

# Roadmap

---

- Midterm, Thursday, November 3, in class; 80 minutes, closed book/notes;
- Sample solutions for homework 2 will be posted by Friday; solutions for homework 3 by Monday.
- Thanksgiving recess: no class on Tuesday, November 22.
- Last lecture: Thursday, December 14; overall, classes end Friday, December 15.

# The TINY language

---

---

$x \in \text{Variables}$

$n \in \text{Integers}$

$c ::= n \mid \#t \mid \#f \mid + \mid - \mid * \mid /$  constants

$v ::= c \mid (\text{lambda } (x \dots) e)$  values

$e ::= v \mid x \mid (e \ e_1 \dots e_k) \mid (\text{if } e_1 \ e_2 \ e_3)$  expressions

$p ::= e$  program

---

This simple functional language does not have constructs to define recursive functions.

Note:  $(\text{lambda } (x \dots) e)$  is a function with one or more arguments (that's what the "...” mean).

We want TINY to be a **lexically scoped** language.

$$\begin{aligned} ev[ & ((\text{lambda } (x) \\ & \quad ((\text{lambda } (z) \\ & \quad \quad ((\text{lambda}(x) (z x)) \\ & \quad \quad \quad 3)) \\ & \quad (\text{lambda } (y) (+ x y)) ) \\ & 1) ] = 4 \end{aligned}$$

“Computation” can be characterized by choosing an application, and substituting formal parameters by their actual arguments.

### Properties of Substitution

- Only formal parameters that are *free* in the function body
- Only capture-free substitution

## Observations

---

Is there any difference between *call-by-value* and *call-by-name* in terms of

- efficiency – How many reduction steps?
- answers computed – Different answers?

Examples:

```
((lambda (x) (+ x x)) ((lambda(x) x) 4))
```

```
((lambda (x) (+ 1 1)) ((lambda(x) x) 4))
```

```
((lambda (x) 1)  
 ((lambda(x) (x x)) (lambda(x) (x x))))
```

Summary:

1. We expect *call-by-name* to have more reduction steps than *call-by-value* (exception: see above).
2. Upon termination, both produce the same answer.
3. There are cases where *call-by-name* terminates, but *call-by-value* does not.

## Another Interpreter for TINY

---

GOAL: Write an “efficient” lexically (statically) scoped interpreter for our example language TINY

What does **efficient** mean?

substitution is expensive since it requires scanning the redex and perform textual replacement each time a function is applied.

How can we avoid substitution *without* changing the “reduction” semantics?

Answer: Use **closures** and **environments**

# An Efficient Interpreter for TINY

---

## Environments

- Defer substitution by recording the bindings for the variables we would substitute in a data structure called an **environment**. If we need the value that a variable denotes, we just look it up in the environment.

*An environment is a finite map from variables to values*

$\rho \in Env = Variables \rightarrow Values$

# An Efficient Interpreter for TINY

---

## Closures

- Pair the environment for the evaluation of an expression with the expression! The environment must contain values for all free variables of the expression. The expression can only be evaluated in its attached environment, making capturing impossible.

Such a pairing is called a **closure**. In our TINY language, only a lambda abstraction is a value that may contain free variables.

*A closure is a pair consisting of an environment and a lambda abstraction*

$$cl \in Closure = \{ \langle \lambda, \rho \rangle \mid FreeVar(\lambda) \subseteq DOM(\rho) \}$$

## Closure Interpreter for TINY

---

NOTE:

- Our set of *values* has changed! Values are now *constants* and *closures*, i.e., lambda abstraction “values” are always “embedded” in closures.
- The definitions of environments and closures are mutually recursive. However, since we do not consider recursion, we are in good shape, i.e., can ignore this fact.

Note: Our closure interpreter *ev* takes a TINY program *and* an **initial environment** as input.

## Our example revisited

---

$((\text{lambda}(x)$   
   $((\text{lambda}(z) ((\text{lambda}(x)(z x)) 3)) (\text{lambda}(y)(+ x y)))) 1)$

---

**substitution**

---

$((\text{lambda}(z)$   
   $((\text{lambda}(x)(z x)) 3))$   
   $(\text{lambda}(y)(+ 1 y)))$

---

$((\text{lambda}(x)$   
   $((\text{lambda}(y)(+ 1 y)) x)) 3)$

---

$((\text{lambda}(y)(+ 1 y)) 3)$

---

$(+ 1 3)$

---

4

## Our example revisited

---

How to apply a closure value to actual argument values?

1. Let  $c_v$  be the closure value  $\langle (\text{lambda}(\mathbf{x}) \ \mathbf{e}), \rho \rangle$ .
2. Apply  $c_v$  to a value  $a_v$  as follows:  
 Evaluate the body  $\mathbf{e}$  of the function in the environment  $\rho$  of the closure **extended** by the mapping of the formal parameter  $\mathbf{x}$  to the actual value  $a_v$  ( $\rho[x \rightarrow a_v]$ ).

$((\text{lambda}(\mathbf{x})$   
 $((\text{lambda}(\mathbf{z}) ((\text{lambda}(\mathbf{x})(\mathbf{z} \ \mathbf{x})) \ 3)) (\text{lambda}(\mathbf{y})(+ \ \mathbf{x} \ \mathbf{y})))) \ 1)$

closure interpreter	
	{ }
$((\text{lambda}(\mathbf{z})$ $((\text{lambda}(\mathbf{x})(\mathbf{z} \ \mathbf{x})) \ 3))$ $(\text{lambda}(\mathbf{y})(+ \ \mathbf{x} \ \mathbf{y})))$	$\{\mathbf{x} \rightarrow 1\}$
$((\text{lambda}(\mathbf{x})$ $(\mathbf{z} \ \mathbf{x})) \ 3)$	$\{\mathbf{x} \rightarrow 1,$ $\mathbf{z} \rightarrow \langle (\text{lambda}(\mathbf{y})(+ \ \mathbf{x} \ \mathbf{y})), \{\mathbf{x} \rightarrow 1\} \rangle\}$
$(\mathbf{z} \ \mathbf{x})$	$\{\mathbf{x} \rightarrow 3,$ $\mathbf{z} \rightarrow \langle (\text{lambda}(\mathbf{y})(+ \ \mathbf{x} \ \mathbf{y})), \{\mathbf{x} \rightarrow 1\} \rangle\}$
$(+ \ \mathbf{x} \ \mathbf{y})$	$\{\mathbf{x} \rightarrow 1, \mathbf{y} \rightarrow 3\}$

4

## More on Scheme

---

– list operations:

+ list building: `'(... ,m ...)` — inserts value of `m` into the list, not the symbol `m`.

– variable binding operations: **define**, **let**, **let\***

```
(let ( (a 2)          ((lambda (a b)
                       (b 3))
      (+ a b))
      (+ a b))
      (+ a b))
```

– `(map f (...))`: applies function `f` separately to each element in the list and returns the list of results.

Example: `(map (lambda(x)(+ x 1)) '(0 1 2 3))`  
evaluates to `'(1 2 3 4)`

– `(apply f (...))`: applies function `f` to an argument list and returns the resulting value

– `(error ...)`: predefined error routine. Terminates program execution and returns error message with optional values

## Closure interpreter *ev* – basic structure

---

*ev* takes as input an AST of an expression **e** and an environment **env**, and returns the AST of a value.

```
(define ev
  (lambda (e env)
    (cond
      ((eq? (car e) '&const) ;; e=(&const c)
       e)
      ((eq? (car e) '&var) ;; e=(&var v)
       (lookup env (cadr e)))
      ((eq? (car e) '&lambda) ;; e=(&lambda parms body)
       (mk-closure env e))
      ((eq? (car e) '&if)
       (let ((a (cadr e)) ;; e=(&if a b c)
             (b (caddr e))
             (c (cadddr e)))
         (ev (if (equal? (ev a env) '&const #f)) c b)
             env)))
      ((eq? (car e) '&apply) ;; e=(&apply f args)
       (let*((f (cadr e))
              (args (caddr e))
              (fv (ev f env))
              (av (map (lambda (a) (ev a env)) args)))
         (if (and (pair? fv) (eq? (car fv) '&const))
             (delta fv av)
             (apply-cl fv av)))))))))
```

## TINY interpreter $ev$ — basic structure

---

Note:

Just by looking at the function  $ev$  we do not know **how** environments or closures are implemented.  $ev$  is said to be *representation independent*.

## TINY interpreter *ev* — closures

---

### list (data) representation

```
(define mk-closure ;; returns (&closure env parm-list body)
  (lambda (env v)
    (cond
      ((eq? (car v) '&lambda)
       '(&closure ,env ,(cadr v) ,(caddr v))))))
```

```
(define apply-cl
  (lambda (vf va)
    (cond
      ((eq? (car vf) '&closure)
       (let ((env (cadr vf)) ;; environment
             (p (caddr vf)) ;; parameter list
             (b (caddrdr vf))) ;; body
         (if (= (length p) (length va))
             (ev b (extend* env p va))
             (error 'apply-cl "wrong number of arguments"))))))))
```

## TINY interpreter *ev* — environments

---

### procedural (functional) representation

```
(define extend
  (lambda (env x v)
    (lambda (y)
      (if (eq? x y)
          v
          (lookup env y)))))
```

```
(define lookup
  (lambda (env y) (env y)))
```

```
(define extend*
  (lambda (env xs vs)
    (if (null? xs)
        env
        (extend* (extend env (car xs) (car vs))
                  (cdr xs)
                  (cdr vs)))))
```

## TINY interpreter *ev* — **delta**

---

*delta* is needed to apply a functional constant (our  $+$ ,  $-$ ,  $*$ ,  $/$  operations) to a value. *ev* uses the corresponding Scheme operations. If a function symbol such as “+” is encountered, it is looked up in the environment and the list ‘(&const *function*+’) is returned. Note that an application of such a function requires all arguments to be constants, i.e., of the form ‘(&const ...).

```
(define delta
  (lambda (f a)
    (let ((R (lambda (s) ‘(&const ,s)))
          (R-1 (lambda (cl)
                 (cond
                  ((eq? (car cl) ‘&const)
                   (cadr cl))
                  (else
                   (error ’delta "non-const args"))))))
      (R (apply (R-1 f) (map R-1 a))))))
```

```
(define interpret-free-var
  (lambda (x)
    ‘(&const ,(eval x))))
(define empty-env interpret-free-var)
```

## TINY interpreter *ev* — parser

---

```
(define parse
  (lambda (m)
    (cond
      ((number? m)  '(&const ,m))
      ((eq? #t m)   '(&const #t))
      ((eq? #f m)   '(&const #f))
      ((name? m)    '(&var ,m))
      ((pair? m)
       (cond
         ((eq? 'if (car m))
          (if (= 4 (length m))
              '(&if ,(parse (cadr m))
                    ,(parse (caddr m)) ,(parse (cadddr m)))
              (error 'parse "Syntax error")))
         ((eq? 'lambda (car m))
          (if (and (= 3 (length m))
                  (list? (cadr m))
                  (andmap name? (cadr m)))
              '(&lambda ,(cadr m) ,(parse (caddr m)))
              (error 'parse "Syntax error")))
         (else
          '(&apply ,(parse (car m)) ,(parse* (cdr m))))))
      (else (error 'parse "Syntax error"))))

(define parse* (lambda (m) (map parse m)))
```

## TINY interpreter *ev* — parser / unparser

---

```
(define name?
  (lambda (s)
    (and (symbol? s) (not (memq s '(if lambda))))))

(define andmap
  (lambda (f l)
    (if (null? l)
        #t
        (and (f (car l)) (andmap f (cdr l)))))

(define unparse
  (lambda (a)
    (cond
      ((eq? (car a) '&const) (cadr a))
      ((eq? (car a) '&var) (cadr a))
      ((eq? (car a) '&if)
       '(if ,(unparse (cadr a)) ,(unparse (caddr a))
            ,(unparse (caddrd a))))
      ((eq? (car a) '&lambda)
       '(lambda ,(cadr a) ,(unparse (caddr a))))
      ((eq? (car a) '&apply)
       (cons (unparse (cadr a)) (map unparse (caddr a))))
      ((eq? (car a) '&closure)
       '(lambda ,(caddr a) ,(unparse (caddrd a))))
      (else (error 'unparse "unexpected syntax tree" a))))))
```

## Call-by-value closure interpreter *evaluate*

---

That's it! We are now ready to put everything together.

---

```
(define evaluate
  (lambda (m)
    (unparse (ev (parse m) empty-env))))
```

---

### Questions

- What parameter passing style does our interpreter use?
- What is the order of evaluation of the actual parameters for a function application?

## Call-by-name closure interpreter

---

How do we have to modify our interpreter to implement call-by-name instead of call-by-value?

### Key idea:

If the argument of an application is not a value, we can postpone its evaluation by wrapping it into a closure that treats the argument expression as a lambda abstraction without parameters. Such a “special” closure is called a **thunk**. If we need the “real” value of an argument expression, we just evaluate the thunk in its environment.

Note that now our environment can contain three types of values, namely *constants*, *closures*, and *thunks*.

See example interpreter on **ilab**:

- name:

~uli/cs515/examples/scheme/NameStatic.ss

## Dynamically-scoped interpreter

---

How do we have to modify our interpreter to implement call-by-value with dynamic scoping?

### Key idea:

The difference between static and dynamic scoping in TINY is with respect to the rules how closures are applied.

- static  $\Rightarrow$  use environment within the closure
- dynamic  $\Rightarrow$  ignore environment within the closure and use current environment at the application.

See example interpreters on **ilab**:

- static:  
    ~uli/cs515/examples/scheme/ValueStatic.ss
- dynamic:  
    HOMEWORK!

## Closure interpreters — Remember the Y

---

```
(lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
```

This does not work for *call-by-value* interpreters!

**Trick:** – Enclose the self application (x x) within a function (lambda(y) ((x x) y))

```
(lambda (f)
  ((lambda (x) (lambda (y) ((f (x x)) y)))
   (lambda (x) (lambda (y) ((f (x x)) y)))))
```

Let's try an example using our call-by-value interpreter:

```
> (define test-fac3
'(((lambda (f)
  ((lambda (x) (lambda (y) ((f (x x)) y)))
   (lambda (x) (lambda (y) ((f (x x)) y)))))
 (lambda (fac)
  (lambda (x)
   (if (equal? x 1) 1 (* x (fac (- x 1))))))) 3))
> (evaluate test-fac3)
> 6
```

**NOTE:** Use (trace ev) to see what happens! Check out [~uli/cs515/examples/scheme/YY.ss](http://uli.cs515/examples/scheme/YY.ss) on the ilab cluster.

## Closure interpreters

---

- Interpreters are important tools to understand and specify the semantics of programming languages.
- Closures are an important concept for any language with functions as first order objects and static scoping. It gives us an idea what “price” we have to pay if we want first–order functions or a particular parameter passing style.
- So far, we only talked about “pure” functional languages without a store and *assignment* (no **side effects**). How to extend our interpreters to deal with a store?
  - Introduce a *store* as a function from addresses to values:  $store = Memory\ Locations \rightarrow Values$
  - Modify environments to map names to memory locations:  $\rho \in Env = Variables \rightarrow Memory\ Locations$
  - Modify *ev* to take one more argument: (define ev (lambda (e env store) ...)). In addition, *ev* will return a value and a store.
  - Of course, there is a lot more work to be done here, but I hope you get the idea.