CSE 341 - Programming Languages - Winter 2010 Scheme

Scheme profile

- Lisp dialect
- can be used in a functional style (but not purely functional)
- dynamic typing; type safe
- exclusively heap-based storage w/ garbage collection
- pass by value with pointer semantics
- lexically scoped (originally Lisp used dynamic scoping)
- first-class functions
- anonymous functions
- syntactically simple, regular (but lots of parens)
    - everything in lists!
    - program-data equivalence (This makes it easy to write Scheme programs that process/produce other programs, e.g. compilers, structure editors, debuggers, etc.)
- typically can be run either interpreted or compiled

Lisp application areas:

- teaching
- AI (expert systems, planning, etc)
- Simulation, Modeling
- Scripting (e.g. emacs)
- Rapid prototyping

Lisp was developed in the late 50s by John McCarthy. The Scheme dialect was developed by Guy Steele and Gerry Sussman in the mid 70s. In the 80s, the Common Lisp standard was devised. Common Lisp has many many features -- Scheme is cleaner.

Primitive Scheme data types and operations

Some primitive (*atomic*) data types:

- numbers
    - integers (examples: 1, 4, -3, 0)
    - reals (examples: 0.0, 3.5, 1.23E+10)
    - rationals (e.g. 2/3, 5/2)
- symbols (e.g. fred, x, a12, set!)

- boolean: Scheme uses the special symbols #f and #t to represent false and true.
- strings (e.g. "hello sailor")
- characters (eg #\c)

In the R5RS standard, case is not significant for symbols (which means it isn't significant for variable names) -- but this varies by dialect: it *is* significant in the Pretty Big dialect, for example. Recommendation: write your programs so that they work correctly whether or not case is significant in symbols. Note that you can have non-alphanumeric characters such as + or - or ! in the middle of symbols. (You can't have parentheses, though.) Here are some of the basic functions that scheme provides for the above datatypes.

- Arithmetic functions (`+`, `-`, `*`, `/`, `abs`, `sqrt`)
- Relational (`=`, `<`, `>`, `<=`, `>=`) (for numbers)
- Relational (`eq?`, `eqv?`, `equal?`) for arbitrary data (more about these later)
- Logical (`and`, `or`, `not`): `and` and `or` are short circuit logical functions.

Some functions are *predicates*, that is, they are truth tests. In Scheme, they return #f or #t.

- `number? integer? pair? symbol? boolean? string?`
- `eqv? equal?`
- `= < > <= >=`

Applying operators, functions

Ok, so we know the names of a bunch of functions. How do we use them? Scheme provides us with a uniform syntax for invoking functions:

```
(function arg1 arg2 ... argN)
```

This means all functions, including arithmetic ones, have *prefix* syntax. Arguments are passed by value (except with *special forms*, discussed later, to allow for things such as short circuiting for boolean operators).

Examples:

```
(+ 2 3)
(abs -4)
(+ (* 2 3) 8)
(+ 3 4 5 1)
;; note that + and * can take an arbitrary number of arguments
;; actually so can - and / but you'll get a headache trying to remember
;; what it means
;;
;; semicolon means the rest of the line is a comment
```

The List Data Type

Perhaps the single most important built in data type in Scheme is the list. In Scheme, lists are

unbounded, possibly heterogeneous collections of data. Examples:

```
(x)
(elmer fudd)
(2 3 5 7 11)
(2 3 x y "zoo" 2.9)
()
```

Box-and-arrow representation of lists:

```
 _______________        _______________
|       |       |      |       |       |
|   o   |   ----|----->|   o   |   o   |
|___|___|_______|      |___|___|___|___|
    |                      |       |
    |                      |       |
  elmer                   fudd     ()
```

Or

```
 _______________        ____________
|       |       |      |       |  / |
|   o   |   ----|----->|   o   | /  |
|___|___|_______|      |___|___|/___|
    |                      |
    |                      |
  elmer                   fudd
```

Notes:

- (x) is not the same as x
- () is the empty list
- Lists of lists: `((a b) (c d))` or `((fred) ((x)))`
- Scheme lists can contain items of different types: `(1 1.5 x (a) ((7)))`

Here are some important functions that operate on lists:

- `length` -- length of a list
- `equal?` -- test if two lists are equal (recursively)
- `car` -- first element of a list
- `cdr` -- rest of a list
- `cons` -- make a new *list cell* (a.k.a. *cons cell*)
- `list` -- make a list

Scheme also predefines compositions of `car` and `cdr`, e.g., `(cadr s)` is defined as `(car (cdr s))`.) All 28 combinations of 2, 3, and 4 a's and d's are defined.

Predicates for lists:

- `null?` -- is the list empty?
- `pair?` -- is this thing a nonempty list?

Evaluating Expressions

Users typically interact with Scheme though a *read-eval-print loop* (*REPL*). Scheme waits for the user to type an expression, reads it, evaluates it, and prints the return value. Scheme expressions (often called *S-Expressions*, for *Symbolic Expressions*) are either lists or atoms. Lists are composed of other S-Expressions (note the recursive definition). Lists are often used to represent function calls, where the list consists of a function name followed by its arguments. However, lists can also used to represent arbitrary collections of data. In these notes, we'll generally write:

```
<S-expression> => <return-value>
```

when we want to show an S-expression and the evaluation of that S-expression. For instance:

```
(+ 2 3)      => 5
(not #t ) => #f
```

Evaluation rules:

1. Numbers, strings, #f, and #t are literals, that is, they evaluate to themselves.
2. Symbols are treated as variables, and to evaluate them, their bindings are looked up in the current environment.
3. For lists, the first element specifies the function. The remaining elements of the list specify the arguments. Evaluate the first element in the current environment to find the function, and evaluate each of the arguments in the current environment, and call the function on these values. For instance:

```
(+ 2 3)              => 5
(+ (* 3 3) 10)       => 19
(= 10 (+ 4 6))       =>  #t
```

Using Symbols (Atoms) and Lists as Data

If we try evaluating `(list elmer fudd)` we'll get an error. Why? Because Scheme will treat the atom `elmer` as a variable name and try to look for its binding, which it won't find. We therefore need to "quote" the names `elmer` and `fudd`, which means that we want scheme to treat them literally. Scheme provides syntax for doing this. The evaluation for quoted objects is that a quoted object evalutes to itself.

```
'x                   => x
(list elmer fudd)   => error! elmer is unbound symbol
(list 'elmer 'fudd) => (elmer fudd)
(elmer fudd)        => error! elmer is unknown function
'(elmer fudd)       => (elmer fudd)
(equal? (x) (x))      => error! x is unknown function
(equal? '(x) '(x))    => #t
(cons 'x '(y z))      => (x y z)
(cons 'x '() )         => (x)
(car '(1 2 3))        => 1
(cdr (cons 1 '(2 3))) => (2 3)
```

Note that there are several ways to make a list:

```
1. '(x y z) => (x y z)
2. (cons 'x (cons 'y (cons 'z '() ))) => (x y z)
3. (list 'x 'y 'z) => (x y z)
```

Internally, quoted symbols and lists are represented using the special function `quote`. When the reader reads `'(a b)` it translates this into `(quote (a b))`, which is then passed onto the evaluator. When the evaluator sees an expression of the form `(quote s-expr)` it just returns `s-expr`. `quote` is sometimes called a "special form" because unlike most other Scheme operations, it doesn't evaluate its argument. The quote mark is an example of "syntactic sugar."

```
  'x          => x
  (quote x)   => x
```

(Alan Perlis: "syntactic sugar causes cancer of the semicolon".)

Variables

Scheme has both local and global variables. In Scheme, a variable is a name which is bound to some data object (using a pointer). There are no type declarations for variables. The rule for evaluating symbols: a symbol evaluates to the value of the variable it names. We can bind variables using the *special form* `define`:

```
(define symbol expression)
```

Using `define` binds `symbol` (your variable name) to the result of evaluating `expression`. `define` is a special form because the first parameter, `symbol`, is not evaluated.

The line below declares a variable called clam (if one doesn't exist) and makes it refer to 17:

```
(define clam 17)

clam          => 17

(define clam 23)  ; this rebinds clam to 23

(+ clam 1) => 24

(define bert '(a b c))
(define ernie bert)
```

Scheme uses pointers: bert and ernie now both point at the same list.

In 341 we'll only use `define` to bind global variables, and we won't rebind them once they are bound, except when debugging.

Lexically scoped variables with `let` and `let*`

We use the special form `let` to declare and bind local, temporary variables. Example:


```
;; general form of let
(let ((name1 value1)
      (name2 value2)
         ...
      (nameN valueN))
   expression1
   expression2
   ...
   expressionQ)

;; reverse a list and double it

;; less efficient version:
(define (r2 x)
  (append (reverse x) (reverse x)))

;; more efficient version:
(define (r2 x)
  (let ((r (reverse x)))
        (append r r)))
```

A problem with `let` in some situations is that while the bindings are being created, expressions cannot refer to bindings that have been made previously. For example, this doesn't work, since x isn't known outside the body:

```
 (let ((x 3) (y (+ x 1))) (+ x y))
```

To get around this problem, Scheme provides us with `let*`:

```
(let* ((x 3)
       (y (+ x 1)))
    (+ x y))
```

Personally I prefer to use `let`, unless there is a particular reason to use let*.

Defining your own functions

Lambdas: Anonymous Functions

You can use the `lambda` special form to create anonymous functions. This special form takes

```
(lambda (param1 param2 ... paramk)  ; list of formals
        expr)                       ; body
```

`lambda` expression evaluates to an anonymous function that, when applied (executed), takes k arguments and returns the result of evaluating `expr`. As you would expect, the parameters are lexically scoped and can only be used in `expr`.

Example:

```
(lambda (x1 x2)
        (* (- x1 x2) (- x1 x2)))
```


Evaluating the above example only results in an anonymous function, but we're not doing anything with it yet. The result of a `lambda` expression can be directly applied by providing arguments, as in this example, which evaluates to 49:

```
((lambda (x1 x2)
         (* (- x1 x2) (- x1 x2)))
 2 -5)   ; <--- note actuals here
```

Defining Named Functions

If you go to the trouble of defining a function, you often want to save it for later use. You accomplish this by binding the result of a `lambda` to a variable using `define`, just as you would with any other value. (This illustrates how functions are first-class in Scheme. This usage of `define` is no different from binding variables to other kinds of values.)

```
(define square-diff
        (lambda (x1 x2)
                (* (- x1 x2) (- x1 x2))))
```

Because defining functions is a very common task, Scheme provides a special shortcut version of `define` that doesn't use `lambda` explicitly:

```
(define (function-name param1 param2 ... paramk)
        expr)
```

Here are some more examples using `define` in this way:

```
(define (double x)
        (* 2 x))

(double 4) => 8


(define (centigrade-to-fahrenheit c)
        (+ (* 1.8 c) 32.0))

(centigrade-to-fahrenheit 100.0) => 212.0
```

The `x` in the `double` function is the formal parameter. It has scope only within the function. Consider the three different `x`'s here...

```
(define x 10)

(define (add1 x)
  (+ x 1))

(define (double-add x)
  (double (add1 x)))

(double-add x)   => 22
```

Functions can take 0 arguments:


```
(define (test) 3)
(test)  => 3
```

Note that this is <u>not the same</u> as binding a variable to a value:

```
(define not-a-function 3)
not-a-function    => 3
(not-a-function)  => ;The object 3 is not applicable.
```

Commenting Style

If Scheme finds a line of text with a semicolon, the rest of the line (after the semicolon) is treated as whitespace. However, a frequently used convention is that one semicolon is used for a short comment on a line of code, two semicolons are used for a comment within a function on its own line, and three semicolons are used for an introductory or global comment (outside a function definition).

Equality and Identity: `equal?`, `eqv?`, `eq?`

Scheme provides three primitives for equality and identity testing:

1. `eq?` is pointer comparison. It returns #t iff its arguments literally refer to the same objects in memory. Symbols are unique (`'fred` always evaluates to the same object). Two symbols that look the same are `eq`. Two variables that refer to the same object are `eq`.
2. `eqv?` is like `eq?` but does the right thing when comparing numbers. `eqv?` returns #t iff its arguments are `eq` *or* if its arguments are numbers that have the same value. `eqv?` does not convert integers to floats when comparing integers and floats though.
3. `equal?` returns true if its arguments have the same structure. Formally, we can define `equal?` recursively. `equal?` returns #t if its arguments are `eqv`, or if its arguments are lists whose corresponding elements are `equal`; and otherwise false. (Note the recursion.) Two objects that are `eq` are both `eqv` and `equal`. Two objects that are `eqv` are `equal`, but not necessarily `eq`. Two objects that are `equal` are not necessarily `eqv` or `eq`. `eq` is sometimes called an identity comparison and `equal` is called an equality comparison.

Examples:

```
(define clam '(1 2 3))
(define octopus clam)               ; clam and octopus refer to the same list

(eq? 'clam 'clam)           => #t
(eq? clam clam)             => #t
(eq? clam octopus)          => #t
(eq? clam '(1 2 3))         => #f ;
(eq? '(1 2 3) '(1 2 3))     => #f
(eq? 10 10)                 => #t ; (generally, but implementation-dependent)
(eq? 10.0 10.0)             => #f ; (generally, but implementation-dependent)
(eqv? 10 10)                => #t ; always
```

```
(eqv? 10.0 10.0)             => #t ; always
(eqv? 10.0 10)               => #f ; no conversion between types
(equal? clam '(1 2 3))       => #t
(equal? '(1 2 3) '(1 2 3))   => #t
```

Scheme provides `=` for comparing two numbers, and will coerce one type to another. For example, `(equal? 0 0.0)` returns `#f`, but `(= 0 0.0)` returns `#t`.

Logical operators

Scheme provides us with several useful logical operators, including `and`, `or`, and `not`. Operators `and` and `or` are special forms and do not necessarily evaluate all arguments. They just evaluate as many arguments as needed to decide whether to return `#t` or `#f` (like the `&&` and `||` operators in Java and C++). However, one could easily write a version that evaluates all of its arguments.

```
(and expr1 expr2 ... expr-n)
; return true if all the expr's are true
; ... or more precisely, return expr-n if all the expr's evaluate to
; something other than #f.  Otherwise return #f

(and (equal? 2 3) (equal? 2 2) #t)  => #f

(or expr1 expr2 ... expr-n)
; return true if at least one of the expr's is true
; ... or more precisely, return expr-j if expr-j is the first expr that
; evaluates to something other than #f.  Otherwise return #f.

(or (equal? 2 3) (equal? 2 2) #t)  => #t

(or (equal? 2 3) 'fred (equal? 3 (/ 1 0)))  => 'fred

(define (single-digit x)
   (and (> x 0) (< x 10)))

(not expr)
; return true if expr is false

(not (= 10 20))     => #t
```

Boolean Peculiarities

In R4 of Scheme the empty list is equivalent to #f, and everything else is equivalent to #t. However, in R5 the empty list is also equivalent to #t! For clarity, I recommend you only use #f and #t for boolean constants.

Conditionals

`if` special form


```
(if condition true_expression false_expression)
```

If `condition` evaluates to true, then the result of evaluating `true_expression` is returned; otherwise the result of evaluating `false_expression` is returned. `if` is a special form, like `quote`, because it does not automatically evaluate all of its arguments.

```
(if (= 5 (+ 2 3)) 10 20)     => 10
(if (= 0 1) (/ 1 0) (+ 2 3)) => 5
; note that the (/ 1 0) is not evaluated

(define (my-max x y)
   (if (> x y) x y))

(my-max 10 20)                   => 20

(define (my-max3 x y z)
   (if (and (> x y) (> x z))
       x
       (if (> y z)
            y
            z)))
```

`cond` -- a more general conditional

The general form of the cond special form is:

```
(cond (test1 expr1)
      (test2 expr2)
      ....
      (else exprn))
```

As soon as we find a test that evaluates to true, then we evaluate the corresponding expr and return its value. The remaining tests are not evaluated, and all the other expr's are not evaluated. If none of the tests evaluate to true then we evaluate exprn (the "else" part) and return its value. (You can leave off the else part but it's not good style.)

```
(define (sign n)
   (cond ((> n 0) 1)
         ((< n 0) -1)
         (else 0)))
```

Tail Recursion

A tail recursive function is one that returns the result of the recursive call back without alteration. (So unlike a function like append, we don't build up a solution as the recursion unwinds.) Examples:

```
(define (all-positive x)
  (cond ((null? x) #t)
        ((<= (car x) 0) #f)
        (else (all-positive (cdr x)))))
;; (all-positive '(3 5 6))  => #t
```



```
;; (all-positive '(3 5 -6))  => #f

(define (my-member e x)
  (cond ((null? x) #f)
        ((equal? e (car x)) #t)
        (else (my-member e (cdr x)))))
```

Scheme compilers handle tail recursion very efficiently, as efficiently as a program that just uses loops instead of recursion. (In particular, tail recursive functions don't use stack space for every recursive call.)

Using Accumulators to Make a Function Tail-recursive

Sometimes you can use an *accumulator* -- an additional parameter to a function that accumulates the answer -- to convert a non-tail recursive function into a tail recursive one. For example, the usual definition for `factorial` isn't tail-recursive:

```
(define (std-factorial n)
  (if (zero? n)
      1
      (* n (std-factorial (- n 1)))))
```

Here is a version that is tail recursive:

```
(define (factorial n)
  (acc-factorial n 1))


;; auxiliary function that takes an additional parameter (the accumulator,
;; i.e. the result computed so far)
(define (acc-factorial n sofar)
  (if (zero? n)
      sofar
      (acc-factorial (- n 1) (* sofar n))))
```

Higher-Order Functions

Scheme includes higher-order functions such as `map` and `filter`, similar to those in Haskell:

```
(map function list)             ;; general form

(map null? '(3 () () 5))        => (() T T ())

(map round '(3 3.3 4.6 5))      => (3 3 5 5)

(map cdr '((1 2) (3 4) (5 6)))  => ((2) (4) (6))

(map (lambda (x) (* 2 x)) '(3 4 5))

(filter (lambda (n) (> n 10)) '(5 10 15 20))  => (15 20)

(define (add-n-to-list alist n)
```



```
  (map (lambda (x) (+ n x)) alist))
```

Note that in the add-n-to-list function, the body of the lambda function can refer to the variable n, which is in the lexical scope of the lambda expression.

map can also be used with functions that take more than one argument. Examples:

```
(map + '(1 2 3) '(10 11 12))   => (11 13 15)
(map (lambda (x y) (list x y)) '(a b c) '(j k l))   => ((a j) (b k) (c l))
```

(This doesn't work in Haskell. Why not?)

Defining Higher-Order Functions

Suppose we wanted to define the 1-argument version of map ourselves. We can do it like this:

```
(define (my-map f s)
   (if (null? s)
       '()
       (cons (f (car s)) (my-map f (cdr s)))))
```

Lexical Closures

Now that we have the code for `my-map`, we can talk about what makes lambda special. A lambda expresssion evaluates to a *lexical closure*, which is a coupling of code and a lexical environment (a scope, essentially). The lexical environment is necessary because the code needs a place to look up the definitions of symbols it references. For example, look again at `add-n-to-list` below.

```
(define (add-n-to-list alist n)
  (my-map (lambda (x) (+ n x)) alist))
```

When the lambda expression is used in `my-map`, it needs to know where to look up the variable name n. It can get the right value for n, because it retains its lexical environment.

Nested Scopes

You can have arbitrarily nested scopes (scopes within scopes within scopes ...). Further, since function names are bound like any other variable, function names also obey the scope rules.

As an example, let's define a simple function `test`:

```
(define (test x)
   (+ x 1))
```

Evaluating `(test 10)` gives `11`.

However, if we evaluate

```
(let ((test (lambda (x) (* x 2)))
     (test 10))
```