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*. 

**Q:** What is a side effect?
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, we already have learned 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)
Recursion
What is Recursion?

Recursion means a function calls itself.

In functional language, we rely only on recursion instead of using loops (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.

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

Write this function in F#?

Square root function in F#?

```fsharp
let sqrt x = // ?
```
Square Roots

Write a square-root function.

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

Write this function in F#?

Square root function in F#?

```fsharp
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.

<table>
<thead>
<tr>
<th>Guess</th>
<th>Quotient</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(2 / 1) = 2</td>
<td>((2 + 1) / 2) = 1.5</td>
</tr>
<tr>
<td>1.5</td>
<td>(2 / 1.5) = 1.3333</td>
<td>((1.3333 + 1.5) / 2) = 1.4167</td>
</tr>
<tr>
<td>1.4167</td>
<td>(2 / 1.4167) = 1.4118</td>
<td>((1.4167 + 1.4118) / 2) = 1.4142</td>
</tr>
<tr>
<td>1.4142</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Implementing Newton's Method.

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

```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 =
```

Compile Error?
Why Not Recursive by Default?

This is basically a language design choice. 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 `square` 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 $\sqrt{\cdot}$. Can we hide the others somehow?

One of the key aspects of abstraction is to hide the implementation details.
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
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 used within the internal functions are accessible without parameter passing.
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 \( x \). 

```ml
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 #04

Modify the myfunc function to compute GCD (Greatest Common Divisor) of two given integers. You can ignore error cases, i.e., integer overflows.

The algorithm formally looks like below.

\[
gcd(a, 0) = a \\
gcd(a, b) = gcd(b, a \mod b). \]

Recursive Patterns
Linear Recursive Functions

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

Example: factorial function.

```ml
let rec factorialA n =
  if n <= 1 then 1
  else n * factorialA (n - 1)
```
Evaluation Process: $\text{factorialA } 6$

$\text{factorialA } 6$
$6 * \text{factorialA } 5$
$6 * (5 * \text{factorialA } 4)$
$6 * (5 * (4 * \text{factorialA } 3))$
$6 * (5 * (4 * (3 * \text{factorialA } 2)))$
$6 * (5 * (4 * (3 * (2 * \text{factorialA } 1)))))$
$6 * (5 * (4 * (3 * (2 * 1)))))$
$6 * (5 * (4 * (3 * 2)))$
$6 * (5 * (4 * 6))$
$6 * (5 * 24)$
$6 * 120$
$720$
Another Implementation

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

Example: factorial function with a counter.

```plaintext
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: \texttt{factorialB 6}

\begin{verbatim}
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
\end{verbatim}
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

defers multiplication operations per each recursion.
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 * 6)
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 \textit{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.
Another Example: Exponentiation

Compute the exponential of a given number.

Simple linear recursion.

```ml
let exp b n =
  if n = 0 then 1
  else b * exp (n - 1)
```
Tail-recursion.

```ocaml
let exp b n =
  let rec iter b counter product =
    if counter = 0 then product
    else iter b (counter - 1) (b * product)
  in iter b n 1
```
Tail-recursion.

```ml
let exp b n =
  let rec iter b counter product =
    if counter = 0 then product
    else iter b (counter - 1) (b * product)
  in iter b n 1
```

Can we make it faster?
Faster Algorithm

No need to multiply $n$ times.

\[ b^n = \begin{cases} 
\left(\frac{b^n}{2}\right)^2 & \text{if } n \text{ is even.} \\
 b \cdot b^{n-1} & \text{if } n \text{ is odd.} 
\end{cases} \]
let isEven n = n % 2 = 0
let square n = n * n

let rec fastExp b n =
    if n = 0 then 1
    elif isEven n then square (fastExp b (n/2))
    else b * fastExp b (n - 1)

elif is equivalent to else if.
Measure Execution Time in REPL

```plaintext
#time exp 2 1000000 #time

#time fastExp 2 1000000 #time

Caveat: the result will be invalid due to integer overflow.
```
Scope
Locally Declared Identifiers

We learned from the previous lecture that let-bindings can be nested, but with a careful indentation.

```ml
let x = 1
let f x = x + x
f 10 // ?
let g a =
  let x = 10
  a + x
g 10 // ?
x // ?
```
Static (Lexical) Scoping vs. Dynamic Scoping

Most programming languages use static scoping, meaning that name resolution depends on the lexical context. In dynamic scoping, however, name resolution depends on the (dynamic) execution context.

An example function area.

```plaintext
let pi = 3.14
let area r = pi * r * r
let myarea =
  let pi = 6.0
  area 10.0 // ?
```
Evaluating a Function with Closure

We can evaluate functions into a value by means of a closure. A closure is a triple:

$$(\text{arg}, \text{body}, \text{env})$$

where arg is the argument expression, body is the function body expression, and the env is an environment.
Closure Example

An example function `area`.

```haskell
let pi = 3.14
let area r = pi * r * r
let myarea =
    let pi = 6.0
    area 10.0 // ?
```

We can represent the closure of `area` as follows:

- **arg**: `r`
- **body**: `pi * r * r`
- **env**: `{pi ↦ 3.14}`
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?