Announcements

• Welcome back!

• Reminder: **Second project** Deadline: Friday, April 8

• Fourth homework will be posted soon.

• Roadmap: Defining interpreters, type systems, logic programming (Prolog), parallel programming
Equality Checking

The `eq?` predicate doesn’t work for lists.
Why not?

1. `(cons 'a '())` produces a new list
2. `(cons 'a '())` produces another new list
3. `eq?` checks if its two arguments are the same
4. `(eq? (cons 'a '()) (cons 'a '()))` evaluates to `#f`

Lists are stored as pointers to the first element (car) and the rest of the list (cdr). This elementary “data structure”, the building block of lists, is called a pair.

![pair diagram]

Symbols are stored uniquely, so `eq?` works on them.
Review: Lists in Scheme

The building blocks for lists are **pairs** or **cons-cells**. Lists use the empty list () as an “end-of-list” marker.

Note: (a.b) is not a list!
Equality Checking for Lists

For lists, need a comparison function to check for the same \textit{structure} in two lists

\begin{verbatim}
(define equal? (lambda (x y) 
  (or (and (atom? x) (atom? y) (eq? x y)) 
      (and (not (atom? x)) (not (atom? y))
        (equal? (car x) (car y))
        (equal? (cdr x) (cdr y))))))
\end{verbatim}

- \((\text{equal? } \text{’a } \text{’a})\) evaluates to \texttt{#t}
- \((\text{equal? } \text{’a } \text{’b})\) evaluates to \texttt{#f}
- \((\text{equal? } \text{’(a) } \text{’(a)})\) evaluates to \texttt{#t}
- \((\text{equal? } \text{’((a)) } \text{’(a)})\) evaluates to \texttt{#f}
Higher-order Functions: map

(define map
  (lambda (f l)
    (if (null? l)
      ()
      (cons (f (car l)) (map f (cdr l))))))

- map takes two arguments: a function and a list
- map builds a new list by applying the function to every element of the (old) list
Higher-order Functions: map

- Example:
  \[
  (\text{map abs } '(-1 2 -3 4)) \Rightarrow \\
  (1 2 3 4)
  \]
  \[
  (\text{map (lambda (x) (+ 1 x)) '(-1 2 -3)) } \Rightarrow \\
  (0 3 -2)
  \]
- Actually, the built-in map can take more than two arguments:
  \[
  (\text{map + '}(1 2 3) ')(4 5 6)) \Rightarrow \\
  (5 7 9)
  \]
Review – Constants and Quotes

• Constants denote particular values. These values cannot be changed. Examples: 1, 2, #t, #f

• Function quote can be used to inhibit evaluation of its argument, converting it into data (e.g.: symbol or list). Examples: (quote a) (quote (a b c 1))

  Abbreviation: (quote a) ≡ ’a

• Functions quasiquote and unquote allow the construction of data structures by allowing unquoted expressions (including symbols) to be evaluated, and the value be inserted into the data structure.

  Example: ((lambda (m) (quasiquote (n (unquote (+ 1 m)) o))) 5)

  ⇒ (n 6 o)

  Abbreviations:

  (quasiquote (a b)) ≡ ‘(a b)

  (unquote m) ≡ ,m

• unquote does not lead to any evaluation in a quoted data structure

  Examples: ((lambda (m) (quote (n (unquote (+ 1 m)) o))) 5)

  ⇒ (n (unquote (+ 1 m)) o)
The TINY language

\[ x \in \text{Variables} \]
\[ n \in \text{Integers} \]
\[ c ::= n \mid \#t \mid \#f \mid + \mid - \mid \ast \mid / \] \hspace{1cm} \text{constants} \\
\[ v ::= c \mid (\text{lambda (x . . .) e}) \] \hspace{1cm} \text{values} \\
\[ e ::= v \mid x \mid (e \ e_1 \ldots e_k) \mid (\text{if } e_1 \ e_2 \ e_3) \] \hspace{1cm} \text{expressions} \\
\[ p ::= e \] \hspace{1cm} \text{program} \\

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

Note: (\text{lambda (x . . .) e}) is a function with one or more arguments (that’s what the “…” mean).

We want TINY to be a \text{lexically scoped} language.
ev[ ((lambda (x)
    ((lambda (z)
        ((lambda(x) (z x))
        3))
    (lambda (y) (+ x y)) )
1) ] = 4

“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
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*

\[ \rho \in Env = Variables \rightarrow Values \]
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.

\[ cl \in \text{Closure} = \{ (\lambda, \rho) | \text{FreeVar}(\lambda) \subseteq \text{DOM}(\rho) \} \]

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

$$(\lambda(x) ((\lambda(z) ((\lambda(x)(z \ x)) \ 3)) (\lambda(y)(+ \ x \ y)))) \ 1)$$

<table>
<thead>
<tr>
<th>substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$(\lambda(z) ((\lambda(x)(z \ x)) \ 3)) (\lambda(y)(+ \ 1 \ y)))$$</td>
</tr>
</tbody>
</table>

| $$(\lambda(x) ((\lambda(y)(+ \ 1 \ y)) x)) \ 3)$$ |
| $$(\lambda(y)(+ \ 1 \ y)) \ 3)$$ |
| (+ 1 3) |

4
Our example revisited

How to apply a closure value to actual argument values?

1. Let $c_v$ be the closure value $\langle (\text{lambda}(x) \ e), \rho \rangle$.

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

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

<table>
<thead>
<tr>
<th>closure interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>{ }</td>
</tr>
<tr>
<td>((\text{lambda}(z)</td>
</tr>
<tr>
<td>{x\rightarrow1}</td>
</tr>
<tr>
<td>((\text{lambda}(x)(z \ x)) 3))</td>
</tr>
<tr>
<td>(\text{lambda}(y)(+ x y))</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>((\text{lambda}(x)</td>
</tr>
<tr>
<td>{x\rightarrow1,</td>
</tr>
<tr>
<td>(z x) 3)</td>
</tr>
<tr>
<td>z\rightarrow{(\text{lambda}(y)(+ x y)), {x\rightarrow1}}</td>
</tr>
<tr>
<td>(z x)</td>
</tr>
<tr>
<td>{x\rightarrow3,</td>
</tr>
<tr>
<td>z\rightarrow{(\text{lambda}(y)(+ x y)), {x\rightarrow1}}</td>
</tr>
<tr>
<td>(+ x y)</td>
</tr>
<tr>
<td>{x\rightarrow1, y\rightarrow3}</td>
</tr>
</tbody>
</table>

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)) 2 3)
     (+ 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 \textit{ev} — basic structure

Note:

Just by looking at the function \textit{ev} we do not know how environments or closures are implemented. \textit{ev} is said to be \textit{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 (cadddr 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 \textit{ev} — \textit{delta}

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

\begin{verbatim}
(define delta
  (lambda (f a)
    (let ((R (lambda (s) ‘(\&\texttt{const} ,s)))
      (R-1 (lambda (cl)
        (cond
          ((eq? (car cl) \&\texttt{const})
           (cadr cl))
          (else
           (error \texttt{delta} "non-const args"))))))
    (R (apply (R-1 f) (map R-1 a))))))

(define interpret-free-var
  (lambda (x)
    ‘(\&\texttt{const} ,(eval x))))

(define empty-env interpret-free-var)
\end{verbatim}
(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 \textit{ev — parser / unparsers}

\begin{verbatim}
(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 (cadddr 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 (cadddr a)))) (else (error 'unparse "unexpected syntax tree" a))))
\end{verbatim}
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?