Week 2: Higher-Order Functions

CSC324 Principles of Programming Languages

David Liu, Department of Computer Science

Functions as abstraction
(+ 32 (* 1 (/ 9 5)))
(+ 32 (* 100 (/ 9 5)))
(+ 32 (* -2 (/ 9 5)))

(lambda (x)
  (+ 32 (* x (/ 9 5))))

(+ 32 (* 100 (/ 9 5)))
(- 32 (* 100 (/ 9 5)))
(* 32 (* 100 (/ 9 5)))

(lambda (x)
  (* x 32 (* 100 (/ 9 5))))
higher-order function (HOF)
a function $f$ that satisfies at least one of:

1. $f$ takes a function as an argument
2. $f$ returns a function
Three famous list HOFs

- Take a list of floats and round each one to two decimal places.
- Take a list of strings and strip trailing whitespace.
- Take a list of temperatures in Celsius and convert them to Farenheit.
- Take a list of HTML elements and extract their attributes.
new_list = []
for x in lst:
    new_item = f(x)
    new_list.append(new_item)
return new_list

# Python
new_list = map(f, lst)

; Racket
(define new_list (map f lst))

-- Haskell
new_list = map f lst
Lab 2!

For a nested list, you often need to recurse on every element of the list.

```scheme
(define (f nested-list)
  (if (not (list? nested-list))
      ...
      (... (map f nested-list) ...)))
```

- Take a list of floats and remove the ones less than 50.
- Take a list of strings and remove the ones that start with 'a'.
- Take a list of students and keep the ones in CSC324.
- Take a list of HTML elements and remove all but the <a> tags.
new_list = []
for x in lst:
    if pred(x):
        new_list.append(x)
return new_list

---

filter

# Python
new_list = filter(pred, lst)

; Racket
(define new_list (filter pred lst))

-- Haskell
new_list = filter pred lst
Generic list iteration (accumulator)

```python
acc = seed
for x in lst:
    acc = update(x, acc)
return acc
```

Generic list “iteration” (recursive)

```scheme
(define (foldl update acc lst)
  (if (null? lst)
      acc
      (foldl update
             (update (first lst) acc)
             (rest lst))))
```
Two more “classic” HOFs

Function composition

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

\[ f \cdot g = \lambda x \rightarrow f(g\ x) \]
Function application (Racket)

(apply f (list x1 x2 ... xn))
; ==
(f x1 x2 ... xn)

Function application (Haskell)

f $ x = f x

($) has the lowest precedence, so avoids parentheses:

count pred lst = length $ filter pred lst
Implementing dynamically-created functions

A naive implementation of functions

\x -> x + 10

Represent a function with a data structure storing:

- a parameter list ([ 'x' ]), and
- a function body (x + 10).
makeAdder \( n = \lambda x \rightarrow x + n \)

\( add1 = \) makeAdder 1
\( add2 = \) makeAdder 2
\( add3 = \) makeAdder 3
If the ... is very long, this is space-inefficient!

```
makeF n = \x -> ... x ... n ...

f1 = \x -> ... x ... 1 ...
f2 = \x -> ... x ... 2 ...
f3 = \x -> ... x ... 3 ...
```

**environment**

a mapping of identifiers to values
\[
\text{makeF } n = \lambda x \rightarrow \ldots x \ldots n \\
\]

\[
f_1 = \text{makeF } 1 \\
f_2 = \text{makeF } 2 \\
f_3 = \text{makeF } 3
\]

\[
f_1 = (\lambda x \rightarrow \ldots x \ldots n \ldots, \{n : 1\}) \\
f_2 = (\lambda x \rightarrow \ldots x \ldots n \ldots, \{n : 2\}) \\
f_3 = (\lambda x \rightarrow \ldots x \ldots n \ldots, \{n : 3\})
\]
**free identifier** (in a function)

an identifier in a function body that is neither a parameter nor bound locally (in a `let`)

In Racket and Haskell (and others), all function values are stored in a data structure called a **closure**:

- a **reference** to a function parameter list & body
- an **environment** storing bindings of all free identifiers in the function body
Python demo!

Closures and static scope
**static** (adjective)
determined only by the program source code

**dynamic** (adjective)
determined only when the program is run

---

**David’s Note**

We’ll cover the remaining slides in next week’s lecture.
E.g., referential transparency is a static property

**Implementing closures**

How are closures implemented? Are they “static” or “dynamic”? Or both?
Function bodies can be processed \textit{statically} (e.g., Haskell compiler generates code once per lambda).

The closure environment (and therefore the closure itself) can only be generated \textit{dynamically}.

But, the closure depends only on where the function is evaluated, \textit{not} on when that function is called.

So we can determine where each free identifier obtains its values \textit{statically}, based on where its function is defined, not where it is called.
**scope** (of an identifier)

The parts of program code that may refer to that identifier.

**static (aka lexical) scope**

The scope of every identifier is determined by the structure of the source code (e.g., by nesting of `lambda` and `let`).

Every identifier obtains its value from the closest enclosing expression that binds it.
David’s fun picture
Smile every day!