@SubSection{Lists}

If e1,...,en all have type ty then the ML expression '[e1;...;en]' has type 'ty list'. The standard functions on lists are 'hd' (head), 'tl' (tail), 'null' (which tests whether a list is empty - i.e. is equal to '[]' (nil)), and the infixed operators '::' (cons) and '!@' (append, or concatenation).

@Verbatim{

```
      - let m = [1;2;(2+1);4];
      > m = [1;2;3;4] : int list

      - hd m, tl m;
        (1,[2;3;4]) : int # (int list)

      - null m, null [];
        (false,true) : bool # bool

      - 0::m;
        [0;1;2;3;4] : int list

      - [1;2] !@ [3;4;5;6];
        [1;2;3;4;5;6] : int list

      - [1;true;2];
      Type Clash  in:  [1;true;2]
      Looking  for:    int
      I have found:    bool
```

}

All the members of a list must have the same type (although this type could be a sum, or disjoint union, type - see 2.4).

@SubSection{Tokens}

A sequence of characters in token quotes (`) is a token.

@Verbatim{

```
       - `this is a token`;
         `this is a token` : tok

       - "this is a token list";
         "this is a token list" : tok list

       - it = ("this is a" !@ [`token`;`list`]);
         true : bool
```

}

The expression "tok1 tok2 ... tokn" is an alternative syntax for [`tok1`; `tok2`; ... ;`tokn`].

@SubSection{Polymorphism}

The list processing functions 'hd', 'tl' etc can be used on all types of lists.

@Verbatim{

```
       - hd [1;2;3];
         1 : int

       - hd [true;false;true];
```

```
          true : bool

        - hd "this is a token list";
          `this` : tok

}

Thus 'hd' has more than one type,
for example above it is used with types '(int list) -> int',
'(bool list) -> bool' and '(tok list) -> tok'.
In fact if ty is @Italic{any} type then 'hd' has the type '(ty list) -> ty'.
Functions, like 'hd', with
many types are called @Italic{polymorphic},
and ML uses
type variables '@*{}a', '@*{}b', '@*{}1', '@*{}2', '@*{}', '@*{}@*{}',
'@*{}@*{}@*{}'  etc
to represent their types.

@Verbatim{

        - hd;
          \ : (@*{}a list) -> @*{}a

        - let rec map f l =
        =    if null l then []
        =              else f(hd l)::map f (tl l);
        > map = \ : (@*{}a -> @*{}b) -> ((@*{}a list) -> (@*{}b list))

        - map fact [1;2;3;4];
          [1; 2; 6; 24] : int list

}

map takes a function f (with argument type @*{}a and result type @*{}b),
and a list l (of elements of type @*{}a), and returns the list obtained
by applying f to each element of l (which is a list of elements
of type @*{}b). map can be used at any instance of its type:
above, both @*{}a and @*{}b were
instantiated to int; below, @*{}a is instantiated to (int list) and @*{}b
to bool. Notice that the instance need not be specified;
it is determined by the typechecker.

@Verbatim{

        - map null [[1;2]; []; [3]; []];
          [false; true; false; true] : bool list

}

@SubSection{Lambda-expressions}

The expression '\x.e' evaluates to a function with
formal parameter x and body e. Thus 'let f x = e' is equivalent
to 'let f = \x.e'. Similarly 'let f(x,y)z = e' is equivalent
to 'let f = \(x,y).\z.e'.
Repeated '\'`s, as in '\(x,y).\z.e', may be abbreviated by
'\(x,y)z.e'.
The character '\' is our representation of
lambda, and expressions like '\x.e' and '\(x,y)z.e' are
called lambda-expressions.

@Verbatim{

        - \x.x+1;
          \ : int -> int
```

```
       - it 3;
         4 : int

       - map (\x.x@*{}x) [1;2;3;4];
         [1;4;9;16] : int list

       - let doubleup = map (\x.x!@x);
       > doubleup = \ : ((@*{}a list) list) -> ((@*{}a list) list)

       - doubleup ["a b";"c"];
         ["a b a b";"c c"] : (tok list) list

       - doubleup [[1;2];[3;4;5]];
         [[1;2;1;2];[3;4;5;3;4;5]] : (int list) list

}

@SubSection{Failure}

Some standard functions @Italic{fail} at run-time on certain arguments,
yielding a token (which is usually the function name) to identify
the sort of failure. A failure with token `t` may also be generated
explicitly by evaluating the expression 'failwith `t`' (or
more generally 'failwith e' where e has type tok).

@Verbatim{

       - hd(tl[2]);
       Failure: hd

       - 1/0;
       Failure: /

       - (1/0)+1000;
       Failure: /

       - failwith (hd "this is a token list");
       Failure: this

}

A failure can be trapped by '?'. The value of the expression
'e1?e2' is that of e1, unless e1 causes a failure, in which case it is
the value of e2.

@Verbatim{

       - hd(tl[2]) ? 0;
         0 : int

       - (1/0)?1000;
         1000 : int

       - let half n =
       =     if n=0 then failwith `zero`
       =            else let m=n/2
       =                 in  if n=2@*{}m then m else failwith`odd`;
       > half = \ : int -> int

}

The function half only succeeds on non-zero even numbers; on
0 it fails with `zero`, and on odd numbers it fails with `odd`.

@Verbatim{
       - half 4;
         2 : int

       - half 0;
       Failure: zero

       - half 3;
       Failure: odd

       - half 3 ? 1000;
         1000 : int

}
```

Failures may be trapped selectively (on token) by '??'; if e1 fails with token '`t`', then the value of 'e1 ??"t1 ... tn" e2' is the value of e2 if t is one of t1,...,tn, otherwise the expression still fails with '`t`'.

```
@Verbatim{

       - half 0 ?? "zero plonk" 1000;
         1000 : int

       - half 1 ?? "zero plonk" 1000;
       Failure: odd

}
```

One may add several '??' traps to an expression, and one may add a '?' trap at the end as a catchall.

```
@Verbatim{

       - half 1
       =   ?? "zero" 1000
       =   ?? "odd"  2000;
         2000 : int

       - hd(tl[half(4)])
       =   ?? "zero" 1000
       =   ?? "odd"  2000
       =   ? 3000;
         3000 : int

}
```