Prof. Liang Huang
huang@qc.cs.cuny.edu
Computer science is not really about computers -- and it's not about computers in the same sense that physics is not really about particle accelerators, and biology is not about microscopes and Petri dishes... and geometry isn't really about using surveying instruments... when some field is just getting started and you don't really understand it very well, it's very easy to confuse the essence of what you're doing with the tools that you use.

— Harry Abelson (1986), Intro to Computer Science Course, MIT

Computer Science is no more about computers than astronomy is about telescopes.

— (Mis)attributed to Edsger Dijkstra, 1970.
Computer Science

• CS is not a science -- it does not require experimental tests
• CS is not engineering -- it does not have physical constraints and it cares more about abstraction than reality
• CS is
  • a “magical” branch of pure and abstract mathematics
  • an “abstract” version of engineering w/o physical constraints
• CS focuses on abstraction and mathematical rigor
  • abstraction is even more important than rigor
  • definitions are important than (mostly boring) proofs
What is PL all about?

• a formal perspective on programs and programming
  • view programs and whole languages as mathematical objects
  • make and prove rigorous claims about them
    • important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...)
• detailed study of a range of language features

• our approaches: abstraction and rigorous proofs
  • geometry: abstract and precise description of space/shape
  • CS: abstract and precise description of “processes”
What this course is not...

- an introduction to programming
  - you should be fluent in at least two mainstream languages
- a comparative survey of different programming languages
  - this course is the “linguistics” of programming languages
  - “comparative literature” rather than English/Russian/Chinese...
- a course on functional programming (FP)
  - though we’ll teach Haskell and FP is everywhere in this course
- a course on compilers
  - you should know lexical analysis, parsing, and abstract syntax
Syllabus

- **Part 0: Background**
  - Functional Programming with Haskell
  - Inductive Proofs
- **Part I: Basics**
  - Operational Semantics
  - \( \lambda \)-calculus
- **Part II: Type Systems**
  - Simply Typed \( \lambda \)-calculus
  - Type Safety
  - Subtyping
Functional Programming
and Haskell Tutorial

(with a comparison to Python, the best non-functional language)
Functional Programming

- **functional languages** (Lisp/Scheme, Haskell, SML/Ocaml, Scala, ...)
  - persistent data structures (immutable once built)
  - recursion as the primary control structure
  - heavy use of higher-order functions (functions that take functions as arguments and return functions as results)

- **imperative languages** (C/C++, Java, Python, ...)
  - mutable data structures
  - looping rather than recursion
  - first-order rather than higher-order programming
  - Python incorporated many functional programming features!
First Impression: Quicksort

```java
public void sort(int low, int high) {
    if (low >= high) return;
    int p = partition(low, high);
    sort(low, p);
    sort(p + 1, high);
}

void swap(int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

int partition(int low, int high) {
    int pivot = a[low];
    int i = low - 1;
    int j = high + 1;
    while (i < j) {
        i++; while (a[i] < pivot) i++;
        j--; while (a[j] > pivot) j--;
        if (i < j) swap(i, j);
    }
    return j;
}
```

```python
def sort(a):
    if a == []:
        return []
    else:
        p = a[0]
        left = [x for x in a if x < p]
        right = [x for x in a[1:] if x >= p]
        return sort(left) + [p] + sort(right)
```

```haskell
qsort [] = []
qsort (p:xs) = qsort left ++ [p] ++ qsort right
 where
    left = [x|x<-xs, x<p]
    right = [x|x<-xs, x>=p]
```

**do not** use C/C++ or Java anymore!
The GHC Interpreter

```haskell
[<lhuang@Mac OS X:~>] ghci
GHCi, version 7.6.3:
Prelude> 3 + 5
8
Prelude> 3 / 5
0.6
Prelude> 3 /= 5
True
Prelude> [1, 2] ++ [3]
[1, 2, 3]
Prelude> let a = [1, 2] ++ [3]
Prelude> a
[1, 2, 3]
Prelude> length a
3
Prelude> reverse a
[3, 2, 1]
Prelude> min 3 5
3
Prelude> [3, "a"]
No instance for (Num Char) ...
```
Lists (and Strings)

- Haskell lists are homogenous (Python: heterogenous)
- Haskell strings are list of chars (Python: string is not list, and no distinction b/w str and char)

```haskell
Prelude> "hello" ++ " " ++ "world"
"hello world"
Prelude> ["h", "e"]
["h","e"]
Prelude> ['h', 'e']
"he"
Prelude> "3" + [3]  
Error
```

```python
>>> "hello" + " " + "world"
"hello world"
>>> ["h", "e"]
["h", "e"]
>>> ['h', 'e']
["h", "e"]
>>> "3" + [3]
["3", 3]
```
Lists are LinkedLists

- Haskell lists are linked lists (Python: array of pointers)
  - cons operator : , head and tail
- but Haskell also support random-access features

```
Prelude> 'A':" SMALL CAT"
"A SMALL CAT"
Prelude> "A":" SMALL CAT"
  Error
Prelude> "A":[" SMALL CAT"]
["A"," SMALL CAT"]
Prelude> head [1,2,3]
1
Prelude> tail [1,2,3]
[2,3]
Prelude> null [1]
False
Prelude> [1,2,3] !! 1
2
```

```python
>>> [1,2,3][0]
1
>>> [1,2,3][1:]
[2,3]
>>> [1] == []
False
>>> [1,2,3][1]
2
```
Function Definition and Application

Prelude> let f x = x+1
Prelude> f 5
6
Prelude> map f [1..10]
[2,3,4,5,6,7,8,9,10,11]

Prelude> let g x y = x+y
Prelude> g 2 3
5
Prelude> (g 2) 3
5
Prelude> let g2 = g 2
Prelude> g2 3
5
Prelude> map g2 [1..10]
[3,4,5,6,7,8,9,10,11,12]
Conditionals

```
Prelude> if 3 < 5 then [] else [3]
[]
Prelude> let min' x y = if x < y then x else y
Prelude> min' 3 5
3
Prelude> let f = min' 3
Prelude> f 5
3
Prelude> f 2
2

>>> [] if 3 < 5 else [3]
[]
>>> define min1(x, y): x if x < y else y
>>> min1(3, 5)
3
>>> define f(x) = lambda x: min1(3, x)
>>> f(5)
3
>>> f(2)
2
```

```python
from functools import partial
define f(x) = lambda x: partial(min1, 3)
```
List Comprehensions

```haskell
Prelude> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
Prelude> let f x = x*2
Prelude> [f x | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
Prelude> map f [1..10]
[2,4,6,8,10,12,14,16,18,20]

Prelude> [x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]

Prelude> let a = [2..20]
Prelude> [x | x <- a, null [y | y<-a, y<x, x `mod` y == 0]]
[2,3,5,7,11,13,17,19]

Prelude> let nouns = ["hobo", "frog", "pope"]
Prelude> let adjs = ["lazy", "grouchy"]
Prelude> [adj ++ " " ++ noun | adj <- adjs, noun <- nouns]
["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog","grouchy pope"]
```
Recursive Functions: fact and fib

• recursion is the essence of functional programming

• also essence of whole computer science, and the human species

```haskell
Prelude> let fact n = if n == 0 then 1 else n * fact (n-1)
Prelude> fact 5
120
Prelude> let fib n = if n < 3 then 1 else fib (n-1) + fib (n-2)
Prelude> fib 10
55
Prelude> let s a = if null a then 0 else head a + s (tail a)
Prelude> s [1,2,3]
6
Prelude> sum [1,2,3]
6
Prelude> let rev a = if null a then [] else rev (tail a) ++ [head a]
Prelude> rev [1,2,3]
[3,2,1]
Prelude> reverse [1,2,3]
[3,2,1]
```

```python
def rev(a): return [] if a==[] else rev(a[1:])+[a[0]]
```
Quicksort (and load)

```python
def qsort(a):
    if a == []:
        return []
    else:
        pivot = a[0]
        left = [x for x in a if x < pivot]
        right = [x for x in a[1:] if x >= pivot]
        return qsort(left) + [pivot] + qsort(right)
```

Prelude> let qsort a = if null a then [] else qsort [x | x<-a, x<head a] ++ [head a] ++ qsort [x | x <- tail a, x >= (head a)]
Prelude> qsort [2,1,3]
[1,2,3]

```haskell
qsort a =
    if null a then []
    else qsort [x | x <- a, x<head a]
        ++ [head a]
        ++ qsort [x | x <- tail a, x >= (head a)]
```

```bash
<lhuan@Mac OS X:~/teaching/PL> cat qsort.hs
qsort a =
    if null a then []
    else qsort [x | x<-a, x<head a]
        ++ [head a]
        ++ qsort [x | x <- tail a, x >= (head a)]

<lhuan@Mac OS X:~/teaching/PL> ghci
```
Pattern Matching

\[
\text{fact } n = \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1)
\]

\[
\text{fact } 0 = 1 \\
\text{fact } n = n \times \text{fact } (n-1)
\]

\[
\text{fib } n = \text{if } n < 3 \text{ then } 1 \text{ else } \text{fib } (n-1) + \text{fib } (n-2)
\]

\[
\text{fib } 1 = 1 \\
\text{fib } 2 = 1 \\
\text{fib } n = \text{fib } (n-1) + \text{fib } (n-2)
\]

\[
\text{rev } a = \text{if } \text{null } a \text{ then } [ ] \text{ else } \text{rev } (\text{tail } a) ++ [\text{head } a]
\]

\[
\text{rev } [ ] = [ ] \\
\text{rev } (x:xs) = (\text{rev } xs) ++ [x]
\]
Quicksort Revisited

qsort a =  
  if null a then []  
  else qsort [x | x <- a, x<head a]  
      ++ [head a]  
      ++ qsort [x | x <- tail a, x >= (head a)]

qsort [] = []  
qsort (x:xs) =  
  let left = [a | a <- xs, a <= x]  
      right = [a | a <- xs, a > x]  
  in  qsort left ++ [x] ++ qsort right

def qsort(a):
  if a == []:  
    return []  
  else:
    pivot = a[0]  
    left = [x for x in a if x < pivot ]  
    right = [x for x in a[1:] if x >= pivot]  
    return qsort(left) + [pivot] + qsort(right)

def qsort(a):
  if a == []:  
    return []
  else:
    pivot = a[0]
    left = [x for x in a if x < pivot ]
    right = [x for x in a[1:] if x >= pivot]
    return qsort(left) + [pivot] + qsort(right)

public void sort(int low, int high)  
{  
  if (low >= high) return;
  int p = partition(low, high);
  sort(low, p);
  sort(p + 1, high);
}

void swap(int i, int j)  
{  
  int temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}

int partition(int low, int high)  
{  
  int pivot = a[low];
  int i = low - 1;
  int j = high + 1;
  while (i < j)  
  {  
    i++; while (a[i] < pivot) i++;
    j--; while (a[j] > pivot) j--;
    if (i < j) swap(i, j);
  }
  return j;
}

two reverses and tail recursion

\[
\begin{align*}
rev\ []} & = [] \\
rev\ (x:xs) & = rev\ xs + [x]
\end{align*}
\]

\[
\begin{align*}
revapp\ []\ b & = b \\
revapp\ (x:xs)\ b & = revapp\ xs\ (x:b)
\end{align*}
\]

\[
\begin{align*}
rev'\ a & = revapp\ a\ []
\end{align*}
\]

tail recursion! -- the recursive call appears at and only at the last step in a function call

optimized by compiler to be a loop