Introduction to Functional Programming
Higher Order Function

Quan Thanh Tho

Hochiminh City University of Technology
Faculty of Computer Science and Engineering
qttho@hcmut.edu.vn

Motivating Example

- Sorting problem

Lambda function

Alonzo Church (1903~1995)

Lambda Calculus

Church-Turing thesis

If an algorithm (a procedure that terminates) exists then there is an equivalent Turing Machine or applicable λ-function for that algorithm.

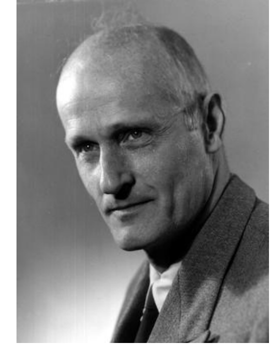
Alan M. Turing (1912~1954)

- Turing Machine
- Turing Test
- Head of Hut 8

Advisor:

Alonzo Church

History



- Origins: formal theory of substitution
  - For first-order logic, etc.
- More successful for computable functions
  - Substitution $\rightarrow$ symbolic computation
  - Church/Turing thesis
- Influenced design of Lisp, ML, other languages
- Important part of CS history and foundations

What is a Functional Language?

- “No side effects”
- Pure functional language: a language with functions, but without side effects or other imperative features

No-Side-Effects Language Test

Within the scope of specific declarations of x_1,x_2, …, x_n, all occurrences of an expression e containing only variables x_1,x_2, …, x_n, must have the same value.

```
begin
    integer x=3; integer y=4;
     5*(x+y)-3
     …   // no new declaration of x or y //
     4*(x+y)+1
end
```

$$\mathrm{sqsum}(x, y) = x \times x + y \times y \qquad (x, y) \mapsto x \times x + y \times y$$

$$\mathrm{id}(x) = x \qquad ?$$

Currying Form

$$(x, y) \mapsto x \times x + y \times y$$

Expressions and Functions

- Expressions

  $x + y \qquad\qquad x + 2*y + z$

- Functions

  $\lambda x.\ (x+y) \qquad\qquad \lambda z.\ (x + 2*y + z)$

- Application

  $(\lambda x.\ (x+y))\ 3 \qquad = \ 3 + y$

  $(\lambda z.\ (x + 2*y + z))\ 5 \qquad = \ x + 2*y + 5$

Parsing: $\lambda x.\ f\ (f\ x) = \lambda x.(\ f\ (f\ (x))\ )$

\lambda x.x $\lambda x.x$ \x.x

(\x.x) y

(\x.y)

Higher-Order Functions

- Given function f, return function $f \circ f$

  $\lambda f.\ \lambda x.\ f\ (f\ x)$

- How does this work?

  $(\lambda f.\ \lambda x.\ f\ (f\ x))\ \ (\lambda y.\ y+1)$

Higher-Order Functions

- Given function f, return function $f \circ f$

  $\lambda f.\ \lambda x.\ f\,(f\,x)$

- How does this work?

$(\lambda f.\ \lambda x.\ f\,(f\,x))\ \ (\lambda y.\ y+1)$

$=\ \ \lambda x.\ (\lambda y.\ y+1)\ ((\lambda y.\ y+1)\ \ x)$

$=\ \ \lambda x.\ (\lambda y.\ y+1)\ (x+1)$

$=\ \ \lambda x.\ (x+1)+1$

Same Procedure (ML)

- Given function f, return function $f \circ f$

  fn f => fn x => f(f(x))

- How does this work?

  (fn f => fn x => f(f(x))) (fn y => y + 1)

  = fn x => ((fn y => y + 1) ((fn y => y + 1) x))

  = fn x => ((fn y => y + 1) (x + 1))

  = fn x => ((x + 1) + 1)

Declarations as “Syntactic Sugar”

```
function f(x) {
    return x+2;
}
f(5);
```

Declarations as "Syntactic Sugar"

```
function f(x) {
    return x+2;
}
f(5);
```

$\underbrace{(\lambda f.\ f(5))}_{\text{block body}}\ \underbrace{(\lambda x.\ x+2)}_{\text{declared function}}$

Declarations as “Syntactic Sugar”

```
function f(x) {
    return x+2;
}
f(5);
```

$$\underbrace{(\lambda f.\ \ f(5))}_{\text{block body}}\ \ \underbrace{(\lambda x.\ x+2)}_{\text{declared function}}$$

Python code:

```
f = lambda x: x + 2
f(5)
```

Declarations as "Syntactic Sugar"

```
function f(x) {
    return x+2;
}
f(5);
```

$$\underbrace{(\lambda f.\ f(5))}_{\text{block body}}\ \underbrace{(\lambda x.\ x+2)}_{\text{declared function}}$$

Python code:

```
(lambda f: f(5))(lambda x: x+2)
```

We can do everything

- The lambda calculus can be used as an "assembly language"
- We can show how to compile useful, high-level operations and language features into the lambda calculus
  - Result = adding high-level operations is convenient for programmers, but not a computational necessity
  - Result = make your compiler intermediate language simpler

Partially applicable function

- Function producing new and more specialized function from its first argument
  - `add = lambda x: lambda y: x+y`

  or
  - def add(x):

    return lambda y: x + y
  - What type of add?
  - How does add operate?
    - `succ = add(1)`
    - `pred = add(-1)`
    - `(add(10)) (5)`
    - `add(10)(5)`

More examples

- Function that takes “more than one” argument
  - `plus = lambda x, y: x+y`
  - `curry = lambda f: lambda x: lambda y: f(x, y)`
  - `curry(plus)(2)(3) = ?`

Functional Arguments

- Higher-order function: a curried function taking other function as argument
  - `square = lambda x: x*x`
  - `twice = lambda f, x: f (f (x))`
  - `twice(square, 3) = ?`

The map utility

- `double = lambda x: x * 2`
- `list(map(double, [1,2,3]))`

The reduce utility

- `from functools import reduce`
- `sum_arr = lambda arr: reduce(lambda x, y: x+y, arr, 0)`
- `sum_arr([1,2,3]) = ?`