∼ Lecture I ∼

Keywords:

functional programming; expressions and values; functions; recursion; types.

References:

- P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.
- J. Backus. Can programming be liberated from the von Neumann style? *Communications of the ACM*, 21:613–641, 1978.
- [MLWP, Chapter 1]

Programming

- Programming is an intellectual activity.

  It is somehow close to proving theorems in mathematics (*cf.*, analysis of algorithms, program verification).

- Programming is hard.

  Software is notoriously unreliable. We need all the tools, principles, *etc.* that we can have to aid programming and thinking about it.

Why Functional Programming ?

- Offers a novel way of thinking about programming.

  Highlights expressiveness and clarity.

- Suitable for quick, easy, reliable, *etc.* prototyping.

  Security via type discipline.

- Susceptible to program correctness and/or verification.

  Ease of mathematical reasoning about programs.

Imperative Programming

State-based computation (= von Neumann style):

Imperative programs rely on modifying a *state* by using *commands*.

Programs are instructions specifying how to modify the state.

- Commands
  - Assignment
  - Control
    - Sequencing
    - Conditionals
    - Iteration

Functional Programming

Input/Output-based computation (= Mathematical style):

A functional program is an *expression*, and executing a program amounts to *evaluating* the expression to a *value*.

Features:

- No state ($\Rightarrow$ no memory cells and no assignment).
- No side effects.
- Referential transparency: One may replace equals by equals.
- Higher-order: Functions are first-class values.
- Static, strong, polymorphic typing.

Imperative vs. Functional

Factorial

`int fact(int n) {`
`  int x = 1;`
`  while (n > 0) {`
`    x = x * n;`
`    n = n - 1;`
`  }`
`  return x`
`}`
	`fun fact(n) =`
	`  if n = 0 then 1`
	`  else n * fact(n-1)`

Functional Programming

Advantages:

- Clearer semantics: programs correspond more directly to abstract mathematical objects.
- Conciseness and elegance: programs are shorter.
- Type system assists in the detection of errors and aids rapid prototyping.
- Better parametrisation and modularity of programs.
- Freedom in implementation; *e.g.*, parallelisation, lazy evaluation.

Disadvantages:

- Some programming needs are harder to fit into a purely functional model; *e.g.*, input/output modes, interactivity and continuously running programs (operating systems, process controllers).
- Historically functional languages have been less efficient than imperative ones; better compilers and runtime systems have largely closed the performance gap.

Imperative vs. Functional

State-based computation	Input/Output-based computation
Sequencing	Composition
Iteration	Recursion
Datatypes	Structured datatypes
—	Higher-order

Difficulties

Some standard responses:

- "It's too hard."
- "My employer doesn't use it."
- "Programs don't run as fast as in C."
- "I hate and/or don't understand all those type errors."
- "I want to do garbage collection/memory management myself."

NB: You will most surely need to change your way of thinking about programming.

Expressions

Expressions have a recursive, tree-like, structure. They are built-up from operators and arguments, by means of applications.

Examples:

1. `fact(1+(2*3))`
2. `fact(fact(4))+1`
3. `1 = 1+1`

In the context of *pure* expressions (*i.e.*, in the absence state change or side-effects), an expression always evaluates to the same value, and can thus be replaced by that value without affecting the program. This is called *referential transparency*.

Functions

Expressions consist mainly of function applications.

Functions may take any type of argument and return any type of result; 'any type' includes functions themselves—which are treated like other data.

Example:

```
fun doubleORsquare n
      = ( if n >= 0 then op+ else op* )(n,n)
```

Recursion

Recursive definition of functions is crucial to functional programming; there is no other mechanism for looping!

Examples:

1.
```
fun gcd (m,n)
  = if m = 0 then m else gcd(n mod m,m)
```

2.
```
fun even(n)
  = if n = 0 then true else odd(n-1)
and odd(n)
  = if n = 0 then false
    else if n = 1 then true
         else even(n-1)
```


Static, strong, polymorphic typing

Types classify data and let us ensure that they are used sensibly.

ML provides *static* (*i.e.*,compile-time), *strong*, *polymorphic* type checking, which can help catch programming errors. Polymorphism abstracts the types of parametric components.

Types are inferred automatically by the interpreter or compiler. Typically, type declarations are not required.

This course

- Basic types and tuples.
- Functions and recursion.
- List manipulation.
- Higher-order functions.
- Sorting.
- Abstraction and modularisation.
- Recursive Datatypes.
- Searching.
- Exceptions.
- Trees.
- Lazy lists.
- Types and type inference.
- Reasoning about functional programs.
- Case studies.