CSC324 Principles of Programming Languages

Lecture 3

September 25, 2019

Exercise 2 Debrief

Do not modify the starter code line beginning with `(provide ...)`

Exercise 2 Debrief

Do not modify the starter code line beginning with `(provide ...)`

Common issue (for both tasks 1 and 2): missing recursive call

```
<expr> = NUM
       | (<op> <expr> <expr>)
       | (if (<comp> <expr> <expr>) <expr> <expr>)
```

In task 2, every time you see `<expr>`, there should be a recursive call!

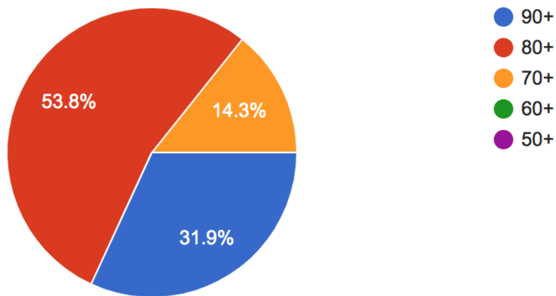
Exercise 2 value pattern matching

Possible helper function:

```
(define/match (op x)
  [('+) +]
  [('-) -]
  ...)
```

What grade are you aiming for in this course?

91 responses



Last class. . .

Functions of various kinds:

- Pattern matching
- Tail recursion
- Higher-order functions
- Currying

These are tool we use to build interesting programs in a functional language.

Today

Today is about **semantics**, evaluating code, and building an **interpreter**.

These are some of the choices we make about the semantics of a language:

- Strict and non-strict evaluation
- Closures & Environments
- Lexical & Dynamic Scoping

To be more specific, let's review the **lambda-calculus** before we begin.

Recall the lambda-calculus

1. Identifier; e.g. x
2. Function expression; e.g. $\lambda x \mapsto x$ (identity function)
3. Function application; e.g. $f\ expr$ (applies f to the expression $expr$)

Recall the lambda-calculus

1. Identifier; e.g. x
2. Function expression; e.g. $\lambda x \mapsto x$ (identity function)
3. Function application; e.g. f *expr* (applies f to the expression *expr*)

Lambda Calculus in Racket:

```
<expr> = ID
       | (lambda (ID) <expr>)
       | (<expr> <expr>)
```

Semantics of the lambda-calculus

- Identifiers and function expressions are already fully-evaluated
- Function applications are evaluated by substituting the argument for the parameter in the body of the function

Semantics of the lambda-calculus

- Identifiers and function expressions are already fully-evaluated
- Function applications are evaluated by substituting the argument for the parameter in the body of the function

But when do we evaluate the function expression and the argument expression?

Semantics of the lambda-calculus

- Identifiers and function expressions are already fully-evaluated
- Function applications are evaluated by substituting the argument for the parameter in the body of the function

But when do we evaluate the function expression and the argument expression?

```
<expr> = ID
       | (lambda (ID) <expr>)
       | (<expr> <expr>)
```

Strict and non-strict evaluation

Order of Operations

Consider `((lambda (x) (* x 2)) (+ 3 1))`

Order of Operations

Consider `((lambda (x) (* x 2)) (+ 3 1))`

One possible evaluation order:

- `((lambda (x) (* x 2)) 4)`
- `(* 4 2)`
- `8`

Order of Operations

Consider `((lambda (x) (* x 2)) (+ 3 1))`

One possible evaluation order:

- `((lambda (x) (* x 2)) 4)`
- `(* 4 2)`
- `8`

Another possible evaluation order:

- `(* (+ 3 1) 2)`
- `(* 4 2)`
- `8`

Order of Operations

Consider `((lambda (x) (* x 2)) (+ 3 1))`

One possible evaluation order:

- `((lambda (x) (* x 2)) 4)`
- `(* 4 2)`
- `8`

Another possible evaluation order:

- `(* (+ 3 1) 2)`
- `(* 4 2)`
- `8`

Q. Do we always get the same answer?

Church-Rosser Theorem (Informal)

For any valid program in the lambda calculus, every possible order of function application must result in the same final value.

Church-Rosser Theorem (Informal)

> *For any valid program in the lambda calculus, every possible order of function application must result in the same final value.*

But what about non-terminating programs? Or programs with errors?

Church-Rosser Theorem (Informal)

> *For any valid program in the lambda calculus, every possible order of function application must result in the same final value.*

But what about non-terminating programs? Or programs with errors?

Sidenote: we **can** have infinite loops in the lambda calculus!

Church-Rosser Theorem (Informal)

> *For any valid program in the lambda calculus, every possible order of function application must result in the same final value.*

But what about non-terminating programs? Or programs with errors?

Sidenote: we **can** have infinite loops in the lambda calculus!

Q. Should `(f #t (/ 1 0))` always fail for all `f`?

Left-to-right Eager Evaluation

When evaluating a function call

1. Evaluate function subexpression being called
2. Evaluate each argument subexpression, left-to-right
3. "Call" the function by substituting the *value* of each argument subexpression into the body of the function.

This is how Racket evaluates function calls.

Q. Which evaluation order is “eager”?

Choice A:

- `((lambda (x) (* x 2)) (+ 3 1))`
- `((lambda (x) (* x 2)) 4)`
- `(* 4 2)`
- `8`

Choice B:

- `((lambda (x) (* x 2)) (+ 3 1))`
- `(* (+ 3 1) 2)`
- `(* 4 2)`
- `8`

Q. Which evaluation order is "eager"?

Choice A:

- `((lambda (x) (* x 2)) (+ 3 1))`
- `((lambda (x) (* x 2)) 4)`
- `(* 4 2)`
- `8`

Choice B:

- `((lambda (x) (* x 2)) (+ 3 1))`
- `(* (+ 3 1) 2)`
- `(* 4 2)`
- `8`

Strict denotational semantics

If an argument expression is undefined (e.g. contains an error), the call expression is undefined.

Example: Try this in Racket!

```
(define (f x y) x)
(f 4 (/ 1 0))
```

Racket has strict denotational semantics (eager evaluation)

Examples of non-eager (non-strict) evaluation

Try entering this in a Racket shell:

```
(or #t (/ 1 0))
```

Examples of non-eager (non-strict) evaluation

Try entering this in a Racket shell:

```
(or #t (/ 1 0))
```

Why do we not see an error?

The identifier `or` does not refer to a function, but rather a *syntactic form* that implements **short-circuiting**.

We'll talk more about syntactic forms in the next 2 weeks.

What about Haskell?

Haskell uses *non-strict semantics* for function calls and name bindings.

When evaluating a function call

1. Evaluate function subexpression being called
2. ~~Evaluate each argument subexpression, left-to-right~~
3. “Call” the function by substituting the *unevaluated* argument subexpressions into the body of the function.

This strategy is called **lazy evaluation**.

Non-strict semantic

Try this in Haskell:

```
f x y = x
f 4 (1/0)
```

Lazy Evaluation in Haskell

Lazy evaluation lets us do cool things in Haskell, like define an infinite list!

List elements are not evaluated until they are needed.

```
> x = [1..10]
> take 5 x
[1,2,3,4,5]
> length x
10
```

Lazy Evaluation in Haskell

Lazy evaluation lets us do cool things in Haskell, like define an infinite list!

List elements are not evaluated until they are needed.

```
> x = [1..10]
> take 5 x
[1,2,3,4,5]
> length x
10
```

```
> y = [1..]
> take 20 y
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
> length y
```

The trouble with lazy evaluation

Recall the discussion on tail recursion, and the function `foldl`.

Here is how `foldl` is defined in Haskell

```
foldl _ acc [] = acc
foldl f acc (x:xs) =
  let acc' = f acc x
  in
      foldl f acc' xs
```

Why is this a problem?

The trouble with lazy evaluation

Recall the discussion on tail recursion, and the function `foldl`.

Here is how `foldl` is defined in Haskell

```
foldl _ acc [] = acc
foldl f acc (x:xs) =
  let acc' = f acc x
  in
      foldl f acc' xs
```

Why is this a problem?

The problem is that `acc'` is not evaluated *before* the recursive call on `f`.

`foldl f acc' xs` reduces to

- `foldl f (f acc x) xs`

Delaying evaluation in Racket

Q. What does this expression evaluate to?

```
(define x (length (range 3000)))
```

Delaying evaluation in Racket

Q. What does this expression evaluate to?

```
(define x (length (range 3000)))
```

Q. What about this?

```
(define (f x) (length (range 3000)))
```

Is `(range 3000)` evaluated when the function is defined?

Delaying evaluation in Racket

Q. What does this expression evaluate to?

```
(define x (length (range 3000)))
```

Q. What about this?

```
(define (f x) (length (range 3000)))
```

Is `(range 3000)` evaluated when the function is defined?

Q. What about this?

```
(define (g) (length (range 3000)))
```

Evaluation of Functions

Function bodies are not evaluated until the function is called!

```
(define (g) (length (range 3000)))
```

This function `g` is called a **thunk**: a nullary function that delays evaluation.

Closures and Environments

Interpreter 101

An **interpreter** executes instructions written in a programming language.

Your calculator application from exercises 1 and 2...

- took an expression as an argument
- returned a value

But what about variables?

Interpreter

More generally, an interpreter should take two arguments:

- the expression to be evaluated
- an **environment**

An **environment** is a collection of name-value bindings.

Example (from Exercise 3)

If we want to evaluate this expression:

```
(let* ((a 3))
   (+ a 1))
```

We'll need to store the environment `{a: 3}` and use it to evaluate `(+ a 1)`.

Example (from Exercise 4)

If we want to evaluate this expression:

```
((lambda (a) (+ a 1)) 3)
```

We also need to store the environment `{a: 3}` and use it to evaluate `(+ a 1)`.

Functions Saving Information

Recall that functions can return functions

```
(define (add-prefix lst1)
  (lambda (lst2) (append lst1 lst2)))

> (add-prefix '(1 2))
```

Functions Saving Information

Recall that functions can return functions

```
(define (add-prefix lst1)
  (lambda (lst2) (append lst1 lst2)))
```

```
> (add-prefix '(1 2))
```

```
#<procedure>
```

Functions Saving Information

Recall that functions can return functions

```
(define (add-prefix lst1)
  (lambda (lst2) (append lst1 lst2)))
```

```
> (add-prefix '(1 2))
```

```
#<procedure>
```

```
> ((add-prefix '(1 2)) '(3 4 5))
'(1 2 3 4 5)
```

What if we called add-prefix many times?

```
(define (add-prefix lst1)
  (lambda (lst2) (append lst1 lst2)))

(define prepend-1 (add-prefix '(1))
(define prepend-2 (add-prefix '(2))
(define prepend-3 (add-prefix '(3))

> (prepend-1 '(4 5 6))
'(1 4 5 6)
> (prepend-3 '(4 5 6))
'(3 4 5 6)
```

We need to store the value of `lst1` for each of `prepend-1` and `prepend-3`. (They need to be evaluated using different environments!)

Closures (Evaluating Function Expressions)

Function expressions should evaluate to **closures**.

A **closure** is a data structure that contains information about:

- the function body
- the environment at the time the function expression is evaluated (i.e. when the function was defined)

Closures (Evaluating Function Expressions)

Function expressions should evaluate to **closures**.

A **closure** is a data structure that contains information about:

- the function body
- the environment at the time the function expression is evaluated (i.e. when the function was defined)

Example: the closure `prepend-1` contains

- its definition (including its body): `(lambda (lst2) (append lst1 lst2))`
- its environment: `{lst1: '(1), ...}`

Evaluating Function Calls

When evaluating a function call `(<expr> <expr> ...)`

- Make sure that the function expression evaluates to a *closure*
- Evaluate its arguments (assuming a strict semantic, with left-to-right eager evaluation)
- Evaluate the function body with... *what environment?*

Evaluating Function Calls

When evaluating a function call `(<expr> <expr> ...)`

- Make sure that the function expression evaluates to a *closure*
- Evaluate its arguments (assuming a strict semantic, with left-to-right eager evaluation)
- Evaluate the function body with... *what environment?*

Choices:

- Environment at the time the function is defined (stored in the closure)
- Environment at the time the function is called

The different choices lead to different types of scoping.

Scoping

What should this evaluate to?

```
(define n 100)
(define (f a) n)
(define (g n) n)
(define (h n) (f 0))

> (f 10)
100
> (g 10)
10
> (h 10)
???
```

Scoping...

- **Lexical Scoping**: environment used is the one in scope *where the function is defined*.
- **Dynamic Scoping**: environment used is the one in scope *where the function is called* (during program execution)

Scoping...

- **Lexical Scoping**: environment used is the one in scope *where the function is defined*.
- **Dynamic Scoping**: environment used is the one in scope *where the function is called* (during program execution)

Q: Is Racket lexically scoped or dynamically scoped?

Scoping...

- **Lexical Scoping**: environment used is the one in scope *where the function is defined*.
- **Dynamic Scoping**: environment used is the one in scope *where the function is called* (during program execution)

Q: Is Racket lexically scoped or dynamically scoped?

A: Lexically scoped.

Haskell Scoping

Q: Is Haskell lexically scoped or dynamically scoped?

Haskell Scoping

Q: Is Haskell lexically scoped or dynamically scoped?

```
n = 100
f a = n
g n = n
h n = f 0

> f 10
100
> g 10
10
> h 10
???
```

Haskell Scoping

Q: Is Haskell lexically scoped or dynamically scoped?

```
n = 100
f a = n
g n = n
h n = f 0

> f 10
100
> g 10
10
> h 10
???
```

A: Lexically scoped.

Python Scoping

Q: Is Python lexically scoped or dynamically scoped?

Python Scoping

Q: Is Python lexically scoped or dynamically scoped?

A: Lexically scoped.

Bash Scoping

```
X="batman"

function printX {
  echo $X
}

function localX {
  local X="superman"
  printX
}

printX
localX
```

Python Scoping Bug

You will run into this bug at some point in your career:

```
adders = []
for i in [1, 2, 3]:
    add_i = lambda x: x + i
    adders.append(add_i)

add1 = adders[0]
print(add1(5))
```

What happened?

Python Scoping Bug

You will run into this bug at some point in your career:

```
adders = []
for i in [1, 2, 3]:
    add_i = lambda x: x + i
    adders.append(add_i)

add1 = adders[0]
print(add1(5))
```

What happened?

Mutation! The value of `i` has changed. We don't have referential transparency here.

Summary

- Order of evaluation in a function application
  - Strict and non-strict evaluation
- Identifier binding / name lookups
  - Closures & Environments
  - Lexical & Dynamic Scoping

When you build interpreters, remember the left-to-right eager evaluation order, and use lexical scoping!