Functional Programming and Scheme

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

Higher-Order Functions

In Scheme, function is a first-class value

(Functions can go wherever expressions go)

- Functions as formal parameters

```
(define (twice f x) (f (f x)))
```

- Functions as real parameters

```
(twice sqrt 16)
```

- Functions as return values

```
(define (addN n) (lambda (m) (+ m n)))
```

Map

`map f l`: applies function `f` to each element of list `l`

```
(map (lambda (x) (+ x 1)) '(1 2 3))
```

```
(map sqrt '(4 9 16 25))
```

```
(map (addN 2) '(1 2 3))
```

Map

In previous lecture, we had:

```
(define (subs a b lst)
     (if (null? lst) '()
          (let ((rest (subs a b (cdr lst))))
               (if (equal? (car lst) a)
                    (cons  b          rest)
                    (cons (car lst) rest)))))
```

Using `map`, we have

```
(define (subs a b lst)
          (map (lambda (c)
                         (if (equal? a c) b c))
                         lst))
```

Map on Multiple Lists

`map f l1 l2 l3`: applies function `f` to elements at the same position of list `l1`,`l2`,`l3` (with same length)

```
(map + '(1 2 3) '(4 5 6) '(7 8 9))
```

```
(map (lambda (x y) (* x y)) '(1 2 3) '(4 5 6))
```

Filter

`filter f l`: return elements for which `f` returns `#t`

```
(filter (lambda (x) (> x 5)) ’(1 4 7 10))
```

```
(filter string? ’(1 “1” 2  “2” 3 “3”))
```

Fold-left

`fold-left f i` (e_1 e_2 e_n): **returns**

`f (f (… (f i` e_1`) …` e_{n-1}`)` e_n

```
(fold-left + 0 ’(1 2 3 4 5))
```

```
(fold-left (lambda (a x) (+ a (* x x)))
                0 '(1 2 3 4 5))
```

Fold-left

In previous lecture, we had:

```
(define (lstSum lst)
    (if (null? lst) 0
        ( + (car lst) (lstSum (cdr lst)))))
```

Using fold-left, we have

```
(define (lstSum lst) (fold-left + 0 lst)
```

Currying and Partial Evaluation

Multi-Argument Functions

```
(define (add m n) (+ m n))
```

add: Int, Int → Int

Two inputs

Pass multiple parameters as a list?

```
(add '(1 2))
```

✗

`add` takes 2 parameters

```
(apply add '(1 2))
```

✓

Multi-Argument Functions

```
(define (add m n) (+ m n))
```

add: Int, Int $\to$ Int

Add 2 to each element in a list?

`map f l`: applies function `f` to each element of list `l`

```
(map (add 2) '(1 2 3))
```

Multi-Argument Functions

```
(define (add m n) (+ m n))
```

add: Int, Int → Int

Add 2 to each element in a list?

`map f l`: applies function `f` to each element of list `l`

```
(map (add 2) ’(1 2 3))
```

?

Multi-Argument Functions

```
(define (add m n) (+ m n))
```

add: Int, Int → Int

Add 2 to each element in a list?

`map f l`: applies function `f` to each element of list `l`

```
(map (add 2) '(1 2 3))
```

`add` takes two parameters

Currying: Every function is treated as taking at most one parameter

```
(define (add m n) (+ m n))
```

add: Int, Int $\to$ Int

Curried version

```
(define (addN n) (lambda (m) (+ m n))
```

addN: Int $\to$ (Int $\to$ Int)

also written as: Int $\to$ Int $\to$ Int (right associative)

Uncurried version

```
(define (add m n)
        (+ m n))
```

add: Int, Int $\rightarrow$ Int

```
(add 2 3)
```

```
(map (add 2)
      '(1 2 3))
```

Curried version

```
(define (addN n)
   (lambda (m) (+ m n))
```

addN: Int $\rightarrow$ (Int $\rightarrow$ Int)

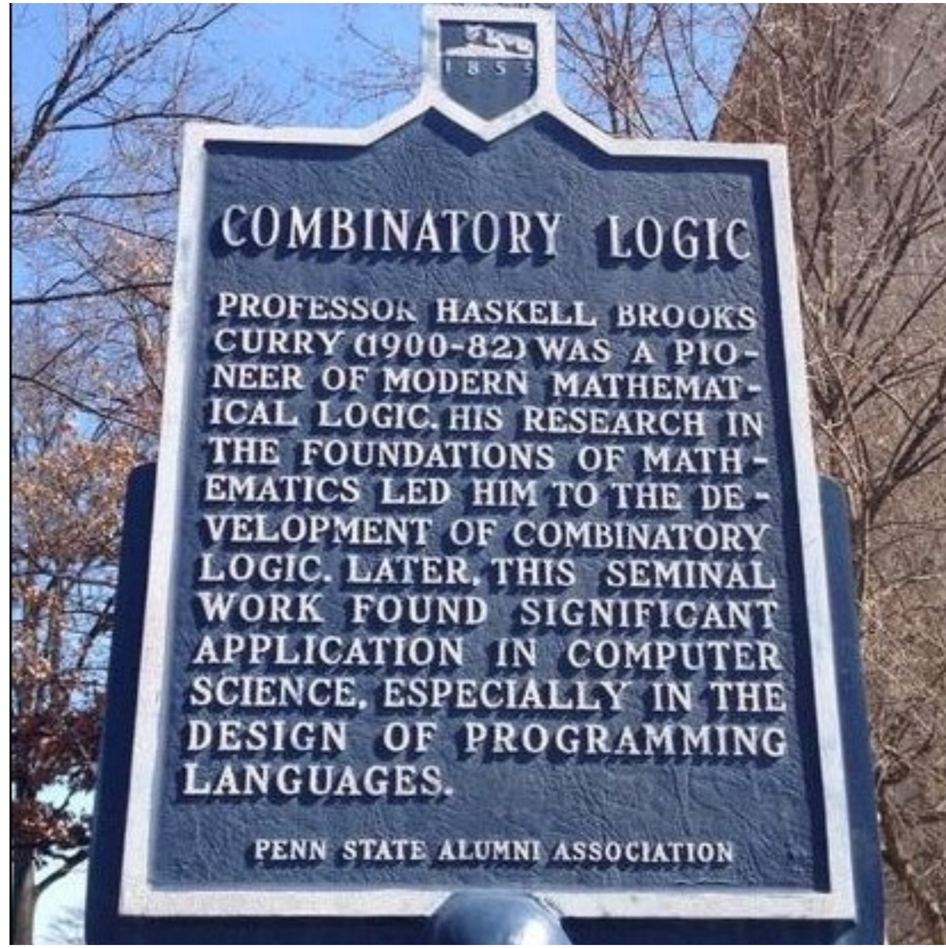
```
((addN 2) 3)
```

```
(map (addN 2)
      '(1 2 3))
```

Curried form allows partial application

Currying

Haskell B. Curry
Penn State 1929-1966



Outside of McAllister Building

Currying

In terms of lambda calculus,

the curried function of $\lambda x_1 x_2 \ldots x_n . e$ is
$\lambda x_1 . (\lambda x_{2.} \big(\ldots (\lambda x_n . e) \big))$

```
(define (curry2 f)
  (lambda (x)
    (lambda (y)
      (f x y))))
```

```
(define (curry3 f)
  (lambda (x)
    (lambda (y)
      (lambda (z)
        (f x y z)))))
```

Partial Evaluation

A function is evaluated with one or more of the leftmost actual parameters

`((curry2 add) 2)` is a partial evaluation of `add`

We can think it as a temporary result,

in the form of a function

Partial Evaluation

A function is evaluated with one or more of the leftmost actual parameters

```
(map ((curry2 add) 2) '(1 2 3))
```

$$(\text{curry2 add}) : \text{Int} \to (\text{Int} \to \text{Int})$$

$$((\text{curry2 add})\ 2) : \text{Int} \to \text{Int}$$

$$(\text{map}) : (\text{Int} \to \text{Int}) \to ([\text{Int}] \to [\text{Int}])$$

$$(\text{map}\ ((\text{curry2 add}) 2)) : ([\text{Int}] \to [\text{Int}])$$

$$(\text{map}\ ((\text{curry2 add}) 2))\ \text{'}(1\ 2\ 3) : [\text{Int}]$$

Uncurrying

In terms of lambda calculus,

the uncurried function of $\lambda x_1.(\lambda x_{2.}(\ldots(\lambda x_n.e)))$ is $\lambda x_1 x_2 \ldots x_n.e$

```
(define (uncurry2 f)
  (lambda (x y)
      ((f x) y)))
```

```
(define (uncurry3 f)
  (lambda (x y z)
        (((f x) y) z)))
```