
COMPUTER SCIENCE 61A

October 30th, 2013

In the next part of the course, we will be working with the **Scheme** programming language. In addition to learning how to write Scheme programs, we will eventually write a Scheme interpreter in Project 4.

Scheme is a dialect of the **Lisp** programming language, a language dating back to 1958. The popularity of Scheme within the programming language community stems from its simplicity – in fact, previous versions of CS 61A were taught in the Scheme language.

1 The Scheme Interpreter

Like **Python**, the **Scheme** language also has an interactive interpreter. On your class accounts, you can access this by typing in `stk`.

```
nova [416] ~ # stk
Warning: DISPLAY variable is not set. Graphics will NOT be initialized.
Welcome to the STk interpreter version 4.0.1-ucbl.3.6 [SunOS-5.10-sun4u]
Copyright (c) 1993-1999 Erick Gallesio - I3S - CNRS / ESSI <eg@unice.fr>
Modifications by UCB EECS Instructional Support Group
Questions, comments, or bug reports to <inst@EECS.Berkeley.EDU>.
STk>
```

Keep this one thing in mind: **Everything is an expression.**

2 Primitives

Scheme has a certain set of atomic primitive expressions. What atomic means is that these expressions cannot be divided up, or at least, not usually created out of smaller

expressions. The ones we'll use the most are ones representing numbers (i.e., integers and floats), the two boolean values and symbols. Scheme also has strings but we'll only use them a little.

```
STk> 123
123
STk> 123.123
123.123
STk> 'a ; this is a symbol
a
STk> #t
#t
STk> #f
#f
STk> "asdf"
"asdf"
```

You'll notice that numbers work pretty much the same in **Scheme** as they do in **Python**. Also, instead of `True` and `False`, we have `#t` and `#f`. Before explaining symbols though, let's look at how we define variables:

```
STk> (define a 3)
a
STk> a
3
```

You'll notice this is a lot different. In **Scheme**, you need to have parentheses and inside, have `define`, then the name of the variable and then the expression (which first gets evaluated) that you want bound to the name. Notice that the space between each of the parts (except the parens) matters. Also, notice that to refer back to `a`, I just typed in `a` just as in **Python**.

The thing is- `a` is a symbol. When you type things into the interpreter, **Scheme** will evaluate it. The rule for evaluating a symbol is to get the value bound to that symbol. This is one difference between strings and symbols- symbols don't evaluate to themselves. However, as you saw above, when you type in `'a`, you get `a`. This is because when you use the single quote, you're telling **Scheme** not to follow the normal rules of evaluation and just have the symbol return as itself. You'll also notice that you can use the single quotes on integers and floats, and the boolean values also. However they're unnecessary because those evaluate to themselves. Finally, let's revisit what you wrote when you did `(define a 3)`. Both `define` and `a` are symbols!

2.1 Questions

1. What gets printed?

```
STk> (define a 1)
?  
STk> a  
?  
STk> (define b a)  
?  
STk> b  
?  
STk> (define c 'a)  
?  
STk> c  
?
```

Solution:

```
STk> (define a 1)
a  
STk> a  
1  
STk> (define b a)  
b  
STk> b  
1  
STk> (define c 'a)  
c  
STk> c  
a
```

3 Evaluating Function Calls and Special Forms

Now just defining variables and printing out primitives isn't very useful. You want to call functions too:

```
STk> (+ 1 2)
3  
STk> (- 2 3)
-1
```

```
STk> (* 6 3)
18
STk> (/ 5 2)
2.5
STk> (+ 1 (* 3 4))
13
```

3.1 Functions

Now you might notice that **Scheme** does function calls differently. In **Scheme**, to call a function, you give the symbol for the function name first, then you give the arguments (remember the spaces). But just as in Python, what you do is you evaluate the operator (the first expression to the right of the `()`), then you evaluate each of the arguments and then apply those evaluated arguments to the thing returned by the first expression. So when you evaluate `(+ 1 2)`, you evaluate the `+` symbol which is bound to a built-in addition function, then you evaluate 1 and 2. Finally, you apply 1 and 2 to the function value bound to `+`.

Some important functions you'll want to use are:

- `+`, `-`, `*`, `/`
- `eq?`, `=`, `>`, `>=`, `<`, `<=`

3.2 Questions

1. What do the following return?

- `(+ 1)`
- `(* 3)`
- `(+ (* 3 3) (* 4 4))`
- `(define a (define b 3))`

Solution:

- 1
- 3
- 25
- a

3.3 Special Forms

However, there are certain things that look like function calls that aren't. These are called special forms and have their own rules for evaluation. You've already seen one- `define` where, of course, the first argument can't be evaluated (or else it'd search for unbound variables!). Another one we'll use for this class is `if`.

An `if` expression looks like this: `(if <CONDITION> <THEN> <ELSE>)` where `<CONDITION>`, `<THEN>` and `<ELSE>` are expressions. How it gets evaluated however is that first, `<CONDITION>` is evaluated. If it evaluates to `#f`, then `<ELSE>` is evaluated. Otherwise, `<THEN>` is evaluated. Everything that is not `#f` is a "true" expression.

```
STk> (if 'this-evaluates-to-true 1 2)
1
STk> (if #f (/1 0) 'this-is-returned)
this-is-returned
```

There are also special forms for the boolean operators which exhibit the same short circuiting behavior that you see in **Python**. The return values are either the value that lets you know the expression evaluates to a true value or `#f`.

```
STk> (and 1 2 3)
3
STk> (or 1 2 3)
1
STk> (or #t (/ 1 0))
#t
STk> (and #f (/1 0))
#f
STk> (not 3)
#f
STk> (not #t)
#f
```

3.4 Questions

1. What does the following do?

```
STk> (if (or #t (/ 1 0)) 1 (/ 1 0))
?
STk> (if (> 4 3) (+ 1 2 3 4) (+ 3 4 (* 3 2)))
?
STk> ((if (< 4 3) + -) 4 100)
?
```

Solution:

```

STk> (if (or #t (/ 1 0)) 1 (/ 1 0))
1
STk> (if (> 4 3) (+ 1 2 3 4) (+ 3 4 (* 3 2)))
10
STk> ((if (< 4 3) + -) 4 100)
-96

```

4 Lambdas, Environments and Defining Functions

Scheme has lambdas too! In fact, lambdas are more powerful in **Scheme** than in **Python**. The syntax is `(lambda (<PARAMETERS>) <EXPR>)`. Like in **Python**, lambdas are function values. Likewise, in **Scheme**, when a lambda expression is called, a new frame is created where the symbols defined in the `<PARAMETERS>` section are bound to the arguments passed in. Then, `<EXPR>` is evaluated under this new frame. Note that `<EXPR>` is not evaluated until the lambda value is called.

```

STk> (define x 3)
x
STk> (define y 4)
y
STk> ((lambda (x y) (+ x y)) 6 7)
13

```

Like in **Python**, lambda functions are also values! So you can do this to define functions:

```

STk> (define square (lambda (x) (* x x)))
square
STk> (square 4)
16

```

You might notice that this is a little tedious though. Luckily **Scheme** has a way out-define:

```

STk> (define (square x) (* x x))
square
STk> (square 5)
25

```

When you do `(define (<FUNCTIONNAME> <PARAMETERS>) <EXPR>)`, **Scheme** will automatically transform it to `(define <FUNCTIONNAME> (lambda (<PARAMETERS>)`

<EXPR>) for you. In this way, lambdas are more foundational to **Scheme** than they are to **Python**.

There is also another special form based around `lambda-let`. The structure of `let` is as follows:

```
(let ( (<SYMBOL1> <EXPR1>)
      ...
      (<SYMBOLN> <EXPRN>) )
  <BODY> )
```

This special form really just gets transformed to:

```
( (lambda (<SYMBOL1> ... <SYMBOLN>) <BODY>) <EXPR1> ... <EXPRN>)
```

You'll notice that what `let` does then is bind symbols to expressions. For example, this is useful if you need to reuse a value multiple times, or if you want to make your code more readable:

```
(define (sin x)
  (if (< x 0.000001)
      x
      (let ( (recursive-step (sin (/ x 3))) )
          (- (* 3 recursive-step)
              (* 4 (expt recursive-step 3))))))
```

4.1 Questions

1. Write a function that calculates factorial. (Note how you haven't been told any methods for iteration.)

```
(define (factorial x)
  )
```

Solution:

```
(define (factorial x)
  (if (< x 2)
      1
      (* x (factorial (- x 1)))))
```

2. Write a function that calculates the Nth fibonacci number

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Solution:

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

5 Pairs and Lists

So far, we have lambdas and a few atomic primitives. How do we create larger more complicated data structures? Well, the most important data-structure from which you'll build most complex data structures out of is the `pair`. A pair is an abstract data type that has the constructor `cons` which takes two arguments, and it has two accessors `car` and `cdr` which get the first and second argument respectively. `car` and `cdr` don't stand for anything really now but if you want the history go to "http://en.wikipedia.org/wiki/CAR_and_CDR"

```
STk> (define a (cons 1 2))
a
STk> a
(1 . 2)
STk> (car a)
1
STk> (cdr a)
2
```

Note that when a `pair` is printed, the `car` and `cdr` element are separated by a period.

A common data structure that you build out of pairs is the list. A list is either the empty list whose literal is `'()`, also known as `nil`, another primitive, or it's a `cons` pair where the `cdr` is a list. (Note the similarity to `Rlists`!)

```
STk> '()
()
STk> nil
```



```
()  
STk> (cons 1 (cons 2 nil))  
(1 2)  
STk> (cons 1 (cons 2 (cons 3 nil)))  
(1 2 3)
```

Note that there are no dots here. That is because when a dot is followed by the left parenthesis, the dot and the left parenthesis are deleted; when a left parenthesis is deleted, its matching right parenthesis is deleted also. You can check if a list is nil by using the `null?` function.

A shorthand for writing out a list is:

```
STk> '(1 2 3)  
(1 2 3)  
STk> '(define (square x) (* x x))  
(define (square x) (* x x))
```

You might notice that the return value of the second expression looks a lot like **Scheme** code. That's because **Scheme** code is made up of lists. When you use the single quote, you're telling **Scheme** not to evaluate the list, but instead keep it as just a list.

This is why **Scheme** is cool. It can be defined within itself.

5.1 Questions

1. Define `map` where the first argument is a function and the second a list. This should work like **Python's** `map`.

```
(define (map fn lst)
```

```
)
```

Solution:

```
(define (map fn lst)  
  (if (null? lst)  
      '()  
      (cons (fn (car lst)) (map fn (cdr lst)))))
```

2. Define `reduce` where the first argument is a function that takes two arguments, the second a default value and the third a list. This should work like **Python's** `reduce`.

```
(define (reduce fn s lst)

)
```

Solution:

```
(define (reduce fn s lst)
  (if (null? lst)
      s
      (fn (car lst) (reduce fn s (cdr lst)))))
```

3. Fill out the following to complete an abstract type for binary trees:

```
(define (make-btree entry left right)
  (cons entry (cons left right)))

(define (entry tree)

)

(define (left tree)

)

(define (right tree)

)
```

Solution:

```
(define (entry tree)
  (car tree))

(define (left tree)
  (car (cdr tree)))

(define (right tree)
  (cdr (cdr tree)))
```

4. Using the above definition, write a function that sums over the binary tree. For our purposes, assume that if there is no left or if there is no right branch, the values for those should return 0.

```
(define (btree-sum tree)
```

```
)
```

Solution:

```
(define (btree-sum tree)
  (if (null? tree)
      0
      (+ (entry tree)
         (btree-sum (left tree))
         (btree-sum (right tree)))))
```

5.2 Extra Questions

1. Write a scheme function that when given an element, a list, and a position, inserts the element into the list at the position.

```
(define (insert element lst position)
```

Solution:

```
(if (= position 0)
    (cons element lst)
    (cons (car lst) (insert element (cdr lst) (- position 1))
```

```
)
```

2. Write a scheme function that when given a list, such as (1 2 3 4), duplicates every element in the list (i.e. (1 1 2 2 3 3 4 4)).

```
(define (duplicate lst)
```

Solution:

```
(if (null? lst)
    lst
    (cons (car lst) (cons (car lst) (duplicate (cdr lst)))))

)
```