Appendix B

Introduction to NUPRL ML

Whenever NUPRL is fired up, several ML *top loops* are created. Running in these windows is an ML interpreter that is embedded into the library, editor, or refiner process. Whenever one has the `ML>` prompt one can type an ML expression, terminate it with `;;` and press ↵. ML will evaluate the expression, and print its value and type.[1]

One's primary interaction with NUPRL 5 is through the navigator and the windows opened by it. However, advanced users will find it necessary to interact with the NUPRL 5 processes for examining parts of NUPRL objects or customizing the behavior of NUPRL 5. In particular, all NUPRL tactics are written in ML, as are a variety of utility functions. The tactics are documented in Chapter 8. The utility functions are described throughout this Chapter.

B.1 The History of ML

Several versions of the programming language ML have appeared over the years, between the time it was first designed and implemented by Milner, Morris and Wadsworth at the University of Edinburgh in the early 1970's, and the time it was settled and standardized in the mid-1980's. The original ML, the *meta-language* of the Edinburgh LCF system, is defined in [GMW79].

The ML used in the NUPRL system is fairly close to the original. It is derived from a early version that Huet at INRIA and Paulson at the University of Cambridge were working on in 1981. Todd Knoblock at Cornell made most of the NUPRL specific modifications in the mid-1980's. NUPRL's ML hasn't changed since then and is not compatible with the ML versions that are widely used today.

The ML of Huet and Paulson is described in the preface to '*The* ML *Handbook*' [CHP84]. Huet used this version in the Formel project; and it subsequently evolved into a version of ML called CAML. Paulson also used it, as part of the first version of Cambridge LCF, but switched to Standard ML in the later versions of Cambridge LCF [Pau87].

The CAML language [CH90, WAL$^+$90] is now rarely used. But there is a scaled down version called CAML-Light which is actively used in teaching programming to over 10,000 engineers a year in France. Its object-oriented version OCaml [Ler00] have become quite popular in recent years and has been used in the implementation of the group communication toolkit Ensemble [Hay98, BCH$^+$00]. The Standard ML language has also become increasingly popular for implementing theorem provers such as HOL [GM93] or Isabelle [Pau90].

[1]Note that the ML prompt is different in each window. It is `ML[(ORB)]>` in the NUPRL process windows for the library, editor, or refiner and may later change into `ML[(lib)]>`, `ML[(edd)]>`, and `ML[(ref)]>`. In the NUPRL 5 top loop it is `ML[EDD]>`, `ML[LIB]>`, or `ML[REF]>`. The latter already provide the double semicolon for terminating an ML expression, so the user does not have to enter `;;` in these windows.

The description of ML that appears in Sections B.3 to B.6 is based very closely on 'The ML Handbook' [CHP84]. It was adapted for NUPRL purposes from LaTeX sources provided by the HOL theorem proving group in Cambridge. For completeness (and historical interest), the preface to '*The* ML *Handbook*' and the preface to '*Edinburgh LCF: a Mechanised Logic of Computation*' are reproduced below.

B.1.1 Preface to 'The ML Handbook'

This handbook is a revised edition of Section 2 of 'Edinburgh LCF', by M. Gordon, R. Milner, and C. Wadsworth, published in 1979 as Springer Verlag Lecture Notes in Computer Science n$^{\underline{o}}$ 78. ML was originally the meta-language of the LCF system. The ML system was adapted to Maclisp on Multics by Gérard Huet at INRIA in 1981, and a compiler was added. Larry Paulson from the University of Cambridge completely redesigned the LCF proving system, which stabilized in 1984 as Cambridge LCF. Guy Cousineau from the University Paris VII added concrete types in the summer of 1984. Philippe Le Chenadec from INRIA implemented an interface with the Yacc parser generator system, for the versions of ML running under Unix. This permits the user to associate a concrete syntax with a concrete type.

The ML language is still under design. An extended language was implemented on the VAX by Luca Cardelli in 1981. It was then decided to completely re-design the language, in order to accommodate in particular the call by pattern feature of the language HOPE designed by Rod Burstall and David MacQueen. A committee of researchers from the Universities of Edinburgh and Cambridge, the Bell Laboratories and INRIA, headed by Robin Milner, is currently working on the new extended language, called Standard ML. Progress reports appear in the Polymorphism Newsletter, edited by Luca Cardelli and David MacQueen from Bell Laboratories. The design of a core language is now frozen, and its description will appear in a forthcoming report of the University of Edinburgh, as 'The Standard ML Core Language' by Robin Milner.

This handbook is a manual for ML version 6.1, released in December 1984. The language is somewhere in between the original ML from LCF and standard ML, since Guy Cousineau added the constructors and call by patterns. This is a LISP based implementation, compatible for Maclisp on Multics, Franzlisp on VAX under Unix, Zetalisp on Symbolics 3600, and Le_Lisp on 68000, VAX, Multics, Perkin-Elmer, etc... Video interfaces have been implemented by Philippe Le Chenadec on Multics, and by Maurice Migeon on Symbolics 3600. The ML system is maintained and distributed jointly by INRIA and the University of Cambridge.

B.1.2 Preface to 'Edinburgh LCF'

ML is a general purpose programming language. It is derived in different aspects from ISWIM, POP2 and GEDANKEN, and contains perhaps two new features. First, it has an escape and escape trapping mechanism, well-adapted to programming strategies which may be (in fact usually are) inapplicable to certain goals. Second, it has a polymorphic type discipline which combines the flexibility of programming in a typeless language with the security of compile-time type checking (as in other languages, you may also define your own types, which may be abstract and/or recursive).

For those primarily interested in the design of programming languages, a few remarks here may be helpful both about ML as a candidate for comparison with other recently designed languages, and about the description of ML which we provide. On the first point, although we did not set out with programming language design as a primary aim, we believe that ML does contain features worthy of serious consideration; these are the escape mechanism and the polymorphic type discipline mentioned above, and also the attempt to make programming with functions—including those of higher type—as easy and natural as possible. We are less happy about the imperative aspects of the language, and would wish to give them further thought if we were mainly concerned with language design. In particular, the constructs for controlling iteration both by boolean conditions and by escape-trapping (which we included partly for experiment) are perhaps too complex taken together, and we are sensitive to the criticism that escape (or failure, as we call it) reports information only in the form of a string. This latter constraint results mainly from our type discipline; we do not know how best to relax the constraint while maintaining the discipline.

Concerning the description of ML, we have tried both to initiate users by examples of programming and to give a precise definition.

B.2 Introduction and Examples

ML is an interactive language. At top-level one can:

- evaluate expressions
- 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. A complete description of the syntax and semantics of ML is given in Section B.3 and Section B.4 respectively.

B.2.1 Expressions

In this tutorial, the ML prompt is `#` so lines beginning with this contain the user's contribution; all other lines are output by the system. The NUPRL ML prompt is different; usually `ML>` is used for the first line of user input, and `>` is used for continuation lines.

1
```
# 2+3;;
5 : int

# it;;
5 : int
```

ML prompted with `#`, the user then typed `⌞2+3;;⌟` followed by a carriage return ↵; ML then responded with `⌞5 : int⌟`, a new line, and then prompted again. The user then typed `⌞it;;↵⌟` and the system responded by typing `⌞5 : int⌟` again. In general to evaluate an expression e one types e followed by a carriage return; the system then prints e's value and type (the type prefaced by a colon). The *value of the last expression* evaluated at top level is remembered in the identifier `it`.

B.2.2 Declarations

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

2
```
# let x=2*3;;
x = 6 : int

# it=x;;
false : bool
```

Notice that declarations do not affect the identifier `it`. To bind the variables $x_1, \ldots, x_n$ simultaneously to the values of the expressions $e_1, \ldots, e_n$ one can perform either the declaration `let` x_1`=`e_1 `and` x_2`=`e_2 `...and` x_n`=`e_n or `let` x_1`,`x_2`,...,`x_n `=` e_1`,`e_2`...,`e_n. These two declarations are equivalent.

```
# let y=10 and z=x;;
y = 10 : int
z = 6 : int

# let x,y = y,x;;
x = 10 : int
y = 6 : int
```
3

A declaration d can be made *local* to the evaluation of an expression e by evaluating the expression d `in` e. The expression e `where` b (where b is a *binding* such as `x=2`) is equivalent to `let` b `in` e.

```
# let x=2 in x*y;;
12 : int

# x;;
10 : int

# x*y where x=2;;
12 : int
```
4

B.2.3 Assignment

Identifiers can be declared *assignable* using `letref` instead of `let`. Values bound to such identifiers can be changed with the assignment expression x`:=`e, which changes the value bound to x to be the value of e. Attempts to assign to non-assignable variables are detected by the type checker.

```
# x:=1;;

unbound or non-assignable variable x
1 error in typing
typecheck failed

# letref x=1 and y=2;;
x = 1 : int
y = 2 : int

# x:=6;;
6 : int

# x;;
6 : int
```
5

The value of an assignment x`:=`e is the value of e (hence the value of `y:=6` is `6`). Simultaneous assignments can also be done:

```
# x,y := y,x;;
(2,6) : (int # int)

# x,y;;
(2,6) : (int # int)
```
6

The type `(int # int)` is the type of pairs of integers.

B.2.4 Functions

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

```
# let f x = 2*x;;                                                    7
f = - : (int -> int)

# f 4;;
8 : int
```

Functions are printed as a dash, `-`, followed by their type, since a function as such is not printable. Application binds more tightly 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:

```
# let add x y = x+y;;                                                8
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`, the function `add` is partially applied to `3`; the resulting value is 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)`:

```
# let add(x,y) = x+y;;                                               9
add = - : ((int #  int) -> int)

# add(3,4);;
7 : int

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

# add 3;;

ill-typed phrase: 3
has an instance of type  int
which should match type  (int #  int)
1 error in typing
typecheck failed
```

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

```
# let sumdiff(x,y) = (x+y,x-y);;                                     10
sumdiff = - : ((int #  int) -> (int #  int))

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

B.2.5 Recursion

The following is an attempt to define the factorial function:

```
# let fact n = if n=0 then 1 else n*fact(n-1);;                    11

unbound or non-assignable variable fact
1 error in typing
typecheck failed
```

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 is not defined before its own declaration, an error results. To make things clear consider:

```
# let f n = n+1;;                                                  12
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, letrec instead of let must be used. The correct recursive definition of the factorial function is thus:

```
# letrec fact n = if n=0 then 1 else n*fact(n-1);;                 13
fact = - : (int -> int)

# fact 3;;
6 : int
```

B.2.6 Iteration

The construct if e_1 then e_2 loop e_3 is the same as if e_1 then e_2 else e_3 in the true case; when e_1 evaluates to false, e_3 is evaluated and control loops back to the front of the construct again. As an illustration, here is an iterative definition of fact using two local assignable variables: count and result.

```
# let fact n =                                                     14
#     letref count=n and result=1
#     in     if count=0
#            then result
#            loop count,result := count-1,count*result;;
fact = - : (int -> int)

# fact 4;;
24 : int
```

Replacing the then in if e_1 then e_2 else e_3 by loop causes iteration when e_1 evaluates to true: e.g., if e_1 loop e_2 else e_3 is equivalent to if not(e_1) then e_3 loop e_2. 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 loop when the whole construct is to be re-evaluated after evaluating the guarded expression:

15

```
# let gcd(x,y) =
#     letref x,y = x,y
#     in     if x>y loop x:=x-y
#            if x<y loop y:=y-x
#            else x;;
gcd = - : ((int #  int) -> int)

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

B.2.7 Lists

If $e_1, \ldots, e_n$ all have type *ty* then the ML expression `[`e_1`;...;`e_n`]` 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 `[]`), and the infixed operators `.` (cons) and `@` (append, or concatenation).

16

```
# 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];;

ill-typed phrase: true
has an instance of type   bool
which should match type   int
1 error in typing
typecheck failed
```

All the members of a list must have the same type (although this type could be a sum, or disjoint union type—see Section B.5).

B.2.8 Tokens

A sequence of characters enclosed between token quotes (‘ – i.e. ascii 96) is a *token*.

17

```
# `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 ‘‘tok$_1$tok$_2$...tok$_n$‘‘ is an alternative syntax for [‘tok$_1$‘;‘tok$_2$‘;...;‘tok$_n$‘].

B.2.9 Strings

A sequence of characters enclosed between string quotes (" – i.e. ascii 34) is a *string*.

```
# "this is a string";;                                              18
"this is a string" : string

# "";;
"" : string
```

Although similar, strings and tokens are implemented differently in Lisp; strings are implemented as character arrays, and tokens as symbols. The implementation affects the efficiency of such operations as comparison and concatenation; Tokens are much slower to concatenate, but faster to compare.

B.2.10 Polymorphism

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

```
# hd [1;2;3];;                                                      19
1 : int

# hd [true;false;true];;
true : bool

# hd [1,2;3,4];;
(1, 2) : (int #  int)
```

Thus `hd` has several types; for example, it is used above with types `(int list) -> int`, `(bool list) -> bool`, and `(int # int) list -> (int # int)`. In fact if *ty* is *any* type then `hd` has the type `(`*ty* `list) ->` *ty*. Functions, like `hd`, with many types are called *polymorphic*, and ML uses type variables `*`, `**`, `***` etc. to represent their types.

```
# hd;;                                                              20
- : (* list -> *)

# letrec map f l = if null l then []
#                            else f(hd l).map f (tl l);;
map = - : ((* -> **) -> * list -> ** list)

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

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

```
# map null [[1;2]; []; [3]; []];;                                   21
[false; true; false; true] : bool list
```

B.2.11 Lambda-expressions

The expression `\`x`.`e evaluates to a function with formal parameter x and body e. Thus the declaration `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 `\`x`.`e and `\(`x`,`y`)` z`.`e are called lambda-expressions.

```
# \x.x+1;;                                                        22
- : (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 = - : (* list list -> * list list)

# doubleup [ [1;2]; [3;4;5] ];;
[[1; 2; 1; 2]; [3; 4; 5; 3; 4; 5]] : int list list

# doubleup [];;
[] : * list list
```

B.2.12 Failure

Some standard functions *fail* at run-time on certain arguments, yielding a string (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`).

```
# hd(tl[2]);;                                                     23
evaluation failed     hd

# 1/0;;
evaluation failed     div

# (1/0)+1000;;
evaluation failed     div

# failwith (hd [‘a‘;‘b‘]);;
evaluation failed     a
```

A failure can be *trapped* by `?`; the value of the expression e_1 `?` e_2 is that of e_1, unless e_1 causes a failure, in which case it is the value of e_2.

```
# hd(tl[2]) ? 0;;                                                 24
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`.

```
# half 4;;                                                    25
2 : int

# half 0;;
evaluation failed zero

# half 3;;
evaluation failed odd

# half 3 ? 1000;;
1000 : int
```

Failures may be*trapped selectively* (on string) by ??; if e_1 fails with token t, then the value of e_1 ?? [t_1;...;t_n] e_2 is the value of e_2 if t is one of t_1,...,t_n; otherwise the expression fails with the value of t.

```
# half(0) ?? [`zero`;`plonk`] 1000;;                          26
1000 : int

# half(1) ?? [`zero`;`plonk`] 1000;;
evaluation failed     odd
```

One may add several ?? traps to an expression, and one may add a ? trap at the end as a catch-all.

```
# half(1)                                                     27
#   ??[`zero`] 1000
#   ??[`odd`]  2000;;
2000 : int

# hd(tl[half(4)])
#   ??[`zero`] 1000
#   ??[`odd`]  2000
#   ? 3000;;
3000 : int
```

One may use ! or !! in place of ? or ?? to cause re-iteration of the whole construct, analogously to using loop in place of then.

```
# let same(x,y) =                                             28
#     if x>y then failwith `greater`
#     if x<y then failwith `less`
#            else x;;
same = - : ((int #  int) -> int)

# let gcd(x,y) =
#     letref x,y = x,y
#     in  same(x,y)
#           !![`greater`] x:=x-y
#           !![`less`]    y:=y-x;;
gcd = - : ((int #  int) -> int)

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

B.2.13 Type abbreviations

Types can be given names:

```
# lettype intpair = int #  int;;                                    29
type intpair defined

# let p = 12,20;;
p = (12, 20) : intpair
```

The new name is simply an abbreviation; for example, `intpair` and `int # int` are completely equivalent. The system always uses the most recently defined name when printing types.

```
# gcd;;                                                             30
- : (intpair -> int)

# gcd p;;
4 : int
```

B.2.14 Abstract types

New types can also be defined by abstraction. For example, to define a type `time` we could use the construct `abstype`:

```
# abstype time = int #  int                                         31
#  with maketime(hrs,mins) = if  hrs<0 or 23<hrs or
#                                   mins<0 or 59<mins
#                                then fail
#                                else abs_time(hrs,mins)
#  and hours t = fst(rep_time t)
#  and minutes t = snd(rep_time t);;
maketime = - : (intpair -> time)
hours = - : (time -> int)
minutes = - : (time -> int)
```

This defines an abstract type `time` and three primitive functions: `maketime`, `hours` and `minutes`. In general, an abstract type declaration has the form `abstype` *ty* `=` *ty′* `with` *b* where *b* is a binding, i.e. the kind of phrase that can follow `let` or `letrec`. Such a declaration introduces a new type *ty* which is represented by *ty′*. Only within *b* can one use the (automatically declared) functions `abs_`*ty* (of type *ty′* `->` *ty*) and `rep_`*ty* (of type *ty* `->` *ty′*), which map between a type and its representation. In the example above `abs_time` and `rep_time` are only available in the definitions of `maketime`, `hours` and `minutes`; these latter three functions, on the other hand, are defined throughout the scope of the declaration. Thus an abstract type declaration simultaneously declares a new type together with primitive functions for the type. The representation of the type (i.e. *ty′*), and of the primitives (i.e. the right hand sides of the definitions in *b*), is not accessible outside the `with`-part of the declaration.

```
# let t = maketime(8,30);;                                          32
t = - : time

# hours t , minutes t;;
(8, 30) : intpair
```

Notice that values of an abstract type are printed as `-`, like functions.

B.2.15 Type constructors

Both `list` and `#` are examples of type constructors; `list` has one argument (hence `* list`) whereas `#` has two (hence `* # **`). Each type constructor has various primitive operations associated with it, for example `list` has `null`, `hd`, `tl`, ... etc, and `#` has `fst`, `snd` and the infix `,`.

```
# let z = it;;                                                    33
z = (8, 30) : intpair

# fst z;;
8 : int

# snd z;;
30 : int
```

Another standard constructor of two arguments is `+`; `* + **` is the disjoint union of types `*` and `**`, and associated with it are the following primitives:

`isl  : (* + **) -> bool`	tests membership of left summand
`inl  : * -> (* + **)`	injects into left summand
`inr  : * -> (** + *)`	injects into right summand
`outl : (* + **) -> *`	projects out of left summand
`outr : (* + **) -> **`	projects out of right summand

These are illustrated by:

```
# let x = inl 1                                                   34
# and y = inr 2;;
x = inl 1 : (int + *)
y = inr 2 : (* + int)

# isl x;;
true : bool

# isl y;;
false : bool

# outl x;;
1 : int

# outl y;;
evaluation failed     outl

# outr x;;
evaluation failed     outr

# outr y;;
2 : int
```

Abstract types such as `time` defined above can be thought of as type constructors with no arguments (i.e. nullary constructors). The `abstype...with...` construct may also be used to define non-nullary type constructors (with `absrectype` in place of `abstype` if these are recursive). For example, trees analogous to LISP S-expressions could be defined by:

```
#  absrectype * sexp = * + (* sexp) #  (* sexp)                        35
#    with cons(s1,s2) = abs_sexp (inr (s1,s2))
#    and car s = fst (outr(rep_sexp s))
#    and cdr s = snd (outr(rep_sexp s))
#    and atom s = isl(rep_sexp s)
#    and makeatom a = abs_sexp(inl a);;
cons = - : ((* sexp #  * sexp) -> * sexp)
car = - : (* sexp -> * sexp)
cdr = - : (* sexp -> * sexp)
atom = - : (* sexp -> bool)
makeatom = - : (* -> * sexp)
```

B.3 Syntax of ML

We shall use variables to range over the various constructs of ML as follows:

Variable	Ranges over
var	variables
con	constructors
ce	constant expressions
ty	types
tab	type abbreviation bindings (see B.5.4)
ab	abstract type bindings (see B.5.5)
d	declarations
b	bindings
p	patterns
e	expressions

Variables and constructors are both represented by identifiers but they are different syntax classes. Identifiers and constant expressions are described in Section B.3.2 below. Types and type-bindings are explained in Section B.5. Declarations, bindings, patterns and expressions are defined by the following BNF-like syntax equations in which:

1. Each variable ranges over constructs as above.
2. The numbers following the various variables are there merely to distinguish between different occurrences—this will be convenient when we describe the semantics in Section B.4.
3. $\{C\}$ denotes an optional occurrence of C, and for n>1 $\{C_1 | C_2 \ldots | C_n\}$ denotes a choice of exactly one of C_1,C_2,...,C_n.
4. The constructs are listed in order of decreasing binding power.
5. 'L' or 'R' following a construct means that it associates to the left (L) or right (R) when juxtaposed with itself (where this is syntactically admissible).
6. Certain constructs are equivalent to others and this is indicated by 'equiv.' followed by the equivalent construct.

B.3.1 Syntax equations for ML

Table B.1 describes ML declarations, Table B.2 bindings, Table B.3 patterns, and Table B.4 on page 187 describes expressions.

d	::=	`let` b	ordinary variables
	|	`letref` b	assignable variables
	|	`letrec` b	recursive functions
	|	`lettype` tab	concrete types
	|	`rectype` cb	recursive concrete types
	|	`abstype` ab	abstract types
	|	`absrectype` ab	recursive abstract types

Table B.1: Declarations

b	::=	p`=`e	simple binding
	|	$id\ p_1\ p_2\ \ldots\ p_n\ \{$`:`$ty\}$ `=` e	function definition
	|	b_1 `and` b_2 $\ldots$ `and` b_n	multiple binding

Table B.2: Bindings

p	::=	`()`		empty pattern
	|	id		variable
	|	p`:`ty		type constraint
	|	p_1`.`p_2	R	list cons
	|	p_1`,`p_2	R	pairing
	|	`[]`		empty list
	|	`[`p_1`;`p_2 $\ldots$ `;`p_n`]`		list of n elements
	|	`(`p`)`		equivalent to p

Table B.3: Patterns

In the syntax equations constructs are listed in order of decreasing binding power. For example, since e_1e_2 is listed before e_1`;`e_2 function application binds more tightly than sequencing and thus e_1e_2`;` e_3 parses as `(`e_1e_2`);` e_3. This convention determines only the relative binding power of different constructs. The left or right association of a construct is indicated explicitly by 'L' for left and 'R' for right. For example, as application associates to the left, the expression $e_1e_2e_3$ parses as `(`e_1e_2`)` e_3, and since e_1 `=>` e_2 `|` e_3 associates to the right, the expression e_1 `=>` e_2 `|` e_3 `=>` e_4 `|` e_5 parses as e_1 `=>` e_2 `|` `(`e_3 `=>` e_4 `|` e_5`)`.

Only functions can be defined with `letrec`. For example, `letrec x = 2-x` would cause a syntax error.

All the variables occurring in a pattern must be distinct. On the other hand, a pattern can contain multiple occurrences of the wildcard `()`.

Spaces (ASCII 32), carriage returns (ASCII 13), line feeds (ASCII 10) form feeds (`^L`, ASCII 12) and tabs (`^I`, ASCII 9) can be inserted and deleted arbitrarily without affecting the meaning (as long as obvious ambiguities are not introduced). For example, the space in `-` x but not in `not` x can be omitted. *Comments*, which are arbitrary sequences of characters surrounded by `%`'s, can be inserted anywhere a space is allowed.

e	$::=$	ce		constant
	$\mid$	var		variable
	$\mid$	e_1 e_2	L	function application
	$\mid$	e`:`ty		type constraint
	$\mid$	`-`e		unary minus
	$\mid$	e_1`*`e_2	L	multiplication
	$\mid$	e_1`/`e_2	L	division
	$\mid$	e_1`+`e_2	L	addition
	$\mid$	e_1`-`e_2	L	subtraction
	$\mid$	e_1`<`e_2		less than
	$\mid$	e_1`>`e_2		greater than
	$\mid$	e_1`.`e_2	R	list cons
	$\mid$	e_1`@`e_2	R	list append
	$\mid$	e_1`=`e_2	L	equality
	$\mid$	`not` e		negation
	$\mid$	e_1 `&` e_2	R	conjunction
	$\mid$	e_1 `or` e_2	R	disjunction
	$\mid$	e_1 *user-infix* e_2	L	user declared infix identifier
	$\mid$	e_1`=>`$e_2 \mid e_3$	R	equivalent to `if` e_1 `then` e_2 `else` e_3
	$\mid$	`do` e		evaluate e for side effects
	$\mid$	e_1`,`e_2	R	pairing
	$\mid$	p`:=`e		assignment
	$\mid$	`fail`		equivalent to `failwith 'fail'`
	$\mid$	`failwith e`		failure with explicit token
	$\mid$	`if` e_1 $\{\texttt{then}\mid\texttt{loop}\}$ e'_1		conditional and loop
		$\{$ `if` e_2 $\{\texttt{then}\mid\texttt{loop}\}$ e'_2		
		$\vdots$		
		`if` e_n $\{\texttt{then}\mid\texttt{loop}\}$ e'_n $\}$		
		$\{$ $\{\texttt{else}\mid\texttt{loop}\}$ e''_n $\}$		
	$\mid$	e $\{\texttt{??}\mid\texttt{!!}\}$ e_1 e'_1		failure trap and loop
		$\{$ $\{\texttt{??}\mid\texttt{!!}\}$ e_2 e'_2		
		$\vdots$		
		$\{\texttt{??}\mid\texttt{!!}\}$ e_n e'_n $\}$		
		$\{$ $\{\texttt{?}\mid\texttt{!}\mid\texttt{?}\backslash id\mid\texttt{!}\backslash id\}$ e''_n $\}$		
	$\mid$	e_1`;`e_2 ... `;`e_n		sequencing
	$\mid$	`[]`		empty list
	$\mid$	`[`e_1`;`e_2 ... `;`e_n`]`		list of n elements
	$\mid$	e `where` b	R	equivalent to `let` b `in` e
	$\mid$	e `whereref` b	R	equivalent to `letref` b `in` e
	$\mid$	e `whererec` b	R	equivalent to `letrec` b `in` e
	$\mid$	e `wheretype` db		equivalent to `lettype` db `in` e
	$\mid$	e `whereabstype` ab		equivalent to `abstype` ab `in` e
	$\mid$	e `whereabsrectype` ab		equivalent to `absrectype` ab `in` e
	$\mid$	d `in` e		local declaration
	$\mid$	*backslash* p_1 p_2 ... p_n`.`e		abstraction
	$\mid$	`(`e`)`		equivalent to e

Table B.4: Expressions

B.3.2 Identifiers and other lexical matters

In this section the lexical structure of ML is defined.

B.3.2.1 Identifiers

A variable (*var*) or *identifier* is a sequence of alphanumerics starting with a letter, where an alphanumeric is either a letter, a digit, a prime (') or an underbar (_). ML is case-sensitive: upper and lower case letters are considered to be different.

B.3.2.2 Constant expressions

The ML constant expressions (*ce*'s) used in Table B.4 are:

1. Integers, i.e. sequences of digits 0,1...,9.
2. Truth values true and false.
3. Tokens and token-lists:
   (a) Tokens consist of any sequence of characters surrounded by token quotes (`), e.g. `This is a single token`.
   (b) Token-lists consist of any sequence of tokens (separated by spaces, returns, line-feed or tabs) surrounded by token-list quotes (``). e.g. ``this is a token-list containing 7 members``. `` tok1 tok2 ... tokn`` is equivalent to [`tok1`; `tok2`; ...; `tokn`].

   In any token or token-list, the occurrence of \\x has the following meanings for different x's:

\0	=	ten spaces
\n	=	n spaces (0<n<10)
\S	=	one space
\R	=	return
\L	=	line-feed
\T	=	tab
\x	=	x taken literally otherwise (e.g. \` to include token quotes in a token or token-list)

4. Strings, consisting of any sequence of characters surrounded by string quotes ("), e.g. "This is a single string". Any " characters within a string must be preceded by \\. The escape sequence \\x for any other character means always to insert the character x.
5. The expression (), called *thing*, which evaluates to the unique object of ML type unit.

B.3.2.3 Prefixes and infixes

The ML *prefixes px* and *infixes ix* are given by:

```
px  ::=  not | - | do
ix  ::=  * | / | + | - | . | @ | = | < | > | & | or | ,
```

In addition, any identifier (and certain single characters) can be made into an infix. Such user-defined infixes bind more tightly than ...=>...|... but more weakly than or. All of them have the same power binding and associate to the left.

Except for `&` and `or`, each infix *ix* (or prefix *px*) has correlated with it a special identifier `$`*ix* (or `$`*px*) which is bound to the associated function. For example, the identifier `$+` is bound to the addition function, and `$@` to the list-append function (see Section B.6 for the meaning of dollared infixes). This is useful for passing functions as arguments; for example, *f*`$@` applies *f* to the append function.

See the descriptions of the functions `ml_paired_infix` and `ml_curried_infix` in Section B.6.1 for details of how to give an identifier infix status.

B.4 Semantics of ML

The evaluation of all ML constructs takes place in the context of an *environment* and a *store*. The environment specifies what the variables and constructors in use denote. Variables may be bound either to *values* or to *locations*. The contents of locations—which must be values—are specified in the *store*. If a variable is bound to a location then (and only then) is it *assignable*. Thus bindings are held in the environment, whereas location contents are held in the store. Constructors may only be bound to values (constructor constants or constructor functions) and this binding occurs when they are declared in a concrete type definition.

The evaluation of ML constructs may either *succeed* or *fail*. In the case of success:

1. The evaluation of a declaration, *d* say, changes the bindings in the environment of the identifiers declared in *d*. If *d* is at top-level, then the scope of the binding is everything following *d*. In *d* `in` *e* the scope of *d* is the evaluation of *e*, and so when this is finished the environment reverts to its original state (see Section B.4.1).
2. The evaluation of an expression yields a value: the value of the expression (see Section B.4.2).

If an assignment is done during an evaluation, then the store will be changed — we shall refer to these changes as *side effects* of the evaluation.

If the evaluation of a construct fails, then failure is signalled, and a string is passed to the context which invoked the evaluation. This string is called the *failure string*, and it normally indicates the cause of the failure. During evaluation, failures may be generated either *implicitly* by certain error conditions, or *explicitly* by the construct `failwith` *e* (which fails with *e*'s value as failure string). For example, the evaluation of the expression `1/0` fails implicitly with failure string `‘div‘`, while that of `failwith ‘`*str*`‘` fails explicitly with failure string `‘`*str*`‘`. We shall say two evaluations fail *similarly* if they both fail with the same failure string. For example, the evaluation of `1/0` and `failwith ‘div‘` fail similarly. Side effects are not undone by failures.

If during the evaluation of a construct a failure is generated, then unless the construct is a failure trap (i.e. an expression built from `?` and/or `!`) the evaluation of the construct itself fails similarly. Thus failures propagate up until trapped, or reaching top level. For example, when evaluating `(1/0)+1000`, the expression `1/0` is first evaluated, and the failure which this evaluation generates causes the evaluation of the whole expression (viz. `(1/0)+1000`) to fail with `‘div‘`. On the other hand, the evaluation of `(1/0)?1000` traps the failure generated by the evaluation of `1/0`, and succeeds with value `1000`. (In general, the evaluation of e_1`?`e_2 proceeds by first evaluating e_1, and if this succeeds with value E, then E is returned as the value of e_1`?`e_2; however, if e_1 fails, then the result of evaluating e_1`?`e_2 is determined by evaluating e_2).

In describing evaluations, when we say that we *pass control* to a construct, we mean that the outcome of the evaluation is to be the outcome of evaluating the construct. For example, if when evaluating e_1`?`e_2 the evaluation of e_1 fails, then we pass control to e_2.

Expressions and patterns can be optionally decorated with types by writing `:`ty after them (e.g. `[]:int list`). The effect of this is to force the type checker to assign an instance of the asserted type to the construct; this is useful as a way of constraining types more than the type checker would otherwise (i.e. more than context demands), and it can also serve as helpful documentation. Details of types and type checking are given in Section B.5, and will be ignored in describing the evaluation of ML constructs in the rest of this section.

If we omit types, precedence information and those constructs which are equivalent to others, then the syntax of ML can be summarized by:

$$
\begin{array}{lll}
d & \texttt{::=} & \texttt{let}\ b \mid \texttt{letref}\ b \mid \texttt{letrec}\ b \\
b & \texttt{::=} & p\texttt{=}e \mid var\ p_l\ p_2 \ldots p_n\ \texttt{=}\ e \mid b_l\ \texttt{and}\ b_2 \ldots\ \texttt{and}\ b_n \\
p & \texttt{::=} & \texttt{()}\ \mid var \mid p_l\texttt{.}p_2 \mid p_l\texttt{,}p_2 \mid \texttt{[]} \mid \texttt{[}p_l\texttt{;}p_2 \ldots\ \texttt{;}p_n\texttt{]} \\
e & \texttt{::=} & ce \mid var \mid e_1\ e_2 \\
 & \mid & px\ e \mid e_1\ ix\ e_2 \mid v\texttt{:=}e \mid\ \texttt{failwith}\ e \\
 & \mid & \qquad \texttt{if}\ e_1\ \{\texttt{then}|\texttt{loop}\}\ e'_1 \\
 & & \{\quad \texttt{if}\ e_2\ \{\texttt{then}|\texttt{loop}\}\ e'_2 \\
 & & \qquad\qquad \vdots \\
 & & \qquad \texttt{if}\ e_n\ \{\texttt{then}|\texttt{loop}\}\ e'_n\ \} \\
 & & \{\qquad\qquad \{\texttt{else}|\texttt{loop}\}\ e''_n\ \} \\
 & \mid & \quad e\quad \{\texttt{??}|\texttt{!!}\}\ e_1\ e'_1 \\
 & & \{\quad \{\texttt{??}|\texttt{!!}\}\ e_2\ e'_2 \\
 & & \qquad\qquad \vdots \\
 & & \qquad \{\texttt{??}|\texttt{!!}\}\ e_n\ e'_n\ \} \\
 & & \{\quad \{\texttt{?}|\texttt{!}|\texttt{?}\backslash id|\texttt{!}\backslash id\}\ e''_n\ \} \\
 & \mid & e_l\texttt{;}e_2 \ldots\texttt{;}e_n \mid \texttt{[]} \mid \texttt{[}e_l\texttt{;}e_2 \ldots\texttt{;}e_n\texttt{]} \mid d\ \texttt{in}\ e \\
 & \mid & \backslash p_1 p_2 \ldots p_n\texttt{.}e
\end{array}
$$

B.4.1 Declarations

Any declaration must be one of the three kinds: `let` b, `letref` b or `letrec` b, where b is a binding. Each such declaration is evaluated by first evaluating the binding b to produce a (possibly empty) set of variable-value pairs, and then extending the environment (in a manner determined by the kind of declaration) so that each variable in this set of pairs denotes its corresponding value. The evaluation of bindings is described below in Section B.4.1.1.

1. Evaluating `let` b declares the variables specified in b to be an ordinary (i.e. non assignable) variable, and binds (in the environment) each one to the corresponding value produced by evaluating b. To understand what are the variables defined in a declaration may require some knowledge about the environment. For example, a declaration `let` f x `=` e declares x if f is a constructor and declares f as the function $\backslash x.e$ otherwise.
2. Evaluating `letref` b declares the variables specified in b to be assignable and thus binds (in the environment) each one to a new location, whose contents (in the store) is set to the corresponding value. The effect of subsequent assignments to the variables will be to change the contents of the locations they are bound to. Bindings (in the environment) of variables to locations can only be changed by evaluating another declaration to supersede the original one.
3. Evaluating `letrec` b is similar to evaluating `let` b except that:
   (a) The binding b in `letrec` b must consist only of function definitions.

(b) These functions are made mutually recursive.

For example, consider:

(a) `let f n = if n=0 then 1 else n*f(n-1)`

(b) `letrec f n = if n=0 then 1 else n*f(n-1)`

The meaning of `f` defined by the first case depends on whatever `f` is bound before the declaration is evaluated, while the meaning of `f` defined by the second case is independent of this (and is the factorial function).

B.4.1.1 The evaluation of bindings

There are three kinds of variable binding each of which, when evaluated, produces a set of variable-value pairs (or fails):

1. *Simple bindings*, which have the form p=e where p is a pattern and e an expression.
2. *Function definitions*, which have the form $id\ p_1 \ldots p_n$ `=` e. This is just an abbreviation for the simple binding id `=` \\$p_1 \ldots p_n$`.`$e$.
3. *Multiple bindings*, which have the form b_1 `and` $b_2 \ldots$ `and` b_n where $b_1, b_2 \ldots, b_n$ are simple bindings or function definitions. As a function definition is just an abbreviation for a certain simple binding, each b_i ($0<i<n+1$) either is, or is an abbreviation for, some simple binding p_i=e_i. The multiple binding b_1`and` $b_2 \ldots$ `and` b_n then abbreviates p_1`,`$p_2 \ldots$`,`p_n `=` e_1`,` $e_2 \ldots e_n$, which is a simple binding.

As function definitions and multiple bindings are abbreviations for simple bindings we need only describe the evaluation of the latter.

A simple binding p=e is evaluated by first evaluating e to obtain a value E (if the evaluation fails then the evaluation of p=e fails similarly). Next the pattern p is *matched* with E to see if they have the same form (precise details are given in Section B.4.1.2). If so, then to each identifier in p there is a corresponding component of E. The evaluation of p=e then returns the set of each identifier paired with its corresponding component. If p and E do not match then the evaluation of p=e fails with failure token `‘MATCH‘`.

B.4.1.2 Matching patterns and expression values

When a pattern p is matched with a value E, either the match succeeds and a set of identifier-value pairs is returned (each identifier in p being paired with the corresponding component of E), or the match fails. We describe, by cases on p, the conditions for p to match E and the sets of pairs returned:

`()`**:** Always matches E. The empty set of pairs is returned.

*var***:** Always matches E. The set consisting of *var* paired with E is returned.

p_1`.`p_2**:** E must be a non-empty list E_1`.`E_2 such that p_1 matches E_1 and p_2 matches E_2. The union of the sets of pairs returned from matching p_1 with E_1 and p_2 with E_2 is returned.

p_1`,`p_2**:** E must be a pair E_1`,`E_2 such that p_1 matches E_1 and p_2 matches E_2. The union of the sets of pairs returned from matching p_1 with E_1 and p_2 with E_2 is returned.

`[`p_1`;`$p_2 \ldots$`;`p_n`]`**:** E must be a list `[`E_1`;`$E_2 \ldots$`;`E_n`]` of length n such that for each i p_i matches E_i. The union of the sets of pairs returned by matching p_i with E_i is produced.

Thus if p matches E, then p and E have a similar 'shape', and each identifier in p corresponds to some component of E (namely that component paired with the identifier in the set returned by the match). Here are some examples:

1. `[`x`;`y`;`z`]` matches `[1;2;3]` with x, y and z corresponding to `1`, `2`, and `3` respectively.
2. `[`x`;`y`;`z`]` does not match `[1;2]` or `[1;2;3;4]`.
3. x`.`y matches `[1;2;3]` with x and y corresponding to `1` and `[2;3]` respectively, because E_1`.[`E_2`;`E_3`...;`E_n`]` = `[`E_1`;`E_2`;`E_3`...;`E_n`]`.
4. x`.`y does not match `[]` or `1,2`.
5. x`,`y matches `1,2` with x and y corresponding to `1` and `2` respectively.
6. x`,`y does not match `[1;2]` .
7. `(`x`,`y`),[(`z`.`w`);()]` matches `(1,2),[[3;4;5];[6;7]]` with x, y, z and w corresponding to `1`, `2`, `3`, and `[4;5]` respectively.

B.4.2 Expressions

If the evaluation of an expression terminates, then either it succeeds with some value, or it fails; in either case assignments performed during the evaluation may cause side effects. If the evaluation succeeds with some value we shall say that value is *returned*.

We shall describe the evaluation of expressions by considering the various cases, in the order in which they are listed in the syntax equations.

ce: The appropriate constant value is returned.

var: The value associated with *var* is returned. If *var* is ordinary, then the value returned is the value bound to *var* in the environment. If *var* is assignable, then the value returned is the contents of the location to which *var* is bound.

e_1 e_2: e_1 and e_2 are evaluated and the result of applying the value of e_1 (which must be a function) to that of e_2 is returned. Due to optimizations in the ML compiler, the order of evaluation may vary.

px e: e is evaluated and then the result of applying *px* to the value of e is returned. `-`e and `not` e have the obvious meanings; `do` e evaluates e for its side effects and then returns `()`.

e_1 *ix* e_2: e_1 `&` e_2 is equivalent to `if` e_1 `then` e_2 `else false`, so sometimes only e_1 needs be evaluated to evaluate e_1 `&` e_2. e_1 `or` e_2 is equivalent to `if` e_1 `then true else` e_2, so sometimes only e_1 needs to be evaluated to evaluate e_1 `or` e_2.

In all other cases e_1 and e_2 are evaluated (in that order) and the result of applying *ix* to their two values is returned. e_1`,`e_2 returns a pair whose first component is the value of e_1, and whose second component is the value of e_2. The meaning of the other infixes are given in Section B.6.

p`:=`e: Every variable in p must be assignable and bound to some location in the environment. The effect of the assignment is to update the contents of these locations (in the store) with the values corresponding to the variables produced by evaluating the binding p=e (see Section B.4.1.1). If the evaluation of e fails, then no updating of locations occurs, and the assignment fails similarly. If the matching to p fails, then the assignment fails with `‘MATCH‘`. The value of p`:=`e is the value of e.

`failwith` e: e is evaluated and then a failure with e's value (which must be a token) is generated.

$$\begin{array}{l} \texttt{if}\ e_1\ \{\texttt{then|loop}\}\ e'_1 \\ \{\texttt{if}\ e_2\ \{\texttt{then|loop}\}\ e'_2 \\ \quad\vdots \\ \texttt{if}\ e_n\ \{\texttt{then|loop}\}\ e'_n\ \} \\ \quad\{\ \{\texttt{else|loop}\}\ e'\ \} \end{array}$$

: e_1, e_2,…,e_n are evaluated in turn until one of them, say e_m, returns `true` (each e_i must be a boolean expression). When the phrase following e_m is `then` e'_m control is passed to e'_m. However, when the phrase is `loop` e'_m then e'_m is evaluated for its side effects, and then control is passed back to the beginning of the whole expression again (i.e. to the beginning of `if` e_1 …).

In the case that all of e_1,e_2…,e_n return `false` and there is a phrase following e'_n, then if this is `else` e' control is passed to e', while if it is `loop` e' then e' is evaluated for its side effects and control is then passed back to the beginning of the whole expression again.

In the case that all of e_1,e_2…,e_n return `false`, but no phrase follows e'_n then `()`, the unique value of type `void` is returned.

$$\begin{array}{l} e\ \{\texttt{??|!!}\}\ e_1e'_1 \\ \{\ \{\texttt{??|!!}\}\ e_2e'_2 \\ \quad\vdots \\ \quad\{\texttt{??|!!}\}\ e_ne'_n\ \} \\ \{\ \{\texttt{?|!|?}\backslash id|\backslash id\}\ e'\ \} \end{array}$$

: e is evaluated and if this succeeds its value is returned. If e fails with failure token *tok*, then each of e_1,e_2…,e_n are evaluated in turn until one of them, say e_m, returns a token list containing *tok* (each e_i must be a token). If `??` immediately precedes e_m, then control is passed to e'_m. If `!!` precedes it, then e'_m is evaluated and control is passed back to the beginning of the whole expression e `{??|!!}` ….

If none of e_1,e_2…, e_n produces a token list containing *tok*, and `?\`e' follows e'_n, then control is passed to e'. But if `!\`e' follows e'_n, then e' is evaluated, and control is passed back to the beginning of the whole expression.

If `?\`*id* e' or `?\`*id* e' follows e'_n, then e' is evaluated in an environment in which *id* is bound to the failure string *tok* (i.e. an evaluation equivalent to `let` *id*=*tok* `in` e' is done), and then depending on whether a `?` of a `!` occurred, the value of e' is returned or control is passed back to the beginning of the whole expression respectively.

If none of e_1,e_2…,e_n returns a token list containing *tok* and nothing follows e'_n, then the whole expressions fails with *tok*.

e_1`;`e_2…`;`e_n**:** e_1,e_2…,e_n are evaluated in that order, and the value of e_n is returned.

`[`e_1`;`e_2…`;`e_n`]`**:** $e_1, e_2, \ldots, e_n$ are evaluated in that order and the list of their values returned. `[]` evaluates to the empty list.

d `in` e**:** d is evaluated and then e is evaluated in the extended environment and its value returned. The declaration d is local to e, so that after the evaluation of e, the former environment is restored.

`\`$p_1p_2 \ldots p_n$`.`e**:** The evaluation of lambda-expressions always succeeds and yields a function value. The environment in which the evaluation occurs (i.e. in which the function value is created) is called the *definition environment*.

1. *Simple lambda-expressions*: `\`p`.`e evaluates to that function which, when applied to some argument yields the result of evaluating e in the current (i.e. application time) store, and in the environment obtained from the definition environment by binding any variables in p to the corresponding components of the argument (see Section B.4.1.1).
2. *Compound lambda-expressions*: A lambda-expression with more than one parameter is curried, i.e. `\`$p_1p_2 \ldots p_n$`.`e is equivalent to `\`p_1`. (\`p_2`. … \`p_n`.`e`)` ….

Thus the free variables in a function keep the same binding they had in the definition environment. So if a free variable is non-assignable in that environment, then its value is fixed to the value it has there. On the other hand, if a free variable is assignable in the definition environment, then it will be bound to a location. Although that binding is fixed, the contents of the location in the store is not, and can be subsequently changed with assignments.

B.5 ML Types

So far, little mention has been made of types. For ML in its original role as the meta-language for proof in LCF, the importance of strict type checking was principally to ensure that every computed value of the type representing theorems was indeed a theorem.[2]

The same effect could probably have been achieved by run-time type checking, but compile-time type checking was adopted instead, in the design of ML. This was partly for the considerable debugging aid that it provides; partly for efficient execution; and partly to explore the possibility of combining polymorphism with type checking. This last reason is of general interest in programming languages and has nothing to do specifically with proof; the problem is that there are many operations (list mapping functions, functional composition, etc.) which work at an infinity of types, and therefore their types should somehow be parameterized – but it is rather inconvenient to have to mention the particular type intended at each of their uses.

The ML type checking system is implemented in such a way that, although the user may occasionally (either for documentation or as a constraint) ascribe a type to an ML expression or pattern, it is hardly ever necessary to do so. The user of ML will almost always be content with the types ascribed and presented by the type checker, which checks every top-level phrase before it is evaluated. (The type checker may sometimes find a more general type assignment than expected.)

B.5.1 Types and objects

Every data object in ML possesses a type. Such an object may possess many types, in which case it is said to be *polymorphic* and possesses a *polytype* – i.e. a type containing type variables (for which we use a sequence of asterisks possibly followed by an identifier or integer) – and moreover it possesses all types which are *instances* of its polytype, formed by substituting types for zero or more type variables in the polytype. A type containing no type variables is a *monotype.*

We saw several examples of types in Section B.2. To understand the following syntax, note that `list` is a postfixed unary (one-argument) type constructor (thereafter abbreviated to *tycon*). The user may introduce new n-argument type constructors. A binary type operator `directory`, for example, can be introduced. The following type expressions will then be types of different kinds of `directory`:

- `(tok, int) directory`
- `(int, int -> int) directory`

The user may even deal with lists of directories, with the type `(int, bool) directory list`

B.5.1.1 The syntax of types

The syntax of ML types is summarized in Table B.5. Type abbreviations are introduced by a `lettype` declaration (see Section B.5.4 below) which allows an identifier to abbreviate an arbitrary monotype. An abstract type likewise consists of an identifier (introduced by an `abstype` or

[2]The NUPRL system also relies on strict type checking to ensure that objects of type `proof` can only be constructed by reference to a fixed set of inference rules.

Types	*ty*	::=	*sty*		Standard (non-infix) type
		|	*ty* # *ty*	R	Cartesian product
		|	*ty* + *ty*	R	Disjoint sum
		|	*ty* -> *ty*	R	Function type
Standard Types	*sty*	::=	unit | int		
		|	bool		
		|	tok | string		Basic types
		|	*vty*		Type variable
		|	*tycon*		Type abbreviation (see Section B.5.4)
		|	*tycon*		Nullary abstract type
		|	*tyarg tycon*	L	Abstract type (see Section B.5.5)
		|	(*ty*)		
Type arguments	*tyarg*	::=	*sty*		Single type argument
		|	(*ty*,...,*ty*)		One or more type arguments
Type variables	*vty*	::=	* | ** | ...		
		|	**id* | ***id* | ...		
		|	*0 | **0 | ...		
		|	*1 | **1 | ...		
		|	⋮ | ⋮ | ...		

Table B.5: ML Type Syntax

absrectype declaration; see Section B.5.5) postfixed to zero or more type arguments. Two or more arguments must be separated by commas and enclosed by parentheses. The type operator list is a predeclared unary type operator; and #, + and -> may be regarded as infix forms of three predeclared binary type operators.

For an object to possess[3] a type means the following: For basic types, all integers possess int, both booleans possess bool , all strings possess string, etc. The only object possessing void (or unit) is that denoted by () in ML. For a type abbreviation *tycon*, an object possesses *tycon* (during execution of phrases in the scope of the declaration of *tycon*) if and only if it possesses the type which *tycon* abbreviates. For compound monotypes ,

1. The type *ty* list is possessed by any list of objects, all of which possess type *ty* (so that the empty list possesses type *ty* list for every *ty*).

2. The type ty_1 # ty_2 is possessed by any pair of objects possessing the types ty_1 and ty_2, respectively.

3. The type ty_1 + ty_2 is possessed by the left-injection of any object possessing ty_1, and by the right-injection of any object possessing ty_2. These injections are denoted by the ML function identifiers inl : * -> * + ** and inr : ** -> * + ** (see Section B.6).

4. A function possesses type ty_1 -> ty_2 if, whenever its argument possesses type ty_1, its result (if defined) possesses type ty_2. (This is not an exact description; for example, a function defined in ML with non-local variables may possess this type even though some assumption about the types of the values of these non-local variables is necessary for the above condition to hold. The constraints on programs listed below ensure that the non-locals will always have the right types).

[3]We shall talk of objects *possessing* types and phrases *having* types, to emphasize the distinction.

5. An object possesses the abstract type *tyarg id* if and only if it is represented (via the abstract type representation) by an object possessing the *tyarg* instance of the right-hand side of the declaration of *id*.

Finally, an object possesses a polytype *ty* if and only if it possesses all monotypes which are substitution instances of *ty*.

B.5.2 Typing of ML phrases

We now explain the constraints used by the type checker in ascribing types to ML expressions, patterns and declarations.

The significance of expression e having type *ty* is that the value of e (if evaluation terminates successfully) possesses type *ty*. As consequences of the well-typing constraints listed below, it is impossible for example to apply a non-function to an argument, or to form a list of objects of different types, or (as mentioned earlier) to compute an object of the type corresponding to theorems which is not a theorem.

The type ascribed to a phrase depends in general on the entire surrounding ML program. In the case of top-level expressions and declarations, however, the type ascribed depends only on preceding top-level phrases. Thus you know that types ascribed at top-level are not subject to further constraint.

Before each top-level phrase is executed, types are ascribed to all its sub-expressions, sub-declarations and sub-patterns according to the following rules. Most of the rules are fairly natural; those which are less so are discussed later. You are only presented with the types of top-level phrases; the types of sub-phrases will hardly ever concern you.

Before giving the list of constraints, let us discuss an example which illustrates some important points. To map a function over a list we may define the polymorphic function `map` recursively as follows (where we have used an explicit abstraction, rather than `letrec map f l = ...`, to make the typing clearer):

```
letrec map = \f.\l. null l => [] | f(hd l).map f(tl l) ;;
```

From this declaration the type checker will infer a *generic* type for `map`. By 'generic' we mean that each later occurrence of `map` will be ascribed a type which is a substitution instance of the generic type.

Now the free identifiers in this declaration are `null`, `hd` and `$.`, which are ML primitives whose *generic* (poly)types are `* list -> bool`, `* list -> *`, and `* # * list -> * list` respectively. The first constraint used by the type checker is that the occurrences of these identifiers in the declaration are ascribed instances of their generic types. Other constraints which the type checker will use to determine the type of `map` are:

- All occurrences of a lambda-bound variable receive the same type.
- Each arm of a conditional receives the same type, and the condition receives type `bool`.
- In each application e `=` `(`$e_1 e_2$`)`, if e_2 receives *ty* and e receives *ty′* then e_1 receives *ty* `->` *ty′*.
- In each abstraction e `=` `\`v`.`e_1, if v receives *ty* and e_1 receives *ty′* then e receives *ty* `->` *ty′*.
- In a `letrec` declaration, all free occurrences of the declared variable receive the same type.

Now the type checker will ascribe the type `(*->**)->* list->** list` to `map`. This is in fact the most general type consistent with the constraints mentioned. Moreover, it can be shown that any instance of this type also allows the constraints to be satisfied; this is what allows us to claim that the declaration is indeed polymorphic.

In the following constraint list, we say p has ty to indicate that the phrase p is ascribed a type ty which satisfies the stated conditions. We use x, p, e, d to stand for variables, patterns, expressions and declarations respectively.

Constants:

1. `()` has type `unit`
2. `0` has type `int` , `1` has type `int`, ...
3. `true` has type `bool`, `false` has type `bool`
4. `‘...‘` has type `tok`
5. `"..."` has type `string`

Variables and constructors: The constraints described here are discussed in Section B.5.3 below.

1. If x is a variable bound by `\`, `fun` or `letref`, then x is ascribed the same type as its binding occurrence. In the case of `letref`, this must be monotype if the `letref` is top-level or an assignment to x occurs within a lambda-expression within its scope.
2. If x is bound by `let` or `letrec`, then x has ty, where ty is an instance of the type of the binding occurrence of x (i.e. the *generic* type of x), in which type variables occurring in the types of current lambda-bound or `letref`-bound identifiers are not instantiated.
3. If x is not bound in the program (in which case it must be an ML primitive), then x has ty, where ty is an instance of the type of x given in Section B.6.

Patterns: Cases for a pattern p:

- `()`: p has ty, where ty is any type.
- p_1`:`ty:
  p_1 and p have an instance of ty.
- p_1`,`p_2: If p_1 has ty_1 and p_2 has ty_2, then p has ty_1 `#` ty_2.
- p_1`.`p_2: If p_1 has ty then p_2 and p have ty `list`.
- `[`p_1`;...;`p_n`]`: For some ty, each p_i has ty and p has ty `list`.

Expressions: Cases for an expression e (not a constant or identifier):

- $e_1 e_2$: If e_2 has ty and e has ty' then e_1 has ty `->` ty'.
- e_1`:`ty: e_1 and e have an instance of ty.
- px e_1: Treated as `$`px`(`e_1`)` when px is a prefix. If e is `-`e_1, then e and e_1 have `int`.
- e_1 ix e_2: Treated as `$`ix`(`e_1`,`e_2`)` if ix is introduced with `ml_paired_infix`, and as `($`ix e_1 e_2`)` if ix is introduced by `ml_curried_infix`. If e is `(`e_1 `&` e_2`)` or `(`e_1 `or` e_2`)` then e, e_1 and e_2 have `bool`.
- e_1`,`e_2: If e_1 has ty_1 and e_2 has ty_2 then e has ty_1`#`ty_2.
- p`:=`e_1: For some ty, p, e_1 and e all have ty.
- `failwith` e_1: e_1 has `tok`, and e has any type.

- `if` e_1 `then` $e'_1 \ldots$ `if` e_n `then` e'_n `else` e': Each e_i has `bool`, and e, each e'_i, and e' all have *ty* for some *ty*. However, this constraint does not apply to an e'_i preceded by `loop` in place of `then`, nor to e' preceded by `loop` in place of `else`. If e' is absent, then *ty* = `void`.
- e'_0 `??` e_1 $e'_1 \ldots$ `??` e_n e'_n `?{\`x`}`e': Each e_i has `tok list`, and e, e'_0, each e'_i and e' all have *ty* for some *ty*. However, this constraint does not apply to an e'_i preceded by `!!` in place of `??` nor to e' preceded by `!` in place of `?`. If $\backslash x$ is present, x has `tok`.
- e_1`;...;`e_n: If e_n has *ty* then e has *ty*.
- `[`e_1`;...;`e_n`]`: For some *ty*, each e_i has *ty* and e has *ty* `list`.
- d `in` e_1: If e_1 has *ty* then e has *ty*. If d is a type definition (see Sections B.5.4 and B.5.5) then *ty* must contain no type defined in d.
- `\`p`.`e_1: If p has *ty* and e_1 has *ty*$'$ then e has *ty* `->`*ty*$'$.

Declarations:

1. Each binding$x$ $p_1 \ldots p_n$ `=` e is treated as x `=` `\`p_1`. ...\`p_n`.`e.
2. `let` p_1 `=` e_1 `and ...and` p_n `=` e_n is treated as `let` p_1`,...,`p_n `=` e_1`,...,`e_n (similarly for `letrec` and `letref`).
3. If d is `let` p`=`e, then d, p and e all have *ty* for some *ty* (similarly for `letref`). Note that e is not in the scope of the declaration.
4. If d is `letrec` x_1`,...,`x_n `=` e_1`,...,`e_n, then x_i and e_i have ty_i, and d has ty_1`#...#`ty_n for some ty_i. In addition, each free occurrence of x_i in $e_1,\ldots,e_n$ has ty_i, so that the type of recursive calls of x_i is the same as the declaring type.

B.5.3 Discussion of type constraints

We give here reasons for our constraints on the types ascribed to occurrences of identifiers. The reader may like to skip this section at first reading.

1. Consider constraint (1) for lambda-bound identifiers. This constraint implies that the expression `let` x `=` e `in` e' may be well-typed even if the semantically-equivalent expression `let` f x `=` e' `in` f e is not, since in the former expression x may occur in e' with two incompatible types which are both instances of the declaring type. The greater constraint on f is associated with the fact that f may be applied to many different arguments during evaluation. To show the need for the constraint, suppose that it is replaced by the weaker constraint for `let`-bound identifiers, so that for example `let f x = if x then 1+x else x(1)` is a well-typed declaration of type `*->int`, in which the occurrences of `x` receive types `*`, `bool`, `int`, `int->int` respectively. In the absence of an explicit argument for the abstraction, no constraint exists for the type of the binding occurrence of `x`. But, because `f` is `let`-bound, expressions such as `f true` and `` f `dog` `` are admissible in the scope of `f`, although their evaluation should result in either nonsense or *run-time* type-errors; one of our purposes is to preclude these.

   The only exception to this rule is for expressions of the form `(\`x`.`e'`)`e, which is treated exactly as `let` x`=`e `in` e'. Here we know the unique instance of type of the argument x, namely the type of e.

2. The analogous restriction for `letref`-bound identifiers is also due to the possibility that the identifier-value binding may change during evaluation (this time because of assignments). Consider the following:

   ```
   letref x = [] in
         (if e then do(x := 1.x) else do(x := [true]) ; hd x) ;;
   ```

   If `letref` were treated like `let`, this phrase would be well-typed and indeed have type `*`, despite the fact that the value returned is either `1` or `true`. So, calling the whole expression e, all manner of larger expressions involving e would be well-typed, even including $e(e)$!

3. Top level `letref`s must be monomorphic to avoid retrospective type constraints at top-level. If this restriction were removed the following would be allowed:

   ```
   letref x = [] ;;
       ⋮
      2.x ;;
   ```

   But on type checking the last phrase, it would appear that the type of `x` at declaration should have been `int list`, not `* list`, and the types of intervening phrases may likewise need constraining.

4. To see the need for the exclusion of polymorphic non-local assignments, consider this example in the HOL system (this example is originally due to Lockwood Morris). (The type `thm` is the type of theorems.)

   ```
   let store,fetch =
       letref x = [] in (\y. x:=[y]) , (\(). hd x ) ;;
   store "T = F" ;;
   let eureka :thm = fetch() ;;
   ```

   Now suppose we lift our constraint. Then in the declaration, `x` has type `* list` throughout its (textual) scope, and `store`, `fetch` receive types `*->* list`, `**->*` respectively. In the two ensuing phrases they get respective types `term->term list`, `void->thm` (instances of their declaring types), and the value of `eureka` is a contradictory formula masquerading as a theorem!

   The problem is that the type checker has no simple way of discovering the types of all values assigned to the polymorphic `x`, since these assignments may be invoked by calls of the function `store` outside the (textual) scope of `x`. This is not possible under our constraint.
   However, polymorphic assignable identifiers are still useful: consider

   ```
   let rev l =
        letref l,l’ = l,[] in
          if null l then l’ loop (l,l’:= tl l, hd l.l’) ;;
   ```

   Such uses of assignable identifiers for iteration may be avoided given a suitable syntax for iteration, but assignable identifiers are useful for a totally different purpose, namely as ‘own variables’ shared between one or more functions (as in the store-fetch example). Our constraint of course requires them to be monomorphic; this is one of the few cases where the user occasionally needs to add an explicit type to a program.

B.5.4 Type abbreviations

The syntax of type abbreviation bindings *tab* is

> *tab* `::=` id_1 `=` ty_1 `and` ... `and` id_n `=` ty_n

Then the declaration

> `lettype` *tab*

in which each ty_i must be a monotype, built from basic types and previously defined types, allows you to introduce new names id_i for the types ty_i. Within the scope of the declaration the expression e`:`id_i behaves exactly like e`:`ty_i, and the type ty_i will always be printed as id_i.

One aspect of such type abbreviations should be emphasized. Suppose for the rational numbers you declare `lettype rat = int # int;;` and set up the standard operations on rationals. Within the scope of this declaration *any* expression of type `int # int` will be treated as though it had type `rat`, and this could be not only confusing but also incorrect (in which case it ought to cause a type failure). If you wish to introduce the type `rat`, isomorphic to `int # int` but not matching it for type checking purposes, then you should use abstract types.

B.5.5 Abstract types

As with concrete types, abstract type constructors may be introduced by a declaration in which type variables are used as dummy arguments (or formal parameters) of the operators. The syntax of abstract type bindings *ab* is

> *ab* `::=` $vtyarg_1$ $tycon_1$ `=` ty_1 `and` ...`and` $vtyarg_n$ $tycon_n$ `=` ty_n `with` *b*

where each $vtyarg_i$ must contain no type variable more than once, and all the type variables in ty_i must occur in $vtyarg_i$. An abstract type declaration takes the form

> {`abstype`|`absrectype`} *ab*

The declaration introduces a set of type operators, and also incorporates a normal binding *b* (treated like `let`) of ML identifiers. Throughout the scope of the abstract type declaration the type operators and ML identifiers are both available, but it is only within *b* that the *representation* of the type operators (as declared in terms of other operators) is available. In an abstract type declaration

> `abs`{`rec`}`type` $vtyarg_1$ id_1 `=` ty_1 `and` ...`and` $vtyarg_n$ id_n `=` ty_n `with` *b*

the sense in which the representation of each id_i is available only within *b* is as follows: the isomorphism between objects of types ty_i and $vtyarg_i$ id_i is available (only in *b*) via a pair of implicitly declared polymorphic functions

> `abs_`id_i `:` ty_i `->` $vtyarg_i$ id_i
> `rep_`id_i `:` $vtyarg_i$ id_i `->` ty_i

which are to be used as coercions between the abstract types and their representations. Thus in the simple case `abstype` a`=` ty `with` x`=`e' `in` e the scope of a is e' and e, the scope of `abs_`a and `rep_`a is e', and the scope of x is e.

As an illustration, consider the definition of the type `rat` of rational numbers, represented by pairs of integers, together with operations `plus` and `times` and the conversion functions

```
inttorat : int->rat
rattoint : rat->int
```

Since rat is a nullary type operation, no type variables are involved, and rat can be defined by:

```
abstype rat = int# int
   with plus(x,y) = (abs_rat(x1*y2+x2*y1, x2*y2)
                     where x1,x2 = rep_rat x
                       and y1,y2 = rep_rat y    )
   and times(x,y) = (abs_rat(x1*y1, x2*y2)
                     where x1,x2 = rep_rat x
                       and y1,y2 = rep_rat y    )
   and inttorat n = abs_rat(n,1)
   and rattoint x = ((x1/x2)*x2=x1 => x1/x2 | failwith `rattoint`
                     where x1,x2 = rep_rat x    ) ;;
```

Most abstract type declarations are probably used at top-level, so that their scope is the remainder of the top-level program. But for non-top-level declarations, a simple constraint ensures that a value of abstract type cannot exist except during the execution of phrases within the scope of the type declaration. In the expression

abs{rec}type $vtyarg_1$ id_1 = ty_1 and ...and $vtyarg_n$ id_n = ty_n with b in e

the type of e, and the types of any non-local assignments within b and e, must not involve any of the id_i.

Finally, in keeping with the abstract nature of objects of abstract type, the value of a top-level expression of abstract type is printed as a dash, - , as functional values are. Users who wish to 'see' such an object should declare a coercion function in the 'with' part of the type declaration, to yield a suitable concrete representation of the abstract objects.

B.6 Primitive ML Identifier Bindings

The primitive ML identifier bindings are described in this Section. Some useful derived functions are in Section B.7. The primitive bindings are of two kinds:

- ordinary bindings;
- dollared bindings (which are preceded by $) having prefix or infix status.

The description of the ML value to which an identifier is bound is omitted if the semantics is clear from the identifier name and type given. For those functions whose application may fail, the failure string is the function identifier.

Predeclared identifiers are not regarded as constants of the language. As with all other ML identifiers, the user is free to rebind them, by `let`, `letref`, etc., but note that in the case of infix or prefix operators rebinding the dollared operator will affect even its non-dollared uses. Predeclared bindings are to be understood as if they had been bound by `let`, rather than by `letref`. In particular, therefore, none of them can be changed by assignment (except, of course, within the scope of a rebinding of the identifier by a `letref`-declaration).

B.6.1 Predeclared ordinary identifiers

```
    fst  : * #  ** -> *
    snd  : * #  ** -> **                      inl  : * -> * + **
                                              inr  : ** -> * + **
    null : * list -> bool                     outl : * + ** -> *
    hd   : * list -> *                        outr : * + ** -> **
    tl   : * list -> * list                   isl  : * + ** -> bool
```

The functions hd and tl fail if their argument is an empty list. The functions outl and outr fail if their arguments are not in the left or right summand, respectively. A function isr is not provided because it is just the complement of isl.

```
    explode : tok -> tok list
    implode : tok list -> tok
```

The function explode maps a token into the list of its single character tokens in order. The function implode maps a list of single character tokens (fails if any token is not of length one) into the token obtained by concatenating these characters. For example:

```
# explode `whosit`;;                                                    1
[`w`; `h`; `o`; `s`; `i`; `t`] : tok list

# implode [`c`;`a`;`t`];;
`cat` : tok

# implode [`cd`;`ab`;`tu`];;
evaluation failed     implode
```

```
    int_to_char : int -> char
    char_to_int : char -> int
```

The function int_to_char on argument i returns the ith character in NUPRL's font. The integer i must be non-negative and less than 256. For arguments less than 128 the integer-character correspondence is the same as in ASCII. The function char_to_int returns integer code of its argument, which must be a one-character token.

```
    string_to_toks : string -> tok list
    toks_to_string : tok list -> string
```

These functions are similar to explode and implode except that they work on strings rather than tokens.

```
    int_to_tok : int -> tok
    tok_to_int : tok -> int
```

These are bound to the obvious type coercion functions, with tok_to_int failing if its argument is not a non-negative integer token.

```
    ml_curried_infix : tok -> unit
    ml_paired_infix  : tok -> unit
```

The functions ml_curried_infix and ml_paired_infix declare their argument tokens to the ML parser as having *infix status*. Infixed functions can either be curried or take a pair as an argument. For example, after executing

```
ml_paired_infix `plus`;;  let x plus y = x+y;;
```

`1 plus 2` is synonymous with `$plus(1,2)` and after executing

```
ml_curried_infix `plus` ;;  let x plus y = x+y ;;
```

`1 plus 2` is synonymous with `$plus 1 2`.[4]

B.6.2 Predeclared dollared identifiers

The following prefix and infix operators are provided as primitives (where the dollar symbol is omitted from the table; the constants are `$do`, and so on):

```
do          : * -> void
not         : bool -> bool
*, /, +, -  : int #  int -> int
>, <        : int #  int -> bool
 =          : * #  * -> bool
 @          : * list #  * list -> * list
 .          : * #  * list -> * list
```

Clarifying remarks:

- `$do` is equivalent to `\x.()`. `do` e evaluates e for its side effects.
- `/` returns the integer part of the result of a division, for example

  ```
  $/(7,3) = 7/3 = 2
  $/(-7,3) = -7/3 = -2
  ```

  The failure token for division by zero is `` `div` ``.
- `-` is the binary subtraction function. Negation (unary minus) is not available as a predeclared function of ML, only as a prefix operator. Of course, the user can define negation if he or she wishes, e.g. by

  ```
  let minus x = -x
  ```

- Not all dollared infix operators are included above: `$,` is not provided since it would be equivalent (as a function) to the identity on pairs, nor is `&` as it has no corresponding call-by-value function (because e `&` e' evaluates to `false` when e does even if evaluation of e' would fail to terminate), nor is `or` analogously.
- The period symbol `.` is an infixed Lisp cons:

  x`.[`x_1`;...;`x_n`]` = `[`x`;`x_1`;...;`x_n`]`
- `=` is bound to the expected predicate for an equality test at non-function types, but is necessarily rather weak, and may give surprising results, at function types. You can be sure that semantically (i.e. extensionally) different functions are not equal, and that semantically equivalent functions are equal when they originate from the same evaluation of the same textual occurrence of a function-denoting expression; for other cases the equality of functions is unreliable (i.e. implementation dependent). For example, after the top-level declarations

[4] Only ordinary identifiers should be used as infixes; infixing other tokens may have unpredictable effects on the parser.

```
let f x = x+1 and g x = x+2;;
let f' = f and h x = f x and h' x = x+1;;
```

`f=f'` evaluates to `true` and `f=g` evaluates to `false`, but the truth values of `f=h`, `f=h'`, and `h=h'` are unreliable. Furthermore, after declaring

```
let plus = \x.\y.x+y;;
let f = plus 1 and g = plus 1;;
```

the truth value of `f=g` is also unreliable.

- `@` is a predeclared list concatenation operator; the symbol `@` has a special parser status and cannot be redeclared as a curried infix.

B.7 General Purpose and List Processing Functions

This Section describes a selection of commonly useful ML functions applicable to pairs, lists and other ML values. All the functions are definable in ML. Each function is documented by:

1. Its name and type.
2. A brief description.
3. An ML declaration defining the function (note that this is not necessarily the definition used: some of the functions are coded directly in Lisp).

Functions preceded by `$` may be used as infix operators (without the `$`), or in normal prefix form or as arguments to other functions (with the `$`).

The functions usually fail with failure string equal to their name; sometimes, however, the failure string is the one generated by the subfunction that caused the failure.

B.7.1 General purpose functions and combinators

The standard primitive combinators are: `I`, `K`, and `S`.

```
I : * -> *
K : * -> ** -> *
S : (* -> ** -> ***) -> (* -> **) -> * -> ***
```

Description: `I` $x = x$ `K` $x\ y = x$ `S` $f\ g\ x = f\ x\ (g\ x)$

Definition:

```
let I x = x

let K x y = x

let S f g x = f x (g x)
```

The derived combinators `KI` (the dual of `K`), `C` (the permutator), `W` (the duplicator), `B` (the compositor) and `CB` (which is declared to be infix) have types:

```
KI : * -> ** -> **
C  : (* -> ** -> ***) -> ** -> * -> ***
W  : (* -> * -> **) -> * -> **
B  : (* -> **) -> (*** -> *) -> *** -> **
CB : (* -> **) -> (** -> ***) -> * -> ***
```

Description:
`KI` $x\ y = y$ `C` $f\ x\ y = f\ y\ x$ `B` $f\ g\ x = f(g\ x)$ `W` $f\ x = f\ x\ x$ `CB` $f\ g\ x = g(f\ x)$

Definition:

```
let KI = K I

let C f x y = f y x

let W f x = f x x

let B f g x = f(g x)

let (f CB g) x = g(f x)
```

The next group of functions are various useful infixed function-composition operators:

```
$o  : ((* -> **) #  (*** -> *)) -> *** -> **
$#   : ((* -> **) #  (*** -> ****)) -> (* #  ***) -> (** #  ****)
$Co : ((* -> ** -> ***) #  (**** -> *)) -> ** -> **** -> ***
```

Description:
$(f$ `o` $g)\ x = f(g\ x)$
$(f$ `#` $g)(x,y) = (f\ x,\ g\ y)$
$(f$ `Co` $g)\ x\ y =$ `C`$(f$ `o` $g)\ x\ y = f\ (g\ y)\ x$

Definition:

```
ml_paired_infix `o`
let (f o g) x = f(g x)

ml_paired_infix `#`
let (f # g)(x,y) = (f x, g y)

ml_paired_infix `Co`
let (f Co g) x y = f (g y) x
```

The following two functions convert between curried and uncurried versions of a binary function.

```
curry   : ((* #  **) -> ***) -> (* -> ** -> ***)
uncurry : (* -> ** -> ***) -> ((* #  **) -> ***)
```

Description: `curry` $f\ x\ y = f(x,y)$ `uncurry` $f\ (x,y) = f\ x\ y$

Definition:

```
let curry f x y = f (x,y)

let uncurry f (x,y) = f x y
```

The next function tests for failure.

```
can : (* -> **) -> * -> bool
```

Description: `can` $f\ x$ evaluates to `true` if the application of f to x succeeds; it evaluates to `false` if the evaluation fails.

Definition:

```
let can f x = (f x; true) ? false
```

The next function iterates a function a fixed number of times.

```
funpow : int -> (* -> *) -> * -> *
```

Description: `funpow` n f x applies f to x n-times: `funpow` n $f = f^n$

Definition:

```
letrec funpow n f x = if n=0 then x else funpow (n-1) f (f x)
```

B.7.2 Miscellaneous list processing functions

The function `length` computes the length of a list.

```
length : * list -> int
```

Description: `length` `[`x_1`;...;`x_n`]` $= n$

Definition:

```
letrec length = fun [] . 0 | (_.l) . 1+(length l)
```

The function `append` concatenates lists; `@` is an uncurried and infixed version of `append`.

```
append : * list -> * list -> * list
```

Description: `append` `[`x_1`;...;`x_n`]` `[`y_1`;...;`y_m`]` $=$ x_1`;...;`x_n`;`y_1`;...;`y_m`]`

Definition:

```
letrec append l1 l2 = if null l1 then l2 else hd l1.append (tl l1) l2
```

The function `el` extracts a specified element from a list. It fails if the integer argument is less than 1 or greater than the length of the list.

```
el : int -> * list -> *
```

Description: `el` i `[`x_1`;...;`x_n`]` $= x_i$

Definition:

```
letrec el i l =
  if null l or i < 1 then failwith `el`
     else if i = 1 then hd l
     else el (i-1) (tl l)
```

The functions `last` and `butlast` compute the last element of a list and all but the last element of a list. Both fail if the argument list is empty.

```
last    : * list -> *
butlast : * list -> * list
```

Description: `last` $[x_1;\ldots;x_n] = x_n$ `butlast` $[x_1;\ldots;x_n] = [x_1;\ldots;x_{n-1}]$

Definition:

```
letrec last l = last (tl l) ? hd l ? failwith `last`

letrec butlast l =
  if null (tl l) then [] else (hd l).(butlast(tl l)) ? failwith `butlast`
```

The next function makes a list consisting of a value replicated a specified number of times. It fails if the specified number is less than zero.

```
replicate : * -> int -> * list
```

Description: `replicate` x n evaluates to $[x;\ldots;x]$, a list of length n.

Definition:

```
letrec replicate x n =
  if n < 0 then failwith `replicate`
     else if n = 0 then []
     else x . (replicate x (n-1))
```

B.7.3 List mapping and iterating functions

```
map : (* -> **) -> * list -> ** list
```

Description: `map` f l returns the list obtained by applying f to the elements of l in turn.

Definition:

```
letrec map f l = if null l then [] else f(hd l). map f (tl l)
```

The following three functions are versions of 'reduce'.

```
itlist     : (* -> ** -> **) -> * list -> ** -> **
rev_itlist : (* -> ** -> **) -> * list -> ** -> **
end_itlist : (* -> * -> *) -> * list -> *
```

Description:

`itlist` f $[x_1;x_2;\ldots;x_n]$ x $= f\ x_1\ (f\ x_2\ (\ \ldots\ (f\ x_n\ x)\ \ldots\ )) = ((f\ x_1) \circ (f\ x_2) \circ \ldots \circ (f\ x_n))\ x$

`rev_itlist` f $[x_1;\ldots;x_{n-1};x_n]$ x $= f\ x_n\ (f\ x_{n-1}\ (\ \ldots\ (f\ x_1\ x)\ \ldots\ )) = ((f\ x_n) \circ (f\ x_{n-1}) \circ \ldots \circ (f\ x_1))\ x$

`end_itlist` f $[x_1;x_2;\ldots;x_{n-1};x_n]$ $= f\ x_1\ (f\ x_2\ (\ \ldots\ (f\ x_{n-1}\ x_n)\ \ldots\ )) = ((f\ x_1) \circ (f\ x_2) \circ \ldots \circ (f\ x_{n-1}))\ x_n$

Definition:

```
letrec itlist f l x =
   if null l then x else f (hd l) (itlist f (tl l) x)

letrec rev_itlist f l x =
   if null l then x else rev_itlist f (tl l) (f (hd l) x)

let end_itlist ff l =
   if null l then failwith `end_itlist`
             else (let last.rest = rev l in  rev_itlist ff rest last)
```

or, equivalently:

```
letrec itlist f     = fun [] . I | (y.l) . \x. f y (itlist f l x)

letrec rev_itlist f = fun [] . I | (y.l) . \x. rev_itlist f l (f y x)
```

B.7.4 List searching functions

The functions described in this section search lists for elements with various properties. Those functions that return elements fail if no such element is found; those that return booleans never fail (`false` is returned if the element is not found).

```
find    : (* -> bool) -> * list -> *
tryfind : (* -> **) -> * list -> **
```

Description: `find` *p l* returns the first element of *l* that satisfies the predicate *p*. `tryfind` *f l* returns the result of applying *f* to the first member of *l* for which the application of *f* succeeds.

Definition:

```
letrec find p    = fun []    . failwith `find`
                     | (x.l) . if p x then x else find p l

letrec tryfind f = fun []    . failwith `tryfind`
                     | (x.l) . (f x ? tryfind f l)
```

The next two functions are analogous to the quantifiers ∃ and ∀.

```
exists : (* -> bool) -> * list -> bool
forall : (* -> bool) -> * list -> bool
```

Description: `exists` *p l* applies *p* to the elements of *l* in order until one is found which satisfies *p*, or until the list is exhausted, returning `true` or `false` accordingly; `forall` is the dual.

Definition:

```
let exists p l = can (find p) l

let forall p l = not(exists ($not o p) l)
```

The next function tests for membership of a list.

```
mem : * -> * list -> bool
```

Description: `mem` *x l* returns `true` if some element of *l* is equal to *x*, otherwise it returns `false`.

Definition:

```
let mem = exists o (curry $=)
```

The following two functions are ML versions of Lisp's `assoc`.

```
assoc     : * -> (* #  **) list -> (* #  **)
rev_assoc : * -> (** #  *) list -> (** #  *)
```

Description: `assoc` x l searches a list l of pairs for one whose first component is equal to x, returning the first pair found as result; similarly, `rev_assoc` y l searches for a pair whose second component is equal to y. For example:

```
# assoc 2 [(1,4);(3,2);(2,5);(2,6)];;                                   1
(2, 5) : (int #  int)

# rev_assoc 2 [(1,4);(3,2);(2,5);(2,6)];;
(3, 2) : (int #  int)
```

Definition:

```
let assoc x     = find (\(x',y). x=x')

let rev_assoc y = find (\(x,y'). y=y')
```

B.7.5 List transforming functions

The next function reverses a list:

```
rev : * list -> * list
```

Description: `rev` $[x_1;\ldots;x_n] = [x_n;\ldots;x_1]$

Definition:

```
let rev = rev1 []
   whererec rev1 l = fun [] . l | (x.l') . rev1 (x.l) l'
```

The following two functions filter a list to the sublist of elements satisfying a predicate.

```
filter    : (* -> bool) -> * list -> * list
mapfilter : (* -> **) -> * list -> * list
```

Description: `filter` p l applies p to every element of l, returning a list of those that satisfy p; evaluating `mapfilter` f l applies f to every element of l, returning a list of results for those elements for which application of f succeeds.

Definition:

```
letrec filter p    = fun []    . []
                       | (x.l) . if p x then (x.filter p l) else filter p l

letrec mapfilter f = fun []    . []
                       | (x.l) . let l' = mapfilter f l in (f x).l' ? l'
```

The following three functions break-up lists.

```
remove    : (* -> bool) -> * list -> (* #  * list)
partition : (* -> bool) -> * list -> (* list #  * list)
chop_list : int -> * list -> (* list #  * list)
```

Description: `remove` p l separates from the rest of l the first element that satisfies the predicate p; it fails if no element satisfies the predicate. `partition` p l returns a pair of lists. The first list contains the elements of l which satisfy p. The second list contains all the other elements of l.

$$\texttt{chop\_list}\ i\ \texttt{[}x_1\texttt{;}\ldots\texttt{;}x_n\texttt{]} = \texttt{[}x_1\texttt{;}\ldots\texttt{;}x_i\texttt{]}\texttt{,}\texttt{[}x_{i+1}\texttt{;}\ldots\texttt{;}x_n\texttt{]}$$

`chop_list` fails if i is negative or greater than n.

Definition:

```
letrec remove p l =
  if p (hd l) then (hd l, tl l)
  else (I # (\ ((hd l) . r))) (remove p (tl l))

let partition p l =
  itlist (\a (yes,no). if p a then ((a.yes),no) else (yes,(a.no))) l ([],[])

letrec chop_list i l =
  if i = 0 then ([],l)
  else (let l1,l2 = chop_list (i-1) (tl l) in  hd l . l1 , l2)
  ? failwith `chop_list`
```

The next function flattens a list of lists:

```
flat : * list list -> * list
```

Description: $\texttt{flat}\ \texttt{[[}l_{11}\texttt{;}\ldots\texttt{;}l_{1m_1}\texttt{];}\ \texttt{[}l_{21}\texttt{;}\ldots\texttt{;}l_{2m_2}\texttt{];}\ \ldots\texttt{[}l_{n1}\texttt{;}\ldots\texttt{;}l_{nm_n}\texttt{]]}$
$= \texttt{[}l_{11}\texttt{;}\ldots\texttt{;}l_{1m_1}\texttt{;}\ l_{21}\texttt{;}\ldots\texttt{;}l_{2m_2}\texttt{;}\ \ldots l_{n1}\texttt{;}\ldots\texttt{;}l_{nm_n}\texttt{]}$

Definition:

```
letrec flat = fun [] . [] | (x.l) . x@(flat l)
```

The next two functions 'zip' and 'unzip' between lists of pairs and pairs of lists.

```
combine : (* list #  ** list) -> (* #  **) list
split   : (* #  **) list -> (* list #  ** list)
```

Description:
$\texttt{combine}\ \texttt{[}x_1\texttt{;}\ldots\texttt{;}x_n\texttt{]}\ \texttt{[}y_1\texttt{;}\ldots\texttt{;}y_n\texttt{]} = \texttt{[(}x_1\texttt{,}y_1\texttt{);}\ldots\texttt{;(}x_n\texttt{,}y_n\texttt{)]}$
$\texttt{split}\ \texttt{[(}x_1\texttt{,}y_1\texttt{);}\ldots\texttt{;(}x_n\texttt{,}y_n\texttt{)]} = \texttt{[}x_1\texttt{;}\ldots\texttt{;}x_n\texttt{]}\texttt{,}\texttt{[}y_1\texttt{;}\ldots\texttt{;}y_n\texttt{]}$

Definition:

```
letrec combine = fun ([],[])          . []
                  |  ((x.lx),(y.ly)) . ((x,y).combine(lx,ly))
                  |  _                . failwith `combine`

letrec split   = fun []         . ([],[])
                  |  ((x,y).l) . let lx,ly = split l in (x.lx,y.ly)
```

B.7.6 Functions for lists representing sets

The following functions behave like the corresponding set-theoretic operations on sets (represented as lists without repetitions).

```
intersect : * list -> * list -> * list
subtract  : * list -> * list -> * list
union     : * list -> * list -> * list
```

Description: intersect l_1 $l_2 = l_1 \cap l_2$ subtract l_1 $l_2 = l_1 - l_2$ union l_1 $l_2 = l_1 \cup l_2$

Definition:

```
let intersect l1 l2 = filter (\x. mem x l2) l1

let subtract l1 l2  = filter (\x. not(mem x l2)) l1

let union l1 l2     = l1 @ subtract l2 l1
```

There are also functions to test if a list is a set, remove duplicates from a list and test two lists for set equality.

```
distinct  : * list -> bool
setify    : * list -> * list
set_equal : * list -> * list -> bool
```

Description: distinct l returns true if all the elements of l are distinct; otherwise it returns false. setify l removes repeated elements from l, leaving the last occurrence of each duplicate in the list. set_equal l_1 l_2 returns true if every element of l_1 appears in l_2 and every element of l_2 appears in l_1; otherwise it returns false.

Definition:

```
letrec distinct l =
 (null l) or (not (mem (hd l) (tl l)) & distinct (tl l))

let setify l = itlist (\a s. if mem a s then s else a.s) l []

let set_equal l1 l2 = (subtract l1 l2 = []) & (subtract l2 l1 = [])
```

B.7.7 Miscellaneous string processing functions

The following functions split strings into 'words'; words2 uses a user supplied separator, while words uses space and carriage-return as separators.

```
words2 : string -> string -> string list
words  : string -> string list
```

Description: words2 \`$c$\` \`$s_1 c s_2 c \dots c s_n$\` = [\`$s_1$\`;\`$s_2$\`; ...;\`$s_n$\`]

words \`$s_1$ $s_2$ ... $s_n$\` = [\`$s_1$\`;\`$s_2$\`; ...;\`$s_n$\`]

Definition:

```
let words2 sep string =
 snd (itlist (\ch (chs,tokl).
                if ch = sep then
                   if null chs then [],tokl
                   else [], (implode chs . tokl)
                else (ch.chs), tokl)
            (sep . explode string)
            ([],[]))

let word_separators = [` `;`\L`]
let words string =
 snd (itlist (\ch (chs,tokl).
                if mem ch word_separators then
                   if null chs then [],tokl
                   else [], (implode chs . tokl)
                else (ch.chs), tokl)
            (` ` . explode string)
            ([],[]))
```

The next three functions (the second of which is an infixed version of the first) are string concatenation operators.

```
concat  : string -> string -> string
$^      : string -> string -> string
concatl : string list -> string
```

Description: `concat` concatenates two strings, `$^` is an infixed version of `concat`, and `concatl` concatenates all the strings in a list of strings.

Definition:

```
let concat s1 s2 = implode(explode s1 @ explode s2)

ml_curried_infix `^`
let s1 ^ s2 = concat s1 s2

let concatl sl = implode(itlist append (map explode sl) [])
```

B.7.8 Failure handling functions

The failure handling functions described here are useful for writing code that fails with a backtrace.

```
set_fail_prefix : string -> (* -> **) -> * -> **
set_fail        : string -> (* -> **) -> * -> **
```

Description: `set_fail_prefix` *s f x* applies f to x and returns the result of the application if it is successful; if the application fails then the string s is concatenated to the failure string and the resulting string propagated as the new failure string.
`set_fail` *s f x* applies f to x and returns the result of the application if it is successful; if the application fails then the string s is propagated as the new failure string.

Definition:

```
let set_fail_prefix s f x = f x ?\s' failwith(concatl[s;`--`;s'])

let set_fail s f x        = f x ? failwith s
```