Learning to program with F#

Jon Sporring

September 17, 2016

Contents

Chapter 6

Constants, functions, and variables

In the previous chapter, we saw how to use F# as a calculator working with literals, operators and built-in functions. To save time and make programs easier to read and debug, it is useful to bind expressions to identifiers either as new constants, functions, or operators. As an example, consider the problem,

Problem 6.1:

For given set constants a, b, and c, solve for x in

$$ax^2 + bx + c = 0 \tag{6.1}$$

To solve for x we use the quadratic formula from elementary algebra,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \tag{6.2}$$

which gives the general solution for any values of the coefficients. Here, we will assume a positive discrimant, $b^2 - 4ac > 0$. In order to write a program, where the code may be reused later, we define a function `discriminant : float -> float -> float` , that is, a function that takes 3 arguments, `a`, `b`, and `c`, and calculates the distriminant. Details on function definition is given in Section 6.2. Likewise, we will define functions `positiveSolution : float -> float -> float` and `negativeSolution : float -> float -> float`, that also takes the polynomial's coefficients as arguments and calculates the solution corresponding to choosing the postive and negative sign for $\pm$ in the equation. Our solution thus looks like Listing 6.1.

Listing 6.1, identifiersExample.fsx:
Finding roots for quadratic equations using function name binding.

```
let discriminant a b c = b ** 2.0 - 4.0 * a * c
let positiveSolution a b c = (-b + sqrt (discriminant a b c)) / (2.0 * a)
let negativeSolution a b c = (-b - sqrt (discriminant a b c)) / (2.0 * a)

let a = 1.0
let b = 0.0
let c = -1.0
let d = discriminant a b c
let xp = positiveSolution a b c
let xn = negativeSolution a b c
printfn "0 = %A * x ** 2.0 + %A * x + %A" a b c
printfn "  has discriminant %A and solutions %A and %A" d xn xp
```

```
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has discriminant 4.0 and solutions -1.0 and 1.0
```

Here, we have further defined names of values `a`, `b`, and `c` used as input to our functions, and the results of function application is bound to the names `d`, `xn`, and `xp`. The names of functions and values given here are examples of identifiers, and with these, we may reuse the quadratic formulas and calculated values later, while avoiding possible typing mistakes and reducing amount of code, which needs to be debugged.

Before we begin a deeper discussion note that F# has adheres to two different syntax: regular and *ligthweight*. In the regular syntax, newlines and whitespaces are generally ignored, while in lightweight syntax, certain keywords and lexemes may be replaced by specific use of newlines and whitespaces. Lightweight syntax is the most common, but the syntaxes may be mixed, and we will highlight the options, when relevant.

· lightweight syntax

The use of identifiers is central in programming. For F# not to be confused by built-in functionality, identifiers must follow a specific grammar: An identifier must start with a letter, but can be followed by zero or more of letters, digits, and a range of special characters except SP, LF, and CR (space, line feed, and carriage return). An identifier must not be a keyword or a reserved-keyword listed in Figures 6.1. An identifier is a name for a constant, an expression, or a type, and it is defined by the following EBNF:

Keywords:

abstract, and, as, assert, base, begin, class, default, delegate, do, done, downcast, downto, elif, else, end, exception, extern, false, finally, for, fun, function, global, if, in, inherit, inline, interface, internal, lazy, let, match, member, module, mutable, namespace, new, null, of, open, or, override, private, public, rec, return, sig, static, struct, then, to, true, try, type, upcast, use, val, void, when, while, with, and yield.

Reserved keywords for possible future use:

atomic, break, checked, component, const, constraint, constructor, continue, eager, fixed, fori, functor, include, measure, method, mixin, object, parallel, params, process, protected, pure, recursive, sealed, tailcall, trait, virtual, and volatile.

Symbolic keywords:

let!, use!, do!, yield!, return!, |, ->, <-, ., :, (,), [,], [<, >], [|, |], {, }, ', #, :?>, :?, :>, .., ::, :=, ;;, ;, =, _, ?, ??, (*), <@, @>, <@@, and @@>.

Reserved symbolic keywords for possible future:

~ and `.

Figure 6.1: List of (possibly future) keywords and symbolic keywords in F#.

Listing 6.2: Identifieres

```
ident = (letter | "_") {letter | dDigit | specialChar};
longIdent = ident | ident "." longIdent; (*no space around "."*)

longIdentOrOp = [longIdent "."] identOrOp; (*no space around "."*)
identOrOp =
  ident
  | "(" infixOp | prefixOp ")"
  | "(*)";

dDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
letter = Lu | Ll | Lt | Lm | Lo | Nl; (*e.g. "A", "B" ... and "a", "b", ...*)
specialChar = Pc | Mn | Mc | Cf; (*e.g., "_"*)

codePoint = ?Any unicode codepoint?;
Lu = ?Upper case letters?;
Ll = ?Lower case letters?;
Lt = ?Digraphic letters, with first part uppercase?;
Lm = ?Modifier letters?;
Lo = ?Gender ordinal indicators?;
Nl = ?Letterlike numeric characters?;
Pc = ?Low lines?;
Mn = ?Nonspacing combining marks?;
Mc = ?Spacing combining marks?;
Cf = ?Soft Hyphens?;
```

Thus, examples of identifiers are `a`, `theCharacter9`, `Next_Word`, `_tok`. Typically, only letters from the english alphabet are used as `letter`, and only _ is used for `specialChar`, but the full definition referes to the Unicode general categories described in Appendix B.3, and there are currently 19.345 possible Unicode code points in the `letter` category and 2.245 possible Unicode code points in the `specialChar` category.

Expressions are a central concept in F#. An expression can be a mathematical expression, such as $3*5$, a function application, such as $f3$, and many other things. Central in this chapter is the binding of values and functions to indentifiers, which is done with the keyword `let`, using the following simplified syntax, e.g., `let a = 1.0`.

Expressions has an enormous variety in how they may be written, we will in this book gradually work through some of the more important facets. For this we will extend the EBNF notation with ellipses: `...`, to denote that what is shown is part of the complete EBNF production rule. E.g., the part of expressions, we will discuss in this chapter is specified in EBNF by,

Listing 6.3: Simple expressions.

```
expr = ...
  | expr ":" type (*type annotation*)
  | expr ";" expr (*sequence of expressions*)
  | "let" valueDefn "in" expr (*binding a value or variable*)
  | "let" ["rec"] functionDefn "in" expr (*binding a function or operator*)
  | "fun" argumentPats "->" expr (*anonymous function*)
  | expr "<-" expr (*assingment*)

type = ...
  | longIdent (*named such as "int"*)

valueDefn = ["mutable"] pat "=" expr;

pat = ...
  | "_" (*wildcard*)
  | ident (*named*)
  | pat ":" type (*type constraint*)
  | "(" pat ")" (*paranthesized*)

functionDefn = identOrOp argumentPats [":" type] "=" expr;
argumentPats = pat | pat argumentPats;
```

In the following sections, we will work through this bit by bit.

6.1 Values

Binding identifiers to literals or expressions that are evaluated to be values, is called value-binding, and examples are `let a = 3.0` and `let b = cos 0.9`. On EBNF the simplified syntax,

Listing 6.4: Value binding expression.

```
expr = ...
  | "let" valueDefn "in" expr (*binding a value or variable*)
```

The `let` bindings defines relations between patterns `pat` and expressions `expr` for many different purposes. Most often the pattern is an identifier `ident`, which *`let`* defines to be an alias of the expression `expr`. The pattern may also be defined to have specific type using the *:* lexeme and a named `type`. The _ pattern is called the *wild card* pattern and, when it is in the value-binding, then the expression is evaluated but the result is discarded. The binding may be mutable as indicated by

· `let`
· :
· wild card

the keyword *mutable*, which will be discussed in Section 6.5, and the binding holds *lexically* for the last expression as indicated by the *in* keyword. For example, letting the identifier p be bound to the value 2.0 and using it in an expression is done as follows,

· mutable
· lexically
· in

Listing 6.5, letValue.fsx:
The identifier p is used in the expression following the in keyword.

```
let p = 2.0 in printfn "%A" (3.0 ** p)
```

```
9.0
```

F# will ignore most newlines between lexemes, i.e., the above is equivalent to writing,

Listing 6.6, letValueLF.fsx:
Newlines after in make the program easier to read.

```
let p = 2.0 in
printfn "%A" (3.0 ** p)
```

```
9.0
```

F# also allows for an alternative notation called *lightweight syntax*, where e.g., the in keyword is replaced with a newline, and the expression starts on the next line at the same column as let starts in, i.e., the above is equivalent to

· lightweight syntax

Listing 6.7, letValueLightWeight.fsx:
Lightweight syntax does not require the in keyword, but expression must be aligned with the let keyword.

```
let p = 2.0
printfn "%A" (3.0 ** p)
```

```
9.0
```

The same expression in interactive mode will also respond the inferred types, e.g.,

Listing 6.8, letValueLightWeightTypes.fsx:
Interactive mode also responds inferred types.

```
> let p = 2.0
- printfn "%A" (3.0 ** p);;
9.0

val p : float = 2.0
val it : unit = ()
```

By the val keyword in the line val p : float = 2.0 we see that p is inferred to be of type float and bound to the value `2.0`. The inference is based on the type of the right-hand-side, which is of type `float`. Identifiers may be defined to have a type using the `:` lexeme, but the types on the left-hand-side and right-hand-side of the `=` lexeme must be identical. I.e., mixing types gives an error,

Listing 6.9, letValueTypeError.fsx:
Binding error due to type mismatch.

```
let p : float = 3
printfn "%A" (3.0 ** p)
```

```
/Users/sporring/repositories/fsharpNotes/src/letValueTypeError.fsx(1,17):
    error FS0001: This expression was expected to have type
     float
but here has type
     int
```

Here, the left-hand-side is defined to be an identifier of type float, while the right-hand-side is a literal of type integer.

An expression can be a sequence of expressions separated by the lexeme `;`, e.g.,

Listing 6.10, letValueSequence.fsx:
A value-binding for a sequence of expressions.

```
let p = 2.0 in printfn "%A" p; printfn "%A" (3.0 ** p)
```

```
2.0
9.0
```

The lightweight syntax automatically inserts the `;` lexeme at newlines, hence using the lightweight syntax the above is the same as,

Listing 6.11, letValueSequenceLightWeight.fsx:
A value-binding for a sequence using lightweight syntax.

```
let p = 2.0
printfn "%A" p
printfn "%A" (3.0 ** p)
```

```
2.0
9.0
```

A key concept of programming is *scope*. In F#, the scope of a value-binding is lexically meaning that when F# determines the value bound to a name, it looks left and upward in the program text for the `let` statement defining it, e.g., · scope

Listing 6.12, letValueScopeLower.fsx:
Redefining identifiers is allowed in lower scopes.

```
let p = 3 in let p = 4 in printfn " %A" p;
```

```
 4
```

F# also has to option of using dynamic scope, where the value of a binding is defined by when it is used, and this will be discussed in Section 6.5.

Scopes are given levels, and scopes may be nested, where the nested scope has a level one lower than its parent.[1] F# distinguishes between the top and lower levels, and at the top level in the lightweight syntax, redefining values is not allowed, e.g.,

Listing 6.13, letValueScopeLowerError.fsx:
Redefining identifiers is not allowed in lightweight syntax at top level.

```
let p = 3
let p = 4
printfn "%A" p;
```

```
/Users/sporring/repositories/fsharpNotes/src/letValueScopeLowerError.fsx
    (2,5): error FS0037: Duplicate definition of value 'p'
```

But using parentheses, we create a *block*, i.e., a *nested scope*, and then redefining is allowed, e.g.,

· block
· nested scope

Listing 6.14, letValueScopeBlockAlternative3.fsx:
A block may be created using parentheses.

```
(
  let p = 3
  let p = 4
  printfn "%A" p
)
```

```
4
```

In both cases we used indentation, which is good practice, but not required here. Bindings inside are not available outside the nested scope, e.g.,

[1] Todo: **Drawings would be good to describe scope**

Listing 6.15, letValueScopeNestedScope.fsx:
Bindings inside a scope are not available outside.

```
let p = 3
(
  let q = 4
  printfn "%A" q
)
printfn "%A %A" p q
```

```
/Users/sporring/repositories/fsharpNotes/src/letValueScopeNestedScope.fsx
    (6,19): error FS0039: The value or constructor 'q' is not defined
```

Nesting is a natural part of structuring code, e.g., through function definitions to be discussed in Section 6.2 and flow control structures to be discussed in Chapter 8. Blocking code by nesting is a key concept for making robust code that is easy to use by others without the user necessarily needing to know the details of the inner workings of a block of code.

Defining blocks is useful for controlling the extend of a lexical scope of bindings. For example, adding a second `printfn` statement,

Listing 6.16, letValueScopeBlockProblem.fsx:
Overshadowing hides the first binding.

```
let p = 3 in let p = 4 in printfn "%A" p; printfn "%A" p
```

```
4
4
```

will print the value 4 last bound to the identifier `p`, since F# interprets the above as `let p = 3 in let p = 4 in (printfn "%A" p; printfn "%A" p)`. Had we intented to print the two different values of `p`, the we should have create a block as,

Listing 6.17, letValueScopeBlock.fsx:
Blocks allow for the return to the previous scope.

```
let p = 3 in (let p = 4 in printfn " %A" p); printfn " %A" p;
```

```
 4
 3
```

Here, the lexical scope of `let p = 4 in ...` is for the nested scope, which ends at `)`, returning to the lexical scope of `let p = 3 in ...`.

6.2 Non-recursive functions

A function is a mapping between an input and output domain. A key advantage of using functions, when programming, is that they *encapsulate code* into smaller units, that are easier to debug and may be reused. F# is a functional first programming language, and offers a number of alternative methods for specifying parameters, which will be discussed in this section. Binding identifiers to functions follows a syntax similar to value-binding,

· encapsulate code

Listing 6.18: Function binding expression

```
expr = ...
  | "let" functionDefn "in" expr (*binding a function or operator*)
```

Functions may also be recursive, which will be discussed in Chapter 8. An example in interactive mode is,

Listing 6.19, letFunction.fsx:
An example of a binding of an identifier and a function.

```
> let sum (x : float) (y : float) : float = x + y in
- let c = sum 357.6 863.4 in
- printfn "%A" c;;
1221.0

val sum : x:float -> y:float -> float
val c : float = 1221.0
val it : unit = ()
```

and we see that the function is interpreted to have the type `val sum : x:float -> y:float -> float`. The `->` lexeme means a mapping between sets, in this case floats. The function is also a higher order function, to be discussed in detail below, and here it suffices to think of `sum` as a function that takes 2 floats as argument and returns a float.

Not all types need to be declared, just sufficient for F# to be able to infer the types for the full statement. In the example, one specification is sufficient, and we could just have specified the type of the result,

Listing 6.20: All types need most often not be specified.

```
let sum x y : float = x + y
```

or even just one of the arguments,

Listing 6.21: Just one type is often enough for F# to infer the rest.

```
let sum (x : float) y = x + y
```

In both cases, since the `+` *operator* is only defined for *operands* of the same type, then when the type of either the result, any or both operands are declared, then the type of the remaining follows directly.

· operator
· operand

As for values, lightweight syntax automatically inserts the keyword `in` and the lexeme `;`,

**Listing 6.22, letFunctionLightWeight.fsx:
Lightweight syntax for function definitions.**

```
let sum x y : float = x + y
let c = sum 357.6 863.4
printfn "%A" c
```

```
1221.0
```

Arguments need not always be inferred to types, but may be of generic type, which F# prefers, when *type safety* is ensured, e.g., · type safety

**Listing 6.23, functionDeclarationGeneric.fsx:
Typesafety implies that a function will work for any type, and hence it is generic.**

```
> let second x y = y
- let a = second 3 5
- printfn "%A" a
- let b = second "horse" 5.0
- printfn "%A" b;;
5
5.0

val second : x:'a -> y:'b -> 'b
val a : int = 5
val b : float = 5.0
val it : unit = ()
```

Here, the function `second` does not use the first argument `x`, which therefore can be of any type, and which F# therefore calls `'a`, and the type of the second element, `y`, can also be of any type and not necessarily the same as `x`, so it is called `'b`. Finally the result is the same type as `y`, whatever it is. This is an example of a *generic function*, since it will work on any type. · generic function

A function may contain a sequence of expressions, but must return a value. E.g., the quadratic formula may be written as,

Listing 6.24, identifiersExampleAdvance.fsx:
A function may contain sequences of expressions.

```
let solution a b c sgn =
  let discriminant a b c =
    b ** 2.0 - 2.0 * a * c
  let d = discriminant a b c
  (-b + sgn * sqrt d) / (2.0 * a)

let a = 1.0
let b = 0.0
let c = -1.0
let xp = solution a b c +1.0
let xn = solution a b c -1.0
printfn "0 = %A * x ** 2.0 + %A * x + %A" a b c
printfn "  has solutions %A and %A" xn xp
```

```
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has solutions -0.7071067812 and 0.7071067812
```

Here, we used the lightweight syntax, where the `=` identifies the start of a nested scope, and F# identifies the scope by indentation. The amount of space used for indentation is does not matter, but all lines following the first must use the same. The scope ends before the first line with the previous indentation or none. Notice how the last expression is not bound to an identifier, but is the result of the function, i.e., in contrast to many other languages, F# does not have an explicit keyword for returning values, but requires a final expression, which will be returned to the caller of the function. Note also that since the function `discriminant` is defined in the nested scope of `solution`, then `discriminant` cannot be called outside `solution`, since the scope ends before `let a = 1.0`.

Lexical scope and function definitions can be a cause of confusion as the following example shows,[2] · lexical scope

Listing 6.25, lexicalScopeNFunction.fsx:
Lexical scope means that $f(z) = 3x$ and not $4x$ at the time of calling.

```
let testScope x =
  let a = 3.0
  let f z = a * z
  let a = 4.0
  f x
printfn "%A" (testScope 2.0)
```

```
6.0
```

Here, the value-binding for `a` is redefined, after it has been used to define a helper function `f`. So which value of `a` is used, when we later apply `f` to an argument? To resolve the confusion, remember that value-binding is lexically defined, i.e., the binding `let f z = a * x` uses the value of `a`, it has by the ordering of the lines in the script, not dynamically by when `f` was called. Hence, **think of lexical scope as substitution of an identifier with its value or function immediately at the place of definition.** I.e., since `a` and `3.0` are synonymous in the first lines of the program, then the function `f` is really defined as, `let f z = 3.0 * x`. Advice

[2]Todo: **Add a drawing or possibly a spell-out of lexical scope here.**

Functions do not need a name, but may be declared as an *anonymous function* using the `fun` keyword and the `->` lexeme, · anonymous function

Listing 6.26, functionDeclarationAnonymous.fsx:
Anonymous functions are functions as values.

```
let first = fun x y -> x
printfn "%d" (first 5 3)
```

```
5
```

Here, a name is bound to an anonymous function, which returns the first of two arguments. The difference to `let first x y = x` is that anonymous functions may be treated as values, meaning that they may be used as arguments to other functions, and new values may be reassigned to their identifiers, when mutable, as will be discussed in Section 6.5. A common use of anonymous functions is as as arguments to other functions, e.g.,

Listing 6.27, functionDeclarationAnonymousAdvanced.fsx:
Anonymous functions are often used as arguments for other functions.

```
let apply f x y  = f x y
let mul = fun a b -> a * b
printfn "%d" (apply mul 3 6)
```

```
18
```

Note that here `apply` is given 3 arguments, the function `mul` and 2 integers. It is not given the result of `mul 3 6`, since that would not match the definition of `apply`. **Anonymous functions and functions as arguments are powerfull concepts, but tend to make programs harder to read, and their use should be limited.** Advice

Functions may be declared from other functions

Listing 6.28, functionDeclarationCurrying.fsx:

```
let mul x y = x*y
let timesTwo = mul 2.0
printfn "%g" (mul 5.0 3.0)
printfn "%g" (timesTwo 3.0)
```

```
15
6
```

Here, `mul 2.0` is a partial specification of the function `mul x y`, where the first argument is fixed, and hence, `timesTwo` is a function of 1 argument being the second argument of `mul`. This notation is called *currying* in tribute of Haskell Curry, and Currying is often used in functional programming, but generally **currying should be used carefully, since currying may seriously reduce readability** · currying Advice

of code.

A *procedure* is a generalisation of the concept of functions, and in contrast to functions procedures need not return values, · procedure

Listing 6.29, procedure.fsx:
A procedure is a function that has no return value, which in F# implies() as return value.

```
let printIt a = printfn "This is '%A'" a
printIt 3
printIt 3.0
```

```
This is '3'
This is '3.0'
```

In F# this is automatically given the unit type as return value. Procedural thinking is useful for *encapsulation* of scripts, but is prone to *side-effects* and should be minimized by being replaced by functional thinking. More on side-effects in Section 6.5. **Procedural thinking is useful for encapsulation, but is prone to side-effects and should be minimized by being replaced by functional thinking.**

· encapsulation
· side-effects
Advice

6.3 User-defined operators

Operators are functions, and in F#, the infix multiplication operator `+` is equivalent to the function `(+)`, e.g.,

Listing 6.30, addOperatorNFunction.fsx:

```
let a = 3.0
let b = 4.0
let c = a + b
let d = (+) a b
printfn "%A plus %A is %A and %A" a b c d
```

```
3.0 plus 4.0 is 7.0 and 7.0
```

All operator has this option, and you may redefine them and define your own operators, but in F# names of user-defined operators are limited by the following simplified EBNF:

Listing 6.31: Grammar for infix and prefix lexemes

```
infixOrPrefixOp = "+" | "-" | "+." | "-." | "%" | "&" | "&&";
prefixOp = infixOrPrefixOp | "~" {"~"} | "!" {opChar} - "!=";
infixOp =
  {"."} (
    infixOrPrefixOp
    | "-" {opChar}
    | "+" {opChar}
    | "||"
    | "<" {opChar}
    | ">" {opChar}
    | "="
    | " |" {opChar}
    | "&" {opChar}
    | "^" {opChar}
    | "*" {opChar}
    | "/" {opChar}
    | "%" {opChar}
    | "!=" )
  | ":=" | "::" | "$" | "?";
opChar =
  "!" | "%" | "&" | "*" | "+" | "-" | ". " | "/"
  | "<" | "=" | ">" | "@" | "^" | "|" | "~";
```

The precedence rules and associativity of user-defined operators follows the rules for which they share prefixes with built-in rules, see Table E.6. E.g., `.*`, `+++`, and `<+` are valid operator names for infix operators, they have precedence as ordered, and their associativity are all left. Using `~` as the first character in the definition of an operator makes the operator unary and will not be part of the name. Examples of definitions and use of operators are,

Listing 6.32, operatorDefinitions.fsx:

```
let (.*) x y = x * y + 1
printfn "%A" (3 .* 4)
let (+++) x y = x * y + y
printfn "%A" (3 +++ 4)
let (<+) x y = x < y + 2.0
printfn "%A" (3.0 <+ 4.0)
let (~+.) x = x+1
printfn "%A" (+.1)
```

```
13
16
true
2
```

Operators beginning with `*` must use a space in its definition, `( *` in order for it not to be confused with the beginning of a comment `(*`, see Chapter 7 for more on comments in code.

Beware, redefining existing operators lexically redefines all future uses of the operators for all types, hence **it is not a good idea to redefine operators, but better to define new.** In Chapter 20 Advice we will discuss how to define type specific operators including prefix operators.

6.4 The Printf function

A common way to output information to the console is to use one of the family of *printf* commands. These functions are special, since they take a variable number of arguments, and the number is decided by the first - the format string, · printf

Listing 6.33: printf statement.

```
"printf" formatString {ident}
```

where a formatString is a string (simple or verbatim) with placeholders. The function printf prints formatString to the console, where all placeholder has been replaced by the value of the corresponding argument formatted as specified, e.g., in `printfn "1 2 %d" 3` the formatString is `"1 2 %d"`, and the placeholder is `%d`, and the `printf` replaced the placeholder with the value of the corresponding argument, and the result is printed to the console, in this case `1 2 3`. Possible formats for the placeholder are,

Listing 6.34: Placeholders in formatString for printf functions.

```
placeholder = "%%" | ("%" [flags] [width] ["." precision] specifier) (* No
    spaces between rules *)
flags = ["0"] ["+"] [SP] (* No spaces between rules *)
width = ["-"] ("*" | [dInt]) (* No spaces between rules *)
specifier = "b" | "d" | "i" | "u" | "x" | "X" | "o" | "e" | "E" | "f" | "F" |
    "g" | "G" | "M" | "O" | "A" | "a" | "t"
```

There are specifiers for all the basic types and more as elaborated in Table 6.1. The placeholder can be given a specified with, either by setting a specific integer, or using the `*` character, indicating that the with is given as an argument prior to the replacement value. Default is for the value to be right justified in the field, but left justification can be specified by the `-` character. For number types, you can specify their format by: `"0"` for padding the number with zeros to the left, when righ justifying the number; `"+"` to explicitely show a plus sign for positive numbers; `SP` to enforce a space, where there otherwise would be a plus sign for positive numbers. For floating point numbers, the precision integer specifies the number of digits displayed of the fractional part. Examples of some of these combinations are,

Specifier	Type	Description
%b	bool	Replaces with boolean value
%s	string	
%c	char	
%d, %i	basic integer	
%u	basic unsigned integers	
%x	basic integer	formatted as unsigned hexadecimal with lower case letters
%X	basic integer	formatted as unsigned hexadecimal with upper case letters
%o	basic integer	formatted as unsigned octal integer
%f, %F,	basic floats	formatted on decimal form
%e, %E,	basic floats	formatted on scientific form. Lower case uses "e" while upper case uses "E" in the formatting.
%g, %G,	basic floats	formatted on the shortest of the corresponding decimal or scientific form.
%M	decimal	
%O	Objects ToString method	
%A	any built-in types	Formatted as a literal type
%a	Printf.TextWriterFormat ->'a -> ()	
%t	(Printf.TextWriterFormat -> ()	

Table 6.1: Printf placeholder string

Listing 6.35, printfExample.fsx:
Examples of printf and some of its formatting options.

```
let pi = 3.1415192
let hello = "hello"
printf "An integer: %d\n" (int pi)
printf "A float %f on decimal form and on %e scientific form, and a char
    '%c'\n" pi pi
printf "A char '%c' and a string \"%s\"\n" hello.[0] hello
printf "Float using width 8 and 1 number after the decimal:\n"
printf "  \"%8.1f\" \"%8.1f\"\n" pi -pi
printf "  \"%08.1f\" \"%08.1f\"\n" pi -pi
printf "  \"% 8.1f\" \"% 8.1f\"\n" pi -pi
printf "  \"%-8.1f\" \"%-8.1f\"\n" pi -pi
printf "  \"%+8.1f\" \"%+8.1f\"\n" pi -pi
printf "  \"%8s\"\n\"%-8s\"\n" "hello" "hello"
```

```
An integer: 3
A char 'h' and a string "hello"
Float using width 8 and 1 number after the decimal:
  "     3.1" "    -3.1"
  "000003.1" "-00003.1"
  "     3.1" "    -3.1"
  "3.1     " "-3.1    "
  "    +3.1" "    -3.1"
  "   hello"
"hello   "
```

Function	Example	Description
printf	printf "%d apples" 3	Prints to the console, i.e., stdout
printfn		as printf and adds a newline.
fprintf	fprintf stream "%d apples" 3	Prints to a stream, e.g., stderr and stdout , which would be the same as printf and eprintf.
fprintfn		as fprintf but with added newline.
eprintf	eprintf "%d apples" 3	Print to stderr
eprintfn		as eprintf but with added newline.
sprintf	printf "%d apples" 3	Return printed string
failwithf	failwithf "%d failed apples" 3	prints to a string and used for raising an exception.

Table 6.2: The family of printf functions.

Not all combinations of flags and identifier types are supported, e.g., strings cannot have number of integers after the decimal specified. The placeholder types "%A", "%a", and "%t" are special for F#, examples of their use are,

Listing 6.36, printfExampleAdvance.fsx:

```
let noArgument writer = printf "I will not print anything"
let customFormatter writer arg = printf "Custom formatter got: \"%A\"" arg
printf "Print examples: %A, %A, %A\n" 3.0m 3uy "a string"
printf "Print function with no arguments: %t\n" noArgument
printf "Print function with 1 argument: %a\n" customFormatter 3.0
```

```
Print examples: 3.0M, 3uy, "a string"
Print function with no arguments: I will not print anything
Print function with 1 argument: Custom formatter got: "3.0"
```

The %A is special in that all built-in types including tuples, lists, and arrays to be discussed in Chapter 9 can be printed using this formatting string, but notice that the formatting performed includes the named literal string. The two formatting strings %t and %a are options for user-customizing the formatting, and will not be discussed further.

Beware, formatString is not a string but a Printf.TextWriterFormat, so to predefine a formatString as, e.g., let str = "hello %s" in printf str "world" will be a type error.

The family of printf is shown in Table 6.2. The function fprintf prints to a stream, e.g., stderr and stdout, of type System.IO.TextWriter. Streams will be discussed in further detail in Chapter 12. The function failwithf is used with exceptions, see Chapter 11 for more details. The function has a number of possible return value types, and for testing the *ignore* function ignores it all, e.g., ignore (failwithf "%d failed apples" 3) · ignore

6.5 Variables

The mutable in let bindings means that the identifier may be rebound to a new value using the <- · <-

lexeme with the following syntax,[3]

Listing 6.37: Value reassignment for mutable variables.

```
expr = ...
  | expr "<-" expr (*assingment*)
```

Mutable data is synonymous with the term *variable*. A variable is an area in the computers working memory associated with an identifier and a type, and this area may be read from and written to during program execution. For example,

· Mutable data
· variable

Listing 6.38, mutableAssignReassingShort.fsx:
A variable is defined and later reassigned a new value.

```
let mutable x = 5
printfn "%d" x
x <- -3
printfn "%d" x
```

```
5
-3
```

Here, an area in memory was denoted `x`, initially assigned the integer value 5, hence the type was inferred to be `int`. Later, this value of `x` was replaced with another integer using the `<-` lexeme. The `<-` lexeme is used to distinguish the assignment from the comparison operator, i.e., if we by mistake had written,

· `<-`

Listing 6.39, mutableEqual.fsx:
Common error - mistaking = and <- lexemes for mutable variables. The former is the test operator, while the latter is the assignment expression.

```
> let mutable a = 0
- a = 3;;

val mutable a : int = 0
val it : bool = false
```

then we instead would have obtained the default assignment of the result of the comparison of the content of `a` with the integer 3, which is false. However, it is important to note, that when the variable is initially defined, then the '=' operator must be used, while later reassignments must use the `<-` expression.

Assignment type mismatches will result in an error,

[3] Todo: **Discussion on heap and stack should be added here.**

Listing 6.40, mutableAssignReassingTypeError.fsx:
Assignment type mismatching causes a compile time error.

```
let mutable x = 5
printfn "%d" x
x <- -3.0
printfn "%d" x
```

```
/Users/sporring/repositories/fsharpNotes/src/
    mutableAssignReassingTypeError.fsx(3,6): error FS0001: This expression
     was expected to have type
     int
but here has type
     float
```

I.e., once the type of an identifier has been declared or inferred, then it cannot be changed.

A typical variable is a counter of type integer, and a typical use of counters is to increment them, for example,

Listing 6.41, mutableAssignIncrement.fsx:
Variable increment is a common use of variables.

```
let mutable x = 5 // Declare a variable x and assign the value 5 to it
printfn "%d" x
x <- x + 1 // Assign a new value -3 to x
printfn "%d" x
```

```
5
6
```

Variables implement dynamic scope, e.g., in comparison with the lexical scope, where the value of an identifier depends on which line in the program, an identifier is defined, dynamic scope depends on, when it is used. E.g., the script in Listing 6.25 defines a function using lexical scope and returns the number `6.0`, however, if `a` is made `mutable`, then the behaviour is different:

Listing 6.42, dynamicScopeNFunction.fsx:
Mutual variables implement dynamics scope rules. Compare with Listing 6.25.

```
let testScope x =
  let mutable a = 3.0
  let f z = a * x
  a <- 4.0
  f x
printfn "%A" (testScope 2.0)
```

```
8.0
```

Here, the respons is `8.0`, since the value of `a` changed befor the function `f` was called.

Variables cannot be returned from functions, that is,

Listing 6.43, mutableAssignReturnValue.fsx:

```
let g () =
  let x = 0
  x
printfn "%d" (g ())
```

```
0
```

declares a function that has no arguments and returns the value 0, while the same for a variable is invalid,

Listing 6.44, mutableAssignReturnVariable.fsx:

```
let g () =
  let mutual x = 0
  x
printfn "%d" (g ())
```

```
/Users/sporring/repositories/fsharpNotes/src/mutableAssignReturnVariable.
    fsx(3,3): error FS0039: The value or constructor 'x' is not defined
```

There is a workaround for this by using *reference cells* by the build-in function `ref` and operators `!` and `:=`,

· reference cells

Listing 6.45, mutableAssignReturnRefCell.fsx:

```
let g () =
  let x = ref 0
  x
let y = g ()
printfn "%d" !y
y := 3
printfn "%d" !y
```

```
0
3
```

That is, the `ref` function creates a reference variable, the '`!`' and the '`:=`' operators reads and writes its value. Reference cells are in some language called pointers, and their use is strongly discouraged, since they may cause *side-effects*, which is the effect that one function changes the state of another, such as the following example demonstrates, · side-effects

Listing 6.46, mutableAssignReturnSideEffect.fsx:

```
let updateFactor factor =
  factor := 2

let multiplyWithFactor x =
  let a = ref 1
  updateFactor a
  !a * x

printfn "%d" (multiplyWithFactor 3)
```

```
6
```

In the example, the function `updateFactor` changes a variable in the scope of `multiplyWithFactor`, which is prone to errors, since the style of programming does not follow the usual assignment syntax. Better style of programming is,

Listing 6.47, mutableAssignReturnWithoutSideEffect.fsx:

```
let updateFactor () =
  2

let multiplyWithFactor x =
  let a = ref 1
  a := updateFactor ()
  !a * x

printfn "%d" (multiplyWithFactor 3)
```

```
6
```

Here, there can be no doubt in `multiplyWithFactor` that the value of 'a' is changing. Side-effects do have their use, but should in general be avoided at almost all costs, and in general it is advised to **refrain from using ref cells**.[4] Advice

[4] Todo: **Add something about mutable functions**

Chapter 7

In-code documentation

Documentation is a very important part of writing programs, since it is most unlikely, that you will be writing really obvious code. And what seems obvious at the point of writing may be mystifying months later to the author and to others. The documentation serves several purposes:

1. Communicate what the code should be doing
2. Highlight big insights essential for the code
3. Highlight possible conflicts and/or areas, where the code could be changed later

The essential point is that coding is a journey in problem solving, and proper documentation is an aid in understanding the solution and the journey that lead to it. Documentation is most often a mixture between in-code documentation and accompanying documents. Here, we will focus on in-code documentation, but arguably this does cause problems in multi-language environments, and run the risk of bloating code.

F# has the following simplified syntax for in-code documentation,

Listing 7.1: Comments.

```
blockComment = "(*" {codePoint} "*)";
lineComment = "//" {codePoint - newline} newline;
```

That is, text framed as a `blockComment` is still parsed by F# as keywords and basic types implying that `(* a comment (* in a comment *) *)` and `(* "*)" *)` are valid comments, while `(* " *)` is invalid.[1]

The F# compiler has an option for generating *Extensible Markup Language* (*XML*) files from scripts using the C# documentation comments tags[2]. The XML documentation starts with a triple-slash `///`, i.e., a lineComment and a slash, which serves as comments for the code construct, that follows immediately after. XML consists of tags which always appears in pairs, e.g., the tag “tag” would look like <tag> ... </tag>. The F# accept any tags, but recommends those listed in Table 7.1. If no tags are used, then it is automatically assumed to be a <summary>. An example of a documented script is,

· Extensible Markup Language
· XML

[1] Todo: **lstlisting colors is bad.**

[2] For specification of C# documentations comments see ECMA-334 3rd Edition, Annex E, Section 2: `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`

Tag	Description
<c>	Set text in a code-font.
<code>	Set one or more lines in code-font.
<example>	Set as an example.
<exception>	Describe the exceptions a function can throw.
<list>	Create a list or table.
<para>	Set text as a paragraph.
<param>	Describe a parameter for a function or constructor.
<paramref>	Identify that a word is a parameter name.
<permission>	Document the accessibility of a member.
<remarks>	Further describe a function.
<returns>	Describe the return value of a function.
<see>	Set as link to other functions.
<seealso>	Generate a See Also entry.
<summary>	Main description of a function or value.
<typeparam>	Describe a type parameter for a generic type or method.
<typeparamref>	Identify that a word is a type parameter name.
<value>	Describe a value.

Table 7.1: Recommended XML tags for documentation comments, from ECMA-334 3rd Edition, Annex E, Section 2.

Listing 7.2, commentExample.fsx:
Code with XML comments.

```
/// The discriminant of a quadratic equation with parameters a, b, and c
let discriminant a b c = b ** 2.0 - 2.0 * a * c

/// <summary>Find x when 0 = ax^2+bx+c.</summary>
/// <remarks>Negative discriminant are not checked.</remarks>
/// <example>
///   The following code:
///   <code>
///     let a = 1.0
///     let b = 0.0
///     let c = -1.0
///     let xp = (solution a b c +1.0)
///     printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
///   </code>
///   prints <c>0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7</c> to the console.
/// </example>
/// <param name="a">Quadratic coefficient.</param>
/// <param name="b">Linear coefficient.</param>
/// <param name="c">Constant coefficient.</param>
/// <param name="sgn">+1 or -1 determines the solution.</param>
/// <returns>The solution to x.</returns>
let solution a b c sgn =
  let d = discriminant a b c
  (-b + sgn * sqrt d) / (2.0 * a)

let a = 1.0
let b = 0.0
let c = -1.0
let xp = (solution a b c +1.0)
printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
```

```
0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7
```

Mono's `fsharpc` command may be used to extract the comments into an XML file,

Listing 7.3, Converting in-code comments to XML.

```
$ fsharpc --doc:commentExample.xml commentExample.fsx
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
```

This results in an XML file with the following content,

Listing 7.4, An XML file generated by `fsharpc`.

```
<?xml version="1.0" encoding="utf-8"?>
<doc>
<assembly><name>commentExample</name></assembly>
<members>
<member name="M:CommentExample.solution(System.Double,System.Double,System
    .Double,System.Double)">
 <summary>Find x when 0 = ax^2+bx+c.</summary>
 <remarks>Negative discriminant are not checked.</remarks>
 <example>
   The following code:
   <code>
     let a = 1.0
     let b = 0.0
     let c = -1.0
     let xp = (solution a b c +1.0)
     printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
   </code>
   prints <c>0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7</c> to the console.
 </example>
 <param name="a">Quadratic coefficient.</param>
 <param name="b">Linear coefficient.</param>
 <param name="c">Constant coefficient.</param>
 <param name="sgn">+1 or -1 determines the solution.</param>
 <returns>The solution to x.</returns>
</member>
<member name="M:CommentExample.discriminant(System.Double,System.Double,
    System.Double)">
<summary>
 The discriminant of a quadratic equation with parameters a, b, and c
</summary>
</member>
</members>
</doc>
```

The extracted XML is written in C# type by convention, since F# is part of the Mono and .Net framework that may be used by any of the languages using Assemblies. Besides the XML inserted in the script, the XML has added `<?xml ...>` header, `<doc>`, `<assembly>`, `<members>`, and `<member>` tags. The header and the `<doc>` tag are standards for XML. The extracted XML is geared towards documenting big libraries of codes and thus highlights the structured programming organization, see Part IV, and `<assembly>`, `<members>`, and `<member>` are indications for where the functions belong in the hierarchy. As an example, the prefix `M:CommentExample.` means that it is a method in the namespace CommentExample, which in this case is the name of the file. Further, the function type val `solution : a:float -> b:float -> c:float -> sgn:float -> float` is in the XML documentation `M:CommentExample.solution(System.Double,System.Double,System.Double,System.Double)`, which is the C# equivalent.

An accompanying program in the Mono suite is `mdoc`, whose primary use is to perform a syntax analysis of an assembly and generate a scaffold XML structure for an accompanying document. With the `-i` flag, it is further possible to include the in-code comments as initial descriptions in the XML. The XML may be updated gracefully by `mdoc` as the code develops, without destroying manually entered documentation in the accompanying documentation. Finally, the XML may be exported to HTML

The primary use of the `mdoc` command is to analyze compiled code and generate an empty XML structure with placeholders to describe functions, values, and variables. This structure can then be updated and edited as the program develops. The edited XML files can then be exported to *Hyper Text Markup Language* (*HTML*) files, which can be viewed in any browser. Using the console, all of this is accomplished by,

· Hyper Text Markup Language
· HTML

Listing 7.5, Converting an XML file to HTML.

```
$ mdoc update -o commentExample -i commentExample.xml commentExample.exe
New Type: CommentExample
Member Added: public static double determinant (double a, double b, double
     c);
Member Added: public static double solution (double a, double b, double c,
     double sgn);
Member Added: public static double a { get; }
Member Added: public static double b { get; }
Member Added: public static double c { get; }
Member Added: public static double xp { get; }
Namespace Directory Created:
New Namespace File:
Members Added: 6, Members Deleted: 0
$ mdoc export-html -out commentExampleHTML commentExample
.CommentExample
```

The primary use of the `mdoc` command is to analyze compiled code and generate an empty XML structure with placeholders to describe functions, values, and variables. This structure can then be updated and edited as the program develops. The edited XML files can then be exported to HTML files, which can be viewed in any browser, an example of which is shown in Figure 7.1. A full description of how to use `mdoc` is found here[3].

[3] http://www.mono-project.com/docs/tools+libraries/tools/monodoc/generating-documentation/

solution Method

Find x when 0 = ax^2+bx+c.

Syntax

[Microsoft.FSharp.Core.CompilationArgumentCounts(Mono.Cecil.CustomAttributeArgument[])]
public static double **solution** (double a, double b, double c, double sgn)

Parameters

a
: Quadratic coefficient.

b
: Linear coefficient.

c
: Constant coefficient.

sgn
: +1 or -1 determines the solution.

Returns

The solution to x.

Remarks

Negative discriminant are not checked.

Example

The following code:

Example
```
let a = 1.0
let b = 0.0
let c = -1.0
let xp = (solution a b c +1.0)
printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
```

prints `0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7` to the console.

Requirements

Namespace:
Assembly: commentExample (in commentExample.dll)
Assembly Versions: 0.0.0.0

Figure 7.1: Part of the HTML documentation as produce by `mdoc` and viewed in a browser.

Chapter 8

Controlling program flow

Non-recursive functions encapsulates code and allows for some control of flow, that is, if there is a piece of code, which we need to to have executed many times, then we can encapsulate it in the body of a function, and then call the function several times. In this chapter, we will look at more general control of flow via loops, conditional execution, and recursion, and therefore we look at further extension of the `expr` rule,

Listing 8.1: Expressions for controlling the flow of execution.

```
expr = ...
  | "if" expr "then" expr {"elif" expr "then" expr} ["else" expr] (*
    conditional*)
  | "while" expr "do" expr ["done"] (*while loop*)
  | "for" ident "=" expr "to" expr "do" expr ["done"] (*simple for loop*)
  | "let" functionDefn "in" expr (*binding a function or operator*)
  | "let" "rec" functionDefn {"and" functionDefn} "in" expr (*recursive fcts*)
```

8.1 For and while loops

Many programming constructs need to be repeated, and F# contains many structures for repetition such as the `for` and `while` loops, which have the syntax,

Listing 8.2: `for`- and `while`-loops.

```
expr = ...
  | "while" expr "do" expr ["done"] (*while loop*)
  | "for" ident "=" expr "to" expr "do" expr ["done"] (*simple for loop*)
```

As an example, consider counting from 1 to 10 with a *`for`*-loop, · `for`

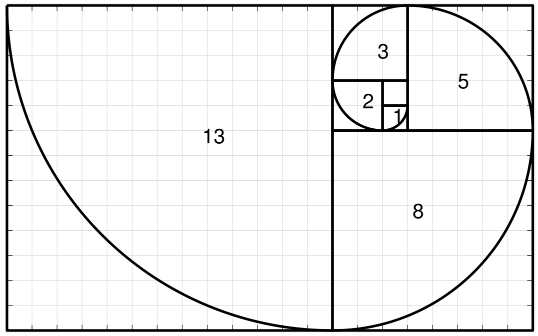

Figure 8.1: The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with radius of the square it is drawn in

Listing 8.3, count.fsx:
Counting from 1 to 10 using a for-loop.

```
> for i = 1 to 10 do printf "%d " i done;
- printfn "";;
1 2 3 4 5 6 7 8 9 10

val it : unit = ()
```

As this interactive script demonstrates, the identifier `i` takes all the values between 1 and 10, but in spite of its changing state, it is not mutable. Note also that the return value of the `for` expression is `()` like the `printf` functions. Using lightweight syntax the block following the *`do`* keyword up to and including the *`done`* keyword may be replaced by a newline and indentation, e.g.,

· do
· done

Listing 8.4, countLightweight.fsx:
Counting from 1 to 10 using a for-loop, see Listing 8.3.

```
for i = 1 to 10 do
  printf "%d " i
printfn ""
```

```
1 2 3 4 5 6 7 8 9 10
```

A more complicated example is,

Problem 8.1:
Write a program that calculates the n'th Fibonacci number.

The Fibonacci numbers is the series of numbers $1, 1, 2, 3, 5, 8, 13 \ldots$, where the $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$, and they are related to Golden spirals shown in Figure 8.1. We could solve this problem with a `for`-loop as follows,

Listing 8.5, fibFor.fsx:
The n'th Fibonacci number is the sum of the previous 2.

```
let fib n =
  let mutable prev = 1
  let mutable current = 1
  let mutable next = 0
  for i = 3 to n do
    next <- current + prev
    prev <- current
    current <- next
  next

printfn "fib(1) = 1"
printfn "fib(2) = 1"
for i = 3 to 10 do
  printfn "fib(%d) = %d" i (fib i)
```

```
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
```

The basic idea of the solution is that if we are given the $(n-1)$'th and $(n-2)$'th numbers, then the n'th number is trivial to compute. And assume that fib(1) and fib(2) are given, then it is trivial to calculate the fib(3). For the fib(4) we only need fib(3) and fib(2), hence we may disregard fib(1). Thus we realize, that we can cyclicly update the previous, current and next values by shifting values until we have reached the desired fib(n).

The *while*-loop is simpler than the `for`-loop and does not contain a builtin counter structure. Hence, if we are to repeat the count-to-10 program from Listing 8.3 example, it would look somewhat like, · `while`

Listing 8.6, countWhile.fsx:
Count to 10 with a counter variable.

```
let mutable i = 1 in while i <= 10 do printf "%d " i; i <- i + 1 done;
printf "\n"
```

```
1 2 3 4 5 6 7 8 9 10
```

or equivalently using the lightweight syntax,

Listing 8.7, countWhileLightweight.fsx:
Count to 10 with a counter variable using lightweight syntax, see Listing 8.6.

```
let mutable i = 1
while i <= 10 do
  printf "%d " i
  i <- i + 1
printf "\n"
```

```
1 2 3 4 5 6 7 8 9 10
```

In this case, the for-loop is to be preferred, since more lines of code typically means more chances of making a mistake. But the while-loop allows for other logical structures. E.g., lets find the biggest Fibonacci number less than 100,

Listing 8.8, fibWhile.fsx:
Search for the largest Fibonacci number less than a specified number.

```
let largestFibLeq n =
  let mutable prev = 1
  let mutable current = 1
  let mutable next = 0
  while next <= n do
    next <- prev + current
    prev <- current
    current <- next
  prev

printfn "largestFibLeq(1) = 1"
printfn "largestFibLeq(2) = 1"
for i = 3 to 10 do
  printfn "largestFibLeq(%d) = %d" i (largestFibLeq i)
```

```
largestFibLeq(1) = 1
largestFibLeq(2) = 1
largestFibLeq(3) = 3
largestFibLeq(4) = 3
largestFibLeq(5) = 5
largestFibLeq(6) = 5
largestFibLeq(7) = 5
largestFibLeq(8) = 8
largestFibLeq(9) = 8
largestFibLeq(10) = 8
```

Thus, while-loops are most often used, when the number of iteration cannot easily be decided, when entering the loop.

Both for- and while-loops are often associated with variables, i.e., values that change while looping. If one mistakenly used values and rebinding, then the result would in most cases be of little use, e.g.,

Listing 8.9, forScopeError.fsx:
Lexical scope error. While rebinding is valid F# syntax, has little effect due to lexical scope.

```
let a = 1
for i = 1 to 10 do
  let a = a + 1
  printf "(%d, %d) " i a
printf "\n"
```

```
(1, 2) (2, 2) (3, 2) (4, 2) (5, 2) (6, 2) (7, 2) (8, 2) (9, 2) (10, 2)
```

I.e., the `let` expression rebinds `a` every iteration of the loop, but the value on the right-hand-side is taken lexically from above, where `a` has the value 1, so every time the result is the value 2.

8.2 Conditional expressions

Consider the task,

Problem 8.2:

Write a function that given n writes the sentence, "I have n apple(s)", where the plural 's' is added appropriately.

For this we need to test the value of n, and one option is to use conditional expressions. Conditional expression has the syntax, The grammar for conditional expressions is,

Listing 8.10: Conditional expressions.

```
expr = ...
  | "if" expr "then" expr {"elif" expr "then" expr} ["else" expr] (*
    conditional*)
```

and an example using conditional expressions to solve the above problem is,

Listing 8.11, conditionalLightweight.fsx:
Using conditional expression to generate different strings.

```
let applesIHave n =
  if n < -1 then
    "I owe " + (string -n) + " apples"
  elif n < 0 then
    "I owe " + (string -n) + " apple"
  elif n < 1 then
    "I have no apples"
  elif n < 2 then
    "I have 1 apple"
  else
    "I have " + (string n) + " apples"

printfn "%A" (applesIHave -3)
printfn "%A" (applesIHave -1)
printfn "%A" (applesIHave 0)
printfn "%A" (applesIHave 1)
printfn "%A" (applesIHave 2)
printfn "%A" (applesIHave 10)
```

```
"I owe 3 apples"
"I owe 1 apple"
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

The `expr` following *`if`* and *`elif`* are *conditions*, i.e., expressions that evaluate to a boolean value. The `expr` following *`then`* and *`else`* are called *branches*, and all branches must have identical type, such that regardless which branch is chosen, then the type of the result of the conditional expression is the same. The result of the conditional expression is the first branch, for which its condition was true.

· `if`
· `elif`
· conditions
· `then`
· `else`
· branches

The sentence structure and its variants gives rise to a more compact solution, since the language to be returned to the user is a variant of "I have/or no/number apple(s)", i.e., under certain conditions should the sentence use “have” and “owe” etc.. So we could instead make decisions on each of these sentence parts and then built the final sentence from its parts. This is accomplished in the following example:

Listing 8.12, conditionalLightweightAlt.fsx:
Using sentence parts to construct the final sentence.

```
let applesIHave n =
  let haveOrOwe = if n < 0 then "owe" else "have"
  let pluralS = if (n = 0) || (abs n) > 1 then "s" else ""
  let number = if n = 0 then "no" else (string (abs n))

  "I " + haveOrOwe + " " + number + " apple" + pluralS

printfn "%A" (applesIHave -3)
printfn "%A" (applesIHave -1)
printfn "%A" (applesIHave 0)
printfn "%A" (applesIHave 1)
printfn "%A" (applesIHave 2)
printfn "%A" (applesIHave 10)
```

```
"I owe 3 apples"
"I owe 1 apple"
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

While arguably shorter, this solution is also more dense, and for a small problem like this, it is most likely more difficult to debug and maintain.

Note that both `elif` and `else` branches are optional, which may cause problems. For example, both `let a = if true then 3` and `let a = if true then 3 elif false then 4` will be invalid, since F# is not smart enough to realize that the type of the expression is uniquely determined. Instead F# looks for the `else` to ensure all cases have been covered, and that `a` always will be given a unique value of the same type regardless of the branches taken in the conditional statement, hence, `let a = if true then 3 else 4` is the only valid expression of the 3. In practice, F# assumes that the omitted branches returns `()`, and thus it is fine to say `let a = if true then ()` and `if true then printfn "hej"`. Nevertheless, it is good practice in F# always to include and `else` branch.

8.3 Recursive functions

Recursion is a central concept in F#. A *recursive function* is a function, which calls itself. From a compiler point of view, this is challenging, since the function is used before the compiler has completed its analysis. However, for this there is a technical solution, and we will just concern ourselves with the logics of using recursion for programming. The syntax for defining recursive functions in F# is,

· recursive function

Listing 8.13: Recursive functions.

```
expr = ...
  | "let" "rec" functionDefn {"and" functionDefn} "in" expr
```

An example of a recursive function that counts from 1 to 10 similarly to Listing 8.3 is,[1]

Listing 8.14, countRecursive.fsx:
Counting to 10 using recursion.

```
let rec prt a b =
  if a > b then
    printf "\n"
  else
    printf "%d " a
    prt (a + 1) b

prt 1 10
```

```
1 2 3 4 5 6 7 8 9 10
```

Here the `prt` calls itself repeatedly, such that the first call is `prt 1 10`, which calls `prt 2 10`, and so on until the last call `prt 10 10`. Calling `prt 11 10` would not result in recursive calls, since when `a` is higher than `10` then the *stopping criterium* is met and a newline is printed. For values of `a` smaller than or equal `b` then the recursive branch is executed. Since `prt` calls itself at the end of the recursion branch, then this is a *tail-recursive* function. Most compilers achieve high efficiency in terms of speed and memory, so **prefer tail-recursion whenever possible.** Using recursion to calculate the Fibonacci number as Listing 8.5.

· stopping criterium
· tail-recursive
Advice

Listing 8.15, fibRecursive.fsx:
The n'th Fibonacci number using recursive.

```
let rec fib n =
  if n < 1 then
    0
  elif n = 1 then
    1
  else
    fib (n - 1) + fib (n - 2)

for i = 0 to 10 do
  printfn "fib(%d) = %d" i (fib i)
```

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
```

[1] Todo: **A drawing showing the stack for the example would be good.**

Here we used the fact that including $\mathrm{fib}(0) = 0$ in the Fibonacci series also produces it using the rule $\mathrm{fib}(n) = \mathrm{fib}(n-2) + \mathrm{fib}(n-1),\ n \geq 0$, which allowed us to define a function that is well defined for the complete set of integers. I.e., a negative argument returns 0. This is a general advice: **make functions that fails gracefully.** Advice

Functions that recursively call each other are called *mutually recursive* functions. F# offers the let-rec-and notation for co-defining mutually recursive functions. As an example, consider the function `even : int -> bool`, which returns true if its argument is even and false otherwise, and the opposite function `odd : int -> bool`. A mutually recursive implementation of these functions can be developed from the following statements: `even 0 = true`, `odd 0 = false`, and `even n = odd (n-1)`: · mutually recursive

Listing 8.16, mutuallyRecursive.fsx:
Using mutual recursion to implement even and odd functions.

```
let rec even x =
  if x = 0 then true
  else odd (x - 1)
and odd x =
  if x = 0 then false
  else even (x - 1);;

let w = 5;
printfn "%*s %*s %*s" w "i" w "even" w "odd"
for i = 1 to w do
  printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
    i  even   odd
    1 false  true
    2  true false
    3 false  true
    4  true false
    5 false  true
```

Notice that in the lightweight notation used here, that the `and` must be on the same indentation level as the original `let`.

Without the `and` keyword, F# will return an error at the definition of `even`. However, it is possible to implement mutual recursion by using functions as an argument, e.g.,

Listing 8.17, mutuallyRecursiveAlt.fsx:
Mutual recursion without the `and` keyword needs a helper function.

```
let rec evenHelper (notEven: int -> bool) x =
  if x = 0 then true
  else notEven (x - 1)

let rec odd x =
  if x = 0 then false
  else evenHelper odd (x - 1);;

let even x = evenHelper odd x

let w = 5;
printfn "%*s %*s %*s" w "i" w "Even" w "Odd"
for i = 1 to w do
  printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
    i  Even   Odd
    1 false  true
    2  true false
    3 false  true
    4  true false
    5 false  true
```

But, Listing 8.16 is clearly to be preferred over Listing 8.17.

In the above we used the `even` and `odd` function problems to demonstrate mutual recursion. There is, of course, a much simpler solution, which does not use recursion at all:

Listing 8.18: A better way to test for parity without recursion.

```
let even x = (x % 2 = 0)
let odd x = not (even x)
```

which is to be preferred anytime as the solution to the problem.

8.4 Programming intermezzo

Using loops and conditional expressions we are now able to solve the following problem

Problem 8.3:

Given an integer on decimal form, write its equivalent value on binary form

To solve this problem, consider odd numbers: They all have the property, that the least significant bit is 1, e.g., $1_2 = 1, 101_2 = 5$ in contrast to even numbers such as $110_2 = 6$. Division by 2 is equal to right-shifting by 1, e.g., $1_2/2 = 0.1_2 = 0.5, 101_2/2 = 10.1_2 = 2.5, 110_2/2 = 11_2 = 3$. Thus by integer division by 2 and checking the remainder, we may sequentially read off the least significant bit. This leads to the following algorithm,

Listing 8.19, dec2bin.fsx:
Using integer division and remainder to write any positive integer on binary form.

```
let dec2bin n =
  let rec dec2binHelper n =
    let mutable v = n
    let mutable str = ""
    while v > 0 do
      str <- (string (v % 2)) + str
      v <- v / 2
    str

  if n < 0 then
    "Illegal value"
  elif n = 0 then
    "0b0"
  else
    "0b" + (dec2binHelper n)

printfn "%4d -> %s" -1 (dec2bin -1)
printfn "%4d -> %s" 0 (dec2bin 0)
for i = 0 to 3 do
  printfn "%4d -> %s" (pown 10 i) (dec2bin (pown 10 i))
```

```
  -1 -> Illegal value
   0 -> 0b0
   1 -> 0b1
  10 -> 0b1010
 100 -> 0b1100100
1000 -> 0b1111101000
```

Another solution is to use recursion instead of the `while` loop:

Listing 8.20, dec2binRec.fsx:
Using recursion to write any positive integer on binary form, see also Listing 8.19.

```
let dec2bin n =
  let rec dec2binHelper n =
    if n = 0 then ""
    else (dec2binHelper (n / 2)) + string (n % 2)

  if n < 0 then
    "Illegal value"
  else
    "0b" +
    if n = 0 then
      "0"
    else
      dec2binHelper n

printfn "%4d -> %s" -1 (dec2bin -1)
printfn "%4d -> %s" 0 (dec2bin 0)
for i = 0 to 3 do
  printfn "%4d -> %s" (pown 10 i) (dec2bin (pown 10 i))
```

```
  -1 -> Illegal value
   0 -> 0b0
   1 -> 0b1
  10 -> 0b1010
 100 -> 0b1100100
1000 -> 0b1111101000
```

Listing 8.19 is a typical imperative solution, where the states `v` and `str` are iteratively updated until `str` finally contains the desired solution. Listing 8.20 is a typical functional programming solution, to be discussed in Part III, where the states are handled implicitly as new scopes created by recursively calling the helper function. Both solutions have been created using a local helper function, since both solutions require special treatment of the cases $n < 0$ and $n = 0$.

Let us compare the two solutions more closely: The computation performed is the same in both solutions, i.e., integer division and remainder is used repeatedly, but since the recursive solution is slightly shorter, then one could argue that it is better, since shorter programs typically have fewer errors. However, shorter program also typically means that understanding them is more complicated, since shorter programs often rely on realisations that the author had while programming, which may not be properly communicated by the code nor comments. Speedwise, there is little difference between the two methods: 10,000 conversions of `System.Int32.MaxValue`, i.e., the number 2,147,483,647, takes about 1.1 sec for both methods on an 2,9 GHz Intel Core i5 machine.

Notice also, that in Listing 8.20, the prefix `"0b"` is only written once. This is advantageous for later debugging and updating, e.g., if we later decide to alter the program to return a string without the prefix or with a different prefix, then we would only need to change one line instead of two. However, the program has gotten slightly more difficult to read, since the string concatenation operator and the `if` statement are now intertwined. There is thus no clear argument for preferring one over the other by this argument.

Proving that Listing 8.20 computes the correct sequence is easily done using the induction proof technique: The result of `dec2binHelper 0` is clearly an empty string. For calls to `dec2binHelper n` with $n > 0$, we check that the right-most bit is correctly converted by the remainder function, and that this string is correctly concatenated with `dec2binHelper` applied to the remaining bits. A simpler way to state this is to assume that `dec2binHelper` has correctly programed, so that in the body of `dec2binHelper`, then recursive calls to `dec2binHelper` returns the correct value. Then we only need to check that the remaining computations are correct. Proving that Listing 8.19 calculates the correct sequence essentially involves the same steps: If $v = 0$ then the `while` loop is skipped, and the result is the initial value of `str`. For each iteration of the `while` loop, assuming that `str` contains the correct conversions of the bits up till now, we check that the remainder operator correctly concatenates the next bit, and that `v` is correctly updated with the remaining bits. We finally check that the loop terminates, when no more 1-bits are left in `v`. Comparing the two proofs, the technique of assuming that the problem has been solved, i.e., that recursive calls will work, helps us focus on the key issues for the proof. Hence, we conclude that the recursive solution is most elegantly proved, and thus preferred.

Bibliography

[1] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[2] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.

[3] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[4] X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. `http://worldpowersystems.com/projects/codes/X3.4-1963/`.

[5] George Pólya. *How to solve it.* Princeton University Press, 1945.

Index