

CSC324: Principles of Programming Languages

Lecture 7

Friday 29 May 2020

A word about the quiz

- Written **during the lab session** on Monday, 8 June, written online
- Still figuring out how people can ask questions of the teaching staff during the quiz; stay tuned.
 - If you are in another class and used something that worked well, let me know and I can look into it.)

A word about the quiz

- Format: short/long answer, as you would expect from a paper midterm in a "normal" class.
- **Open** interpreter & language documentation, open course notes
- **Closed** googling/stack overflow/collaboration w/ someone/anything else
- We will trust, but we will verify
 - If trust is broken for Quiz 1, Quiz 2 will require invasive anti-cheating setups that nobody except the shareholders of ProctorU Inc. will want.

Last time...

- We introduced the semantics of **eager** and **lazy evaluation**
- We saw how Racket's evaluation semantics are eager
 - but lazy evaluation can be mimiced by using nullary functions called **thunks**
- We saw how Haskell's evaluation semantics are lazy
 - but eager evaluation can be forced using the `seq` built-in function

Today:

- Scoping: lexical and dynamic
- Practicalities of implementing closures

Free identifiers and Closures



```
(define x 3)
(define a-thunk (λ () (+ x 1)))
```

Recall: expressions may reference an identifier defined outside its local scope.

Definition: A **free identifier** is an identifier within a function body that:

- Is not a parameter to the function
- Is not bound in a local let-expression

Free identifiers and Closures



```
(define x 3)
(define a-thunk (lambda () (+ x 1)))
```

Up to this point, we haven't talked precisely about how our substitution model of function application behaves with free identifiers.

The semantics of a language w.r.t. free identifiers is called its **scoping rules**.

Free identifiers and Closures



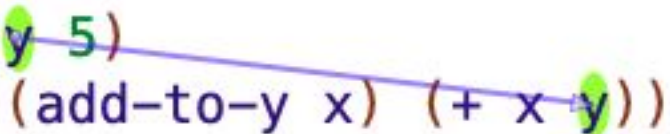
```
(define x 3)
(define a-thunk (lambda () (+ x 1)))
```

Definition: An identifier's **scope** is the "area" of the program that the identifier is visible and can be referenced in.

Lexical (static) scoping

Definition: A body of a function is said to be **lexically scoped** if, when evaluated, its free variables remain bound to their bindings **when the function was created**, not when the function was called.

```
(define y 5)
(define (add-to-y x) (+ x y))
```



```
> (add-to-y 13)
18
> (let ([y 31337])
    (add-to-y 13))
18
>
```

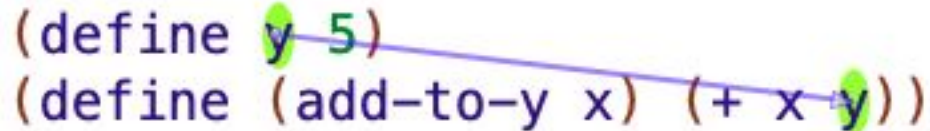
Lexical (static) scoping

👍 : Note that lexical scoping preserves referential transparency; irrespective of what `y` is locally-bound to, `(add-to-y 13)` produces the same value.

👍 : Lexical scoping lets us implement closures

👎 : The substitution evaluation model needs some refinement for lexical scoping (we'll talk about this later!)

```
(define y 5)
(define (add-to-y x) (+ x y))
```



```
> (let ([y 31337])
    (add-to-y 13))
18
> (let ([y -99])
    (add-to-y 13))
18
> (let ([y "not even a number!"])
    (add-to-y 13))
18
>
```

Dynamic scoping

Definition: A body of a function is said to be **dynamically scoped** if, when evaluated, its free variables are resolved to values in the **dynamic (runtime) context that the function is called in.**

```
; A hypothetical language and  
; not Racket!  
(define y 5)  
(define (add-to-y x) (+ x y))
```

Dynamic scoping

Evaluating a dynamically-scoped expression: We look for the correct binding for `y` by beginning in the innermost scope, and working outwards until we find one.

[demo: Example in GNU Common Lisp:]

```
; A hypothetical language and  
; not Racket!  
(define y 5)  
(define (add-to-y x) (+ x y))  
  
(add-to-y 13)
```

Dynamic scoping

Evaluating a dynamically-scoped expression: We look for the correct binding for `y` by beginning in the innermost scope, and working outwards until we find one.

If we evaluate `(add-to-y 13)` here...

```
; A hypothetical language and  
; not Racket!  
(define y 5)  
(define (add-to-y x) (+ x y))  
  
(add-to-y 13)
```

Dynamic scoping

Evaluating a dynamically-scoped expression: We look for the correct binding for `y` by beginning in the innermost scope, and working outwards until we find one.

We perform function application and find this expression, where the `"y"` in question refers to the top-level one

```
; A hypothetical language and  
; not Racket!  
(define y 5)  
(define (add-to-y x) (+ x y))  
  
(+ 13 y)
```

Dynamic scoping

Evaluating a dynamically-scoped expression: We look for the correct binding for `y` by beginning in the innermost scope, and working outwards until we find one.

We substitute the `y` we have in scope with its value...

```
; A hypothetical language and  
; not Racket!  
(define y 5)  
(define (add-to-y x) (+ x y))  
  
(+ 13 5)
```

Dynamic scoping

Evaluating a dynamically-scoped expression: We look for the correct binding for `y` by beginning in the innermost scope, and working outwards until we find one.

...and evaluate the expression.

```
; A hypothetical language and  
; not Racket!
```

```
(define y 5)  
(define (add-to-y x) (+ x y))
```

18

Dynamic scoping

Evaluating a dynamically-scoped expression: We look for the correct binding for `y` by beginning in the innermost scope, and working outwards until we find one.

Now let's try another example.
Remember that in lexical scoping, this produced the same value.

```
; A hypothetical language and  
; not Racket!  
(define y 5)  
(define (add-to-y x) (+ x y))  
  
(let ([y -99])  
  (add-to-y 13))
```

Dynamic scoping

Evaluating a dynamically-scoped expression: We look for the correct binding for `y` by beginning in the innermost scope, and working outwards until we find one.

We perform function application, and find that the `y` value of `-99` is set in a closer scope than the top-level definition...

```
; A hypothetical language and  
; not Racket!  
(define y 5)  
(define (add-to-y x) (+ x y))  
  
(let ([y -99])  
  (+ 13 y))
```

Dynamic scoping

Evaluating a dynamically-scoped expression: We look for the correct binding for `y` by beginning in the innermost scope, and working outwards until we find one.

So we substitute `y` with `-99` instead...

```
; A hypothetical language and  
; not Racket!  
(define y 5)  
(define (add-to-y x) (+ x y))  
  
(let ([y -99])  
  (+ 13 -99))
```

Dynamic scoping

Evaluating a dynamically-scoped expression: We look for the correct binding for `y` by beginning in the innermost scope, and working outwards until we find one.

And end up producing a different expression altogether.

```
; A hypothetical language and  
; not Racket!  
(define y 5)  
(define (add-to-y x) (+ x y))  
  
(let ([y -99])  
  -86)
```

Dynamic scoping

Evaluating a dynamically-scoped expression: We look for the correct binding for `y` by beginning in the innermost scope, and working outwards until we find one.

```
; A hypothetical language and  
; not Racket!
```

```
(define y 5)  
(define (add-to-y x) (+ x y))
```

-86

Dynamic scoping

👍 : Dynamic scoping falls out of the substitution model we've been discussing so far

👊 : Hard to reason about: the behaviour of the function depends on where you call it from! (Referential transparency is broken! Closures may or may not work, depending on what else is in scope at the time!)

```
; A hypothetical language and  
; not Racket!  
(define y 5)  
(define (add-to-y x) (+ x y))
```

```
> (add-to-x 13)  
18  
> (let ([y -99])  
    (add-to-x 13))  
-86
```

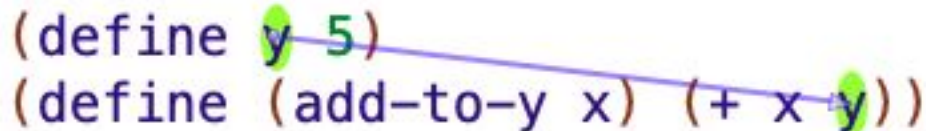
Practicalities of implementing closures

What does a language runtime need in order to implement the functionality of closures?

Depends on our evaluation model!

Implementing lexically-scoped closures

We saw how lexical scoping **binds** values that an expression closes over.



```
(define y 5)
(define (add-to-y x) (+ x y))
```

Therefore, the under-the-hood implementation of add-to-y must have:

- The function to run when the closure is called
- A mapping of free identifiers to their values

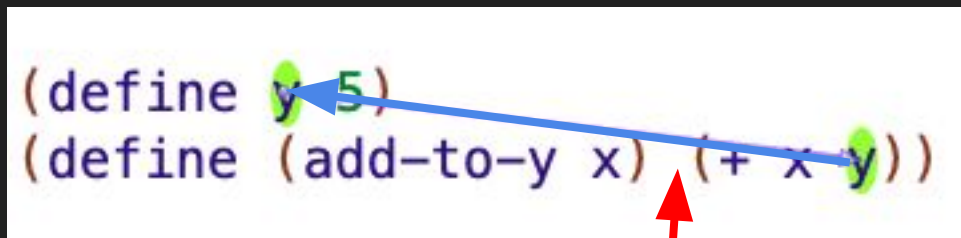
```
> (add-to-y 13)
18
> (let ([y 31337])
    (add-to-y 13))
18
>
```


Implementing lexically-scoped closures

We saw how lexical scoping **binds** values that an expression closes over.

Therefore, the under-the-hood implementation of add-to-y must have:

- The function to run when the closure is called
- A mapping of free identifiers to their values



The diagram shows two lines of code: `(define y 5)` and `(define (add-to-y x) (+ x y))`. The identifier `y` in the first line is highlighted in green. The identifier `y` in the second line is also highlighted in green. A blue arrow points from the green `y` in the second line to the green `y` in the first line. A red arrow points from the text below to the green `y` in the second line.

```
(define y 5)
(define (add-to-y x) (+ x y))
```

*"environment pointer": points from the **use** of an identifier to its **definition***

Implementing lexically-scoped closures

We saw how lexical scoping **binds** values that an expression closes over.

Therefore, the under-the-hood implementation of add-to-y must have:

- The function to run when the closure is called
- A mapping of free identifiers to their values

Environment
x: ???
y: 5
+: <procedure>
...

Closure
code

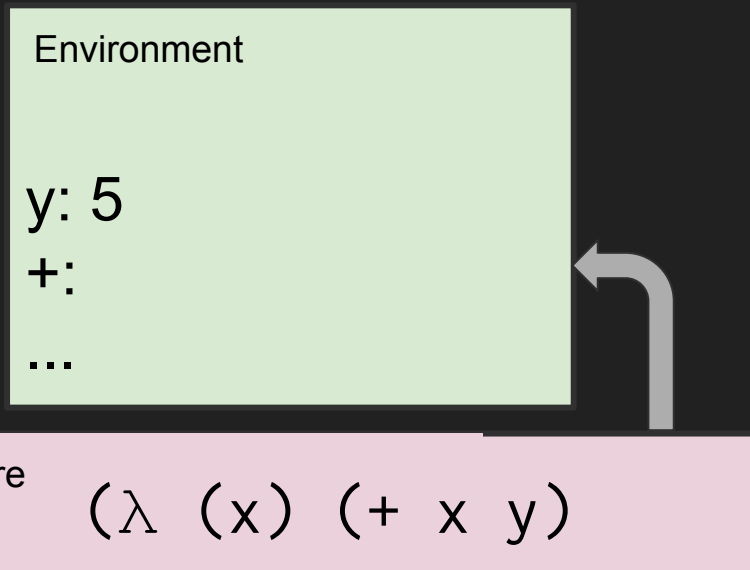
(λ (x) (+ x y))



Implementing lexically-scoped closures

We saw how lexical scoping **binds** values that an expression closes over.

Therefore, we say that a function call expression **evaluates the code of the closure** in the **environment of the closure**



Where is x ? It's not a free identifier, so not in the environment!

Implementing lexically-scoped closures

```
#lang racket
```

```
; top-level bindings: + and y  
; are defined and bound here  
; (define (+ x y) ...)  
(define y 5)
```

```
(define (add-to-y x) ; closure environment: the  
  (+ x y))          ; top-level y is bound here
```

```
(let [(y -99)]  
  (add-to-y 1))
```

closure environment

+: <procedure>

y: 5

function environment

x: 1

Closure
code

$(\lambda (x) (+ x y))$

Implementing dynamically-scoped closures

```
; A hypothetical language and  
; not Racket!  
(define y 5)  
(define (add-to-y x) (+ x y))  
  
(let ([y -99])  
  (add-to-y 13))
```

