Lec 4: Recursion

CS220: Programming Principles

Sang Kil Cha

Recap: Function

Mathematical Functions

```
let addByOne x = x + 1
```

This function maps a domain (integers) onto a range (integers).

A function is a relation that associates each element of a set X to a single element of another set Y[1]:

$$f : X \mapsto Y.$$

[1] A quote from Wikipedia.

Mathematical Function is Pure

We say a function is ***pure*** if it is a mathematical function. Particularly, we say a function is pure if it satisfies the following properties:

1. Its return value is always determined by its argument.
2. Its evaluation has ***no side effects***.

Mathematical Function is Pure

We say a function is ***pure*** if it is a mathematical function. Particularly, we say a function is pure if it satisfies the following properties:

1. Its return value is always determined by its argument.
2. Its evaluation has ***no side effects***.

Q: What is a side effect?

How Can We Create Side Effects?

You don't need to know! (yet). In fact, you already have seen one ***impure*** function:

```
printfn "Hello World"
```

The Power of Pure Functions

1. They are trivially parallelizable.
2. They only need to be evaluated once for a certain input. Thus, they can benefit from caching.
3. And many more ... (we will see them later in this course)

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don't accept late responses.

Recursion

Motivation

How can we make a control-flow (for-loop, while-loop, etc.) when there is no side-effect?

```
for i in range(0, 10):
    print(i)
```

What is Recursion?

Recursion means a function calls itself.

In purely functional programming, we rely only on recursion instead of using loops (e.g., `for`, `while` statements).

Square Roots

Write a square-root function.

$$\sqrt{x} = y \text{ such that } y \geq 0 \text{ and } y^2 = x$$

Square Roots

Write a square-root function.

$$\sqrt{x} = y \text{ such that } y \geq 0 \text{ and } y^2 = x$$

Square root function in F#?

```
let sqrt x = // ?
```

Square Roots

Write a square-root function.

$$\sqrt{x} = y \text{ such that } y \geq 0 \text{ and } y^2 = x$$

Square root function in F#?

```
let sqrt x = // ?
```

What is the difference?

How to Compute Square Roots?

Newton's Method: we start with a random guess, and iteratively improve our guess until we reach a certain threshold.

Example: Computing the square root of 2.

Guess	Quotient	Average
1	(2 / 1) = 2	((2 + 1) / 2) = 1.5
1.5	(2 / 1.5) = 1.3333	((1.3333 + 1.5) / 2) = 1.4167
1.4167	(2 / 1.4167) = 1.4118	((1.4167 + 1.4118) / 2) = 1.4142
1.4142	...	...

In Python?

```
threshold = 0.001

def isGoodEnough(guess, x):
return abs(guess * guess - x) < threshold

def improve(guess, x):
return (guess + (x / guess)) / 2.0

def newton(initialGuess, x):
guess = initialGuess
while not isGoodEnough(guess, x):
  guess = improve(guess, x)
return guess

def sqrt(x):
return newton(1.0, x)
```

Q: How do we avoid assignment?

Implementing Newton's Method.

```
let threshold = 0.001

let square x = x * x

let isGoodEnough guess x =
  abs (square guess - x) < threshold

let improve guess x = (guess + (x / guess)) / 2.0

let newton guess x = // Doesn't compile
  if isGoodEnough guess x then guess
  else newton (improve guess x) x

let sqrt x = newton 1.0 x
```

Compile Error?

F# functions are not recursive by default. We should use the keyword `rec` to make a function recursive. Below is the “diff”.

```
@@ -7,7 +7,7 @@

  let improve guess x = (guess + (x / guess)) / 2.0

- let newton guess x = // Doesn't compile
+ let rec newton guess x =
```

Why Not Recursive by Default?

This is basically a language design choice. F# gives the flexibility for users to choose whether the name of a let-binding can be referenced within the scope of its own body. SML, which is F#'s ancestor, uses recursion by default, for instance.

Revisiting Abstraction

At each function we don't need to care how other functions are implemented because functions provide abstraction about the procedures. For instance,

- Any implementation of `sqaure` works fine for `isGoodEnough`.
- Parameter names for a function does not matter for the caller of the function.

Hiding Details

The only function that is needed by a user would be `sqrt`. Can we hide the others somehow?

> One of the key aspects of abstraction is to hide the implementation details.

Hide with local definitions.

```
let sqrt x =
  let threshold = 0.001
  let square x = x * x
  let isGoodEnough guess x =
    abs (square guess - x) < threshold
  let improve guess x =
    (guess + (x / guess)) / 2.0
  let rec newton guess x =
    if isGoodEnough guess x then guess
    else newton (improve guess x) x
  newton 1.0 x
```

Hide with local definitions.

```
let sqrt x =
  let threshold = 0.001
  let square x = x * x
  let isGoodEnough guess x =
    abs (square guess - x) < threshold
  let improve guess x =
    (guess + (x / guess)) / 2.0
  let rec newton guess x =
    if isGoodEnough guess x then guess
    else newton (improve guess x) x
  newton 1.0 x
```

The variable `x` is accessible without parameter passing.

Make it simpler without explicitly passing around the variable x.

```
let sqrt x =
  let threshold = 0.001
  let square x = x * x
  let isGoodEnough guess =
    abs (square guess - x) < threshold
  let improve guess =
    (guess + (x / guess)) / 2.0
  let rec newton guess =
    if isGoodEnough guess then guess
    else newton (improve guess)
  newton 1.0
```

Make it simpler without explicitly passing around the variable x.

```
let sqrt x =
  let threshold = 0.001
  let square x = x * x
  let isGoodEnough guess =
    abs (square guess - x) < threshold
  let improve guess =
    (guess + (x / guess)) / 2.0
  let rec newton guess =
    if isGoodEnough guess then guess
    else newton (improve guess)
  newton 1.0
```

Variables in a function are only usable within the function.

In-Class Activity #03

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.
   - `git clone https://github.com/KAIST-CS220/CS220-Main.git`
2. Move in to the directory `CS220-Main/Activities`
   - `cd CS220-Main`
   - `cd Activities`

The Problem

Modify the `gcd` function to compute GCD (Greatest Common Divisor) of two given integers.

The algorithm formally looks like below.

$$\text{gcd}(a, 0) = a$$
$$\text{gcd}(a, b) = \text{gcd}(b, a \bmod b).$$

Recursive Patterns

Linear Recursive Functions

A function that makes a single call to itself each time the function runs.

Example: factorial function.

```
let rec factorialA n =
  if n <= 1 then 1
  else n * factorialA (n - 1)
```

Evaluation Process: `factorialA 6`

```
factorialA 6
6 * factorialA 5
6 * (5 * factorialA 4)
6 * (5 * (4 * factorialA 3))
6 * (5 * (4 * (3 * factorialA 2)))
6 * (5 * (4 * (3 * (2 * factorialA 1))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
6 * 120
720
```

Another Implementation

Let's increment a counter, and multiply the counter value for each recursion.

Example: factorial function with a counter.

```
let factorialB n =
  let rec iter product counter max =
    if counter > max then product
    else iter (counter * product) (counter + 1) max
  iter 1 1 n
```

Evaluation Process: `factorialB 6`

```
factorialB 6
iter 1 1 6
iter 1 2 6
iter 2 3 6
iter 6 4 6
iter 24 5 6
iter 120 6 6
iter 720 7 6
720
```

Comparison

```
factorialA 6
6 * factorialA 5
6 * (5 * factorialA 4)
6 * (5 * (4 * factorialA 3))
6 * (5 * (4 * (3 * factorialA 2)))
6 * (5 * (4 * (3 * (2 * factorialA 1))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
6 * 120
720
```

```
factorialB 6
iter 1 1 6
iter 1 2 6
iter 2 3 6
iter 6 4 6
iter 24 5 6
iter 120 6 6
iter 720 7 6
720
```

Comparison

```
factorialA 6
6 * factorialA 5
6 * (5 * factorialA 4)
6 * (5 * (4 * factorialA 3))
6 * (5 * (4 * (3 * factorialA 2)))
6 * (5 * (4 * (3 * (2 * factorialA 1))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
6 * 120
720
```

```
factorialB 6
iter 1 1 6
iter 1 2 6
iter 2 3 6
iter 6 4 6
iter 24 5 6
iter 120 6 6
iter 720 7 6
720
```

`factorialA` defers multiplication operations per each recursion.

Cost of Deferred Operations

`factorialA` needs ***memory*** to keep track of the operations to be performed later on, the size of which grows linearly with n.

Cost of Deferred Operations

`factorialA` needs ***memory*** to keep track of the operations to be performed later on, the size of which grows linearly with n.

Therefore, we prefer the second pattern, i.e., `factorialB`, which is often called ***iterative process***, and the function is referred to be ***tail-recursive***.

Tail Recursion

A tail-recursive function is a function that calls itself at the end, i.e., the ***tail***, of the function without any computation after the return of recursive calls.

> If there is no computation after the return of recursive calls, we don't need to defer operations!

Tree Recursion

When there are multiple recursive calls within a function.

Example: Write a function that computes Fibonacci numbers.

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{otherwise} \end{cases}$$

Can you write it in a tail-recursive manner?

Always Use Tail-Recursion?

Tail-recursive functions are efficient, but can be ***unintuitive***!

> The trade-off between readability (understandability) vs. efficiency.

Conclusion

- Functions in programming language represent a procedure, i.e., they show ***how*** to operate.
- ***Recursion*** is a natural way to represent ideas. When humans perform repetitive tasks, we do the same thing over and over again until we reach a terminating condition.
- There are typical patterns in writing a recursive function.
- ***Tail recursion*** is important for performance-critical functions.

Question?