CSC324 Principles of Programming Languages

Lecture 3

September 28/29, 2020

Announcement

- Ex2 unit tests are released on Markus
- Test 1 is this week!
- Ex3 is due this Saturday

Exercise 2 Debrief

Very well done! Common issue: style and pattern matching

Exercise 2 Debrief

Very well done! Common issue: style and pattern matching

- Most of the time, you'll have more freedom to choose the programming style. This time, we asked for pattern matching.

Exercise 2 Debrief

Very well done! Common issue: style and pattern matching

- Most of the time, you'll have more freedom to choose the programming style. This time, we asked for pattern matching.
- The recursive structure of your function should follow the grammar. Are you missing a base case? Unnecessarily checking if the inner `<expr>` is a number?

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

Exercise 2 Debrief

Very well done! Common issue: style and pattern matching

- Most of the time, you'll have more freedom to choose the programming style. This time, we asked for pattern matching.
- The recursive structure of your function should follow the grammar. Are you missing a base case? Unnecessarily checking if the inner `<expr>` is a number?

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

- Pattern matching: **order matters**; no need to pattern match on numbers if you handle all the other cases first!

```
(define/match (calculate x)
  [((list op a b)) ...]
  [(num)           num])
```

Last minute test 1 reminders

- Start the test between 5 and 15 min past the hour
- The test is 30 minutes long; manage your time well!
- **Shut down email/BbCollaborate/Discord and other means of communication**
- If you're disconnected or have technical issues with Quercus, let Lisa know (you can use email in this case)
  - Document time the issue started/resolved
  - Take a picture of the Quercus quiz not loading, with the time

Last minute test 1 reminders

- Start the test between 5 and 15 min past the hour
- The test is 30 minutes long; manage your time well!
- **Shut down email/BbCollaborate/Discord and other means of communication**
- If you're disconnected or have technical issues with Quercus, let Lisa know (you can use email in this case)
  - Document time the issue started/resolved
  - Take a picture of the Quercus quiz not loading, with the time
- Do not discuss the test until after 8pm

Overview

Today is about **semantics**, evaluating code, and towards 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

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

Example:

```
((lambda (x) y) ((lambda (x) x) z))
```

But which application do we perform first?

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`

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?

```
; we can have non-terminating lambda-calculus programs!
((lambda (x) (x x)) (lambda (x) (x x)))
```

In general, evaluation orders matter.

Evaluation Order

Two ways to think about evaluation order

- Eager vs. lazy evaluation
  - Order of evaluation
- Strict vs. non-strict semantics
  - Error propagation

Left-to-right Eager Evaluation

When evaluating a function call

1. Evaluate function subexpression
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

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

Strict denotational semantics

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

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

```
; Racket example:
(define (f x y) x)
(f 4 (/ 1 0))
```

Racket functions has strict denotational semantics

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 after the reading week.

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:

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

g z = g z  -- infinite loop
f 4 (g 1)
```

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.

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

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

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

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 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)

A function expression evaluates to a **closure**.

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)

A function expression evaluates to a **closure**.

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), ...}`

Example

Explain, step by step, how this Racket expression is evaluated

```
((lambda (y) 3) 4)
```

Example

Explain, step by step, how this Racket expression is evaluated

```
((lambda (y) 3) 4)
```

1. Evaluate `(lambda (y) 3)`, get a closure with the params, body `3`, environment

Example

Explain, step by step, how this Racket expression is evaluated

```
((lambda (y) 3) 4)
```

1. Evaluate `(lambda (y) 3)`, get a closure with the params, body `3`, environment
2. Evaluate the argument `4`, which evaluates to 4

Example

Explain, step by step, how this Racket expression is evaluated

```
((lambda (y) 3) 4)
```

1. Evaluate `(lambda (y) 3)`, get a closure with the params, body `3`, environment
2. Evaluate the argument `4`, which evaluates to `4`
3. To evaluate the full function call, evaluate the body expression `3` with the additional binding `{y:4}` in the environment. This expression evaluates to . . . 3!

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 (Demo)

```
X="batman"

function printX {
  echo $X
}

function localX {
  local X="superman"
  printX
}

printX
localX
```

Python Scoping Bug (Demo)

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 (Demo)

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.

Fixing the Python Scoping Bug

We need to create a new scope (a new version of `i`)

```
def make_adder(i):
  return lambda x: x + i

adders = []
for i in [1, 2, 3]:
  add_i = make_adder(i)
  adders.append(add_i)

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

Breakout Group

Explain, step by step, how this Racket expression is evaluated

```
(((lambda (x)
    (lambda (y) (* x y)))
  (+ 2 3))
 10)
```

Step 1 (function call)

The full expression is a function call.

First thing that gets evaluated is the **function expression**

```
(((lambda (x)                ;
    (lambda (y) (* x y)))    ;
  (+ 2 3))                   ;
 10)
```

Step 2 (inner function call)

This expression is a function call, so we need to evaluate *its* function expression first

```
((lambda (x)                ;
    (lambda (y) (* x y))) ;
  (+ 2 3))
```

Step 3 (fn expression in inner call)

This expression is a function expression, which evaluates to a closure. We'll invent a notation to encode the closure

```
(lambda (x)
  (lambda (y) (* x y)))

==> #(closure
      (x) ; param name
      (lambda (y) (* x y)) ; body
      {}  ; environment)
```

Step 4 (argument in inner call)

The function body from step 2 is evaluated. We can keep evaluating the argument expressions.

```
((lambda (x)                ; DONE
    (lambda (y) (* x y))) ;
  (+ 2 3)) ; TODO
```

To evaluate `(+ 2 3)`, we evaluate the + (identifier lookup), 2 and 3, and then perform the computation encoded by +. We get the result 5

Step 5 (actual inner call)

Now that we have both the function expression:

```
#(closure (x) (lambda (y) (* x y)) {})
```

and the argument 5 evaluated, we evaluate the function body of the closure, binding the parameter to the argument value. In other words, evaluate

```
(lambda (y) (* x y)) ; with environment {x: 5}
```

This expression evaluates to a closure

```
#(closure (y) (* x y) {x: 5})
```

Step 6 (argument in outer call)

The function body from step 1 is evaluated. We can keep evaluating the argument expressions.

```
(((lambda (x)                ; DONE
    (lambda (y) (* x y)))   ;
  (+ 2 3))                   ;
 10) ; TODO
```

The argument expression 10 evaluates to the number 10.

Step 7 (argument in outer call)

Now we can evaluate the full expression! We want to apply

```
#(closure (y) (* x y) {x: 5})
```

...with the argument 10. So we evaluate the body of the closure

```
(* x y) ; with envrionment {x: 5, y: 10}
```

This involves evaluating:

- `*`, an identifier that evaluates to the multiplication builtin
- `x`, an identifier that evaluates to 5
- `y`, an identifier that evaluates to 10
- the actual multiplication, which evaluates to 50

What to do next

1. Complete week 3 quiz; ask questions on Piazza
2. Test 1 this week!
3. Exercise 3 due Saturday 10pm
4. **You also have everything you need to complete exercise 4 task 1**
5. Read week 4 notes and attempt quiz