CIS 343 - Structure of Programming Languages

Nathan Bowman

Based on slides by Ira Woodring

Scheme (Follows the Sebesta Text Chapter 15)

Scheme

The first functional programming language was LISP, created at MIT in 1959. LISP has changed a great deal over the years though, adding imperative features, static typing, and other features. It is not a very pure functional language these days, though it is quite efficient and still in use.

Today there are two main versions of LISP in use, CLISP and Scheme. We will use MIT-Scheme in this class.

In the beginning LISP only had two types, **atoms** and **lists** (LISP stood for **Lis**t **P**rocessor). The following is a list:

```
(list 'apple 'orange 'banana 'grape)
(apple orange banana grape)
```

This list is made of 4 atoms.

A more complicated list might be

```
(list (list 'fuji 'smith 'macintosh) (list 'mandarin 'navel) 'ban

((fuji smith macintosh) (mandarin navel) banana ((seeded seedless
```

Which is made of various combinations of lists and atoms.

Notice what we did to create these lists - we called a function named `list`. In LISP we call functions using **prefix notation**. This means that the operator comes first (here the operator is the function name `list`).

```
(list element_1 ... element_x)
```

Similarly, we could perform addition as such:

```
(+ 20 22)
42
```

On the EOS system you can start the `mit-scheme` interpreter as follows:

```
rlwrap mit-scheme
```

You may even want to add an alias to your ~/.bashrc file:

```
alias mit-scheme="rlwrap mit-scheme"
```

`rlwrap` is a readline wrapper. It allows certain programs that don't accept arrow keys for history (such as mit-scheme or sqlplus) to have those facilities. You don't have to use `rlwrap` but it will make using mit-scheme much easier.

Once in the interpreter you will see prompts like the following:

```
1 ]=>
```

This indicates that you are in the REPL loop (**R**ead **E**valuate **P**rint **L**oop). It will read an evaluate expressions on and on forever (or until it is closed). Though this looks different, it is no different than being in a Python or Ruby interpreter.

Also like Python or Ruby, files can be loaded.

```
1 ]=> (load "fib.scm")
;Loading "fib.scm"... done
;Value: fib

1 ]=> (fib 7)

;Value: 13
```

Scheme has many built-in functions for dealing with numbers.

+, -, *, /, SQRT, LOG, SIN, MAX, MIN, MODULO, and others are provided.

Of note is that *some* functions allow multiple parameters. For instance, + and * take an arbitrary number of parameters and compute them all with the given function. It is up to you to determine the usage of each.

Remember: everything in functional programming comes down to functions

It is essential that we are able to create our own functions in a Scheme program. Scheme functions are declared with our old friend `lambda`

```
(lambda (x) (* x x x))
```

The syntax is

```
(lambda (arg1 arg2 ...) (function-body))
```

How can we call our new function? Just like any other

```
1 ]=> ((lambda (x) (* x x x)) 8)
;Value: 512
```

This is the same syntax as usual, but the entire `lambda` (including parentheses) is the function

```
(fun args)
(fun 8)
(   (lambda (x) (* x x x))   8)
```

Note that lambda expressions can take an arbitrary number of parameters:

```
(
        (lambda (a b c d)
                (+ (* a b) (* c d))
        )
1 2 3 4)
```

It is useful for us to define nameless functions at times, but often we want to be able to call upon a function more than once. In this case we need to provide a binding for the function (a name) so that we can call it again later.

The `DEFINE` keyword allows us to create bindings. This is not just for functions -- this is how we bind any name to any value.

```
]=> (define pi 3.14159)
]=> pi
;Value: 3.14159
```

```
(define pi 3.14159)
```

"But wait! You said functional languages don't have variables."

`pi` is not a variable -- it is a name for a specific value. It will not change throughout execution, the same way that actual math `pi` is a name that always means the same thing

"Oh, so you can't do..."

```
(define pi 3.14159)
(define pi 3)
```

...Sadly, you can, because Scheme is not a pure functional language.

But, 1) don't do that because it is bad, and 2) definitely don't do that in this class or you will lose points

Similarly, avoid other imperative constructs such as `set!`, `do`, and `for-each`

We can now create functions with `lambda` and bind names with `define`

These can be combined in a straightforward way to create a named function

```
(define area-circle (lambda (radius) (* 2 pi radius)))

(area-circle 1)
;Value: 3.14159
```

Because creating a named function is so common, there is "syntactic sugar" to do it more conveniently

```
(define (area-circle radius) (* 2 pi radius))
```

More generally:

```
(define (function-name arg1 arg2 ...) (function-body))
```

```
(define area-circle (lambda (radius) (* 2 pi radius)))
```

```
(define (area-circle radius) (* 2 pi radius))
```

These two lines do the exact same thing. The first makes more sense in the larger context of the language -- it is very clear what is happening.

The second is more convenient and idiomatic, so you should use it. But, be aware of what is really going on.

Function composition is critical to effective functional programming

We can combine functions by creating *functions that return functions*

`lambda` gives us everything we need, and `define` helps us keep the result around

This is an example from "Simply Scheme" by Harvey and Wright

https://people.eecs.berkeley.edu/~bh/ssch9/lambda.html

Step 1: Function that adds

```
(+ 1 3)
```

Step 2: Function that adds specific number

```
(define (add-three number)
  (+ number 3))
```

Step 3: Function that creates function that adds specific number

```
(define (make-adder num)
  (lambda (x) (+ x num)))
```

```
((make-adder 3) 10)

;Value: 13
```

```
(define add-five (make-adder 5))
(add-five 1)

;Value: 6
```

Particular example is trivial, but general pattern is extremely powerful

You now know how to

- start and run Scheme
- create functions (named or unnamed)
- call functions
- bind names to values
- compose functions into new functions