VIA University College

Programming Concepts and Languages

Spring 2024

Learning Objectives

- By the end of this session, you will be able to:
  - explain the structured types in F#, which are extensively used in functional programming
  - identify some basic types of values (Integers, char, String, bool)
  - write programs using pattern matching
  - develop programs using the different core types
  - develop small programs to create a list of items and access elements of a list
  - develop small F# functions using list functions and list comprehensions
  - develop small F# error handling functions using the “failwith”

F# Program Structure

- F# program consists of a number of declarations
  - let declarations – most commonly used
  - declarations of types, exceptions – coming later
- let declarations define names for values
  - `let numberPresent = 23`
- After the = we can define any expression
  - Evaluated to a value and used as the definition for the name
    - `let averageAttRate = 23/20`
- Values can also include functions
  - `let square = fun x -> x * x`
- Difference with other languages
  - functions and non-functions can be defined in the same way

Numerical Types

- F# has a number of numerical types:

Table 2-1. Numerical primitives in F#

Type	Suffix	.NET Type	Range
`byte`	`uy`	`System.Byte`	0 to 255
`sbyte`	`y`	`System.SByte`	−128 to 127
`int16`	`s`	`System.Int16`	−32,768 to 32,767
`uint16`	`us`	`System.UInt16`	0 to 65,535
`int, int32`		`System.Int32`	-2^{31} to $2^{31}-1$
`uint32`	`u`	`System.UInt32`	0 to $2^{32}-1$
`int64`	`L`	`System.Int64`	-2^{63} to $2^{63}-1$
`uint64`	`UL`	`System.UInt64`	0 to $2^{64}-1$
`float`		`System.Double`	A double-precision floating point based on the IEEE 64 standard. Represents
`float32`	`f`	`System.Single`	A single-precision floating point based on the IEEE 32 standard. Represents values with approximately 7 significant digits.
`decimal`	`M`	`System.Decimal`	A fixed-precision floating-point type with precisely 28 digits of precision.

Functions and Operators on Numerical Types

Operator	Description	Example	Result
+	Addition	`1 + 2`	`3`
–	Subtraction	`1 – 2`	`–1`
*	Multiplication	`2 * 3`	`6`
/	Division	`8L / 3L`	`2L`
**	Power[a]	`2.0 ** 8.0`	`256.0`
%	Modulus	`7 % 3`	`1`

[a] Power, the ** operator, only works for `float` and `float32` types. To raise the power of an integer value, you must either convert it to a floating-point number first or use the `pown` function.

- Bitwise operators: all integer types
- Type conversion functions: same name as type converted to

  `int 18.2 ⇒ 18`

  `int 18L ⇒ 18`

Characters

- Type char for characters
- Syntax:

```
let dkVowels = ['a'; 'e'; 'i'; 'o'; 'u'; 'æ'; 'ø'; 'å'];;
```

- Characters are elements in strings

Table 2-6. Character escape sequences

Character	Meaning
\'	Single quote
\"	Double quote
\\	Backslash
\b	Backspace
\n	Newline
\r	Carriage return
\t	Horizontal tab

Strings

- Strings are defined by enclosing a series of characters in double quotes.
  - can span multiple lines
- uses indexer to access a character like arrays
  - But are immutable and can't be modified

```
> let classId = "IT-PCL1-S24";;
> classId.Length;;
> classId.[7] ;;
```

Booleans

- Type `bool` for the two booleans values `true, false`
- Boolean operators and functions:

Operator	Description	Example	Result
`&&`	And	`true && false`	`false`
`||`	Or	`true || false`	`true`
`not`	Not	`not false`	`true`

- Relational operator returning a boolean value:

Operator	Description	Example	Result
`<`	Less than	`1 < 2`	`True`
`<=`	Less than or equal to	`4.0 <= 4.0`	`True`
`>`	Greater than	`1.4e3 > 1.0e2`	`True`
`>=`	Greater than or equal to	`0I >= 2I`	`False`
`=`	Equal to	`"abc" = "abc"`	`True`
`<>`	Not equal to	`'a' <> 'b'`	`True`

- Can compare elements from any “comparable” type
  - returns −1, 0, or 1 depending on whether the first parameter is less than, equal to, or greater than the second.

Conditional

- F# has a conditional if-then-else expression:

  `if true then x else y` **gives** `x`

  `if false then x else y` **gives** `y`

- we can write expressions like

  `if x > 0 then x else -x`

- However, the two branches must have the same type
- This means, `if x > 0 then 17 else 'a'` is illegal

Functions I

- functions take some arguments and return a result
- you can define your own functions or use predefined functions
- A little unusual syntax: no parentheses around arguments

```
add10 20
addXY 10 20
```

- The space between function and argument can be seen as a special operator:
  - function application
- Function application binds harder than any other operator
- i.e.:
  - `f x + y` means `(f x) + y`, `not f (x + y)`
- Forgetting this is one of the common mistake by beginners

Functions II

- Having functions as a kind of values is powerful
  - this is one of the defining features of functional programming.
- function definitions are can be abbreviated:

```
let square x = x * x          // abbreviation
let square  = fun x -> x * x // without the abbreviation
```

- To call (or “apply”) a function, just put it in front of an expression:

```
let pclSqrA = square 5
let pclSqrB = square (square (2+5))
let pclSqrC = (fun x -> x * x) 5
```

pclSqrC = pclSqrA = 25

- N/B: parentheses are not needed in function definitions/values nor in applications (another difference with many languages).
  - Instead they are used only when needed to group things together.

Type Inference

- Every variable and expression in F# must have a type assigned to it.
  - The compiler infers these types automatically and reports inconsistent types as errors.
  - Inference means that you usually do not need to declare the types for variables.
  - However, one could add types to help the inference, debugging, etc.:
  - `let add (x : float) y = x + y`
- Functions are given types like: float -> float -> float
  - N/B: functions are just another kind of value.

Pattern Matching I

- F# has a case construct

```
match expr with
   | pattern1 -> expr1
   | pattern2 -> expr2
   ....
```

- Every case has a pattern for the argument
- Cases are checked in order, the first that matches the argument is selected
- types like numerical types, have constants and variables as possible patterns

Pattern Matching II

- Pattern matching can also be used with booleans, numbers, and other types.
- Note: `match ... with` is just another expression, it is evaluated and returns a result
- useful with structured data, where it can be used to conveniently pick out parts of data structures.

Exercise

- Convert the previous factorial function to use pattern matching

```
let rec factorial n =
    if n < 1 then 1
    else n * factorial (n - 1)
```

Core Types

Table 2-9. Core types in F#

Signature	Name	Description	Example
`unit`	Unit	The unit value	`()`
`int, float`	Concrete type	A concrete type	`42, 3.14`
`'a, 'b`	Generic type	A generic (free) type	
`'a -> 'b`	Function type	A function returning a value	`fun x -> x + 1`
`'a * 'b`	Tuple type	An ordered grouping of values	`("eggs", "ham")`
`'a list`	List type	A list of values	`[ 1; 2; 3], [1 .. 3]`
`'a option`	Option type	An optional value	`Some(3), None`

Tuple : System.Tuple<_> type

- Tuples are similar to records, or objects
- A tuple is like a container for data with a fixed number of slots
- Example:

```
('a', 23, 3.142)
```

- This is a three-tuple whose first component is a character, the second an integer, and the third a floating-point number
- It has the tuple type `char * int * float`
- Tuples can contain any type of data, for instance:

```
(add, (23, 'd')) :  (int -> int -> int) * (int * char)
```

- Thus, there are really infinitely many tuple types

Tuple : Construction & Deconstruction/pattern

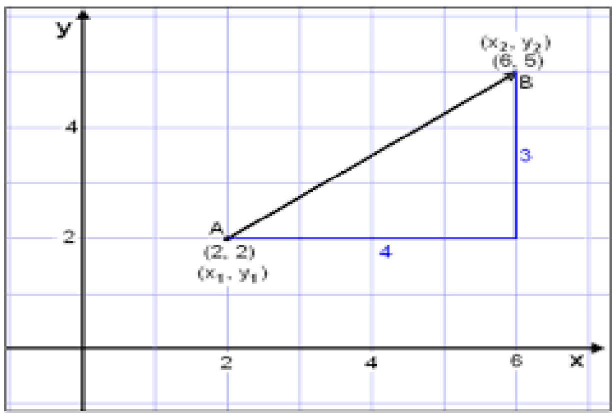
```
// Construction
let student1 = (111401, "Anders")

// Triple tuple
let student2 = (111402, "Catalin", "IoT")

// Deconstruction - using fst, snd or pattern
let studentId = fst student1
let studentName = snd student1

let (x', y') = student1
```

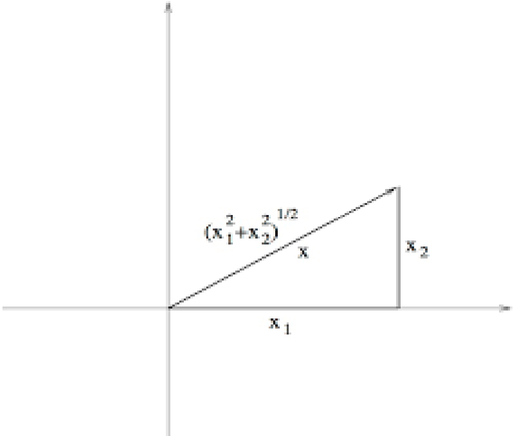
Use of Tuples

- To simply add 2 numbers

```
tupledAdd(x, y) = x + y
```

- Use tuples with two floats to represent 2D-vectors





- Define functions vadd, vsub, vlen to add, subtract, and compute the length of vectors:

```
vecAdd : (float * float) -> (float * float) -> (float * float)
vecSub : (float * float) -> (float * float) -> (float * float)
vecLen : (float * float) -> float
```

Lists

- Lists are:
  - very important data structures in functional programming
  - sequences of elements
- Lists can be arbitrarily long, but all elements must be of the same type
- If `a` is a type, then a list is the type “list of `a`”
- Example: `int list, char list, (int list) list,`

  `(float -> int) list`
- A list can contain elements of any type (as long as all have the same type)

Constructing Lists I

- Lists are constructed from:
  - the `empty list : []`
  - the "cons" operator, which puts an element in front of a list: `::`
- Example: `1::(2::(3::[]))`
- `::` and `[]` are called constructors: they "construct" data structures
- N/B: This is different with constructors in object-oriented languages

Constructing Lists II

- `1::2::3::[]` is same as `1::(2::(3::[]))`
  - i.e.: "`::`" is right-associative)
- `[1;2;3]` is another shorthand for `1::(2::(3::[]))`
- The first element of a nonempty list is the head:
- `List.head [1;2;3]` ⇒ `1`
- `List.head` is a function from the List module, thus the prefix `List.`
- The list of the remaining elements is the tail
- `List.tail [1;2;3]` ⇒ `[2;3]`

List Ranges

- Can declare a list of ordered numeric values
- The first expression specifies the lower bound of the range
- The second specifies the upper bound

```
x = [1 .. 10]:   int list
```

- With optional step value, the result becomes a list of values in the range between two numbers separated by the stepping value

```
tens = [0 .. 10 .. 50] : int list
```

Some List Functions

- F# has many builtin functions on lists:
- `List.length ls`
  - computes the length of the list `ls`
  - Example: `List.length ['a';'b';'c']` ⇒ `3`
- `List.sum ls`,
  - sums all the numbers in `ls`
  - Example: `List.sum [1;2;3]` ⇒ `6`
- Let's program them as an exercise

Exercises – List functions

Exercises 2.1.1 – 2.2.4

N/B

- Define an F# function to duplicate the element of a list
  - Example:

```
pclDupliLstComp ["F#";"Python"] gives ["F#"; "F#"; "Python"; "Python"]
```

- Define an F# function to replicate the elements of a list given number of times
  - Example:

```
pclDupliLstComp2 ['A'; 'C'; 'T'; 'G'] 3;; gives
['A'; 'A'; 'A'; 'C'; 'C'; 'C'; 'T'; 'T'; 'T'; 'G'; 'G'; 'G']
```

List Functions

- F# has many built-in functions on lists:
- `List.length ls`
  - computes the length of the list `ls`
  - Example: `List.length ['a';'b';'c']` ⇒ `3`
- `List.sum ls`,
  - sums all the numbers in `ls`
  - Example: `List.sum [1;2;3]` ⇒ `6`

pclSum – Closer look

- `pclSum [1;2;3] = sum 1::(2::(3::[]))`

  `⇒ 1+(2+(3+0)) ⇒ 6`

- `list 1::(2::(3::[]))` and sum expression for `1+(2+(3+0))`

  are similar in their tree structure

- `pclSum` basically replaces `::` with `+` and then calculates the result

List Comprehensions: [ ] (aka generators)

- an elegant way of defining lists using other lists
- made up of elements returned via yield

```
let numbersNear x  =
    [
        yield x - 1
        yield x
        yield x + 1
   ]
```

`numbersNear 3 ?  ⇒ [2; 3; 4]`

- List comprehension can contain F# code including function declarations
- First appearance of `for loops`

List Comprehensions II

- List comprehensions are an elegant way of defining lists using other lists.
- for is used to for generators, which enumerate the elements of another list
- if allows filtering based on a Boolean
- yield adds an element to the list being created.

```
let xl =
    [   let negate x = -x
        for i in 1 .. 10 do
            if i % 2 = 0 then
                yield negate i
            else
                yield i ]
```

xl ⇒ [1; −2; 3; −4; 5; −6; 7; −8; 9; −10]

List Comprehensions III

- `->` can be used with `for ... in` as an abbreviation of `do yield`

```
// Generate the first ten multiples of a number
multiplesOf x = [ for i in 1 .. 10 do yield x * i ]


// Simplified list comprehension
multiplesOf2 x = [ for i in 1 .. 10 -> x * i ]
```

List Comprehensions IV

- **for** allows use of patterns:

```
pclSqrs n = [ for i in 1 .. n -> (i, i*i) ]

pclSqrsAdd n = [ for (i,psq) in pclSqrs n -> i + psq ]

pclSqrsAdd 4 ⇒ [2; 6; 12; 20]
```

- **yield!** (yield Bang) puts a whole list of values into the output list.

```
let yb =
    [for a in 1 .. 5 do
     match a with
      | 3 -> yield! ["P"; "C"; "L"]
      | _ -> yield a.ToString()]
```

```
yb ⇒ ["1"; "2"; "P"; "C"; "L"; "4"; "5"]
```

yield vs yield!

- yield allows use of patterns:

```
listWithYield =
    [ for i in 0 .. 10 .. 20 do
            yield [ i .. 1 .. i+9 ]]
```

- yield! (yield bang) puts a whole list (including sub lists) of values into the output list.

```
listWithYieldBang =
    [ for i in 0 .. 10 .. 20 do
            yield! [ i .. 1 .. i+9 ] ]
```

List module functions

Function and type	Description
`List.exists` `('a -> bool) -> 'a list -> bool`	Returns whether or not an element in the list satisfies the search function.
`List.rev` `'a list -> 'a list`	Reverses the elements in a list.
`List.tryfind` `('a -> bool) -> 'a list -> 'a option`	Returns `Some(x)` where `x` is the first element for which the given function returns `true`. Otherwise returns `None`. (`Some` and `None` are covered shortly.)
`List.zip` `'a list -> 'b list -> ('a * 'b) list`	Given two lists with the same length, returns a joined list of tuples.
`List.filter` `('a -> bool) -> 'a list -> 'a list`	Returns a list with only the elements for which the given function returned `true`.
`List.partition` `('a -> bool) -> 'a list -> ('a list * 'a list)`	Given a predicate function and a list, returns two new lists; the first where the function returned `true`, the second where the function returned `false`.

Errors and Non-termination

- Now what about `factorial (-1)?`

```
factorial(-1)⇒(-1)*factorial(-2)⇒(-1)*(-2)*factorial(-3)⇒ . . .
```

- Infinite recursion! Will never terminate.
- For computations that never return anything, we use the notation "⊥" for the "resulting value"
- Thus, factorial(-1) = ⊥
- Remember factorial is really just defined for natural numbers, not for negative numbers
- It's good practice to have controlled error handling of out-of-range arguments

F# failwith Function

- F# has an `failwith` function, which when executed prints a string and stops the execution
- Example:

```
failwith "You cannot input negative argument to
this function"
```

- (Strings in F# are written within quotes, like “Please put me in quotes")
- Add error handling to the factorial function.