Announcements

• My office hours: Fridays, 10:30am - noon, CoRE 318
• First homework problem set has been posted. Will post solutions at the end of the week.
• SP numbers or other issues? See me during our break.
Functional Programming

Pure Functional Languages

Scott Chapter 10.

Fundamental concept: application of (mathematical) functions to values

1. Referential transparency: The value of a function application is independent of the context in which it occurs

   • value of $f(a,b,c)$ depends only on the values of $f$, $a$, $b$ and $c$

   • It does not depend on the global state of computation

   ⇒ all vars in function must be local, or parameters
Pure Functional Languages

1. The concept of assignment is **not** part of functional programming
   - no explicit assignment statements
   - variables bound to values only through the association of actual parameters to formal parameters in function calls
   - function calls have no side effects
   - thus no need to consider global state

2. Control flow is governed by function calls and conditional expressions
   - ⇒ no iteration
   - ⇒ recursion is widely used
Pure Functional Languages

1. All storage management is implicit
   • needs garbage collection

2. Functions are *First Class Values*
   • Can be returned as the value of an expression
   • Can be passed as an argument
   • Can be put in a data structure as a value
   • (Unnamed) functions exist as values
LISP

- Functional language developed by John McCarthy in the mid 50’s
- Semantics based on *Lambda Calculus*
- All functions operate on lists or symbols: (called “S-expressions”)
- Only five basic functions: list functions *cons*, *car*, *cdr*, *equal*, *atom* and one conditional construct: *cond*
- Useful for list-processing applications
- Programs and data have the same syntactic form: S-expressions
- Used in Artificial Intelligence
- SCHEME: Developed in 1975 by G. Sussman and G. Steele as a version of LISP

⇒ we are using SCHEME here

You can call SCHEME interpreters on the ilab cluster by saying: *mzscheme* or *racket* (command line interpreter); *drracket* (window-based interpreter).
S-expressions are 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!
Special (Primitive) Functions

- **eq?**: identity on names (atoms)
- **null?**: is list empty?
- **car**: selects first element of list \((\text{contents of address part of register})\)
- **cdr**: selects rest of list \((\text{contents of decrement part of register})\)
- **(cons element list)**: constructs lists by adding element to front of list
- **quote** or ’: produces constants
Special (Primitive) Functions

• **car** and **cdr** can break up any list:
  \[-(\text{car} \ (\text{cdr} \ (\text{cdr} \ '((a) \ b \ (c \ d)))))\]  
  \[-(\text{caddr} \ '((a) \ b \ (c \ d)))\]

• **cons** can construct any list:
  \[-(\text{cons} \ 'a \ ()\)\]  
  \[-(\text{cons} \ 'd \ '(e))\]  
  \[-(\text{cons} \ '((a) \ b) \ '(c \ d))\]  
  \[-(\text{cons} \ '((a) \ b) \ '((a) \ b))\]
Other Functions

- + − ∗ / numeric operators, e.g.,
  
  (+ 5 3) = 8, (- 5 3) = 2
  
  (* 5 3) = 15, (/ 5 3) = 1.6666666

- = < > comparison operators for numbers

- Explicit type determination and test functions:

  ⇒ All return Boolean values: #f and #t

  - (number? 5) evaluates to #t
  - (zero? 0) evaluates to #t
  - (symbol? ’sam) evaluates to #t
  - (list? ’(a b)) evaluates to #t
  - (null? ’()) evaluates to #t

Note: SCHEME is a strongly typed language.
Other Functions

- `(number? 'sam)` evaluates to `#f`
- `(null? '(a))` evaluates to `#f`
- `(zero? (- 3 3))` evaluates to `#t`
- `(zero? '(- 3 3))` ⇒ type error
- `(list? (+ 3 4))` evaluates to `#f`
- `(list? '(+ 3 4))` evaluates to `#t`
READ-EVAL-PRINT Loop

**READ:** Read input from user:
   a function application

**EVAL:** Evaluate input:
   \( (f \ arg_1 \ arg_2 \ldots \ arg_n) \)
   1. evaluate \( f \) to obtain a function
   2. evaluate each \( \arg_i \) to obtain a value
   3. apply function to argument values

**PRINT:** Print resulting value:
   the result of the function application
READ-EVAL-PRINT Loop Example

> (cons 'a (cons 'b '(c d)))
(a b c d)

1. Read the function application
   (cons 'a (cons 'b '(c d)))

2. Evaluate cons to obtain a function

3. Evaluate 'a to obtain a itself

4. Evaluate (cons 'b '(c d)):
   (a) Evaluate cons to obtain a function
   (b) Evaluate 'b to obtain b itself
   (c) Evaluate '(c d) to obtain (c d) itself
   (d) Apply the cons function to b and (c d) to obtain (b c d)

5. Apply the cons function to a and (b c d) to obtain (a b c d)

6. Print the result of the application:
   (a b c d)
Quotes Inhibit Evaluation

;; Same as before:
> (cons 'a (cons 'b '(c d)))
(a b c d)

;; Now quote the second argument:
> (cons 'a '(cons 'b '(c d)))
(a cons (quote b) (quote (c d)))

;; Instead, un-quote the first argument:
> (cons a (cons 'b '(c d)))
ERROR: unbound variable: a
Quotes Inhibit Evaluation

;; Some things evaluate to themselves:
> (list 1 2 #t #f)
(1 2 #t #f)

;; They can also be quoted:
> (list '1 '2 '#t '#f)
(1 2 #t #f)
Defining Global Variables

> (define foo '(a b c))
#<unspecified>

> (define bar '(d e f))
#<unspecified>

> (append foo bar)
(a b c d e f)

> (cons foo bar)
((a b c) d e f)

> (cons ’foo bar)
(foo d e f)
Defining Scheme Functions

(define <fcn-name> (lambda (<fcn-params>)
  <expression>))

Example: Given function pair? (true for non-empty lists, false o/w) and function not (boolean negation):

(define atom?
  (lambda (object) (not (pair? object))))

Evaluating (atom? '(a)):
1. Obtain function value for atom?
2. Evaluate '(a) obtaining (a)
3. Evaluate (not (pair? object))
   a) Obtain function value for not
   b) Evaluate (pair? object)
      i. Obtain function value for pair?
      ii. Evaluate object obtaining (a)
      Evaluates to #t
   Evaluates to #f
Evaluates to #f
Conditional Execution: if

(if <condition> <result1> <result2>)

1. Evaluate <condition>

2. If the result is a “true value” (i.e., anything but #f), then evaluate and return <result1>

3. Otherwise, evaluate and return <result2>

(define abs-val
  (lambda (x)
    (if (>= x 0) x (- x))))

(define rest-if-first
  (lambda (e l)
    (if (eq? e (car l)) (cdr l) '()))
Conditional Execution: cond

(cond (<condition1> <result1>))
  (<condition2> <result2>)
  ...
  (<conditionN> <resultN>)) ; optional else
  (else <else-result>)) ; clause

1. Evaluate conditions in order until obtaining one that returns a true value
2. Evaluate and return the corresponding result
3. If none of the conditions returns a true value, evaluate and return <else-result>
Conditional Execution: cond

(define abs-val
  (lambda (x)
    (cond ((>= x 0) x)
          (else (- x)))))

(define rest-if-first
  (lambda (e l)
    (cond ((null? l) '())
          ((eq? e (car l)) (cdr l))
          (else '()))))
Recursive Scheme Functions: Abs-List

• (abs-list ’(1 -2 -3 4 0)) ⇒ (1 2 3 4 0)
• (abs-list ’()) ⇒ ()

(define abs-list
  (lambda (l)
    (if (null? l)
      ’()
      (cons (abs-val (car l)) (abs-list (cdr l))))))
Recursive Scheme Functions: Append

(append '(1 2) '(3 4 5) ⇒ (1 2 3 4 5)
(append '(1 2) '(3 (4) 5) ⇒ (1 2 3 (4) 5)
(append () '(1 4 5)) ⇒ (1 4 5)
(append '(1 4 5) '()) ⇒ (1 4 5)
(append '() '()) ⇒ ()

(define append
  (lambda (x y)
    (cond ((null? x) y)
          ((null? y) x)
          (else (cons (car x) (append (cdr x) y))))
  )


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**.

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 structure in two lists

\[
\text{(define equal?)}
\]
\[
\text{(lambda (x y)}
\]
\[
\text{  (or (and (atom? x) (atom? y) (eq? x y)))}
\]
\[
\text{    (and (not (atom? x)) (not (atom? y)))}
\]
\[
\text{      (equal? (car x) (car y))}
\]
\[
\text{        (equal? (cdr x) (cdr y))})}
\]

- (equal? 'a 'a) evaluates to #t
- (equal? 'a 'b) evaluates to #f
- (equal? '(a) '(a)) evaluates to #t
- (equal? '((a)) '(a)) evaluates to #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 \texttt{quote} can be used to inhibit evaluation of its argument, converting it into data (e.g.: symbol or list). Examples: \texttt{(quote a)} \texttt{ (quote (a b c 1))}

  \textbf{Abbreviation:} \texttt{(quote a)} \equiv \texttt{'a}

• Functions \texttt{quasiquote} and \texttt{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: \texttt{((lambda (m) (quasiquote (n (unquote (+ 1 m)) o))) 5)}
  \[\Rightarrow (n \ 6 \ o)\]

  \textbf{Abbreviations:}
  \[
  (\texttt{quasiquote (a b)}) \equiv \texttt{'(a b)}
  \]
  \[
  (\texttt{unquote m}) \equiv \texttt{,m}
  \]

• \texttt{unquote} does not lead to any evaluation in a quoted data structure

  Examples: \texttt{((lambda (m) (quote (n (unquote (+ 1 m)) o))) 5)}
  \[\Rightarrow (n \ (\texttt{unquote (+ 1 m)}) \ o)\]
The TINY language

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

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

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

We want TINY to be a \textbf{lexically scoped} language.
ev – An Interpreter for TINY

\[ ev[(\text{lambda} \ (x) \ ((\text{lambda} \ (z) \ ((\text{lambda}(x) \ (z \ x)) \ 3)) \ (\text{lambda} \ (y) \ (+ \ x \ y))) \ 1)] = 4 \]

“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
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
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.

\[
\text{An environment is a finite map from variables to values}
\]

\[
\rho \in Env = \text{Variables} \rightarrow \text{Values}
\]
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.

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

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

\[
((\lambda(x)
   ((\lambda(z) ((\lambda(x)(z \ x)) \ 3)) (\lambda(y)(+ \ x \ y)))) \ 1)
\]

 substitution

\[
((\lambda(z)
   ((\lambda(x)(z \ x)) \ 3))
(\lambda(y)(+ \ 1 \ y)))
\]

\[
((\lambda(x)
   ((\lambda(y)(+ \ 1 \ y)) \ x)) \ 3)
((\lambda(y)(+ \ 1 \ y)) \ 3)
(+ \ 1 \ 3)
\]
4
Our example revisited

How to apply a closure value to actual argument values?

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

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 extended by the mapping of the formal parameter $x$ to the actual value $a_v$ ($\rho[x \rightarrow a_v]$).

\[
\begin{align*}
((\lambda(x) ( (\lambda(z) ((\lambda(x)(z \ x)) \ 3)) (\lambda(y)(+ x y)))) \ 1)
\end{align*}
\]

---

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

\[
\begin{align*}
\text{(define ev} \\
\text{  (lambda (e env)} \\
\text{    (cond} \\
\text{      ((eq? (car e) ’&const) ;; e=(&const c) } \\
\text{        e) } \\
\text{      ((eq? (car e) ’&var) ;; e=(&var v) } \\
\text{        (lookup env (cadr e))) } \\
\text{      ((eq? (car e) ’&lambda) ;; e=(&lambda parms body) } \\
\text{        (mk-closure env e)) } \\
\text{      ((eq? (car e) ’&if) } \\
\text{        (let ((a (cadr e)) ;; e=(&if a b c) } \\
\text{          (b (caddr e)) } \\
\text{          (c (cadddr e))) } \\
\text{        (ev (if (equal? (ev a env) ’(&const #f)) c b) env))) } \\
\text{      ((eq? (car e) ’&apply) ;; e=(&apply f args) } \\
\text{        (let*((f (cadr e)) } \\
\text{          (args (caddr e)) } \\
\text{          (fv (ev f env)) } \\
\text{          (av (map (lambda (a) (ev a env)) args))) } \\
\text{          (if (and (pair? fv) (eq? (car fv) ’&const)) } \\
\text{            (delta fv av) } \\
\text{            (apply-cl fv av))))))))
\end{align*}
\]
Next class

- Lambda calculus
- Review of different defining interpreters
- Type systems