@Title{ML}

@Section{Introduction and examples}

ML is an interactive language. The system repeatedly prompts for input and reports the results of computations; this interaction is said to happen at the @Italic{top level} of evaluation. At the top level one can evaluate expressions or perform declarations.

To give a first impression of the system, we reproduce below a session at a terminal in which simple uses of various ML constructs are illustrated. To make the session easier to follow, it is split into a sequence of sub-sessions displayed in boldface. Each sub-session is accompanied by an explanation; the complete session consists of the concatenation of the boldface areas. A complete description of the syntax of ML is given in ---, and of the semantics in ---.

@SubSection{Expressions}

The ML prompt is '- ', and so lines beginning with this contain the user's contribution; all other lines are output by the system.

```
@Verbatim{

       - 2+3;
         5 : int

       - it;
         5 : int

}
```

ML prompted with '- '; the user then typed '2+3;' followed by a return; ML then responded with ' 5 : int', a new line, and then prompted again. The user then typed 'it;' followed by a return, and the system responded by typing ' 5 : int' again.

In general to evaluate an expression e one types 'e;' followed by a return; the system then prints e's value and type. The value of the last expression evaluated at top level is remembered in the identifier 'it'.

@SubSection{Declarations}

The declaration 'let x = e' evaluates e and binds the resulting value to x.

```
@Verbatim{

       - let x = 2@*{}3;
       > x = 6 : int

       - it = x;
         false : bool

}
```

The prefix '> ' indicates that a new declaration is taking place, as opposed to an evaluation which is prefixed by ' '. Notice that declarations do not effect 'it'. To bind x@Sub{1},...,x@Sub{n} simultaneously to the values of e@Sub{1},...,e@Sub{n} one can perform either the declaration

'let x@Sub{1}=e@Sub{1} and x@Sub{2}=e@Sub{2} ... and x@Sub{n}=e@Sub{n}'
or, equivalently, the declaration
'let x@Sub{1},...,x@Sub{n} = e@Sub{1},...,e@Sub{n}'.
In the first case we use a @Italic{environment operator}
(or @Italic{declaration operator@Roman{)}}
'and', while in the second case we use a @Italic{structured variable}
(or @Italic{varstruct@Roman{)}} 'x@Sub{1},...,x@Sub{n}'.

@Verbatim{

```
- let y = 10 and z = x;
> y = 10 : int
| z = 6 : int

- let x,y = y,x;
> x = 10 : int
| y = 6 : int
```

}

Note that the declaration prefix '> ' converts to '| ' after the first
definition.

Cascaded declarations are obtained by the environment operator 'enc'
(enclose) which makes
earlier declarations available in later declarations, and has otherwise the
same effect as 'and'.

@Verbatim{

```
- let x = 10 enc y = x+5;
> x = 10 : int
| y = 15 : int

- let x = 5 and y = x+5;
> x = 5 : int
| y = 15 : int
```

}

Private declarations are obtained by the environment operator 'ins'
(inside) which makes
a declaration available inside another declaration, but not anywhere else.

@Verbatim{

```
- let x = 7 ins y = x+5;
> y = 12 : int

- x;
  5 : int
```

}

Complex declarations can be bracketed by '!{' and '!}'; otherwise
the 'and' operator binds stronger than 'enc' and 'ins' (which have the
same binding power) and all three operators are right associative.
In the following example declaration brackets make a difference.

@Verbatim{

```
- let x = 10 and !{z = 5 ins y = 10+z!};
> x = 10 : int
| y = 15 : int
```

```
}

A declaration d can be made local to the evaluation of an expression e
by evaluating 'let d in e'.
The expression 'e where d' is equivalent to 'let d in e'.

@Verbatim{

       - let x = 2 in x@*{}y;
         30 : int

       - x;
         10 : int

       - x@*{}y where x = 2;
         30 : int

}

@SubSection{Functions}

To define a function f with formal parameter x and body e one performs
the declaration: 'let f x = e'.
To apply f to an actual parameter e one evaluates the expression: 'f e'.

@Verbatim{

       - let f x = 2@*{}x;
       > f = \ : int -> int

       - f 4;
         8 : int

}

Functions are printed as a '\' followed by their type ('\' is chosen as
an ascii approximation of the greek letter lambda). Application
binds tighter than anything else in the language; thus, for example,
'f 3 + 4' means '(f 3)+4' not 'f(3+4)'.
Functions of several arguments can be defined:

@Verbatim{

       - let add x y = x+y;
       > add = \ : int -> (int -> int)

       - add 3 4;
         7 : int

       - let f = add 3;
       > f = \ : int -> int

       - f 4;
         7 : int

}

Application associates to the left so 'add 3 4' means '(add 3)4'.
In the expression 'add 3', add is partially applied to 3; the
resulting value is a function - the function of type
'int -> int' which adds 3 to its argument.
Thus add takes its arguments one at a time; we could have
made add take a single argument of the Cartesian product type
'int # int':

@Verbatim{
```

```
        - let add(x,y) = x+y;
        > add = \ : (int # int) -> int

        - add(3,4);
          7 : int

        - let z = (3,4) in add z;
          7 : int

        - add 3;
        Type Clash  in:  (add 3)
        Looking  for:    int # int
        I have found:    int

}
```

As well as taking structured arguments (e.g. '(3,4)') functions may also return structured results.

```
@Verbatim{

        - let sumdiff(x,y) = (x+y,x-y);
        > sumdiff = \ : (int # int) -> (int # int)

        - sumdiff(3,4);
          (7,~1) : int # int

}
```

Incidentally, note that the unary negation operation on numbers is '~' instead of '-'; hence one should write '~3' for negative numbers and '~(n-1)' for the complement of n-1.

@SubSection{Recursion}

The following is an attempt to define the factorial function:

```
@Verbatim{

        - let fact n = if n=0 then 1 else n@*{}fact(n-1);
        Unbound Identifier: fact

}
```

The problem is that any free variables in the body of a function have the bindings they had just before the function was declared; 'fact' is such a free variable in the body of the declaration above, and since it isn't defined before its own declaration, an error results. To make things clear consider:

```
@Verbatim{

        - let f n = n+1;
        > f = \ : int -> int

        - let f n = if n=0 then 1 else n@*{}f(n-1);
        > f = \ : int -> int

        - f 3;
          9 : int

}
```

Here 'f 3' results in the evaluation of '3@*{}f(2)', but now the first f is used so 'f(2)' evaluates to 2+1=3, hence the expression 'f 3' results in 3@*{}3=9. To make a function declaration hold within its own body 'let rec' instead of 'let' must be used. The correct recursive definition of the factorial function is thus:

@Verbatim{

```
        - let rec fact n = if n=0 then 1 else n@*{}fact(n-1);
        > fact = \ : int -> int

        - fact 3;
          6 : int
```

}

'rec' is another environment operator like 'and', 'enc' and 'ins'; it can be nested inside complex declarations, and it binds more weakly than 'and' but more strongly that 'enc' and 'ins'.

@SubSection{Assignment and sequencing}

Assignment operations act on @Italic{reference} objects. A reference is an updateble pointer to an object. References are 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 the trivial value '()' (called @Italic{triv@Roman{),}} which is the only object of the trivial type '.' (also called @Italic{triv@Roman{).}}

@Verbatim{

```
         - let a = ref 3;
         > a = (ref 3) : int ref

         - a:=5;
           () : .

         - !!a;
           5 : int
```

}

When several side-effecting operations have to be executed in sequence, it is useful to use @Italic{sequencing} '(e@Sub{1}; ... ;e@Sub{n})' (the parenthesis are needed), which evaluates e@Sub{1} ... e@Sub{n} in turn and returns the value of e@Sub{n}.

@Verbatim{

```
         - (a:=!!a+1; a:=!!a/2; !!a);
           3 : int
```

}

@SubSection{Iteration}

The construct 'if e@Sub{1} then e@Sub{2} elseloop e@Sub{3}' is the same as 'if e@Sub{1} then e@Sub{2} else e@Sub{3}' in the true case; when e@Sub{1} evaluates to false, e@Sub{3} is evaluated and control loops back to the front of the construct

again.
As an illustration here is an iterative definition
of 'fact' which uses two local assignable variables:
'count' and 'result' (note that the prompt '-' changes to '=' when an
expression
spans several lines).

@Verbatim{

```
        - let fact n =
        =     let count = ref n and result = ref 1
        =     in      if !!count=0
        =             then !!result
        =             elseloop (result := !!count @*{} !!result;
        =                       count := !!count-1);
        > fact = \ : int -> int

        - fact 4;
          24 : int
```

}

The 'then' in 'if e1 then e2 else e3' may be replaced by 'thenloop'
to cause iteration when e1 evaluates to true. Thus 'if e1 thenloop e2 else e3'
is equivalent to 'if not(e1) then e3 elseloop e2'.
The conditional/loop construct can have a number of conditions,
each preceded by 'if'; the expression guarded by
each condition may be preceded by 'then', or by 'thenloop' when
the whole construct is to be reevaluated after evaluating the
guarded expression:

@Verbatim{

```
        - let gcd(x,y) =
        =     let x,y = ref x, ref y
        =     in      if !!x>!!y thenloop x:=!!x-!!y
        =             if !!x<!!y thenloop y:=!!y-!!x
        =             else !!x;
        > gcd = \ : (int # int) -> int

        - gcd(12,20);
          4 : int
```

}

The 'else' branch must always be present in normal conditionals and in
the iterative forms.