Closures and Closure-Creating Interpreters

CIS400 (Compiler Construction)

Kris Micinski, Fall 2021

In this lecture, we start looking at how to compile λ

Closures, conceptually, are very important to understand. We will need to build closures (in assembly/LLVM) to implement our language.

λ forms the core of every functional programming language

Even if you care about non-functional programming reasons, λs are generally-powerful abstractions, so learning to compile them will help you understand (e.g.,) objects, etc…

At runtime, a lambda cannot simply be represented by a piece of code.

Consider the following example:

```
(define f (lambda (x) (lambda (y) x)))
(define x (f 1))
(define y (f 2))
```

Let's say that we return just **this** lambda as the result from applying f

At runtime, a lambda cannot simply be represented by a piece of code.

Consider the following example:

```
(define f (lambda (x) (lambda (y) x)))
(define x (f 1))
(define y (f 2))
```

Then, when execution gets **here**, how do we get **x**!?

One (bad) solution: substitution

We could **substitute** for x and return a new **copy** of the lambda

```
(define f (lambda (x) (lambda (y) x)))
(f 1)
```

E.g., we could create a new copy

```
(lambda (y) 1)
```

E.g., we could create a new copy

```
(lambda (y) 1)
```

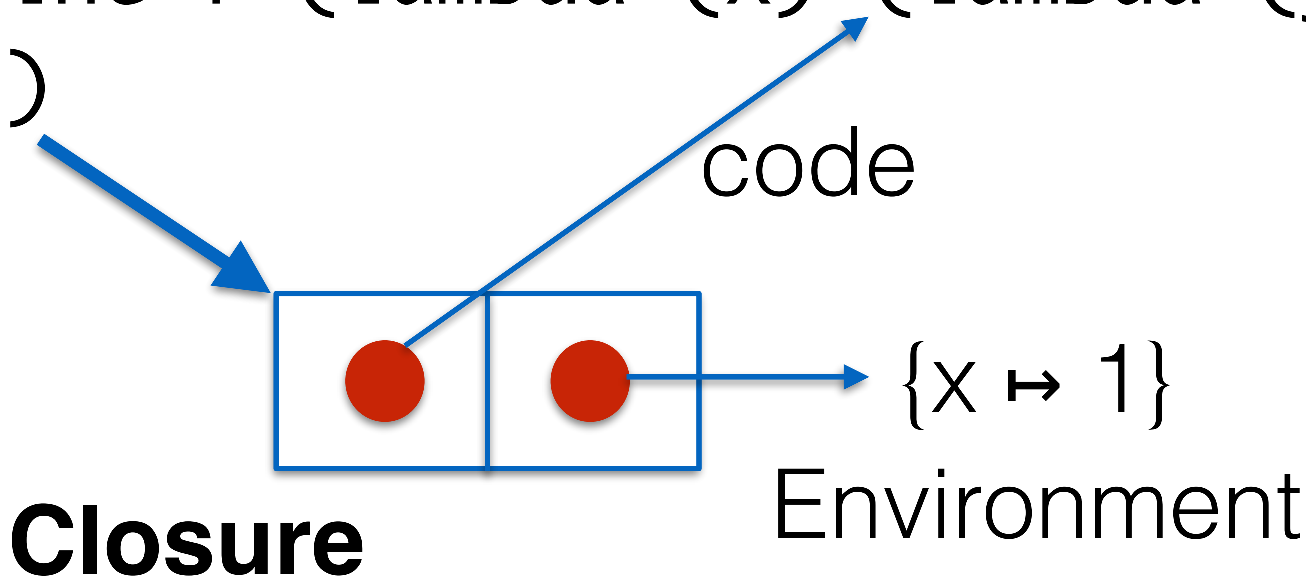
This is a bad solution for several reasons:

- Substitution is very costly
  - Every substitution is, in general, O(n), where n lambda's depth, paid at **each application**
  - Since (in functional languages) almost everything is a function application, this is prohibitively slow
- Second, substitution would create lots of copies of code (mostly unchanged) that we would store
  - E.g., on the heap via dynamic allocations
  - Large allocations (and particularly copying) are slow

Closures

- Instead, modern languages represent lambdas (at runtime) as **closures**
  - A **pair** of the **lambda** and an **environment** (dictionary mapping variables to their values)

```
(define f (lambda (x) (lambda (y) x)))
(f 1)
```

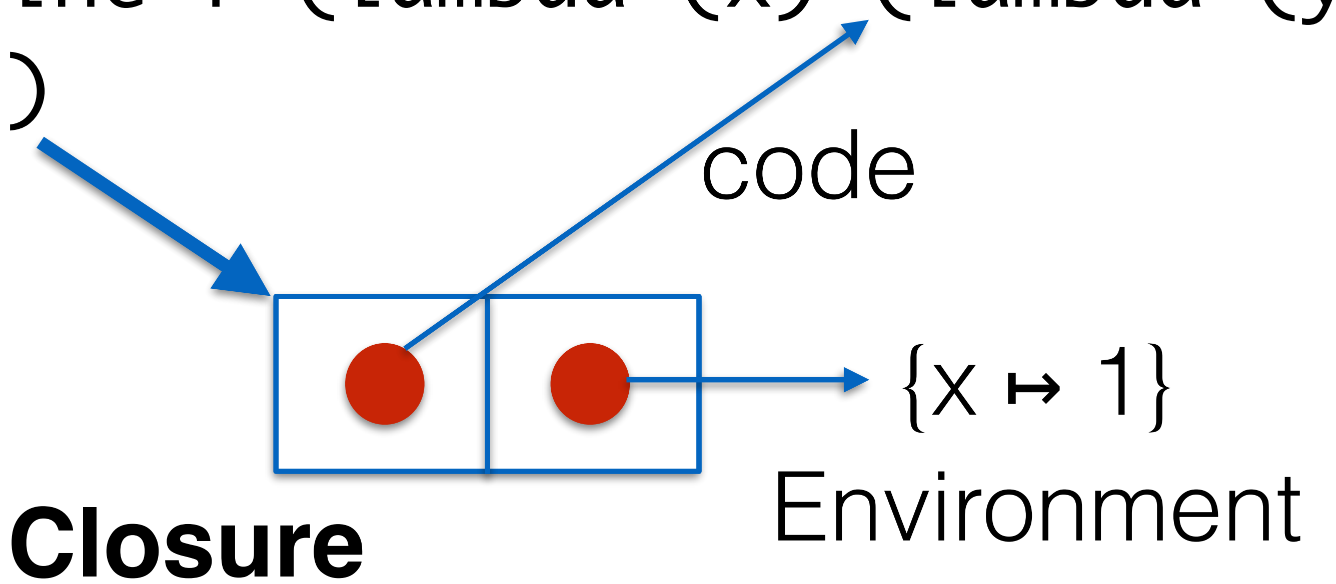


Closures are **code + data**

Some consequences of this…

- Every expression that **returns** a lambda must **allocate** a closure (and copy into the environment)
    - This is project 4—closure conversion
- Means we must also have cheap (fast to allocate, fast lookup) **environments**
- Function **invocation** must load the code from the closure, and must use the environment when executing that code

```
(define f (lambda (x) (lambda (y) x)))
(f 1)
```



This is why Racket represents lambdas as
#<procedure>

```
(lambda (y) 1)
```

#<procedure> is Racket's representation of closures

```
(define (counter i)
  (lambda () (cons i (lambda () (counter (+ i 1))))))
```

E.g., (counter n) returns a lambda that allows you to count up from n.
What variables are captured by **this** lambda?

```
(define (prim? op) (member op '(+ - * / == !=)))

;; lambda calc + arith
(define (expr? e)
  (match e
    [(? number? n) #t]
    [(? boolean? b) #t]
    [(? symbol? x) #t]
    [`(prim ,(? prim?) ,(? expr?) ,(? expr?)) #t]
    [`(lambda (,x) ,(? expr?)) #t]
    [`(,(? expr?) ,(? expr?)) #t]
    [`(if ,(? expr? e-guard) ,(? expr? e-t) ,(? expr? e-f)) #t]
    [_ #f]))
```

```
(define (value? v)
  (match v
    [(? number? n) #t]
    ;; closures are pairs of lambdas and hashes
    ;; (key-value maps) from symbols to values
    [`(clo (lambda (,(? symbol?)) ,(? expr? e))
           ,(? (hash/c symbol? value?))) #t]
    [_ #f]))
```

```
;; closure-creating (big-step) interpreter
;; Takes two arguments: an expression, and an environment
(define (interp e env)
  (match e
    [(? number? n) e]
    [(? boolean? b) b]
    [(? symbol? x) (hash-ref env x)] ;; look x up in the environment
    [`(lambda (,x) ,e-body) `(closure ,e ,env)] ;; create closure
    [`(,e0 ,e1)
     (match-define `(closure (lambda (,x) ,e-b) ,env+) (interp e0 env))
     (interp e-b (hash-set env+ x (interp e1 env)))]
    [`(prim ,op ,e0 ,e1)
     (define v0 (interp e0 env))
     (define v1 (interp e1 env))
     ((match op ['+ +] ['- -] ['* *] ['/ /]
        ['== (lambda (x y) (equal? x y))]) v0 v1)]
    [`(if ,e-g ,e-t ,e-f)
     (if (interp e-g env) (interp e-t env) (interp e-f env))]))
```