ML under Unix

Luca Cardelli

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

ML is a statically-scoped **functional** programming language. Functions are first class objects which can be passed as parameters, returned as values and embedded in data structures. Higher-order functions (i.e. functions receiving or producing other functions) are used extensively.

ML is an **interactive** language. An ML session is a dialogue of questions and answers with the ML system. Interaction is achieved by an incremental compiler, which has some of the advantages of interpreted languages (fast turnaround dialogues) and of compiled languages (high execution speed).

ML is a **strongly typed** language. Every ML expression has a type, which is determined statically. The type of an expression is usually automatically inferred by the system, without need of type definitions. The ML type system guarantees that any expression that can be typed will not generate type errors at run time. Static typechecking traps at compile-time a large proportion of bugs in programs.

ML has a **polymorphic type system** which confers on the language much of the flexibility of type-free languages, without paying the conceptual cost of run-time type errors or the computational cost of run-time typechecking.

ML has a rich collection of data types. Moreover the user can define new **abstract data types**, which are indistinguishable from the system predefined types. Abstract types are used extensively in large programs for modularity and abstraction.

ML has an **exception-trap** mechanism, which allows programs to handle uniformly system and user generated exceptions. Exceptions can be selectively trapped, and exception handlers can be specified.

ML programs can be grouped into **modules**, which can be separately compiled. Dependencies among modules can be easily expressed, and the sharing of common submodules is automatically guaranteed. The system keeps track of module versions to detect compiled modules which are out of date.

This manual describes an implementation of ML running on VAX under the Unix operating system.

Unix is a trade mark of Bell Laboratories.
VAX is a trade mark of Digital Equipment Corporation.

Contents

1. Introduction

ML is a *functional* programming language. Functions are first class objects which can be passed as parameters, returned as values and embedded in data structures. Higher-order functions (i.e. functions receiving or producing other functions) are used extensively. Function application is the most important control construct, and it is extremely uniform: all functions take exactly one argument and return exactly one result. Arguments and results can however be arbitrary structures, thereby achieving the effect of passing many arguments and producing many results. Arguments to functions are evaluated before the function calls.

ML is an *interactive* language. An ML session is a dialogue of questions and answers with the ML system. Interaction is achieved by an incremental compiler, which has some of the advantages of interpreted languages (fast turnaround dialogues) and of compiled languages (high execution speed).

ML is *statically scoped*. All the variables are associated with values according to where they occur in the program text, and not depending on run-time execution paths. This avoids name conflicts in large programs, because variable names can be hidden in local scopes, and prevents accidental damage to preexisting programs. Static scoping greatly improves the security and, incidentally, the efficiency of an interactive language, and it is necessary for implementing higher-order functions with their conventional mathematical meaning.

ML is a *strongly typed* language. Every ML expression has a type, which is determined statically. The type of an expression is usually automatically inferred by the system, without need for type definitions. This *type inference* property is very useful in interactive use, when it would be distracting having to provide all the type information. However, it is always possible to specify the type of any expression or function, e.g. as good documentation practice in large programs. The ML type system guarantees that any expression that can be typed will not generate type errors at run time (for some adequate definition of what a type error is). Static typechecking traps at compile-time a large proportion of bugs in programs that make extensive use of the ML data structuring capabilities (typechecking does not help in numerical programs!). Usually, only truly "logical" bugs are left after compilation.

ML has a *polymorphic type system*. Type expressions may contain type variables, indicating, for example, that a function can work uniformly on a class of arguments of different (but structurally related) types. For example the length function which computes the length of a list has type α list -> int (which is automatically inferred from the obvious untyped recursive definition). Length can work on lists of any type (lists of integers, lists of functions, lists of lists, etc.), essentially because it disregards the elements of the list. The polymorphic type system confers on ML much of the flexibility of type-free languages, without paying the conceptual cost of run-time type errors or the computational cost of run-time typechecking.

ML has a rich collection of data types. Moreover the user can define new *abstract data types*, which are indistinguishable from the system predefined types. Abstract types are used extensively in large programs for modularity and abstraction. They put a barrier between the implementor of a package and its users, so that changes in the implementation of an abstract type will not influence the users, as long as the external interface is preserved. Abstract types also provide a clean extension mechanism for the language. If a new data type is needed which cannot be effectively implemented with the existing ML primitives (e.g. bitmaps for graphics), this can be still specified and prototyped in ML as a new abstract type, and then efficiently implemented and added to the basic system. This path has already been followed for arrays, which are not part of the language definition and can be considered as a predefined abstract type which just happens to be more efficient than its ML specification would lead one to believe.

ML has an *exception-trap* mechanism, which allows programs to uniformly handle system and user generated exceptions. Exceptions can be selectively trapped, and handlers can be specified.

ML programs can be grouped into *modules*, which can be separately compiled. Depedencies among modules can be easily expressed, and the sharing of common submodules is automatically guaranteed. The system keeps track of module versions to detect compiled modules which are out

```
                else true;

result:                 false : bool
```

The if part must be a boolean expression; two predefined constants true and false denote the basic boolean values; and the boolean operators are not, & (and) and or.

1.2. Declarations

Variables can be bound to values by *declarations*. Declarations can appear at the 'top-level', in which case their scope is *global*, or in scope blocks, in which case they are have a limited *local* scope spanning a single expression. Global declarations are introduced in this section, and local declarations in the next one.

Declarations are not expressions: they establish bindings instead of returning values. Value bindings are introduced by the keyword val, additional value bindings are prefixed by and:

```
declaration:    val a == 3
                and b == 5
                and c == 2;

bindings:           val a == 3 : int
                    val b == 5 : int
                    val c == 2 : int

expression:     (a + b) div c;

result:             4 : int
```

In this case we have defined the variables a, b and c at the top level; they will always be accessible from now on, unless redefined. Value bindings printed by the system are always prefixed by val, to distinguish them from type bindings and module bindings which we shall encounter later.

You may notice that all the variables must be initialized when introduced. Their initial values determine their types, which do not need to be given explicitly. The type of an expression is generally inferred automatically by the system and printed after the result.

Value declarations are also used to define functions, with the following intuitive syntax:

```
declaration:    val f x == x + 1;

binding:            val f : int -> int

declaration:    val g(a,b) == (a + b) div 2;

binding:            val g : (int # int) -> int

expression:     f 3, g(8,4);

result:             4,6 : int # int
```

(the arrow -> denotes the function-space type operator.)

The function f has one argument x. The result of a function is just the value of its body, in this case x+1. Arguments to functions do not need to be parenthesized in general (both in definitions and applications): the simple juxtaposition of two expressions is interpreted as a function application. Function application is treated as an invisible infix operator, and it is the strongest-binding

```
                in (a + b) div 2 end;

result:                 4 : int
```

here the identifiers a and b are bound to the values 3 and 5 respectively for the extent of the expression (a + b) div 2. No top-level binding is introduced; the whole let construct is an expression whose value is the value of its body.

Variables can be locally redefined, hiding the previous definitions for the extent of the local scope. Outside the local scope, the old definitions are not changed:

```
declaration:    val a == 3
                and b == 5;

bindings:               val a == 3 : int
                        val b == 5 : int

expression:     (let val a == 8 in a + b end), a;

result:                 13,3 : int # int
```

The body of a scope-block can access all the variables declared in the surrounding environment (like b), unless they are redefined (like a).

The and keyword is used to define sets of independent bindings: none of them can use the variables defined by the other bindings. However, a declaration often needs variables introduced by previous declarations. At the top-level this is done by introducing those declarations sequentially (here two declarations are typed on the same line):

```
declarations:   val a == 3;  val b == 2 * a;

bindings:               val a == 3 : int
                        val b == 6 : int
```

Similarly, declarations can be composed sequentially in local scopes by separating them by ';':

```
expression:     let     val a == 3;
                        val b == 2 * a
                in a,b end;

result:                 3,6 : int # int
```

In simple cases, the effect of ';' can be achieved by nested scope blocks; here is an equivalent formulation of the previous example:

```
expression:     let val a == 3
                in      let val b == 2 * a
                        in a,b end
                end;

result:                 3,6 : int # int
```

is actually a pair of values.

As every function actually has a single argument, the standard function definition looks like val f x == .. . We can put the definition of plus in this form as follows:

```
declaration:    val plus x ==
                    let val a,b == x
                    in a + b end;
```

What we did in the previous definitions of plus was to use a *pattern* (a,b) for x, which implicitly associated a and b respectively with the first and second components of x . These patterns come in many forms. For example a pattern [a;b;c] matches a list of exactly three elements, which are bound to a, b and c; a pattern first::rest matches a non-empty list whose first element is associated with first, and the rest to rest; similarly first::second::rest matches a list of at least two elements; etc. The most common patterns are of course tuples, like (a,b,c), but more complicated patterns can be constructed by nesting, like ([a;b],c,(d,e)::$). The special pattern $ matches any value without establishing any binding. Patterns can conveniently replace the use of selector functions for unpacking data.

Patterns do not only appear in function parameters: they can also be used in simple variable declarations:

```
declaration:    val f[a;b;c] == a,b,c;

declaration:    val a,b,c == f[1;2;3];

bindings:           val a == 1 : int
                    val b == 2 : int
                    val c == 3 : int
```

In the above example we have a function f returning three values, and the pattern a,b,c is used to unpack the result of f[1;2;3] into its components.

Recursive functions are defined by placing the keyword rec in a declaration, just before the function definition:

```
declaration:    val rec factorial n ==
                    if n = 0
                    then 1
                    else n * factorial(n-1);

binding:            val factorial : int -> int
```

The keyword rec is needed to identify the recursive occurrence of the identifier factorial with its defining occurrence. If rec is omitted, the identifier factorial is searched for in the environment outside the definition; in this case there is no previous definition of factorial, and an error would be reported.

The effect of rec propagates to whole groups of definitions; this is how mutually recursive functions are defined:

```
val rec f a == .. g ..
and g b == .. f .. ;
```

Functions can also be defined by *case analysis*, as a sequence of pattern-action pairs separated by vertical bars:

operations, whose types are:

```
nil     : 'a list
::      : ('a # 'a list) -> 'a list
null    : 'a list -> bool
hd      : 'a list -> 'a
tl      : 'a list -> 'a list
@       : ('a list # 'a list) -> 'a list
```

Note that if these operators were not polymorphic, we would need different primitive operators for all possible types of list elements! The 'a shared by the two arguments of :: (cons) prevents any attempt to build lists containing objects of different types.

One can always determine the type of any ML function or object just by typing its name at the top level; the object is evaluated and, as usual, its type is printed after its value.

```
expression:   [];

result:           [] : 'a list

expression:   hd;

result:           fun : ('a list) -> 'a
```

1.7. Higher-order functions

ML supports higher-order functions, i.e. functions which take other functions as arguments or deliver functions as results. A good example of use of higher-order functions is in *partial application*:

```
declaration:  val times a b == a * b;

binding:          val times : int -> (int -> int)

expression:   times 3 4;

result:           12 : int

declaration:  val twice == times 2;

binding:          val twice : int -> int

expression:   twice 4;

result:           8 : int
```

Note the type of times, and how it is applied to arguments: times 3 4 is read as (times 3) 4. Times first takes an argument 3 and returns a function from integers to integers; this function is then applied to 4 to give the result 12. We may choose to give a single argument to times (this is what is meant by partial application) to define the function twice, and supply the second argument later.

The function composition function, defined below, is a good example of partial application (it also has an interesting polymorphic type). Note how patterns are used in its definition.

```
declaration:  val comp (f,g) x == f (g x);
```

Every exception is associated with an *exception string*, which describes the reason of failure: in this case "hd" and in the previous example "div".

Exceptions can be *trapped* before they propagate all the way to the top level, and an alternative value can be returned:

```
expression:     hd [] ? 3;

result:             3 : int
```

The question mark operator traps all the exceptions which happen during the execution of the expression on its left, and returns the value of the expression on its right. If the left hand side does not raise exceptions, then its value is returned and the right hand side is ignored.

Exceptions can be trapped selectively by a double question mark, followed by a list of strings: only exceptions with strings appearing in that list are trapped, while the other exceptions are propagated outward.

```
expression:     hd [] ?? ["hd"; "tl"] 3;

result:             3 : int

expression:     1 div 0 ?? ["hd"; "tl"] 3;

exception:          Exception: div
```

The user can generate exceptions by the escape keyword followed by a string, which is the exception reason.

```
expression:     escape "fail";

exception:          Exception: fail

expression:     hd [] ? escape "emptylist";

exception:          Exception: emptylist
```

There is no real distinction between system and user generated exceptions: both can be trapped and are reported at the top level in the same way.

1.9. Types

Types describe the *structure* of objects, i.e. whether they are numbers, tuples, lists, strings etc. In the case of functions, they describe the structure of the function arguments and results, e.g. int to int functions, string list to bool functions, etc.

A type only gives information about attributes which can be computed at compile time, and does not help distinguishing among different classes of objects having the same structure. Hence the set of positive integers is not a type, nor is the set of lists of length 3.

On the other hand, ML types can express structural relations inside objects, for example that the right part of a pair must have the same type (whatever that type is) as the left part of the pair, or that a function must return an object of the same type as its argument (whatever that type may be).

Types like bool, int, and string, which do not show any internal structure, are called *basic*. Compound types are built by *type operators* like # (cartesian product), list (list) and −> (function space). Type operators are usually infix or suffix: int # int, int list and int −> int are the types of

```
declaration:    val (a : int) == 3;

declaration:    val f (a : int, b : int) : int == a + b;

declaration:    val f ((a,b) : int # int) == (a + b) : int;
```

The last two examples are equivalent. A ': *type*' just before the == of a function definition refers to the result type of the function.

1.10. Concrete types

A *concrete type* is a type for which *constructors* are available, together with other operations. A constructor is an operation which creates elements of some concrete type, by assembling simpler objects of other types.

Constructors are invertible functions: whatever a constructor does, can be undone by disassembling the parts which had been put together by the constructor. This allows constructors (or actually, their inverses) to be used in patterns to destructure data. We have already seen examples of this dual usage concerning the tuple constructor ' .. , .. ' and the list constructors nil and :: (cons).

A concrete type and its constructors should be considered as a single conceptual unit. Whenever a new concrete type is defined, its constructors are defined at the same time. Wherever a concrete type is known, its constructors are also known.

New concrete types and their constructors can be defined by *type declarations*. A type declaration defines at the same time a new type name, the names of the constructors for that type and the structure of the new type. In general a new type can consist of several alternatives, separated by '|'; for each of them we have a separate constructor, followed by the keyword 'of' and the type of the argument of that constructor. The keyword 'of' and the succeding type can be omitted: in this case the constructor is also called a *constant* of the new type.

For example, money can be a coin of some value (in cents), a bill of some value (in dollars), a check of some bank for some amount (in cents), or the absence of money.

```
declaration:    type money == nomoney | coin of int | bill of int | check of string # int;

bindings:           type money == nomoney | coin of int | bill of int | check of string # int
                    con nomoney : money
                    con coin : int -> money
                    con bill : int -> money
                    con check : (string # int) -> money
```

Here nomoney, coin, bill and check are money constructors (nomoney is a money constant). Constructors can be used like ordinary functions in expressions:

```
declaration:    val nickel == coin 5
                and dime == coin 10
                and quarter == coin 25;
```

But they can also be used in patterns:

```
declaration:    val   amount nomoney  == 0 |
                      amount (coin cents) == cents |
                      amount (bill dollars) == 100 * dollars |
                      amount (check(bank,cents)) == cents;

binding:                    val amount : money -> int
```

```
                    (parts*red + parts'*red') div totalparts,
                    (parts*blue + parts'*blue') div totalparts,
                    (parts*yellow + parts'*yellow') div totalparts)
                end
        end;
```

```
bindings:       abstype color
                val white == - : color
                val red == - : color
                val blue == - : color
                val yellow == - : color
                val mix : (int # color # int # color) -> color
```

Composite colors can be obtained by mixing the primary colors in different proportions:

```
declaration:    val green == mix(2,yellow,1,blue)
                and black == mix(1,red,2,mix(1,blue,1,yellow))
                and pink  == mix(1,red,2,white);
```

The bindings determined by the abstract type declaration are: an abstract type color whose internal structure is hidden; four colors (white, red, blue and yellow, printed as -) which are abstract objects whose structure is hidden; and a function to mix colors. No other operation can be applied to colors (unless defined in terms of the primitives), not even equality: if needed, an equality on colors can be defined as part of the abstract type definition.

An operation like mix, which takes abstract objects and produces abstract objects, typically uses patterns to get the representations of its arguments, manipulates the representations, and then uses constructors to produce the result.

As we said, the structure of abstract types and objects is hidden; for example val redpart,$,$ == pink does not give the intensity of red in that color, but produces a type error:

```
expression:     val redpart,$,$ == pink;

type error:         Type Clash in:    ((redpart,$,$) == pink)
                    Looking for :     'a # 'b # 'c
                    I have found :    color
```

The correct thing to do is val blend(redpart,$,$) == pink, but only where blend is available.

This protection mechanism allows one to change the representation of an abstract type without affecting the programs which are already using that type. For example we could have three pigments "red", "blue" and "yellow", and let a color be a list of pigments, where each pigment can appear at most 15 times; white would be blend [], etc. If we define all the operations appropriately, nobody will be able, from the outside, to distinguish between the two implementations of colors.

1.12. Interacting with the ML system

ML is an interactive language. It is entered by typing ml as a Unix command, and it is exited by typing *Control-D*.

Once the system is loaded, the user types phrases (i.e. expressions, top level declarations or commands), and some result or acknowledgement is printed. This cycle is repeated over and over, until the system is exited.

Every top-level phrase is terminated by a semicolon. A phrase can be distributed on several lines by breaking it at any position where a blank character is accepted. Several phrases can also be written on the same line.

```
- it;
  3 : int

- it + z;
Unbound Identifier: z

- it;
  3 : int

- 1 div 0;
Exception: div

- it,it;
  3,3 : int # int
```

The it variable is very useful in interaction, when used to access values which have 'fallen on the floor' because they have been computed as expressions but have not been bound to any variable. it is not a keyword or a command, but just a plain variable (you can even temporarily redefined it by typing val it == *exp*;). Every time that an expression *exp*; is entered at the top level, this is considered an abbreviation for val it == *exp*;. Hence it is statically scoped, so that old its and new its are totally independent and simply hide each other.

1.13. Errors

Syntax errors report the fragment of code which could not be parsed, and often suggest the reason for the error. The erroneous program fragment is at most one line long; ellipses appear at its left when the source code was longer than a line. It is important to remember that the error was found at the *extreme right* of the error message.

```
- if true then 3;
Syntax Error: if true then 3;
I was expecting an "else"
```

Identifiers, type identifiers and type variables can be undefined; here are the messages given in those situations (type variables must be present on the left of a type definition whenever they are used on the right):

```
- noway;
Unbound Identifier: noway

- 3 : noway;
Unbound Type Identifier: noway

- type t == 'noway
Unbound Type Variable: 'noway
```

Type errors show the context of occurrence of the error and the two types which could not be matched. The context might not correspond exactly to the source code because it is reconstructed from internal data structures.

```
- 3 + true;
Type Clash in:    (3 + true)
Looking for :     int
I have found :    bool
```

```
declaration:    type unit == ();

bindings:       type unit == ()
                con () : unit
```

except that the above is not legal ML syntax: user defined constants must be identifiers, and () is treated specially.

3.2. Booleans

The predefined constants

```
true, false     : bool                      (constants)
```

denote the respective boolean values. Boolean operators are:

```
not             : bool -> bool
&, or           : (bool # bool) -> bool     (infixes)
```

Both arguments of & and or are always evaluated. In some languages & and or evaluate only one argument when this is sufficient to determine the result; the same effect can be achieved in ML by using nested conditional expressions.

The boolean type is not primitive; it can be defined in ML as:

```
declaration:    type bool == false | true;

bindings:       type bool == false | true
                con false : bool
                con true : bool
```

All the boolean operations (including if-then-else) can then be defined or simulated by case analysis.

3.3. Integers

The integer operators are:

```
.. ~2, ~1, 0, 1, 2, ..   : int                      (constants)
~                        : int -> int
+, -, *, div, mod        : (int # int) -> int       (infixes)
>, >=, <=, <             : (int # int) -> bool      (infixes)
```

Negative integers are written ~3, where '~' is the complement function (while '-' is difference). Integers have unbounded precision; arithmetic exceptions can only be generated by integer division div and module mod, which escapes with string "div" when their second argument is zero.

The integer type could in principle be defined in ML as:

```
declaration:    type rec nat == one | succ of nat;

declaration:    type int ==  neg of nat | zero | pos of nat;
```

```
null            : 'a list -> bool
hd              : 'a list -> 'a
tl              : 'a list -> 'a list
length          : ('a list) -> int
@               : ('a list # 'a list) -> 'a list          (infix)
rev             : ('a list) -> ('a list)
map             : ('a -> 'b) -> (('a list) -> ('b list))
fold, revfold   : (('a # 'b) -> 'b) -> (('a list) -> ('b -> 'b))
```

- null tests whether a list is empty.
- hd (head) extracts the first element of a list; it escapes with string "hd" on empty lists.
- tl (tail) returns a list where the first element has been removed; it escapes with string "tl" on empty lists.
- length returns the length of a list.
- @ (append) appends two lists (e.g. [1;2]@[3;4] is the same as [1;2;3;4]).
- rev returns the reverse of a list.
- map takes a function and then a list, and applies the function to all the elements of the list, returning the list of the results (i.e. map f [e_1; .. ; e_n] is [f (e_1); .. ; f (e_n)]).
- fold maps a binary function to an n-ary function over a list (e.g. it can map + to the function computing the sum of the elements of a list), it takes first a binary function, then the list to accumulate, then the value to return on empty lists (i.e. fold f [e_1; .. ; e_n] e is f (e_1, .. f (e_n,e)..)).
- revfold is just like fold but accumulates in the opposite direction (i.e. revfold f [e_1; .. ; e_n] e is f (e_n, .. f (e_1,e)..)).

The type of lists can be defined in ML as:

```
declaration:    infix :: ;

declaration:    type rec 'a list == nil | :: of int # (int list);

bindings:       type 'a list == nil | :: of int # (int list);
                con nil : 'a list
                con :: : ('a # ('a list)) -> ('a list)
```

All the list operations can then be defined by case analysis. The bracket notation (e.g. [1;2]) is just an abbreviation for the compositions of list constructors (e.g. 1::2::nil).

typing rule: if e_1 : t and .. and e_n : t then [e_1; .. ; e_n] : t list

3.6. Strings

A string constant is a sequence of characters enclosed in quotes, e.g. "this is a string".

Operation on strings are:

```
" ... "         : string                          (constants)
size            : string -> int
extract         : (string # int # int) -> string
explode         : string -> string list
implode         : string list -> string
explodeascii    : string -> int list
implodeascii    : int list -> string
intofstring     : string -> int
stringofint     : int -> string
```

```
binding:        type 'a refpair == refpair of ('a ref) # ('a ref)
                con refpair : (('a ref) # ('a ref)) -> ('a refpair)

declaration:    val r == ref 3;
                val p == refpair(r,r);

bindings:       val r == ref 3 : int ref
                val p == refpair(ref 3,ref 3) : int refpair

expression:     r := 5;

result:         () : unit

expression:     p;

result:         refpair(ref 5,ref 5) : int refpair
```

3.8. Arrays

Arrays are ordered collections of values with a constant access time retrieval operation. They have a lower bound and a size, which are integers, and can be indexed by integer numbers in the range lowerbound ≤ i ≤ lowerbound+size. Arrays can have any positive or null size, but once they are built they retain their initial size.

Arrays over values of type *t* have type *t* array. Arrays can be built over elements of any type; arrays of arrays account for multi-dimensional arrays.

The array primitives are:

```
array          : (int # int # 'a) -> 'a array
arrayoflist    : (int # 'a list) -> 'a array
lowerbound     : 'a array -> int
arraysize      : 'a array -> int
sub            : ('a array # int) -> 'a                  (infix)
update         : ('a array # (int # 'a)) -> unit         (infix)
arraytolist    : 'a array -> 'a list
```

- array makes a constant array (all items equal to the third argument) of size n≥0 (second argument) from a lowerbound (first argument). It escapes with string "array" if the size is negative.
- arrayoflist makes an array out of a list, given a lower bound for indexing.
- lowerbound returns the lower bound of an array.
- arraysize returns the size of an array.
- sub extracts the i-th item of an array. It escapes with string "sub" if the index is not in range.
- update updates the i-th element of an array with a new value. It escapes with string "update" if the index is not in range.
- arraytolist converts an array into the list of its elements.

Arrays are not a primitive concept in ML: they can be defined as an abstract data type over lists of assignable references. This specification of arrays, which is given below, determines the semantics of arrays, but does not have a fast indexing operation. Arrays are actually implemented at a lower level as contiguous blocks of memory with constant time indexing. Here is an ML specification of arrays, semantically equivalent to their actual implementation (see sections "Lists", "Updatable references" and "Lexical declarations" to properly understand this program).

```
infix sub;
infix update;
```

```
val op update == ();
val arraytolist == ();
```

This transforms ML into a purely applicative language, by making all the side-effecting operations and types inaccessible.

3.9. Patterns and matches

The left hand side of an == in a value definition can be a simple variable, or a more complex *pattern* involving several distinct variables. Patterns are also used in formal parameters of functions and in case expressions. In all these situations a value has to be decomposed according to its structure; the matching process produces a set of *bindings* by associating the variables in the pattern with the corresponding parts of the value.

A pattern can be a $ sign (matching any value), an identifier (preceded by op if infix), a constant, a constructor, an infix constructor, a list pattern, a tuple pattern, a type specification pattern, or a parenthesized pattern.

syntax:

simp_pat ::=
- $
- {op} *Ide*
- *con*
- [*pat*; .. ;*pat*]
- (*pat*)

pat ::=
- *simp_pat*
- *con simp_pat*
- *simp_pat con simp_pat*
- *pat*, .. ,*pat*
- *pat* : *type*

The pattern matching rules are given recursively on the structure of the pattern. The letter *k* denotes constructors, *a* and *b* denote variables, *u* and *v* denote values, and *p* and *q* denote patterns.

Pattern	Value	Match(Pattern,Value)
$	v	∅
a	*v*	{*a* = *v*}
()	()	∅
k	*k*	∅
k p	*k v*	Match(*p*,*v*)
$[p_1;..;p_n]$	$[v_1;..;v_n]$	$\text{Match}(p_1,v_1) \cup .. \cup \text{Match}(p_n,v_n)$ $n \geq 0$
$p_1,..,p_n$	$v_1,..,v_n$	$\text{Match}(p_1,v_1) \cup .. \cup \text{Match}(p_n,v_n)$ $n \geq 0$

Example:

declaration:
```
val (a,b) :: [c;$] == [1,2; 3,4; 5,6];
```

bindings:
```
val a == 1 : int
val b == 2 : int
val c == 3,4 : int # int
```

The variables in a single pattern must all be distinct. Patterns are not expressions; they can only

not part of larger patterns). To completely specify the type of a fun-expression we may use either of the two forms:

```
expression:   fun (a: bool). fun (b: int, c: int). (if a then b else c) : int;

expression:   (fun a. fun (b,c). if a then b else c) : bool -> ((int # int) -> int);

result:           fun : bool -> ((int # int) -> int)
```

In declarations, we have the following options (and more):

```
declaration:  val f (a: bool) (b: int, c: int) : int == if a then b else c;

declaration:  val f (a: bool) ((b,c) : int # int) == (if a then b else c) : int;

binding:          val f : bool -> ((int # int) -> int)
```

The first form should be preferred; note how the result type of the function precedes the == sign of the definition.

This explicit type information is often redundant. However, the system checks the explicit types against the types it infers from the expressions. Introducing explicit type information is a good form of program documentation, and helps the system in discovering type errors exactly where they arise. Sometimes, when explicit type information is omitted, the system is able to infer a type (not the intended one) from an incorrect program; this 'misunderstanding' is only discovered in some later use of the program and can be difficult to trace back.

The type of a function expression is determined by the type of its binder (or binders) and the type of its body (or bodies):

typing rule: if $p_1 : t$ and .. and $p_n : t$ and $e_1 : t'$ and .. and $e_n : t'$
then (fun p_1. e_1 | .. | p_n. e_n) : t -> t'

The type automatically inferred by the system is the 'most polymorphic' of the typings which obey the typing rules. For example, from the previous typing rule we can infer that (fun x. x) : int -> int, and also that (fun x. x) : 'a -> 'a; the latter type is more polymorphic (in the sense that the int -> int can be obtained by instantiating the type variables of 'a -> 'a); in fact it is the most general typing, and is taken as the default type for (fun x. x).

3.11. Application

There are two lexical categories of identifiers, which determine how functions are syntactically applied to arguments: *nonfix* and *infix*. Any identifier can be declared to fall in one of these categories: see section "Lexical declarations" about how to do this.

Nonfix identifiers are the ordinary ones; they are applied by juxtaposing them with their argument (in this section we use *f* and *g* for functions and *a*, *b*, *c* for arguments):

```
syntax:   f a
```

The expression *a* can also be parenthesized (as any expression can), obtaining the standard application syntax *f* (*a*). It is common *not* to use parentheses when the argument is a simple variable, a string (*f* "abc") or a list (*f* [1;2;3]). It is necessary to use parentheses when the argument is an infix expression like a tuple or an arithmetic operation (*f* (*a*,*b*,*c*), *f* (*a*+*b*)) because the precedence of application is normally greater than the precedence of the infix operators (*f a*,*b* and *f a*+*b* are interpreted as (*f a*),*b* and (*f a*)+*b*). Function application binds stronger than any operator: see the

type of a sequencing expression is the type of e_n. Example:

expression: (a := 4; a := !a div 2; !a)

result: 2 : int

Note that (e) is equivalent to e, and that () is the unity.

typing rule: if $e : t$ then $(e_1; .. ; e_n; e) : t$ $n \geq 1$

3.14. While

The while construct can be used for iterative computations. It is included in ML more as a matter or style and convenience than necessity, as all *tail recursive* functions (e.g. functions which end with a recursive call to themselves or to other functions) are automatically optimized to iterations by the compiler.

The while construct has two parts: a test, preceded by the keyword while and a body, preceded by the keyword do. If the test evaluates to true then the body is executed, otherwise () is returned. This process is repeated until the test yields false (if ever) or an exception occurs.

syntax: while e_1 do e_2

The result of a terminating while construct is always (), hence whiles are only useful for their side effects. The body of the while is also expected to yield () at every iteration.

typing rule: if e_1 : bool and e_2 : unit
then (while e_1 do e_2) : unit

As an illustration, here is an iterative definition of factorial which uses two local assignable variables count and result:

```
declaration:  val fact n ==
                  let val count == ref n
                     and result == ref 1
                  in     (while !count <> 0 do
                                 (result := !count * !result;
                                  count := !count-1);
                          !result)
                  end;
```

3.15. Case

A case expression is just a different syntax for pattern matching: see section "Patterns and matches" for a description of *matches*.

syntax: case *exp* of *match*

The expression *exp* is matched against the *match*, which is a set of pattern-action pairs; the first pattern to match is activated, and the corresponding action is evaluated and returned as the result of the case expression.

```
declaration:  type color == red | purple | yellow
```

level, a *failure message* is printed. In this version of the ML system, exception packets are limited to string values, which are then called *exception strings*, and failure messages print the contents of the exception strings.

exception: Exception: *reason*

where *reason* is the content of the exception string mentioned above. Note that the exception string is not the value of a failing expression: failing expressions have no value, and exception strings are manipulated by mechanisms which are independent of the usual value manipulation constructs. The exception string is often the name of the system primitive or user function which raised the exception.

User exceptions can be raised by the escape construct:

syntax: escape *exp*

The expression *exp* above must evaluate to a string, which is the exception string.

The propagation of exceptions can only be stopped by the *trap* construct.

The result of the evaluation of e_1 ? e_2 is normally the result of e_1, unless e_1 raises an exception, in which case it is the result of e_2.

The result of the evaluation of e_1 ?? e_2 e_3 (where e_2 evaluates to a string list *sl*) is normally e_1, unless e_1 raises an exception, in which case it is the result of e_3 whenever the exception string is one of the strings in *sl*; otherwise the exception is propagated.

The result of the evaluation of e_1 ?\ v e_2 is normally the result of e_1, unless e_1 raises an exception, in which case it is the result of e_2, where the variable v is associated with the exception string and can be used in e_2.

syntax: e_1 ? e_2

syntax: e_1 ?? e_2 e_3

syntax: e_1 ?\ v e_2

Some system exceptions may be raised at any time, independently of the expression which is being evaluated. They are "interrupt", which is generated by pressing the *DEL* key during execution, and "collect", which is generated when there is no space left. Even these exceptions can be trapped.

The typing rule for escape says that an exception is compatible with every type, i.e. that it can be generated without regard to the expected value of the expression which contains it. The rules for the trap operators state that the exception handler must have the same type as the possibly failing expression, so that the resulting type is the same whether the exception is raised or not.

typing rules: (escape *e*) : 'a

if $e : t$ and $e' : t$ then $(e \ ?\ e') : t$

if $e : t$ and *sl* : string list and $e' : t$ then $(e \ ??\ sl\ e') : t$

if $e : t$ and v : *string* and $e' : t$ then $(e \ ?\backslash\ v\ e') : t$

```
expression:     (fun x. x) : int -> int;

result:             fun : int -> int

expression:     (fun x. x + 1) : 'a -> 'a;

result:             fun : int -> int
```

are accepted, even if, according to the previous discussion, 3 does *not* have type 'a; (fun x. x) has a better type than int -> int; and (fun x. x + 1) may or may *not* have type 'a -> 'a, according to how it behaves outside the integer domain.

3.19. Equality

The standard equality predicate is *structural equality*: two objects are equal when they are the same atomic object, or all their components are equal.

The mathematical definition of equality over domains is in general undecidable, hence some restrictions are imposed on the objects which can be compared, in order to obtain a decidable predicate. Equality is considered to be a predefined overloaded operator, i.e. an infinite collection of decidable equality operators, whose types are instances of ('a # 'a) -> bool.

Equality is disallowed on polymorphic objects, on functions, on streams and on elements of abstract data types. When needed, these objects can be marked by data having a decidable equality, but then the user must write a special purpose equality routine.

Equality on reference object is not structural equality, but 'sameness', or pointer equality (e.g. ref 3 = ref 3 is false).

Equality over abstract types might be implemented as equality over the corresponding concrete representations. However in this case abstract types would not be transparent to a change of representation (e.g. from an int list to an int->int), as equality would behave differently on the different representations. Also, if we implement abstract sets by concrete multisets, equality could distinguish between sets which should be equal, giving dangerous insights on the chosen representation, as well as not corresponding to set equality.

Here are examples of 'right' and 'wrong' comparisons (at the top level). The 'wrong' comparisons produce compile time type errors:

Right	Wrong
`() = ();`	
`true = false;`	
`3 = 3;`	
`"a" = "";`	
`(2,"a") = (2,"a");`	
`[] = ([]: int list);`	`[] = [];`
`[1;2] = [3];`	
`(fun a,b. a=b) : string#string -> bool;`	`fun a,b. a=b;`
	`fun a. a = fun a. a;`
	`set[3] = set[3];`

where set is an abstract type.

{op} *Ide simp_pat .. simp_pat* {: *type*} == *exp*
{op} *Ide simp_pat* {: *type*} == *exp* | .. | *Ide pat* {: *type*} == *exp*

Each declaration imports the variables used in the expressions **exp** and exports the variables defined on the left of ==. Here are some examples:

declaration: val x == 2 * 3;

binding: val x == 6 : int

declaration: val a,b == 4,[];

bindings: val a == 4 : int
val b == [] : 'a list

declaration: val K x y == x;

binding: val K : 'a -> ('b -> 'a)

declaration: val null nil == true | null (a :: b) == false;

binding: val null : ('a list) -> bool

Note that function definitions require simple (i.e. possibly parenthesized) patterns as formal parameters. In particular, parameters which have the form of constructors applied to patterns must be parenthesized (e.g. val f (ref a) == a; is legal, but val f ref a == a; is not).

5.1.2. Parallel bindings

To bind the variables $x_1..x_n$ simultaneously to the values of $e_1..e_n$ one can write either of the following declarations:

val x_1, .. ,x_n == e_1, .. ,e_n;

val x_1 == e_1 and .. and x_n == e_n;

In the first case we use a composite pattern, while in the second case we use the infix operator **and**:

syntax: *parallel_value_binding* ::=
value_binding and *value_binding*

The meaning of **and** is that the bindings of the two sides are established 'in parallel', in the sense that the variables exported from the left side are not imported to the right side, and vice versa. The whole construct exports the union of the variables of the two declarations; it is illegal to declare the same variable on both sides of an **and**.

For example, note how it is possible to swap two values without using a temporary variable:

declaration: val a == 10 and b == 5;

bindings: val a == 10 : int
val b == 5 : int

declaration: val a == b and b == a;

bindings:
```
type 'a pair == pair of 'a # 'a
con pair : ('a # 'a) -> ('a pair)
```

declaration:
```
type ('a, 'b) pairpair == pairpair of ('a # 'b) # ('a # 'b);
```

bindings:
```
type ('a, 'b) pairpair == pairpair of ('a # 'b) # ('a # 'b)
con pairpair : (('a # 'b) # ('a # 'b)) -> (('a, 'b) pairpair)
```

All the type variables appearing on the right of the defining == sign must appear on the left as parameters of the type being defined. The list of parameters on the left of the == sign must be a sequence of distinct type variables.

5.2.2. Parallel bindings

As in value bindings, the and operator can be used to combine type binding in parallel, in the sense that the variables exported from the left side are not imported to the right side, and vice versa.

syntax:
parallel_type_binding ::=
type_binding and *type_binding*

The whole construct exports the union of the variables defined by the two type bindings; it is illegal to declare the same variable on both sides of an and.

5.2.3. Recursive bindings

The operator rec is used to define recursive and mutually recursive types. The binding rec *tb* exports the variables exported by the type binding *tb*, and imports the variables imported by *tb* and the variables exported by *tb*.

syntax:
recursive_type_binding ::=
rec *type_binding*

declaration:
```
type rec 'a tree == leaf of 'a | node of ('a tree) # ('a tree);
```

bindings
```
type 'a tree == leaf of 'a | node of ('a tree) # ('a tree)
con leaf : 'a -> ('a tree)
con node : (('a tree) # ('a tree)) -> ('a tree)
```

If rec were omitted, the type identifier tree on the right of == would not refer to its defining occurrence on the left of ==, but to some previosly defined tree type (if any) in the surrounding scope.

5.3. Abstract type declarations

Abstract types can be defined by the *abstype-with* declaration, which is an abbreviation of a special form of private declaration (see section "Private declarations", below).

syntax:
abstract_declaration ::=
abstype *type_binding* with *declaration* end

An abstract type defined by the abstype keyword is like a concrete type defined by the type keyword, but its constructors are only available in the declaration following the with keyword.

declaration:
```
abstype position == position of int # int
```

which are not redefined in d_2.

```
expression:     let val a == 10;   val b == a + 5
                in b end;

result:             15 : int
```

5.5. Private declarations

The export declaration allows one to hide some of the bindings of a declaration, while making other bindings available to the outside.

```
syntax:     private_declaration ::=
                export export_list .. export_list from declaration end

syntax:     export_list ::=
                abstype type_ide .. type_ide
                type type_ide .. type_ide
                val ide .. ide
```

The type identifiers and the value identifiers (which should be declared in declaration), are exported to the outside, while the other type identifiers and value identifiers defined in *declaration* are hidden. In the case of a type export, the constructors of that type are automatically exported. In the case of an abstype export the constructors are not exported. Constructors cannot be listed in a val section.

```
declaration:    export val a c
                from  (val a,b == 1,2;
                       val c == a + b)
                end;

bindings:           val a == 1 : int
                    val c == 3 : int

declaration:    export val increment fetch
                from  (val count == ref 0;
                       val increment () == count := !count + 1
                       and fetch () == !count)
                end;

bindings:           val increment : unit -> unit
                    val fetch : unit -> int

declaration:    export abstype increasing_pair
                    val increasing_pair high low
                from  type increasing_pair == pair of int # int;
                    val increasing_pair(x,y) ==
                        if x>y then escape"increasing_pair"
                        else pair(x,y)
                    and low (pair(x,y)) == x
                    and high (pair(x,y)) == y
                end;
```

```
syntax:          lexical_declaration ::=
                       infix lde .. lde
                       nonfix lde .. lde

declarations:
                       infix  <--> ;
                       nonfix + ;
```

This example declares an infix operator <--> and puts the operator + back to a status of normal non-infix identifier. These operators can now be uses in expressions and declarations according to their new lexical status.

A lexical declaration can be used in scopes-blocks. In this case the newly defined fixity status is only active in the local scope.

6. I/O streams

Input-output is done on streams. A stream is like a queue; characters can be read from one end and written on the other end. Reads are destructive, and they wait indefinitely on an empty stream for some character to be written. In what follows, a "file" is a file on disk which has a "file name"; a "stream" is an ML object (it is a pair of Unix file descriptors, one open for input and the other one open for output).

```
file        : string -> stream
save        : (string # stream) -> unit
stream      : unit -> stream
channel     : string -> stream
input       : (stream # int) -> string
output      : (stream # string) -> unit
lookahead   : (stream # int # int) -> string
caninput    : stream -> bool
terminal    : stream
```

- Streams are associated with file names in the operating system. The operation file takes a string (a file name) and returns a new stream whose initial content is the content of the corresponding file. It escapes with string "file" if the stream is not available (e.g. the file name syntax is wrong, or the file is locked). If no file exists with that file name, a new empty stream is returned (hence, empty files and streams are indistinguishable from non-existent ones). The same file name can be requested several times; every time a new independent stream is generated.

- A copy of an existing stream can be associated with a file name (i.e. written to a file) by the save operation which takes a string (the file name) and a stream and returns unit. The stream is unaffected by this operation. An exception with string "save" is raised if the association cannot be carried out. Reads and writes on streams do not affect the files they come from. Conversely, a save operation on a file does not affect the streams which have been extracted from that file; it only affects the result of a subsequent file.

- The operation stream returns a new empty stream. It accounts for temporary (unnamed) files. Moreover, a stream-filename association can be removed by reassociating an empty stream with that file name.

- Input operations are destructive; the characters read are removed from the stream. input takes a stream *s* and a number *n* and returns a string of *n* characters reading them from the stream *s*. If there are less than *n* characters in the stream, it waits indefinitely until more characters are written on the stream. The wait can be interrupted by the *DEL* key producing an "interrupt" exception. caninput returns true if a stream is not empty; the input operations do not fail on empty streams, they wait indefinitely for something to be written on the stream.

```
val right : 'd pair -> 'd
```

a declaration import M can also appear in local scopes and inside functions, wherever a declaration is accepted (the binding module /usr/lib/ml/lib above is explained later).

Importing can result in *loading* a module, when that module is imported for the first time in an interactive session, or in *sharing* an already loaded module, all the subsequent times. The loading/sharing mechanism is explained below in "Module sharing".

No evaluation takes place when a module is compiled. Instead, the module body is evaluated every time the module is *loaded*. If the module body contains side effecting operations (such as input/output), they have effect at loading time, i.e. they do not have effect if the module is being shared.

More precisely, we can say that a module is an *environment generator*; loading a module corresponds to generating a new environment, and sharing a module corresponds to using an already generated environment.

7.1. Module hierarchies

A module A can import other modules B, C, etc. The import relations between modules determine a module hierarchy. The hierarchy is dynamic, because of the possibility of conditional imports and of hidden imports caused by functions imported from other modules.

A module can import other modules for several reasons.

```
module:     module A1
            body
                val f a ==
                    let import B1
                    in .. end
            end;

module:     module A2
            body
                export
                    val f g
                from
                    import B2;
                    val f a == ..
                    and g b == ..
                end
            end;

module:     module A3
            body
                import B3;
                val f a == ..
                and g b == ..
            end;
```

The first example above shows a module B1 locally imported for the private use of a function in A1. The second example shows a module B2 imported for private use in a module A2; the export construct guarantees that B2 is not exported from A2. The third example shows a module B3 which is imported by A3, and also reexported.

Sometimes an imported module B *must* be reexported from an importing module A. This happens when a type identifier t defined in B is contained in the (types of the) bindings exported by A.

```
bindings:       module /usr/lib/ml/lib
                module A
                val b == 5 : int
```

In what follows, when we want to make those hidden bindings explicit, we write them indented under the respective module bindings:

```
declaration:    import B;

bindings:       module /usr/lib/ml/lib
                module A
                    val a == 3 : int
                val b == 5 : int
```

7.2. Module sharing

In an interactive session, a compiled module can be imported several times, but it is only loaded *once*, the first time it is imported. All the following imports of that module simply fetch the already loaded module. For example, two consecutive top level import A; declarations are semantically equivalent to only one of them, and no extra work is actually done in the second import.

At any moment there is at most one copy of a module in the system, and that copy is shared among all the modules which import it. Sharing means that types are shared, and that values are shared; for example:

```
module:         module A
                body
                    abstype T == T of int
                    with  val absT n = T n
                          and repT (T n) == n
                    end;
                    val a == ref 3
                end;

module:         module B
                includes A
                body
                    val BabsT,BrepT == absT,repT
                    and b == a
                end;

module:         module C
                includes A
                body
                    val CabsT,CrepT == absT,repT
                    and c == a
                end;

module:         module D
                includes B C
                body
                end;

declaration:    import D;
```

interface (i.e. the types of the exported bindings) changes.

8. The ML system

8.1. Entering the system

The ML system is entered under Unix by typing ml as a shell command. A library of standard functions is then loaded. This library is installation dependent and is meant to contain the functions which are most used locally. After the system prompt, you can start typing expressions or definitions, or loading source files.

8.2. Loading source files

An ML source file looks exactly like a set of top-level phrases, terminated by semicolons. A file prog.ml containing ML source code can be loaded by the use top-level command:

```
use "prog.ml";
```

where the string surrounded by quotes can be any Unix file name or path. The file extension .ml is normally used for ML source files, but this is not compulsory.

Source files may again contain use commands to load other source files in a cascade. There is a Unix-dependent limit on the depth of recursive loading: the message Cannot open file: *filename* is printed when such limit is exceeded. The file *filename* will not be loaded, but the loading of the previously opened files will continue.

Typing *DEL* while loading a file will interrupt loading and bring you back to the top level. This might not happen immediately, because some parts of the compilation process are not interruptible, but it will have effect as soon as the current phrase has finished compiling.

8.3. Exiting the system

Type *Control-D* to exit the system. No program or data created during an ML session is preserved, except for compiled modules.

8.4. Error messages

The most common syntax and type errors are described in section "Errors". Here are some unfrequent messages which are generated in particular situations.

An 'unimplemented' error is given when a value or type identifier is specified in an export export list but is not defined in the corresponding from part. This is different from the normal 'undefined' error for undefined identifiers.

```
- export val a from val b == 3 end;
Unimplemented Identifier: a

- export type t from type u == int end;
Unimplemented Type Identifier: t
```

A type error occurs when two isomorphism types having the same name are defined, and are allowed to interact:

```
- let abstype A == A of int
      with val absA n == A n end;
      abstype A == A of int
      with val repA (A n) == n end
  in repA(absA 3) end;
Type Clash in:    (repA (absA 3))
```

```
StackCode?
OptimizerOff?
AssemblerCode?
ObjectCode?
Environment?
StopBeforeExec?
CheckHeap?
MemoryAllocation?
Watch? hex:
Timing?

_
```

The menu stops at every line, waiting for a y followed by a *CarriageReturn* for yes, or a *CarriageReturn* for no. The menu can be exited at any point by *DEL*: the options chosen up to that point will have effect; the other ones are unchanged. To reset the options to the initial default state, reenter the menu and answer no to all questions. All options have effect until changed again.

- *ParseTree* prints the parse tree of every subsequent top level phrase. This can be useful to see how the precedence of operators works.
- *Types* prints the type of every subsequent top level phrase, before executing it. It is useful only in debugging the system, as the type is the same as the type of the result which is printed after evaluation.
- *TypeVariables* prints all the hidden attributes of type variables. The following is for typechecking wizards only. 'a[i] means that 'a has an occurrence level i (i.e. it is used in a fun-nesting of depth i). Generic type variables are written 'a[l] (l for infinity). Weak type variables are written _'a[i], and non-generic type variables are written !'a[i].
- *Match* prints the intermediate structures generated during the compilation of patterns. Not for human consumption.
- *StackCode* for every subsequent top-level phrase, prints the intermediate Functional Abstract Machine code produced by the compiler [Cardelli 83], after peephole optimization.
- *OptimizerOff* switches off the peephole optimizer for the abstract machine code. The optimizer works mainly on multiple jumps, function return, partial application, and tail recursion.
- *AssemblerCode* for every subsequent top-level phrase, prints the VAX assembler instructions produced from the abstract machine code.
- *ObjectCode* for every subsequent top-level phrase, prints the final hexadecimal result of the compilation.
- *Environment* for every subsequent top-level phrase, prints all the the types currently defined with their definition, and all the variables currently defined with their value and type.
- *StopBeforeExec* for every subsequent top-level phrase, stops immediately before executing the result of compilation, asking for a confirmation to proceed.
- *CheckHeap* for every subsequent top-level phrase, makes a complete consistency check of the data in the heap, and reports problems.
- *MemoryAllocation* reports every Unix memory allocation or deallocation, except the ones caused by Pascal.
- *Watch* is used for internal system debugging.
- *Timing* at the end of every evaluation, gives the parsing time (Pars), typechecking time (Anal), translation to abstract machine code and optimization time (Comp), translation to VAX code time (Assm), total compilation time (Total) and run time (Run). All in milliseconds.

8.6. Garbage collection and storage allocation

Garbage collection is done by a two-spaces copying compacting garbage collector. The size of the spaces grows as needed; garbage collection may escape with string "collect" if Unix refuses to grant more space when needed. For technical reasons, the allocation of very large data structures may escape with string "alloc", even if there is still memory available.

Appendix A. Lexical classes

Ascii characters are classified into the following categories:

- Illegal — Unacceptable in a source program. These characters are ignored, and a warning message is printed.
- Eof — End-of-file character. It terminates an interactive session, or stops the process of reading a source ml file. A top level control-D is interpreted as Eof, thereby exiting the system.
- Blank — Characters which are interpreted as a space character " ".
- Digit — Digits.
- Letter — Letters and other characters which can be used to form identifiers.
- Symbol — A class of character which can be used to form operators (see appendix "Syntax of lexical entities").
- Escape — Escape character in strings. It behaves as a Symbol outside strings.
- Quote — String quotation character.
- Delim — One-character punctuation marks and parentheses.

Moreover, any sequence of legal character enclosed between { and } or end of file is a *comment* (except when { and } appear in a string). Comments can be nested. Each comment is considered equivalent to a single space character. An isolated } is treated as a Delim and may produce various kinds of syntax errors.

```
nul  Eof;       soh  Illegal;   stx  Illegal;
etx  Illegal;   eot  Illegal;   enq  Illegal;
ack  Illegal;   bel  Illegal;   bs   Illegal;
ht   Blank;     lf   Blank;     vt   Blank;
ff   Blank;     cr   Blank;     so   Illegal;
si   Illegal;   dle  Illegal;   dc1  Illegal;
dc2  Illegal;   dc3  Illegal;   dc4  Illegal;
nak  Illegal;   syn  Illegal;   etb  Illegal;
can  Illegal;   em   Illegal;   sub  Illegal;
esc  Illegal;   fs   Illegal;   gs   Illegal;
rs   Illegal;   us   Illegal;   ' '  Blank;
'!'  Symbol;    '"'  Quote;     '#'  Symbol;
'$'  Delimiter; '%'  Symbol;    '&'  Symbol;
'''  Letter;    '('  Delim;     ')'  Delim;
'*'  Symbol;    '+'  Symbol;    ','  Delim;
'-'  Symbol;    '.'  Delim;     '/'  Symbol;
'0'  Digit;     '1'  Digit;     '2'  Digit;
'3'  Digit;     '4'  Digit;     '5'  Digit;
'6'  Digit;     '7'  Digit;     '8'  Digit;
'9'  Digit;     ':'  Symbol;    ';'  Delim;
'<'  Symbol;    '='  Symbol;    '>'  Symbol;
'?'  Symbol;    '@'  Symbol;    'A'  Letter;
'B'  Letter;    'C'  Letter;    'D'  Letter;
'E'  Letter;    'F'  Letter;    'G'  Letter;
'H'  Letter;    'I'  Letter;    'J'  Letter;
'K'  Letter;    'L'  Letter;    'M'  Letter;
'N'  Letter;    'O'  Letter;    'P'  Letter;
'Q'  Letter;    'R'  Letter;    'S'  Letter;
'T'  Letter;    'U'  Letter;    'V'  Letter;
'W'  Letter;    'X'  Letter;    'Y'  Letter;
'Z'  Letter;    '['  Delim;     '\'  Escape;
```

Appendix B. Keywords

abstype
and
body
case
do
else
end
export
escape
from
fun
if
import
in
includes
infix
let
local
module
nonfix
of
op
rec
then
type
use
val
where
while
with
:
?
??
?\
==

input	Read from stream	Nonfix
output	Write to stream	Nonfix
lookahead	Peek from stream	Nonfix
caninput	Stream status	Nonfix
terminal	Terminal stream	Nonfix

Appendix E. Precedence of operators

Infix operators in order of decreasing precedence (see section "Lexical declarations").

{application}	*left associative*
sub	*left associative*
* div mod	*left associative*
+ – ^	*left associative*
:: @	*right associative*
= <> > < >= <=	*right associative*
&	*right associative*
or o	*right associative*
{user-infix}	*left associative*
update :=	*right associative*
.. , .. , ..	*n-ary infix*

Appendix G. Syntax of lexical entities

```
Letter ::=
  "a" | .. | "z" | "A" | .. | "Z" |  "_".

Digit ::=
  "0" | .. | "9".

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

Character ::= .. (see appendix "Lexical classes" for a list of legal characters)

Ide ::=
  Letter {Letter | Digit | "'"} |
  {Symbol}1.

Integer ::=
  {Digit}1.

String ::=
  """ {Character} """.

Con ::=
  "(" ")" |
  Integer |
  String |
  Ide.

TypeIde ::=
  Ide.

TypeVar ::=
  "'" Ide.
```

```
    Pat "==" Exp |
    ["op"] Ide {Pat}1 [":" Type] "==" Exp |
    {["op"] Ide Pat [":" Type] "==" Exp / "|"}2 |
    ValBind "and" ValBind |
    "rec" ValBind.

TypeBind ::=
    [TypeParams] TypeIde "==" {Ide ["of" Type] / "|"}1 |
    TypeBind "and" TypeBind |
    "rec" TypeBind.

TypeParams ::=
    TypeVar | "(" {TypeVar / ","}1 ")".

SimplePat ::=
    "$" |
    ["op"] Ide |
    Con |
    "[" {Pat / ";"} "]" |
    "(" Pat ")".

Pat ::=
    SimplePat |
    Con SimplePat |
    Pat ":" Type.
    SimplePat Con SimplePat |
    {Pat / ","}2 |

Match ::=
    {Pat "." Exp / "|"}1

Type ::=
    TypeVar |
    [TypeArgs] TypeIde |
    {Type / "#"}2 |
    Type "->" Type |
    "(" Type ")".

TypeArgs ::=
    Type |
    "(" {Type / ","}1 ")".

ModuleName ::= Ide | String.
```

Notes:

- The top-level command monitor provides a menu of check-prints for monitoring the inner workings of the compiler.
- The top-level command use *filename* loads a file of ML definitions and expressions.
- Recursive definitions can only contain bindings which are syntactically function bindings.

Appendix J. System installation

The ML system usually comes on a tape containing a single directory mlkit (and sometimes a backup copy of it mlkit.BACKUP). The tape can be read by a tar -x mlkit command, which creates an mlkit subdirectory in the current directory; mlkit can be kept in any user or system directory.

The mlkit directory contains several files and subdirectories. README repeats the information contained in this section. VERSION contains the system version. doc contains this manual in troff format, a paper on the ML abstract machine, and a list of known ML sites. src contains the source programs. progs contains some example ml programs. pub contains the ML object code, libraries and shell scripts.

To install the system enter the pub subdirectory, and read the install script to make sure that its execution is not going to cause any damage on your system. To run install you probably need write permission on /usr/lib and /usr/bin. When you are sure that everything is ok, type install.

Apart from install, pub contains an usr-bin-ml script and an usr-lib-ml subdirectory. The install procedure simply moves usr-bin-ml to /usr/bin/ml, and usr-lib-ml to /usr/lib/ml. usr-lib-ml contains the real ML executable code, called mlsys, a file which is loaded every time the system is entered, called boot, a precompiled library module, called lib.sp and lib.im, and the library module source, called lib.ml. The library lib.ml should be considered as part of the ML system, and it is loaded (by boot) every time the system is entered; it can be extended by the users, but not reduced. usr-lib-ml also contains other user-defined ML library modules, which have to be explicitely imported when needed in user programs, and a file genlib.ml which recompiles all the libraries.

After install, the ML system can be run from any directory by typing ml. If you instead wish to call the system FORTRAN, say, just rename /usr/bin/ml to /usr/bin/FORTRAN.

If for some reason you need to recompile the system, you should proceed as follows. Most of the work is done by the Makefile file supplied with the source programs (say: make in mlkit/src). This generates an mlsys executable file which should be stripped (say: strip mlsys) and moved to /usr/lib/ml/mlsys.

The new ML system is now available, but the ML library has to be recompiled. When loading the new system for the first time, an error message 'Module /usr/lib/ml/lib must be compiled' will appear. Immediately after, type 'use "/usr/lib/ml/genlib.ml";': this will generate the new libraries for all the future uses of the new system. Now exit the system (don't try to work: the library has not been loaded yet). From now on, the library will be loaded automatically when entering the system.

The library module lib.ml can be changed at will, as long as the basic ML primitives (like @) defined there are preserved. It contains some basic ML functions, and can be extended with the locally most used utilities. After editing the file /usr/lib/ml/lib.ml, enter ML and type 'use "usr/lib/ml/lib.ml";' to compile and install the new library. Exit the system and reenter it, to actually load the new library.

- Create this file, containing the C function "Address DoFoo(argn, .. ,arg1)", implementing the desired behavior. The order of arguments must be the inverse of the order in which "foo(arg1, .. ,argn)" is called from ML. WARNING: do not attempt to allocate or change ML data structures in this function without deeply understanding how the garbage collector works. It is safe to inspect (read only) the ML data structures passed as parameters (see appendix "Vax data formats"), and to return them unchanged. The result of this function must match the ML type declared in mlanal.p. If DoFoo needs to produce an ML exception, it can do so by calling "DoFailwith(*StrFoo)"

● File Makefile

- Add a line "FOOOP = amglob.h amfooop.c".
- Add "amfooop.o" to the definition of "MLSYS".
- Add a line "amfooop.o: $(FOOOP); CC amfooop.c".

Appendix M. VAX data formats

Each segment "+---+" in the pictures is one byte. The symbol "^" below a data structure represents the location pointed to by pointers to that structure; fields preceding "^" are only used during garbage collection and are inaccessible to the ML operations (and hence to the user). Unboxed data is kept on the stack, or in the place of pointers in other data structures; unboxed data does not require storage allocation. Pointers can be distinguished from unboxed data as the former are > 64K. There are automatic conversions between SmallIntegers and BigIntegers, so that the ML operations only see the type int.

Unit

0 (unity) (unboxed)

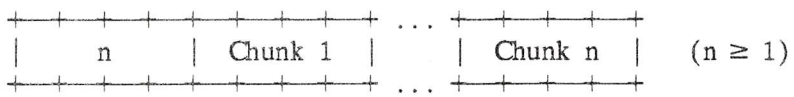
Boolean

0 (false) (unboxed)
1 (true) (unboxed)

SmallInteger

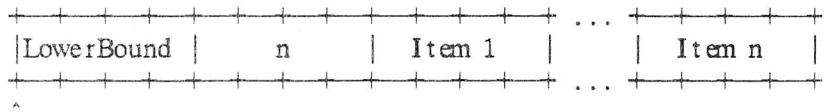
-32768 .. +32767 (unboxed)

BigInteger



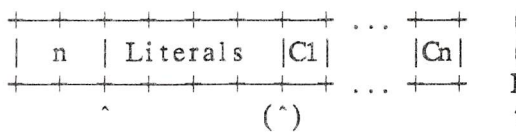
Small Record

Array



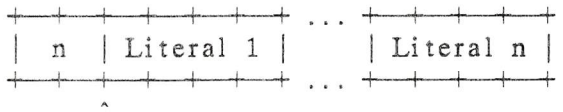
(n ≥ 0)

Text



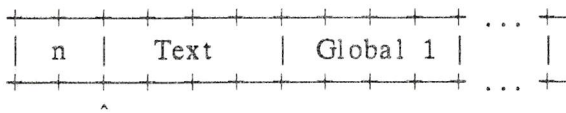
(n ≥ 1)
(^) when pointed from a Closure.
Literals may be Nil or point to Literals

Literals



(n ≥ 1)
Literal may be a Text, a String or a BigInteger

Closure



(n ≥ 0)
Text points to a Text

Appendix O. Transient

Several minor problems still afflict modules. These will be fixed gradually in the short and medium term. Meanwhile the following things should be noted.

With the exception of the library functions in /usr/lib/ml/lib.ml, modules must not use predefined functions as argumentless functional objects, e.g. + in fold (op +) [1;2;3;4] 0. At the moment this will produce and 'Unbound identifier' error. However, it is possible to replace (op +) by (fun(x,y). x+y).

Under some complicated circumstances the sharing of types will fail. The scenario is: module A privately imports module B, and then exports a type T defined in B by an export export list; then module C imports A and B and makes T-from-A interact with T-from-B. This will very likely never happen to you. However, if you get an unexplained type conflict (probably two different abstract types having the same name), this might be it. You should be able to fix this by making sure that that type is exported by includes declarations, as opposed to export export lists.

If a module A is recompiled, all the modules depending on it must be recompiled, even if the interface of A has not changed.

Sometimes, when a module B which imports A has to be recompiled, instead of the message Module B must be compiled, the less precise message Module importing A must be compiled is produced.

There is no check on the restriction that modules exporting a type identifier (maybe as part of the type of an exported binder), must also export the type definition. However, nothing bad can of from this.

On some occasions, precompiled modules can give undefined type errors when importing them, or when compiling modules which import them. This can happen if an export declaration rearranges the order of types and values in the body of a module. To fix this, put the export lists back in an appropriate dependency order. You can look at the .sp file for that module to see what is happening. Nothing bad can come from this.

Consider the following scenario: module A is defined; module B, which imports A, is defined; module B is imported; module A is redefined. At this point module B is obsolete, and one would expect a version error in importing B again. This would happen if module B had not already be imported; however an import B at this point will share the already imported module B, without any complaint. At this point we have a B including the old A, while we might expect the new A to be in action. To get the new A into play, you must recompile B and reimport it. Nothing bad can come of this, except confusion.

Input/Output. The current plans are to implement an efficient buffering with in-core cache for internal streams, and to use unbuffered 'raw' I/O on external streams. Internal streams would still look unbuffered to the user, but all the disk I/O would be done in chunks. The current situation is fluid.

At the moment a limited number of streams can be open in an ML session and they cannot be closed. When this limit is exceeded, the operations Stream and File fail.
This problem will be solved in a future version by an automatic cache of open streams.

The operations implemented so far are file, save, stream, caninput, input, output and terminal (not yet working: lookahead and channel).

The exception "corruption" is produced when something goes very wrong (e.g. after a very deep recursion which overruns the stack); the system is currently unsafe after a corruption. You should rarely be thrown out of the system, but on extreme corruptions you may get a Run out of memory message and a core dump.

On end of stream, inchar waits forever; type *DEL* to exit input waits and infinite loops (a trappable exception "interrupt" is produced) (ignore the longjmp botch and System Crash! messages which are sometimes generated).

Bignums are not yet implemented. Integers currently range between ~32768 and 32767.

References

[1] L.Cardelli. "The Functional Abstract Machine", Bell Labs Technical Memorandum TM-83-11271-1, Bell Labs Technical Report TR-107, 1983.

[2] M.Gordon, R.Milner, C.Wadsworth. "Edinburgh LCF", Lecture Notes in Computer Science, n.78, Springer-Verlag, 1979.

[3] R.Milner. "A theory of type polymorphism in programming", Journal of Computer and System Science 17, 348-375, 1978.

of date.

1.1. Expressions

ML is an expression-based language; all the standard programming constructs (conditionals, declarations, procedures, etc.) are packaged into expressions yielding values. Strictly speaking there are no statements: even side-effecting operations return values.

It is always meaningful to supply arbitrary expressions as arguments to functions (when the type constraints are satisfied), or to combine them to form larger expressions in the same way that simple constants can be combined.

Arithmetic expressions have a fairly conventional appearance; the result of evaluating an expression is presented as a value and its type separated by a colon:

```
expression:     (3 + 5) * 2;

result:             16 : int
```

Sequences of characters enclosed in quotes "" are called *strings*:

```
expression:     "this is it";

result:             "this is it" : string
```

Tuples of values are built by an infix operator ',' (comma); the type of a tuple is given by the infix type operator #, which denotes cartesian products.

```
expression:     3,4;

result:             3,4 : int # int

expression:     3,4,5;

result:             3,4,5 : int # int # int
```

Lists are enclosed in square brackets and their elements are separated by semicolons. The list type operator is a suffix: int list means list of integers.

```
expression:     [1; 2; 3; 4];

result:             [1; 2; 3; 4] : int list

expression:     [3,4; 5,6];

result:             [(3,4); (5,6)] : (int # int) list
```

Conditional expressions have the ordinary if-then-else syntax (but else cannot be omitted):

```
expression:     if true then 3 else 4;

result:             3 : int

expression:     if (if 3 = 4 then false else true)
                then false
```

infix operator; expressions like f 3 + 4 are parsed like (f 3) + 4 and not like f (3 + 4).

The function g above has a pair of arguments, a and b; in this case parentheses are needed for, otherwise g 8,4 is interpreted as (g 8), 4.

The identifiers f and g are plain variables which denote functions. Function variables do not need to be applied to arguments:

```
expression:     f, f 3;

result:             fun,4 : (int -> int) # int

declaration:    val h == g;

binding:            val h : (int # int) -> int
```

In the first example above, f is returned as a function and is paired with a number. Functional values are always printed fun, without showing their internal structure. In the second example, g is bound to h as a whole function. Instead of val h == g we could also have written val h(a,b) == g(a,b).

Variables are statically scoped, and their types cannot change. When new variables are defined, they may 'hide' previously defined variables having the same name, but they never 'affect' them. For example:

```
declaration:    val f x == a + x;

binding:            val f : int -> int

declaration:    val a == [1;2;3];

binding:            val a == [1;2;3] : int list

expression:     f 1;

result:             4 : int
```

here the function f uses the top-level variable a, which was bound to 3 in a previous example. Hence f is a function from integers to integers which returns its argument plus 3. Then a is redefined at the top level to be a list of three integers; any subsequent reference to a will yield that list (unless a is redefined again). But f is not effected at all: the old value of a was 'frozen' in f at the moment of its definition, and f keeps adding 3 to its arguments.

This kind of behavior is called *lexical* or *static* scoping of variables. It is quite common in block-structured programming languages, but it is rarely used in languages which, like ML, are interactive. The use of static scoping at the top-level may sometimes be counterintuitive. For example, if a function f calls a previously defined function g, then redefining g (e.g. to correct a bug) will not change f, which will keep calling the old version of g.

1.3. Local scopes

Declarations can be made local by embedding them between the keywords let and in in a scope-block construct. Following the keyword in there is an expression, called the *body* of the scope-block, terminated by the keyword end. The scope of the declaration is limited to this body.

```
expression:     let   val a == 3
                      and b == 5
```

1.4. Lists

List are *homogeneous*, i.e. all their elements must have the same type. It is possible to create lists of any type, e.g. lists of strings, lists of lists of integers, lists of functions (of some given type), etc. Many functions dealing with lists can work on lists of any kind (e.g. computing the length), and they do not have to be rewritten every time a new kind of list is needed. Other list functions are more specialized, like taking the sum of all the elements in a list. Even specialized functions can often be quickly defined from general list utilities; for example, summation can be defined by distributing the integer sum operation by a general fold function which folds lists under binary operations.

The fundamental list operations are nil, the empty list, and the infix :: (cons), which appends an element (its left argument) to the head of a list (its right argument). The square-brackets notation for lists (e.g. [1;2;3]), is an abbreviation for a sequence of cons operations terminated by nil; hence [] is another way of writing nil. The system always uses the square-brackets notation when printing lists.

```
nil                 is      []
1 :: [2; 3]         is      [1; 2; 3]
1 :: 2 :: 3 :: nil  is      [1; 2; 3]
```

Other predefined operations on lists include:

- null, which returns true if its argument is nil, and false on any other list.
- hd, which returns the first element of a non-empty list.
- tl, which strips the first element from the head of a non-empty list.
- @ (append), which concatenates lists.

```
null []             is      true
null [1; 2; 3]      is      false
hd [1; 2; 3]        is      1
tl [1; 2; 3]        is      [2; 3]
[1; 2] @ []         is      [1; 2]
[] @ [3; 4]         is      [3; 4]
[1; 2] @ [3; 4]     is      [1; 2; 3; 4]
```

1.5. Functions

All functions take exactly one argument and deliver exactly one result. However arguments and results can be arbitrary structures, thereby achieving the effect of passing multiple arguments and returning multiple results. For example:

```
declaration:    val plus(a,b) == a + b;
```

the function plus above takes a *single* argument (a,b), which is a pair of values. This could seem to be a pointless distinction, but consider the two following ways of using plus:

```
expression:     plus(3,4);

expression:     let val x == 3,4
                in plus x end;
```

The first use of plus is the standard one: two arguments 3 and 4 seem to be supplied to it. However we have seen that ',' is a pair-building operator, and it constructs the pair 3,4 *before* applying plus to it†. This is clearer in the second use of plus, where plus receives a single argument, which

† This is true conceptually; in practice a compiler can optimize away the extra pair constructions in most situations.

```
declaration:    val rec summation nil == 0 |
                    summation (head :: tail) == head + summation tail;

binding:            val summation : int list -> int
```

The patterns are evaluated sequentially from left to right. When a pattern matches the argument, the variables in the pattern are bound to the respective parts of the argument, and the corresponding action is executed. It there are several patterns matching some argument, only the first one is activated. If all patterns fail to match some argument, a run-time exception occurs.

1.6. Polymorphism

A function is said to be *polymorphic* when it can work uniformly over a class of arguments of different data types. For example consider the following function, computing the length of a list:

```
declaration:    val rec length nil == 0 |
                    length ($ :: tail) == 1 + length tail;

binding:            val length : 'a list -> int

expression:     length [1; 2; 3], length ["a"; "b"; "c"; "d"];

result:             3,4 : int#int
```

The type of length contains a *type variable* ('a), indicating that any kind of list can be used; e.g. an integer list or a string list. Any identifier or number prefixed by a prime ', and possibly containing more primes, is a type variable.

A type is called *polymorphic*, or a *polytype*, if it contains type variables, otherwise it is called *monomorphic*, or a *monotype*. A type variable can be replaced by any ML type to form an *instance* of that type, for example, int list is a monomorphic instance of 'a list. The instance types can contain more type variables, for example ('b # 'c) list is a polymorphic instance of 'a list.

Several type variables can be used in a type, and each variable can appear several times, expressing contextual relationships between components of a type; for example, 'a # 'a is the type of all pairs having components of the same type. Contextual constraints can also be expressed between arguments and results of functions, like in the identity function, which has type 'a -> 'a, or the following function which swaps pairs:

```
declaration:    val swap(x,y) == y,x;

binding:            val swap : ('a # 'b) -> ('b # 'a)

expression:     swap([],"abc");

result:             "abc",[] : string # ('a list)
```

Incidentally, you may notice that the empty list [] is a polymorphic object of type 'a list, in the sense that it can be considered an empty integer list, or an empty string list, etc..

In printing out polymorphic types, the ML system uses the type variables 'a, 'b, etc. in succession (they are pronounced 'alpha', 'beta', etc.), starting again from 'a at every new top-level declaration. After 'z there are ''a, .. ''z, '''a, .. '''z, etc., but these are rarely necessary. Type variables are really anonymous objects, and it is not important how they are expressed as long as the contextual relations are clear.

Several primitive functions are polymorphic. For example, we have already encountered the list

```
binding:        val comp : (('a -> 'b) # ('c -> 'a)) -> ('c -> 'b)

declaration:    val fourtimes == comp(twice,twice);

binding:        val fourtimes : int -> int

expression:     fourtimes 5;

result:         20 : int
```

Function composition comp takes two functions f and g as arguments, and returns a function which when applied to an argument x yields f (g x). Composing twice with itself, by partially applying comp to the pair twice,twice, produces a function which multiplies numbers by four. Function composition is also a predefined operator in ML; it is called o (infix), so that the composition of f and g can be written f o g.

Suppose now that we need to partially apply a function f which, like plus, takes a pair of arguments. We could simply redefine f as val f a b == f(a,b): the new f can be partially applied, and uses the old f with the expected kind of arguments.

To make this operation more systematic, it is possible to write a function which transforms any function of type, ('a # 'b) -> 'c (which requires a pair of arguments) into a function of type 'a -> ('b -> 'c) (which can be partially applied); this process is usually called *currying* a function:

```
declaration:    val curry f a b == f(a,b);

binding:        val curry : (('a # 'b) -> 'c) -> ('a -> ('b -> 'c))

declaration:    val curryplus == curry plus;

binding:        val curryplus : int -> (int -> int)

declaration:    val successor == curryplus 1;

binding:        val successor : int -> int
```

The higher-order function curry takes any function f defined on pairs, and two arguments a and b, and applies f to the pair (a,b). If we now partially apply curry to plus, we obtain a function curryplus which works exactly like plus, but which can be partially applied.

1.8. Exceptions

Certain primitive functions may raise *exceptions* on some arguments. For example, division by zero interrupts the normal flow of execution and escapes at the top level with an error message informing us that division failed:

```
expression:     1 div 0;

exception:      Exception: div
```

Another common exception is raised when trying to extract the head of an empty list:

```
expression:     hd [];

exception:      Exception: hd
```

integer pairs, lists and functions. Strictly speaking, basic types are type operators which take no arguments. Type operators can be arbitrarily nested: e.g. ((int -> int) list) list is the type of lists of lists of integer functions.

Type variables can be used to express polymorphic types. Polytypes are mostly useful as types of functions, although some non-functional objects, like [] : 'a list, are also polymorphic. A typical example of polymorphic function is hd : 'a list -> 'a, which extracts the first element (of type 'a) of a list (of type 'a list, in general). The type of hd tells us that hd can work on any list, and that the type of the result is the same as the type of the elements of the list.

Every type denotes a *domain*, which is a set of objects, all of the given type; for example int # int is (i.e. denotes) the domain of integer pairs, and int -> int is the domain of all integer functions. An object can have several types, i.e. it can belong to several domains. For example the identity function (fun x. x) has type int -> int as it maps any object (of type integer) to itself (of type integer), but it is also has the type bool -> bool for the same reason. The most general type for the identity function is 'a -> 'a because all the types of the identity are instances of it. 'a -> 'a gives more information than int -> int, for instance, because the former encompasses all the types that the identity function can have and thus expresses all the ways that the identity function can be used. Hence 'a -> 'a is a 'better' type for (fun x. x), although int -> int is not wrong. The ML typechecker always determines the 'best' type for an expression, i.e. the one which corresponds to the smallest domain, given the information contained in that expression.

A type constraint can be appended to an expression, in order to *specify* the type of the expression:

```
expression:     3 : int;

result:             3 : int

expression:     [3,4; (5,6) : int # int];

result:             [(3,4); (5,6)] : (int # int) list
```

In the above examples, the type specifications following ':' do not really have any effect. The types are independently inferred by the system and checked against the specified types. Any attempt to specify a type incorrectly will result in a type error:

```
expression:     3 : bool;

type error:         Type Clash in:    3 : bool
                    Looking for :     bool
                    I have found :    int
```

However, a type specification can restrict the types inferred by the system by constraining polymorphic objects or functions:

```
expression:     [] : int list;

result:             [] : int list

expression:     (fun x. x) : int -> int;

result:             fun : int -> int
```

note that the type normally inferred for [] is 'a list and for (fun x. x) is 'a -> 'a.

Type specifications can be used in bindings:

Note that quarter is not a constructor, and we cannot have a clause 'amount quarter == 25' in the above definition (quarter would be interpreted as a normal variable, like x).

A type can be entirely made of constants, in which case we have something similar to Pascal enumeration types. A type can also have a single constructor; then the type definition can be considered as an *abbreviation* for the type following of.

```
declaration:    type color == red | blue | yellow;

declaration:    type point == point of int # int;
```

If the definition of a type involves type variables, the type is called *parametric* and all the type variables used on the right hand side of the == must be listed on the left hand side as type parameters: here are type operators with one and two type parameters:

```
declaration:    type 'a predicate == predicate of 'a -> bool
                and ('b,'c) leftprojection == leftprojection of ('b # 'c) -> 'a;

bindings:           type 'a predicate == 'a -> bool
                    con predicate : ('a -> bool) -> ('a predicate)
                    type ('b,'c) leftprojection == leftprojection of ('b # 'c) -> 'b
                    con leftprojection : (('b # 'c) -> 'b) -> ('b,'c) leftprojection
```

Recursive types are introduced by the keyword rec. Here is how the predefined list type is defined:

```
declaration:    type rec 'a list == nil | :: of 'a # 'a list;

bindings:           type 'a list == nil | :: of 'a # ('a list)
                    con nil : 'a list
                    con :: : ('a # ('a list)) -> ('a list)
```

1.11. Abstract types

An abstract data type is a type with a set of operations defined on it. The structure of an abstract type is hidden from the user of the type, together with the all the constructors of the underlying structure. The associated operations are the only way of creating and manipulating objects of that type. However, the hidden structure and constructors are available while defining the operations.

An abstract data type provides an interface between the use and the implementation of a set of operations. The structure of the type, and the implementation of the operations, can be changed while preserving the external meaning of the operations.

For example we can define an abstract type (additive-) color which is a combination of three primary colors which can each have an intensity in the range 0..15:

```
declaration:    abstype color == blend of int # int # int
                with val white  == blend(0,0,0)
                and red == blend(15,0,0)
                and blue == blend(0,15,0)
                and yellow == blend(0,0,15)
                and mix (parts: int, blend(red,blue,yellow): color,
                      parts': int, blend(red',blue',yellow'): color) : color ==
                      if parts < 0 or parts' < 0 then escape "mix"
                      else  let val totalparts == parts + parts'
                            in blend(
```

There is no provision in the ML system for editing or storing programs or data which are created during an ML session. Hence ML programs and definitions are usually prepared in text files and loaded interactively. The use of a multi-window editor like Emacs is strongly suggested: one window may contain the ML system, and other windows contain ML definitions which can be modified (without having to exit ML) and reloaded when needed, or they can even be directly pasted from the text windows into the ML window.

An ML source file looks exactly like a set of top-level phrases, terminated by semicolons. A file prog.ml containing ML source code can be loaded by the use top-level command:

```
use "prog.ml";
```

where the string surrounded by quotes can be any Unix file name or path. The files loaded by use may contain more use commands (up to a certain stacking depth), so that the loading of files can be cascaded. The file extension .ml is normally used for ML source files, but this is not compulsory.

When working interactively at the top level, the first two characters of every line are reserved for system prompts.

```
- val a == 3
  and f x == x + 1;
> val a == 3 : int
| val f : int -> int

- a + 1;
  4 : int
```

'- ' is the normal system prompt, indicating that a top-level phrase is expected, and ' ' is the continuation prompt indicating that the phrase on the previous line has not been completed. Again, all the top level expressions and declarations must be terminated by a semicolon.

In the example above, the system responded to the first definition by confirming the binding of the variables a and f; '> ' marks the confirmation of the a new binding, followed '| ' on the subsequent lines when many variables are defined simultaneously. In general, the prompts '> ' and '| ' are followed by a variable name, an == sign, the new value of the variable, a ':', and finally the type of the variable. If the value is a functional object, then the == sign and the value are omitted.

When an expression is evaluated, the system responds with a blank prompt ' ', followed by the value of the expression (just fun for functions), a ':' and the type of the expression.

Every phrase-answer pair is followed by a blank line.

The variable it is always bound to the value of last expression evaluated; it is undefined just after loading the system, and is not affected by declarations or by computations which for some reason fail to terminate.

```
- 1 + 1;
  2 : int

- it;
  2 : int

- it + 1;
  3 : int

- val a == 5;
> val a == 5 : int
```

The example above is a common matching type error. A different kind of matching error can be generated by self-application, and in general by constructs leading to circularities in the type structures:

```
- fun x. x x;
Type Clash  in:    (x x)
Attempt to build a self-referential type
by equating var:   'a
to type expr:      'a -> 'b

- val rec f $ == f;
Type Clash  in:    f == (fun (). f)
Attempt to build a self-referential type
by equating var:   'a
to type expr:      'b -> 'a
```

Less common error messages are described in section "Error messages".

2. Lexical matters

The following lexical entities are recognized: identifiers, keywords, integers, strings, delimiters, type variables and comments. See appendix "Syntax of lexical entities" for a definition of their syntax.

An identifier is either (i) a sequence of letters, numbers, primes ‘’’ and underscores ‘_’ not starting with a digit, or (ii) a sequence of special characters (like *, @, /, etc.).

Keywords are lexically like identifiers, but they are reserved and cannot be used for program variables. The keywords are listed in appendix "Keywords".

Integers are sequences of digits.

Strings are sequences of characters delimited by string quotes ‘"’. String quotes and non-printable characters can be introduced in strings by using the escape character ‘\’. The details of the escape mechanism are described in appendix "Escape sequences for strings".

Type variables are only accepted in type expressions. They are identifiers starting with the character ‘’’. For example: ', '', '1 'tyvar, etc.

Delimiters are parentheses ((, [,) and]), and other one-character punctuation marks like ‘.’ and ‘;’. Delimiters never stick to any other character, so that no space is ever needed around them.

Comments are enclosed in curly brackets ‘{’ and ‘}’, and can be nested.

3. Expressions

Expressions denote values, and have a type which is statically determined. Expressions can be constants, identifiers, data constructors, conditionals, fun-expressions, function applications, infix operator applications, scope blocks, while expressions, case expressions, exception and trap expressions, and type specifications. All these different kinds of expressions are described in this section.

3.1. Unit

The expression () (called *unity*) denotes the only object of type unit.

```
()          : unit          (constant)
```

The unit type is useful when defining functions with no argument or no values: they take a unit argument or produce a unit result. There are no primitive operations on unity.

The unit type is not strictly primitive; it could be introduced by the declaration:

3.4. Tuples

A *tuple* is a fixed-length heterogeneous sequence of values. The unity value (i.e. ()) might be considered as a zero-length tuple. There are no tuples of length one; they are identified with their unique component. Tuples of length two or more are written as sequences of values separated by commas:

syntax: $exp_1, .. , exp_n$ $n \geq 2$

Tupling is an n-ary operation; in order to build non-flat tuples of tuples, parentheses must be used. For example the following is not a 4-tuple: it is a triple, whose second component is a pair:

```
expression:   1, (2, 3), 4;

result:       1,(2,3),4 : int # (int # int) # int
```

The type of an n-ary tuple is an n-ary cartesian product, denoted by the symbol #. Parentheses can be used to express non-flat cartesian products.

There are no primitive functions on tuples, except the basic use of commas to build them (note that it is not possible to define a function which extracts the first element of a tuple for tuples of any length). Destructuring must be done by pattern matching:

```
declaration:  val a,b,c == 1,2,3;

result:       a == 1 : int
              b == 2 : int
              c == 3 : int
```

The type of a tuple is the cartesian product of the types of its components:

typing rule: if $e_1 : t_1$ and .. and $e_n : t_n$
then $(e_1 , .. , e_n) : t_1 \# .. \# t_n$

Tuples are one of the few really fundamental types in ML; they cannot be explained in terms of other constructs in the language.

3.5. Lists

A list of elements exp_i is written:

syntax: $[exp_1; .. ; exp_n]$ $n \geq 0$

with [] as the empty list. All the elements of a list must have the same type, otherwise a compile-time type error will occur.

The basic constructors for lists are:

```
nil     : 'a list                        (constant)
::      : ('a # 'a list) -> 'a list      (infix constructor)
```

- nil is the empty list (the same as []).
- :: (cons) adds an element to a list (e.g. 1::[2;3;4] is the same as [1;2;3;4]).

Some other frequent operations on lists are predefined in the system:

- The escape character for strings is \; the conventions used to introduce quotes and non-printable characters inside strings are described in appendix "Escape sequences for strings".
- size returns the length of a string.
- extract extracts a substring from a string: the first argument is the source string, the second argument is the starting position of the substring in the string (the first character in a string is at position 1), and the third argument is the length of the substring; it escapes with string "extract" if the numeric arguments are out of range.
- explode maps a string into the list of its characters, each one being a 1-character string.
- implode maps a list of strings into a string which is their concatenation.
- explodeascii is like explode, but produces a list of the Ascii representations of the characters contained in the string.
- implodeascii maps a list of numbers interpreted as the Ascii representation of characters into a string containing those characters; it escapes with string "implodeascii" if the integers are not valid Ascii codes.
- intofstring converts a numeric string to the corresponding integer number, negative numbers start with '~'; it may escape with string "intofstring" if the string is not numeric.
- stringofint converts an integer to a string representation of the necessary length; negative numbers start with '~'.

For Ascii characters we intend here full 8-bit codes in the range 0..255.

3.7. Updatable references

Assignment operations act on *reference* objects. A reference object is an updatable pointer to another object. References are, together with arrays, the only data objects which can be side effected; they can be inserted anywhere an update operation is needed in variables or data structures.

References are created by the operator ref, updated by := and dereferenced by !. The assignment operator := always returns unity. Reference objects have type t ref, where t is the type of the object contained in the reference.

```
ref     : 'a -> 'a ref                  (constructor)
!       : 'a ref -> 'a
:=      : 'a ref # 'a -> unit           (infix)
```

Here is a simple example of the use of references. A reference to the number 3 is created and updated to contain 5, and its contents are then examined.

```
declaration:    val a == ref 3;

binding:            val a == ref 3 : int ref

expression:     a := 5;

result:             () : unit

expression:     ! a;

result:             5 : int
```

References can be embedded in data structures. Side effects on embedded references are reflected in all the structures which share them:

```
declaration:    type 'a refpair == refpair of 'a ref # 'a ref;
```

```
export

    abstype array
    val array arrayoflist lowerbound arraysize sub update arraytolist

from

    type 'a array == arr of {lowerbound:} int # {size:} int # {table:} 'a ref list;

    val rec el (n: int, (head :: tail): 'a list) : 'a ==
        if n=0 then head else el(n-1,tail);

    val array (lb: int, length: int, item: 'a) : 'a array ==
        if length<0 then escape "array"
        else arr(lb,length,list length)
            where rec val list n ==
                if n=0 then [] else (ref item) :: list(n-1)
            end

    and arrayoflist (lb: int, list: 'a list) : 'a array ==
        arr(lb,length list,map ref list)

    and lowerbound (arr(lb,$,$): 'a array) : int == lb

    and arraysize (arr($,size,$): 'a array) : int == size

    and (arr(lb,size,table): 'a array) sub (n: int) : 'a ==
        let val n' == n-lb
        in  if n'<0 or n'>=size then escape "sub"
            else !(el(n',table))
        end

    and (arr(lb,size,table): 'a array) update (n: int, value: 'a) : unit ==
        let val n' == n-lb
        in  if n'<0 or n'>=size then escape "update"
            else el(n',table):=value
        end

    and arraytolist (arr($,$,table): 'a array) : 'a list ==
        map (op !) table

end;
```

Applicative programming fans should type the following declarations, or introduce them in the standard library:

```
type 'a ref == ref;
type 'a array == array;
val op := == ();
val ! == ();
val arrayoflist == ();
val lowerbound == ();
val arraysize == ();
val op sub == ();
```

appear as part of value declarations, functions and case expressions.

A *match* is a sequence of pattern-action pairs separated by vertical bars.

syntax: *match* ::=
pat . exp | .. | *pat . exp*

Matches are used to destructure values and to execute fragments of code depending on the form of values: they replace test and selection operations. The patterns in a match are matched in turn, from left to right, against some value *v*. The first successful match is used to destructure *v*, and the variables in the pattern are bound to the respective parts of *v*. Then the corresponding expression is executed, and its result is returned as the result of the matching. If none of the patterns matches *v*, then an exception "match" is raised.

Matches are not expressions; they can only appear in function expressions, case expressions and, in a sugared form, in clausal function definitions.

3.10. Functions

Functions can be introduced by a *curried definition*, for partially applicable functions, or by a *clausal definition*, for case analysis:

syntax: {op} *f simp_pat .. simp_pat* {: *type*} == *exp* (curried)

syntax: {op} *f simp_pat* {: *type*} == *exp* | .. | *f pat* {: *type*} == *exp* (clausal)

- The op keyword should be used when *f* is an infix; alternatively *pat f pat'* can be used instead of op *f (pat,pat')*.
- The *simp_pat* patterns are $, variables, constants, lists and parenthesized patterns, in particular they must be parenthesized when they have the form (*pat' : type*).
- The optional *type* expressions refer to the type of the result of the function.
- Curried functions cannot be defined by case analysis: if needed, the arguments can be analyzed after the ==.

Unnamed functions can be introduced as expressions by the use of the *fun-notation*, which is just a match preceded by the keyword fun:

syntax: fun *match*

A fun expression denotes a function, which can be associated with an identifier by a normal value definition, for example:

```
declaration:    val plus == fun x . fun y . x + y

declaration:    val null == fun nil . true | a :: b . false;
```

A curried function definition is a sugaring of the first declaration above, and a clausal function definition is a sugaring of the second declaration:

```
declaration:    val plus x y == x + y

declaration:    val plus nil == true | plus (a :: b) == false;
```

The type of every expression and pattern can be specified by suffixing a colon and a type expression to it (note that parentheses are needed around type specifications in patterns, when they are

appendix "Precedence of operators" for details. In case of partial application, the form *f g a*, is interpreted as *(f g) a*, and not as *f (g a)*; note that they are both equally meaningful.

Infix identifiers can be applied in two ways:

syntax: *a f b*

syntax: op *f* (*a*,*b*)

The first form is the expected one; an infix identifier has always a type matching ('a # 'b) −> 'c. The second possibility derives from the fact that there must be a way of passing an infix operator f as an argument to a function g; g(f) will produce a syntax error, but g(op f) is accepted. In general the op keyword coerces any infix identifier to a nonfix one.

The typing rule for function application states that the type of the argument must match the type of the domain of the function:

typing rule: if $f : t \rightarrow t'$ and $a : t$ then $(f\ a) : t'$

For example, (fun x. x) has the type 'a −> 'a and all the instances of it, e.g. int −> int. Hence (fun x. x) 3 has type int.

3.12. Conditional

The syntactic form for conditional expressions is:

syntax: if e_1 then e_2 else e_3

The expression e_1 is evaluated to obtain a boolean value. If the result is true then e_2 is evaluated; if the result if false then e_3 is evaluated. Example:

```
expression:   val sign n ==
                  if n < 0 then ~1
                  else if n = 0 then 0
                  else 1;
```

The else branch must always be present.

The typing rule for the conditional states that the if branch must be a boolean, and that the then and else branches must have compatible types:

typing rule: if $e_1 : \text{bool}$ and $e_2 : t$ and $e_3 : t$
then (if e_1 then e_2 else e_3) : t

Note that the types of two branches only have to match, not to be identical. If one branch is more polymorphic than the other, than it also has the type of the other (by instantiation of the type variables), and the above rule can be applied.

3.13. Sequencing

When several side-effecting operations have to be executed in sequence, it is useful to use *sequencing*:

syntax: $(e_1; \ldots ; e_n)$ $\quad n \geq 2$

(the parentheses are needed), which evaluates $e_1 \ldots e_n$ in turn and returns the value of e_n. The

```
and fruit == apple | plum | banana;
```

bindings:
```
type color == red | purple | yellow
type fruit == apple | plum | banana
```

declaration:
```
val fruitcolor (fruit: fruit): color ==
    case fruit of
        apple. red |
        banana. yellow |
        plum. purple;
```

binding:
```
val fruitcolor : fruit -> color
```

The case construct is a convenient form of saying if fruit=apple then red else if fruit=banana then yellow else if fruit=plum then purple else escape"match" (else escape"match is never executed in this case because the list of cases is exhaustive).

The typing rule for case is:

typing rule: if $x : t'$ and $p_1 : t'$ and .. and $p_n : t'$
and $e_1 : t$ and .. and $e_n : t$
then (case x of $p_1 . e_1 | .. | p_n . e_n$) : t

This says that all the patterns must have the same type as value being inspected, and that the values returned for each case must have the same type.

3.16. Scope blocks

A *scope block* is a control construct which introduces new variables and delimits their scope. Scope blocks have the same function of begin-end constructs in Algol-like languages, but they have a fairly different flavor due to the fact of being expressions returning values, instead of groups of statements. There are two kinds of scope blocks:

syntax: let *declaration* in *exp* end

syntax: *exp* where *declaration* end

The let construct introduces new variable bindings in the *declaration* part which can be used in the *exp* part (and there only). Newly introduced variables hide externally declared variables having the same name for the scope of *exp*; the externally declared variables remain accessible outside *exp*.

The where construct behaves just like let; it simply inverts the order in which the expression and the declaration appear.

The value returned by a scope block is the value of its expression part. Similarly the type of a scope block is the type of its expression part. The different kinds of declarations are described in section "Declarations".

typing rule: if $e : t$ then (let d in e end) : t

3.17. Exceptions and traps

An *exception* (sometimes also called a *failure*) can be raised by a system primitive or by a user program. When an exception is raised, the execution of the current expression is abandoned, and an *exception packet* is propagated outward, tracing back along the history of function calls and expression evaluations which led to the exception. If the exception is allowed to propagate up to the top

3.18. Type semantics and type specification

The set of all values is called V: it contains basic values, like integers, and all the composite objects built on elements of V, like pairs, functions, etc. The structure of V can be given by a recursive type equation, of which there are known solutions (+ is disjoint union, # is cartesian product and → is continuous function space):

$$V = Bool + Int + .. + (V + V) + (V \# V) + (V \to V) \qquad [1]$$

All functions are interpreted as functions from V to V, so when we say that f has type int −> bool we do not mean that f has domain int and codomain bool in the ordinary mathematical sense. Instead we mean that *whenever* f *is given an integer argument it produces a boolean result*. This leaves f free to do whatever it likes on arguments which are not integers: it might return a boolean, or it might not. This idea leads to the following definition for function spaces:

$$A \to B = \{f \in V \to V \mid a \in A \text{ implies } f\ a \in B\} \qquad [2]$$

where → is the conventional continuous function space, while −> is the different concept that we are defining. A and B are *domains* included in V, and A −> B is a domain included in V → V, and hence embedded in V by [1].

In this way we can give meaning to monotypes like int −> int, but what about polytypes? Consider the identity function (fun x. x) : 'a −> 'a; whenever it is given an argument of type 'a it returns a result of type 'a. This means that when given an integer it returns an integer, when given a boolean it returns a boolean, etc. Hence by [2], fun x. x belongs to the domains int −> int, bool −> bool, and to $d \to d$ for any domain d. Therefore fun x. x belongs to the intersection of all those domains, i.e. to $\cap_{d \in T}\ d \to d$ (where T is the set of all domains). We can now take the latter expression as the meaning of 'a −> 'a.

In general a polymorphic object of type σ['a] belongs to all the domains σ[d/'a], and hence to their intersection.

Some surprising facts derive from these definitions. First, 'a is not the type of all objects, as one might expect; in fact the meaning of 'a is the intersection of all domains. The only element contained in all domains is the divergent computation, which is therefore the only object of type 'a.

Second, 'a −> 'a, as a domain, is smaller than any of its instances, for example int −> int. In fact any function returning an 'a when given an 'a, must return an int when given an int (i.e. 'a −> 'a $\subseteq$ int −> int), but an int −> int function is not required to return a boolean when given a boolean (i.e. int −> int $\supset$ 'a −> 'a). Hence a function like (fun x. x+1) : int −> int is not an 'a −> 'a.

Similarly, 'a list as a domain is smaller than int list, bool list, etc. In fact 'a list is the intersection of all list domains and only contains the empty list (and the undefined computation).

When *specifying* a type in ML, by the notation:

syntax: *exp* : *type*

the effect is to take the *meet* of the type of the expression inferred by the system and of the type following ':' specified by the user (if this meet exists) as the type of the expression. The meet of two domains is their intersection.

This implicit join operation explains why the specifications:

expression: 3 : 'a;

result: 3 : int

4. Type expressions

A type expression is either a *type variable*, a *basic type*, or the application of a *type operator* to a sequence of type expressions. Type operators are usually suffix, except for cartesian product # and the infix operator ->.

4.1. Type variables

A type variable is an identifier starting with ' and possibly containing other ' characters. Type variables are used to represent polymorphic types.

4.2. Type operators

A basic type is a type operator with no parameters, like int. A parametric type operator, like -> or list, takes one or more arguments which are arbitrary type expressions, as in 'a -> ('a list) list). If an operator takes many parameters, these are separated by commas and enclosed in parentheses, as in ('a, 'b) tree. See appendix "Predefined type identifiers" for a list of the predefined type operators.

5. Declarations

Declarations are used to establish bindings of variables to values. Every declaration is said to *import* some variables, and to *export* other variables. The imported variables are the ones which are used in the declaration, and are usually defined outside the declaration (except in recursive declarations). The exported variables are the ones defined in the declaration, and that are accessible in the scope of the declaration.

A declaration can be a value binding, a type binding, a sequential composition, a private declaration, a module import, a lexical declaration or a parenthesized declaration.

syntax:
declaration ::=
    val *value_binding*
    type *type_binding*
    abstract_declaration
    sequential_declaration
    private_declaration
    import_declaration
    lexical_declaration
    (*declaration*)

5.1. Value bindings

Value bindings define values and functions, and are prefixed by the keyword val. After val there can be a simple definition, a parallel binding or a recursive binding.

syntax:
value_binding ::=
    simple_value_binding
    parallel_value_binding
    recursive_value_binding

5.1.1. Simple bindings

The simplest form of value binding, called a *value definition*, introduces a single pattern or function:

syntax:
simple_value_binding ::=
    pat == *exp*

bindings:
```
val a == 5 : int
val b == 10 : int
```

5.1.3. Recursive bindings

The operator rec builds recursive bindings, and it is used to define recursive and mutually recursive functions. The binding rec *d* exports the variables exported by the binding *d*, and imports the variables imported by *d* and the variables exported by *d*.

syntax:
```
recursive_value_binding ::=
        rec value_binding
```

declaration:
```
val rec fib n == if n < 2 then 1 else fib(n-1) + fib(n-2);
```

Note that if rec were omitted, the identifier fib on the right of == would not refer to its defining occurrence on the left of ==, but to some previously defined fib value (if any) in the surrounding scope.

A recursive value declaration can only define functions; simple value bindings are not allowed to be recursive. Moreover, all the bindings contained in a rec must be *syntactically* functions, e.g. simple, curried or clausal function bindings, or simple variable bindings having fun-expressions on the right of ==. A definition like val rec g == f; is not accepted, even if f is a function.

5.2. Type bindings

Type bindings define new types and their constructors, and are prefixed by the keyword type. After type there can be a definition, a parallel binding or a recursive binding.

syntax:
```
type_binding ::=
        simple_type_binding
        parallel_type_binding
        recursive_type_binding
```

5.2.1. Simple bindings

The simplest form of type binding,introduces a single, possibly parametric, type and its constructors:

syntax:
```
simple_type_binding ::=
        type_Ide == type_cases
        type_var type_Ide == type_cases
        (type_var, .. ,type_var) type_Ide == type_cases
```

syntax:
```
type_cases ::=
        con {of type_exp} | .. | con {of type_exp}
```

Here are examples of types with zero, one and two type parameters:

declaration:
```
type intpair == intpair of int # int;
```

bindings:
```
        type intpair == intpair of int # int
        con intpair : (int # int) -> intpair
```

declaration:
```
type 'a pair == pair of 'a # 'a;
```

```
with  val origin == position (0,0)
      and stepx (position(x,y)) == position(x+1,y)
      and stepy (position(x,y)) == position(x,y+1)
end;
```

bindings:

```
abstype position
val origin == - : position
val stepx : position -> position
val stepy : position -> position
```

In this example, a position can only be achieved by small steps from the origin. We cannot 'jump' to an arbitrary position because there is no operation to directly build one.

Here is the definition of a parametric recursive type of binary trees with leaves of type 'a:

declaration:

```
abstype rec 'a tree == leaf of 'a | node of 'a tree # 'a tree
with  val mkleaf a == leaf a
      and mknode (t,t') == node(t,t')
      and isleaf (leaf $) == true | isleaf(node $) == false
      and left (node(t,$)) == t
      and right (node($,t)) == t
      and tip (leaf a) == a
end;
```

bindings:

```
abstype 'a tree
val mkleaf : 'a -> 'a tree
val mknode : ('a tree # 'a tree) -> 'a tree
val isleaf : 'a tree -> bool
val left : 'a tree -> 'a tree
val right : 'a tree -> 'a tree
val tip : 'a tree -> 'a
```

The constructors leaf and node are only known during the definition of the basic operations on trees, and are not exported outside the abstract type definition. All users of the tree abstract type will be unable to take advantage of the concrete representation of trees given by 'leaf of 'a | node of 'a tree # 'a tree', thus making this type 'abstract'.

The with operator can be preceded by several type definitions connected by and and rec; this allows one to define mutually recursive abstract types:

declaration:

```
abstype rec a == .. b ..
      and b == .. a ..
with .. end;
```

5.4. Sequential declarations

Cascaded, or 'sequential' declarations are provided by the environment operator ';', which makes earlier declarations available inside later declarations.

syntax:

sequential_declaration ::=
declaration ; *declaration*

In d_1 ; d_2, the variables exported by d_1 are imported into d_2, but not vice versa. The variables exported by the whole declaration are the ones exported by d_2, plus the ones exported by d_1

```
bindings:       abstype increasing_pair
                val increasing_pair : (int # int) -> increasing_pair
                val low : increasing_pair -> int
                val high : increasing_pair -> int
```

In the first example, the variable b is unknown at the top level. The second example shows how a variable can be made private to a function or a group of functions: only the functions increment and fetch have access to count, so that nobody can corrupt count. The third example is an abstract data type increasing_pair: note that the constructor pair has been hidden, so that it is impossible to generate non-increasing pairs.

The 'abstype *tb* with *d*' declaration construct (see section "Abstract declarations") is just an abbreviation for 'export *xl* from type *tb; d* end', where the types defined in *tb* are listed in an abstype export list in *xl*, and the types and values defined in *d* are listed in appropriate export lists in *xl*.

The increasing_pair example above can be written more conveniently as:

```
declaration:    abstype increasing_pair == pair of int # int
                with  val increasing_pair(x,y) ==
                            if x>y then escape"increasing_pair"
                            else pair(x,y)
                      and low (pair(x,y)) == x
                      and high (pair(x,y)) == y
                end;

bindings:       abstype increasing_pair
                val increasing_pair : (int # int) -> increasing_pair
                val low : increasing_pair -> int
                val high : increasing_pair -> int
```

5.6. Import declarations

The import environment operator acts on precompiled modules, and it is explained in detail in section "Modules". A declaration import *M* imports no variables from the surrounding environment, and exports the variables exported by the module *M*.

```
syntax:         import_declaration ::=
                      import module_name .. module_name
-

declaration:    import minmax;

bindings:       val min : (int # int) -> int
                val max : (int # int) -> int
```

where the module minmax contains definitions for the min and max functions.

5.7. Lexical declarations

Special kinds of declaration, introduced by the keywords nonfix and infix are used to specify lexical attributes of identifiers. These declarations have only a lexical meaning; they do not introduce bindings, nor causes evaluations.

All identifiers have a property called *fixity*, which can be *infix* or *nonfix* (i.e. normal). The fixity of an identifier determines the way arguments are syntactically supplied to it (see section "Application" for the patterns of usage). Fixity can be specified in a *lexical* declaration, before the identifier is used in declarations or expressions:

• Output operations are constructive; the characters written are appended to the end of the stream. output writes a string of characters at the end of a stream.

• The operation lookahead reads characters from a stream without affecting it. The arguments are like in the string operation extract: the first argument is the source stream, the second argument is the starting position of the string to be extracted from the stream (the first character in a stream is at position 1), and the third argument is the length of the string to be extracted; it escapes with string "lookahead" if the numeric arguments are out of range.

• There are two flavors of streams, which can be called *internal* and *external* streams. Streams returned by file and stream are internal, because ML is the only process which can access them. Other streams, like the predefined terminal stream, are external because the ML system holds only one end of them, while the other end belongs to some external process. Some external streams may be read-only or write-only; the I/O operations escape when trying to read from a write-only only stream, or write to a read-only stream.

• The operation channel will be provided to open new external streams, for example to allow direct communication between two ML systems, or between ML and an external process. The way this will work is still to be defined.

All the above operations may escape for various I/O error. Multiplexed read and multiplexed write operations can be obtained by passing the same stream to several readers and writers respectively (i.e. to different parts of a program).

7. Modules

A *module* is a set of bindings (values, functions and types) which can be compiled and stored away, and later *imported* as a declaration wherever those bindings are needed. Modules are identified by *module names*. Module names can syntactically be represented as simple identifiers, or as strings containing Unix file paths (to access modules in directories other than the current one). Here is a module called Pair which defines a type and two functions.

```
module:     module Pair
            body
                abstype 'a pair == pair of 'a # 'a
                with  val newpair (a,b) == pair(a,b)
                      and left (pair(a,$))  ==  a
                      and right (pair($,b))  ==  b
                end
            end;
```

Module definitions can only appear at the top level, and they cannot rely on bindings defined in the surrounding scope (e.g. previous top level definitions). All the bindings used by a module must either be defined in the module itself, or imported from other modules.

The processing of a module definition is called a module *compilation*, whose only effect is to produce a separately compiled version of the module. Module definitions do not evaluate to values, and they do not introduce new bindings.

Modules can be defined interactively, but usually they are contained in external source files. In the latter case, a command 'use "Mod.ml";' (see section "Loading source files") can be used to compile a module definition contained in the file Mod.ml. Once a module is compiled, it can be imported:

```
declaration:    import Pair;

bindings:           module /usr/lib/ml/lib
                    abstype 'a pair
                    val newpair : ('b # 'b) -> ('b pair)
                    val left : 'c pair -> 'c
```

Exporting those binders without exporting B may potentially generate objects of an unknown (in the current scope) type t, on which no operations are available. Hence the following restriction applies: whenever a type identifier is involved in the exports of a module, its type definition must also be exported.

There is an alternative notation for modules which are imported and reexported:

```
module:     module A
            includes B C
            body
                ..
            end;

module:     module A
            body
                import B C;
                ..
            end;
```

The two forms above are equivalent; the first one is just an abbreviation for the second one. The includes keyword should be understood as the set-theoretical inclusion of the bindings of B and C; not as the inclusion of the source lines constituting the definition of B and C.

Every module automatically imports a standard library module called /usr/lib/ml/lib, which contains all the predefined ML types, functions and values (remember that a module cannot access any outside binding, except the ones explicitly imported; this also applies to the predefined ML identifiers). The library module is shown as a module /usr/lib/ml/lib binding, on import:

```
module:     module A
            body
                val a == 3
            end;

declaration: import A;

bindings:       module /usr/lib/ml/lib
                val a == 3 : int
```

The module /usr/lib/ml/lib binding is only an abbreviation for all the bindings defined in /usr/lib/ml/lib.ml: they are really imported and (re-)defined at this point.

Similarly, when importing a module B which exports a module A, the bindings of A are not explicitly listed. Instead a module A binding is presented as an abbreviation.

```
module:     module A
            body
                val a == 3
            end;

module:     module B
            includes A
            body
                val b == 5
            end;

declaration: import B;
```

```
bindings:       module /usr/lib/ml/lib
                module B
                    module /usr/lib/ml/lib
                    module A
                        module /usr/lib/ml/lib
                        type T
                        val absT : T -> int
                        val repT : int -> T
                        val a == (ref 3) : int ref
                    type T
                    val BabsT : T -> int
                    val BrepT : int -> T
                    val b == (ref 3) : int ref
                module C
                    module /usr/lib/ml/lib
                    module A
                        module /usr/lib/ml/lib
                        type T
                        val absT : T -> int
                        val repT : int -> T
                        val a == (ref 3) : int ref

                    type T
                    val CabsT : T -> int
                    val CrepT : int -> T
                    val c == (ref 3) : int ref
```

Note that A is imported by D through two different import paths.

Sharing of types means that the module A is not typechecked twice: if it were, than we would have two *different* abstract types called T and CrepT(BabsT 3), for example, would produce a type error.

Sharing of values means that the module A is not evaluated twice: hence b and c are the *same* reference variable, so that any side effect on b will be reflected on c, and vice versa.

Sharing of values also has the desirable effect that multiply imported functions (e.g. the library functions) are not replicated.

7.3. Module versions

The system automatically keeps track of module versions, to make sure, for example, that when a module is significantly modified, all the modules which depend on it are recompiled. The system however does not automatically recompile obsolete modules; it simply produces an error message and an exception. For example, when an obsolete module A is trying to import a new version of module B, we have:

```
declaration:    import A;

exception:          Module A must be compiled
                    Exception: link
```

This error message is generated in a variety of circumstances, but the effect is always the same: module A must be recompiled.

A new version of a module is generated whenever that module is recompiled and its external

```
Looking for :     A
I have found :    A
Clash between two different types having the same name: A
```

Equality has its own type errors:

```
- let val f a == a
    in f = f end;
-  Invalid type of args to "=" or "<>": 'a -> 'a
I cannot compare functions

- val f a == (a = a);
Invalid type of args to "=" or "<>": 'a
I cannot compare polymorphic objects

- let abstype A == int
      with   val absA n == A n
             and repA (A n) == n
  in absA 3 = absA 3 end;
Invalid type of args to "=" or "<>": A
I cannot compare abstract types
```

In modules, all variables must be either locally declared or imported from other modules: global variables cannot be used:

```
- val a == 3;
>  val a == 3 : int

- module A
  body val b == a end;
Modules cannot have global variables.
Unbound Identifier:  a
```

Modules which have never been compiled, or which are obsolete, produce a link exception:

```
- import A;

Module A must be compiled
Exception: link
```

8.5. Monitoring the compiler

The top level command monitor; gives access to a menu of commands for monitoring the internal functions of the ML compiler. In most cases these commands produce some check prints of internal data structures. Some options, like printing the parse trees, can be useful in becoming familiar with the system.

```
- monitor;
"y" for Yes; <CR> for No.
ParseTree?
Types?
TypeVariables?
Match?
```

The frequency of garbage collection depends on the amount of active data and follows a simple adaptive algorithm; when there is little active data garbage collection is frequent, when there is much active data garbage collection is less frequent.

Garbage collection is not interruptible: pressing the *DEL* key during collection has no effect until the end of collection.

Data structures are kept in different pages according to their format. Pages are linked into three lists: the 'other space' list, the 'active' list and the 'free' list. There are always as many pages in the 'other space' list as in the union of the 'active' and 'free' lists.

']'	Delim;	'^'	Symbol;	'_'	Letter;
'`'	Symbol;	'a'	Letter;	'b'	Letter;
'c'	Letter;	'd'	Letter;	'e'	Letter;
'f'	Letter;	'g'	Letter;	'h'	Letter;
'i'	Letter;	'j'	Letter;	'k'	Letter;
'l'	Letter;	'm'	Letter;	'n'	Letter;
'o'	Letter;	'p'	Letter;	'q'	Letter;
'r'	Letter;	's'	Letter;	't'	Letter;
'u'	Letter;	'v'	Letter;	'w'	Letter;
'x'	Letter;	'y'	Letter;	'z'	Letter;
'{'	Delim;	'|'	Symbol;	'}'	Delim;
'~'	Symbol;	DEL	Illegal;		

Appendix C. Predefined identifiers

true	Logic true	Nonfix constant
false	Logic false	Nonfix constant
not	Logic not	Nonfix
&	Logic and	Infix
or	Logic or	Infix
~	Complement	Nonfix
+	Plus	Infix
-	Difference	Infix
*	Times	Infix
div	Divide	Infix
mod	Modulo	Infix
=	Equal	Infix
<>	Different	Infix
>	Greater than	Infix
<	Less than	Infix
>=	Greater-eq	Infix
<=	Less-eq	Infix
size	String length	Nonfix
extract	Substring extraction	Nonfix
explode	String explosion	Nonfix
implode	String implosion	Nonfix
explodeascii	String to Ascii conv.	Nonfix
implodeascii	Ascii to string conv.	Nonfix
^	String concat.	Infix
intofstring	String to int conv.	Nonfix
stringofint	Int to string conv.	Nonfix
nil	Empty list	Nonfix constant
::	List cons	Infix constructor
hd	List head	Nonfix
tl	List tail	Nonfix
null	List null	Nonfix
length	List length	Nonfix
@	List append	Infix
map	List map	Nonfix
rev	List reverse	Nonfix
fold	List folding	Nonfix
revfold	List rev folding	Nonfix
ref	New reference	Nonfix constructor
!	Dereferencing	Nonfix
:=	Assignment	Infix
array	New array	Nonfix
arrayoflist	List to array	Nonfix
lowerbound	Array lower bound	Nonfix
arraysize	Array size	Nonfix
sub	Array indexing	Infix
update	Array update	Infix
arraytolist	Array to list	Nonfix
o	Function comp	Infix
file	Stream from file	Nonfix
save	Stream to file	Nonfix
stream	Empty stream	Nonfix
channel	External channel	Nonfix

Appendix D. Predefined type identifiers

unit	Unit Type	Nonfix
bool	Boolean Type	Nonfix
int	Integer Type	Nonfix
string	String Type	Nonfix
#	Cartesian Product	N-ary infix
->	Function Space	Infix
list	List Type	Suffix
ref	Reference Type	Suffix
array	Array Type	Suffix

Appendix F. Metasyntax

Strings between quotes '"' are terminals.

Identifiers are non-terminals.

Juxtaposition is syntactic concatenation.

'|' is syntactic alternative.

'[]' is the empty string.

'[..]' is zero or one times (i.e. optionally) ' .. '.

'{ .. }n' is n or more times ' .. ' (default n=0).

'{ .. / -- }n' means n (default 0) or more times '..' separated by '--'.

Parentheses '(..)' are used for precedence.

Appendix H. Syntax

Syntactic alternatives are in order of decreasing precedence.

```
Phrase ::=
  [Exp | SimpleDecl | Module | Use | Monitor] ";".

Module ::=
  "module" ModuleName ["includes" {ModuleName}1] "body" Decl "end".

Use ::=
  "use" String.

Monitor ::=
  "monitor".

SimpleExp ::=
  ["op"] (Ide | Con) |
  "[" {Exp / ";"} "]" |
  "(" {Exp / ";"}1 ")".

Exp ::=
  SimpleExp |
  Exp  SimpleExp |
  Exp ":" Type |
  Exp Ide Exp |
  {Exp / ","}2 |
  "escape" Exp |
  "if" Exp "then" Exp "else" Exp |
  "while" Exp "do" Exp |
  "let" Decl "in" Exp "end" |
  Exp "where" Decl "end" |
  "case" Exp "of" Match |
  Exp "?" Exp |
  Exp "??" Exp Exp |
  Exp "?\" Ide "." Exp.
  "fun" Match.

SimpleDecl ::=
  "val" ValBind |
  "type" TypeBind |
  "abstype" TypeBind "with" Decl "end" |
  "export" ExportList "from" Decl "end" |
  "import" {ModuleName}1 |
  "infix" {Ide}1 | "nonfix {Ide}1.

Decl ::=
  SimpleDecl |
  Decl ";" Decl |
  "(" Decl ")".

ExportList ::=
  {"type" {Ide / ","}1 | "val" {Ide / ","}1}.

ValBind ::=
```

Appendix I. Escape sequences for strings

The escape character for strings is \; it introduces characters according to the following code:

\1 .. \9	One to nine spaces
\0	Ten spaces
\R	Carriage return
\L	Line feed
\T	Horizontal tabulation
\B	Backspace
\E	Escape
\N	Null (Ascii 0)
\D	Del (Ascii 127)
\^c	Ascii control char ^c (c any appropriate char)
\c	c (for any other char c different from #)
\#s	128 + s (for any of the previous sequences s)

Strings are printed at the top level in the same form in which they are read; that is surrounded by quotes, with all the \ and with no non-printable characters. Output operations instead print strings in their crude form.

Appendix K. Introducing new primitive operators

This appendix describes how to add to the ML system a new function "foo: t" that cannot be otherwise defined in ML. It has to be defined as a new primitive operation in the Functional Abstract Machine [Cardelli 83] and implemented in C, Pascal, Assembler, or any other language which respects VAX procedure call standards.

This procedure is complex, dangerous and not advisable. Before attempting it, one should consider implementing "foo" as an external program activated through the channel primitive as a subprocess.

- File mlglob.h
  - Add "OpFoo" to enumeration type "SMOperationT" IN ASCII ALPHABETIC ORDER!.
  - Add "OpFoo: ();" to the case list of type "SMCodeT".
  - Add "AtomFoo: AtomRefT" in section "{Parser Vars}".
  - If foo is prefix (infix, etc.) add "SynOpFoo" to the enumeration type "SynPrefixOpClassT" ("SynInfixOpClassT", etc.).
- File mlscan.p
  - Add "AtomFoo := TableInsertAtom(3,'foo');" in procedure "SetupTable" (where "3" is the length of "foo").
  - Add "PushSynRoleIde(AtomFoo);" in the same procedure (assuming that foo is nonfix, else use PushSynRoleInfix, etc. as appropriate, with second argument "SynOpFoo").
- File mlanal.p
  - Add "Predefined(AtomFoo,OpFoo,n,t);" to the procedure "SetupEnvironment", where foo: t (you can understand how to express t from the code of SetupEnvironment), and n is the arity of foo (i.e. the number of arguments it expects on the stack).
- File mldebg.p
  - Add "SetupMon(EmitSimpleOp(OpFoo));" to the procedure "SetupEnvValues" (if foo is not monadic, use SetupBin etc. Monadic, diadic, etc means that foo takes 1,2, etc. arguments on the top of the stack and returns a result there, after popping the arguments). It is essential that the order of setups in this procedure matches the order of definitions in SetupEnvironment (in mlanal.p).
- File mlconv.p
  - Add "OpFoo: Operand:=0{nil};" to the case list in function "ConvertOperand".
- File amglob.h
  - Add "#define OpFoo n" IN ASCII ALPHABETIC ORDER in section "Fam OpCodes"; make sure that all the opcodes are SEQUENTIALLY NUMBERED.
  - Add "extern Address DoFoo();" in section "Externals".
  - If foo may escape with string "foo", declare "extern Address *StrFoo;" in section "FailStrings".
- File amevalop.c
  - Add "OpFoo: CallOp((Address)DoFoo,n); break;" to the case list in procedure "Assem", where n is the arity of Foo (i.e. how many arguments it expects on the stack).
- File ammall.c
  - If foo may escape with string "foo", declare "Address *StrFoo;" and add "StrFoo = PushGCBox(StringFromC("foo"));" to the procedure "AllocFailStrings".
- File amdebg.c
  - Add "case OpFoo: printf("Foo"); break;" to the case list of procedure "AMStatPrint".
- File amfooop.c

Appendix L. System limitations

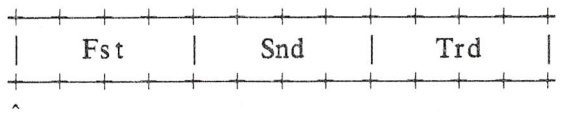
There is a limit on the size of any single ML object, due to storage allocation problems. This limit is determined by the constant "PageSize" in the file "amglob.h". All the limitations mentioned on this section are further constrained by this restriction.

The semantic limit on the length of strings is 64K chars.

The syntactic limit on the length of string quotations in a source program is also 64K chars.

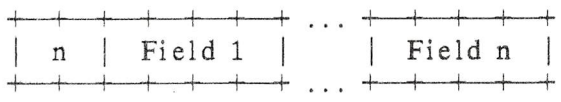
Several memory areas inside the system have fixed size. When one of these is exceeded, compilation fails and a message is printed. The only way to fix this is to search for the point in the source program where the error message is generated, increase the corresponding area (usually by redefining a constant) and recompile the system.

Medium Record



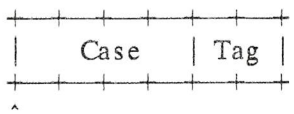
_ Record

0 (nullrecord) (unboxed)

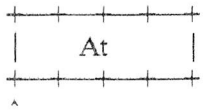


(n > 0)

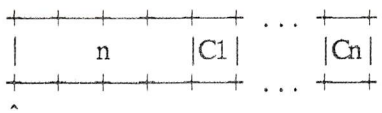
Variant



Reference



String



(n ≥ 0)

Appendix N. Ascii codes

0 nul	1 soh	2 stx	3 etx	4 eot	5 enq	6 ack	7 bel
8 bs	9 ht	10 nl	11 vt	12 np	13 cr	14 so	15 si
16 dle	17 dc1	18 dc2	19 dc3	20 dc4	21 nak	22 syn	23 etb
24 can	25 em	26 sub	27 esc	28 fs	29 gs	30 rs	31 us
32 sp	33 !	34 "	35 #	36 $	37 %	38 &	39 '
40 (	41)	42 *	43 +	44 ,	45 −	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 '	97 a	98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 del

Deep recursions producing stack overflow will badly corrupt the system; (this is the most likely cause of crash). The ml exception "corruption", or the message System Crash! may be generated. If you come across this problem, try to rewrite your programs in a linear recursive style, so that the compiler can optimize them to iterations, or directly in iterative style using while-do. This problem cannot be solved easily, given the current Unix memory management primitives.

The copying garbage collection algorithm is recursive. Because of this, the system can crash during garbage collection when (1) very deep (thousands of levels) lists or other structures are present in memory and (2) not enough virtual memory can be obtained for the system stack, e.g. because of unix limits on the size of page tables, or for lack of swap space on disk.

ML Syntax

```
Phrase ::=
  [Exp | SimpleDecl | Module |
   "use" String | "monitor"] ";".

Module ::=
  "module" ModuleName
  ["includes" {ModuleName}1]
  "body" Decl "end".

SimpleExp ::=
  ["op"] (Ide | Con) |
  "[" {Exp / ";"} "]" |
  "(" {Exp / ";"}1 ")".

Exp ::=
  SimpleExp |
  Exp SimpleExp |
  Exp ":" Type |
  Exp Ide Exp |
  {Exp / ","}2 |
  "escape" Exp |
  "if" Exp "then" Exp "else" Exp |
  "while" Exp "do" Exp |
  "let" Decl "in" Exp "end" |
  Exp "where" Decl "end" |
  "case" Exp "of" Match |
  Exp "?" Exp |
  Exp "??" Exp Exp |
  Exp "?\" Ide "." Exp.
  "fun" Match.

SimpleDecl ::=
  "val" ValBind |
  "type" TypeBind |
  "abstype" TypeBind "with" Decl "end" |
  "export" ExportList "from" Decl "end" |
  "import" {ModuleName}1 |
  "infix" {Ide}1 | "nonfix" {Ide}1.

Decl ::=
  SimpleDecl |
  Decl ";" Decl |
  "(" Decl ")".

ExportList ::=
  {("abstype" | "type" | "val") {Ide / ","}1}1.

ValBind ::=
  Pat "==" Exp |
  ["op"] Ide {SimplePat}1 [":" Type] "==" Exp |
  {["op"] Ide SimplePat [":" Type] "==" Exp / "|"}2 |
  ValBind "and" ValBind |
  "rec" ValBind.

TypeBind ::=
  [Params] TypeIde "==" {Ide ["of" Type] / "|"}1 |
  TypeBind "and" TypeBind |
  "rec" TypeBind.

Params ::=
  TypeVar | "("{TypeVar / ","}1 ")".
```

Syntactic alternatives and infix operators are listed in order of decreasing precedence.

```
SimplePat ::=
  "$" |
  ["op"] Ide |
  Con |
  "[" {Pat / ";"} "]" |
  "(" Pat ")".

Pat ::=
  SimplePat |
  Con SimplePat |
  Pat ":" Type |
  SimplePat Con SimplePat |
  {Pat / ","}2 |

Match ::= {Pat "." Exp / "|"}1

Type ::=
  TypeVar |
  [TypeArgs] TypeIde |
  {Type / "#"}2 |
  Type "->" Type |
  "(" Type ")".

TypeArgs ::= Type | "(" {Type / ","}1 ")".

Letter ::= "a" | .. | "z" | "A" | .. | "Z" | "_".
Digit ::= "0" | .. | "9".
Symbol ::= !#%&*+-/:<=>?@\^`|~.
Character ::= ht | .. | cf | " " | .. | "~".
Ide ::= Letter {Letter | Digit | "'"} | {Symbol}1.
Integer ::= {Digit}1.
String ::= """ {Character} """.
Con ::= "(" ")" | Integer | String | Ide.
TypeIde ::= Ide.
TypeVar ::= "'" Ide.
ModuleName ::= Ide | String.
```

Keywords

abstype and body case do else end export escape from fun import if in includes infix let local module nonfix of op rec then type use val where while with : ? ?? ?\ ==

Operations

{application}L {sub}L {* div mod}L {+ - ^}L {:: @}R {= <> > < >= <=}R {&}R {or o}R {user-infix}L {update :=}R true false not ~ size extract explode implode explodeascii implodeascii intofstring stringofint nil hd tl null length map rev fold revfold ref ! array arrayoflist lowerbound arraysize sub update arraytolist file save stream channel input output lookahead caninput terminal user-nonfix

Constructors

integers strings tuples () true false nil :: ref user-defined

Metasyntax

Strings between quotes ' " ' are terminals.
Identifiers are non-terminals.
Juxtaposition is syntactic concatenation.
' | ' is syntactic alternative.
'[..]' is zero or one times (i.e. optionally) ' .. '.
'{ .. }n' is n (default 0) or more times ' .. ' .
'{ .. / -- }n' is n or more times ' .. ' separated by ' -- '.
Parentheses '(..)' are used for precedence.