**[Notes on Functional Programming in F# 2.0](https://moodle.cis.fiu.edu/v2.1/mod/page/view.php?id=8948" \o "Notes on Functional Programming in F# 2.0)**

**by Geoffrey Smith**

1. **A Quick Overview of F#**

The F# design was initiated in 2002 by Don Syme and others at Microsoft Research in Cambridge, England; the first major pre-release was in 2005. In April 2010, F# 2.0 was released, and made a fully-supported language in Visual Studio 2010.

Main goal: Bring functional programming as in the ML family to the .NET platform.

But F# is actually a "multi-paradigm" language combining functional programming with imperative object-oriented programming.

Our class will emphasize the functional programming side of F#, which computes results not through repeated *assignments to variables* but instead through *applications of functions* and *recursion*.

Other important attributes of F#:

* Based on the popular ML dialect called *OCaml* from INRIA in France.
* Gives full access to the *.NET libraries* and supports *interoperability* with other .NET languages (C#, C++, VB, etc.)
* Functions are *first class values* that can be created and used as freely as any other kind of value.
* *Static scoping* and *eager evaluation*, as in most modern languages.
* *Static typing* via *type inference*: normally F# code need not include any type annotations, and the F# compiler automatically infers the best possible *polymorphic type*, allowing code reuse.
* *Pattern-matching syntax* for function definitions by case analysis.
* Rich *module* system.
* *Exception handling*.
* Automatic storage management via *garbage collection*.
* Incremental compiler supporting *interactive program development*.
* Succinct syntax that makes some keywords optional, and which pays attention to *code indentation*.

1. **F# Programming Environments**

There are several environments available for using F#:

* Stand-alone F# Interactive fsi
* Visual Studio Scripting
* Visual Studio Projects

**Stand-alone F# Interactive fsi**

* On Windows, fsi can be run from the Visual Studio Command Prompt (found in Development Tools>Microsoft Visual Studio 2010>Visual Studio Tools).
* On other platforms (such as Mac OS X and Linux), fsi is included in mono, version 2.10.6, which can be downloaded from [www.mono-project.com](http://www.mono-project.com/).
* Upon start up, you will see
* Microsoft (R) F# 2.0 Interactive build 4.0.40219.1
* Copyright (c) Microsoft Corporation. All Rights Reserved.
* For help type #help;;
* >
* You can enter F# code directly at the prompt >, terminating with ;;, and getting immediate results.
* fsi is thus very nice for quick-and-dirty experiments with F#!
* You can also load a code file myfile.fsx by using the command
* > #load "myfile.fsx";;

(This file should be located in the same directory where you started fsi.)

* Note that all your definitions are implicitly put into the *namespace* Myfile.

Hence to access a function foo, you need to refer to it as Myfile.foo.

Or, alternatively, you can first do the command

> open Myfile;;

**Visual Studio Scripting**

* *I recommend this mode for our section of COP 4555. We will use Visual Studio 2015 in this class.*
* In this environment, you have a window displaying the file containing your F# program, and below it you have an F# Interactive window (as with fsi).
* "IntelliSense" gives helpful information about whatever piece of code you move the mouse over in the code window. Also, you can type F1 to get detailed documentation.
* Create file with File/New/File (Ctrl+N) and choose Script/F# Script File.
* Open the F# Interactive with View/F# Interactive (Ctrl+Alt+F). (You need to do this only once.)
* Select all the text in your script file with Edit/Select All (Ctrl+A).
* Execute your code in the F# Interactive with (Alt+Enter).
* As with the stand-alone console, you can also type directly into the F# Interactive, terminating your input with ;;.

**Visual Studio Projects**

* The mode gives you a debugger, but it is considerably more "heavyweight" than using scripting, so I think it is not the best choice for COP 4555.
* Create file with File/New/Project (Ctrl+Shift+N) and choose F# Application.
* Run as a console application with Start Without Debugging (Ctrl+F5).
* Run up to your first break point with Start Debugging (F5).
* Single step through code with Step Into (F11).

1. **F# Basics**

Now let's explore the basics of F# by looking at some simple examples using the F# Interactive, in either Visual Studio or fsi.

The prompt is >, and you terminate your input with ;;. Here's a simple example:

> 3 + 4 \* 5;;

val it : int = 23

Here F# is telling you that the expression you entered (it) has type int and value 23.

Notice that on a multi-line input, the prompt is not shown again until the input is complete:

> 2 + 3

\* 4;;

val it : int = 14

(Under fsi, the prompt instead changes to -.)

As a small convenience, the identifier it is bound to the value of the last expression entered:

> it;;

val it : int = 14

More usefully, you can use let to bind a value to whatever identifier you want:

> let x = 6;;

val x : int = 6

> x+x;;

val it : int = 12

The above is a *global* declaration of x, whose scope extends to the end of the session (unless a new declaration masks it).

The possibility of a new declaration of x is a bit confusing, since you might then think that let expressions are simply *assignments to variables*. But really let expressions are *declarations of constants*. For instance, if we now say

> let x = x+1;;

val x : int = 7

then we have not *incremented* the value of x. Instead, we have simply defined a *new* x which masks visibility to the old x. We can demonstrate that this is so by considering *local declarations* using the syntax

let *pat* = *exp1* in *exp2*

Here, the scope of the declaration is limited to *exp2*:

> let y = 17 in 3\*y;;

val it : int = 51

> y;;

y;;

^^

stdin(17,1): error FS0039: The value or constructor 'y' is not defined.

Here is an example that shows that let is *not* assignment:

> let z = 5 in (let z = 3\*z+1 in 2\*z) + z;;

val it : int = 37

Here the inner let expression defines z to be 16 in 2\*z, so it has value 32. But once we leave the scope of the inner let, the value of z is still 5, so we get the final result 37.

Particularly useful are *function* declarations:

> let succ n = n+1;;

val succ : int -> int

In this syntax, succ is the name of the function and n is its parameter. The expression n+1 after = gives the value that the function returns; thus we do not use a return statement.

Looking at the output, we note that F# does not display the *value* of succ, since it is a *function* with no good human-readable representation. But it does display the *type* of succ, which it *infers* to be int -> int. This means that succ takes an input of type int and produces an output of type int. The basic idea here is that the type of the parameter n can be inferred from the way that n is *used* in the body of the function---the expression n+1 tells us that n must be an int. We will study type inference in much more detail later in the semester.

In function calls, parentheses are not needed; you just write the name of the function followed by its argument:

> succ 12;;

val it : int = 13

> succ 3\*7;;

val it : int = 28

In the second example, notice that one could have parsed the expression in two ways. F# resolves this ambiguity by giving prefix function application higher precedence than any infix operator. Of course, parentheses can be used to get the other parse:

> succ (3\*7);;

val it : int = 22

Note that if the argument is a negative literal, then F# requires a separating space to prevent it from parsing the "-" as a binary operator:

> succ -12;;

val it : int = -11

> succ-12;;

succ-12;;

-----^^

stdin(20,6): error FS0001: The type 'int' does not match the type 'int -> int'

The syntax of our declaration of succ looks quite different from the declarations we considered before. Here's a different, equivalent way of defining succ:

> let succ = fun n -> n+1;;

val succ : int -> int

This syntax emphasizes that we are binding a value (which happens to be a *function*) to the identifier succ. Here the expression fun x -> x+1 is called an *anonymous*function; its value is a function (with no name) that accepts an input x and produces as output x+1. Anonymous functions are quite useful, as there are often situations where we want to use a function just once; in that case, it is convenient not to have to give it a name.

It is quite common to use *local declarations* in the body of a function to compute intermediate results. The usual layout is to indent the body, breaking it over multiple lines and omitting the optional keyword in:

> let cos\_squared r =

let c = cos r

c\*c;;

val cos\_squared : float -> float

> cos\_squared (3.14159/4.0);;

val it : float = 0.5000006634

> c;;

c;;

^

stdin(10,1): error FS0039: The value or constructor 'c' is not defined

Notice that this local let allows us to use the value of cos r twice, while only calculating it once.

You should also be aware that F# is fussy about the indentation of your code, using it sometimes to resolve ambiguities.

> let cos\_squared r =

let c = cos r

c\*c;;

let c = cos r

----^^^

stdin(15,5): error FS0588: Block following this 'let' is unfinished. Expect an expression.

(The indentation rules are sometimes confusing, I think. They are explained in detail at <http://msdn.microsoft.com/en-us/library/dd233191.aspx>.)

More complex F# functions typically require the use of *recursion*, which syntactically requres the keyword rec:

> let rec fact n = if n = 0 then 1 else n \* fact (n-1);;

val fact : int -> int

> fact 5;;

val it : int = 120

Notice that if e1 then e2 else e3 is an *expression* in F#, rather than a *statement*. It is just like (e1 ? e2 : e3) in C or Java.

You should of course write comments in your F# programs; these can be written in several ways:

(\* A comment \*)

// A one-line comment

/// A comment that Visual Studio will display

We conclude our discussion of F# basics with a couple of useful commands for the F# Interactive. You can toggle timing on and off with #time;;, and you can interrupt a long (or infinite!) computation by typing ctrl . in Visual Studio and control c in fsi:

> let rec fib n =

if n=0 then 0

elif n=1 then 1

else fib(n-1)+fib(n-2);;

val fib : int -> int

> #time;;

--> Timing now on

> fib 42;;

Real: 00:00:06.034, CPU: 00:00:05.865, GC gen0: 0, gen1: 0, gen2: 0

val it : int = 267914296

> fib 100;;

- Interrupt

We continue our discussion in the next sections by explaining the different *data types* supported by F#.

1. **Primitive Data Types in F#**

F# includes a large number of *primitive data types*:

* unit has only one value: ()

This is a degenerate type; it is mainly used for functions that take no inputs or return no result. For example,

> printf "hello\n";;

hello

val it : unit = ()

* bool includes true, false and operators not, &&, and ||, which do *short circuit* evaluation as in C and Java.
* int includes ... -2, -1, 0, 1, 2, ... using 32-bit two's complement arithmetic and supporting the usual operators: +, -, \*, /, %, =, <>, <, <=, >, >=, abs, ...
* bigint includes ... -2I, -1I, 0I, 1I, 2I, ... of arbitrary size and supports the int operators.
* float includes numbers of the form 3.17 or 2.4e17, and supports the int operators and also cos, floor, sqrt, \*\* (exponentiation), ...

Note that F# has no automatic coercions among numeric types:

> 4.2 \* 3;;

4.2 \* 3;;

------^

stdin(25,7): error FS0001: The type 'int' does not match the type 'float'

Use float : int -> float, int : float -> int.

* string includes strings of the form "my string\n", where as usual \t is tab and \n is newline. It supports =, < (comparing by lexicographic order), <=, ..., +(concatenation), String.length, and the postfix operator .[] for selecting characters from a string:
* > "cat".[2];;

val it : char = 't'

* char includes characters of the form 'b'.

Complete documentation of these types can be found at <http://msdn.microsoft.com/en-us/library/dd233181.aspx>.

Now we turn our attention to F#'s *type constructors*, which let us build more complex types out of simpler types.

1. **Tuples**

We can form a new value by combining values into a *tuple*:

> (17, "abc", true);;

val it : int \* string \* bool = (17, "abc", true)

Note that this type is like a *Cartesian product* in mathematics. (The parentheses around tuples are actually optional, but I find it clearer not to omit them.) To select the components of a *pair* (i.e. a tuple of length 2), you can use the operators fst and snd:

> snd (3, true);;

val it : bool = true

But usually we use *pattern matching syntax* to process tuples. Here's a simple example:

> let (a,b,c) = (17, "bird", true);;

val c : bool = true

val b : string = "bird"

val a : int = 17

Here (a,b,c) is a *pattern* that is matched against the tuple, binding identifiers a, b, and c.

Pattern matching is particularly useful in function definitions:

> let rec power (m,n) = if n = 0 then 1.0 else m \* power (m, n-1);;

val power : float \* int -> float

> power (2.0, 10);;

val it : float = 1024.0

In this definition, the pattern (m,n) is matched against the input to the function, giving names for the first and second components that can then be used in the function body. A definition using fst and snd is much less readable:

let rec power pair =

if snd pair = 0 then 1.0 else fst pair \* power (fst pair, snd pair - 1)

Having tuples as *first-class values* in F# allows a simple and uniform view of functions: we can say that every function takes exactly *one input* and returns exactly *one output*! For example, here is a function that uses tuples to accept two inputs and to produce two outputs:

> let quo\_rem (x,y) = (x/y, x%y);;

val quo\_rem : int \* int -> int \* int

> quo\_rem (17, 5);;

val it : int \* int = (3, 2)

> quo\_rem (quo\_rem (17, 5));;

val it : int \* int = (1, 1)

1. **Functions**

Every F# function has a type of the form t1 -> t2, where t1 is the type of the input and t2 is the type of the output, and (as dictated by the principle of *uniformity*) wheret1 and t2 can be *any* types whatsoever.

**A. Type Inference and Polymorphism**

As we have seen, F# functions can typically be written without *any* type annotations, because the F# compiler automatically *infers* the type of the function by seeing how the parameters are used in the function body.

One particularly interesting possibility is that inference may not find a unique type for a function. Consider for example a function that swaps the elements of a pair:

> let swap (x,y) = (y,x)

Here we can see that the types of x and y could be anything at all, and swap would still make sense. So it would be most unfortunate to infer any single type as this would limit the reusability of swap. Instead F# captures the (infinitely many!) possible types for swap by using *type variables* 'a and 'b:

val swap : 'a \* 'b -> 'b \* 'a

In such a type, the type variables should be understood to be implicitly *universally quantified*. More explicitly, we could write the type as

(forall 'a, 'b)('a \* 'b -> 'b \* 'a)

The idea is that we can replace 'a and 'b by whatever types we want, giving many types for swap:

int \* bool -> bool \* int

string \* string -> string \* string

(int -> int) \* float -> float \* (int -> int)

...

And indeed F# allows all such uses of swap:

> swap (4, true);;

val it : bool \* int = (true, 4)

> swap ("cat", "dog");;

val it : string \* string = ("dog", "cat")

> swap ((fun n -> n+1), 5.0);;

val it : float \* (int -> int) = (5.0, <fun:it@9>)

We say that swap is a *polymorphic function*, and we will later see many more useful examples of polymorphism, particularly when we study *lists*.

**B. Typing Issues with Overloaded Operators**

The overloaded operator + is defined on a variety of numeric types (int, bigint, float, ...) and also on string (denotating concatenating), but it is not polymorphic in the way that swap is. This poses a problem for type inference on some functions that use +. Consider, for example,

> let double x = x+x;;

Here there is no clue which meaning of + should be used, so several types for double are possible. But it would be incorrect to give double the polymorphic type 'a -> 'a.

F# deals with this problem in a rather crude way. In the absence of any clues to the contrary, it simply defaults to the int version of an overloaded numeric operator (+, -, \*,/, ...). Here this gives the typing

val double : int -> int

This policy seems motivated by concerns about *implementability*. While one could give a uniform, pointer-based implementation of the polymorphic swap, this would be much harder for double since the implementations of int and float addition are completely different.

If you wanted a different type for double, then this is one of the rare cases where you must use a type annotation:

> let double (x:string) = x+x;;

val double : string -> string

or

> let double x : string = x+x;;

val double : string -> string

(which tells F# that double returns a string.)

It is very important to understand that the "default to int" policy applies *only* to overloaded *numeric* operators. It does *not* apply to polymorphic functions like swap. Nor does it apply to overloaded *comparison* operators like < and >=. These are handled in a very interesting way, using *constrained quantification*. Here's an example:

> let inorder (x,y,z) = (x <= y) && (y <= z);;

val inorder : 'a \* 'a \* 'a -> bool when 'a : comparison

Function inorder tests whether the elements of a triple are in nondecreasing order. To be able to do this, it needs to be able to do <= on each of the elements. F# therefore gives inorder the polymorphic type 'a \* 'a \* 'a -> bool under the *constraint* that the type variable 'a can only be replaced by a type that satisfies the comparisonconstraint. This constraint is satisfied by many, by not all, types. In particular, it is not satisfied by *function types*. Thus

> inorder (3I, 4I, 5I);;

val it : bool = true

> inorder (3.7, 4.8, 4.799);;

val it : bool = false

> inorder (swap, swap, swap);;

inorder (swap, swap, swap);;

---------^^^^

stdin(33,10): error FS0001: The type '('a \* 'b -> 'b \* 'a)' does not support the 'comparison' constraint. For example, it does not support the 'System.IComparable' interface.

Very similarly, the equality operators = and <> are handled using the equality constraint:

> let different (x,y,z) = (x<>y) && (y<>z) && (x<>z);;

val different : 'a \* 'a \* 'a -> bool when 'a : equality

(We remark, finally, that inline functions using overloaded numeric operators are handled in a quite interesting way, but we will not discuss them here.)

**C. Curried Functions**

We now consider functions of type t1 -> (t2 -> t3), whose output type (t2 -> t3) is itself a function type. Such functions dynamically create new functions when they are called.

Consider the following function definition:

> let add (a, b) = a+b;;

val add : int \* int -> int

> add (3,4);;

val it : int = 7

add expects to be given its two arguments in a pair. It is an error to give it only one argument:

> add 3;;

add 3;;

----^^

stdin(55,5): error FS0001: This expression has type

int

but is here used with type

int \* int.

Consider the following variant of add:

> let cadd a = (fun b -> a+b);;

val cadd : int -> (int -> int)

We have written cadd so that it takes one argument a and returns a *function* that takes a second argument b. Thus cadd is a version of add that can take its arguments *one at a time*:

> cadd 3;;

val it : (int -> int) = <fun:it@15-5>

> it 4;;

val it : int = 7

> let add7 = cadd 7;;

val add7 : (int -> int)

> add7 10;;

val it : int = 17

> (cadd 5) 8;;

val it : int = 13

cadd is said to be in "curried" form (after logician Haskell Curry).

In F#, -> associates to the *right*, so the type of cadd can be written without parentheses, which is what F# actually does:

> let cadd a = (fun b -> a+b);;

val cadd : int -> int -> int

Notice that the types int -> (int -> int) and (int -> int) -> int have completely different meanings! [Can you think of an expression that would have the latter type?]

Also, F# function application associates to the *left*, which means that we can omit the parentheses in (cadd 5) 8.

> cadd 5 8;;

val it : int = 13

In general we can have an application like e1 e2 e3 e4, which is equivalent to (((e1 e2) e3) e4).

Because curried functions are quite common in F#, F# allows an easier syntax for defining them. In this syntax, the arguments are just written in sequence, separated by blanks:

> let cadd a b = a+b;;

val cadd : int -> int -> int

This is that syntax that is ordinarily used by experienced F# programmers.

It is worth noting that anonymous functions can also be curried:

> fun a b -> a+b;;

val it : int -> int -> int = <fun:clo@0>

Finally, we note that F# infix operators are typically curried. But we cannot simply use them by themselves, because we get a syntax error:

> \*;;

\*;;

^

stdin(46,1): error FS0010: Unexpected symbol '\*' in interaction

So F# lets us *convert* an infix operator to a prefix function by surrounding it with parentheses. This lets us use it usefully as a curried function:

> (\*);;

val it : (int -> int -> int) = <fun:it@47-10>

> let triple = (\*) 3;;

val triple : (int -> int)

> triple 7;;

val it : int = 21

1. **Lists**

Following LISP, F# supports arbitrary-length, immutable lists of values:

> [1; 2; 3];;

val it : int list = [1; 2; 3]

All elements of a list must be of the same type.

Note that there are infinitely many list types:

* [1;2;3] : int list
* ["a";"b"] : string list
* [[3];[4;5];[]] : int list list

F# allows us to form a list using a numeric range:

> [1..5];;

val it : int list = [1; 2; 3; 4; 5]

And F# provides a rich set of built-in operations for lists:

* [] is the empty list.
* :: (cons) returns a list with a new element at the front:
* "a" :: ["b"; "c"] = ["a"; "b"; "c"]

Lists are built up from [] using :: and they can be decomposed in the same way:

[1;2;3] = 1::[2;3] = 1::2::[3] = 1::2::3::[]

This implies that :: associates to the *right*.

* List.length returns the length of a list.
* List.rev returns the reverse of a list.
* @ appends two lists together: [1;2;3]@[4;5] = [1;2;3;4;5]
* List.isEmpty tests whether a list is empty.
* List.head returns the first element of a list, and List.tail returns all elements after the first:
* List.head [1;2;3] = 1, List.tail [1;2;3] = [2;3]

(In LISP, these operations were called car and cdr.)

* List.map applies a function to all elements of a list and returns the list of results.
* > List.map (fun n -> n\*n) [1;2;3;4];;
* val it : int list = [1; 4; 9; 16]
* List.filter applies a predicate to each element of a list and returns the list of elements that satisfy the predicate.
* > List.filter (fun x -> x%2 = 0) [1;2;3;4;5];;
* val it : int list = [2; 4]
* List.reduce uses a binary function to "reduce" a list down to a single value, associating to the left:
* > List.reduce (\*) [1..10];;
* val it : int = 3628800

We emphasize that none of these operations are *destructive*; they all return *new* lists.

**Aside**: map and reduce on immutable lists are related to the "MapReduce" model advocated by Google in [research.google.com/archive/mapreduce.html](http://research.google.com/archive/mapreduce.html):

"Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system."

The types of these functions are interesting, because they are all *polymorphic*.

[] : 'a list

:: : 'a \* 'a list -> 'a list

List.length : 'a list -> int

List.rev : 'a list -> 'a list

@ : 'a list -> 'a list -> 'a list

List.isEmpty : 'a list -> bool

List.head : 'a list -> 'a

List.tail : 'a list -> 'a list

List.map : ('a -> 'b) -> 'a list -> 'b list

List.filter : ('a -> bool) -> 'a list -> 'a list

List.reduce : ('a -> 'a -> 'a) -> 'a list -> 'a

Let's try some examples with map:

> List.map String.length ["cat"; "fish"; "elephant"; "dinosaur"];;

val it : int list = [3; 4; 8; 8]

> List.map (fun xs -> 5::xs) [[1;2];[3]];;

val it : int list list = [[5; 1; 2]; [5; 3]]

> List.map (fun n -> (5,n)) [true; false];;

val it : (int \* bool) list = [(5, true); (5, false)]

> List.map (fun (x,xs) -> x::xs) [(3,[4;5]); (2,[1])];;

val it : int list list = [[3; 4; 5]; [2; 1]]

> List.map ((\*) 7) [1..5];;

val it : int list = [7; 14; 21; 28; 35]

The polymorphism of F#'s list operators is very useful. But as you play with them, you are likely to run into a confusing type inference error due to something called the **value restriction**:

> List.rev [];;

List.rev [];;

^^^^^^^^^^^

stdin(48,1): error FS0030: Value restriction.

The value 'it' has been inferred to have generic type

val it : '\_a list

Either define 'it' as a simple data term, make it a function with explicit arguments or, if you do not intend for it to be generic, add a type annotation.

Very briefly, the issue is that F# does not allow a *function application* to get a polymorphic type. Note that you can get rid of the error here by adding a type annotation:

> List.rev ([] : int list);;

val it : int list = []

Later, we will discuss the value restriction thoroughly in my [Notes on F#'s Value Restriction](https://moodle.cis.fiu.edu/v2.1/mod/page/view.php?id=8952); for now you should not worry about any such errors that you get.

**Stylistic remark**: We can add an element to the front of a list by forming a list of length one and appending it to the front: [5]@[1;2;3]. But it is cleaner here to just use cons: 5::[1;2;3].

Now that we know about F#'s primitives for manipulating lists, let's try to define some interesting functions ourselves. First, let's define a function to multiply together all the numbers in an int list:

> let rec prod ns =

if List.isEmpty ns then 1 else List.head ns \* prod (List.tail ns);;

val prod : int list -> int

> prod [1..5];;

val it : int = 120

Operationally, prod will be calculated as follows:

prod [2;3;4]

= 2 \* prod [3;4]

= 2 \* (3 \* prod [4])

= 2 \* (3 \* (4 \* prod []))

= 2 \* (3 \* (4 \* 1))

= 2 \* (3 \* 4)

= 2 \* 12

= 24

(However, this operational description is not really the best way way to understand recursive functions like prod. Shortly, we will present a [*Checklist for Programming with Recursion*](https://moodle.cis.fiu.edu/v2.1/mod/page/view.php?id=8949) that gives effective techniques for systematically *deriving* correct recursive programs.)

Actually, experienced F# programs would not define prod in the way that we did. Instead, they would define it using an elegant case analysis style that does *pattern matching* on the form of the input:

> let rec prod = function

| [] -> 1

| n::ns -> n \* prod ns;;

val prod : int list -> int

The first case of the definition uses the pattern [], which matches only the empty list; it says that the output should be 1 in that case. The second case uses the less obvious pattern n::ns. This pattern matches any list that is formed by *cons*ing some value n with some other value ns. In other words, it matches any non-empty list. Moreover, pattern matching *binds* the head and tail of the list to the identifiers n and ns, respectively. These bindings let us describe the result very cleanly, as n \* prod ns.

Alternatively, we can do the pattern matching via an explicit match expression:

> let rec prod ms =

match ms with

| [] -> 1

| n::ns -> n \* prod ns;;

val prod : int list -> int

You can also use integers in patterns:

> let rec downFrom = function

| 0 -> []

| n -> n :: downFrom (n-1);;

val downFrom : int -> int list

> downFrom 5;;

val it : int list = [5; 4; 3; 2; 1]

F# tries the patterns in order, so downFrom 0 is processed by the first pattern and not the second, even though the pattern n matches anything.

We can use downFrom and prod to give another definition of factorial:

> let fact n = prod (downFrom n);;

val fact : int -> int

> fact 5;;

val it : int = 120

Alternatively, we could have written

> let fact = downFrom >> prod;;

val fact : (int -> int)

using F#'s *function composition* operator >>, which has the property that

(f >> g) x = g (f x)

Notice that function composition has an interesting polymorphic type:

> (>>);;

val it : (('a -> 'b) -> ('b -> 'c) -> 'a -> 'c) = <fun:it@23-9>

This type implies that it would be type correct to do the composition in the *other order*:

let mystery = prod >> downFrom

What type does mystery get? What does mystery compute?

Yet another way of writing factorial is to use F#'s *forward pipeline* operator |>:

> (|>);;

val it : ('a -> ('a -> 'b) -> 'b) = <fun:it@31-14>

> let fact n = n |> downFrom |> prod;;

val fact : int -> int

The F# designers are quite fond of this coding style.

1. **Some Rules of F# Syntax**

The syntax of F# is quite free, so it is important that you have a clear understanding of F#'s rules of precedence and associativity. Otherwise you may find yourself completely misinterpreting a piece of code!

The most important rules to remember are that prefix function application

* binds tighter than any infix operator, and
* associates to the left.

Hence

List.map (fun n -> n\*n) [1;2;3]@[4;5;6]

parses as

((List.map (fun n -> n\*n)) [1;2;3])@[4;5;6]

and hence produces output [1; 4; 9; 4; 5; 6].

Infix operators have the following precedences, ordered from *highest* to *lowest*:

\*\*

\* / %

+ -

:: @

= <> < <= >> |>

&&

||

All of these operators associate to the left, except for \*\*, ::, and @, which associate to the right:

> 2.0 \*\* 2.0 \*\* 3.0;;

val it : float = 256.0

You also need to know the precedence rules for *types* in F#. These are especially important to know, because F# always prints out types with as few parentheses as possible.

Here is the precedence of type constructors, again ordered from *highest* to *lowest*:

list

\*

->

And -> associates to the right.

Hence

int -> bool \* string list

parses as

int -> (bool \* (string list))

**Some F# Idioms**

F#'s syntax often allows something to be expressed in several different ways. This flexibility is convenient, but can also be a source of confusion. Suppose for example that we wish to define a curried recursive function foo with two parameters. If we do not wish to do pattern matching on either parameter, then we can define foo in several ways:

1. let rec foo xs ys = ...
2. let rec foo xs = fun ys -> ...
3. let rec foo = fun xs -> fun ys -> ...
4. let rec foo = fun xs ys -> ...

Now suppose we want to do pattern matching on just the *second* parameter. In this case, it seems cleanest to use option 2 above, but using the keyword function instead offun:

let rec foo xs = function

| [] -> ...

| y::ys -> ...

Using this idiom, for example, we can define List.map in just three lines of code:

> let rec map f = function

| [] -> []

| x::xs -> f x :: map f xs;;

val map : ('a -> 'b) -> 'a list -> 'b list

What if we want to do pattern matching on just the *first* parameter? In this case, we could write

let rec foo = function

| [] -> fun ys -> ...

| x::xs -> fun ys -> ...

but this is perhaps ugly. Here it seems clearer to use a match expression instead:

let rec foo xs ys =

match xs with

| [] -> ...

| x::xs -> ...

Finally, what if we want to do pattern matching on *both* parameters? In this case, we might expect to be able to write

let rec foo = function

| [] ys -> ...

| xs [] -> ...

| x::xs y::ys -> ...

but F# does not allow this syntax---each clause after a function can contain only a *single* pattern. In this case, the best approach again seems to be to use a match:

let rec foo xs ys =

match (xs, ys) with

| ([], ys) -> ...

| (xs, []) -> ...

| (x::xs, y::ys) -> ...