

CS 3721: Programming Languages Lab

Lab #02: Recursion over Lists in Scheme

Due date: February 5, 3:30pm. At the beginning of the next recitation.

Goals of this lab:

- How to conditionally evaluate expressions using `cond` ; What `lambda` is ; How to create and use an anonymous function with `lambda` ; How to use `define` and `lambda` to create a named function ; How to build a simple function that recurses over a list

1. Conditionals in Scheme. We will talk about two control-flow operators, *if*, and *cond*.
cond syntax:

(cond < clause₁ > < clause₂ > ...)

Each *< clause >* should have the form:

(< test > < expression >)

and the last *< clause >* may be an “else clause” which has the form:

(else < expression >)

Semantics: A **cond** expression is evaluated by evaluating the *< test >* expressions of successive *< clause >*s in order until one of them evaluates to a true value. When a *< test >* evaluates to a true value, then the corresponding *< expression >* in its *< clause >* is evaluated, and the result of that *< expression >* is returned as the result of the entire **cond** expression.

What does the following *cond* expression evaluate to? Fill in the blank.

```
(cond
  ((< 3 1) 3)
  ((> 4 2) (> 2 4))          => (evaluates to) _____
  ((= 5 5) '(= 5 5))
  (else 'else)
)
```

Write your own example expression using the *cond* operator, and then write down what that expression evaluates to.

if syntax:

(if < test > < consequent > < alternate >) , or

(if < test > < consequent >)

Semantics: An **if** expression is evaluated as follows: first, *< test >* is evaluated. If it yields a true value, then *< consequent >* is evaluated and its value is returned. Otherwise *< alternate >* is evaluated and its value is returned. Write an example expression using the *if* operator, and then write down what that expression evaluates to.

2. Function expressions. Scheme computation is based on functions and recursive calls instead of on assignment and iterative loops.

A scheme function has the form:

$(\text{lambda } (< \text{parameters } >) < \text{function body } >)$, where $< \text{parameters } >$ are *list of identifiers*, and $< \text{function body } >$ is an expression.

example:

$(\text{lambda } (x) (* x x))$, evaluates to a procedure. One can call this function with the argument $(+ 2 2)$ by the following expression:

$(\underbrace{(\text{lambda } (x) (* x x))}_{\text{procedure}} \underbrace{(+ 2 2)}_{\text{parameter}})$, this evaluates to 16.

★ “lambda” makes it possible to write *anonymous functions* which are the functions that do not have a declared name.

Now, create a function that takes a list as a parameter and builds a new list appending the input list to the input list, again. (*Hint*: Use the operator *cons* while building your list.). Test your function with the following expression.

$> (\underbrace{(\text{lambda } \dots)}_{\text{your anonymous function}} \underbrace{'(a b)}_{\text{input list}}) \overset{\text{should evaluate to}}{\Rightarrow} (\text{list } (\text{list } 'a 'b) 'a 'b)$

3. You can use the keyword “define” to name your functions. An example:

```
(define mySquare (lambda (x) (* x x)))
```

You can call this function simply by typing the following.

```
> (mySquare 4)      =>      16
```

In Scheme every value is either null, a pair, or an atom. Scheme provides predicates *null?* and *pair?* for determining whether a value is '() or a cons cell, respectively. Moreover, for any x , the expression $(\text{or } (\text{null? } x) (\text{pair? } x) (\text{atom? } x))$ should evaluate to true. Now, define the function *atom?* which takes a single parameter x and returns whether x is an atomic value. (*Hint*: use *null?*, and *pair?* operators.)

4. Recursion is the primary programming technique in functional languages such as Lisp or Scheme. A recursive program is composed of three parts: bind a name to a function using define, the base case implementation of the recursive algorithm, and the recursive case implementation of the algorithm (by calling itself).

In general, keep in mind the followings while processing lists recursively:

★ (null? myList) tests if there are no more list elements. Base Case

★ (car myList) accesses the current element of the list (first element of the remaining list), and is usually used to process each element.

★ (cdr myList) is usually passed as a parameter to the recursive call.

★ the recursive function should take the list (or the part of the list that still needs to be processed) as a parameter.

Define a function *member?*, which scans through a list(passed as its second argument) for an element matching its first argument. *member?* evaluates to #t if there is a match, otherwise it evaluates to #f. (*Hint*: use *eqv?*)