

Appendix A

Standard ML crash course

This chapter¹ introduces parts of the Standard ML language as used in this book. The many textbooks on Standard ML include Paulson's [75] and Hansen and Rischel's [38]. There is also a formal Definition [68]. The examples below were executed with Moscow ML [72], but should work with other interactive implementations, such as Standard of New Jersey (SML/NJ) [94].

A.1 Getting started

The Moscow ML interactive system is started by typing `mosml` at the shell prompt. It allows you to enter declarations and evaluate expressions:

```
$ mosml
Moscow ML version 2.01 (January 2004)
Enter 'quit();' to quit.
- fun fac n = if n = 0 then 1 else n * fac (n-1);
> val fac = fn : int -> int
- fac 10;
> val it = 3628800 : int
```

Text following a dash (-) is entered by the user; text following an angle symbol (>) is the ML system's response. You can quit Moscow ML by typing `'quit();'` or control-D (under Unix) or control-Z followed by newline (under MS Windows). See the Moscow ML Owner's Manual [86] for more detailed information about the Moscow ML interactive system, compiler, and tools.

Type `help "lib"`; for an overview of built-in function libraries, and e.g. `help "Array"`; for help on functions in the Array library. See also the Moscow ML Library Documentation [85] and the Standard ML Basis Library [33].

¹From Peter Sestoft: *Programming Language Concepts*. Draft 0.26 of 2004-02-01. Royal Veterinary and Agricultural University, and IT University of Copenhagen, Denmark. Copyright © 2004.

A.2 Expressions, declarations and types

Standard ML is a functional language: a computation is performed by evaluating an *expression* such as `3+4`. If you enter an expression in the interactive system, followed by a semicolon (`;`) and a newline, it will be evaluated:

```
- 3+4;
> val it = 7 : int
```

The system responds with the value (7) as well as the type (`int`, for integer number) of the expression.

A *declaration* `val v = e` introduces a variable `v` whose value is the result of evaluating `e`. For instance, this declaration introduces variable `res`:

```
- val res = 3+4;
> val res = 7 : int
```

After the declaration one may use `res` in expressions:

```
- res * 2;
> val it = 14 : int
```

A.2.1 Arithmetic and logical expressions

Expressions are built from constants such as `2`, variables such as `res`, and operators such as multiplication (`*`). Figure A.1 summarizes predefined Standard ML operators.

In addition, expressions may involve functions, such as the predefined function `sqrt` from the `Math` library. Function `Math.sqrt` computes the square root of a floating-point number, which has type `real`. We can compute the square root of `2.0` like this:

```
- val y = Math.sqrt 2.0;
> val y = 1.41421356237 : real
```

Floating-point constants must be written with a decimal point (`0.2`) or in scientific notation (`2E~1`) to distinguish them from integer constants.

For a list of all library functions, use the built-in `help` function, or see the *Moscow ML Library Documentation* [85], which is available also online.

Logical expressions have type `bool`:

```
- val large = 10 < res;
> val large = false : bool
```

Logical expressions can be combined using logical ‘and’ (conjunction), written `andalso`, and logical ‘or’ (disjunction), written `orelse`. Like the corresponding operators (`&&` and `||`) of C/C++/Java/C# they are sequential, so that `andalso` will evaluate its right operand only if the left operand is `true` (and dually for `orelse`):

```
- y > 0.0 andalso 1.0/y > 7.0;
> val it = false : bool
```

Logical expressions are typically used in *conditional expressions*, written `if e1 then e2 else e3`, which correspond to `(e1 ? e2 : e3)` in C/C++/Java/C#:

```
- if 3 < 4 then 117 else 118;
> val it = 117 : int
```

Operator	Type	Meaning
!	'a ref -> 'a	Dereference
f e		Function application
/	real * real -> real	Quotient
div	int * int -> int	Quotient (round down)
mod	int * int -> int	Remainder (of div)
*	num * num -> num	Product
+	num * num -> num	Sum
-	num * num -> num	Difference
^	string * string -> string	Concatenate
::	'a * 'a list -> 'a list	Cons onto list (right-assoc.)
@	'a list * 'a list -> 'a list	Append lists (right-assoc.)
=	'a * 'a -> bool	Equal to
<>	'a * 'a -> bool	Not equal to
<	numtxt * numtxt -> bool	Less than
<=	numtxt * numtxt -> bool	Less than or equal to
>	numtxt * numtxt -> bool	Greater than
>=	numtxt * numtxt -> bool	Greater than or equal to
:=	'a ref * 'a -> unit	Reference assignment
o	('b->'c)*('a->'b)->('a->'c)	Function composition
before	'a * 'b -> 'a	Return first argument

Figure A.1: Standard ML operators grouped according to precedence. Operators at the top have high precedence (bind strongly). For overloaded operators, `num` means `int` or `real`, and the type `numtxt` means `num` or `string` or `char`. All operators are left-associative, except `::` and `@`.

A.2.2 String values and operators

A text string has type `string`. A string constant is written within double quotes (`"`). The string concatenation operator (`^`) constructs a new string by concatenating two strings:

```
- val title = "Professor";
> val title = "Professor" : string
- val name = "Lauesen";
> val name = "Lauesen" : string
- val junkmail = "Dear " ^ title ^ " " ^ name ^ ", You have won $$$!";
> val junkmail = "Dear Professor Lauesen, You have won $$$!" : string
```

The string function `size` returns the length (number of characters) of a string:

```
- val n = size junkmail;
> val n = 41 : int
```

A.2.3 Types and type errors

Every expression has a type, and the compiler checks that operators and functions are applied only to expressions of the correct type. There are no implicit type conversions. For instance, `Math.sqrt` expects an argument of type `real` and thus cannot be applied to the argument expression `2`, which has type `int`. Standard ML types are summarized in Figure A.2; see also Section A.9. The compiler complains in case of type errors, and refuses to compile the expression:

```
- Math.sqrt 2;
! Toplevel input:
! Math.sqrt 2;
!      ^
! Type clash: expression of type
!   int
! cannot have type
!   real
```

The error message points to the argument expression `2` as the culprit and explains that it has type `int` which is not the same as `real`. It is up to the reader to conclude why an expression of type `real` was expected at that point.

Some arithmetic operators and comparison operators are *overloaded*, as indicated in Figure A.1. For instance, the plus operator (`+`) can be used to add two expressions of type `int` or two expressions of type `real`, but not to add an `int` and a `real`. Overloaded operators default to `int` when there are no `real` or `string` or `char` arguments.

Type	Meaning	Examples
Primitive types		
int	Integer number	0, 12, ~12
real	Floating-point number	0.0, 12.0, ~12.1, 3E-6
bool	Logical	true, false
string	String	"A", "", "den Haag"
char	Character	#"A", #" "
exn	Exception	Overflow, Fail "index"
Functions (Sections A.2.4–A.2.6, A.8.3, A.10)		
real -> real	Function from int to bool	Math.sqrt
real -> bool	Function from int to bool	isLarge
int * int -> int	Function taking int pair	addp
int -> int -> int	Function taking two ints	addc
Pairs and tuples (Section A.4)		
unit	Empty tuple	()
int * int	Pair of integers	(2, 3)
int * bool	Pair of int and bool	(2100, false)
int * bool * real	Three-tuple	(2, true, 2.1)
Lists (Section A.5)		
int list	List of integers	[7, 9, 13]
bool list	List of Booleans	[false, true, true]
string list	List of strings	["foo", "bar"]
Records (Section A.6)		
{x : int, y : int}	Record of two ints	{x=2, y=3}
{y:int, leap:bool}	Record of int and bool	{n=2100, leap=false}

Figure A.2: Some monomorphic Standard ML types

A.2.4 Function declarations

A *function declaration* begins with the keyword `fun`. The example below defines a function `circleArea` that takes one argument `r` and returns the value of `Math.pi * r * r`. The function can be applied (called) simply by writing the function name before an argument expression:

```
- fun circleArea r = Math.pi * r * r;
> val circleArea = fn : real -> real
- val a = circleArea 10.0;
> val a = 314.159265359 : real
```

The system infers that the type of the function is `real -> real`. That is, the function takes a floating-point number as argument and returns a floating-point number as result.

Similarly, this declaration defines a function `doubl` from `real` to `real`:

```
- fun doubl x = 2.0 * x;
> val doubl = fn : real -> real
- doubl 3.5;
> val it = 7.0 : real
```

A function may take any type of argument and produce any type of result. The function `junkmail` takes two arguments of type `string` and produces a result of type `string`:

```
- fun junkmail title name =
  "Dear " ^ title ^ " " ^ name ^ ", You have won $$$!";
> val junkmail = fn : string -> string -> string
- junkmail "Director" "Tofte";
> val it = "Dear Director Tofte, You have won $$$!" : string
```

A.2.5 Recursive function declarations

A function may call any function, including itself. That is, a function declaration may be *recursive*:

```
- fun fac n = if n=0 then 1 else n * fac(n-1);
> val fac = fn : int -> int
- fac 7;
> val it = 5040 : int
```

If two functions need to call each other by so-called *mutual recursion*, they must be declared in one declaration beginning with `fun` and separating the declarations by `and`:

```

- fun even n = if n=0 then true else odd (n-1)
  and odd n = if n=0 then false else even (n-1);
> val even = fn : int -> bool
  val odd = fn : int -> bool

```

A.2.6 Type constraints

As you can see from the examples, the compiler automatically infers the type of a declared variable or function. Sometimes it is good to use an explicit *type constraint* for documentation. For instance, we may explicitly require that the function's argument *x* has type *real*, and that the function's result has type *bool*:

```

- fun isLarge (x : real) : bool = 10.0 < x;
> val isLarge = fn : real -> bool
- isLarge 89.0;
> val it = true : bool

```

If the type constraint is wrong, the compiler refuses to compile the declaration. A type constraint cannot be used to convert a value from one type to another as in C. Thus to convert an *int* to a *real*, you must use the function `real : int -> real`. Similarly, to round a *real* to an *int*, you must use the function `round : real -> int`.

A.2.7 The scope of a binding

The scope of a variable binding is that part of the program in which it is visible. In a `let`-expression `let dec in exp end`, the declaration *dec* may introduce a variable binding whose scope is the expression *exp* only, without disturbing any existing variables, not even with the same name. For instance:

```

- val x = 5; (* old x is 5 : int *)
> val x = 5 : int
- let val x = 3 < 4 (* new x is true : bool *)
  in
  if x then 117 else 118 (* using new x *)
  end;
> val it = 117 : int
- x; (* old x is still 5 *)
> val it = 5 : int

```

To introduce a variable binding with limited scope in a declaration, use a local-declaration `local dec in dec end`. The difference between `let` and `local` is that the former is an expression (which has a value) whereas the latter is a declaration (which binds one or more names):

198 *Pairs and tuples*

```
- val x = 5; (* old x is 5 : int *)
> val x = 5 : int
- local val x = 3 < 4 (* new x is true : bool *)
  in
    val z = if x then 117 else 118 (* using new x *)
  end;
> val z = 117 : int
- x; (* old x is still 5 *)
> val it = 5 : int
```

A.3 Pattern matching

Functions can be defined by case, using *pattern matching*. The patterns are tried in order from top to bottom, and the right-hand side corresponding to the first matching pattern is evaluated. For instance, calling `fac 7` will find that 7 does not match the pattern 0, but it does match the variable pattern `n`, so `n` gets bound to 7 and the right-hand side `7 * fac(7-1)` gets evaluated:

```
- fun fac 0 = 1
  | fac n = n * fac(n-1);
> val fac = fn : int -> int
- fac 7;
> val it = 5040 : int
```

Patterns can be used also in connection in case-expressions, similar to but much more powerful than switch-statements in C/C++/Java/C#:

```
- fun fac n =
  case n of
    0 => 1
  | _ => n * fac(n-1);
> val fac = fn : int -> int
```

The above example uses the *wildcard pattern* (`_`) which matches any value.

A.4 Pairs and tuples

A *tuple* has a fixed number of components, all of which may be of different types. A *pair* is a tuple with two components. For instance, a pair of integers is written simply `(2, 3)`, and its type is `int * int`:

```
- val p = (2, 3);
> val p = (2, 3) : int * int
```

```
- val w = (2, true, 3.4, "blah");
> val w = (2, true, 3.4, "blah") : int * bool * real * string
```

The operators #1, #2, ... select the first, second, ... component of a tuple:

```
- #2 w;
> val it = true : bool
- #4 w;
> val it = "blah" : string
```

A function may take a pair as an argument:

```
- fun add (x, y) = x + y;
> val add = fn : int * int -> int
- add (2, 3);
> val it = 5 : int
```

Function `add` takes only one argument, but that argument is a pair of type `int * int`. Pairs are useful for representing values that belong together; for instance, the time of day can be represented as a pair of hours and minutes:

```
- val noon = (12, 0);
> val noon = (12, 0) : int * int
- val talk = (15, 15);
> val talk = (15, 15) : int * int
```

Pairs can be nested to any depth. For instance, a function can take a pair of pairs as argument. Note the type of function `earlier`:

```
- fun earlier ((h1, m1), (h2, m2)) =
    h1 < h2 orelse (h1 = h2 andalso m1 < m2);
> val earlier = fn : (int * int) * (int * int) -> bool
```

The empty tuple is written `()` and has type `unit`. This seemingly useless value is returned by functions that are called for their side effect only, such as `print`:

```
- print "Hello!\n";
Hello!
> val it = () : unit
```

Thus the `unit` type serves much the same purpose as the `void` return type in C/C++/Java/C#.

A.5 Lists

A *list* contains zero or more elements, all of the same type. For instance, a list may hold three integers; then it has type `int list`:

```
- val x1 = [7, 9, 13];
> val x1 = [7, 9, 13] : int list
```

The empty list is written `[]`, and the operator `::` called ‘cons’ prepends an element to an existing list. Hence this is equivalent to the above declaration:

```
- val x2 = 7 :: 9 :: 13 :: [];
> val x2 = [7, 9, 13] : int list
- val equal = (x1 = x2);
> val equal = true : bool
```

The cons operator `::` is right associative, so `7 :: 9 :: 13 :: []` reads `7 :: (9 :: (13 :: []))`, which is the same as `[7, 9, 13]`.

A list `ss` of strings can be created just as easily as a list of integers; note that the type is `string list`:

```
- val ss = ["Dear ", title, " ", name, ", You have won $$$!"];
> val ss = ["Dear ", "Professor", " ", "Lauesen", ", You have won $$$!"] :
string list
```

The elements of a list of strings can be concatenated to a single string using the `concat` function from the `String` library:

```
- val junkmail2 = String.concat ss;
> val junkmail2 = "Dear Professor Lauesen, You have won $$$!" : string
```

Functions on lists are conveniently defined using pattern matching and recursion. The `sum` function computes the sum of an integer list:

```
- fun sum [] = 0
  | sum (x::xr) = x + sum xr;
> val sum = fn : int list -> int
- val x2sum = sum x2;
> val x2sum = 29 : int
```

The `sum` function definition says: The sum of an empty list is zero. The sum of a list whose first element is `x` and whose tail is `xr`, is `x` plus the sum of `xr`.

Many other functions on lists follow the same paradigm:

```

- fun prod []      = 1
  | prod (x::xr) = x * prod xr;
> val prod = fn : int list -> int
- val x2prod = prod x2;
> val x2prod = 819 : int
- fun len []      = 0
  | len (x::xr) = 1 + len xr;
> val 'a len = fn : 'a list -> int
- val x2len = len x2;
> val x2len = 3 : int
- val sslen = len ss;
> val sslen = 5 : int

```

Note the type of `len` — since the `len` function does not use the list elements, it works on all lists regardless of the element type; see Section A.9.

The append operator (`@`) creates a new list by concatenating two given lists:

```

- val x3 = [47, 11];
> val x3 = [47, 11] : int list
- val x1x3 = x1 @ x3;
> val x1x3 = [7, 9, 13, 47, 11] : int list

```

The append operator does not copy the list elements, only the ‘spine’ of the left-hand operand `x1`, and it does not copy its right-hand operand at all. In the computer’s memory, the tail of `x1x3` is exactly the same as `x3`.

A.6 Records and labels

A *record* is basically a tuple whose components are labelled. Instead of writing a pair (`"Kasper", 886`) of a name and the associated phone number, one can use a record. This is particularly useful when there are many components:

```

- val x = { name = "Kasper", phone = 886 };
> val x = {name = "Kasper", phone = 886} : {name : string, phone : int}

```

Note how the type of a record is written, with colon (`:`) instead of equals (`=`). One can extract the components of a record using a *record component selector*:

```

- #name x;
> val it = "Kasper" : string
- #phone x;
> val it = 886 : int

```

One can also use pattern matching to extract the fields of a record:

202 *Datatypes*

```
- fun show { name, phone } = name ^ " has extension " ^ Int.toString phone;  
> val show = fn : {name : string, phone : int} -> string  
- show x;  
> val it = "Kasper has extension 886" : string
```

A.7 Exceptions

Exceptions can be declared, thrown and caught as in C++/Java/C#. In fact, the exception concept of those languages is inspired by Standard ML. An exception declaration declares an exception constructor, of type `exn`. A `raise` expression throws an exception:

```
- exception IllegalHour;  
> exn IllegalHour = IllegalHour : exn  
- fun mins h =  
    if h < 0 orelse h > 23 then raise IllegalHour  
    else h * 60;  
> val mins = fn : int -> int  
- mins 25;  
! Uncaught exception:  
! IllegalHour
```

A `handle-expression` (`e1 handle exn => e2`) evaluates `e1` and returns its value, but if `e1` throws exception `exn`, it evaluates `e2` instead. This serves the same purpose as `try-catch` in C++/Java/C#:

```
- (mins 25) handle IllegalHour => ~1;  
> val it = ~1 : int
```

A.8 Datatypes

Datatypes, sometimes called *algebraic datatypes*, are useful when data of the same type may have different numbers and types of components. For instance, a person may either be a `Student` who has a name, or a `Teacher` who has a name and a phone number. Defining a person datatype means that we can have a list of person values, regardless of whether they are `Students` or `Teachers`. (Recall that all elements of a list must have the same type).

```
- datatype person =  
    Student of string (* name *)  
    | Teacher of string * int (* name and phone no *)  
> con Student = fn : string -> person
```

```

    con Teacher = fn : string * int -> person
- val people = [Student "Niels", Teacher("Peter", 831)];
> val people = [Student "Niels", Teacher("Peter", 831)] : person list
- fun getphone (Teacher(name, phone)) = phone
  | getphone (Student name)           = raise Fail "no phone";
> val getphone = fn : person -> int
- getphone (Student "Niels");
! Uncaught exception:
! Fail "no phone"

```

A.8.1 The option datatype

A frequently used datatype is the option datatype, used to represent the presence or absence of a value.

```

- datatype intopt =
  SOME of int
  | NONE;
> con NONE = NONE : intopt
con SOME = fn : int -> intopt
- fun getphone (Teacher(name, phone)) = SOME phone
  | getphone (Student name)           = NONE;
> val getphone = fn : person -> intopt
- getphone (Student "Niels");
> val it = NONE : int option

```

In Java and C#, some methods return null to indicate the absence of a result, but that is a poor substitute for an option type, both in the case where the method should never return null, and in the case where null is a legitimate result from the method. The type inferred for function `getphone` clearly says that we cannot expect it to always return an integer, only an `intopt`, which may or may not hold an integer.

The `Option` library defines a polymorphic option datatype.

A.8.2 Binary trees represented by recursive datatypes

A datatype declaration may be recursive, which means that a value of the datatype `t` can have a component of type `t`. This can be used to represent trees and other data structures. For instance, a binary integer tree `inttree` may be defined to be either a leaf `Lf` or a branching node `Br` which holds an integer and left subtree and a right subtree:

```

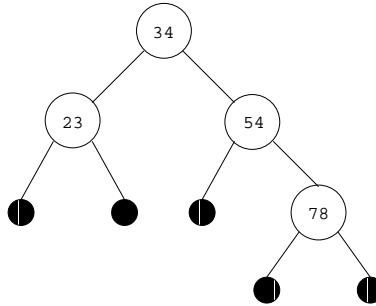
- datatype inttree =
  Lf

```

204 *Datatypes*

```
| Br of int * inttree * inttree;  
> con Br = fn : int * inttree * inttree -> inttree  
  con Lf = Lf : inttree  
  - val t1 = Br(34, Br(23, Lf, Lf), Br(54, Lf, Br(78, Lf, Lf)));  
> val t1 = Br(34, Br(23, Lf, Lf), Br(54, Lf, Br(78, Lf, Lf))) : inttree
```

The tree represented by `t1` has 34 at the root node, 23 at the root of the left subtree, etc., like this:



Functions on trees and other datatypes are conveniently defined using pattern matching and recursion. This function computes the sum of the nodes of an integer tree:

```
- fun sumtree Lf = 0  
  | sumtree (Br(v, t1, t2)) = v + sumtree t1 + sumtree t2;  
> val sumtree = fn : inttree -> int  
- val t1sum = sumtree t1;  
> val t1sum = 189 : int
```

The definition of `sumtree` reads: The sum of a leaf node is zero. The sum of a branch node `Br(v, t1, t2)` is `v` plus the sum of `t1` plus the sum of `t2`.

A.8.3 Curried functions

A function of type `int * int -> int` that takes a pair of arguments is closely related to a function of type `int -> int -> int` that takes two arguments. The latter is called a *curried* version of the former. For instance, function `addc` below is a curried version of function `addp`. Note the types of `addp` and `addc` and how the functions are applied to arguments:

```
- fun addp (x, y) = x + y;  
> val addp = fn : int * int -> int  
- fun addc x y = x + y;
```

```

> val addc = fn : int -> int -> int
- val res1 = addp(17, 25);
> val res1 = 42 : int
- val res2 = addc 17 25;
> val res2 = 42 : int

```

A major advantage of curried functions is that they can be partially applied. Applying `addc` to only one argument, 17, we obtain a new function of type `int -> int`. This new function adds 17 to its argument and can be used on as many different arguments as we like:

```

- val addSeventeen = addc 17;
> val addSeventeen = fn : int -> int
- val res3 = addSeventeen 25;
> val res3 = 42 : int
- val res4 = addSeventeen 100;
> val res4 = 117 : int

```

A.9 Type variables and polymorphic types

We saw in Section A.5 that the type of the `len` function was `'a list -> int`:

```

- fun len [] = 0
  | len (x::xr) = 1 + len xr;
> val 'a len = fn : 'a list -> int

```

The `'a` is a *type variable*. Note that the prefixed prime (`'`) is part of the type variable name `'a`. When using the `len` function, the type variable `'a` may be instantiated to any type whatsoever, and it may be instantiated to different types at different uses. Here `'a` gets instantiated to `int` and then to `string`:

```

- len [7, 9, 13];
> val it = 3 : int
- len ["Oslo", "Aarhus", "Gothenburg", "Copenhagen"];
> val it = 4 : int

```

A.9.1 Equality type variables

Let us define a function `member` to test whether a given element `x` is found in a list:

```

- fun member x [] = false
  | member x (y::yr) = x=y orelse member x yr;
> val 'a member = fn : 'a -> 'a list -> bool

```

The type variable `'a` is a special kind of type variable that can be instantiated only by a type that *admits equality*: one on which the equality predicate (`=`) can be used. Namely, for `member` to work, it must be possible to use the equality predicate to compare `x` to the list's elements.

The primitive types (`int`, `string`, ...) all admit equality. For instance, one can define a function to test whether a given day name is a workday:

```
- fun workday d = member d ["Mon", "Tue", "Wed", "Thu", "Fri"];
> val workday = fn : string -> bool
```

Some types do not admit equality, in particular function types `t -> u`. In general, it is impossible to decide whether two function declarations define the same (mathematical) function. Hence the compiler rejects this declaration:

```
- fun trig f = member f [Math.sin, Math.cos, Math.tan];
! Toplevel input:
!   fun trig f = member f [Math.sin, Math.cos, Math.tan];
!           ^^^^^^^^^
! Type clash: expression of type
!   real -> real
! cannot have equality type 'a
```

A.9.2 Polymorphic datatypes

Some data structures, such as a binary trees, have the same shape regardless of the element type. Fortunately, we can define polymorphic datatypes to represent such data structures. For instance, we can define the type of binary trees whose leaves can hold a value of type `'a` like this:

```
- datatype 'a tree =
    Lf
  | Br of 'a * 'a tree * 'a tree;
> con 'a Br = fn : 'a * 'a tree * 'a tree -> 'a tree
con 'a Lf = Lf : 'a tree
```

Compare this with the monomorphic integer tree in Section A.8.2. Values of this type can be defined exactly as before:

```
- val t1 = Br(34, Br(23, Lf, Lf), Br(54, Lf, Br(78, Lf, Lf)));
> val t1 = Br(34, Br(23, Lf, Lf), Br(54, Lf, Br(78, Lf, Lf))) : int tree
```

The type of `t1` is `int tree`, where the type variable `'a` has been instantiated to `int`.

Likewise, functions on such trees can be defined as before:

```

- fun sumtree Lf          = 0
  | sumtree (Br(v, t1, t2)) = v + sumtree t1 + sumtree t2;
> val sumtree = fn : int tree -> int
- fun count Lf           = 0
  | count (Br(_, t1, t2)) = 1 + count t1 + count t2;
> val 'a count = fn : 'a tree -> int

```

The argument type of `sumtree` is `int tree` because the function adds the node values, which must be of type `int`.

The argument type of `count` is `'a tree` because the function ignores the node values, and therefore works on an `'a tree` regardless of the node type `'a`.

A.9.3 Type abbreviations

When a type, such as `(string * int) list`, is used frequently, it is convenient to abbreviate it using a name such as `intenv`:

```

- type intenv = (string * int) list;
> type intenv = (string * int) list
- fun bind1 (env : intenv) (x : string, v : int) : intenv = (x, v) :: env;
> val bind1 = fn : (string * int) list -> string * int -> (string * int) list

```

The type declaration defines an *abbreviation*, not a new type, as can be seen from the compiler's response. This also means that the function can be applied to a perfectly ordinary list of `string * int` pairs:

```

- bind1 [("age", 41)] ("phone", 831);
> val it = [("phone", 831), ("age", 41)] : (string * int) list

```

A.10 Higher-order functions

A *higher-order function* is one that takes another function as an argument. For instance, function `map` below takes as argument a function `f` and a list, and applies `f` to all elements of the list:

```

- fun map f []          = []
  | map f (x::xr)      = f x :: map f xr;
> val ('a, 'b) map = fn : ('a -> 'b) -> 'a list -> 'b list

```

The type of `map` says that it takes as arguments a function from type `'a` to type `'b`, and a list whose elements have type `'a`, and produces a list whose elements have type `'b`. The type variables `'a` and `'b` may be independently instantiated to any types. For instance, we can define a function `doubl` of type `real -> real` and use `map` to apply that function to all elements of a list:

208 *Higher-order functions*

```
- fun doubl x = 2.0 * x;  
> val doubl = fn : real -> real  
- map doubl [4.0, 5.0, 89.0];  
> val it = [8.0, 10.0, 178.0] : real list
```

Or we may apply a function `isLarge` of type `real -> bool` (defined on page 197) to all elements of a real list:

```
- map isLarge [4.0, 5.0, 89.0];  
> val it = [false, false, true] : bool list
```

A.10.1 Anonymous functions

Sometimes it is inconvenient to introduce named auxiliary functions. In this case, one can write an anonymous function *expression* using `fn` instead of a named function *declaration* using `fun`:

```
- fn x => 2.0 * x;  
> val it = fn : real -> real
```

This is particularly useful in connection with higher-order functions such as `map`:

```
- map (fn x => 2.0 * x) [4.0, 5.0, 89.0];  
> val it = [8.0, 10.0, 178.0] : real list  
- map (fn x => 10.0 < x) [4.0, 5.0, 89.0];  
> val it = [false, false, true] : bool list
```

The function `tw` defined below takes a function `g` and an argument `x` and applies `g` twice. Using `tw` one can define a function `quad` that applies `doubl` twice, thus multiplying its argument by 4.0:

```
- fun tw g x = g (g x);  
> val 'a tw = fn : ('a -> 'a) -> 'a -> 'a  
- val quad = tw doubl;  
> val quad = fn : real -> real  
- quad 7.0;  
> val it = 28.0 : real
```

An anonymous function always takes exactly one argument. To take more than one argument, make the anonymous function return a new anonymous function that takes the next argument:

```
- fn x => fn y => x+y;  
> val it = fn : int -> int -> int
```

A.10.2 Higher-order functions on lists

Higher-order functions are particularly useful in connection with polymorphic datatypes. For instance, one can define a function `filter` that takes as argument a predicate (a function of type `'a -> bool`) and a list, and returns a list containing only those elements for which the predicate is `true`. This may be used to extract the even elements (those divisible by 2) in a list:

```
- fun filter p [] = []
  | filter p (x::xr) = if p x then x :: filter p xr else filter p xr;
> val 'a filter = fn : ('a -> bool) -> 'a list -> 'a list
- val even = filter (fn i => i mod 2 = 0) [4, 6, 5, 2, 54, 89];
> val even = [4, 6, 2, 54] : int list
```

Note that the `filter` function is polymorphic in the argument list type. The `filter` function is predefined in the `List` library. Another very general predefined polymorphic higher-order list function `foldr`, for *fold right*:

```
- fun foldr f e [] = e
  | foldr f e (x::xr) = f(x, foldr f e xr);
> val ('a, 'b) foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

An application `foldr f e xs` recursively replaces:

```
[]          by e
(x :: xr)  by f(x, xr)
```

Many other functions on lists can be defined in terms of `foldr`:

```
fun len xs = foldr (fn (_, res) => 1+res) 0 xs;
> val 'a len = fn : 'a list -> int
fun sum xs = foldr (fn (x, res) => x+res) 0 xs;
> val sum = fn : int list -> int
fun prod xs = foldr (fn (x, res) => x*res) 1 xs;
> val prod = fn : int list -> int
fun map g xs = foldr (fn (x, res) => g x :: res) [] xs;
> val ('a, 'b) map = fn : ('a -> 'b) -> 'a list -> 'b list
fun concat xss = foldr (fn (xs, res) => xs @ res) [] xss;
> val 'a concat = fn : 'a list list -> 'a list
fun filter p xs = foldr (fn (x,r) => if p x then r else x :: r) [] xs
> val 'a filter = fn : ('a -> bool) -> 'a list -> 'a list
```

The functions `map`, `filter`, `foldr` and many other on lists are predefined in the `List` library.

A.11 References

A *reference* is a handle to a *memory cell*. A reference in Standard ML is similar to a reference in Java/C# or a pointer in C/C++, but it cannot be null and the memory cell it points to cannot be uninitialized and cannot be accidentally overwritten by a memory write operation.

A new unique reference memory cell and a reference to it is created by applying the `ref` constructor to a value. Applying the dereferencing operator (`!`) to a reference gives the value in the corresponding memory cell. The value in the memory cell can be changed by applying the assignment (`:=`) operator to a reference and a new value:

```
- val r = ref 177;
> val r = ref 177 : int ref
- val v = !r;
> val v = 177 : int
- r := 288;
> val it = () : unit
- !r;
> val it = 288 : int
```

A typical use of references and memory cells is to create a sequence of distinct names or labels:

```
- val nextlab = ref ~1;
> val nextlab = ref ~1 : int ref
- fun newLabel () = (nextlab := 1 + !nextlab; "L" ^ Int.toString (!nextlab));
> val newLabel = fn : unit -> string
- newLabel();
> val it = "L0" : string
- newLabel();
> val it = "L1" : string
- newLabel();
> val it = "L2" : string
```