Accessibility Services needs dependable **volunteer note-takers** to assist students living with a disability to achieve academic success!

Volunteers report that by giving to the U of T community their class attendance and note taking skills improve.

All you have to do is attend classes regularly & submit your notes consistently:

1. Register Online as a Volunteer Note-Taker at: https://clockwork.studentlife.utoronto.ca/custom/misc/home.aspx
2. Follow the link that says Volunteer Notetakers
3. Select your course and upload a sample of your notes
4. Once you have been selected as a note-taker you'll get an email notifying you to upload your notes.

Volunteers may receive co-curricular credit or a certificate of appreciation.

Assignment 1 has been posted!

(But do Lab 3 and Exercise 3 first!)

free identifier (of a function)

An identifier that is *not bound* inside the function (i.e., not a parameter, or within a let)

closure

A data structure storing two things:

1) a reference to function code
2) an environment containing bindings for all free identifiers in the function body

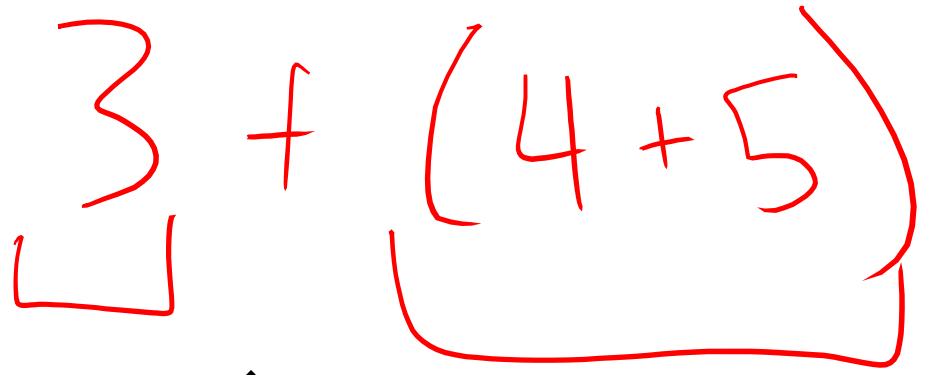
```
(define (make-f n)
  (lambda (x)
    ; REALLY long body
    (... n ... x ...)))

(define f-1 (0x00ff48, {n: 1}))
(define f-2 (0x00ff48, {n: 2}))
(define f-3 (0x00ff48, {n: 3}))
```

When we evaluate a lambda (function value), we get a closure.

The closure environment is created based on when the function is evaluated, **not** when it is called.

A simple recursive interpreter




```
(define/match (interpret expr)
  [((? number?)) expr]
  [(list '+ l r)
   (+ (interpret l)
      (interpret r))]
```

environment

a mapping of identifiers to values

Simple interpreter strategy

pass environment recursively;
accumulate bindings in scopes
(e.g., `define`, `let*`)

```
(define/match (interpret env expr)
  [(_ (? number?)) expr]
  [(_ (? symbol?)) (hash-ref env expr)]
  [(_ (list '+ l r))
   (+ (interpret env l)
      (interpret env r))]
```

```
(define (f a b) (+ a b))

(f 1 2)
; ==>
(+ 1 2)
```

```
(define (f a b) (+ a b))

(interpret ___ (f 1 2))
; ==>
(interpret {a: 1, b: 2} (+ a b))
```

```
(define (f x) (lambda (n) (+ n x)))
(define g (f 1)) ; g == (0x…, {x: 1})

(interpret ___ (g 2))
; ==>
(interpret {n: 2} (+ n x))
```

```
(define (f x) (lambda (n) (+ n x)))
(define g (f 1)) ; g == (0x…, {x: 1})

(interpret ___ (g 2))
; ==>
(interpret {n: 2, x: 1} (+ n x))
```

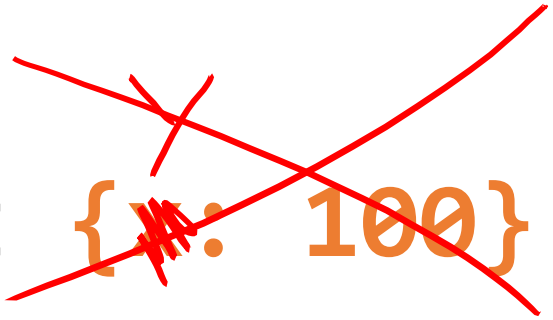
```
(define (f x) (lambda (n) (+ n x)))
(define g (f 1)) ; g == (0x…, {x: 1})
(let* ([x 100])
  (g 2))
```

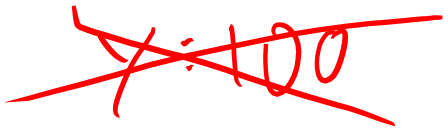
```
(define (f x) (lambda (n) (+ n x)))
(define g (f 1)) ; g == (0x…, {x: 1})
(let* ([x 100])
  (g 2))

(interpret {x: 100} (g 2))
; ==>
(interpret ??? (+ n x))
```

```
(define (f x) (lambda (n) (+ n x)))
(define g (f 1)) ; g == (0x…, {x: 1})
(let* ([x 100])
  (g 2))

(interpret {x: 100} (g 2))
; ==>
(interpret {n: 2, x: 1} (+ n x))
```





When a function is called, its body is evaluated under an environment composed of

1. its parameter bindings
2. its closure environment

lexical (static) scope

free identifiers in function bodies have scope determined by location in source code—i.e., where the function is *defined*

dynamic scope

free identifiers in function bodies have scope determined by runtime environment—i.e., where the function is *called*

Lexical scoping and
things we already know about

Lexical scoping and
~~things we already know about~~
the call stack

Lexical scoping and

~~things we already know about~~

mutation

Evaluation order

When all subexpressions are “valid”, evaluation order doesn’t matter.

strict denotational semantics

strict denotational semantics for function calls

if an argument expression is undefined,
the call expression is undefined

typically implemented using
left-to-right eager evaluation

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

(f (error “failed”))
```

```
f = \x -> 1

f (error “failed”)
```

non-strict denotational semantics for function calls

if an argument expression is undefined,
the entire is undefined *only if that argument is required to evaluate* ***the program****

In strict languages, functions can’t “skip” arguments.

```
(define (avg numbers)
  (if (equal? 0 (length numbers))
      0
      (/ (sum numbers) (length numbers))))
```

“short-circuiting”

In almost every programming language, boolean **and (&&)** and **or (||)** are not functions.

Laziness and accumulators
(Didn’t get to this, will do next week)