

Appendix A

F# crash course

This chapter¹ introduces parts of the F# programming language as used in this book. F# belongs to the ML family of programming languages, which includes classical ML from 1978, Standard ML [?] from 1986, CAML from 1985, Caml Light [60] from 1990, and OCaml [73] from 1996, where F# most resembles the latter. All of these languages are strict, mostly functional, and statically typed with parametric polymorphic type inference.

In Microsoft Visual Studio 2010, F# is included by default. If you use Visual Studio 2008, you need to download it from <http://msdn.microsoft.com/fsharp/> and install it. The examples below were executed with F# May 2009 CTP (F# 1.9.6.16) under Visual Studio 2008. You can also run F# on MacOS and Linux, using the Mono implementation: see ‘Using F# with Mono’ in the README file of the F# distribution.

A.1 What files are provided for this chapter

File	Contents
<code>appendix.fs</code>	All examples shown in this chapter

A.2 Getting started

To get the F# interactive prompt, start Visual Studio and choose View > Other Windows > F# Interactive. It allows you to enter declarations and evaluate expressions:

¹From Peter Sestoft: *Programming Language Concepts for Software Developers*. Draft 0.36 of 2009-08-21. IT University of Copenhagen, Denmark. Copyright © 2009.

```

Microsoft F# Interactive, (c) Microsoft Corporation, All Rights Reserved
F# Version 1.9.6.16, compiling for .NET Framework Version v2.0.50727
> let rec fac n = if n=0 then 1 else n * fac (n-1);;
val fac : int -> int
> fac 10;;
val it : int = 3628800
> #q;;

```

Text starting with an angle symbol (>) is entered by the user; the other lines hold the ML system's response.

A.3 Expressions, declarations and types

F# is a mostly-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 double semicolon (`;;`) and a newline, it will be evaluated:

```

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

```

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

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

```

> let res = 3+4;;
val res : int = 7

```

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

```

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

```

A.3.1 Arithmetic and logical expressions

Expressions are built from constants such as `2` and `2.0`, variables such as `res`, and operators such as multiplication (`*`). Figure A.1 summarizes predefined F# operators.

Expressions may involve functions, such as the predefined function `sqrt`. Function `sqrt` computes the square root of a floating-point number, which has type `float`, a 64-bit floating-point number. We can compute the square root of `2.0` like this:

```

> let y = sqrt 2.0;;
val y : float = 1.414213562

```

Floating-point constants must be written with a decimal point (2.0) or in scientific notation (2E0) to distinguish them from integer constants.

To get help on F#, consult <http://msdn.microsoft.com/fsharp/> where you may find the library documentation and the draft language specification [96], or see the F# Wiki at <http://strangelights.com/fsharp/Wiki/>.

You can also use (static) methods from the .NET class libraries, after opening the relevant namespaces:

```
> open System;;
> let y = Math.Sqrt 2.0;;
val y : float = 1.414213562
```

Logical expressions have type bool:

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

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

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

Logical negation is written not e :

```
> not false ;;
val it : bool = true
```

The (!) operator is used for another purpose, as described in section A.12.

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 : int = 117
```

A.3.2 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:

Operator	Type	Meaning
<code>f e</code>		Function application
<code>!</code>	<code>'a ref -> 'a</code>	Dereference
<code>-</code>	<code>num -> num</code>	Arithmetic negation
<code>**</code>	<code>float * float -> float</code>	Power
<code>/</code>	<code>float * float -> float</code>	Quotient
<code>/</code>	<code>int * int -> int</code>	Quotient, round toward 0
<code>%</code>	<code>int * int -> int</code>	Remainder of int quotient
<code>*</code>	<code>num * num -> num</code>	Product
<code>+</code>	<code>num * num -> num</code>	Sum
<code>-</code>	<code>num * num -> num</code>	Difference
<code>::</code>	<code>'a * 'a list -> 'a list</code>	Cons onto list (right-assoc.)
<code>^</code>	<code>string * string -> string</code>	Concatenate (right-assoc.)
<code>@</code>	<code>'a list * 'a list -> 'a list</code>	List append (right-assoc.)
<code>=</code>	<code>'a * 'a -> bool</code>	Equal
<code><></code>	<code>'a * 'a -> bool</code>	Not equal
<code><</code>	<code>'a * 'a -> bool</code>	Less than
<code>></code>	<code>'a * 'a -> bool</code>	Greater than
<code><=</code>	<code>'a * 'a -> bool</code>	Less than or equal
<code>>=</code>	<code>'a * 'a -> bool</code>	Greater than or equal
<code>&&</code>	<code>bool * bool -> bool</code>	Logical 'and' (short-cut)
<code>*</code>	<code>bool * bool -> bool</code>	Logical 'or' (short-cut)
<code>,</code>	<code>'a * 'b -> 'a * 'b</code>	Tuple element separator
<code>:=</code>	<code>'a ref * 'a -> unit</code>	Reference assignment

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

```

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

```

The instance property `Length` on a string returns its length (number of characters):

```

> junkmail.Length;;
val it : int = 41

```

A.3.3 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, `sqrt` expects an argument of type `float` and thus cannot be applied to the argument expression `2`, which has type `int`. Some F# types are summarized in Figure A.2; see also Section A.10. The compiler complains in case of type errors, and refuses to compile the expression:

```

> sqrt 2;;
sqrt 2;;
-----^
stdin(51,6): error FS0001: The type 'int' does not support
any operators named 'Sqrt'

```

The error message points to the argument expression `2` as the culprit and explains that it has type `int` which does not support any function `Sqrt`. It is up to the reader to infer that the solution is to write `2.0` to get a constant of type `float`.

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 `float`, but not to add an `int` and a `float`. Overloaded operators default to `int` when there are no `float` or `string` or `char` arguments.

A.3.4 Function declarations

A *function declaration* begins with the keyword `let`. 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:

Type	Meaning	Examples
Primitive types		
int	Integer number (32 bit)	0, 12, ~12
float	Floating-point number (64 bit)	0.0, 12.0, ~12.1, 3E-6
bool	Logical	true, false
string	String	"A", "", "den Haag"
char	Character	"#A", "# " "
Exception	Exception	Overflow, Fail "index"
Functions (Sections A.3.4–A.3.6, A.9.3, A.11)		
float -> float	Function from float to float	sqrt
float -> bool	Function from float to bool	isLarge
int * int -> int	Function taking int pair	addp
int -> int -> int	Function taking two ints	addc
Pairs and tuples (Section A.5)		
unit	Empty tuple	()
int * int	Pair of integers	(2, 3)
int * bool	Pair of int and bool	(2100, false)
int * bool * float	Three-tuple	(2, true, 2.1)
Lists (Section A.6)		
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.7)		
{x : int; y : int}	Record of two ints	{x=2; y=3}
{y:int; leap:bool}	Record of int and bool	{y=2100; leap=false}
References (Section A.12)		
int ref	Reference to an integer	ref 42
int list ref	Reference to a list of integers	ref [7; 9; 13]

Figure A.2: Some monomorphic F# types.

```
> let circleArea r = System.Math.PI * r * r;;
val circleArea : float -> float
> let a = circleArea 10.0;;
val a : float = 314.1592654
```

The system infers that the type of the function is `float -> float`. That is, the function takes a floating-point number as argument and returns a floating-point number as result. This is because the .NET library constant `PI` is a floating-point number (a double).

Similarly, this declaration defines a function `mul2` from `float` to `float`:

```
> let mul2 x = 2.0 * x;;
val mul2 : float -> float
> mul2 3.5;;
val it : float = 7.0
```

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

```
> let junkmail title name =
    "Dear " ^ title ^ " " ^ name ^ ", You have won $$$!";;
val junkmail : string -> string -> string
> junkmail "Vice Chancellor" "Tofte";;
val it : string = "Dear Vice Chancellor Tofte, You have won $$$!"
```

Note that F# is layout-sensitive (like a few other programming languages, such as Haskell and `C`). If the second line of the `junkmail` function declaration had had no indentation at all, then we would get a long error message (but strangely, in this particular case the declaration would still be accepted):

```
> let junkmail title name =
    "Dear " ^ title ^ " " ^ name ^ ", You have won $$$!";;
    "Dear " ^ title ^ " " ^ name ^ ", You have won $$$!";;
    ^^^^^^^
stdin(89,1): warning FS0058: possible incorrect indentation [...more...]
val junkmail : string -> string -> string
```

A.3.5 Recursive function declarations

A function may call any function, including itself; but then its declaration must start with `let rec` instead of `let`. That is, a function declaration may be *recursive*:

212 *Expressions, declarations and types*

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

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

```
> let rec 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 : int -> bool
val odd : int -> bool
```

A.3.6 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 `float`, and that the function's result has type `bool`:

```
> let isLarge (x : float) : bool = 10.0 < x;;
val isLarge : float -> bool
> isLarge 89.0;;
val it : bool = true
```

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 `float`, you must use function `float : int -> float`. Similarly, to convert a `float` to an `int`, use a function such as `floor`, `round` or `ceil`, all of which have type `float -> int`.

A.3.7 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:

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

```

val it : int = 117
> x;;
(* old x is still 5 *)
val it : int = 5

```

A.4 Pattern matching

Like all languages in the ML family, but unlike most other programming languages, F# supports *pattern matching*. Pattern matching is performed by a `match e with ...-expression`, which consists of the expression `e` whose value must be matched, and a list (...) of match branches. For instance, the factorial function can be defined by pattern matching on the argument `n`, like this:

```

> let rec fac n =
    match n with
    | 0 -> 1
    | _ -> n * fac(n-1);;
val fac : int -> int

```

The patterns in a `match` 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 *wildcard pattern* (`_`) which matches any value. so the right-hand side `n * fac(n-1)` gets evaluated.

A slightly more compact notation for one-argument function definitions uses the `function` keyword, which combines parameter binding with pattern matching:

```

> let rec fac =
    function
    | 0 -> 1
    | n -> n * fac(n-1);;
val fac : int -> int

```

Pattern matching in the ML languages is similar to but much more powerful than `switch`-statements in C/C++/Java/C#, because matches can involve also tuple patterns (section A.5) and algebraic datatype constructor patterns (section A.9) and any combination of these. This makes the ML-style languages particularly useful for writing programs that process other programs, such as interpreters, compilers, program analysers and program transformers.

Moreover, ML-style languages, including F#, usually require the compiler to detect both *incomplete* matches and *redundant* matches; that is, matches that either leave some cases uncovered, or that have some branches that are not usable:

214 *Pairs and tuples*

```
> let bad1 n =
  match n with
  | 0 -> 1
  | 1 -> 2;;
warning FS0025: Incomplete pattern matches on this expression.
For example, the value '2' may not be covered by the pattern(s).
> let bad2 n =
  match n with
  | _ -> 1
  | 1 -> 2;;

  | 1 -> 2;;
  -----^
warning FS0026: This rule will never be matched
```

A.5 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`:

```
> let p = (2, 3);;
val p : int * int = (2, 3)
> let w = (2, true, 3.4, "blah");;
val w : int * bool * float * string = (2, true, 3.4, "blah")
```

A function may take a pair as an argument, by performing pattern matching on the pair pattern (x, y) :

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

In principle, 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:

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

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`:

```
> let earlier ((h1, m1), (h2, m2)) = h1<h2 || (h1=h2 && m1<m2);;
val earlier : ('a * 'b) * ('a * 'b) -> 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 `WriteLine` from the .NET class library:

```
> System.Console.WriteLine "Hello!";;
Hello!
val it : unit = ()
```

Thus the `unit` type serves much the same purpose as the `void` return type in C/C++/Java/C# — but the name is mathematically more appropriate.

A.6 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`:

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

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:

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

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 of `ss` is `string list`:

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

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

216 Lists

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

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

```
> let rec sum xs =
    match xs with
    | [] -> 0
    | x::xr -> x + sum xr;;
val sum : int list -> int
> let x2sum = sum x2;;
val x2sum : int = 29
```

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:

```
> let rec prod xs =
    match xs with
    | [] -> 1
    | x::xr -> x * prod xr;;
val prod : int list -> int
> let x2prod = prod x2;;
val x2prod : int = 819
> let rec len xs =
    match xs with
    | [] -> 0
    | x::xr -> 1 + len xr;;
val len : 'a list -> int
> let x2len = len x2;;
val x2len : int = 3
> let sslen = len ss;;
val sslen : int = 4
```

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.10.

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

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

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 shared with the list `x3`. This works as expected because lists are *immutable*: One cannot destructively change an element in list `x3` and thereby inadvertently change something in `x1x3`, or vice versa.

Some commonly used F# list functions are shown in Figure A.3.

Function	Type	Meaning
<code>append</code>	<code>'a list -> 'a list -> 'a list</code>	Append lists
<code>exists</code>	<code>('a -> bool) -> 'a list -> bool</code>	Does any satisfy...
<code>filter</code>	<code>('a -> bool) -> 'a list -> 'a list</code>	Those that satisfy...
<code>fold</code>	<code>('r -> 'a -> 'r) -> 'r -> 'a list -> 'r</code>	Fold (left) over list
<code>foldBack</code>	<code>('a -> 'r -> 'r) -> 'r -> 'a list -> 'r</code>	Fold (right) over list
<code>forall</code>	<code>('a -> bool) -> 'a list -> bool</code>	Do all satisfy...
<code>length</code>	<code>'a list -> int</code>	Number of elements
<code>map</code>	<code>('a -> 'b) -> 'a list -> 'b list</code>	Transform elements
<code>nth</code>	<code>'a list -> int -> 'a</code>	Get <i>n</i> 'th element
<code>rev</code>	<code>'a list -> 'a list</code>	Reverse list

Figure A.3: Some F# list functions, from the `List` module. The function name must be qualified by `List`, as in `List.append [1; 2] [3; 4]`. Some of the functions are polymorphic (Section A.10) or higher-order (Section A.11.2). For the list operators `cons (::)` and `append (@)`, see Figure A.1.

A.7 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. Before one can create a record value, one must create a record type, like this:

```
> type phonerec = { name : string; phone : int };;
type phonerec =
  {name: string;
   phone: int;}
> let x = { name = "Kasper"; phone = 5170 };;
val x : phonerec = {name = "Kasper";
                   phone = 5170;}
```

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*, very similar to field access in Java and C#:

218 *Raising and catching exceptions*

```
> x.name;;
val it : string = "Kasper"
> x.phone;;
val it : int = 886
```

A.8 Raising and catching exceptions

Exceptions can be declared, raised 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 `Exception`. A `raise expression` throws an exception:

```
> exception IllegalHour;;
exception IllegalHour
> let mins h =
    if h < 0 || h > 23 then raise IllegalHour
    else h * 60;;
val mins : int -> int
> mins 25;;
> [...] Exception of type 'FSI_0151+IllegalHourException' was thrown.
    at FSI_0152.mins(Int32 h)
    at <StartupCode$FSI_0153>.$FSI_0153.main@()
stopped due to error
```

A `try-with-expression` (`try e1 with 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#:

```
> try (mins 25) with IllegalHour -> -1;;
val it : int = -1
```

As a convenient shorthand, one can use the function `failwith` to throw the standard `Failure` exception, which takes a string message as argument. The variant `failwithf` takes as argument a `printf`-like format string and a sequence of arguments, to construct a string argument for the `Failure` exception:

```
> let mins h =
    if h < 0 || h > 23 then failwith "Illegal hour"
    else h * 60;;
val mins : int -> int
> mins 25;;
Microsoft.FSharp.Core.FailureException: Illegal hour
> let mins h =
    if h < 0 || h > 23 then failwithf "Illegal hour, h=%d" h
```

```

    else h * 60;;
val mins : int -> int
> mins 25;;
Microsoft.FSharp.Core.FailureException: Illegal hour, h=25

```

A.9 Datatypes

A *datatype*, sometimes called an *algebraic datatype* or *discriminated union*, is 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 only a name, or a Teacher who has both 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).

```

> type person =
    | Student of string                               (* name *)
    | Teacher of string * int;;                       (* name and phone no *)
type person =
    | Student of string
    | Teacher of string * int
> let people = [Student "Niels"; Teacher("Peter", 5083)];;
val people : person list = [Student "Niels"; Teacher ("Peter",5083)]
> let getphone person =
    match person with
    | Teacher(name, phone) -> phone
    | Student name         -> raise (Failure "no phone");;
val getphone : person -> int
> getphone (Student "Niels");;
Microsoft.FSharp.Core.FailureException: no phone

```

A.9.1 The option datatype

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

```

> type intopt =
    | Some of int
    | None;;
type intopt =
    | Some of int
    | None
> let getphone person =

```

```

match person with
| Teacher(name, phone) -> Some phone
| Student name         -> None;;
val getphone : person -> intopt
> getphone (Student "Niels");;
val it : intopt = None

```

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.

In F#, a polymorphic datatype `'a option` is predefined.

A.9.2 Binary trees represented by recursive datatypes

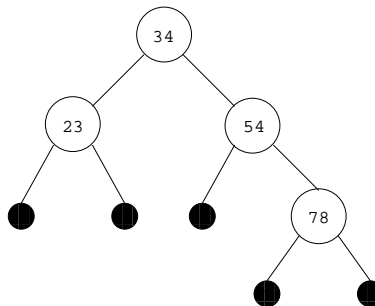
A datatype declaration may be recursive, which means that a value of the datatype `t` can have a component of type `t`. This can be used to represent trees and other data structures. For instance, a binary integer tree `inttree` may be defined to be either a leaf `Lf` or a branching node `Br` which holds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

```

> type inttree =
| Lf
| Br of int * inttree * inttree;;
type inttree =
| Lf
| Br of int * inttree * inttree
> let t1 = Br(34, Br(23, Lf, Lf), Br(54, Lf, Br(78, Lf, Lf)));;
val t1 : inttree = Br (34,Br (23,Lf,Lf),Br (54,Lf,Br (78,Lf,Lf)))

```

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:

```
> let rec sumtree t =
  match t with
  | Lf -> 0
  | Br(v, t1, t2) -> v + sumtree t1 + sumtree t2;;
val sumtree : inttree -> int
> let t1sum = sumtree t1;;
val t1sum : int = 189
```

The definition of `sumtree` reads: The sum of a leaf node `Lf` 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.9.3 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:

```
> let addp (x, y) = x + y;;
val addp : int * int -> int
> let addc x y = x + y;;
val addc : int -> int -> int
> let res1 = addp(17, 25);;
val res1 : int = 42
> let res2 = addc 17 25;;
val res2 : int = 42
```

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:

```
> let addSeventeen = addc 17;;
val addSeventeen : (int -> int)
> let res3 = addSeventeen 25;;
val res3 : int = 42
> let res4 = addSeventeen 100;;
val res4 : int = 117
```

A.10 Type variables and polymorphic functions

We saw in Section A.6 that the type of the `len` function was `'a list -> int`:

```
> let rec len xs =
  match xs with
  | []     -> 0
  | x::xr  -> 1 + len xr;;
val len : 'a list -> int
```

The `'a` is a *type variable*. Note that the prefixed prime (`'`) is part of the type variable name `'a`. In a call to 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 first to `int` and then to `string`:

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

A.10.1 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:

```
> type 'a tree =
  | Lf
  | Br of 'a * 'a tree * 'a tree;;
type 'a tree =
  | Lf
  | Br of 'a * 'a tree * 'a tree
```

Compare this with the monomorphic integer tree in Section A.9.2. Values of this type can be defined exactly as before, but the type is slightly different:

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

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:

```

> let rec sumtree t =
    match t with
    | Lf -> 0
    | Br(v, t1, t2) -> v + sumtree t1 + sumtree t2;;

val sumtree : int tree -> int
> let rec count t =
    match t with
    | Lf -> 0
    | Br(v, t1, t2) -> 1 + count t1 + count t2;;
val count : '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 `v`, and therefore works on an `'a tree` regardless of the node type `'a`.

Function `preorder1 : 'a tree -> 'a list` returns a list of the node values in a tree, in *preorder*, that is, the root node comes before the left subtree which comes before the right subtree:

```

> let rec preorder1 t =
    match t with
    | Lf -> []
    | Br(v, t1, t2) -> v :: preorder1 t1 @ preorder1 t2;;
val preorder1 : 'a tree -> 'a list
> preorder1 t1;;
val it : int list = [34; 23; 54; 78]

```

A side remark on efficiency: When the left subtree `t1` is large, then the call `preorder1 t1` will produce a long list of node values, and the list append operator (`@`) will be slow. Moreover, this happens recursively for all left subtrees.

Function `preorder2` does the same job in a more efficient, but slightly more obscure way. It uses an auxiliary function `preo` that has an *accumulating parameter* `acc` that gradually collects the result without ever performing an append (`@`) operation:

```

> let rec preo t acc =
    match t with
    | Lf -> acc
    | Br(v, t1, t2) -> v :: preo t1 (preo t2 acc);;
val preo : 'a tree -> 'a list -> 'a list
> let preorder2 t = preo t [];;
val preorder2 : 'a tree -> 'a list

```

224 *Higher-order functions*

```
> preorder2 t1;;  
val it : int list = [34; 23; 54; 78]
```

The following relation holds for all t and xs : $\text{preo } t \text{ } xs = \text{preorder1 } t @ xs$. From this it follows that $\text{preorder2 } t = \text{preo } t [] = \text{preorder1 } t @ [] = \text{preorder1 } t$.

A.10.2 Type abbreviations

When a type, such as $(\text{string} * \text{int}) \text{ list}$, 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  
> let bind1 (env : intenv) (x : string, v : int) : intenv = (x, v) :: env;;  
val bind1 : intenv -> string * int -> intenv
```

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 $\text{string} * \text{int}$ pairs:

```
> bind1 [("age", 47)] ("phone", 5083);;  
val it : intenv = [("phone", 5083); ("age", 47)]
```

A.11 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:

```
> let rec map f xs =  
    match xs with  
    | [] -> []  
    | x::xr -> f x :: map f xr;;  
val map : ('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 `mul2` of type $\text{float} \rightarrow \text{float}$ and use `map` to apply that function to all elements of a list:

```
> let mul2 x = 2.0 * x;;
val mul2 : float -> float
> map mul2 [4.0; 5.0; 89.0];;
val it : float list = [8.0; 10.0; 178.0]
```

Or we may apply a function `isLarge` of type `float -> bool` (defined on page 212) to all elements of a float list:

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

A.11.1 Anonymous functions

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

```
> fun x -> 2.0 * x;;
val it : float -> float = <fun:clo@0-1>
```

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

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

The function `tw` defined below takes a function `g` and an argument `x` and applies `g` twice; that is, it computes `g(g x)`. Using `tw` one can define a function `quad` that applies `mul2` twice, thus multiplying its argument by 4.0:

```
> let tw g x = g (g x);;
val tw : ('a -> 'a) -> 'a -> 'a
> let quad = tw mul2;;
val quad : (float -> float)
> quad 7.0;;
val it : float = 28.0
```

An anonymous function created with `fun` may take any number of arguments. A function that takes two arguments is similar to one that takes the first argument and then returns a new anonymous function that takes the second argument:

```

> fun x y -> x+y;;
val it : int -> int -> int = <fun:clo@0-2>
> fun x -> fun y -> x+y;;
val it : int -> int -> int = <fun:clo@0-3>

```

The difference between `fun` and `function` is that a `fun` can take more than one parameter but can have only one match case, whereas a `function` can take only one parameter but can have multiple match cases. For instance, two-argument `increaseBoth` is most conveniently defined using `fun` and one-argument `isZeroFirst` is most conveniently defined using `function`:

```

> let increaseBoth = fun i (x, y) -> (x+i, y+i);;
val increaseBoth : int -> int * int -> int * int
> let isZeroFirst = function | [0] -> true | _ -> false;;
val isZeroFirst : int list -> bool

```

A.11.2 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:

```

> let rec filter p xs =
    match xs with
    | [] -> []
    | x::xr -> if p x then x :: filter p xr else filter p xr;;
val filter : ('a -> bool) -> 'a list -> 'a list
> let onlyEven = filter (fun i -> i%2 = 0) [4; 6; 5; 2; 54; 89];;
val onlyEven : int list = [4; 6; 2; 54]

```

Note that the `filter` function is polymorphic in the argument list type. The `filter` function is predefined in F#'s `List` module. Another very general predefined polymorphic higher-order list function is `foldr`, for *fold right*, which exists in F# under the name `List.foldBack`:

```

> let rec foldr f xs e =
    match xs with
    | [] -> e
    | x::xr -> f x (foldr f xr e);;
val foldr : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

```

One way to understand `foldr f xs e` is to realize that it recursively replaces:

```

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

```

The `foldr` function presents a general procedure for processing a list, and is closely related to the visitor pattern in object-oriented programming, although this may not appear very obvious.

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

```

> let len xs = foldr (fun _ res -> 1+res) xs 0;;
val len : 'a list -> int
> let sum xs = foldr (fun x res -> x+res) xs 0;;
val sum : int list -> int
> let prod xs = foldr (fun x res -> x*res) xs 1;;
val prod : int list -> int
> let map g xs = foldr (fun x res -> g x :: res) xs [];;
val map : ('a -> 'b) -> 'a list -> 'b list
> let listconcat xss = foldr (fun xs res -> xs @ res) xss [];;
val listconcat : 'a list list -> 'a list
> let stringconcat xss = foldr (fun xs res -> xs ^ res) xss "";;
val stringconcat : string list -> string
> let filter p xs = foldr (fun x r -> if p x then r else x :: r) xs [];;
val filter : ('a -> bool) -> 'a list -> 'a list

```

The functions `map`, `filter`, `fold`, `foldBack` and many others are predefined in the `F# List` module; see Figure A.3.

A.12 F# mutable references

A *reference* is a handle to a *memory cell*. A reference in F# 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:

```

> let r = ref 177;;
val r : int ref = {contents = 177;}
> let v = !r;;
val v : int = 177
> r := 288;;

```

```

val it : unit = ()
> !r;;
val it : int = 288

```

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

```

> let nextlab = ref -1;;
val nextlab : int ref = {contents = -1;}
> let newLabel () = (nextlab := 1 + !nextlab; "L" ^ string (!nextlab));;
val newLabel : unit -> string
> newLabel();;
val it : string = "L0"
> newLabel();;
val it : string = "L1"
> newLabel();;
val it : string = "L2"

```

References are used also to implement efficient algorithms with destructive update, such as graph algorithms.

A.13 F# arrays

An F# array is a zero-based indexable fixed-size collection of elements of a particular type, just like a .NET array, but it uses a different syntax for array creation, indexing and update. The F# array type is generic in its element type:

```

> let arr = [| 2; 5; 7 |];;
val arr : int array = [|2; 5; 7|]
> arr.[1];;
val it : int = 5
> arr.[1] <- 11;;
val it : unit = ()
> arr;;
val it : int array = [|2; 11; 7|]
> arr.Length;;
val it : int = 3

```

The .NET method `System.Environment.GetCommandLineArgs()` has type `string array` and holds the command line arguments of an F# when invoked from a command prompt. The element at index 0 is the name of the running executable, the element at index 1 is the first command line argument, and so on (as in C).

A.14 Other F# features

The F# language has a lot more features than described in this appendix, including facilities for object-oriented programming and convenient use of the .NET libraries and programs written in other .NET languages, and many advanced functional programming features. For a more comprehensive description of F# and its libraries, see [97, 96] and the F# resources linked from the course homepage.

A.14.1 Pitfalls and problems

In general it is extremely useful that an F# program can call any .NET library method, but this can also produce rather confusing results. For instance, an attempt to apply .NET method `String.Concat` to an F# string list — rather than a string array, as was probably intended — may give this mysterious result:

```
> System.String.Concat ss;;
val it : string =
    "Microsoft.FSharp.Collections.FSharpList`1+_Cons[System.String]"
```

What happened here is probably that the overload `Concat(params Object[])` from .NET class `System.String` was used, with the entire F# list as the single member of the `Object` array. Therefore the result of the function call is obtained by calling `System.Object.ToString()` on the F# list, and that method (if not overridden) will return a string representation of the name of the object's *type*, not the list's *value*. In this case, we see that what the type known as `string list` in F# has a more unwieldy name seen from .NET.