Since this course is lab-based and programming-intensive, we’ll need to be quite familiar with the most commonly used features of the programming language that we share, which is Scheme. We’ll use a subset of the language as specified by the Revised Report on the Algorithmic Language Scheme.

Scheme is an expression-based language, and every step in the execution of a Scheme program is the evaluation of an expression. Usually, the process of evaluation results in the computation of a single value, although Scheme also provides for cases in which evaluation results in more than one value, in no values, in an “unspecified” value, in the raising of an exception, or in some other transfer of control flow.

Most composite expressions in Scheme are constructed on the same pattern: a list of the (syntactic) constituents of the expression, separated (typically) by whitespace and enclosed in parentheses. I’ll use the term S-structure to refer to this pattern of construction.

Many of the constituents of Scheme expressions are identifiers. Identifiers can be used as names for storage locations (that is, as variables), as syntactic keywords, or as designators for values of Scheme’s “symbol” data type. Scheme allows identifiers to contain letters, digits, hyphens, underscores, and many other punctuation marks, provided that the initial character is not a digit, plus sign, hyphen, or full stop (but the three specific variables +, -, and \ldots are allowed as exceptions to this rule).

Here are some examples to illustrate the variety of possible identifiers: q, lambda, λ, soup, list->vector, a34kTMNs, ->-, fee-fie-fo-fum, $amount, tax%, *page-number*, foo.bar.baz, index/count, matrix@, and ~^~.

Our subset of Scheme includes expressions of seven general types: literals, variables, sequencing expressions, lambda-expressions, procedure calls, conditionals, and local binders.

(1) A literal is a constant, directly depicting its value. Scheme provides literals with special, idiosyncratic forms for Boolean values (\#t and \#f), numbers (numerals, such as 42, 22/7, and -1.98), Unicode characters (such as \#\A for the Latin capital letter A), and character strings (such as "Hello, world!"). In addition, one can form a literal for a Scheme symbol by prefixing an apostrophe (pronounced “quote”) to the identifier that designates it (thus 'foo denotes the symbol foo).

The apostrophe actually indicates that the expression to which it is attached should not be evaluated, because it already depicts the result of the evaluation process. So one can use a quote to form a literal denoting a list by placing the elements of the list in an S-structure and prefixing a quote. For example, '(\#t 42 \A foo) is a literal denoting a list of four elements: the true Boolean value, the integer 42, the Latin capital letter A, and the symbol foo.

(2) A variable is the name of a (notional) storage location. The value of the variable is whatever is stored in the location that it names. Scheme allows any value to be stored in any location, without regard to its type, so variables are untyped.

(3) A sequencing expression is an S-structure comprising the syntactic keyword begin and one or more subexpressions, collectively called the body of the begin-expression. When a begin-expression is evaluated, each of its subexpressions is evaluated, in order, and whatever values the last subexpression yields become the values of the entire begin-expression. (The values of the other
subexpressions are discarded, so normally those other subexpressions are evaluated only for their side effects.)

The subexpressions of a `begin`-expression are typically placed on separate lines and indented, thus:

```
(begin
  (display "The sum of 7 and 5 is ")
  (display (+ 7 5))
  (display ".")
  (newline))
```

(4) A `lambda`-expression is an S-structure comprising the syntactic keyword `lambda`, a *formals specification*, and one or more subexpressions, collectively called the *body*. The value of a `lambda`-expression is always a procedure. The body of the `lambda`-expression is not immediately evaluated, but instead stored for possible subsequent evaluation, when and if the procedure is *invoked*. The formals specification contains zero or more variables (the *parameters* of the procedure) and defines the interface through which values supplied when the procedure is invoked are made available inside the body of the procedure.

For instance, consider the `lambda`-expression

```
(lambda (num)
  (* num num))
```

The formals specification is `num` and the body is `(* num num)`.

A formals specification can have any of three forms. The most common is an S-structure comprising zero or more parameters. In this case, when the procedure denoted by the `lambda`-expression is invoked, the caller must supply exactly one value for each of these identifiers, and these values are stored into newly allocated storage locations before the body of the procedure is evaluated. The `lambda`-expression shown above illustrates this kind of formals specification.

Alternatively, the formals specification can consist of a single variable (not enclosed in parentheses). In this case, when the procedure denoted by the `lambda`-expression is invoked, the caller may supply any number of values, which are collected into a list. This list is then placed in a newly allocated storage location bound to the variable before the body of the procedure is evaluated. For example:

```
(lambda arguments
   (/ (apply + arguments) (length arguments)))
```

The third kind of formals specification is a hybrid of these: an S-structure containing two or more variables, with a full stop (also set off by whitespace) between the last two variables. In this case, the caller supplies a value for each of the variables preceding the full stop and zero or more additional values. These additional values are collected into a list that becomes the value of the post-full-stop variable before the body of the procedure is evaluated. For example:

```
(lambda (first . rest)
  (display first)
  (for-each (lambda (arg)
                     (display #\space)
                     (display arg))
            rest)
  (newline))
```

The subexpressions making up the body of a `lambda`-expression are typically placed on separate lines and indented, as shown above.
(5) A procedure call is an S-structure comprising one or more subexpressions. The evaluation of a procedure call begins with the evaluation of all of the subexpressions, in arbitrary order. The value of the first expression must be a procedure, which is invoked and given the values of the remaining expressions as arguments, as described above. After the parameter bindings are in place, the expressions that make up the body of the invoked procedure are evaluated, in order, as in a sequencing expression. The values of the last expression in the body of the procedure that is invoked become the values of the procedure call.

Note that a procedure call can have more than one value, if it occurs in a context where all of the values can be used—typically, in a binding specification within a let-values expression.

Scheme’s base language and standard libraries contain a few hundred variables that are, in effect, bound before program execution begins to storage locations containing built-in procedures. Calls to these predefined procedures have the same general form as calls to user-defined procedures, and (from the programmer’s point of view) they are invoked in the same way. The built-in procedures of the base language are described in chapter 6 of the Revised Report on the Algorithmic Language Scheme. Here are the ones that will be particularly important in this course:

- Arithmetic operations on exact values: +, -, *, quotient, remainder, modulo, abs, gcd, max, min, expt.
- Arithmetic predicates: <, >, <=, >=, zero?, negative?, positive?, odd?, even?.
- The Boolean predicate not.
- List procedures: cons, car, cdr, cadr, list, length, append, reverse, list-ref.
- Character predicates: char=??, char<?, char<=?, char>?, char>=?.
- String procedures: string, string-length, string-ref, substring, string-append, string-copy.
- String predicates: string=??, string<?, string<=?, string>??, string>=?.
- Higher-order procedures: map, apply, for-each.
- Type conversions: string->number, number->string, string->list, list->string, string->symbol, symbol->string, char->integer, integer->char.
- Programming aids: error.

Procedure calls are normally written on single lines if possible; otherwise, line breaks are inserted between subexpressions.

(6) Scheme supports several kinds of conditional expressions. The most basic of these is the if-expression, an S-structure comprising the syntactic keyword if and three subexpressions (the test, the consequent, and the alternate).

In the evaluation of an if-expression, the test is evaluated first. This evaluation must yield a single value. Normally this value is one of the two Boolean values, the true one or the false one; if the test expression has a non-Boolean value, that value is said to be truish.

If the value of the test expression is the true Boolean value, or any truish value, then the consequent is evaluated, and the resulting values are the values of the entire if-expression. If the value of the test is the false Boolean value, then the alternate is evaluated, and the resulting values are the values of the entire if-expression. Thus the values of an if-expression are specified by two cases that are distinguished by the value of the test expression.

Here’s a simple example of an if-expression:

(if (positive? residue)
    residue
    (+ residue modulus))
Sometimes one wants to choose conditionally among three or more cases, each selected by a separate condition. The **cond**-expression is designed for this situation. It is an S-structure comprising the syntactic keyword **cond** and one or more **cond-clauses**. Each cond-clause is itself an S-structure comprising one or more subexpressions, of which the first is the test and the others make up the action.

The evaluation of a **cond**-expression consists of the evaluation of the test in each cond-clause until one is found to have a value that is the true Boolean value or a truish value; along the way, the evaluation of each test must yield a single value. When such a test has been detected, the subexpressions in the action of the same cond-clause are evaluated, in order, as if they were the body of a sequencing expression, and the values of the last of those subexpressions are the values of the entire **cond**-expression. (If the action in the selected cond-clause does not have any subexpressions in it, then the value of the entire **cond**-expression is unspecified.)

If the value of each of the tests is the false Boolean value, then the value of the **cond**-expression is unspecified. To make it easier to avoid such cases, the last cond-clause in a cond-expression can contain the syntactic keyword **else** in place of a test. If such a cond-clause is reached during the evaluation of a **cond**-expression, the subexpressions making up its action are evaluated, and again the values of the last of these subexpressions are the values of the **cond**-expression.

Here’s an example of a **cond**-expression:

```
(cond ((>= score 95) 'A)
      ((>= score 88) 'B)
      ((>= score 80) 'C)
      ((>= score 70) 'D)
      (else 'F))
```

A third kind of conditional, the **and**-expression, is an S-structure comprising the syntactic keyword **and** and zero or more subexpressions. The evaluation of an **and**-expression consists in the evaluation of its subexpressions, in order, until either one of them is found to have the false Boolean value or all of them have been evaluated. In the former case, the value of the **and**-expression is also the false Boolean value. In the latter case, the value of the last subexpression is the value of the entire **and**-expression. (If there are no subexpressions, the value of the **and**-expression is the true Boolean value.) Each of the evaluated subexpressions must have a single value.

For example:

```
(and (integer? something)
     (not (negative? something))
     (exact? something))
```

A fourth kind of conditional, the **or**-expression, is a kind of dual of the **and**-expression. It is an S-structure comprising the syntactic keyword **or** and zero or more subexpressions. The evaluation of an **or**-expression consists in the evaluation of its subexpressions, in order, until either one of them is found to have a value that is not the false Boolean value or all of them have been evaluated. In the former case, the value of the entire **or**-expression is the value of the non-false subexpression. In the latter case, the value of the entire **or**-expression is the false Boolean value. Each of the evaluated subexpressions must have a single value.

For example:

```
(or (null? ls)
    (null? (cdr ls))
    (null? (cddr ls)))
```

In conditional expressions, each subexpression after the first is typically placed on a separate line, and the subexpressions are indented so as to form a vertical column, as in the examples shown above.
(7) A local binder is an S-structure comprising one of the syntactic keywords `let`, `let*`, `letrec`, `letrec*`, `let-values`, and `let*-values`; a binding list; and a sequence of one or more subexpressions, similar to the body of a `begin`-expression. When a local binder is evaluated, a fresh storage location is bound to each of the variables in its binding list, and values obtained by evaluating subexpressions within the binding list are placed in those storage locations; then the subexpressions in the body are evaluated in order, as in a sequencing expression, and the values of the last subexpression become the values of the entire local binder.

The various local binders differ mainly in the mechanics of how, and in what order, values are recovered from the subexpressions inside the binding list and placed in the storage locations to which the local variables are bound.

In a `let`-expression, the binding list is an S-structure comprising zero or more binding specifications. Each binding specification is also an S-structure, this one comprising a variable and a subexpression called an initializer. The evaluation of the `let`-expression begins with the evaluation of all of the initializers, in unspecified order (so that none of them can presuppose or depend on any of the others), after which the value of each initializer is placed in the storage location bound to the corresponding variable. Then evaluation proceeds with the body of the `let`-expression.

Here’s an example of a `let`-expression:

```
(let ((numerator (* 35 scale))
      (denominator (* 10 back-scale)))
   (/ numerator denominator))
```

A `let*`-expression contains a binding list with the same syntactic structure, but processes it differently. The binding specifications are taken up in order. As each one is reached, its initializer is evaluated and the result is placed in the storage location associated with its variable. This binding now becomes available during the evaluation of subsequent initializers. The effect is that each binding specification creates a new scope containing the subsequent binding specifications as well as the body of the `let*`-expression. (In a `let`-expression, by contrast, there is only one new scope, and it contains only the body of the `let`-expression.)

For example:

```
(let* ((width (* font-size (string-length str)))
       (height (* width
                 (apply max (map vertical
                                  (string->list str))))))
   (draw-background-box! start-x start-y width height bg-color)
   (draw-string! start-x start-y str text-color))
```

A `letrec`-expression again contains a binding list with the same syntactic structure. This time, however, all of the new variables are bound to storage locations first, before any of the initializers are evaluated; then the initializers are evaluated, in unspecified order, in a scope that includes the new bindings. At first glance, this might seem pointless, because values are placed into the newly allocated storage locations only after all of the initializers have been evaluated. However, the initializers can be lambda-expressions, and the newly bound variables can occur in the bodies of those lambda-expressions. This makes it possible to have local names for recursive procedures.

For example:

```
(letrec ((sum (lambda (ls)
                (if (null? ls)
                    0
                    (+ (car ls) (sum (cdr ls))))))
   (sum '(3 6 9 12)))
```
The letrec*-expression combines the features of the let*- and letrec-expressions: As in a let*-expression, each binding specification in a letrec*-expression is dealt with separately, in order, and each creates a new scope that contains subsequent binding specifications as well as the body. As in a letrec-expression, a fresh storage location is bound to the variable first, and the evaluation of the initializer is performed in a scope that includes the new binding, so that the variable's value can be a recursive procedure.

Each initializer in a let-, let*, letrec-, or letrec*-expression must have a single value.

In a let-values-expression, each binding specification contains a formals specification instead of just a variable. The initializer, which in this case is typically an expression that has multiple values, is evaluated and the resulting values are associated with the variables that occur in the formals specification, exactly as in the invocation of a procedure. The order in which the binding specifications are processed is unspecified, and there is only one new scope, containing just the body of the let-values-expression; all of the identifiers in the binding list are bound in this scope.

For example:

```scheme
(let-values (((left-weight left-height) (explore (lt tr)))
    ((right-weight right-height) (explore (rt tr))))
  (values (+ left-weight right-weight (internal-weight tr))
    (+ (max left-height right-height) 1)))
```

The idea is that the explore procedure will return two values, and the two identifiers in each formals specification will be bound to storage locations containing those values within the body of the let-values-expression.

The let*-values-expression combines the features of let* and let-values-expressions. As in a let-values-expression, each binding specification in a let*-values-expression contains a formals specification and an initializer, typically one that has multiple values. As in a let*-expression, each binding specification in a let*-values-expression is dealt with separately, in order, and each creates a new scope that contains subsequent binding specifications as well as the body.

There is one final variation on the let-expression, a local binder called the named let-expression. In this construction, a variable appears between the syntactic keyword let and the binding list. When the named let-expression is evaluated, a procedure value is constructed, with the variables from the binding list as its parameters and the body of the named let-expression as its body. This procedure value is then placed in a fresh storage location bound to the extra variable. Next, the binding list is processed, just as in any other let-expression. Finally, the body of the named let-expression is evaluated, and the value of the last expression in that body becomes the value of the entire named let-expression. The feature that makes named let-expressions so useful is that the scope of the binding for the extra variable includes the body of the named let-expression, so that the implicitly defined procedure can be recursive. This makes it straightforward to create single-use recursive procedures in-line.

```scheme
(let loop ((rest ls)
     (so-far 0))
  (if (null? rest)
      so-far
      (loop (cdr rest) (+ so-far (car rest)))))
```

In a local binder, the binding specifications after the first are typically placed on separate lines and indented to form a vertical column, and the subexpressions in the body are also placed on separate lines and indented uniformly. This convention is illustrated in the preceding examples.
Exercises

1. Write a Scheme character literal that names the reverse solidus (backslash) character.

2. Is 1+ an identifier that could be used as a variable in Scheme? What about *2, $3, or =>4?

3. Write a lambda-expression that denotes a binary predicate (that is, a two-argument procedure that normally returns a Boolean value) testing whether the point specified by its arguments, considered as x- and y-coordinates in the Euclidean plane, lies within the unit circle centered at the origin.

4. Write a conditional expression that has the null string as its value if the value of the variable quantity is exactly 1, and otherwise has as its value a one-character string in which the only character is the Latin small letter s.

5. Write an expression (in Scheme) that computes a root of the polynomial $18x^2 + 132x + 242$ using the quadratic formula.

6. Write a letrec-expression that computes the square root of 113 using Newton’s method.