Main topics of the week:
- Functional Programming
- Scheme

Functional Programming
- Very different paradigm from procedural
- Mimic mathematical functions as much as possible
- Everything is a value
- Functions encapsulate everything
- All call by value – no assignment and no variables in a pure functional language, so
  no side effects of function calls.
- No iteration – repetition must be done by recursion
- Programs consist of function definitions and applications
- Execution consists of evaluating function applications
- In pure functional language, there is no state (no use of memory) so a function always
  produces the same result given the same parameters (called referential transparency).
  E.g., no static or global variables to affect function evaluation – i.e., no state.

Scheme Introduction
- Simple dialect of Lisp. Scheme came along to fix a bug in Lisp – static scope instead
  of the dynamic scope of Lisp.
- Based on lists of things (values or lists) – only two “types”: atoms and lists
- Highly recursive
- Everything is a list – a list has a first element and a tail, so is actually a pair.
- Scheme is an untyped language

Scheme Basics
- Names can be letters, digits, underscores, and special characters except parentheses.
- Case insensitive.
- Interpreter is read-evaluate-write infinite loop. To invoke scheme: the command
  scheme starts interactive interpreter.
- Expressions read by interpreter are in form of a list (using parentheses).
- Prefix notation – operator/function followed by arguments, i.e., appears as a list
  enclosed in parentheses.
- Evaluation consists of evaluating parameters, then applying function.
- Literals evaluate to themselves.
- Primitive functions available:
  - Arithmetic: + - * / sqrt
  - Evaluation: eval quote
  - List selectors: car cdr
  - List constructor: cons
  - Predicates: eq? null? list? (#t is true #f is false)
  - Selection: if cond
• Function construction: **lambda define** (define can be used for defining “constants” – names for expressions; or for defining functions, in which case lambda is unnecessary)

• Output: **display newline**

The basic structure in scheme is a **pair**. For example, (cons 1 2) constructs a pair where the first element is 1 and the second element is 2. Historically, the first element of a pair was referred to as the address and the second element as the decrement. These names came from the names of the registers used on the original IBM machine where Lisp was developed. The operator **car** comes from Contents of Address Register, i.e., the head of the list, and **cdr** comes from Contents of Decrement Register.

Scheme uses an inductive definition of a list:

1) The empty list, represented by () (empty parens, or nil in Lisp)
2) If L is a list, then (cons element L) is a list e.g., (cons 1 (cons 2 ()))

Symbols can be defined in Scheme to bind them to the value of expressions by

(\textit{define d (+ 3 3)})

and in this case the expression \textit{d} would have the value 6. If we quote the value we are defining, it will not be evaluated, e.g.,

(\textit{define d ‘(+ 3 3)})

d and then the value of \textit{d} would just be (+ 3 3). This can in turn be evaluated by

(eval d)

We can define **functions** with arguments in Scheme with a special form of define by

(\textit{define (f x) (+ x x)})

Note the definition of a function specifies a name for the function, formal parameter names, and the body of the function – what the function will evaluate to. A function can be also be defined **anonymously** by

(\textit{lambda (x) (+ x x)})

In this case the value of the lambda expression is the function, which you can think of as a little program. Combining the simple form of define with lambda, we can get the same effect as the special function form of define: we are binding the function (as a value) to a symbol, giving it a name by

(\textit{define f (lambda (x) (+ x x))})

This is exactly what the special function form of define does. With either way of binding \textit{f} to a function, we evaluate the function by:

(f 3)

which would return the value 6.

Everything in Scheme is evaluation, i.e., the interpreter reads a line and evaluates it. Numbers are recognized as numeric values. To see the difference of data versus tokens, consider that

‘(1 2) is equivalent to (cons 1 (cons 2 ()))

At the prompt in Scheme, the **car** of the expression (a list) must be a function or keyword, i.e., it is always interpreted as a program. To view something as data, it must be quoted, using the builtin function **quote** or simply the single quote mark:

‘(1 2)
quote ( 1 2)
but we can force evaluation with the builtin eval as in the example above.

Note that scheme is call-by-value (but we can use quote to get call by name).

Suppose we have a function defined:

(define f (lambda(x) 0))

which is a constant function. In this case, delaying evaluation would be more efficient
since the argument x is not used in the body of the function, so delaying evaluation
means we do not have to bother evaluating the argument at all.

Consider the difference between

(eval `(+ 2 3)) and (eval (+ 2 3))

In the first case Scheme builds the data, then uses eval to do the addition. In the second
case, the list is evaluated to 5, then evaluated again to 5.

In the following, the first gives a correct answer of 1, but the second is an error:

(car `(1 3)) and (car (1 3))

Scheme has a notion of higher order functions, which are functions that accept
functions as arguments. So with a function f already defined, we can insert it into a list,
and then extract it for evaluation, e.g.,

((car (cons f ())) 5)

This applies f to 5.

Since Scheme is a language of expressions that are evaluated, it does not have control
flow constructs. However, there are conditional constructs that can be used in
expressions similar to the ternary expression of C/C++/Java:

(if #t 2 3)
yields the value 2. The Boolean values true and false are written as #t and #f. A
conditional is necessary for doing interesting things, like writing a recursive function:

> (define csum (lambda(L) (if (null? L) 0
(+ (car L) (csum (cdr L)))))
> (csum (list 1 2 3))

6

Some predicates we can use in Scheme are number? symbol? null? equal?
which test if something is a number, symbol, the empty list, or if two things are
structurally equal (as opposed to eq? which tests if two things have the same
representation). We can evaluate a sequence of conditionals in more list like form by

(cond (#t 2) (#f 3) (#t 4) (else 5))

Here the list of pairs is evaluated sequentially and the value of the expression is the cdr of
the first pair whose car is true.

Scheme also permits the binding of expressions to symbols with the let construct
which takes a list of pairs, each consisting of a symbol and an expression, and a final
expression. The expression of each pair is evaluated and then the final expression is
evaluated, where the symbols have been bound to the expression value in each pair.
There is a sequential variant let* where the pairs are evaluated sequentially, with the
symbol bindings accumulated and effective for the next evaluation.

> (define x 0)
> (let ((x 2) (y x)) y)
0
> x
0
> (let* ((x 2) (y x)) y)
2
To see the difference and similarity between data and program, consider the following:

\[(+ 1 2) \quad \text{versus} \quad (\text{cons} + (\text{cons} 1 (\text{cons} 2 ()')))\]

The first is evaluated with the builtin operator + and results in a value of 3. The second results in a value which is a list whose car is +, and whose cdr is the list whose car is 1 and whose cdr is the list whose car is 2 and whose cdr is nil. That is, the latter is a data structure only. If we were to surround the second by an eval, we would get the value 3 because the data structure has the procedure + and it would be evaluated with the other items in the list as arguments to the procedure. So we see that the same list structure is used for pure data and also for program code. Now suppose that we consider the expression:

\[(\text{cons} + (\text{cons} 1 2))\]

At first glance this appears to be the same data structure we just defined, i.e., a list of +, 1, and 2. However, if we try to evaluate it, we get an error because the procedure + is being applied to a single argument which is a cell with 1 and 2. That is a very different structure from a list of 1 and 2.

Scheme appears to be low level, but it is useful because of its flexibility. Programs can be changed at runtime. In fact, nothing is type checked until runtime, no structures are statically defined. Since everything is in terms of lists, we already have that basic structure, but the structure of the elements is left until evaluation. Programs can then be more easily modified on the fly without going back through a compilation. Some pieces can be left undefined – as long as they aren’t executed (evaluated) they won’t produce an error. The list structure gives us a powerful mechanism for various constructions, e.g., writing a interpreter is easier than in Java where you would first have to define the structure for representing a parser.

Let’s look at the example again to see the difference between data and a program:

\[
(\text{define} \ A \ '(+ \ 1 \ 2))
\]

\[
((\text{eval} \ (\text{car} \ A)) \ 5 \ 6)
\]

I.e., we need to eval (car A) to get the function. If not, we just get the symbol +. Note the first item of a list must evaluate to a function rather than just a symbol.

Sharing of data: recall scheme is call by value – what we really mean is that evaluation is done on an argument list rather than delayed to the function body. But scheme can share data. Consider

\[
(\text{lambda} \ (x) \ (\text{cons} \ x \ x))(\text{cons} \ 1 \ 2))
\]

Here the anonymous function is given an argument and applies cons to construct a list where the first and second elements are both the argument given. In this case, there is one representation of the list of 1 and 2, but the function uses this representation twice. Thus, the constructed list has its car and cdr both point to the same representation. Sharing data is more efficient and we leave it to the interpreter to know when there are no active references to shared data. In particular, scheme does Garbage Collection and so must keep careful track of when data can be garbage collected, i.e., memory released for reuse. Sharing data obviously makes this more complicated.

Here’s another example of a higher order function. This function takes two functions as arguments and evaluates to the composition of the two functions:
(define compose (lambda (f g) (lambda (x) (f (g x)))))
We would use this like:
((compose f g) x)
For example, if we define two functions, we can now compose them.
> (define (f1 x) (* x x))
> (define (g1 y) (+ y 10))
> ((compose f1 g1) 3)
169
> ((compose g1 f1) 3)
19

Suppose we want to define a function that takes a list and produces the list with all
the elements in reverse order. This is a typical place to use recursion by reducing the list
to a smaller one and eventually to a base case that is trivially reversed. To combine the
recursion results, we take the first element, and stick it at the end of the reversal of the
remaining part of the list. So we need a place to build the reversed list. We define our
function rev to take two lists, the first being the one we want reversed, and the second
being the reversal so far:
(define (rev l r) (cond  ((null? l) r)
                        (else (rev (cdr l) (cons (car l) r)))))
The idea is to build up the reversed list as the second argument. When the first argument
is the empty list, we have reversed the entire list. Thus, we have:
> (rev '(1 2 3 4) ())
(4 3 2 1)

Of course, if we give a non-empty list as the second argument, we get peculiar looking
results:
> (rev '(1 2 3 4) '(5 6 7 8))
(4 3 2 1 5 6 7 8)

We should really view this as a helper function and define a function that takes a single
list as an argument:
(define (reverse l) (rev l ()))

So why not just define reverse as a function on a single list to start with? The problem is
that when we use cons to tack the first element to the end of the reversed list, we get a list
structure that is weighted toward the front, e.g.,
> (define (rv l) (cond  ((null? l) l)
                        (else (list (rv (cdr l)) (car l))))))
> (rv '(1 2 3 4))
(((4 3) 2) 1)

That is, we’re appending elements as lists, so the construction of the lists goes toward the
beginning via the recursion. One way to solve this is to define an append function that
takes two lists and appends one to the other, preserving the list structure:
(define (append l1 l2)
  (cond  ((null? l1) l2)
        (else (cons (car l1) (append (cdr l1) l2)))))
Then we can define a reverse function that uses recursion on the tail. However, this still means that there is a pending operation (namely append) on the recursion on the tail. That is, we cannot optimize away the deep nesting of the recursive calls to rv since we need the result from each call to rv to perform the evaluation of append. So although this achieves the simpler interface of a single argument, it still results in deep recursion that cannot be optimized away.

\[
\text{(define (rv l)}
\text{(cond ((null? l) ()))}
\text{(else (append (rv (cdr l)) (list (car l))))))}
\]

Here’s another example of using tail recursion, this time not for an actual list, but rather for the factorial function, where an accumulator is used for the result. Notice that the difference between this and a direct implementation of factorial with a single argument is that the multiplication here takes place progressively, as \( n \) is reduced, rather than all at the end when the factorial of 1 is calculated. Of course, here we must be careful to supply the correct value of 1 for the original accumulator, but that can be achieved by using this definition as a helper function and wrapping it by a function that calls it with the properly initialized result. A good implementation can then optimize away the nesting of the calls since there is no pending operation to perform on return. (I.e., you can think of it as the recursive call actually overlaying its caller.)

\[
> \text{(define (fact n res) (cond ((= n 0) res))}
\text{(else (fact (- n 1) (* n res))))}
\]

Here are some expressions to help understand about the structure of lists in scheme:

\[
> \text{(cons 1 2)}
\text{(1 . 2)}
\]

\[
> \text{'}(1 2)
\text{1 2}
\]

\[
> \text{(eq? (cons 1 2) '}(1 2))
\#f
\]

\[
> \text{(equal? (cons 1 2) '}(1 2))
\#f
\]

In the above two cases, we see there is a structural as well as object difference in the list of 1 and 2 compared to the list we form by cons’ing the objects 1 and 2. In a picture, these would look like:

```
  •            1
 / \          /   \
1   •    2    2   •
```

In the above two cases, we see there is a structural as well as object difference in the list of 1 and 2 compared to the list we form by cons’ing the objects 1 and 2. In a picture, these would look like:
In this next example, we have two lists which are the same on a structural and value basis, but are different objects. In a picture, each list looks like the second in the above picture.

```
> (cons 1 ' (2))
(1 2)
> (eq? ' (1 2) (cons 1 ' (2)))
#f
> (equal? ' (1 2) (cons 1 ' (2)))
#t
```

Recall our discussion of scope earlier for imperative languages. There, we think of variables as naturally having block scope, i.e., a name in an expression refers to the variable of that name declared within the closest enclosing block. This is also called lexical or static scope. Scheme implements static scope. The “blocks” here are enclosing sub-expressions. A define in scheme can be global – at the outmost scope, or a define can also be nested within an expression, say a function body definition. So for example, we could have the following three definitions in scheme:

```scheme
(define (foo) 1)
(define (bar) (foo))
(define (foobar) (define (foo) 6) (bar))
```

Here we are defining a function named ‘foo’ that simply evaluates to the number 1. We then define a function ‘bar’ that evaluates to applying the function ‘foo’. So if you evaluate ‘bar’, at this point you would certainly expect to see the value 1. Now we also define a function ‘foobar’ which evaluates to applying ‘foo’, but insert a re-definition of ‘foo’ before the expression defining ‘foobar’. So the question is, what should you expect to see when you evaluate ‘foobar’? Since scheme is static scope, we need only look at the definition of ‘bar’ – even though there is a closer ‘foo’ in the definition of ‘foobar’, the definition of ‘bar’ sees only the global ‘foo’ which is defined as 1.

```
> (foobar)
1
```

This is natural and intuitive. However, there is a concept of dynamic scope – where the “closest enclosing block” resolution is replaced by “latest definition”. In the case above, this would mean that when applying ‘foobar’, its definition defines a new ‘foo’, so by the time we apply ‘bar’, the most recent (dynamic!) definition of ‘foo’ would be used, resulting in a value of 6. Dynamic scope was actually a bug in the original Lisp implementations, and scheme ‘fixed’ it, i.e., implemented the more intuitive static scope. If you try the code above in Lisp (with the necessary slight syntax changes), the value produced will be 6, showing that dynamic scope was used. Dynamic scope would seem to make programs more difficult to understand, so is not generally implemented by most languages. However, the fundamental idea of this dynamic binding is at the heart of
virtual functions in C++ and Java, and in that environment, when wrapped with the notion of objects and polymorphism, does make intuitive sense, and is a powerful programming idiom.

Following is an example of a simple function that looks up values in a table. The table is a list of pairs of values, and the function returns the second element of a pair whose first element matches the given value.

```
> (define (lookup v slist) (cond ((null? slist) ())
   ((equal? v (car (car slist))) (car (cdr (car slist)))))
   (else (lookup v (cdr slist)))))

> (define slist '(("a" 1) ("b" 2) ("c" 3) ("d" 4)))
> (lookup "a" slist)
1
> (lookup 3 slist)
() 
> (lookup "c" slist)
3
```