Learning A Second Language

Jeetaditya Chatterjee

# How many people have used a language other than python in the within the last decade?

If your hand has gone up then I applaud you. You have passed the second barrier that comes to programmers (the first being learning your first language). you have broken that barrier of locking your programming thinking into the language you are using. If you have not we will break that barrier today and hopefully you should be more equipped to walk out into the world of programming languages and see them not a monolith but a medium to have a conversation with your computer

# so I presume that this code snippet will make little sense to people

so unless you are familiar with obscure academic languages then I will presume that you can't read this code all that well

does anyone want to guess what this code does?

**give hints**

* well we can probably say that lst is a list and that function is a function
* we can probably presume that the last line calls the function function.
* other than that we will probably struggle to read more without some sort of indepth look at this languages semantics

(define lst '(23 3 1 4 2 4 69 420) )  
  
(define (function input)  
 (cond  
 ((or (null? input)  
 (null? (cdr input)))  
 input)  
 (else  
 (let ((pivot (car input))  
 (rest (cdr input)))  
 (append  
 (function  
 (filter (lambda (x) (< x pivot)) rest))  
 (list pivot)  
 (function  
 (filter (lambda (x) (>= x pivot)) rest)))))))  
  
(function lst)

* Our Output:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| * 1 | * 2 | * 3 | * 4 | * 4 | * 23 | * 69 | * 420 |

## the basic syntax

Now while syntax is important its is just a medium and you should know how to use and exploit and once understood is the easiest part of learning a language the syntax of scheme is very simple being comprised of 2 major forms in scheme. atomic forms and s-expressions atomic forms are things like Numbers Strings Symbols Booleans values and Characters.

s-expression's or symbolic expression's are a combinations of these atoms to form larger structures. usually the first form after the first paren is the function call then the next form is an argument to that function.

if you use a single quote before the start of a paren or before a symbol as shown in the atom then it will be treated literally the first construct is how we construct lists in scheme. the second is useful for passing around functions and variables without having to resort to some weird string conversion hacks

* atoms
* 1 ; numbers  
  "string"  
  'sym ; symbols  
  #t #f ;; booleans  
  #\c ; harcters
* s-exps
* (some-function arg1 arg2 arg3...)  
  (some-function '("a" "list" "of" "strings")) ; an example of a string list  
  (some-function 'some-symbol)

## that weird define keyword

in python there is a keyword to define functions. def you then provide it with a name and a set of arguments so on. variables are even simpler you just name it and use the asignment operator (also known as =) to name a value.

but notice how we use the same keyword in a very similar way. so lets have a look at what it does. <https://www.gnu.org/software/guile/manual/html_node/Definition.html> **use website**

this explains variables but not the function.. or does it. a concept in lisp is that functions are first class citizens this means we can pass them around like any other value. scheme takes this further by making you define them in the same way.

here all we are doing is binding the lambda or anonymous functions to the name func. the syntax you saw before provides a nicer syntax to do the same thing

we actually have a similar thing in python with the lambda keyword

* the actual way to define functions
* (define func  
   (lambda (x) (\* x 2)))  
    
  (func 2)
* 4
* python lambdas
* func = lambda x: x \* 2  
  return func(2)
* 4

(define lst '(23 3 1 4 2 4 69 420) )  
  
(define (function input)  
 (cond  
 ((or (null? input)  
 (null? (cdr input)))  
 input)  
 (else  
 (let ((pivot (car input))  
 (rest (cdr input)))  
 (append  
 (function  
 (filter (lambda (x) (< x pivot)) rest))  
 (list pivot)  
 (function  
 (filter (lambda (x) (>= x pivot)) rest)))))))  
  
(function lst)

## Whats cond?

cond takes multiple tests and an associated action and performs the first action which is true. its very similar to the if elif else tree but has much less syntax asociated with it.

notice that last element. that else clause. it looks like any other symbol or call. thats because for the most part it is. This meta programming aspect of scheme is not something we will discuss today but if you are interested I would recommend looking up scheme macros. in a word its a way of defining syntactic structures like the ones we see here within the language itself

x = 2  
if x == 0:  
 return "zero"  
elif x == 1:  
 return "one"  
elif x == 2:  
 return "two"  
else:  
 return "you do a lot of counting"

(define x 2)  
  
(cond ((= x 0)  
 "zero")  
 ((= x 1)  
 "one")  
 ((= x 2)  
 "two")  
 (else ;; always evaluates to true  
 "you do a lot of counting"))

(define lst '(23 3 1 4 2 4 69 420) )  
  
(define (function input)  
 (cond  
 ((or (null? input)  
 (null? (cdr input)))  
 input)  
 (else  
 (let ((pivot (car input))  
 (rest (cdr input)))  
 (append  
 (function  
 (filter (lambda (x) (< x pivot)) rest))  
 (list pivot)  
 (function  
 (filter (lambda (x) (>= x pivot)) rest)))))))  
  
(function lst)

## or?

or takes a list of symbolic expressions that evaluate to true or false and returns true if one of them returns true. this is like the or keyword in python but means you don't need to chain a lot of or calls to do the same thing. there is another function called and which will return false if any of the functions return false.

return False or False or True

True

(or #f #f #t)

#t

## let

this is the final scheme construct we will be discussing let allows us to bind values to names within the binding itself. this is like creating a variable in a function in python. it can only be accessed in that function. its the same for let bindings. they can only be accessed in the body of the binding. this is useful for a few reasons. one it makes

## the actual body of this function

# the same code in other languages

## haskell

quicksort :: Ord a => [a] -> [a] -- take any type that can be ordered  
quicksort [] = []  
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)  
 where  
 lesser = filter (< p) xs  
 greater = filter (>= p) xs  
  
-- quicksort [2,3,1]

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |

## ruby

def quicksort (arr)  
 return arr if arr.empty?  
  
 pivot, \*rest = arr  
  
 (quicksort (rest.filter { |x| x < pivot })).append(  
 [pivot].append(  
 quicksort (rest.filter { |x| x >= pivot }))).flatten  
end  
  
quicksort([2,3,1])

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |

# learning to learn

# any questions