The Scheme Programming Language (continued)

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

September 15, 2021

Table of Contents

Table of Contents

Programming languages that are part of the Lisp family have foundational support for lists.

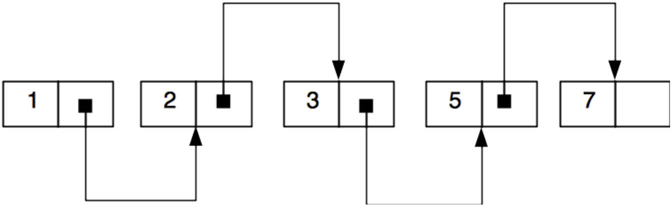
List are implemented as linked lists.

List: '(1 2 3 5 7)

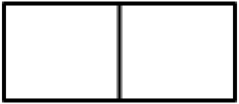
Dot Notation: '(1 . (2 . (3 . (5 . (7 . '()))))))

cons List: (cons 1 (cons 2 (cons 3 (cons 5 (cons 7 '())))))

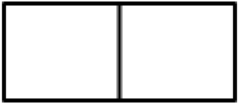
Diagram:



cons Cell

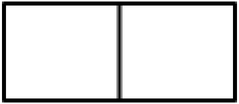


cons Cell



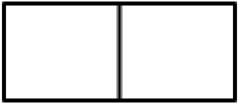
- We can access the left side of the cons cell using the `car` function.

cons Cell



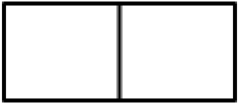
- We can access the left side of the cons cell using the `car` function.
- We can access the right side of the cons cell using the `cdr` functions.

cons Cell



- We can access the left side of the cons cell using the `car` function.
- We can access the right side of the cons cell using the `cdr` functions.
- The names `car` and `cdr` are historical hardware references dating back to the 1950s.

cons Cell



- cons cells can be used to store pairs of objects.
  - Example: `(cons 2 3) => '(2 . 3)` (note the dot syntax)
- cons cells are more often used for making lists
  - First element is typically a value, while the second element is either another cons cell or an empty list `'()`.
  - Example: `(cons 1 (cons 2 (cons 3 '()))) => '(1 2 3)`
  - `(first '(1 2 3)) => 1`
  - `(rest '(1 2 3)) => '(2 3)`

`first` and `rest` vs. `car` and `cdr`

- Use `first` and `rest` for lists.
- In Racket, you must use `car` and `cdr` for non-list cons cells.

S-expressions

An S-expression is one of the following:

- A primitive (e.g., a number, a string, a symbol)
- A list without dots (e.g., `(1 3 5)` and `(+ x y)`)
- A list in dot notation (e.g., `'(1 . (3 . (5 . ())))`)
- Special dot notation case: `(1 3 5 . 7)`
  - Equivalent to `(1 . (3 . (5 . 7)))`

A Lisp program is a collection of S-expressions.

What is the difference between `(+ 2 4)` and `'(+ 2 4)`

What is the difference between `(+ 2 4)` and `'(+ 2 4)`

`(+ 2 4)` is interpreted to call the function + with arguments 2 and 4. In the case of `'(+ 2 4)`, the `'` tells the interpeter not to evaluate what's after it, and so it returns the list `(+ 2 4)` unevaluated.

Quoting

' is shorthand for the `quote` function.

Example: `'x` is equivalent to `(quote x)`.
`'(1 3 5)` is the equivalent of `(quote (1 3 5))`.

Operations on Lists

- `cons` can easily be used to add to the front of a list, returning a new list.
  - Example: `(cons 1 '(2 3)) => '(1 2 3)`
- Use the `append` function to concatenate two lists.
  - Example: `(append '(1 2) '(3 4)) => '(1 2 3 4)`
- Use the `length` function to get the length of the list.
- Use the `empty?` function to check if the list is empty.
- `reverse` reverses a list.

Iterating Over Lists

Since we are coding in functional programming style, we need to use recursion. Thankfully, `first` and `rest` make it easy to write list-traversal code in an easy-to-use manner.

Iterating Over Lists

Since we are coding in functional programming style, we need to use recursion. Thankfully, `first` and `rest` make it easy to write list-traversal code in an easy-to-use manner.

```
; Return the sum of the elements in elem
(define (element-sum elems sum)
  (if (empty? elems)
      sum
      (element-sum (rest elems) (+ sum (first elems)))))
```

Table of Contents

Anonymous Functions with `lambda`

Definition (`lambda`)

`(lambda (arg1 ... argN) function-body)` creates an anonymous function with arguments `arg1` to `argN` that evaluates `function-body`.

Example of Usage of Anonymous Functions

Remember the sort comparison I gave during the last function? Here is an example of using one

```
(sort '(3 2 5 4)
      (lambda (a b)
        (cond ((< a b) -1)
              ((> a b) 1)
              (else 0))))
```

We are passing a custom comparison function that sort can use to compare two numbers. (Note that this is a custom sort function, not Racket's built-in `sort`.)

It turns out that

```
(define (function-name arg1 ...  argN) function-body)
```

is simply syntactic sugar for

```
(define function-name
        (lambda (arg1 ... argN) function-body))
```

In order words, we assign an anonymous function a name.

```
(define (normal-compare a b)
  (cond ((< a b) -1)
        ((> a b) 1)
        (else 0)))

(sort '(3 2 5 4) normal-compare)
```

Table of Contents

Technically we could define variables using `lambda` by declaring unused function arguments:

```
; Computes the hypotenuse z = sqrt(x^2 + y^2)
(define (hypotenuse x y)
  ((lambda (xs ys)
     (expt (+ xs ys) 0.5))
   (expt x 2) (expt y 2)))
```

`xs` is set to the result of `(expt x 2)`, and `ys` is set to the result of `(expt y 2)`.

Scheme provides syntactic sugar for defining local variables. We can use `let` to define local variables.

Definition (`let`)

`(let ((var1 def1) ... (varN defN)) expr)` defines the variables `var1 = def1` to `varN = defN`, where `def1` to `defN` are expressions. `expr` is able to use these variables.

```
; Computes the hypotenuse z = sqrt(x^2 + y^2)
(define (hypotenuse x y)
  (let ((xs (expt x 2))
        (ys (expt y 2)))
    (expt (+ xs ys) 0.5)))
```

What is wrong with this code?

```
; double the numbers in a list
(define (double-list elems)
  (if (empty? elems)
      '()
      (let ((elem (first elems))
            (double-elem (* elem 2)))
        (cons double-elem (double-list (rest elems))))))
```

What is wrong with this code?

```
; double the numbers in a list
(define (double-list elems)
  (if (empty? elems)
      '()
      (let ((elem (first elems))
            (double-elem (* elem 2)))
        (cons double-elem (double-list (rest elems))))))
```

The culprit is in the definition of `double-elem`. The value of `double-elem` is set to `(* elem 2)`. The problem, though, is that we cannot evaluate `elem` because `let` definitions have no access to variables defined by that same `let`.

Here is one possible solution:

```
; double the numbers in a list
(define (double-list elems)
  (if (empty? elems)
      '()
      (let ((elem (first elems)))
        (let ((double-elem (* elem 2)))
          (cons double-elem (double-list (rest elems)))))))
```

We could have a `let` embedded in a `let`, which solves the scoping problem, but this can get unwieldy for long lists of definitions with dependencies.

```
; double the numbers in a list
(define (double-list elems)
  (if (empty? elems)
      '()
      (let* ((elem (first elems))
             (double-elem (* elem 2)))
        (cons double-elem (double-list (rest elems))))))
```

A better solution is `let*`, which allows the definition of variables that depend on previous variable definitions within the same `let*` definition list.

We can set anonymous functions to variables using `let` and `let*`:

```
(define (normal-sort elems)
  (let ((compare (lambda (a b)
                   (cond ((< a b) -1)
                         ((> a b) 1)
                         (else 0)))))
    (sort elems compare)))
```

Be cautious about declaring recursive functions inside of a `let` or a `let*`. The following code does not work:

```
(define (factorial n)
  (let ((fact (lambda (x answer)
                (if (<= x 1)
                    answer
                    (fact (- x 1) (* x answer))))))
    (fact n 1)))
```

It does not work because the lambda has no access to `fact`. To get it to work, we replace `let` or `let*` with `letrec`.

```
(define (factorial n)
  (letrec ((fact (lambda (x answer)
                   (if (<= x 1)
                       answer
                       (fact (- x 1) (* x answer))))))
    (fact n 1)))
```

Summary

- Scheme has a family of functions used for defining local variables: `let`, `let*`, and `letrec`.
- Use `let` when defining local variables that have no dependencies on other local variables defined in the same `let` scope.
- Use `let*` when defining local variables that depend on previously-defined variables within the same `let*` list.
- Use `letrec` when defining recursive local functions using `lambda`.
- When in doubt, use `letrec`, but stylistically use the most restrictive function of the `let` family applicable to your program (e.g., don't use a `let*` when a `let` is appropriate).

Table of Contents

The Downside of Recursion

- In functional programming, recursion is the preferred way of performing repetitive tasks that would normally be implemented as loops in procedural programs.

The Downside of Recursion

- In functional programming, recursion is the preferred way of performing repetitive tasks that would normally be implemented as loops in procedural programs.
- However, deep levels of recursion could result in a stack overflow error, caused by too many function calls on the call stack.

The Downside of Recursion

- In functional programming, recursion is the preferred way of performing repetitive tasks that would normally be implemented as loops in procedural programs.
- However, deep levels of recursion could result in a stack overflow error, caused by too many function calls on the call stack.
- To solve this problem, we can use tail recursion.

Tail Recursion

Tail recursion is a style of recursion where the final call of a recursive function is a call to itself.

Recursion vs. Tail Recursion

Without tail recursion:

```
(define (factorial n)
  (if (<= n 1) 1 (* n (factorial (- n 1)))))
```

Recursion vs. Tail Recursion

Without tail recursion:

```
(define (factorial n)
  (if (<= n 1) 1 (* n (factorial (- n 1)))))
```

The reason why this is not tail recursive is because when $n \geq 1$, `factorial`'s final (tail) function call is `*`.

With tail recursion

```
(define (factorial n)
  (letrec ((fact (lambda (x answer)
                   (if (<= x 1)
                       answer
                       (fact (- x 1) (* x answer))))))
    (fact n 1)))
```

With tail recursion

```
(define (factorial n)
  (letrec ((fact (lambda (x answer)
                   (if (<= x 1)
                       answer
                       (fact (- x 1) (* x answer))))))
    (fact n 1)))
```

The reason why this is tail recursive is because when $n \geq 1$, `fact`'s final (tail) function call is `fact`.

Another example of tail recursion:

```
(define (my-length elems)
  (letrec ((my-len (lambda (e len)
                     (if (empty? e)
                         len
                         (my-len (rest e) (+ 1 len))))))
    (my-len elems 0)))
```

Why Tail Recursion

- Scheme and many other functional programming language interpreters offer tail call optimization that is able to detect tail recursion and execute them in such a way where it is essentially a `do` loop, thus avoiding the repeated use of the call stack and thus avoiding stack overflow errors.
- Standards-compliant Scheme interpreters *must* implement tail call optimization.
- We'll learn more about the implementation details when we cover virtual machines and compilation in two weeks.

Table of Contents

Functional programming style involves a lot of recursive list traversals.

What if the language provided library functions that facilitated common patterns, such as:

- Performing an operation on all elements of the list (e.g., doubling all the numbers, or making all strings in a list uppercase)
- Removing elements from a list (e.g., removing a list of all non-prime numbers)
- Performing an aggregate operation on a list returning a single element (e.g., adding all numbers in a list)

What if the language provided library functions that facilitated common patterns, such as:

- Performing an operation on all elements of the list (e.g., doubling all the numbers, or making all strings in a list uppercase)
- Removing elements from a list (e.g., removing a list of all non-prime numbers)
- Performing an aggregate operation on a list returning a single element (e.g., adding all numbers in a list)

We have these features in Scheme: map, filter, and fold (respectively)

Map Function

Definition (map)

`(map function list1 ... listN)` applies `function` to all elements of each list `list1` to `listN`, where each list element serves as an argument to `function`. For example, if there are N lists, then there are N arguments to `function`.

Example

```
; Answer is '(1 4 9 16 25)
(map (lambda (x) (* x x)) '(1 2 3 4 5))
; Answer is '(5 7 9)
(map + '(1 2 3) '(4 5 6))
```

Filter Function

Definition (filter)

`(filter function list)` applies `function` (with one argument) to each element of `list`, constructing a new list where `function` returned `#t`.

Example

```
(filter even? '(1 2 3 4 5)) => '(2 4)
(filter (lambda
          (x) (= (remainder x 3) 0))
        '(1 2 3 4 5 6 7 8 9 10)) => '(3 6 9)
```

Sadly, R5RS Scheme does not contain a `filter` function, but newer Schemes such as R6RS Scheme and Racket do, and it is simple to define our own version.

`foldl` and `foldr`

CAUTION: In Racket these are known as `foldl` and `foldr`, but in R6RS Scheme these are `fold-left` and `fold-right`. Unfortunately these are unavailable in R5RS Scheme, but once again these functions are trivial to define.

Definition

`foldl` and `foldr` `(foldl function init list1 ... listN)` and `(foldr function init list1 ... listN)` apply `function` to the elements of lists `list1` to `listN`. `function` accepts $N + 1$ arguments, where one of the arguments is `init`, which is the accumulated aggregate value. `foldl` starts from the left, while `foldr` starts from the right.

Examples

```
; Return the sum of all elements in the list
(foldl + 0 '(1 2 3 4))
```

The above code is the equivalent of performing the sum $0 + 1$ (where 0 is from the `init` parameter) and storing the result in `init`, then $1 + 2$, then $3 + 3$, then finally $6 + 4$, resulting in 10.

Examples

```
(foldl
  (lambda (x count)
    (if (even? x)
        (+ 1 count)
        count))
  0
  '(1 2 3 4 5 6 7 8 9 10))
```

The above code counts the number of even numbers between 1 and 10.

Table of Contents

Quicksort Demo

I will be giving a demo of Quicksort implemented in Scheme using the DrRacket IDE.

Table of Contents

Topics To Be Covered

- The Concept of State in Programming
- Mutation in Scheme
- Environments (used for evaluation purposes)