CS450

Structure of Higher Level Languages

Lecture 18: Lexical/dynamic scoping

Tiago Cogumbreiro

Today we will learn...

- Lexical scoping
- Dynamic scoping
- Function closures
- Compute which variables are captured by a function declaration

> Acknowledgment: Today's lecture is inspired by Professor Dan Grossman's wonderful lecture in CSE341 from the University of Washington: Video 1 Video 2 Video 3 Video 4

Lexical Scope

Lexical Scope

- ***Binding:*** association between a variable and a value.
- ***Scope*** of a binding: the text where occurrences of this name refer to the binding
- ***Lexical (or static) scope***: the innermost lexically-enclosing construct declaring that variable

> **Did you know?** In Computer Science, ***static analysis*** corresponds to analyzing the source code, without running the program.

```
(define (f)
  (define x 10) ; visible: f
  (define y 20) ; visible: f, f.x
  (+ x y))      ; visible: f, f.x, f.y

; visible: f
(define x 1)
; visible: f, x
(define y (+ x 1))
; visible: f, x, y
(check-equal? (f) 30) ; yields (+ f.x f.y)
```

Dynamic Scope

Lexical scope vs dynamic scope

- Lexical scoping is the default in all popular programming languages
- With lexical scoping, we can analyze the source code to identify the scope of ***every*** variable
- With lexical scoping, the programmer can reason about each function independently

What is a dynamic scope?

- Variable scope depends on the calling context
- Renders all variables global

> appeared in McCarthy's Lisp 1.0 as a bug and became a feature in all later implementations, such as MacLisp, Gnu Emacs Lisp.

Moreau, L. Higher-Order and Symbolic Computation (1998) 11: 233. DOI:10.1023/A:1010087314987

```
;; NOT VALID RACKET CODE!!!
(define (f) x)

(define (g x) (f))
(check-equal? (g 10) 10)

(define x 20)
(check-equal? (f) 20)
```

Example

- What is the result of evaluating (g)?

```
(define x 1)

(define (f y) (+ y x))

(define (g)
  (define x 2)
  (define y 3)
  (f (+ x y)))

(check-equal? (g) ???)
```

Example

- What is the result of evaluating (g)?

```
(define x 1)

(define (f f:y) (+ f:y x))

(define (g)
  (define g:x 2)
  (define g:y 3)
  (f (+ g:x g:y)))

(check-equal? (g) 6)
```

Why lexical scoping?

- Lexical scoping is important for using functions-as-values
- To implement our Mini-Racket we will need to implement lexical scoping

Example

- What is the result of evaluating (g)?

```
(define (g) x)

(define x 10)

(check-equal? (g) ??)
```

Example

- What is the result of evaluating `(g)`?

```racket
(define (g) x)
; (g) throws an error here
(define x 10)

(check-equal? (g) 10)
```

We can define a function `g` that refers to an undefined variable `x`; variable `x` must be defined before calling `g`.
In Racket, variable definition produces a side-effect, as the definition of `x` impacted a previously defined function `g`. ***In Module 5, we implement the semantics of `define`.***

Accessing variables outside a function

The body of a function can refer to variables defined outside of that function.

> It can access variables is defined outside of the function, but where exactly?

The function's body can access any variable that is accessible/visible when the function is **defined**, which is known as the **lexical scope**.

In the following example, the function returns 3 and not 10, even though variable x is now 10.

```
; For a given x create a new function that always returns x
(define (getter x) (lambda () x))
(define get3 (getter 3)) ; At creation time, x = 3
(define x 10)
(check-equal? 3 (get3)) ; At call time, x = 10
```

Function closures

Recall that functions capture variables

Function closure

- **A function closure is the return value of function declaration** (***i.e.***, the function ***value***)
- **Definition:** A function closure is a pair that stores a function declaration and its lexical environment (***i.e.***, the state of each variable captured by the function declaration)
- The technique of creating a function closure is used by compilers/interpreters to represent function values

Recall that function declarion $\neq$ function definition:

- Function declaration: `( lambda ( variable* ) term+)`
- Function definition: `( define (variable+ ) term+ )`

Now we know what a function closure is

1. How to compute the variables in a closure
2. When to set the values of each variable in a closure

Function closures: captured variables

> ***It is crucial for us to know how variables are captured in Racket.***

Given an expression the set of free variables can be defined inductively:

- When the expression is a variable `x`, the set of free variables is `{ x }`.
- When the expression is a `(lambda (x) e)`, the set of free variables is that of expression `e` minus variable `x`.
- When the expression is a function application `(e1 e2)`, the set of free variables is the union of the set of free variables of `e1` and the set of free variables of `e2`.

Captured variables: Given an expression `(lambda (x) e)` a function closure ***captures*** the set of free variables of expression `(lambda (x) e)`.

Captured variables examples

Let us compute fv `(lambda (x) (+ x y))`:

1. The free-variables of a λ are the free variables of the body of the function minus parameter x.

fv `(lambda (x) (+ x y))` = fv `(+ x y)` $\setminus \{x\}$

2. We are now in a case of function application, which is the union of the free variables of each of its sub expressions.

fv `(+ x y)` $\setminus \{x\} = \left(\mathrm{fv}(+) \cup \mathrm{fv}(x) \cup \mathrm{fv}(y)\right) \setminus \{x\}$

4. Finally, we reach the case where each argument of `free-vars` is a variables.

$$\left(\mathrm{fv}(+) \cup \mathrm{fv}(x) \cup \mathrm{fv}(y)\right) \setminus \{x\} = \left(\{+\} \cup \{x\} \cup \{y\}\right) \setminus \{x\} = \{+, x, y\} \setminus \{x\} = \{+, y\}$$

What creates an environment?

Definition: At any execution point there is an environment, which maps each variable to a value.

What creates environments:

- Each branch inside a `cond` creates an environment
- The body of a function creates an environment

What updates an environment:

- The arguments of a `lambda` are added to the function's body environment
- A `(define x e)` updates the current environment by adding/updating variable `x` and setting it to the value that results from evaluating `e`

Example 1: capture an argument

> The `lambda` is capturing `x` as the parameter of `getter` at creation time, so when we call `(getter3)` we get `(lambda () 3)`.

```
(define (getter x)
  (lambda () x)) ; getter:x

(define get3 (getter 3)) ; getter:x = 3; (lambda () getter:x)
(check-equal? 3 (get3))
```

Example 3: `cond` starts a new scope

> Function `getter` captured `x` at the outermost scope (the `x` captured at function declaration time). Inside the branches of `cond` we have ***a new scope***, which means that `getter` is unaffected by the redefinition of `x`.

```
(define (getter) x) ; root.x
(define x 10)        ; root.x = 10
; Each branch of the cond creates a new environment
; so it does not affect getter
(cond [#t (define x 20) (check-equal? 10 (getter))])
(check-equal? 10 (getter))
```

Example 3: define shadows parameters

> Function `getter` returns variable `x` from the environment of function `f`. When calling `f 20` the last value of variable `x` in the scope of `f` is `10`, due to `(define x 10)`, which overwrites the function's parameter `x=20`.

```
(define (f x)
  (define (getter) x) ; f.x = ?
  (define x 10)       ; f.x = 10
  getter)

(define g (f 20))
(check-equal? 10 (g))
```