding as a constructor or constant, within the scope of that binding; also the identifier absid, within the with part of an Abstract type binding which introduces id as a type constructor.

Var = Id \ Con (Thus, constructors cannot be re-declared as variables within their scope).

3. Type variables

Tyvar ::= 'a | 'b | ... | 'z (pronounced alpha, beta, ... ?)

4. Type constructors

Tycon ::= any type constructor or constant declared by a Data or 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 2 : SYNTAX OF BARE ML

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

EXPRESSIONS

exp ::=	var		(variable)
	con		(constant, constructor)
	exp1 exp2	L	(application)
	exp : ty		(constraint)
	exp1 , exp2	R	(pairing)
	escape exp		(escape)
	exp trap match		(escape trapping)
	let dec in exp		(local declaration)
	fun match		(function abstraction)
match ::=	v1.exp1 | ... | vn.expn		(matching)

VARSTRUCTS

v ::=	any		(wild card)
	var		(variable)
	con {v_seq}		(construction)
	v : ty		(constraint)
	v1 , v2	R	(pairing)

TYPES

ty ::=	tyvar		(type variable)
	{ty_seq} tycon	L	(type construction)
	ty1 # ty2	R	(product type)
	ty1 → ty2	R	(function type)

DECLARATIONS

dec ::=	{rec} var vb	(variables)
	{rec} data db	(data)
	{rec} abstype ab with dec	(abstract types)
	local dec1 in dec2	(local)
	dec1 ; dec2	(sequence)

VARIABLE BINDINGS

vb ::=	v ← exp	(simple)
	vb1 and vb2	(simultaneous)

DATA BINDINGS

db ::=	{tyvar_seq} id ← constrs	(simple)
	db1 and db2	(simultaneous)
constrs ::=	id1 {of ty1} | ... | idn {of tyn}	

ABSTRACT BINDINGS

ab ::=	{tyvar_seq} id ⇔ ty	(simple)
	ab1 and ab2	(simultaneous)

COMMANDS

Com ::=	dec	(declaration)
	exp	(expression)

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 binding ('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 varstruct; the need for "repmonoid" is naturally avoided.

TABLE 4 : PREDEFINED DATA DECLARATIONS FOR STANDARD ML

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 | c_1 of token | ... | c_n of token ;

   (where $\{c_1, \ldots, c_n\}$ is the character set)

CONVENTION : $c_{i_1}(\cdots(c_{i_k}$ empty$)\cdots)$ is represented instead by \`$c_{i_1}\cdots c_{i_k}$\`

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

CONVENTION : zero, pos(succ^{k-1}one), neg(succ^{k-1}one) 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 5 : STANDARD ABBREVIATIONS AND ALTERNATIVE FORMS

Note : the abbreviations are not to suggest implementation, but merely to show that there is a semantically equivalent "Raw ML" form.

1. Expressions

quit	↦ escape \`quit\`	
exp1 or exp2	↦ exp1 trap any . exp2	
exp1 orif "t1 .. tn" exp2	↦ exp1 trap (\`t1\` . exp2 | ... | \`tn\` . exp2 | x . escape x)	(n≥0)
exp where dec	↦ let dec in exp	
case exp of match	↦ (fun match) exp	
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)
"t1 .. tn"	↦ [\`t1\` ; ... ; \`tn\`]	(n≥0)
exp1 ; exp2	↦ let var any ← exp1 in exp2	
while exp1 do exp2	↦ f() where rec f() ← if exp1 then (exp2 ; f()) else ()	

2. Varstructs

[v1 ; ... ; vn]	↦ v1 :: ... :: vn :: nil	(n≥0)
"t1 .. tn"	↦ [\`t1\` ; ... ; \`tn\`]	(n≥0)

3. Variable Bindings

id v1 .. vn {: ty} ← exp	↦ id ← fun v1 ... vn {: ty} . exp	(n≥1)
? v1 id v2 v3 .. vn {: ty} ← exp (v1 id v2) v3 ... vn {: ty} ← exp	↦ infix id v1 ... vn {: ty} ← exp (when id has infix status)	(n≥2)

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 treating external ML files as declarations; see Section 5.

TABLE 6: PREDEFINED IDENTIFIERS IN STANDARD ML

NONFIXES

	Type	Meaning
~	$int \rightarrow int$	minus
@	$\alpha\ ref \rightarrow \alpha$	contents
fst	$\alpha \# \beta \rightarrow \alpha$	unpairing
snd	$\alpha \# \beta \rightarrow \beta$	unpairing
hd	$\alpha\ list \rightarrow \alpha$	lists
tl	$\alpha\ list \rightarrow \alpha\ list$	lists
map	$(\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$	lists
rev	$\alpha\ list \rightarrow \alpha\ list$	lists
explode	$token \rightarrow token\ list$	tokens
implode	$token\ list \rightarrow token$	tokens
not	$bool \rightarrow bool$	negation
unparse	$\Delta \rightarrow token$	unparsing
parse	$token \rightarrow \Delta$	parsing
ref	$\mu \rightarrow \mu\ ref$	new reference (in expressions)
	$\alpha \rightarrow \alpha\ ref$	reference (in varstructs)
it		value of last expression
openread	$token \rightarrow unit$	input/output
openwrite	$token \rightarrow unit$	input/output
read	$token \rightarrow \Delta$	input/output
write	$token \rightarrow \Delta \rightarrow unit$	input/output
readchar	$token \rightarrow token$	input/output
writechar	$token \rightarrow token \rightarrow unit$	input/output

INFIXES

	Type	Association	Precedence	Meaning
*	$int \# int \rightarrow int$	L	50	Arithmetic
div	$int \# int \rightarrow int$	L	50	Arithmetic
mod	$int \# int \rightarrow int$	L	50	Arithmetic
/\	$bool \# bool \rightarrow bool$	L	50	Conjunction
+	$int \# int \rightarrow int$	L	40	Arithmetic
−	$int \# int \rightarrow int$	L	40	Arithmetic
\/	$bool \# bool \rightarrow bool$	L	40	Disjunction
::	$\alpha \# \alpha\ list \rightarrow \alpha\ list$	R	30	List cons
++	$\alpha\ list \# \alpha\ list \rightarrow \alpha\ list$	R	30	List append
=	$\Gamma \# \Gamma \rightarrow bool$	L	20	Comparison
<>	$\Gamma \# \Gamma \rightarrow bool$	L	20	Comparison
<	$int \# int \rightarrow bool$	L	20	Integer order
>	$int \# int \rightarrow bool$	L	20	Integer order
<=	$int \# int \rightarrow bool$	L	20	Integer order
>=	$int \# int \rightarrow bool$	L	20	Integer order
o	$(\beta \rightarrow \gamma) \# (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$	L	10	Composition
&	$(\alpha \rightarrow \beta) \# (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$	L	10	Rev. composition
#	$(\alpha \rightarrow \beta) \# (\alpha \rightarrow \gamma) \rightarrow \alpha \rightarrow (\beta \# \gamma)$	R	10	Pairing of functions
,	$\alpha \# \beta \rightarrow \alpha \# \beta$	R	1	Pairing
:=	$\mu\ ref \# \mu \rightarrow unit$	L	0	Assignment

Note: μ is any monotype
Δ is any mono-data-type
Γ is any type built from reference types by data type constructors.

TABLE 7' : EXPRESSIONS IN STANDARD ML

exp ::=

var	(variable)
con	(constant, constructor)
exp1 exp2	(application)
exp : ty	(type constraint)
exp1 infix exp2	(infixed application)
escape exp	(escape with token)
quit	(escape with 'quit')
if exp1 then exp2 else exp3	(conditional)
case exp of match	(case analysis)
while exp1 do exp2	(iteration)
exp trap match	(escape trapping)
exp1 or exp2	(universal escape trapping)
exp1 orif "t1 ..tn" exp2	(selective escape trapping)
exp1 ; exp2	(sequence)
[exp1 ; .. ; expn]	(list ; n≥0)
"t1 ... tn"	(token list ; n≥0)
exp where dec	(local declaration)
let dec in exp	(local declaration)
fun match	(function abstraction)
fun v1 ... vn . exp	(Curried abstraction ; n≥1)

match ::= v1 . exp1 | ... | vn . expn (Matching ; n≥1)

TABLE 8 : DECLARATIONS AND BINDINGS IN STANDARD ML

DECLARATIONS:

dec ::=	exp	(vacuous declaration)
	use 'filename'	(external ML file)
	{rec} var vb	(variable declaration)
	{rec} data db	(data declaration)
	{rec} abstype ab with dec	(abstract type declaration)
	local dec1 in dec2	(local declaration)
	dec1 ; dec2	(declaration sequence)

VARIABLE BINDINGS:

vb ::=	v ← exp	(simple binding)
	id v1 … vn {: ty} ← exp	(function binding : $n \geq 1$)
?	(v1 infix v2) v3 … vn {: ty} ← exp	(infix function binding : $n \geq 2$)
	vb1 and vb2	(simultaneous binding)

DATA BINDINGS:

db ::=

{tyvar_seq} id ← constrs	(simple)
db1 and db2	(simultaneous)

constrs ::=

id1 {of ty1} | … | idn {of tyn}	($n \geq 1$)

ABSTRACT TYPE BINDINGS:

ab ::=

{tyvar_seq} id ⟺ ty	(simple)
ab1 and ab2	(simultaneous)

TABLE 9 : VARSTRUCTS, TYPES and COMMANDS IN STANDARD ML

VARSTRUCTS :

v ::=	<u>any</u>	(wild card)
	var	(variable)
	con {v_seq}	(construction)
	<u>ref</u> v	(reference)
	abstycon v	(abstraction)
	v : ty	(type constraint)
	v1 infix v2	(infixed construction)

TYPES :

ty ::=	tyvar	(type variable)
	{ty_seq} tycon	(type construction)
	ty1 # ty2	(product type)
	ty1 → ty2	(function type)

COMMANDS :

com ::=	dec	(declaration)
	exp	(expression)
	<u>infix</u> id1 ... idn {prec} {ass}	(infix status)
	<u>nonfix</u> id1 ... idn	(nonfix status)

prec ::= 1 | 2 | ...

ass ::= <u>left</u> | <u>right</u>