CIS 343 - Structure of Programming Languages

Functional Programming Languages and Scheme Introduction

(Follows the Sebesta Text Chapter 15)

Overview

- Imperative languages are all based on the von Neumann architecture. These are by far the most used languages.
- Instead of the von Neumann architecture, functional languages are based on a mathematical model of computing.

Overview

There have been many important computer scientists that have suggested functional programming might be a better alternative to imperative programming.

- John Backus, for instance, was a proponent of these languages.
- In 1977 he won the ACM Turing Award for his work on Fortran.
- With this award is given an opportunity to present a lecture to the community. He lectured on the merits of functional programming. He noted that this paradigm:

Overview

- Lacks side effects
- Isn't affected by context
- (Due to the above mentioned reasons) Is easier to understand during and after development

Overview

Imperative languages involve the idea of state. As commands are executed state changes.

- The solution to a problem is the memory's final state.
- This is extremely hard to read, as we need to keep track of variables' state as we peruse the code.
- This is not a problem in functional programs, as they have no state.
    - Well... in an ideal world they wouldn't. They try to minimize state as much as possible.

Overview

Remember that a mathematical function maps members from a domain to a range.

Some mappings are as follows:

```
y = x^2 x∃ℤ

z^2 = x^2 + y^2 for x∃ℝ, y∃ℝ
```

Or

Overview

z = B[x][y]

	y				
x	1	2	3	4	5
	2	4	6	8	10
	3	6	9	12	15
	4	8	12	16	20
	5	10	15	20	25

Overview

Note that whether we are dealing with a function described as an expression or one described as a table, pass one or multiple parameters from the domain.

- A value from the range is returned.
- There are no other possibilities.

Overview

The big thing we need to remember when talking about functional languages is that there are no variables (as there is no state).

- Therefore when we pass a value from a domain we should always get the same element from the range.
- The range value that is returned is **not** dependent on any other program state.
- We are mapping parameters to values, **not** producing a value through some sequence of operations.

Overview

To mirror our book as closely as possible we are going to use the following syntax when defining functions:

- function name first
- list of parameters in parentheses
- a mapping Expression

For instance (from the book):

```
cube(x) ≡ x ∗ x ∗ x
```

Overview

Notice that I'm not saying that `cube(x)` is equal to a value. I'm using the '≡' character.

- This character should be read (in this context) as "defined as".
- In imperative languages we would see an expression such as `cube(x) ≡ x * x * x` and imagine `x` to be a variable - that is we imagine `x` to be a value currently stored in some memory location.

Overview

We SHOULD NOT view `x` this way in functional languages.

- In a functional language `x` stands for a value that is bound; whatever it stood for cannot change during evaluation (it is a constant). We then view

```
cube(x) ≡ x * x * x
```

as

```
cube(2.0) ≡ 2.0 * 2.0 * 2.0
```

(ie, if the parameter passed was 2.0).

Overview

What we are talking about here is a different mental model of computation that clears away some of our preconceptions of what programming has to be.

- We may think that we already view the expression as `cube(2.0) ≡ 2.0 * 2.0 * 2.0` when `cube(2.0)` is called, but from my experience we don't.
- We have been "burned" so many times while debugging code that in the backs of our minds we know we can expect changes in state to occur. Forget that now.

Overview

While studying functional languages you will come across the concept of the **lambda expression**.

- You have probably seen these in other languages such as Python, JavaScript, or even Java and C++ (though syntax differes greatly across languages).
- This is based on the work of Alonzo Church, who was working on a formal model of functions called **lambda calculus**.
- He was interested in the foundations of mathematics. Later it was shown that the lambda calculus could represent any Turing machine.

Overview

Lambda expressions are nameless functions. In lambda calculus our cube function would be

$\lambda(x) = x * x * x$

In Python we expressed it as:

```
g = lambda x: x*x*x
```

Overview

The original Lambda Calculus does not allow for multiple parameters to be passed. Instead, we would **curry**. This means that we apply the first parameter, then return a function that can accept the second.

For instance:

```
def add(a, b):
  return a + b
```

Would become

```
def add(a):
  return lambda b: a + b

add(2)(7)
9
```

Overview

To understand functional programming we must understand the concept of **higher-order** functions.

- These are functions that can take other functions as parameters.
- This allows us to solve problems via the **composition of functions**.

We've seen the composition of functions before in math, i.e. if we have f(x) and g(x) we may compute f(g(x))

Overview

As an example, let's try to write an absolute value function as the composition of two functions:

```
f(x) = x * x
g(x) = sqrt(x)
abs(x) = g(f(x)) = sqrt(x * x)
```

Overview

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.

Overview

In the beginning LISP only had two types, **atoms** and **lists** (LISP stood for **List** **P**rocessor).

The following is a list:

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

This list is made of 4 atoms.

Overview

A more complicated list might be

```
(list (list 'fuji 'smith 'macintosh) (list 'mandarin 'navel) 'banana (list (list 'seeded 'seedless) 'white 'red' blue)))

((fuji smith macintosh) (mandarin navel) banana ((seeded seedless) white red blue))
```

Which is made of various combinations of lists and atoms.

Overview

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)
```

Overview

Similarly, we could perform addition as such:

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

Overview

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

```
rlwrap /usr/local/mit-scheme/bin/mit-scheme
```

Overview

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

```
alias mit-scheme="rlwrap /usr/local/mit-scheme/bin/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.

Overview

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

```
]=>
```

- 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.

Overview

Also like Python or Ruby, files can be loaded.

```
;Loading "fib.scm"... done
;Value: fib

1 ]=> (fib 7)

;Value: 13
```

Overview

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

+, -, *, /, 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. Read the docs!

Overview

It is essential that we are able to create our own functions in a Scheme program. To do so, we can either create an anonymous or a named function. An anonymous function is created with the LAMBDA keyword:

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

This example is essentially useless, as we have passed no parameters and we have not bound the function to some identifier. We could though, do this:

Overview

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

Here we bound x with the value 8.

Overview

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

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

Overview

- 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 created bindings. We can bind more than just functions, however. For instance, we could

```
(define pi 3.14159)
```

Overview

Once we have created a binding we can use it:

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

This particular binding (a function binding) makes use of the previously bound `pi` (as well as the bound parameter radius).

Overview

This previous code may have looked a bit confusing, so let's examine it further.

A function definition has the following parts:

```
(define (function_name param1 ... paramX) (function_body))

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

Overview

A problem exists when it comes to the interactions of a functional program and a user-namely that of input. Input exists to change state. As functional languages do not like to change state, the very idea of input is antithetical to functional programming. Different languages deal with this in different ways. LISP (and Scheme) have adopted imperative procedures over the years to accommodate. This means they are not "pure" functional languages.

- Other languages still need to allow state changes, so they isolate that code.

Overview

Predicate functions are functions that return Boolean values. Some predicate functions provided by Scheme are

=, eq?, NOT, >, <, >=, <=, EVEN?, ODD?, ZERO?

Functions that return #t or #f (true or false) but contain words end with the '?' character.

If a list is being examined, a non-empty list returns #t and an empty list returns #f.

Overview

Control flow in Scheme is accomplished via `IF`, `COND`, and `CASE` (branching), and recursion (repetition).

IF:

```
(IF predicate then_expression else_expression)
(if (eq? n 1701)
  #t  ;; What to do if the statement is true.
  #f  ;; What to do if the statement evaluated to false.
)
```

Overview

A more complicated example:

```
(define (fib n)
  ;; Calculate the nth Fibonacci number recursively
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Overview

COND is used for multiple conditionals. For instance:

```
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))
```

Notice there are multiple predicates here.

Overview

CASE compares *keys* to *objects*:

```
(case (car '(c d))
      ((a e i o u) 'vowel)
      ((w y) 'semivowel)
      (else 'consonant))
```

Overview

When evaluating statements we need to be careful we are conveying the correct semantics. For instance:

```
1 ]=> (list a b c d)

;Unbound variable: d
;To continue, call RESTART with an option number:
; (RESTART 3) => Specify a value to use instead of d.
; (RESTART 2) => Define d to a given value.
; (RESTART 1) => Return to read-eval-print level 1.

2 error>
```

What is wrong here?

Overview

We are calling a function with parameters. The parameters must be bound before the function can be called.

`a`, `b`, `c`, and `d` don't have any bindings.

What we really want to do is this:

```
(list (quote a) (quote b) (quote c) (quote d))

;;; OR THE SHORTCUT:
(list 'a 'b 'c 'd)
```

Overview

The QUOTE and ' shortcut are functions that cause the parameter not to be evaluated, but to instead be used as is.

- We must be very careful when we are programming that we QUOTE parameters we don't wish to be evaluated.

Overview

Since LISP was created to process lists, it makes sense that there are quite a few built-in functions for the manipulation of lists.

Two of the most common are `CAR` and `CDR`.

`CAR` returns the first element of a list, `CDR` returns all but the first.

Overview

Don't try to assign meaning to these names; they are holdovers from instructions on the IBM 704 machines LISP was originally created for. They have to do with accessing parts of memory locations.

Overview

```
2=> (define l (list 'apple 'banana 'grape 'orange))

;Value: l

2=> (car l)

;Value: apple

2=> (cdr l)

;Value 19: (banana grape orange)
```

Overview

To get the second element in a list, we could do the following:

```
(car (cdr l))
```

We could even create a function (from book):

```
(define (second a_list) (car (cdr a_list)))
(second l)
```

Overview

There are shortcuts as well:

```
CAAR = (car (car E))
CADR = (car (cdr E))
```

others exist as well.

Overview

The `LET` statement is used to create a local binding. Once the `LET` function has created a binding in some local scope the binding cannot be rebound.

From the book, an example to compute quadratic roots:

```
(define (quadratic_roots a b c)
  (let (
    (root_part_over_2a
                (/ (sqrt (- (* b b) (* 4 a c))) (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
  )
  (list (+ minus_b_over_2a root_part_over_2a)
        (- minus_b_over_2a root_part_over_2a))
  )
)
```

We created two bindings and provided an expression inside of which we could use the named bindings.

Overview

The thing I see students do the most often (in regards to LET statements), is put the code for the expression they wish to use the bindings for inside of the LET statement. This is not imperative programming, we cannot do this:

```
(let ((pi (/ 22 7))))(write pi)
```

We must do this:

```
(let ((pi (/ 22 7)))(write pi))
```

(Notice the expression `(write pi)` is a parameter to the function)

Overview

A good understanding of recursion is essential in functional languages, as they lack iteration. For instance, if we want print all members of a list in an imperative language, we would do something like this:

```
for(Member m:list){
  print m;
}
```

In a functional language though, we use recursion:

```
(define l (list 'a 'b 'c 'd))

(define (print_elements a_list)
  (cond
    ((null? a_list))
    (else (write (car a_list))(print_elements (cdr a_list)))
  )
)

abcd
;Value: #t
```

Overview

You may have noticed above that the return value from the function was `#t`. We never explicitly stated this, so where did that come from?

Like Ruby, Scheme doesn't require a `return` statement to return a value. The final value in an expression is what is returned. So what happened above? The list was recursively broken into two parts - the first element, and the rest. The first element was printed, then we recursively printed the rest. When there were no more parts the `null?` function returned `#t`.

Coding

How could we change this code to print the list backward?

```
(define l (list 'a 'b 'c 'd))

(define (print_elements a_list)
  (cond
    ((null? a_list))
    (else (write (car a_list))(print_elements (cdr a_list)))
  )
)

abcd
;Value: #t
```

```scheme
(define l (list 'a 'b 'c 'd))

(define (print_elements a_list)
  (cond
    ((null? a_list))
    (else (print_elements (cdr a_list))(write (car a_list)))
  )
)

dcba
;Unspecified return value
```

Why unspecified return value?

The last line that is run is not the null check anymore. It is a write statement! How could we fix it to return a #t or #f value?

If we needed it to return #t or #f we could do this:

```
(define (print_elements a_list)
  (cond
    ((null? a_list))
    (else (print_elements (cdr a_list))(write (car a_list))#t)
  )
)
```

What will this code do?

```scheme
(define l (list 1 2 3 4))
(define (new_func a_list)
  (cond
    ((null? a_list) 0)
    (else (+ (car a_list)(new_func (cdr a_list))))
  )
)
(new_func l)
```

Practice

Logon to EOS. Start mit-scheme.

> Remember to rlwrap it!

```
rlwrap /usr/local/mit-scheme/bin/mit-scheme
```

Practice

Problem 1 (the classic):

Write a function called fact that takes a single parameter. The function should return the value of the factorial of the parameter.

Don't look this up. Struggle with it for a few minutes. Try things and see what happens.

```scheme
(define (fact n)
  (cond
    ((<= n 1) 1)
    (else (* n (fact (- n 1))))
  )
)
```

Practice

Problem 2;

Write a function that reverses a list.

```scheme
(define (r a_list)
  (cond
    ((null? a_list) #t)
    (else (r (cdr a_list))(write (car a_list))#t)
  )
)
```

However, there does exist a `reverse` function.

Scheme does allow the use of strings. Strings can be created in a variety of ways, the simplest being

```
"ira"
```

Scheme views characters in a string as ASCII values, and represents them as such:

```
#\i #\r #\a
```

We can make a string into a list, or a list into a string via the `list->string` or `string->list` functions:

```
(define name "ira")
(string->list name)
;Value 40: (#\i #\r #\a)

(define l (list #\i #\r #\a))
(list->string l)
;Value 43: "ira"
```

Practice

Problem 3:

Knowing what you now know, write a function that reverses a string.

```
(define (flip str)
  (list->string (reverse (string->list str))))
```

Alternatively you could have called your own reverse function.

Much like Java we can't compare strings with the `eq?` function. Instead we need to use `string=?`.

```
1 ]=> (eq? "ira" "ira")

;Value: #f

1 ]=> (string=? "ira" "ira")

;Value: #t
```

Practice

Problem 4:

Write a function that checks if a string is a palindrome or not.

It should return either `#t` or `#f`.

Make use of what we already have.

```
(define (pal str)
  (string=? str (flip str))
)
```

Practice

Problem 5:

Write a function that returns the *i*th value in a list.

```scheme
(define (findi my_list i)
  (cond
    ((eq? i 0) (car my_list))
    (else (findi (cdr my_list) (- i 1)))
  )
)
```