Problem Solving, Programming, and Calculation

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

`bjorn.lisper@mdh.se`
`http://www.idt.mdh.se/~blr/`

Overview

Basic concepts of functional programming: computation by calculation, values, expressions, types, declarations

First glance of F#: introduction, basic language elements

Some simple functional programming examples/exercises

Computation by Calculation

In “ordinary” programming languages, computation alters state by writing new values to program variables:

State before: $\mathtt{x} = 3, \mathtt{y} = 4$

```
x = 17 + x + y
```

State after: $\mathtt{x} = 24, \mathtt{y} = 4$

This means you perform computations in *sequence*

There is a *control flow* that decides this sequence:

```
for(i=0,i++,i<17)
  { if (j > i) then x++ else y++;
    x = y + j;
  }
```

In (pure) functional languages, computation is simply *calculation*:

$$
\begin{aligned}
\texttt{3*(9 + 5)} &\Longrightarrow \texttt{3*14} \\
&\Longrightarrow \texttt{42}
\end{aligned}
$$

Forget about assignments, sequencing, loops, if-then-else, . . .

Forget everything you know about programming! (Well, almost)

You might wonder how to do useful stuff just by calculating, we'll see later . . .

In functional languages we have:

- *functions*
- *recursion* (instead of loops)
- *expressive data types* (much more than numbers)
- *advanced data structures*

and calculation on top of that

Functions

Functions, mathematically: sets of pairs where no two pairs can have the same first component:

$f = \{(1,17),(2,32),(3,4711)\}$

$f(2) = 32$

Or, given the same argument the function always returns the same value

(cannot have $f(2) = 32$ and $f(2) = 33$ at the same time)

Functions model *determinism*: that outputs depend predictably on inputs

Something we want to hold for computers as well . . .

Defining a function by writing all pairs can be very tedious

Often defined by simple *rules* instead

$simple(x, y, z) = x \cdot (y + z)$

These rules express *abstraction*: that a similar pattern holds for many different inputs

(“For all x, y, z, $simple(x, y, z)$ equals $x \cdot (y + z)$”)

Abstraction makes definitions shorter and easier to grasp

A good property also for software, right?

Recursion

Mathematical functions are often specified by *recursive* rules

Recursion means that a defined entity refers to itself in the definition

This seems circular, but can make sense

Example: the factorial function “!” on natural numbers

$0! = 1$

$n! = n \cdot (n-1)!, \quad n > 0$

Recursion corresponds to loops in ordinary programming

Pure Functional Languages

Pure functional languages implement mathematical functions

A functional language is pure if there are no *side-effects*

A side effect means that a function call does something more than just return a value

An example in C:

```
int f(int x)
{
  n++;
  return(n+x);
}
```

Side effect for `f`: global variable `n` is incremented for each call

This means that `f` returns different values for different calls, even when called with the same argument

Much harder to reason mathematically about such functions: for instance,

`f(17) + f(17)` $\neq$ `2*f(17)`

Side effects require a more complex model, and thus makes it harder to understand the software

In pure functional languages, functions are specified by side-effect free rules (declarations)

In F#:

```
let simple x y z = x*(y + z)
```

Each rule defines a calculation for any actual arguments:

```
simple 3 9 5  ==>  3*(9 + 5)
              ==>  3*14
              ==>  42
```

Just put actual arguments into the right-hand side and go!

Compare this with an execution model that must account for side-effects

Exercise

Calculate `simple (simple 2 3 4) 5 6`

Note that we can do the calculation in different order

Do we get the same result?

More on this later . . .

The mathematical view makes it possible to *prove* properties of programs

When calculating, all intermediate results are *mathematically equal*:

$$\texttt{simple 3 9 5} = \texttt{3*(9 + 5)} = \texttt{3*14} = \texttt{42}$$

For instance, prove that `simple x y z` = `simple x z y` for all `x`, `y`, `z`:

$$\begin{aligned}\texttt{simple x y z} &= \texttt{x*(y + z)}\\ &= \texttt{x*(z + y)}\\ &= \texttt{simple x z y}\end{aligned}$$

We cannot do this for functions with side-effects

Expressions, Values, and Types

Calculation is performed on *expressions*:

```
simple (simple 2 3 4) 5 6
```

Expressions are calculated into *values*:

simple (simple 2 3 4) 5 6 $\Longrightarrow$ 154

Values are also expressions, which cannot be calculated any further

Types

Many programming languages have *types*

Types represent sets of values (like integers)

Types help avoiding certain programming errors, like adding an integer with a character

A programming language can be:

- *strongly typed*: every program part must have a legal type
- *weakly typed*: every program part can have a legal type, but need not
- *untyped*: no types exist

A First Introduction of F#

F# is a strongly typed, functional language

It uses *eager evaluation* (compute function arguments fully before making the call), but also supports *lazy evaluation* and *computation by demand* (don’t compute anything before it’s needed)

It is *higher order* (functions are ordinary data)

It is *not pure* (side effects possible), but encourages pure programming

It supports *imperative* and *object-oriented* programming

It has a *polymorphic* type system, with *type inference*, *subtyping*, and *overloading* of common operators

It has *syntactic facilities* to help write clear and understandable programs

F# implementation

F# comes from Microsoft Research

Current version is 4.5

Included in VS 2017, can also be freely downloaded

Can also be run on Mac/linux under Mono

Three ways to run F# compiler:

- In Visual Studio
- `fsc` – batch compiler
- `fsi` – interactive command line compiler

Basic Language Elements of F#

A first introduction:

- Values
- Types (atomic values, composed values)
- Operators and predefined functions
- Other syntactic forms

Numerical Types

F# has a number of numerical types, some important examples:

`int`	32-bit integers (`1`, `-3`, `54873`, ...)
`float`	double precision (64-bit) floats (`1.0`, `3.14159`, `3.2e3`, ...)
`int64`	64-bit integers (`1L`, `-3L`, `54873L`, ...)
`single`	single precision (32-bit) floats (`1.0f`, `3.14159f`, `3.2e3f`, ...)

Functions and Operators on Numerical Types

The usual ones:

`+`, `-`, `*`, `/`, `%` (modulus): all numeric types

Operands must have same type: `2 + 3` OK, `2 + 3.0` not OK

`**` (power): all floating-point types

Bitwise operators: all integer types

Most “typical” math functions (trigonometric, exponent, etc)

Type conversion functions: same name as type converted to:

`int 17.3` $\Longrightarrow$ `17`

`int 17L` $\Longrightarrow$ `17`

Numerical Expressions

Numerical expressions look like in most languages:

```
x + 7*y
```

```
3.14159/(x + 1.0) - 33.0
```

(`x + 7*y` is the same as `x + (7*y)`, since `*` *binds* stronger than `+`)

Characters

Type `char` for characters

Syntax: `'a'`, `'b'`, `'\n'` (newline), . . .

Characters are elements in *strings*

More on strings later

Booleans

Type `bool` for the two booleans values `true`, `false`

Boolean operators and functions: `&&` (and), `||` (or), `not`

Relational operator returning a boolean value: `=`, `<>`, `<`, `<=`, `>=`, `>`

Can compare elements from any “comparable” type (more on this later)

Conditional

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

`if true then x else y` $\Longrightarrow$ `x`

`if false then x else y` $\Longrightarrow$ `y`

So we can write expressions like

```
if x > 0 then x else -x
```

However, the two branches must have the same type

Thus, `if x > 0 then 17 else 'a'` is illegal

Functions

Functions take a number of arguments and return a result

Some predefined ones, and you can define your own

A little unusual syntax: no parentheses around arguments

```
sqrt 17.0
```

```
exp 17.0 3.0
```

(There are good reasons for this syntax, more on this later)

The space between function and argument can be seen as a special operator: *function application*

Function application binds harder than any other operator

Thus, `f x + y` means `(f x) + y`, not `f (x + y)`

(Common beginner's mistake to forget this)

Declarations

You can define your own entities

Entities of any type can be defined by a *let declaration*

`let pi = 3.14159` defines `pi` to be a floating-point constant

`let simple x y z = x*(y+z)` defines `simple` to be a function in three arguments

Types of Defined Entities

Note that we did not give any types in the definitions

F# manages to find types automatically!

This is called *type inference*

A very simple example:

```
let pi = 3.14159
```

Here, `pi` will automatically obtain the type `float` since `3.14159` can only have type float

Explicit Type Declarations

However, we can also give explicit types (sometimes useful)

`e : t` declares `e` to have type `t`

For instance:

`let f x = (x + x) : float` forces the right-hand side of the declaration to have type `float`.

(If we just write `let f x = (x + x)`, then F# will assume that “+” is plus on the type `int`.)

What about `let f x = (x + x) : char`?

Recursive Definitions

As a first exercise, recall the factorial function:

$0! = 1$

$n! = n \cdot (n-1)!, \quad n > 0$

Define it in F#!

(Answer on next slide)

Factorial in F#

```
let rec fac n = if n = 0 then 1 else n*fac (n-1)
```

Note `let rec`: *recursive* definition, `fac` in right-hand side must be same as in left-hand side. More in this later

Exercise:

Calculate `fac 3`

$$
\begin{array}{ll}
\texttt{fac 3} & \Longrightarrow \texttt{if 3 = 0 then 1 else 3}\star\texttt{fac (3-1)} \\
 & \Longrightarrow \texttt{if false then 1 else 3}\star\texttt{fac (3-1)} \\
 & \Longrightarrow \texttt{3}\star\texttt{fac (3-1)} \\
 & \Longrightarrow \texttt{3}\star\texttt{fac 2} \\
 & \Longrightarrow \texttt{3}\star\texttt{(if 2 = 0 then 1 else 2}\star\texttt{fac (2-1))} \\
 & \Longrightarrow \texttt{3}\star\texttt{(if false then 1 else 2}\star\texttt{fac (2-1))} \\
 & \Longrightarrow \texttt{3}\star\texttt{2}\star\texttt{fac (2-1)} \\
 & \text{... etc ...} \\
 & \Longrightarrow \texttt{3}\star\texttt{2}\star\texttt{1}\star\texttt{fac (1-1)} \\
 & \Longrightarrow \texttt{3}\star\texttt{2}\star\texttt{1}\star\texttt{fac 0} \\
 & \Longrightarrow \texttt{3}\star\texttt{2}\star\texttt{1}\star\texttt{1} \\
 & \Longrightarrow \texttt{6}
\end{array}
$$

Eventually we'll reach the "base case" `fac 0`. This is what makes recursion work here!

Pattern Matching

Another way to write the factorial function: use *patterns*

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

For “atomic” types like numeric types, the possible patterns are constants (matching exactly that constant), or variables (matching any value)

So we can define `fac` like this instead:

```
let rec fac n = match n with
                | 0 -> 1
                | _ -> n*fac (n-1)
```

If `n` is `0` return `1`, otherwise return `n*fac (n-1)`

“_” is a “wildcard”, matches anything

More on Pattern Matching

Note that `match ... with` is just another expression, it is evaluated and returns a result

Pattern matching is really useful with structured data, where it can be used to conventiently pick out parts of data structures. More on this later

Nontermination and Errors

Now what about `fac (-1)`?

$\texttt{fac(-1)} \Longrightarrow \texttt{(-1)*fac(-2)} \Longrightarrow \texttt{(-1)*(-2)*fac(-3)} \Longrightarrow \cdots$

Infinite recursion! Will never terminate.

For computations that never return anything, we use the notation "$\perp$" for the "resulting value"

Thus, $\texttt{fac(-1)} = \perp$

Remember `fac` 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

The F# failwith Function

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

E.g.,

```
failwith "You cannot input this number to this function"
```

(Strings in F# are written within quotes, like `"Hello world"`)

A version of `fac` with error handling:

```
let rec fac n =
  match n with
  | 0 -> 1
  | _ -> if n > 0
         then n*fac (n-1)
         else failwith "Negative argument to fac"
```

Would we be able to write this case-by case (`n > 0`, `n = 0`, `n < 0`)?

Yes, pattern-matching can be extended with *guards*. More on this later

Factorial “Old Style”

How would you implement the factorial function in an imperative language, with loops and program variables that can be assigned new values?

Factorial “Old Style” (II)

My suggestion would be something like this (in C):

```
int fac (int n) {
  int acc = 1;
  while (n \= 0)
  {
    acc = n*acc;
    n = n-1;
  }
  return(acc);
}
```

Notice how `acc` is used to successively accumulate the product. Could we do something similar in a pure functional language?

Factorial with Accumulating Arguments

The answer is yes:

```
let rec fac1 acc n = match n with
                     | 0 -> acc
                     | _ -> fac1 (n*acc) (n-1)
let fac n = fac1 1 n
```

This solution uses a help function `fac1` with two arguments

The second argument is an *accumulating argument*, where we successively collect the result. It plays the same role as `acc` in the C loop

Exercise: calculate `fac 3` with this new definition

McCarthy's Transformation

The derivation of `fac` with accumulating arguments from the C loop is an example of *McCarthy's Transformation*.

This transformation is a systematic method to transform imperative programs into pure functional programs

Theorem: *any imperative program that can be expressed as a flowchart can be transformed into a pure functional program computing the same thing*

So imperative programming is just a special case of functional programming, and loops are just a special case of recursion

Local Definitions

The `fac` version with accumulating argument uses a help function `fac1`

This function is globally defined

However, only used by `fac`

We may want to *hide* it in the definition of `fac`

F# has a `let (rec) ... in ...`-construct for *local* definitions:

```
fac n = let rec fac1 acc n = match n with
                             | 0 -> acc
                             | _ -> fac1 (n*acc) (n-1)
        in fac1 1 n
```

Defines `fac1` locally in the expression after “`in`”. *Not* visible outside!

`let (rec) ... in`-expressions are ordinary expressions and return values, can be used wherever ordinary expressions can be used:

```
17 + let x = fac 3 in x + 3*x ⟹
17 + let x = 6 in x + 3*x ⟹ 17 + (6 + 3*6) ⟹ 41
```

Also note that the defined entity (`x` here) only needs to be computed once – saves work!

So `let` can be used also to save work when the same result is to be used many times

Function Types

A function that takes an argument of type `a` and returns a value of type `b` has the *function type* `a -> b`

For instance, `fac : int -> int`

(Note resemblance with mathematical notation)

What about functions with several arguments?

```
let simple x y z = x*(y+z)
simple : int -> int -> int -> int
```

Last type is result type, preceding types are the respective argument types

(We’ll explain later why multi-argument function types look this way)

Types for Data Structures

F# has two important predefined types for data structures: *tuples* and *lists*

(Also some more types for this, but we'll leave them for now)

Lists are very important data structures in functional programming

They are sequences of elements

Lists can be arbitrarily long, but (in F#) *all elements must be of the same type*

If `a` is a type, then `a list` is the type "list of `a`"

E.g. `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

Lists are built 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

(Constants like `17` and `'x'` are also constructors, but they only construct themselves)

(Don't mix up with constructors in object-oriented langages)

`1::2::3::[]` is same as `1::(2::(3::[]))` (“`::`” 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]` $\Longrightarrow$ `1`

(`List.head` is a function from the *List module*, thus the prefix `List.` More on modules later)

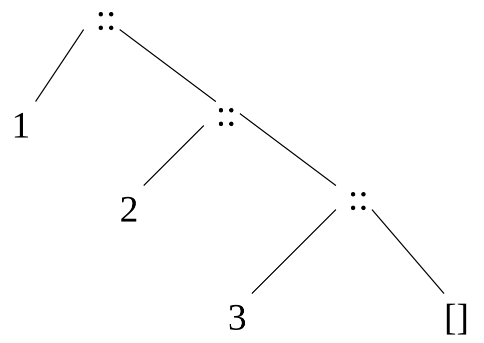
The list of the remaining elements is the *tail*

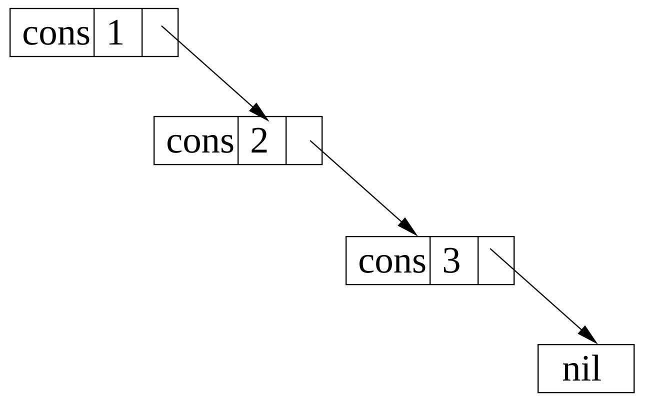
`List.tail [1;2;3]` $\Longrightarrow$ `[2;3]`

Also a function `List.isEmpty` to check if a list is empty or not:

`List.isEmpty []` $\Longrightarrow$ `true`

To the left is a graphical picture of `[1;2;3]` as an expression tree:





Underneath, there is a linked data structure (shown to the right)

In conventional languages you'd have to manage the links yourselves.
Functional programming runtime systems handle them automatically

Some List Functions

F# has many builtin functions on lists in the `List` module. Let’s look at some and try to program them, as an exercise

`List.length xs`, computes the length of the list `xs`

`List.length ['a';'b';'c']` $\Longrightarrow$ 3

`List.sum xs`, sums all the numbers in `xs`

`List.sum [1;2;3]` $\Longrightarrow$ `6`

(Solutions on next slide)

```
let rec length l =
  if List.isEmpty l
    then 0
    else 1 + length (List.tail l)

let rec sum  l =
  if List.isEmpty l
    then 0
    else List.head l + sum (List.tail l)
```

With Pattern Matching

```
let rec length l = match l with
                   | []      -> 0
                   | (x::xs) -> 1 + length xs

let rec sum  l = match l with
                 | []      -> 0
                 | (x::xs) -> x + sum xs
```

The pattern `(x::xs)` matches *any list that is constructed with a cons* (that is, any non-empty list)

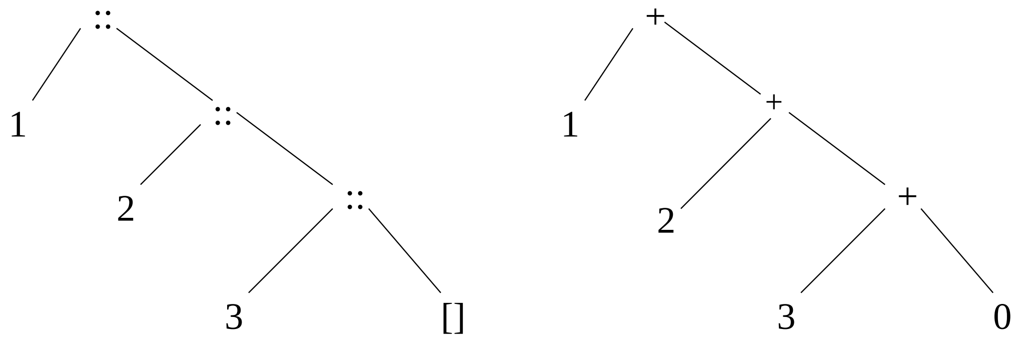
`x` gets bound to the head, and `xs` to the tail

An Observation on `sum`

We have:

`sum [1,2,3]` $=$ `sum 1::(2::(3::[]))` $\Longrightarrow$ `1+(2+(3+0))` $\Longrightarrow$ `6`

Note the similarity between tree for list `1::(2::(3::[]))` and sum expression for `1+(2+(3+0))`:



`sum` basically replaces `::` with $+$ and then calculates the result

This is a common pattern! We'll get back to this when we treat higher-order functions

Also: in a sense, data structures (like lists) in F# are just expressions where the "operators" ($=$ constructors) cannot be further evaluated: thus, the constructors are left in the result (where they build the data structure)

Tuples

Tuples are similar to records, or objects

A tuple is like a container for data with a fixed number of slots

An example: `('a',17,3.14159)`

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`

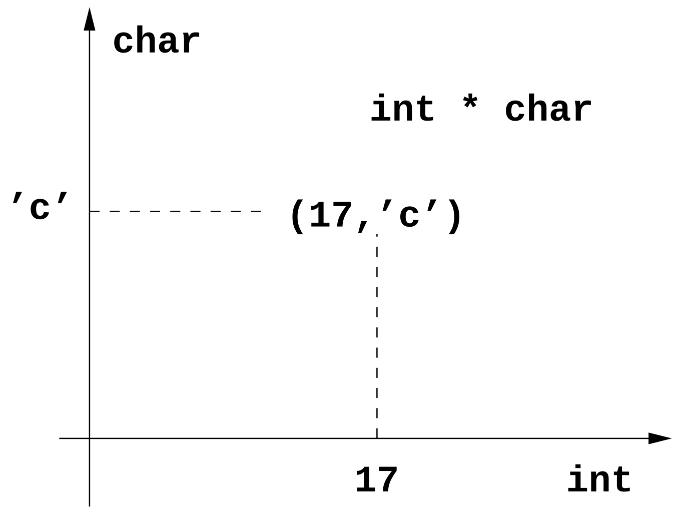
Tuples can contain *any* type of data, for instance:

```
(fac,(17,'x')) : (int -> int) * (int * char)
```

Thus, there are really infinitely many tuple types

Tuples (II)

Tuple types are similar to cartesian products on sets:



Similar to vector spaces. An n-tuple is like a vector with n elements, although the elements do not have to be numbers

An Example of the Use of Tuples

Use tuples with two floats to represent 2D-vectors

Define functions `vAdd`, `vSub`, `vLen` to add, subtract, and compute the length of vectors:

```
vAdd,vSub : (float * float) -> (float * float) -> (float * float)
vLen : (float * float) -> float
```

(Solutions on next slide)

```
let vAdd (x1,y1) (x2,y2) = (x1+x2,y1+y2) : float * float
let vSub (x1,y1) (x2,y2) = (x1-x2,y1-y2) : float * float
let vLen (x,y) = sqrt (x*x + y*y) : float
```

Note the pattern-matching to get the components of the argument tuples

(Why did we have to type the function bodies?)

Another Matching Example with Tuples

`take n xs`, returns list of first `n` elements from the list `xs`

`take 2 ['a';'b';'c']` $\Longrightarrow$ `['a';'b']`

Let's define it using the `match ... with` construct!

(Solution on next slide)

```
let rec take n xs =
    match (n,xs) with
    | (0,_)      -> []
    | (_,[])     -> failwith "taking too many elements"
    | (_,x::xs) -> x :: take (n-1) xs
```

Note how we constructed a tuple just to be able to match on both arguments to `take`

The second, “local” `xs` “obscures” the “outer” `xs` in the recursive call `take (n-1) xs`

Excercise: provide some error handling for the case `n < 0`!