

## Scheme—A Dialect of Lisp

The basic idea is that of symbolic programming using list expressions and recursive functions. Scheme is a functional programming language that is meant to be used for pragmatic software development. So, like ML, it is “impure” in that it does allow for assignment, but the core is a pure functional language based on expressions. Scheme systems are based on the Read-Eval-Print loop interpreter model, as we saw earlier. Most scheme systems have a small C or C++ run-time system, and the rest of the language is written in Scheme itself!

Scheme was designed and implemented by Guy Steele and Gerald Sussman at MIT in 1975. It is influenced syntactically and semantically by Lisp and conceptually by Algol. The first Scheme system was called Rabbit. There are two key historical papers: *Lambda: The Ultimate Declarative* and *Lambda: The Ultimate Imperative*.

- Lisp contributed simple syntax, uniform representation of programs as lists and garbage collected heap-allocated data.
- Algol contributed lexical scoping and block structure.
- Lisp and Algol both defined recursive functions. Scheme implements **proper tail-recursive functions**, where possible.

Scheme has a formal mathematical semantics derived from a mathematical model of computation called the *Lambda Calculus*. The lambda calculus was defined by Alonzo Church in the mid 1930s as a computational theory of recursive functions, that is essentially equivalent in computational power to *Turing machines*, defined by Alan Turing also in the mid 1930s.

- Turing machines are abstract machines that emphasize computation as a series of state transitions driven by symbols on an input tape, which leads naturally to an **imperative** style of programming based on assignment.
- The lambda calculus emphasizes expressions and function, which naturally leads to a **functional** style of programming based on evaluation of expressions by **function application** to argument values (hence the term *applicative style*).

Both Turing machines and the Lambda Calculus are idealized, mathematical models of computation. Scheme and ML (and other functional languages) have a *denotational semantics* based on the lambda calculus. That is, the meaning of all the syntactic programming constructs in the language are defined in terms of mathematical functions.

## Summary of Scheme's Key Features

- Scheme is **statically (lexically) scoped**, such that the binding of each variable is lexically apparent. Scheme uses the `let`, `let*` and `letrec` operators to define variables bindings within local scopes.
- Scheme has **dynamic** or **latent** typing, meaning that types are associated with values at run-time. A variable assumes the type of the value that is bound to them at run-time. So the type of a variable changes dynamically during execution, depending on the value bound to it at a given point in time.
- All objects created at run-time have potentially unlimited life-time or *extent*. This is made possible by the existence of automatic garbage collection. A thing is garbage if it can no longer be used in a future computation in a program.
- Scheme implementations are required to be properly **tail recursive**. So, any recursively specified procedure that can be turned into an equivalent iteration will be. So, you can specify an iteration as a syntactically recursive procedure.
- Scheme procedures are **first-class objects**. This means that procedures can be *created dynamically* (!), stored in data structures, returned as results of expressions or procedures. Because functions are defined as lists in Scheme, they can be treated as data and you can create them on-the-fly as part of your program's execution. This means that a scheme program can "evolve" its behavior as it runs!
- Scheme data objects (e.g., lists) are first-class objects as well. They are all heap allocated, garbage collected, and can be passed as arguments to functions, returned as results, and combined to form larger data structures.
- Scheme supports many different types: numbers, characters, strings, symbols and lists. Numbers include integers, real, complex, and arbitrary precision rational numbers.
- Scheme also includes a large set of built-in functions for manipulation of lists and others data objects.
- Arguments to Scheme procedures are always passed by value, as with C/C++. So, actual arguments are always evaluated before a procedure is called, whether or not the procedure needs the values. This is called **eager or strict evaluation**, as we have discussed previously. There are functional languages, such as Miranda and Haskell that use **lazy or non-strict evaluation**, which means that actual arguments are not evaluated until they are needed in an expression within a function.

## Scheme Syntax and Naming Conventions

- Scheme programs are made up of: keywords, variables, structured forms (e.g., lists), numbers, characters, strings, quoted vectors, quoted lists, quoted symbols, whitespace and comments
- Identifiers (keywords, variables and symbols) are formed from the following alphanumeric characters: a-z, A-Z, 0-9 and ? ! . + - \* / < = > : \$ % ^ & \_ ~
- Identifiers cannot start with a character that would form a number: 0-9, -, +, .
- Predicate names end in the question mark symbol ?. For example, eq?, zero?, string=?
- Type predicates are the name of the type followed by a ?: pair?, string?
- Builtin character, string, and vector functions start with the name of the type. For example: string-append
- Procedures that convert an one type of object to another use the -> symbol: string->num
- The names of procedures (except I/O procedures) that cause **side-effects** end with the exclamation point: set!
- Strings are formed using double quotes: “Hello, world”
- Numbers are just numbers: 42, 3.14
- Some procedures are named by operator symbols: +, \*, / and they denote different functions depending on their operands: 1+2 is integer addition, 1.0+3.14 is floating point addition. So + names two different functions: one that takes integer type objects and one that takes real type objects.

## Simple Expressions

The official language definition for Scheme is the *Revised<sup>5</sup> Report on the Algorithmic Language Scheme*. Available from the course web site. I strongly recommend that you read the first 3-4 sections of this language report.

An expression in Scheme has the form:  $(E_1 E_2 \dots E_n)$

Where  $E_1$  evaluates to an operator and  $E_2$  through  $E_n$  are evaluated as operands. So, expressions in Scheme are defined in **prefix** notation. The advantage is that an operator can be applied to 1 or more arguments as part of one expression.

Some examples executed using the DrScheme interpreter:

```
(+ 1 2 3 4 5) => 15 ; infix: 1 + 2 + 3 + 4 + 5
```

```
(+ 1 (* 2 3) 4 5) => 16 ; infix: 1 + (2 * 3) + 4 + 5
```

Note that Scheme does dynamic type checking and automatic type coercion (in this case promotion of an integer to a float):

```
(+ 2.7 10) => 12.7
```

Scheme uses **inner-most** evaluation, meaning that arguments are evaluated first, then substituted as parameters to functions.

```
(define (square x) (* x x))  
(square (+ 2 3)) => (square 5) => (* 5 5) => 25
```

Note: once the subexpression  $(+ 2 3)$  is evaluated, the memory for this list can be garbage collected.

We can also use a lambda syntactic form to define the square function.

```
(define square (lambda (n) (* n n)))  
(square 0.5) => 0.25
```

## Top-Level Bindings

The **define** directive is used to introduce a *binding* of an identifier to some object, which is either a value or an expression. In scheme, we don't have assignment, as in imperative languages. Instead, we establish bindings of variables to values within some scope. The define directive is almost always used to introduce name bindings at the outermost (global) scope of the interpreter. The way to think of a name and value binding is as a symbol table mapping names to values within the current scope:

```
(define size 2) => size
(* 2 size) => 4
(define sum (+ 1 2 3 4 5)) => sum
sum => 15
(eval '(+ 1 2 3 4 5)) => 15
```

We can delay the evaluation of a (sub-)expression by *quoting* the expression, which tells the interpreter not to evaluate the (sub-)expression, but to bind the expression as a *literal* to a name. The evaluation of the expression is delayed until you explicitly request the evaluation.

```
(define sum '(+ 1 2 3 4 5)) => sum
sum => (+ 1 2 3 4 5)
(eval sum) => 15
```

Quoting using `'` is shorthand for the builtin quote operator, which treats an object as a literal expression:

```
(define sum (quote (+ 1 2 3 4 5))) => sum
```

Scheme is a functional programming language, but it does have a few imperative style commands, which end with the `!` mark. For example, to imperatively change the value of a variable binding, rather than establish a new binding, use the imperative set command:

```
(define x 2) ;; x must be bound before it can be set
(set! x 5) ;; changes the binding of x to the value 5
x => 5
(define x 0) => error: cannot redefine name x
(set! x 0) => 0
```

## Summing a list in Scheme

It is interesting to analyze the prefix sum expression `(+ 1 2 3 ... n)` in Scheme.

In C/C++/Java, we must use infix notation to express a sum of `n` integers:

```
int sum = 1 + 2 + 3 + ... + n;
```

or write a loop:

```
for (int sum = 1; sum < n+1; sum++);
```

Since the '+' operator is left associative, the expression `(+ 1 2 3 4 5)` is typically evaluated left to right as follows:

```
((((1 + 2) + 3) + 4) + 5) => 15
```

Another way to think of computing a sum is by *folding* the + operator into the elements of a list from the left with an initial value of zero, since we will fold *from the left* and + is the left-associate integer addition operator:

```
(foldl + 0 '(1 2 3 4 5)) => 15
```

```
(define (sum lst) (foldl + 0 lst))
```

```
(sum '(1 2 3 4 5)) => 15
```

Which expands to `(+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))` because the `foldl` function folds the operator + into the list from the left. Note that you could just as well fold *from the right*, which results in: `(+ 1 (+ 2 (+ 3 (+ 4 (+ 5 0)))))`. Which is more space efficient: `foldl` or `foldr`? Notice that with multiplication we fold a 1 as the initial start value, not zero.

```
(define (mult lst) (foldl * 1 lst))
```

```
(mult '(1 2 3 4 5)) => 120
```

**Q: Could you implement a non-recursive (i.e., iterative) factorial function using `foldl/foldr`?**

## Conditional Expressions and Predicates

Scheme defines builtin bindings for constant symbols representing the boolean values true and false

```
#t => #t
#f => #f
```

Conditional expressions are of the form:

```
(cond (<p1> <e1>)
      (<p2> <e2>)
      ...
      (<pn> <en>))
```

Which consists of a builtin primitive `cond` followed by parenthesized pairs of expressions (`<pi>` `<ei>`) called *clauses*. The expression `pi` is a *predicate* (truth function) that evaluates to either `#t` or `#f`. Each predicate sub-expression is evaluated in the order it appears in the enclosing `cond` expression. As soon as one is found to be true, the corresponding *consequent expression* `<e>` is evaluated and returned as the result of the `cond` expression. There are a couple of special cases. If the last predicate expression is the builtin symbol `else`, then if all other predicates evaluate to `#f`, then the expression associated with the `else` clause is returned as the result of the conditional expression:

```
(define (abs x)
  (cond ((< x 0)
        (- x))
        (else x)))
```

Builtin order predicate operators in Scheme include: `=`, `<`, `<=`, `>`, `>=`. Equality predicates are more complicated. They include `eq?`, `eqv?`, `equal?`. Use `eq?` to compare two objects that would have the same internal representation, `eqv?` to compare two numbers of characters, `equal?` to compare two objects for structural equality:

```
(eq? 'a 'a) => #t
(eq? 1.0 1.0) => #f
(eqv? 1.0 1.0) => #t
(eqv? "abc" "abc") => #f
(equal? "abc" "abc") => #t
```

## Conditional Expressions and Predicates

A special conditional, the `if` expression, can be used when there are only two cases:

```
(if <predicate> <consequent> <alternative>)  
(if <predicate> <consequent>)
```

```
(define (abs x)  
  (if (< x 0)  
      (- 0 x)  
      x))
```

```
((if #f + *) 3 4) - 12
```

Logical composition operators are also builtin:

```
(and <e1> ... <e2>)  
(or <e1> ... <e2>)  
(not <e1>)
```

Note that in the case of the `and` expression, the sub-expressions are evaluated left-to-right. If any one evaluates to `#f`, then the value of the `and` expression is `#f`. Otherwise the `and` expression evaluates to the value of the last subexpression. The expression evaluates to `#t`. This “short-circuit” boolean evaluation is common in most languages. Similarly, in the `or` expression, the subexpressions are evaluated left-to-right and if any subexpression evaluates to a true value (i.e. not `#f`), then that value is returned as the value of the `or` expression, and the remaining subexpressions are not evaluated. If all subexpressions evaluate to `#f`, then the value of the `or` expression is `#f`. The expression `(or)` evaluates to `#f`. The `not` expression evaluates to `#t` if the subexpression evaluates to `#f`, otherwise the `not` expression evaluates to `#f`.

Examples:

```
(and (> x 5) (< x 10))  
(define (>= x y) (or (> x y) (= x y)))  
(define (>= x y) (not (< x y)))
```

## Functions

In Scheme, the term *procedure* is used to describe what are actually functions (which only cause a side-effect is using `set!` or `set-car!` or `set-cdr!`). It is confusing that a functional language like Scheme based on functions uses the term *procedure* generically to include non side-effecting functions, while imperative languages like C/ C++ use the term *function* generically to describe side-effecting procedures!

Scheme procedures create objects called *closures*. A closure consists of the function definition along with a set of **lexically-scoped, statically** determined *bindings* that represent the environment at the time of definition. In Scheme, closures are first-class objects that can be passed as values to other functions, stored in data structures, and returned as the result of a function.

The general form of a Scheme function definition is:

```
(define (<name> <formal parameters>) <body>)
```

```
(define (square x) (* x x))  
square => #<procedure:square>
```

Once defined, we can use the name binding for the procedure as a prefix operator on an argument expression or in the definition of additional procedures:

```
(square 5) => 25  
(square (+ 2 5)) => 49
```

```
(define (sum-of-squares x y) (+ (square x) (square y))) => sum-of-squares  
(sum-of-squares 3 4) => 25
```

## Functions are Lambda Expressions

A simple recursive definition of the factorial function is defined as:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Equivalently, we could use a *lambda expression* to construct an anonymous procedure and then bind the factorial name to the procedure that we have constructed:

```
(define factorial (lambda (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

A lambda expression constructs an anonymous procedure (closure) and has the general form:

```
(lambda (<formal parameters>) <body>)
```

When the lambda expression is evaluated, the environment in which it is evaluated is remembered. When the procedure constructed by the expression is called, the actual arguments are substituted for the formal parameter, the environment for the procedure is then complete and the <body> is evaluated using the bindings of the environment augmented with the actual arguments. Each expression in the <body> is evaluated sequentially in order.

A lambda expression can be used to construct an anonymous procedure to be used as an operator in an expression:

```
((lambda (x y z) (+ x y z)) 1 2 3) => 6 ;; sum the first three integers
```

Anonymous lambda expressions are also often passed as arguments to **higher-order** functions.

```
(map (lambda (x) (* x x)) '(1 2 3 4 5)) => 15
(sort '(99 -1 0 18 32 -66) (lambda (x y) (> x y))) => (99 32 18 0 -1 -66)
(foldl * 1 (5 4 3 2 1)) => 120
```

## Let Expressions

A let expression allows the definition of local variable bindings. The general form of a let expression is:

```
(let ((<var1> <exp1>)
      (<var2> <exp2>)
      ...
      (<vark> <expk>))
  <body>)
```

The expressions <exp<sub>i</sub>> are all evaluated, then <body> is evaluated with each <var<sub>i</sub>> in <body> bound to the value obtained from evaluating each <exp<sub>i</sub>>. The result of the let expression is the value obtained from evaluating <body>.

For example:

```
(define pi 3.14) ; pi bound to 3.14 in the global environment
(define (sum-of-pisquare) (+ (square pi) (square pi)))
```

requires that the subexpression

```
(square pi)
```

be computed twice. Rather than define another global variable binding for this expression, we can localize a binding using a let expression:

```
(define (sum-of-pi-squared)
  (let ((pi-squared (square pi)))
    (+ pi-squared pi-squared)))
```

## Let Expressions are Lambda Expressions

It turns out that a let expression is just “syntactic sugar” for a lambda expression:

Note that the first part of a let expression is a list of variables that are not bound to the value of the corresponding expressions until ALL of the expressions have been evaluated. The body of the let expression is then evaluated with these names bound as local variables. This is exactly the same semantics as applying actual arguments to formal parameters of a lambda expression:

```
( (lambda (<var1> . . . . <vark>) <body>) <exp1> . . . <expk> )
```

Notice what this does:

- the lambda expression creates an anonymous procedure as the value of the inner expression.
- because of call-by-value, each of the expressions passed as arguments are ALL evaluated before the procedure is called.
- when the procedure is called, the variables representing formal parameters are bound to the values of the arguments and used in the evaluating the body of the procedure:  $\text{var}_i \leftarrow \text{exp}_i$
- So the effect of this evaluate and then bind is the same semantics as the let expression:

```
(let ((<var1> <exp1>)
      (<var2> <exp2>)
      . . .
      (<vark> <expk>))
  <body>)
```

It turns out that a let expression is actually a syntactic macro that is convenient for expressing lexically and locally scoped variable bindings over some expression(s) representing the body of the let (equivalently body of a lambda)

## Other Let Expression Forms

Like `let`, the `let*` binding construct creates a local set of variable bindings. However, with `let*`, the bindings are performed sequentially, from left to right so that earlier variable bindings apply to later variable bindings.

```
(let* ((<var1> <exp1>)
      (<var2> <exp2>)
      ...
      (<vark> <expk>))
  <body>)
```

Examples:

```
(define x 0) => x
x => 0
```

```
(let ((x 2) (y x)) y) => 0
```

```
(let* ((x 2) (y x)) y) => 2
```

`Let*` is also a form of syntactic sugar for a lambda expression. Because the binding order is important here, we have to lexically nest the lambda expressions and the application of the arguments. So the above `let*` expression is really just syntactic sugar for:

```
((lambda (x) (lambda (y) (y))) x) 2) => 2
```

## Lists in Scheme

Lists are pervasive in Scheme, to the point where just about everything is a list (but not quite everything). However, the interpreter *evaluates* most lists as an operator followed by operands, and returns a result. However, we can treat expressions as lists:

```
(+ 1 2 3 4 5) => 15 ; list evaluated as an expression resulting in the value 15
```

```
`(+ 1 2 3 4 5) => (+ 1 2 3 4 5) ; a list of symbols
```

The empty list is denoted by

```
()
```

The reserved word **nil** may or may not denote an empty list, depending on the Scheme implementation.

All of the following are lists:

```
(how are you doing so far)
(((how) are) ((you) (doing so)) far)
(() () () ())
(list 'a 'b 'c 'd) => (a b c d)
```

Useful operations on lists:

```
(null? x) ; x must be a list
(car x) ; head of the list
(cdr x) ; tail of the list
(cons 'a x) ; construct a list with a as the head and x as the tail
(list? `()) => #t
(list? (cons 'a `())) => #t
(eq? `(a b) `(a b)) => #f
(eqv? `(a b) `(a b)) => #f
(equal? `(a b) `(a b)) => #t
(equal? `(a b) (list 'a 'b)) => #t
```

## Cons, Car, and Cdr Operations

The primitives ‘car’ and ‘cdr’ are historical Lisp functions. Car used to mean the “contents of the address register” and cdr meant the “contents of the data register”. These terms refer to the way lists were originally implemented in early Lisp on mainframe computers. The names have stuck with us however. The best way to think of car is as a function that returns the first element of a list, or *head*, and cdr returns the “rest of the list” or the *tail*.

Car, cdr, and cons operations on lists:

```
(car '(a b c)) => a
(car '((a b c) x y z)) => (a b c)
(car '()) => ??
```

**Law of Car: car is defined only for non-empty lists**

```
(cdr '(a b c)) => (b c)
(cdr '((a b c) x y z)) => (x y z)
(cdr '(a)) => ()
(cdr '((x) t r)) => (t r)
(cdr 'a) => ??
(cdr '()) => ??
```

**Law of Cdr: cdr is defined on for non-empty lists. The cdr of a non-empty list is always another list**

```
(car (cdr '((b) (x y) ((c)))))) => ??
(cdr (car '((x) t r))) => ()
```

It is often useful in functions to use a let expression to bind the head and tail of an argument list to local variables:

```
(define sum (lambda (lst)
  (if (null? lst)
      0
      (let ((head (car lst))
            (tail (cdr lst)))
        (+ head (sum tail))))))
```

## Constructing Lists

Cons constructs news lists.

```
(cons 'peanut '(butter and jelly)) => (peanut butter and jelly)
(cons '(banana and) '(peanut butter and jelly)) => ((banana and) peanut butter and jelly)
(cons 'a '()) => (a)
```

**The Law of cons: cons takes two arguments. The second argument to cons must be list. The result is a list.**

The primitive null? is defined only for lists

```
(null? '(a b c)) => #f
(null? '()) => #t
```

Appending two lists is different than cons'ing a list

```
(append '(a b c) '(d)) => (a b c d)
```

The list primitive makes a new list

```
(list 'a) => (a)
(list 'a 'b 'c) => (a b c)
(list 'a '(b)) => (a (b))
```

A list can also be formed as a **dotted pair** which only takes up one “cons cell” of memory.

```
(define pair '(2 . 4) ) => (2 . 4)
(car pair) => 2
(cdr pair) => 4
(set-car! pair 3)
(set-cdr! pair 6)
pair => (3 . 6)
```

## Recursive Functions

Functions in Scheme, as in all functional languages, are very often written recursively.

```
(define length
  (lambda (lst)
    (if (null? lst)
        0
        (+ 1 (length (cdr lst)))))

(length '()) => 0
(length '(1 2 3 4 5)) => 5
(length '(()()()()())) => 5
(length '(how long is this list)) => 5
```

```
(define fact
  (lambda (lst)
    (if (= n 0)
        1
        (n * fact (- n 1)))))
```

```
(fact 5) => 120
```

**Q: Could you implement the length function using foldl and an anonymous lambda function?**

```
(define flength (lambda (lst)
  (foldl (lambda (_ n) (+ n 1)) 0 lst)))
```

In Scheme, the foldl and foldr functions are useful higher order functions, because they take a function as an argument and fold that function into a list either from the left or the right. The anonymous lambda function in flength ignores its first argument, because it doesn't care what is in the list, only how many elements there are.

```
(foldl f base '(x1 ... xn)) => (f xn ... (f x1 base))
(foldr f base '(x1 ... xn)) => (f x1 ... (f xn base))
```

## Higher-Order Functions

A **higher-order function** is a function that takes one or more functions as arguments, and possibly returns a function as a result.

For example, the `foldl` function we saw earlier is a higher-order function, since one of the arguments to the function is another function:

```
(foldl + 0 '(1 2 3 4 5))
```

A commonly used function in Scheme is the `map` function, which maps a function across a list:

```
(map abs '(-1 2 -3 4 5)) => (1 2 3 4 5)
```

How do we implement a higher-order function like this recursively?

```
(define map (lambda (fun lst)
  (if (null? lst)
      0
      (cons (fun (car lst)) (map fun (cdr lst))))))
```

```
(map square '(1 2 3 4 5)) => (1 4 9 16 25)
```

```
(map (lambda (x) (* x x)) '(1 2 3 4 5)) => (1 4 9 16 25)
```

**Q1: Can you implement a tail recursive version of `map`? Try it for homework!**

**Q2: Can you implement `map` using `foldl`? Try it for homework!**

**Q3: Can you implement `foldl` as tail recursive function? What about `foldr`?**

## Homework Assignment

1. For each of the following simple Scheme expressions, what result is printed by the interpreter? Assume the expressions are to be typed into the interpreter in the order in which they appear:

```
10
`(a b c d e)
"hello, world"
(and)
(or)
(or #f #f #f #f #f #t)
(not (not #t))
(and #t #t (not #f) (or #f #t))
(+ -1 2 -3 4 -5)
(- 9 1 (* 3 4) (/ 6 2))
(+ (* 2 4) (- 4 6))
(define a 3)
(set! a 5)
(define b (+ a 1))
(+ a b (* a b))
(= a b)
(if (and (> b a) (< b (* a b))) b a)
(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))
(+ 2 (if (> b a) b a))
(* (cond ((> a b) a)
      ((< a b) b)
      (else -1))
   (+ a 1))
((lambda x x) 1 '2' three 4 "five")
`(a . (b . (c . ())))
```

## Homework Assignment

1. Work out the answers before you type them in.

```
(car (cdr (list 1)))
(cdr (car '(a)))
(car (cdr (car '((a b c) 1 2 (3)))))
(reverse (list 1 2 3 4 5))
(list? '())
(list? a)
(cons 'a '(1 2 3 4))
(cons '(a b) '(c))
(cons 5 ())
(cons 5 6)
(cons 5 (cons 6 '()))
(cons 5 (cons 6 (cons 7 8)))
```

2. Implement a function that reverses the elements of a list. For example:

```
(reverse '(1 2 3 4 5)) => (5 4 3 2 1)
```

3. Show what happens when the following function is evaluated with b positive, negative and zero and explain how the procedure works:

```
(define (a-plus-abs-b a b) ((if (> b 0) + -) a b))
```

4. Define a function using a lambda for the following mathematical equation, making sure that the common subexpressions are defined as local variables using a let/let\* expression. Then Redefine the function using just lambda expressions without a let/let\* expression. Show that both implementations evaluate to the same results.

$$f(x, y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y)$$

5. Go through the Scheme code examples handed out in class and ensure that you understand how they work. The best way to do this is to try to figure out by hand the answer and then type them into an interpreter to verify that you got it right.
6. **Turn in a listing of your code for problems 2 & 5 along with examples that show they evaluate properly.**