Lazy Evaluation

Substitution Revisited

(define (f x) e1)

(f e2)

$\Longrightarrow$

Evaluate e1 after substituting e2 for all occurrences of x

Substitution Example

(define (inc x) (+ x 1))

(inc 5)

(+ 5 1)

Two Strategies

(define (inc x) (+ x 1))

(inc (+ 2 3))

Simplify, then substitute
(and simply some more…)

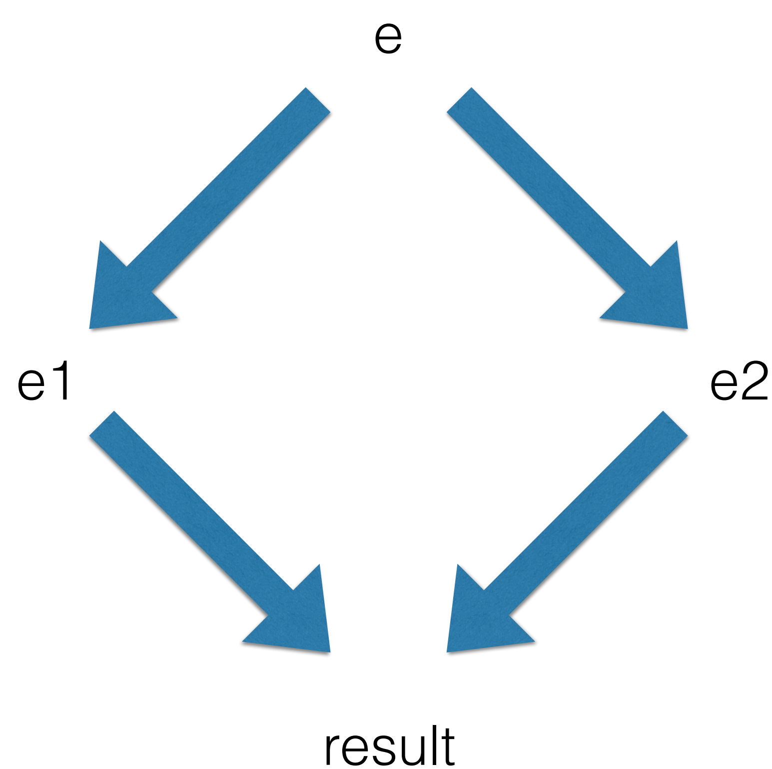
Substitute, then simplify

(inc 5)
(+ 5 1)
6

(+ (+ 2 3) 1)
(+ 5 1)
6

Does the Order Matter?

- Church-Rosser Theorem:
  - In the absence of side effects,
  - If both methods terminate,
  - The will terminate with the same answer



Applicative vs. Normal Order

Applicative Order ("Eager")

- Pass the <u>evaluated</u> actual parameter
- Evaluate the body of the function using that

Normal Order ("Lazy")

- Pass the <u>unevaluated</u> actual parameter
- Evaluate the body of the function using that
- If you ever use the formal parameter, then evaluate the actual parameter

Two Strategies (Revisited)

(define (inc x) (+ x 1))

(inc (+ 2 3))

Applicative Order
(Eager)

(inc 5)
(+ 5 1)
6

Normal Order
(Lazy)

(+ (+ 2 3) 1)
(+ 5 1)
6

“In The Absence of Side Effects…”

```
y = 0;

int f(int x)
{

    return x * 2;
}

f(y + 1);
```

```
y = 0;

int f(int x)
{
    y = 3;
    return x * 2;
}

f(y + 1);
```

“If They Both Terminate…”

```
int a = … ;

int f(int x, int y)
{
    return x > 0? y : 0;
}

f(a, 1 / 0);
```

Strategic Laziness

```
int a = … ;

int f(int x, int y)
{
    return x > 0? y : 0;
}

f(a, function_that_takes_two_hours());
```

But…

```
int a = … ;

int f(int x, int y)
{
    return x > 0? (y * y * y) : 0;
}

f(a, function_that_takes_two_hours());
```

Memoization

```
int a = … ;

int f(int x, int y)
{
    return x > 0? (y * y * y) : 0;
}

f(a, function_that_takes_two_hours());
```

Evaluate on first use

Re-use after that

Call by Name vs. Call by Need

- Lazy evaluation uses call-by-name parameter passing
  - In the absence of side-effects, and if it terminates,
    results are the same as call-by-value parameter passing
  - Can terminate even when eager evaluation would fail
  - Can avoid unnecessary computation but may do redundant computation
- Or more modern lazy languages use call-by-need
  - Memoization
  - Benefits of call-by-name but just as fast as call-by-value

Special Forms

Applicative Order ("Eager")

- Default behavior is eager
- Lazy special forms

Normal Order ("Lazy")

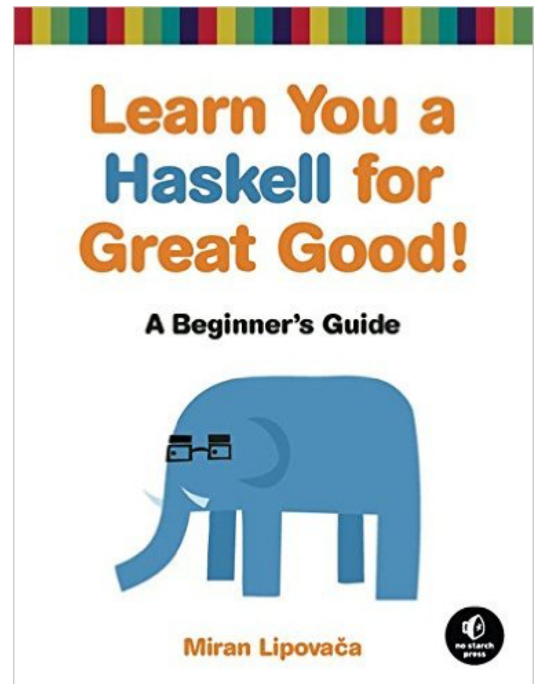
- Default behavior is lazy
- Eager special forms (i.e., "do it!")

Haskell

- Lazy evaluation (call-by-need parameter passing)
- Haskell is a pure functional language
  - No side effects (more on that later…)

Getting Started

- We will use the Glasgow Haskell Compiler
    - GHC = compiler
    - GHCI = interactive REPL
- Online tutorials at
    - http://book.realworldhaskell.org/read/
    - http://learnyouahaskell.com



Basics

- REPL interpreter
- Infix mathematical operations
- Pattern matching function definitions / calls
  (a bit like what we saw in Prolog)
- All the things you'd expect from a good functional language:
  - List handling
  - First-class and higher-order functions
  - Etc.

Let (local bindings)

```
let x = 3 in x + 1

let y = 4 + 9 in y * x
```

Modules

- Function definitions go in modules (files that are loaded)
- Load modules with ":load"
- Global definitions (x = …) allowed only in modules, not in the REPL

In file "Fac.hs":

```
fac 0 = 1
fac n = n * fac (n-1)
```

In REPL:

```
:load Fac.hs
fac 10
```

Lists

- Can be defined by just listing elements
- Can be defined by a range
- Can be defined by simple repeating interval over a range
- Can be defined using list comprehension
- Operations:
  - “head” - returns the first element
  - “tail” - returns a list of the rest
  - “take” - returns the first n things in a list
  - “takeWhile” - returns elements of the list as long as a supplied predicate is true
  - “:” - like “cons” in Lisp / Scheme / Racket
  - “++” - concatonation operator
  - “zip” - pairs elements of two lists and returns a list of two-element sublists

```
let x = [4, 8, 15, 16, 23, 42] in
  tail x
```

```
[1..10]
```

```
[1,3..11]
```

```
1 : [2..10]
```

```
let x = [1,3..11] in
  let y = [2,4..12] in
    zip x y
```

List Comprehension

- Can define lists using list comprehension
(like we saw in Erlang)
  - Filter based on criteria
  - Map to produce resulting list

```
myData = [1,2,3,4,5,6,7]

twiceData = [2 * x | x <- myData]
-- [2,4,6,8,10,12,14]

twiceEvenData =
   [2 * x| x <- myData,
          x `mod` 2 == 0]
```

Functions

- Can define functions based on pattern matching
- Can use multiple boolean cases

```
fac 0 = 1
fac n = n * fac (n-1)

dupfirst (x:xs) = x:x:xs

scoreToLetter n
  | n > 90 = 'A'
  | n > 80 = 'B'
  | n > 70 = 'C'
  | n > 60 = 'D'
  | otherwise = 'F'
```

First-Class and Higher-Order Functions

- Anonymous functions (lambda)
- Usual array of higher-order functions:
  - map
  - filter
  - foldl / foldr
  - etc

```
\x -> x + 1
```

```
let x = [1..10] in
  map (\x -> x + 1) x
```

Types

- In Haskell, everything has a type
  - Don't have to declare unless necessary to avoid ambiguity
  - Can declare if you want
  - Can use type variables

```
x :: Int

ones :: [Int]

scoreToLetter :: Int -> Char

filter :: (a -> Bool) -> [a] -> [a]
```

Let's Get Lazy

- All parameter passing is lazy (the formal parameter is bound to the <u>unevaluated</u> actual parameter)
- "let" is also lazy

Infinite Lists

- With lazy evaluation you can make conceptually infinite lists and other data structures
- Evaluated only as needed
- Use “head” or “take” to force evaluation and get elements
- Higher-order functions on infinite lists can produce other infinite lists

```
x = [1.. ]

ones = 1 : ones
```

What's This?

```
let x = 1 : 1 : zipWith (+) x (tail x) in
  take 5 x
```

Hint: zipWith (+) [1,3,4] [2,5,8]
returns [1+2, 3+5, 4+8]