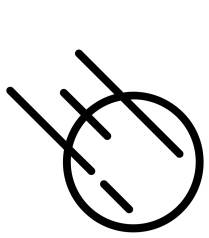


Functional Programming

- Functional programming is a **declarative programming paradigm** where programs are constructed by applying and composing functions.
- Function definitions are **expressions that map values to other values**, rather than a sequence of imperative statements which update the running state of a program.



Functional Programming

Everything is a Value!

- ...including functions!
- This sets functional programming apart from imperative programming where statements like loops and conditionals do not represent values but change of an explicit machine state



Lambda Calculus

- Let's explore this using the lambda calculus before we commit to any particular language.
- Recall that in the lambda calculus we construct functions as lambda expressions and these functions can be applied to values, e.g.

$$\begin{array}{ccc} \text{Function application} & & \text{Substitution} \\ \underbrace{\hspace{10em}} & & \underbrace{\hspace{10em}} \\ (\lambda x. x + 1) \ 1 & \Rightarrow & x + 1[x \leftarrow 1] \Rightarrow 1 + 1 \Rightarrow 2 \end{array}$$

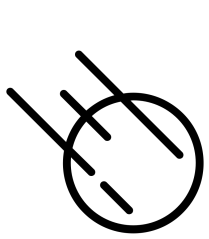


Lambda Calculus

- Functions can be input values to other functions!

Function as value

$$\begin{aligned} & \underbrace{(\lambda y. y \ 1)}_{\text{Function as value}} (\lambda x. x + 1) \Rightarrow y \ 1 [y \leftarrow (\lambda x. x + 1)] \\ & \Rightarrow (\lambda x. x + 1) \ 1 \Rightarrow 2 \end{aligned}$$

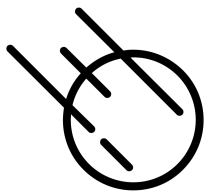


Lambda Calculus

- Functions as return values from functions
 - That is, functions computing new functions!

Function as return value

$$\begin{aligned} & \overbrace{(\lambda x. (\lambda y. x + y))} \text{Function as return value} 1 1 \Rightarrow (\lambda y. x + y) 1 [x \leftarrow 1] \\ & \Rightarrow (\lambda y. 1 + y) 1 \Rightarrow 1 + y [y \leftarrow 1] \Rightarrow 1 + 1 \Rightarrow 2 \end{aligned}$$



Functional Programming

- Functional programming is declarative in that the programs deal more with the **what** rather than the **how**.
- One way to think about this is: in declarative programming we “declare” **what to do for each input configuration**.
- This is in stark contrast to imperative programming where we describe **how to solve the whole problem** in one go without subdivision.

```
-- imperative solution
function len with list do
  let remaining_list = list.
  let cnt = 0.
  repeat
    let [_|remaining_list] = remaining_list.
    let cnt = cnt + 1.
  until remaining_list is [].
end

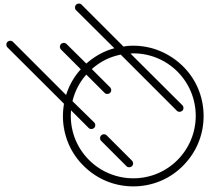
let q = [ 1 to 10].
assert (len q == 10).
```

“The How”

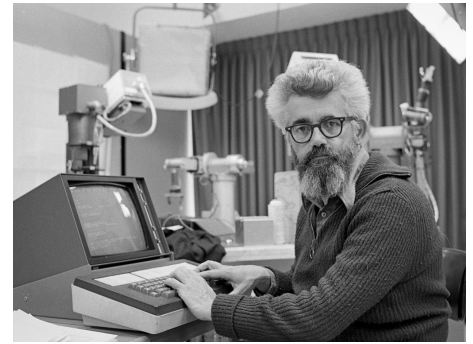
```
-- declarative solution
function len
  with [] do
    0
  with [_|remaining_list] do
    1 + len remaining_list
  end

let q = [ 1 to 10].
assert (len q == 10).
```

“The What”



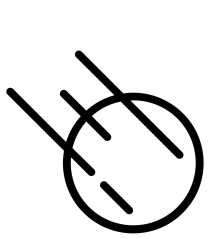
Lisp



Dr John McCarthy, computer scientist,
1927 – 2011.



- Lisp was developed by John McCarthy in the late 1950's early 60's to solve problems in AI.
- It is the oldest functional programming language.
- Its syntax has been inspired by the lambda calculus.
- It introduced novel features such as recursion and garbage collection.
- It is still in use today as Common Lisp (ANSI compliant).
- Modern descendants: Scheme, Racket, Clojure



Lisp

$(\lambda x. x + 1) 1 \Rightarrow 2$

```
Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>
```

```
Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010
```

```
Type :h and hit Enter for context help.
```

```
[1]> ((lambda (x) (+ x 1)) 1)
2
```

```
[2]> (defun inc (x) (+ x 1))
INC
```

```
[3]> (inc 1)
2
```

```
[4]> █
```

$(\lambda y. y 1)(\lambda x. x + 1) \Rightarrow 2$

```
[1]> ((lambda (y) (apply y '(1))) (lambda (x) (+ x 1)))
2
[2]> █
```

$(\lambda x. (\lambda y. x + y)) 1 1 \Rightarrow 2$

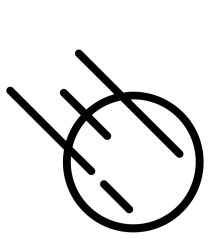
```
[1]> (apply (apply (lambda (x) (lambda (y) (+ x y))) '(1)) '(1))
2
[2]> █
```




Robin Milner, computer scientist
1934 – 2010.



- Robin Milner designed ML as the implementation language for his proof assistant LCF (Logic for Computable Functions) in the 1970's.
- It can be considered the first modern functional programming language,
 - Statically type checked
 - A syntax that is easily recognized by today's developers
 - Very influential, virtually every modern functional programming language can trace its ancestry back to ML
- It is also one of the few high-level programming languages with a full mathematical specification.
- Dialects of ML in wide use today: SMLNJ, Ocaml, F#



ML

$(\lambda x. x + 1) 1 \Rightarrow 2$

```
Standard ML of New Jersey (64-bit) v110.95 [built: Sun Nov 06 00:04:31 2022]
```

```
- (fn x => x + 1) 1;
```

```
val it = 2 : int
```

```
-
```

$(\lambda y. y 1)(\lambda x. x + 1) \Rightarrow 2$

```
- (fn y => y 1)(fn x => x+1);
```

```
val it = 2 : int
```

```
-
```

$(\lambda x. (\lambda y. x + y)) 1 1 \Rightarrow 2$

```
- (fn x => (fn y => x+y)) 1 1;
```

```
val it = 2 : int
```

```
-
```

```
Standard ML of New Jersey (64-bit) v110.95 [built: Sun Nov 06 00:04:31 2022]
```

```
- fun inc x = x+1;
```

```
val inc = fn : int -> int
```

```
- inc 1;
```

```
val it = 2 : int
```

```
-
```



Function as Values: Another Look

A Type is a Set of Values.

- If we view functions as values, then they have to belong to a type.
- We can use ML's type system to compute the function types,

```
- fun inc x = x+1;  
val inc = fn : int -> int
```

```
- (fn x => 2*x);  
val it = fn : int -> int  
- █
```

```
- fun fold (x,y) = x+y;  
val fold = fn : int * int -> int  
- █
```



Function as Values: Another Look

- In the previous slide we saw that we have at least two different types

$int \rightarrow int$ ← Function Types → $int * int \rightarrow int$

```
- (fn x => 2*x);  
val it = fn : int -> int  
- |
```

```
- fun inc x = x+1;  
val inc = fn : int -> int
```

*“**All** functions that map integers to integers”*

```
- fun fold (x,y) = x+y;  
val fold = fn : int * int -> int  
- |
```

*“**All** functions that map pairs of integers to integers”*



Function as Values: Another Look

- Since we now have function types we can declare variables of that type,

ML

```
- val x:int->int = (fn x => x+1);  
val x = fn : int -> int  
- █
```

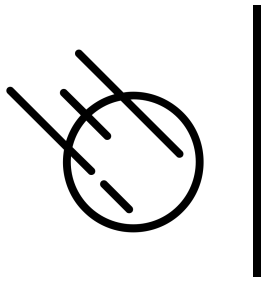
Rust

```
fn main() {  
    let x: fn(i32) -> i32 = |x| x + 1;  
}
```



Function as Values: Another Look

- Every function belongs to a particular function type.
- We can view a function as a value in the set of all values of a particular type.
- This particularly visible in statically typed languages like ML and Rust.
 - But it is also supported in dynamically typed languages like Python and Asteroid.
 - In Asteroid, all functions are members of the type 'function'.



Reading

- Please read Chapter I in the following paper,

lutzhamel.github.io/CSC493/docs/intro-fp-barendregt.pdf