

# Scheme for the Faint of Heart

This handout provides a simple introduction to Scheme for students who are already familiar with programming, but either unfamiliar or uncomfortable with Scheme. Scheme has a reputation for being a difficult and confusing language. There are definitely things about Scheme that are painful for new programmers, the best known example being its use of parentheses. However, in my experience, students who are already mature programmers get past these problems fairly quickly. They may not particularly like the parentheses, but they learn to deal with them quite readily.

In a sense, the real problem students have with Scheme is that it's too easy. In many introductory programming courses, you spend enough time figuring out compiler error messages and finding memory leaks that you never really get to write complicated programs. However, in languages like Scheme, there aren't a lot of compiler errors to figure out and you can't have memory leaks, so your instructor is free, for better or worse, to go on and spend time talking about complicated programming techniques such as constraint propagation and memoization. When I've talked with students who don't like Scheme, none of the things they say they dislike, except the parentheses, are actually features of Scheme. They say they don't like things like recursion, pattern matching, and constraint propagation. Those aren't Scheme things, those are programming things. It's just that writing a constraint propagator in C would be hard enough that no one would ever try to do it in an undergraduate course.

The bad news for these students is that we're going to use Scheme for this course because it would be a pain to write most of our code in C. Another reason is that Scheme is one of the few languages small enough to let you fit an integrated development environment onto a robot with 4MB of RAM. Another choice would be FORTH, but then we'd go insane trying to write an inference engine. The good news, however is that the focus in this class is not going to be on recursion, tree matching, or cdr'ing down lists (at least until the advanced parts of the course). Most of the Scheme code we write will be pretty mundane.

In point of fact, *no* off-the-shelf programming language is particularly well suited to programming robots, for reasons that will hopefully become clear in the class. Most agent architectures are just special-purpose programming languages. Some have very sophisticated features, such as non-deterministic search, while others are stripped down with just the right features to write sensory-motor loops. Either way, the only way to really understand them is to understand what their features are, how they're compiled, how they're executed, what requirements they make on the underlying computing hardware, and, of course, what they have to do with robots. Unfortunately, understanding how to write a real compiler requires learning a lot of material that is really tangential to the topic at hand. While we will cover a certain amount of this in the advanced parts of the course, it is silly to spend three weeks teaching the theory of LALR parsing in a robotics course.

The alternative is to use a language that is sufficiently malleable that it can be used to *emulate* whatever programming languages we want. That's really what Scheme is good for. It's a language that already has a parser built into it (in fact, that's why the parentheses are there), as well as user definable compile-time rewrite rules, and a sufficiently powerful object model to allow you to emulate the object models of most other programming languages.

## Prefix notation

The most common complaint about Scheme is its use of *prefix notation*. Normal algebraic notation puts operators like + and \* between their arguments, as in:

$$a * b + c$$

This is called *infix notation*. There are two big advantages of infix notation:

1. It's familiar
2. You can establish *precedence* conventions, such as "multiplication and division before addition and subtraction," that make it clear that "a\*b+c" means "(a\*b)+c" rather than "a\*(b+c)," thus saving some parentheses.

However, there are some problems with infix notation too. For one thing, it can be difficult to remember what has precedence over what. For example, in C or C++, you get such gems of perspicuity as

$$(*--p->foo+1)(2, 4)$$

which involves the following operations:

- Decrementing a pointer
- Dereferencing a pointer
- Dereferencing a pointer to a struct with a member foo
- Adding 1 to a pointer, and
- Calling a procedure (specified by a pointer) with the arguments 2 and 4

But the only way to know the order in which these operations will be performed is to memorize the precedence rules for C and C++.

Another problem with infix notation is that you can only use it for a few operations because you have to build knowledge of the set of infix operators and their precedences into the compiler. Any additional operations (e.g. user-defined procedures) you want to add have to be done in *prefix notation*, in which the name of the operation always goes first<sup>1</sup>:

$$\text{myprocedure}(a, b, c)$$

Prefix notation has the advantage that it's extensible: you can always add new operators to it. As long as you always include parentheses around the arguments, you can always tell which word is the operator and which expressions are the arguments to which operators.

For better or worse (most Scheme programmers think better, most C programmers think worse), Scheme uses prefix notation for everything, even simple arithmetic expressions. So instead of saying

$$a * b + c$$

we say

$$+(*(a, b), c)$$


---

<sup>1</sup>C++ overloading does allow you to define new infix operators, in a sense, as long as they're all spelled +, -, \*, etc. It has definite advantages, but it also means you have to remember whether << means push or pop when applied to a stack.

which people understandably find illegible. On the other hand, as programmers we learned a long time ago that indentation is a good thing. If we write the expression as:

```
+ (* (a, b)
  c)
```

then it's easier to see what's going on.

Now, this isn't really the way you'd write it in Scheme; it's the way you'd write it in C, if C didn't have + and \* as infix operators. In Scheme, you still use prefix notation, but you use spaces instead of commas to separate the arguments and you put the open parenthesis *before* the operator, not after:

```
(+ (* a b)
  c)
```

This makes the notation a little more uniform and makes long argument lists a little shorter.

But why write it this way? After all, it's considerably more verbose, and probably a lot less readable than just saying "a\*b+c". Some programmers will tell you they actually find prefix notation more intuitive and readable than infix notation, even for simple examples like a\*b+c. I don't know how many programmers would claim this, but certainly most beginning students don't agree. On the other hand, when we get to more complicated expressions like our friend

```
(*--p->foo+1) (2, 4)
```

(yes, this is valid, meaningful C code) and translate them into ersatz Scheme code:

```
(functioncall (+ (-> foo
                  (* (-- p)))
              1)
              2
              4)
```

then it's a lot clearer what's being done to what. That may just be because of the use of indentation, but it's hard to indent the infix code in a meaningful way. The best I can come up with is

```
(*--p
  ->foo
  +
  1)
(2, 4)
```

which seems worse than the original.

You may or may not find yourself liking the Scheme notation. It's certainly not a requirement of the course. Personally, I started out as a C programmer and learned Scheme relatively late in life. Now I find C code painful to read. However, your mileage may vary.

## Other differences from Algol-style languages

### *Manifest typing*

As with any programming language, the basic entities in Scheme are

- *Data types* (integer, string, float, *etc.*),
- *Data values* (1, "blah blah blah", 3.1415927, ...),
- *Variables* (x, y, z, myvar, ...), and
- *Procedures* (sqrt, quicksort, printf, ...)

However, there are some important differences from the programming languages you may be used to. One difference is that although Scheme has types, it doesn't have any type *declarations*. Whereas in C or FORTRAN, you say "I want an *integer* variable, x," in Scheme you just say "I want a variable, x," and then you put an integer in it. It's an integer variable as long as you store integers in it, but you could perfectly well store a float or even a string in it later on in the program. Scheme doesn't care so long as you only use the data value for operations that make sense for it. For example, you shouldn't try to perform arithmetic operations on strings or Boolean values (true and false). In the theory of programming languages, Scheme is said to have *run-time type checking*, or *manifest typing*, because the types of arguments to procedures are checked as they're used, not as they're compiled.

Manifest typing is both a blessing and a curse. There are a lot of programs that are a difficult or impossible to write without it. Compilers and interpreters, for example, have to have procedures that take an expression in the language and compute its value. Some of those expressions will have integer values in them, while others will have floating-point, or even string values. Run-time type checking is a blessing because it makes it possible to write these kinds of procedures *at all*. When you write an interpreter in a language that doesn't support run-time type checking, the first thing you have to do is implement run-time type checking yourself. On the other hand, run-time type checking is also a curse because lots of mundane programming errors that would normally be noticed by a C compiler will be missed entirely by a Scheme compiler. We live in an imperfect world.

## ***Procedures as data***

Another difference between Scheme and Algol-style languages is that it lets you freely pass procedures around as data values. While most Algol-style languages allow you to use procedures as data, they typically impose a number of restrictions on it. Scheme does not. Scheme doesn't even make a distinction between names that denote procedures, like "printf", and names that denote variables, like "x". To Scheme, they're all just variables. It's just that in Scheme, printf (or its equivalent) happens to be a variable whose value is a procedure. The same is true for +.

This has some ramifications that may make your head hurt, but we can ignore them for now.

## ***Automatic storage management***

Scheme compilers and interpreters periodically check all data structures in memory to see if they're still in use by the program. If they're not, it frees them automatically. This is called *garbage collection*. It (mostly) frees the programmer from having to do memory management. Since a large fraction of the money spent on software engineering every year is spent paying programmers to track down memory leaks, this is an important feature<sup>2</sup>.

## ***Declaring variables***

You declare a variable in Scheme by saying

---

<sup>2</sup> The alert reader may be suspicious of what "still in use by the program really means." The garbage collector will reclaim a data structure if and only if it is impossible to write a program that could access that data structure. In particular, it uses the following rules:

- Any data structure pointed to by a variable is still in use
- Any data structure pointed to by data structure still in use, is also still in use.

```
(define name initial-value)
```

This lets Scheme know that you want there to be a variable named *name* and that you want it to be set to *initial-value*. For example, you could say:

```
(define x 10)
```

to create a variable, *x*, with an initial value of 10. It's called an initial value because you can change it later on, if you want.

A note on *typesetting conventions*: we will set the names of variables in `define` statements in **boldface** to make the code easier to read. It lets you scan down the code quickly to find where one definition ends and another begins. We will also put comments, which are prefixed by a semicolon, “;”, rather than by two slashes, “//”, as in C++, in *italics*. It is possible to put `define` statements inside of other `define` statements. When we begin doing that later, we will put the names for the inner defines in **boldface italics**.

## Declaring procedures

Procedures are written using *I-notation* (“lambda notation”). In formal mathematics, the symbol *I* is used to notate functions, that is mappings from arguments to values. You might say something like

$$\text{inc} = Ix. x+1$$

to say that “inc” is the function that maps a number *x* to *x*+1. In programming languages, the same notation is used to write procedures, except the syntax is a little different. For example, most keyboards don't have a *I* key, so we spell it out as the word “lambda”. Also, since this is Scheme, we have to put things in parentheses. So we end up saying

```
(define inc (lambda (x) (+ x 1)))
```

which just says that `inc` is a procedure that takes an argument *x*, adds 1 to it, and returns the result. The general form of lambda is

```
(lambda (argument-list ...) body)
```

where *argument-list* ... is a list of the names of the arguments to the procedure and *body* is the value to be returned. Since this is a rather cumbersome notation for something as simple as the increment procedure, people generally prefer the short-hand

```
(define (name argument-list ...) body)
```

or, which you will often want to break up into multiple lines and indent

```
(define (name argument-list ...)
  body)
```

This defines *name* to be a variable whose value is a procedure with arguments *argument-list* ... and *body*. So the `inc` procedure becomes:

```
(define (inc x)
  (+ x 1))
```

Writing it this way is more readable, and also looks more like the notation used to declare procedures in C. But do remember that it's just creating a normal variable and setting its initial value to a procedure. There's no special difference between variables that are bound to procedures and variables that are bound to numbers, in Scheme. This will be important later on.

## Full support for lexical scoping

As with any civilized language, Scheme lets you define local variables within your procedures and expressions. The simplest way to do this is with another `define`. For example, suppose we wanted to compute the solutions to the quadratic equation,

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Then we would probably want to store the value of the radical (the square root) in a local variable, so we don't have to compute it twice. Then we'd just say:

```
(define (solve-quadratic a b c)
  (define radical (sqrt (- (square b)
                           (* 4 a c))))
  (values (/ (+ (- b) radical)
            (* 2 a))
          (/ (- (- b) radical)
            (* 2 a))))
```

(Scheme lets you return more than one value from a procedure if you put the magic keyword “values” in at the end. Don't worry about it for now.)

Putting the *internal define* in `solve-quadratic` creates a local variable, named “radical”, to hold the value of the radical. How does it know that when you say “radical” you mean the local variable “radical” and not some other variable radical? Well the same way that almost every other language knows, it looks for the “closest” definition of a variable named “radical” in the surrounding program text. The control of what variable definitions are visible in what contexts is called *scoping* and the technique of resolving variable scope based on location in the program text is called *lexical scoping*. Virtually all modern programming languages use lexical scoping, including C and C++. Scheme has what's sometimes referred to as “full lexical scoping,” which lets us play some neat tricks, as we'll see in a moment.

Now remember that procedure names are just another kind of variable, so that means we can have local *procedures* too. So we could make the code more concise by writing the following. Note that comments are written in Scheme with semicolons:

```
(define (solve-quadratic a b c)
  (define radical (sqrt (- (square b)
                           (* 4 a c))))
  ;; Returns one of the solutions, either the + solution or the
  ;; - solution.
  (define (solution plus-or-minus)
    (/ (plus-or-minus (- b) radical)
       (* 2 a)))

  ;; The real return value
  (values (solution +)
          (solution -)))
```

This says that you get a solution to the quadratic by computing

```
(/ (plus-or-minus (- b) radical)
   (* 2 a)))
```

and filling in `plus-or-minus` with either the `+` procedure or the `-` procedure. So we just make `solution` a procedure that takes `plus-or-minus` as a parameter and then pass in either `+` or `-` when we call it. There, you see. We *told* you procedures were just data, but did you believe us? Noooooo.....

Now here's something very weird. Suppose that for some reason we didn't actually want to return two values from the procedure, we just wanted to return one thing. Well there are a couple of ways. Most obviously, we could use an array or some kind of record structure. In Scheme, we might be more likely to use a list (see below). However, what we're trying to do is to return a single data object that will allow the user to obtain either solution (the `+` solution or the `-` solution) on demand. If you think about it, we already have such a data object, namely the procedure `solution`. We keep telling you that procedures are just data, after all. Why not return it as the value of `quadratic`?

How do we tell Scheme to return a function as a value? We just say

```
(define (solve-quadratic a b c)
  (define radical (sqrt (- (square b)
                           (* 4 a c))))
  ;; Returns one of the solutions, either the + solution or the
  ;; - solution.
  (define (solution plus-or-minus)
    (/ (plus-or-minus (- b) radical)
       (* 2 a)))

  ;; The real return value
  solution)
```

and we go ahead and call it as we would any other function. For example:

```
> (define s (solve-quadratic 1 5 1))
> (s +)
-0.208712
> (s -)
-4.79129
>
```

Here we called `solve-quadratic` to get the solution to the quadratic equation

$$1x^2 + 5x + 1 = 0$$

It returned the answer in the form of a function that computes whichever solution you specify. Internally, `solve-quadratic` stored that function in the variable `solution`. After `solve-quadratic` returned, we stored the function in the variable `s`. Then we called the function, first with the argument `+`, then with the argument `-`, to get both solutions.

At this point, you're probably wondering how on earth it can run the `solution` procedure when the procedure it was defined in, `solve-quadratic`, has long since returned. After all, the variables that `solution` wants to access, `a`, `b`, `c`, and `radical`, should be long since destroyed. The answer is that Scheme realizes they'll be needed in the future and saves them in a place where `solution` will be able to get at them. Moreover, if we call `solve-equation` repeatedly, it will keep making new copies of the `solution` procedure, each with its own set of values for `a`, `b`, `c`, and `radical`. That's why we say that

Scheme has “full support for lexical scoping.” It makes sure that procedures do what you expect them to, even when they’re local procedures whose parent procedures are long since dead.

## ***Interactive programming***

This brings up another unusual thing about Scheme: it’s *interactive*. You’re not limited to typing your program into an editor and then running a compiler over it. When you start up Scheme, it prints out a prompt and lets you type commands at it. One kind of command you can type is a `define` command:

```
> (define (inc x) (+ x 1))
```

but you can also type expressions you want it to evaluate (i.e. run) and it will run them and print out the result. For example:

```
> (inc 7)
8
>
```

One of the nice things about interactive languages is that if you make a mistake in one of your procedures, you just reenter the definition. For example:

```
> (define (inc x) (add x 1))
> (inc 7)
Error: undefined variable "add"
1> ,reset
> (define (inc x) (+ x 1))
8
>
```

Also, when your program generates an error, Scheme lets you examine the stack to find out what was called with what arguments and what values your local variables have. This can make programming a lot more pleasant because you can write a little bit of code, test it by running it directly, fix it, and write a little more code.

## **Some data types**

OK. Now you know the basic things that make Scheme and other dynamic languages useful and unusual. Now let’s get down to details. Scheme supports the usual suite of data types, plus a few others. Here they are.

### ***Numbers***

Scheme supports the same kinds of data types as anyone else. First of all, there are integers (whole numbers). Integers are written the same way in Scheme as in any other programming language – in decimal notation. Unlike most languages, however, there is no “largest” integer that is allowed. Scheme can handle integers with arbitrary numbers of digit, up to the limit of the number of digits that your computer’s memory can accommodate. Scheme also supports floating-point numbers, that is, approximations to real numbers. Again, they’re written in decimal notation as in any other programming language, for example “3.1415927.”

Scheme also supports some number types that most languages didn’t used to support. For example, rational numbers (e.g.  $\frac{3}{4}$ ) and complex numbers (e.g.  $\frac{3}{4} + 1.23i$ ). This used to be an unusual feature of Scheme (although FORTRAN always supported complex numbers), but now languages like C++ can be easily extended to support complex and rational numbers. For the most part, you shouldn’t worry about

these extra kinds of numbers. We won't need them for most of our work. But don't be surprised if you divide 3 by 4 and get a rational result rather than a floating-point result.

Here are some of the built-in procedures that Scheme provides for operating on numbers.

Scheme	Equivalent C expression	Meaning
(/ a b)	a/b	a divided by b a real or rational number
(quotient a b), (remainder a b)	a/b, a%b	Integer portion of a/b, remainder of a/b
(* a b), (+ a b), (- a b)	a*b, a+b, a-b	Same as C
(- a)	-a	Same as C
(abs a), (min a b ...), (max a b ...)	abs(a), fabs(a)	Absolute value, minimum, and maximum
(bitwise-and a b), (bitwise-ior a b), (bitwise-not a)	a&b, a b, ~a	Bit-wise logic operations on integers
(sqrt a)	sqrt(a)	Square root
(exp a)	exp(a)	Natural exponent
(log a)	log(a)	Natural log
(sin a), (cos a), (tan a)	sin(a), cos(a), tan(a)	Sine, cosine, tangent
(asin a), (acos a), (atan a)	asin(a), acos(a), atan(a)	Arcsine, arccosine, arctangent
(atan a b)	atan2(a, b)	Arctangent of a/b, but gives the right answer when b=0.

## Booleans

Scheme also supports Boolean values (truth values), namely true and false. Whereas in C, the integer 0 is used for false, and all non-zero integers stand for true, in Scheme they're a separate data type. The value true is written

```
#t
```

and the false value is written

```
#f
```

However, as in C, most procedures that accept a boolean argument will accept any value at all. If the value is different from #f, then it is assumed to mean true. Here are some Scheme operators that used Booleans:

Scheme	Equivalent C expression	Meaning
(if t a b)	t?a:b, or if (t) a; else b;	If t is true, then a, else b
(if t a)	if (t) a;	If t is true, then do a and return its value, otherwise do nothing and return value is undefined.
(or a b c ...)	a    b    c ...	Evaluate a, b, c, in turn until a non-false one is found, and return its value, if all are false, return false.
(and a b c ...)	a && b && c ...	Same, but look for a false value.

		If all are non-false, return the value of the last item, otherwise return false.
(= a b)	a == b	Returns true when a and b are equal. Only used for numbers
(eq? a b)	a == b	Returns true when a and b point at the same data structure in memory.
(equal? a b)	<i>None</i>	Returns true when a and b point at data structures that would print out identically.
(string=? a b)	C: !strcmp(a,b), C++ CStrings: a==b	True if a, b are identical strings
(number? a)	<i>None</i>	True if a is a number, that is, (number? 1) = #t (number? 1.5) = #t (number? "string") = #f
(integer? a)	<i>None</i>	True if a is an integer, that is: (integer? 1) = #t (integer? 1.5) = #f (integer? "string") = #f
(rational? a)	<i>None</i>	True if a is a rational number

<code>(inexact? a)</code>	<i>None</i>	True if a is an approximate representation of a number (i.e. floating-point)
<code>(complex? a)</code>	<i>None</i>	True if a is a complex number
<code>(string? a)</code>	<i>None</i>	True if a is a string, that is: <code>(string? 1) = #f</code> <code>(string? 1.5) = #f</code> <code>(string? "string") = #t</code>
<code>(list? a)</code>	<i>None</i>	True if a is a list (see below)
<code>(symbol? a)</code>	<i>None</i>	True if a is a symbol (see below)

(procedure? a)	<i>None</i>	<p>True if a is a procedure. i.e.</p> <pre>(procedure? 5) = #f (procedure? (+ 4 1))               = #f (procedure? +) = #t</pre> <p>This may be confusing to you. If so, more will be explained about it later.</p>
----------------	-------------	---

## Strings and characters

Scheme strings are typed like anyone else's strings: using double-quotes. However, they differ from C and C++ strings in that you don't have to worry about declaring their lengths or freeing up their storage when you're done with them. It's all taken care of automatically by the garbage collector.

Characters are different from C. In C, characters are a subtype of integer. In Scheme, they're a separate data type, although they can be converted to and from integers using `code-char` and `char-code`. Also, rather than writing the character X as 'X', as in C, it is written `#\X`.

Here are the built-in procedures that you'll used on strings in Scheme:

Scheme	Equivalent C	Meaning
(char=? a b)	$a==b$	True if a and b are identical characters. Generates an error if one isn't a character. If you don't want the error, use <code>eq?</code>
(char-code c)	(int)c	Turns c into an integer.
(code-char i)	(unsigned char)i	Turns the integer i into a character.
(string? a)	<i>None</i>	True if a is a string
(string-ref i s)	s[i]	Returns the I'th character of s.

(string-append a b)	C: { char *output = malloc(maxlen); strcpy(output, a); strcat(output, b); }	Concatenation of two strings
(substring s start end)	None	The substring of s starting at the start'th character and running to the end'th character
(string=? a b)	See Booleans section	Compares two strings

## Symbols

If you use the string constant "blah" twice in your program, then the program will have two different strings running around in it, each of which reads "blah". That means that in order to compare two string variables, the `string=?` procedure has to compare them character by character.

Symbols are essentially just like strings, but the language's run-time system goes to a fair amount of trouble to insure that there is never more than one copy of a given symbol. That means you can compare two symbols using the `eq?` primitive, which just asks whether its arguments point at the same data structure in memory. So symbols tend to be more efficient for many applications.

Symbols are written in Scheme *without quotes*. In other words, the symbol version of "blah" is just

```
blah
```

This can get confusing when you say something like

```
(eq? a blah)
```

because it's unclear whether you mean `blah`, the variable, or `blah`, the symbol. In one case, you're asking Scheme to get the argument for `eq?` by looking up the value of a variable, namely `blah`. In the other case, you're providing the argument directly by giving the data value itself, the symbol `blah`.

The way you make the distinction in Scheme is to put a *single quote* or `'` before the symbol when you mean `blah` as a symbol (i.e. as a piece of data to be used literally). When you don't put a quote on the word, then you mean it to be a variable, and you're asking Scheme to look up the variable's value.

## Example

How would you ask whether the value of the variable `a` is the symbol `blah`?

Answer:

```
(eq? a 'blah)
```

## Lists

Scheme has built-in support for handling collections of data. Any data value you can write by itself, you can also write as a part of a list of data values. Lists are also considered perfectly good pieces of data. Since we said *any* piece of data can be included in a lists, that ought to mean that you can make lists be components of other lists. You can.

Lists are written as a series of data items separated by spaces and/or line breaks, with the whole thing enclosed in parentheses so you can tell where it starts and ends. This is Scheme, after all. Here are some examples of lists:

```
(1 2 3 4 5 6)
(1 2 (3 4) 5 6)
()
(1 "this is a string" 5 +)
```

The first of these lists obviously has 6 elements in it. However, the second one only has 5 elements. It's just that one of those elements happens to be a list with two elements, itself. The third list may seem odd. It's a list with *no* elements in it. It's sometimes called *the empty list*. The last list has four elements – an integer, a string, another integer, and a symbol.

Here's a confusing list:

```
(if (< a 0) (- a) a)
```

This is a list with four elements: the symbol `if`, the list `(< a 0)`, the list `(- a)`, and the symbol `a`. This means we have the same ambiguity in writing lists that we had with symbols, and it's resolved in the same way. When you write a list as an argument an argument in your program, you put a single quote at the beginning of it and Scheme will understand that you mean the parenthesized expression as a list, and not as a piece of code to be executed.

The similarity between lists and code will be confusing for you for a while, but it will end up being extremely convenient for some of the robot programming you do.

Here are some common procedures for working with lists. Some are not official parts of the Scheme language, but they are known to all Scheme programmers and they will all be available to you in our class' programming environment. The footnotes discuss issues of how lists are implemented in Scheme. They are important for students who want to understand Scheme deeply, but will mostly be irrelevant for this course.

Procedure	Meaning
<code>(first l)</code> <sup>3</sup>	The first element of the list <code>l</code> (if any)
<code>(second l)</code>	The second element of the list <code>l</code> (if any)
<code>(third l)</code>	The third element of the list <code>l</code> (if any)
<code>(fourth l)</code>	The fourth element of the list <code>l</code> (if any)
<code>(rest l)</code> <sup>4</sup>	Returns all but the first element of <code>l</code> (if any). E.g. <code>(rest '(1 2 3)) = (2 3)</code>

<sup>3</sup> Note that `first`, `second`, `third`, `fourth`, and `rest`, are not part of the standard Scheme spec. They are normally named `car`, `cadr`, `caddr`, `caddr`, and `cdr`, respectively, for historical reasons. However, these synonyms are common and are much more legible. They are also the names most commonly used in similar languages that use lists, although some call `first` and `rest` “head” and “tail,” respectively.

<code>(list-ref n l)</code>	The n'th element of the list l (if any)
<code>(cons f r)</code>	Creates a list in which f is the first element and r is the rest of the list. That is, <sup>5</sup> $(\text{cons } (\text{first } l) (\text{rest } l)) = l$
<code>(list? x)</code>	True if x is a list
<code>(pair? x)</code>	Lists are constructed from simpler data structures called pairs. Anything that is a list is also a pair. Checking if a data object is a pair is cheaper than checking whether it's a list, so this will sometimes be used instead.
<code>(null? l)</code>	True if l is the empty list, i.e. <code>()</code>
<code>(member x l)</code>	True if x appears as an element in l
<code>(assoc x l)</code>	Usuable only when l is a list of lists. Scans l for a list whose first element is x, and returns it. For example: $(\text{assoc } 'b \ '((a\ 1)\ (b\ 2)\ (c\ 3))) = (b\ 2)$ so $(\text{second } (\text{assoc } 'b \ '((a\ 1)\ (b\ 2)\ (c\ 3)))) = 2$ Assoc, which is short for "association," is often used to keep track of correspondences between values.  If the element x does not appear in l, then assoc returns #f.
<code>(map p l)</code>	Runs the procedure p on each element of the list l, collects all the results together, and returns them as a new list. For example: $(\text{map } \text{sqrt} \ '(1\ 4\ 9\ 16)) = (1\ 2\ 3\ 4)$
<code>(for-each p l)</code>	Runs the procedure p on each element of l, but doesn't return the results.
<code>(append l1 l2)</code>	Returns the list formed by first taking all the elements of l1 and then all the elements of l2. Thus: $(\text{append } '(a\ b\ c) \ '(1\ 2\ 3)) = (a\ b\ c\ 1\ 2\ 3)$
<code>(reduce p i l)</code>	Used for computing sums and products of lists. For example: $(\text{reduce } +\ 0 \ '(1\ 2\ 3\ 4)) = 10$ $(\text{reduce } *\ 1 \ '(1\ 2\ 3\ 4)) = 24$  In general, reduce runs p on i and the first element of l, then takes the result and runs p on it and the second element of l, then take the result again and runs p on it and the third element of l, etc., until all

<sup>4</sup> This is an efficient operation to perform because lists are represented as linked sequences of pairs, each pointing at the next. When a list is assigned to a variable, it really just points at the first pair in the list. The rest operation just give you back a pointer to the pair pointed at by the first pair.

<sup>5</sup> Note that these two lists are equal?, but not eq?. In other words, cons really does create a new list, although technically only the first pair is new, the rest are shared with the list r.

	<p>the elements are used up.</p> <p>Here's another example:</p> <pre>(reduce append   `()   `(a b c)   (1 2 3)   (do re me)) = (a b c 1 2 3 do re me)</pre>
<code>(filter p l)</code>	<p>Returns a new list with all the elements of <code>l</code> for which the procedure <code>p</code> returns true. Thus:</p> <pre>(filter integer?   `(1 2 2.5 3 test "test")) = (1 2 3)</pre>
<code>(apply p l)</code>	<p>Calls the procedure <code>p</code> with the arguments in the list <code>l</code>. For example, you can't say:</p> <pre>(+ `(1 2 3 4 5))</pre> <p>because addition isn't defined for lists. Rather than passing <code>+</code> one argument, which is a list, you need to pass it the elements of the list as separate arguments, as in:</p> <pre>(+ 1 2 3 4 5)</pre> <p>However, if you call <code>apply</code> with both <code>+</code> and the list as arguments:</p> <pre>(apply + `(1 2 3 4 5))</pre> <p>then <code>apply</code> will call <code>+</code> <i>for you</i> using the elements of the list as arguments. So it's as if you had said <code>(+ 1 2 3 4 5)</code>.</p> <p>Of course, you could have used <code>reduce</code> too.</p>
<code>(any p l)</code>	<p>True if <code>p</code> returns true on some element of <code>l</code>. For example,</p> <pre>(any number? `(a b 2 c)) = #t (any number? `(a b d c)) = #f</pre>
<code>(every p l)</code>	<p>True if <code>p</code> returns true for every element of <code>l</code>. for example,</p> <pre>(every symbol? `(a b 2 c)) = #f (every symbol? `(a b d c)) = #t</pre>

## Procedures

You may have been confused above by the use of procedures as *arguments* to other procedures. When you say

```
(filter number? `(1 x 2 3 y z))
```

you're passing a procedure, `number?`, as a parameter to another procedure, `filter`. In Scheme, procedures are perfectly normal data objects. You can pass them as parameters, return them as values, and even store them in data structures. Most languages (C, C++, Pascal, Java) let you do this, at least to a limited extent. But Scheme (and other similar languages, e.g. ML) make it much easier to do. More will be said of this later. In the mean time, here are some useful procedures that take other procedures as arguments:

Procedure	Meaning
<code>(procedure? p)</code>	True if <code>p</code> is a procedure
<code>(map p l)</code>	Runs the procedure <code>p</code> on each element of the list <code>l</code> , collects all the results together, and returns them as a new list. See above.
<code>(for-each p l)</code>	Runs the procedure <code>p</code> on each element of <code>l</code> , but doesn't return the results.
<code>(reduce p i l)</code>	Applies <code>p</code> to <code>i</code> and every element of <code>l</code> . See above.
<code>(filter p l)</code>	Returns a new list with all the elements of <code>l</code> for which the procedure <code>p</code> returns true. See above.
<code>(apply p l)</code>	Calls the procedure <code>p</code> with the arguments in the list <code>l</code> . See above
<code>(some? p l)</code>	True if <code>p</code> returns true on some element of <code>l</code> . See above.
<code>(every? p l)</code>	True if <code>p</code> returns true for every element of <code>l</code> . See above.

## Vectors

Scheme also supports vectors, which are just like C arrays. Lists and vectors are functionally similar, except that lists are represented as linked sequences of pairs, while vectors are single contiguous chunks of memory that get allocated all at once. The advantage of vectors is that they let you get at a specified element in constant time, whereas lists require linear time to access an element. The advantage of lists is that they can be built up little by little, which makes procedures like `filter`, which can't predict the length of their results in advance, more efficient. Otherwise, they're largely the same.

Procedure	C equivalent	Meaning
<code>(vector a b c ...)</code>	(complicated)	Creates a new vector whose elements are <code>a</code> , <code>b</code> , <code>c</code> , <i>etc.</i>
<code>(make-vector l v)</code>	<code>p = malloc(l); memset(p, l, v);</code>	Creates a new vector of length <code>l</code> , all of whose elements are <code>v</code> .
<code>(vector-ref v i)</code>	<code>v[i]</code>	Returns the <code>i</code> 'th element of <code>v</code> .
<code>(vector-set! v i n)</code>	<code>v[i] = n</code>	Sets the <code>i</code> 'th element of <code>v</code> to <code>n</code> .
<code>(vector? x)</code>	<i>None</i>	True if <code>x</code> is a vector.

## Other data types

Most real Scheme implementations also support record types (i.e. structs) and object-oriented programming, although the syntax and procedures for records and objects haven't been standardized. We will use some of these other data types later.

## Examples

- Write a procedure to compute the mean value of a list of numbers.

**Solution:**

```
(define (sum l)
  (reduce + 0 l))

(define (mean l)
  (/ (sum l)
     (length l)))
```

- Write a procedure to compute the standard deviation of a list of numbers.

**Solution:**

We can write it either of two ways. The inefficient way is to use the direct definition of variance, which is that it's the mean of the numbers you get when you subtract the mean from each of the numbers and square them. That's the variance, the standard deviation is just the square root of the variance. So we could write it like this:

```
(define (standard-deviation l)
  (sqrt (variance l)))

(define (variance l)
  ;; Variance is the mean of the square of the elements with the
  ;; mean subtracted off.
  (mean (map square (subtract-mean l))))

;; Subtract the mean of l from each element of l.
(define (subtract-mean l)
  ;; First some local definitions.
  (define m (mean l))
  (define (less-mean value)
    (- value m))

  ;; Now the result value
  (map less-mean l))
```

Or we could just write it all as one big function:

```
(define (standard-deviation l)
  ;; First some local definitions.
  (define m (mean l))
  (define (less-mean value)
    (- value m))

  ;; Now the result value
  (sqrt (mean (map square (map less-mean l)))))
```

There's also a short-cut formula for the variance, which is that it's the square of the mean minus the mean of the squares:

```
(define (standard-deviation l)
  (- (square (mean l))
     (mean (map square l))))
```

This is pretty cool when you consider that we haven't even told you how to write a loop yet. One of the advantages of using lists and higher-order procedures like map and reduce is that it prevents you from having to write and debug loops that you would have to write in other languages. It's less efficient in terms of computer time, but it's more efficient in the use of your time. And don't worry - we won't be using this style of programming for the inner loops of our real-time robot code.

## Problems

- Write a procedure to compute the number of symbols in a list.
- Write a procedure to compute the number of integers in a list.
- Write a procedure to compute the sum of the integers in a list.
- Suppose we give you a list, `syms`, of the form:
 

```
((symbol number) (symbol number) ... (symbol number))
```

and another list, `syms`, of the form:

```
(symbol symbol ... symbol)
```

Write a procedure to compute the sum of all the numbers associated with the symbols in `syms`.

- Suppose you're given a list of lists of numbers, `lln`. Write a procedure to compute the product of all the numbers in `lln`.

## Control flow within procedures

### Conditionals

Otherwise, C-like Scheme code isn't really that different from C. For example, you can write conditional expressions. However, instead of saying

```
if (test) statement2 else statement2
```

as in C, you just say

```
(if test expr1 expr2)
```

So the syntax is really pretty simple. Note that unlike C, you can include an `if` statement in the middle of an expression. For example:

```
(+ (if (> a 0)
      a
      (- a))
    4)
```

So it serves the purpose of both the `if` statement and the `?:` construct from C (for those who know about `?:` in C).

When you want to chain lots of conditionals together, you can use `cond`. A `cond` expression consists of a bunch of "clauses", each of which has a test expression, followed by a value expression:

```
(cond (test value)
      (test value)
      ...
      (else value)))
```

Scheme runs each test expression until it finds one that's true, and then it returns the value of the associated value expression. The `else` clause is always accepted. We can write the absolute value function using `cond` as:

```
(define (abs a)
  (cond ((> a 0)
        a)
        ((< a 0)
        (- a))
        (else
         a)))
```

Of course, the `else` clause could have been removed by changing one of the other two clauses to read `<=` instead of `<` (or `>=` instead of `>`), but this way we can demonstrate the use of `else`.

The last kind of conditional expression in Scheme is the `case` expression. `Case` is the equivalent of the C `switch` statement. The syntax of `case` is:

```
(case index-expr
  ((value ...) expr)
  ((value ...) expr)
  ...
  (else
   expr))
```

Scheme computes the value of `index-expr` and finds the clause with a matching `value` and returns the value of its associated expression, `expr`. If none match, it uses the `expr` from the `else` clause.

## Assignment statements

This isn't actually a control flow issue, but this is the appropriate place to tell you how to assign a value to a variable. It's easy. Just say:

```
(set! name value)
```

and it will change the value of `name` to `value`. Note that you **cannot** use `define` to assign a new value to a variable within a program. `Define` is a declaration, not an executable statement, so you can't put it in the middle of a loop. You should use `set!` whenever you want to have a program change the value of a variable.

## Sequences of expressions

We've been lying to you. We told you that lambda expressions had an argument list and an expression, called the *body*, that gave the procedure's return value. Actually, the body can be an arbitrary sequence of expressions. When the procedure is called, Scheme will evaluate each expression, in order, and then return the value of the last expression. For example:

```
(lambda ()
  (set! x (+ x 1))
  x)
```

This is a procedure that takes no arguments and whose effect is to increment `x` and then return `x`'s value.

You will find that there are times when you wish you could specify a series of expressions to evaluate in order, but you can't. For example, suppose you wanted to write a procedure that decremented `x` and returned its value, when `x` was greater than 0, and otherwise just returned 0. You could imagine writing something like:

```
(if (> x 0)
    (set! x (+ x 1)) x
    0)
```

but this would totally confuse Scheme, since it wouldn't be clear where the *then* part of the `if` ended and where the *else* part began. So Scheme only allows one expression for each. If you want to put in a series of expressions for either one, then you can use the `begin` expression, which allows a series of expressions to be grouped together and used in places where only a single expression would normally be allowed:

```
(if (> x 0)
    (begin (set! x (+ x 1))
           x)
    0)
```

Thus it's directly analogous to `{ ... }` in C or to `begin ... end` in Pascal. Note that the value returned by a `begin` expression is the value returned by its last sub-expression.

## More on local variables

There's another way of making local variables. It's called `let` and you use it like this:

```
(let ((name1 value1)
      (name2 value2)
      ...)
    body ...)
```

It evaluates (runs) the *body* with a set of new local variables, *name1*, *name2*, etc. with values *value1*, *value2*, etc. The value returned from the `let` is the value returned from the *body*.

There are two minor differences between `let` and `define`. `Define` can only appear at the beginning of a procedure, whereas you're allowed to use `let` anywhere. Internal `defines` and `lets` are also scoped somewhat differently. The major impact of this is that you can't use `let` to create a recursive function. You have to use `define` for that.

## Loops and assignments

Scheme's looping and assignment constructs are pretty much like any other programming language's constructs. Whereas in C++, you might say:

```
for (int x=1; x<10; x=x+1)
    sum = sum+x;
```

in Scheme, you just say:

```
(do ((x 1 (+ x 1)))
    ((= x 10))
    (set! sum (+ sum x)))
```

the general form for `do` loops is

```
(do ((name initial-value update-rule) ...)
    (exit-test)
    body ...)
```

it says to make a new variable named *name* set it to *initial-value*, run the *body*, and then reset *name* to the value of *update-rule*, and rerun the *body*, over and over again, until *exit-test* becomes true. Note that you can bind several variables at once, and you can also put several different operations to perform into the *body*. For example,

```
(do ((i 0 (+ i 1))
    (sum 0 (+ sum
            (list-ref i l))))
    ((= i (length l))
     (write sum)
     (newline)))
```

will print out first the first element of `l`, then the sum of the first two elements, then the sum of the first three elements, etc. Not that anyone would ever want to do such a thing...

## Extending the syntax of the language

**This section is incomplete and stinks. I'll fix it as soon as I have a chance.**

Finally, Scheme gives us a way of extending the syntax of the language by specifying rewrite rules that convert expressions into simpler expressions at compile time. These rewrite rules are called *macros*.

You define a macro in Scheme using `define-syntax`:

```
(define-syntax name rewrite-procedure)
```

`Define-syntax` allows you to specify the rewrite rule in the form of an arbitrary Scheme procedure. The easiest way to generate a rewrite procedure is with the `syntax-rules` macro. `Syntax-rules` takes a set of patterns and a set of rewrites for each pattern and compiles them into a procedure suitable for use with `define-syntax`.

```
(syntax-rules (keyword ...)
  (pattern replacement) ...)
```

We'll discuss the *keyword* field later. For the moment, let's talk about how patterns and replacements are written. We discussed above how `if` won't let you specify multiple expressions to run for the consequence and alternative. A popular macro to add to the language is the `when` macro which lets you specify a series of expressions to run when a condition is true. We can define `when` in terms of `if` using a rewrite rule:

```
(define-syntax when
  (syntax-rules ()
    ((when test expression ...)
     (if test (begin expression ...))))))
```

This says that `when` should be expanded by changing it into an `if` whose second argument is the second argument of the `if` and whose consequence is a `begin` expression formed from the rest of the arguments to `when`. The `...` means that that `expression` can match a series of things in the source code, although we then have to paste them all in (by specifying `...` again) in the output of the rewrite rule. Since `test` doesn't have a `...`, it can only match on `if`'s arguments. The `unless` macro, which is the opposite of `when`, is defined the same way, except that we insert a call to `not`:

```
(define-syntax unless
  (syntax-rules ()
    ((when test expression ...)
     (if (not test) (begin expression ...))))))
```

Now it turns out that not that many people bother to add `when` and `unless` to the language because `cond` will let you specify a series of expressions anyway. However, suppose Scheme didn't have `cond`. Since it has `if`, we could actually add `cond` to the language by rewriting it into `if` and `begin`. We can do this using a macro that expands a single test/consequent clause of the `cond` into an `if`/`begin` pair and another call to `cond`:

```
(define-syntax cond
  (syntax-rules ()
    ((cond (test expression ...) other-clauses ...)
      (if test
          (begin expression ...)
          (cond other-clauses ...))))))
```

The macro expander will then expand the `cond` into an expression that has another call to `cond`. This may seem counter-productive, except that the new call to `cond` has fewer clauses. Eventually it will run out of clauses. Unfortunately, there's a problem here, which is that the pattern matcher won't be able to match a `cond` with no clauses to the pattern given in the rule. Fortunately, the pattern matcher in `syntax-rules` will let us specify multiple patterns and rewrite rules. It will find the first pattern that matches and use its associate rewrite rule. So we can fix up the definition to add an extra pattern to handle the empty `cond` case:

```
(define-syntax cond
  (syntax-rules ()
    ((cond)
     #f)
    ((cond (test expression ...) other-clauses ...)
      (if test
          (begin expression ...)
          (cond other-clauses ...))))))
```

This says that a `cond` with no clauses should be replaced with the constant `#f`. If the `cond` has clauses, then it won't be able to match the first pattern/replacement pair because there will be no place in the pattern to match the clauses. So the pattern matcher will continue on to the second pattern/replacement pair, where the first pattern clause will be matched to the pattern variables `test`, and `expression ...`, and the other clauses, if any, will be matched to the pattern variable `other-clauses ...`.

This leaves us with one bug in our implementation of `cond`. If we give the system the expression

```
(define (abs a)
  (cond ((> a 0)
        a)
        (else
         (- a))))
```

then we want it to generate

```
(define (abs a)
  (if (> a 0)
      a
      (- a)))
```

Unfortunately, our macro will transform it into

```
(define (abs a)
  (if (> a 0)
      a
      (if else
          (- a)
          #f)))
```

So we need to add a special case to our macro for `else` clauses:

```
(define-syntax cond
  (syntax-rules ()
    ((cond (else expression ...))
     (begin expression ...))
    ((cond)
     #f)
    ((cond (test expression ...) other-clauses ...)
     (if test
         (begin expression ...)
         (cond other-clauses ...)))))
```

Unfortunately, there's a subtle bug here. In the last clause, the pattern matcher treated `test` and `expression ...` as pattern variables to be matched to the expressions in the call to the macro. By the same reasoning, it will treat `else` as a pattern variable to be matched against the code, rather than a specific word to look for. Thus, if we give it the code

```
(cond ((> x y)
      a)
      ((< y z)
      b))
```

then it will expand it into:

```
(if (> x y)
    a
    b)
```

rather than

```
(if (> x y)
    a
    (if (< y z)
        b
        #f)))
```

which is very different.

So what we need is a way of telling the pattern matcher that when we say `else`, we really mean a specific word that can only be matched to that specific word in the input, and not just a pattern variable that can be matched to anything. This is where the *keywords* argument of `syntax-rules` comes in. Any symbols put in the *keywords* section will be treated as specific words that can only be matched to themselves, not as general pattern variables. So the corrected form of the code is:

```
(define-syntax cond
  (syntax-rules (else)
    ((cond (else expression ...))
      (begin expression ...))
    ((cond)
      #f)
    ((cond (test expression ...) other-clauses ...)
      (if test
          (begin expression ...)
          (cond other-clauses ...))))))
```

## Problems

These problems are recommended mainly for graduate students or for students who enjoy Scheme and want to learn more.

1. What would happen if we had put the `else` clause of the `cond` macro at the end, rather than at the beginning?
2. Write a macro for while loops. It should take as arguments the condition and body of the loop and it should expand either into a `do` loop (if you're new to Scheme) or into a tail recursion (if you're a Scheme hacker).
3. Write a macro, `setit!`, that let's you say things like:

```
(setit! x 4)
(setit! (car x) 4)
(setit! (cdr x) 4)
(setit! (vector-ref x) 4)
```

these should expand into `set!`, `set-car!`, `set-cdr!`, and `vector-set!`, respectively.

4. Write a macro, `push`, that adds an element to the head of a list. Make it act like `setit!` so that you can have it push onto the `car` of a list:

```
(push x 4)           ;Adds 4 to the beginning of the list x
(push (car x) 4)    ;Adds 4 to the beginning of the list pointed to
                    ;by (car x)
(push (cdr x) 4)
(push (vector-ref x) 4)
```

5. (Harder) Make your answer for 4 work when the argument to `car`, `cdr`, or `vector-ref` is an arbitrary expression. You want to make sure that it only computes the value of the expression once. Note that this is not just a matter of efficiency – the expression for the argument may include side effects, such as a call to a procedure that keeps a count of the number of times its called, which you do not want performed twice.