Functional programming
28 and 30 Sept. 2020

==========
Functional programming

Functional languages such as Lisp/Scheme and ML/Haskell/OCaml/F# are an attempt to realize Church's lambda calculus in practical form as a programming language.

The key idea: do everything by composing functions.
No mutable state; no side effects.

So how do you get anything done?

Recursion

Takes the place of iteration.

Some tasks are "naturally" recursive. Consider for example the function

$$\text{gcd}(a, b) = \begin{cases} a & \text{if } a = b \\ \text{gcd}(a-b, b) & \text{if } a > b \\ \text{gcd}(a, b-a) & \text{if } b > a \end{cases}$$

(Euclid's algorithm).

We might write this in C as

```
int gcd(int a, int b) {
   /* assume a, b > 0 */
   if (a == b) return a;
   else if (a > b) return gcd(a-b, b);
   else return gcd(a, b-a);
}
```

Other tasks we're used to thinking of as naturally iterative:

```
typedef int (*int_func) (int);
int summation(int_func f, int low, int high) {
   /* assume low <= high */
   int total = 0;
   int i;
   for (i = low; i <= high; i++) {
      total += f(i);
   }
   return total;
}
```

$$\sum_{low \le i \le high} f(i)$$

But there's nothing sacred about this "natural" intuition.
Consider:

```
int gcd(int a, int b) {
   /* assume a, b > 0 */
   while (a != b) {
      if (a > b) a = a-b;
      else b = b-a;
   }
   return a;
}
```

```
typedef int (*int_func) (int);
int summation(int_func f, int low, int high) {
   /* assume low <= high */
   if (low == high) return f(low);
   else return f(low) + summation(f, low+1, high);
}
```

More significantly, the recursive solution doesn't have to be any more expensive than the iterative solution. In OCaml, the gcd function would be written

```
let rec gcd a b =
  if a = b then a
  else if a > b then gcd (a - b) b
  else gcd a (b - a);;
```

Things to notice in this code:
- top-level forms, let
- rec
- necessity of else
- application via justaposition, use of parentheses
- double semicolons (tells REPL you're done and it should interpret)

Note that the recursive call is the ***last*** thing `gcd` does — no further computation after the return. This is called ***tail recursion***.
Functional language compilers will translate this as, roughly:

```
gcd(a, b) {
top:
   if a == b return a
   elsif a > b
      a := a - b
      goto top
   else
      b := b - a
      goto top
}
```

Functional programmers get good at writing functions that are naturally tail recursive. For example, instead of

```
let rec sum1 f low high =
  if low = high then f low
  else (f low) + (sum1 f (low + 1) high);;
```

we could write

```
let rec sum2 f low high st =
  if low = high then st + (f low)
  else sum2 f (low + 1) high (st + (f low));;
```

Here 'st' is a subtotal that accumulates what we've added up so far.

Things to notice in this code:
- Function application groups more tightly than addition.
- We could have left off the parentheses around "f low".
- In general, "normal" functions group left to right;
  - operators have precedence.

Unfortunately, now we have to provide an extra zero parameter to the call:

```
# sum1 (fun x -> x*x) 1 10;;
- : int = 385

# sum2 (fun x -> x*x) 1 10 0;;
- : int = 385
```

Things to notice in this code:
- `fun` is a ***lambda expression*** — a function definition

To get rid of that extra parameter, we can wrap it:

```
let sum3 f low high =
  let rec helper low st =
   let new_st = st + (f low) in
   if low = high then new_st
   else helper (low + 1) new_st in
  helper low 0;;
```

Things to notice in this code:
- internal let
- lexical nesting
- lack of rec on declaration of sum3
  - (compiler wouldn't have complained; just unnecessary)

NB: This tail recursive code exploits the associativity of addition; a compiler is unlikely to do that for us automatically. There exist automatic mechanisms to turn non-tail-recursive functions into tail-recursive ones, using what's known as ***continuation passing style***, but that wouldn't be as efficient in this case.

Sometimes you'll hear someone argue that recursion is algorithmically inferior to iteration. Fibonacci numbers are sometimes given as an example:

```
let rec fib1 n =
  match n with
  | 0 -> 1
  | 1 -> 1
  | _ -> fib1 (n-1) + fib1 (n-2);;
```

This takes $O(2^n)$ time, where O(n) is possible. In a von Neumann language we are taught to write

```
int fib(int n) {
   int f1 = 1; int f2 = 1;
   int i;
   for (i = 2; i <= n; i++) {
      int temp = f1 + f2;
      f1 = f2; f2 = temp;
   }
   return f2;
}
```

But there's no reason why we have to do it the slow way in OCaml.
We can write the following instead:

```
let fib2 n =
  let rec helper f1 f2 i =
   if i = n then f2
   else helper f2 (f1 + f2) (i + 1) in
  helper 0 1 0;;
```

Thinking about recursion as a direct, mechanical replacement for iteration is the wrong way to look at things. One has to get used to thinking in a recursive style.

NB: One can actually do better than O(n) for Fibonacci numbers. In particular, F(n) is the nearest whole number to $\varphi^n/\text{sqrt}(5)$, where $\varphi = (1 + \text{sqrt}(5))/2$, but this has high constant-factor costs and problems with numeric precision. For modest n, the O(n) algorithm is perfectly respectable.

NB2: OCaml has imperative features, so we ***can*** write the iterative version. It runs against the grain of the language, however (like writing C-like code in C++, only worse), and you won't be allowed to do it in this course.

NB3: Recursion isn't enough by itself to create a really useful functional language. You also need of ***higher-order functions*** (functional forms). More on this later.

A more complete list of necessary features for functional programming, many of which are missing in some imperative languages:
- recursion
- 1st class and high-order functions (including unlimited extent)
- serious polymorphism
- powerful list facilities
- fully general aggregates
- structured function returns
- garbage collection

Lisp also has
- homoiconography
- self-definition
- read-eval-print

ML/Haskell/F# have
- Milner type inference
- pattern matching
- implicit currying
- syntactic sugar: list comprehensions, monads

These are not necessarily present in other functional langs.

There are lots of functional programming languages.
Lisp and ML are the roots of the two main trees.

Lisp
- dates from about 1960.
- originally developed by John McCarthy, who received the Turing Award in 1971.
- inspired by the lambda calculus, Alonzo Church's mathematical formulation of the notion of computation (which you may have seen a bit of in 173).
- two most widely used dialects today are Common Lisp (big, full-featured) and Scheme/Racket (smaller and more elegant, but getting bigger).

ML
- dates from the mid-to-late 1970s.
- originally developed by Robin Milner, who received the Turning Award in 1991.
- intended to be safer and more readable than Lisp
- two most widely used dialects today are SML and OCaml.
  Many academics consider SML more elegant, but OCaml is more "practical" -- it has a better toolchain and is widely used in industry.
- Microsoft's F# is an OCaml descendant.
- Haskell is an ML descendant (through Miranda); it's the leading language for research in functional programming, and is increasingly popular in industry as well. Haskell is distinguished for being ***purely*** functional (no imperative features at all) and for using lazy (normal-order) evaluation.

Advantages of functional languages:

- lack of side effects makes programs easier to understand
- lack of explicit evaluation order (in some languages) offers possibility of parallel evaluation (e.g. MultiLisp)
- lack of side effects and explicit evaluation order simplifies some things for a compiler (provided you don't blow it in other ways)
- programs are often surprisingly short
- language can be small yet very "powerful"

Challenges:

- difficult (but not impossible!) to implement efficiently on von Neumann machines
  - lots of copying of data through parameters
  - (apparent) need to create a whole new array in order to change one element
  - very heavy use of references (space and time and locality problem)
  - frequent procedure calls
  - heavy space use for (non-tail) recursion
    - but anything you can write with a loop in an imperative language is straightforward to write as tail recursion
  - requires garbage collection

- difficult to integrate I/O into purely functional model
  leading approach is the monads of Haskell — sort of an imperative wrapper around a purely functional program; allows functions to be used not only to calculate values, but also to decide on the order in which imperative actions should be performed.

Requires a different mode of thinking by the programmer.