Computer Science 433
Programming Languages
The College of Saint Rose
Fall 2014

Topic Notes: Functional Programming with Scheme

Our next topic is to look at a different programming paradigm: *functional programming*. Recall from the first week of class, the following brief description of functional programming languages:

"The primary mechanism for computing in a functional language is, unsurprisingly, the application of (often recursive) functions to parameters.

- pure functional programming has no variables or assignment statements!
- very convenient in some contexts
- not well-suited for others
- functional languages are usually interpreted rather than compiled"

Functional languages generally have a much simpler and "cleaner" syntax than object-oriented/imperative languages. This has led to the adoption of functional languages as a first language for introductory computer science courses at some schools.

The ideas of functional programming have gained popularity in recent years with the growing use of languages like Scala and Clojure, in part due to he fact that concurrency is now a very common feature of programs.

From `http://clojure.org/rationale`: "Customers and stakeholders have substantial investments in, and are comfortable with the performance, security and stability of, industry-standard platforms like the JVM. While Java developers may envy the succinctness, flexibility and productivity of dynamic languages, they have concerns about running on customer-approved infrastructure, access to their existing code base and libraries, and performance. In addition, they face ongoing problem dealing with concurrency using native threads and locking. Clojure is an effort in pragmatic dynamic language design in this context. It endeavors to be a general-purpose language suitable in those areas where Java is suitable. It reflects the reality that, for the concurrent programming future, pervasive, unmoderated mutation simply has to go."

We will examine functional programming with an older, but still very relevant language called Scheme.

Mathematical Basis of Functional Languages

Those who have taken some college mathematics have probably seen the idea of *composite functions*:

$$F(x) \circ G(x) = F(G(x))$$

We would read this "F follows G" or "F of G of x". It means we form a new function by first applying G then F.

In functional programming, we combine often simple functions to build more complex ones.

We will also see *higher-order functions* (or, *functional forms*), where a function receives functions as parameters and/or returns a function as a result.

This can be done in some conventional programming languages (passing of function pointers in C), but it is much more natural in a functional programming language.

Introduction to Scheme

The specific language we will study most in this context is called *Scheme*.

Scheme was developed from the successful functional language LISP by Sussman and Steele in 1975. The first Scheme compiler appeared in 1978. The most popular standard (R5RS) was developed in 1998.

There are a number of implementations of Scheme, one of which is called MIT/GNU Scheme, which is (unsurprisingly) developed at MIT.

We have it installed on mogul, and you can run it by issuing the `scheme` command.

Launching the program enters into MIT/GNU Scheme's "read-eval-print loop" (REPL) and we are issued a prompt. To start, we will just type in some examples at this prompt.

```
MIT/GNU Scheme running under GNU/Linux
Type `^C' (control-C) followed by `H' to obtain information about interrupts

Copyright (C) 2011 Massachusetts Institute of Technology
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Image saved on Tuesday November 8, 2011 at 10:45:46 PM
  Release 9.1.1     || Microcode 15.3 || Runtime 15.7 || SF 4.41
  LIAR/x86-64 4.118 || Edwin 3.116

1 ]=>
```

This prompt tells us that we are at "level 1" in the interpreter. That number can change as we interact with the system.

First, we can specify some numeric values to Scheme:

```
1 ]=> 1

;Value: 1
```

All that did was cause the number to be echoed back to us. But it does tell us something – the REPL evaluated our input as "1" and printed it back.

We can specify any integer or floating-point value in this fashion, and we should get our number back.

We can also specify boolean values, which in Scheme are specified by `#t` and `#f`

```
1 ]=> #f

;Value: #f

1 ]=> #t

;Value: #t
```

What if we try to evaluate a word?

```
1 ]=> proglang

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

2 error>
```

That seems much less successful. Scheme expected the word to be a variable name, and we have not defined any variable with that name.

We will not worry yet about the restart options, but note that the prompt has changed. The "2" tells us we are not operating at level number 2 because of the error. If all we want to do is to get back to level number 1, we can type "Ctrl-g" to move back a level.

However, we can echo text back if we give Scheme a string literal:

```
1 ]=> "Hello"

;Value 13: "Hello"
```

Having Scheme echo back values we type isn't especially exciting. Fortunately, we can ask it to compute things. Let's try some addition:

```
1 ]=> 2 + 2

;Value: 2

1 ]=>
;Value 13: #[arity-dispatched-procedure 13]

1 ]=>
;Value: 2

1 ]=>
```

That doesn't seem very good. We never got a "4" and we got some bizarre message in the middle. Scheme has interpreted our command as three separate commands, somehow evaluating "+" to 13.

The syntax of addition in Scheme is different. It is a *function* and functions in Scheme are specified as *lists*. They are in prefix notation inside of parentheses, where the first element of the list is the name of the function and the remaining list items are the parameters:

```
1 ]=> (+ 2 2)

;Value: 4
```

Better!

We can also use other operators and include more than 2 operands:

```
1 ]=> (- 6 3)

;Value: 3

1 ]=> (* 7 2 4)

;Value: 56

1 ]=> (/ 9 2)

;Value: 9/2

1 ]=> (/ 9 2.0)
```

```
;Value: 4.5

1 ]=> (+ 1 2 3 4 5)

;Value: 15
```

Furthermore, a function operand can itself be a function call:

```
1 ]=> (* (+ 4 9) (- 8 4))

;Value: 52
```

Earlier, we saw that when we typed a word, Scheme tried to interpret it as a variable. So we can set values of variables and use them in expressions.

```
1 ]=> (define temperature 56)

;Value: temperature

1 ]=> temperature

;Value: 56

1 ]=> (* temperature 4)

;Value: 224

1 ]=> (define temperature (- temperature 4))

;Value: temperature

1 ]=> temperature

;Value: 52
```

Consider this example:

```
1 ]=> (define radius 2)

;Value: radius

1 ]=> (define pi 3.14159)
```

```
;Value: pi

1 ]=> (define area (* pi radius radius))

;Value: area

1 ]=> area

;Value: 12.56636
```

If we change the value of `radius` at this point, should we expect the area to be updated?

```
1 ]=> (define radius 3)

;Value: radius

1 ]=> area

;Value: 12.56636
```

It doesn't update `area`, but why would we expect it to? If we had the following sequence in a Java program, how would it behave?

```
  radius = 2;
  area = Math.PI * radius * radius;
  System.out.println(area);
  radius = 3;
  System.out.println(area);
```

The `area` would only be updated again if we recomputed it.

Or... if we made area a method:

```
  radius = 2;
  System.out.println(area(radius));
  radius = 3;
  System.out.println(area(radius));
```

We can do the same here:

```
1 ]=> (define radius 2)

;Value: radius
```

```
1 ]=> (define (area) (* pi radius radius))

;Value: area

1 ]=> (area)

;Value: 12.56636

1 ]=> (define radius 3)

;Value: radius

1 ]=> (area)

;Value: 28.274309999999996
```

We have defined a function, albeit not a very nice one since it relies on two variables being defined. But we can see that when we `define` something in parentheses, we are defining it to be a function. Soon we will see that this is just a shorthand way of specifying this:

```
1 ]=> (define area (lambda () (* pi radius radius)))

;Value: area

1 ]=> (define pi 3.14159)

;Value: pi

1 ]=> (define radius 3)

;Value: radius

1 ]=> (area)

;Value: 28.274309999999996

1 ]=>
```

Getting around MIT/GNU Scheme

Before we continue with more Scheme, here are some handy things to know about using MIT/GNU Scheme.

To quit and return to the Unix prompt, issue the command

```
(exit)
```

We can have Scheme print messages for us to the console output with the `write` function.

Alternately, hit the `<Ctrl-d>` key at the prompt.

```
1 ]=> (write "Hello, World!")
"Hello, World!"
;Unspecified return value
```

Alternately, we can use `display`, and we will get our output without the quotes.

So we have the equivalent of `printf` or `System.out.println`.

We can also write programs in our favorite text editor and load them into MIT/GNU Scheme.

See Example:
`/home/cs433/examples/hello_scheme`

The example shows us another important feature: the `;` character is used to start a single-line comment in Scheme.

If our program is in a file `hello.scm`, we can load it in one of two ways: with a command-line parameter when we launch GNU/Scheme:

```
% scheme -load hello.scm
MIT/GNU Scheme running under MacOSX
Type ‘ˆC’ (control-C) followed by ‘H’ to obtain information about interrupt

Copyright (C) 2011 Massachusetts Institute of Technology
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Image saved on Thursday September 27, 2012 at 9:18:00 PM
  Release 9.1.1 || Microcode 15.3 || Runtime 15.7 || SF 4.41 || LIAR/C 4.11
  Edwin 3.116
;Loading "hello.scm"..."Hello, Scheme World!"
;... done

1 ]=>
```

In this case, our output is printed hidden away next to a message that we are loading the specified file.

We can also launch Scheme first, then use the `load` function.

```
1 ]=> (load "hello.scm")

;Loading "hello.scm"..."Hello, Scheme World!"
;... done
;Unspecified return value
```

Or if we wanted to have something in the file that defines some functions and variables so we can then make use of it:

See Example:
`/home/cs433/examples/area/area.scm`

This defines `pi` as a variable and `area` as a function. Then if we define `radius` appropriately and call the `area` function, it should produce the result we want.

```
1 ]=> (load "area.scm")

;Loading "area.scm"... done
;Value: area

1 ]=> (define radius 2.5)

;Value: radius

1 ]=> (area)

;Value: 19.6349375

1 ]=>
```

Next, we look back at the case where we attempted to use a value we hadn't defined. Specifically, let's see what happens if we load our `area` function when we haven't defined `radius`.

```
1 ]=> (area)

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

2 error>
```

We see Scheme has given us three options, in addition to a fourth option which is "ignore the problem and start working at the level 2 REPL".

Try each out, and see how Scheme lets us recover from errors in a variety of ways.

We can also use the `read` function to obtain a value from the terminal and use it however we'd like.

See Example:
`/home/cs433/examples/area/area_input.scm`

So we have basic I/O capabilities and we can define variables and functions, and call those functions.

However, this is not how we normally want to call a function – requiring that a particular variable is defined and has an appropriate value. We want to pass *parameters*. We can do just that:

See Example:
`/home/cs433/examples/area/area_function.scm`

This program defines the `area` function to have a single parameter named `radius`. We can then use that parameter inside the function.

Note that now, we can call `area` and pass it a parameter. In the program, that value is the return of the `read` function.

Once the program is loaded, we can call it with any parameters we wish.

Data Types in Scheme

Scheme supports a rich set of data types.

Numeric Data

We have seen that Scheme can manipulate integer, real (what most languages would call floating point), and boolean data.

It also includes some more unusual numeric types: rational and complex.

Scheme maintains an important property about the numbers it uses: their *exactness*:

See `http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/Exactness.html#Exactness`

In most languages, we know that some values (in particular, floating point values) are approximations. Scheme formalizes this so we can tell whether a given numeric value is exact or inexact.

Scheme provides functions that can tell us if a number is exact or inexact. These are a special class of functions called *predicates* that test their parameter for some condition and return a boolean result:

```
1 ]=> (exact? 1)

;Value: #t
```

```
1 ]=> (inexact? 1)

;Value: #f

1 ]=> (exact? (/ 1 3))

;Value: #t

1 ]=> (exact? (/ 1.0 3.0))

;Value: #f
```

There are a wide variety of operators we can use on numeric data as well.

See `http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/Numerical-operations.html#Numerical-operations` in the Scheme documentation for some examples.

The first several there are predicates that check whether a value is a valid instance of a given type.

Strings and Characters

We have already seen how we can deal with string constants in Scheme.

We can specify character constants (the equivalent of single-quoted char constants in C/C++/Java):

```
#\a
#\space
```

There are many operations on characters, most of which are not important for our purposes.

A few that may be of interest:

```
1 ]=> (char-upper-case? #\a)

;Value: #f

1 ]=> (char-lower-case? #\a)

;Value: #t

1 ]=> (char-alphabetic? #\Z)

;Value: #t
```

```
1 ]=> (char-numeric? #\x)

;Value: #f

1 ]=> (char-numeric? #\4)

;Value: #t
```

There are several functions we can use to construct and manipulate strings – we will go through some as seen on

`http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/Strings.html#Strings`

Lists

Lists are a fundamental data structure in Scheme, and its predecessor language, LISP.

An *atom* is an individual element of some primitive type.

A *list* is a collection of atoms and/or lists. Typically, a list is represents as a *pair* of items `<A, B>`, where `A` is the first item of the list (called the *car*), and `B` is the rest of the list (called the *cdr*). This is basically a linked list representation.

Before we look at some more detailed examples, note that we can specify symbols (usually words) without having Scheme attempt to evaluate them by placing a `’` in front of them.

```
1 ]=> ’x

;Value: x
```

This is a shorthand for

```
1 ]=> (quote x)

;Value: x
```

If we left off the `’` or `quote` function, we would get an error:

```
1 ]=> x

;Unbound variable: x
```

We can also use this to introduce lists into our Scheme programs:

```
1 ]=> ’(a b c)

;Value 2: (a b c)
```

Without the `’`, Scheme would try to evaluate and look for a function `a` and pass it parameters `b` and `c`.

See `http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/Lists.html#Lists`

Most of the terminology here is important.

Many but not all of the functions are ones we will make use of. Some to look at in particular:

`cons`, `xcons`, `car`, `cdr`, `caar` (and friends), `list`. Some extra examples of these:

```
1 ]=> (car ’(a b c))

;Value: a

1 ]=> (car ’(((a b) c) d (e)))

;Value 2: ((a b) c)

1 ]=> (cdr ’(a b c))

;Value 3: (b c)

1 ]=> (cdr ’(((a b) c) d (e)))

;Value 4: (d (e))

1 ]=> (cons ’this ’(is weird stuff))

;Value 5: (this is weird stuff)

1 ]=> (cons ’((goofy)) ’((pluto minnie) mickey))

;Value 6: (((goofy)) (pluto minnie) mickey)

1 ]=> (cons (car ’(a b c)) ’(big cow))

;Value 7: (a big cow)

1 ]=> (car (cons (cdr ’(not much doing)) ’(this part goes away)))
```

```
;Value 8: (much doing)

1 ]=> (car ’(cons ’a ’(b c)))

;Value: cons

1 ]=> (list ’a ’b)

;Value 9: (a b)

1 ]=> (append ’(a) ’(b))

;Value 10: (a b)

1 ]=> (list ’(a b (c)) ’d ’e)

;Value 11: ((a b (c)) d e)

1 ]=> (append ’(h (e) f)
              ’(i (j) k)
              ’(a b c))

;Value 12: (h (e) f i (j) k a b c)

1 ]=> (append () ’(a b c))

;Value 13: (a b c)
```

More functions to look at:

`make-list`, `iota`, `list?`, `length`, `null?`, `first` .. `tenth`, `sublist`, `list-head`, `list-tail`, `append`, the filtering functions `filter`, `remove`, and `partition`.

Mapping

One of the great powers of a functional language is the ability to *map* a function over a list.

For example, if we have lists of numbers, we can create a new list containing their elementwise sums:

```
1 ]=> (map + ’(1 2 3) ’(4 5 6) ’(7 8 9))

;Value 14: (12 15 18)
```

This can be done with list functions:

```
1 ]=> (map car '((alice has triangle hair) (dilbert wears a tie) (wally is

;Value 16: (alice dilbert wally)
```

Or we can apply a function we define to each element of a list:

```
1 ]=> (map (lambda (n) (* n 2)) '(7 23 1))


;Value 17: (14 46 2)
```

Writing Functions

We would like to be able to write our own functions to perform complex tasks, using Scheme's builtin functions as building blocks.

We saw before how to define a function. Let's do one with that will also let us practice with list operations. We want to write a function that will work like `cons`, but which will construct a list from 2 atoms and a list instead of 1 atom and a list.

See Example:
`/home/cs433/examples/cons2`

Conditionals

An essential building block for many tasks is the ability to make decisions – the conditional construct. We rely on this in our imperative languages, and it plays a key role in functional programming as well.

See: `http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/Conditionals.html#Conditionals`

Scheme provides a straightforward `if` function, as well as more general purpose `cond` and `case` functions. There are also a number of predicates (some of which we have seen) as well as the standard comparison functions for equality and inequality. We can also construct more complex boolean expressions with `not`, `and`, and `or`.

We proceed with some examples:

```
1 ]=> (< 2 4)

;Value: #t

1 ]=> (<= 3 3)
```

```
;Value: #t

1 ]=> (not #f)

;Value: #t

1 ]=> (if (< 4 5) ’x ’y)

;Value: x

1 ]=> (if (> 4 5) ’x ’y)

;Value: y

1 ]=> (if (< 4 5) ’ifpart)

;Value: ifpart

1 ]=> (if (> 4 5) ’ifpart)

;Unspecified return value

1 ]=> (cond ((= 1 2) ’nope)
            ((not (= 1 2)) ’yep)
)

;Value: yep

1 ]=> (cond ((equal? ’a ’b) ’cow)
            ((member ’a ’(c b a d)) ’cat)
            (#t ’dog)))

;Value: cat
```

Let’s practice using these to write some functions that can determine if a given list contains 2 elements. The function should work like this:

```
(elements2 ’(a b))
```

returns `#t` because its parameter is a list containing exactly 2 elements.

```
(elements2 ’(a b c))
```

returns `#f`, as its parameter is a list with 3 elements.

We will assume that the parameter passed in is in fact a list.

See Example:
`/home/cs433/examples/elements2`

Recursive Functions

Our next function demonstrates a cornerstone of functional programming: a *recursive* function.

What's everyone's favorite recursive function (or at least one of them)? The factorial, of course.

We will use this rule to calculate a factorial:

$$fact(n) = \begin{cases} 1 & \text{if } n < 2 \\ n \cdot fact(n-1) & \text{otherwise} \end{cases}$$

See Example:
`/home/cs433/examples/factorial`

How about a function that turns a list into a list of lists? We'll call it `listem`.

For example, if we call

```
(listem '(a b c))
```

we should get back

```
((a) (b) (c))
```

See Example:
`/home/cs433/examples/listem`

And next, a function to remove all instances of an element from a list and return a new list with those instances removed.

So on this call:

```
(remove_all 'a '(b a d a f a))
```

we should be returned the list

```
(b d f)
```

See Example:
`/home/cs433/examples/remove_all`

For another example, we will write a function that takes a list as its parameter and returns another list where any numbers in the list have been incremented.

If we call:

```
(add1_num '(8 bob (alice) 23 (3) dilbert))
```

would return

```
(9 bob (alice) 24 (3) dilbert)
```

But what if we want to find lists in any "sublists" so the above would become:

```
(9 bob (alice) 24 (4) dilbert)
```

And for a more complex call (we'll call this a "deep" function):

```
(add1_num_deep '(hi (1 2 a) (10 (20) ((30 31) 100)) deep))
```

would produce

```
(hi (2 3 a) (11 (21) ((31 32) 101)) deep)
```

See Example:
`/home/cs433/examples/add1_num`

For a function that has similar structure but a different functionality, we will extract all numbers from a (potentially nested) list.

```
(extract_numbers '(a b c 1 2 3 a s c))
```

would return

```
(1 2 3)
```

and

```
(extract_numbers '(hi (1 2 a) (10 (20) ((30 31) 100)) deep))
```

returns

```
(1 2 10 20 30 31 100)
```

See Example:
`/home/cs433/examples/extract_numbers`

We can then modify it to add the numbers instead of extract a list of numbers. Notice the fact that the operations to compute the result are all on numbers now (0 for base case, addition to build the result) instead of on lists (empty list for base case, `cons`/`append` to build the result).

But if we already have the `extract_numbers` function and want to implement `add_all_numbers`, there is a simpler way. The builtin function `apply` takes a function and a list as its parameters and calls that function with the list as its parameters.

We can use a recursive procedure to generate a list of prime numbers. One technique for generating lists of primes is called the Sieve of Eratosthenes. The idea is as follows. We start with a list of all of the numbers up to some upper limit, starting with 2. Our algorithm proceeds by working left to right through the list of numbers. For each number we encounter, we remove all of its multiples ("cross them out"). When we get to the end of the list, everything remaining is a prime.

See Example:
`/home/cs433/examples/sieve`

We can first look at a C implementation (iterative) then a Scheme implementation (recursive).

Our next example involves sorting a list of numbers. We will look at the insertion sort.

See Example:
`/home/cs433/examples/insertion_sort`

See the comments in the Scheme file for more.

Functions as parameters

But there is no need to write a function with such limited capabilities. This one can only work on lists of numbers and sort them in increasing order.

Let's instead build a more general insertion sort function that takes a comparison function in addition to the list to be sorted.

See Example:
`/home/cs433/examples/general_insertion_sort`

Again, see the comments in the file for details, but note the extra parameter to the sorting function and its "insert one" helper, and the use of that parameter as the name of a function to call when it comes time to compare two list elements to determine their relative order.

Armed with this general function, we can either use it directly by passing an appropriate function, or define some other functions that operate on different kinds of lists and sorts them in different ways.

Backtracking

Functional programming provides a convenient mechanism for implementation of *backtracking* algorithms. With a backtracking algorithm, we try different alternatives at each decision point, in turn, until we find a satisfactory solution.

A fundamental example here is maze running. If a choice to follow one path in the maze fails (leads to a dead end), we go back (backtrack) to the most recent decision point and try a different direction. If all directions from a given point have led to dead ends, we backtrack further to a prior decision point and try a different alternative.

In Scheme, the general form of a backtracking algorithm looks like the following, where `state` is the current state of the problem and `move` is the next move to be attempted to get closer to a solution.

```
(define (backtracker state move)
  (cond ((solution? state) state) ; a solution?
        ((no_move? move) #f) ; path failed
        ((illegal_move state move)  ; cannot apply move to state, so
         (backtracker state (next_move move)))  ; try next move instead
        ((backtracker (make_move state) first_move)) ; make move and continu
        (#t (backtracker state (next_move move))) ; prev move failed, try ne
  )
)
```

So what does this all mean? The easiest way to see is through an example. Consider the n-queens problem, which is a generalization of the 8 queens problem. In this problem, we try to place n queens on an $n \times n$ chess board such that no 2 queens can attack each other. For those not familiar with the rules of chess, this means that no 2 queens can occupy the same row, column, or diagonal on the board.

See Example:
`/home/cs433/examples/nqueens`

Again, the code in the example has comments describing the procedure.

Other Scheme Constructs

There is much more of Scheme we could consider:

- Definition of local names with `let` and related constructs.
- Grouping of function calls (much like curly braces let us group statements in C/Java) with `begin`.
- Iteration with the "named `let`" and `do` constructs.

We may touch on some of these as we discuss general programming language concepts and compare how various languages handle them.