CS 61A Notes

Tikhon Jelvis

January 26, 2011

Contents

1 Intro to Scheme

1.1 Basics

In this course, we will be using a language called "Scheme". This is a simple, functional language of the Lisp family. The lisp family is a very old one; out of all of the currently active programming languages only Fortran is older. We are using a particular distribution of Scheme called "STk".

Scheme is a very simple and elegant language. It is interpreted, not compiled, so it has a prompt. This makes testing quick bits of code easier.

Scheme does not use infix notation; all the operators are just functions. Instead, Scheme uses *prefix* notation. (+ 1 2 3) is the same as 1 + 2 + 3.

Parentheses matter. You have to have them! Most of the time, parentheses are used to call functions; of course, there are also other applications.

There is no distinction between "activity" and "stuff" per se. Functions and data are interchangeable. This is very different from what people are used to with other programming languages; but it is a very important idea in Scheme and worth understanding fully.

1.2 Quoting Stuff

Quote (′) lets you specify words. Numbers can also be operated on like words, using the same methods like `first` and `last`. Here is a simple example:

```
(first (bf 'word)) o
```

You can also quote sentences! Sentences are just groups of words; to create a sentence by quoting you first have to surround the group of words with parentheses. Here is how you would go about it:

```
(first (bf '(This is a sentence))) is
```

1.3 Special Forms

Arguments to Scheme procedures are generally evaluated before being passed to the procedure; however, there are some exceptions (special forms):

- `define` The first argument of define doesn't get evaluated.
- `and/or` The arguments are not always all evaluated; the expressions stop being evaluated as soon as a false or true value is found, respectively. If no such values are found throughout, all of the expressions are evaluated.
- `cond` The conditions get evaluated sequentially until a true one is found; after that only that condition's corresponding expression gets evaluated.
- `if` Only one of the two latter expressions get evaluated depending on the condition.
- `lambda` This is much like define; the body does not get evaluated but is rather used to create a procedure with that body.
- `let`, `let*` This is basically a `lambda` in disguise.

1.4 Errors

As in any programming exercise, errors are to be expected; no programmer is perfect, after all. Errors are not indication of failure; they are merely temporary difficulties to be resolved.

Most often, the error message returned by STk will be worth reading; they are designed to help the programmer resolve whatever is causing the program to fail—do not ignore these messages!

The process of removing errors from code is generally referred to as "debugging", or removing bugs. Errors are called bugs for an amusing historical reason that everyone probably knows anyhow. Debugging is, as has been mentioned before, a very natural and expected part of programming.

1.5 Procedures

Procedures are the fundamental building blocks of a Scheme program. They are Scheme's equivalent of function—note that a function is not a procedure and a procedure is not a function despite their being similar. You can define your own procedures using define.

```
(define (square x) (* x x))
```

The x in (square x) is what is called a "formal parameter", or more generally simply a "parameter" or an "argument" to the procedure. These are values that are passed to the procedure when it is called; it can refer to them internally by the name specified (in this case x). You can have multiple parameters in a procedure, or you can have none.

1.6 Functions, Procedures...

As stated before, functions and procedures are not the same thing in Scheme. Some other languages have functions that are, in fact, more similar to procedures as talked about in Scheme rather than functions.

The concept of a "function" in Scheme is mainly mathematical; it is exactly the same as it is in math, in fact. Functions are rules that map the elements of one set, called the domain, to one element each of another set called the range. How they go about this is immaterial; all that matters is what each element of the domain gets mapped to. Note the word "element": this implies (rather unsubtly) that functions need not work on numbers; in fact, functions can even operate on other functions!

Procedures are Scheme's actual version of functions. In math, only the "what" is interesting—a function is defined just by what it maps to what. Computer science, unlike math, is concerned with the "how" as well, and this is where procedures come in. The simplest way to think about procedures is as a function as well as an explanation of how to get the result given the arguments.

You can have two *different* procedures represent the same function. A function is just the abstract pairing of sets whereas the procedure is *also* the steps taken to get the correct results. E.g. $f(x) = 2x + 6$ and $g(x) = 2(x + 3)$ are the same *function*—for all x, $f(x) = g(x)$—but are different *procedures* as they take different steps to get the result.

Note that, despite the fact that procedures are different from functions, programmers tend to use the two interchangeably if they are not paying attention. Usually context is all that is needed to figure out exactly what is meant. Procedures will sometimes be called functions, but functions will probably be never referred to as procedures.

1.7 Decisions

1.7.1 The `cond` Statement

Most non-trivial programs require decisions at some point. This is one of the most basic approaches to controlling the flow of a program: the program can be set to react differently to different conditions.

There are two main forms for making decisions: `cond` and `if`. `cond` is useful for more than two options whereas `if` is only really useful for decisions with two possible outcomes.

The `cond` statement works like this:

```
(cond (condition expression) (condition2 expression2) (else
alternate-expression))
```

A good example of its use is the buzz program which plays the game "buzz"—for each consecutive number, it returns the number unless the number contains a 7 or is a multiple of 7, in which case it returns "buzz".

```
(cond ((equal? (remainder n) 0) 'buzz) ((member? 7 n) 'buzz) (else
n))
```

The `cond` statement checks each of the conditions in turn. If any condition evaluates to true, `cond` evaluates the corresponding expression and stops going through the conditions.

The `if` statement is, strictly speaking, a simpler version of `cond`. It is useful in one particular case, namely when there is only one condition. It looks like this:

```
(if condition then-clause else-clause)
```

This is simpler, more readable but also less powerful than cond. Nested `if` statements should be avoided because they make the program look odd thanks to the indenting; they can often be replaced by `cond` statements.

2 Paradigms

Computer programming is, basically, very easy—even young kids can do it! Programmers have much less to worry about than other types of engineers: unlike steel beams, if statements won't collapse under pressure; procedures will not wear out with time and program will, generally, not explode violently sending shrapnel all over the room.

Now, given this, it seems that programming would be a strictly easier field to work in than any other type of engineering, and yet it isn't. How can this be? The answer lies in scale: since all of the parts of a program are simpler than those of, say, a car, it is possible to have *more* of them. A car might have hundreds of thousands of parts to it—cars are very complex examples of engineering. However, this pales in comparison to some computer programs which can span hundreds of *millions* of lines of code.

Given this scale, a large portion of computer science is about managing it. There are patterns and approaches to simplifying problems; compartmentalizing parts of the solution and working in large teams.

Programming paradigms are such methods for coping with complexity. Basically, they are fundamental design decisions that help organizing very complex aggregations of sometimes disparate code. We will be covering four paradigms in this class:

1. **Functional programming**: using functions to do everything. This is where you get to learn about what all the elitist Haskell people use. Things don't change and, broadly, there are no side-effects.

2. **Object-oriented programming (OOP)**: Here we simulate the world in terms of objects and methods! More on this later, most likely.

3. **Server-client programming**—this is the paradigm that underlies the internet!

4. **Declarative programming**—rule-based programming and the like—somewhat esoteric. Logic programming (think Prolog) probably falls into this category too.

We will only be covering the first two in significant detail; we will just brush on the other two.

2.1 Functional Programming

Functional programming is the first paradigm we will cover.

In functional programming, there are no assignments. Variables don't change values. There are no side-effects or the like. All of the functions have a nice property: they will always return the same value given the same arguments. This means that `random`, for example, is not a function in the strictly functional way.

2.1.1 Parallelism

One of the main advantages of functional programming is that it is really easy to *parallelize* the program. The myriad problems that OOP programmers run into when parallelizing simply do not exist. This is important because parallelization is the main way computers are becoming faster, mostly because engineers are running into limits on the speed of individual chips.

The reason that it is so easy to parallelize functional code is that it is very easy to break it into atomic components—the functions—that are not significantly interdependent.

2.1.2 Evaluation

There are different ways to evaluate multiple nested functions. Scheme uses *applicative* order. Here the inner-most functions get evaluated first, then whatever they return gets evaluated, moving outwards like this.

The other approach is called *normal* order. This returns the same answer as the applicative order in most cases. This one only evaluates functions near the end, after substituting everything. Basically, it arrives to the same place as applicative order but taking different steps.

As long as you use only *functions*, the two are the same. With non-functions (e.g. `random`), normal order is different from both applicative and what is expected. This is where it is useful to distinguish the *functional* functions from just procedures.

On a more pragmatic note, the two orders also have another difference: since functions get called different amounts of times in the two methods, the performance can vary. Of course, it is probably relatively easy to optimize normal order of purely functional procedures; however, there may still be a difference.

3 Functions as Data

Procedures are not different from data; they are, in Scheme, *first-class* citizens. They can be used anywhere that non-procedure data could be:

- Procedures can be given as parameters to other procedures
- Procedures can be returned by procedures
- Procedures can be part of data structures like lists.

This is part of what makes Scheme, despite its apparent simplicity, a rather powerful language; operating on procedures can lead to some very effective abstractions greatly simplifying certain programming tasks.

3.1 Generalizing Patterns

It is easy to have multiple procedures that are similar in essence but different in more specific terms. For example, take some area procedures:

```
(define pi 3.1415)

(define (square s) (* s s)) (define (circle r) (* pi r r)) (define
(sphere r) (* 4 pi r r)) ...
```

These can easily be generalized into one procedure:

```
(define (area shape r) (* shape r r))

(define square 1) (define circle pi) (define sphere (* 4 pi)) ...
```

This is one way to abstract away details. The idea of generalizing away details like this is very powerful—it clarifies what you mean and makes working with it much simpler.

3.1.1 Example: Sums, Functions as Parameters

Another good example for generalizing procedures are sums like the following:

$$\sum_{i=a}^{b} i^2 \tag{1}$$

This could be written as:

```
(define (sumsq a b) (if (> a b) 0 (+ (* a a) (sumsq (+ a 1) b))))
```

You might also have a sum of cubes, as so:

```
(define (sumcube a b) (if (> a b) 0 (+ (* a a a) (sumcube (+ a 1)
b))))
```

In general, what you want is a function that takes the sum from *a* to *b* of an arbitrary function. Now you can take advantage of the fact that procedures are interchangeable with data, making the function which you are summing one of the formal parameters of the procedure.

```
(define (sum func a b) (if (> a b) 0 (+ (func a) (sum func (+ a 1)
b))))
```

This is a much more general function that can get the sum of an arbitrary function over and arbitrary interval. You could use this like so:

```
(define (square x) (* x x)) (sum square 1 5)
```

The latter example is much more concise and readable than the first definition of `sumsq`; it also makes it much easier to get sums of other arbitrary functions.

3.2 Higher-Order Functions

A higher-order functions (in practice, procedure), is one that either takes a function as an argument or returns a function. The examples in 3.1 on page 7 are higher-order procedures because they take a procedure as the argument. Higher-order functions are one of the defining features of Scheme; they can be sued to create powerful abstractions and make programming much easier.

Many other languages do not have higher-order functions at all; the canonical example of this would be Java which really does not even have functions at all.

3.2.1 Anonymous Procedures

Let's go back to the `sum` example. We want to take the sum of a cube:

```
(sum (* x x x) 1 10)
```

This won't work! The `(* x x x)` does not mean "a function which cubes an integer"; it means "x times x time x". When the interpreter goes to try this as it evaluates the arguments of sum, it attempts this multiplication; upon finding that `x` does not exist, it will throw an error. What we really want is not "x times x times x", but rather an anonymous function that cubes its argument.

In math, there is a concept called mapping which shows the difference between named and anonymous functions. It is written as follows:

$$x \to x^3 \tag{2}$$

This means that x maps to x^3. This notation makes an *anonymous* function in math. Compare this to, say, $c(x) = x^3$, which creates a *named* function (namely c).

We can also do this in Scheme—that is, we can create anonymous procedures. This is done using the `lambda` construct. I guess it would be:

$$(\lambda\ (x)\ (*\ x\ x\ x)) \tag{3}$$

in math. However, in scheme you have to spell it out:

```
(lambda (x) (* x x x))
```

This would be an anonymous procedure that cubes a number. You can use this with the `sum` procedure defined in section 3.1.1 on page 8 as follows:

```
(sum (lambda (x) (* x x x)) 1 10)
```

3.2.2 Some Examples

Take this nice function as an example of a higher-order procedure:

```
(define (compose f g) (lambda (x) (f (g x))))
```

This procedure takes two procedures f and g and returns a new one $f \circ g$. This is a higher-order procedure as it takes two procedures as parameters and returns a procedure.

The absolute value function could then be rendered as:

```
(define (abs x) ((compose sqrt square) x))
```

Another good example, taken directly from math, is the *derivative*. Here is the code for it:

```
(define (derive f) (lambda (x) (/ (- (f (+ x 0.00000001)) (f x))
0.00000001)))
```

The important thing to note here is the domain and range of the `deriv` procedure: the domain and the range are not made of numbers; rather, they are sets of *functions*. This is exactly what a higher-order function is.

Note that functions that return other functions do *not* need to take functions as arguments. Here is a very simple example of a function like this:

```
(define (make-adder num) (lambda (x) (+ x num)))
```

This function takes a number and returns a function that adds that number to its sole parameter. It can be used like this:

```
(define (plus4 x) (make-adder 4))
```

Now `plus4` can be used as follows:

```
(plus4 3) 7
```

You can also use `make-addr` *without* defining a particular function. This is valid code:

```
((make-addr 4) 8) 12
```

3.3 Variable Scope

All named values in Scheme have some sort of scope. The named values include both formal parameters of procedures and values or functions created using `define`.

Sometimes, when computing something in a procedure, it is convenient to name values (say, the result of some subexpression) within the function. One way of doing this is creating a new procedure within the procedure; this new procedure's formal parameters are named values within the function.

However, this is suboptimal; having an extra named procedure that only gets called once and exists only to name values is not the best of options. This can be addressed by using a `lambda` instead of a named procedure.

3.3.1 The `let` Statement

Using a `lambda` to create locally named values is great, but it is not particularly readable. To address this, there is a special form called `let` that does the same thing as the outline `lambda` method, but more readably. It is used as follows:

```
(let bindings body)
```

Here the `bindings` are name value pairs. The bindings part looks like this:

```
((name value) (name2 value2)
```

A `let` statement like this:

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

really means this:

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

Since the `let` is just really a `lambda` expression, the value expressions get evaluated *before* being associated with names. Thus doing:

```
(let ((a 1) (b (+ 1 a))) (+ a b))
```

would not be valid, as when `(b (+ 1 a))` gets evaluated, the name `a` has not been defined yet.

3.3.2 A Varient of `let`: `let*`

The statement `let*` is an abbreviation of nested `let` statements. Thus,

```
(let* ((a 1) (b a) body))
```

is the same as

```
(let (a 1) (let (b a) body))
```

In practice, this means that using `let*`, unlike `let`, lets you use values defined earlier in later definitions. This statement is not defined in the book; there is no reason to ever use it (that said, there is no reason to use `let` either, strictly speaking).

4 Efficiency

We are only covering efficiency of programs for a bit here. Efficiency will be covered in much more detail in 61b. We usually care about two types of efficiency: running time and space. Running time is about how long the program takes to complete; space is how much memory it uses.

4.1 Running Time

The running time of a program is not measured in seconds or any other unit of time; this does not tell us how fast the program is—it tells us how fast your operating system and computer are as well as the program. Instead, we measure the time in abstract units which are basically constant-time, atomic procedures. We break the procedure down into its building-blocks and count those.

An Example: `squares`

Take, for example, the `squares`, a procedure which takes a sentence of numbers and returns a sentence containing the squares of those numbers. This procedure is recursive; each time it takes a certain set of steps:

1. procedure call: constant time.
2. `empty?`: constant time.
3. `se`: when the first argument is a word and the second is a sentence, this is constant.
4. `square -> *`: constant time.
5. `if`: the total time depends on the arguments; without the arguments, this runs in constant time.
6. `first`: constant time
7. `bf`: for sentences, this is constant time.

For our purposes, all of these operations are constant time. Six of these operations are needed for each recursive call of `squares`.

This means that the total time for `squares` is then about equal to $7n + 2$ where n is the number of numbers in the sentence. However, this analysis is not perfect, so the actual numbers (7 and 2) are not very useful. Ultimately, what is important is the sort of expression this is: in this case, the procedure is *linear*. This means that doubling n roughly doubles the running time. Since the coefficients only really matter when n is low, we can safely ignore them.

4.1.1 Another Example: Insertion Sort

Insertion sort is one of the simplest sorting algorithms around. It takes a sentence of numbers and returns a sentence of the same numbers now sorted in numerical order.

It is a very simple algorithm; it goes through the sentence word by word, comparing each word to the previous word, putting it into place and thus sorting the list.

This procedure is written using two procedures, one of which is the helper procedure `insert` and `sort` which is the actual sorting procedure. First, we look at the helper procedure, which is made up of:

- `cond`: constant
- `empty?`: constant
- `se`: constant
- <: constant
- `first`: constant
- `se`: constant

- `se`: constant
- `first`: constant
- procedure call: constant
- `bf`: constant

The helper method has 12 procedures in the worst case. Since it is recursive, `insert` can be said to take $12n + 3$ time, which is linear.

The sort procedure itself is next. This has a bunch of constant procedures as well as `insert`, which is linear. Thus, a single call of `sort` is linear, and `sort` is recursive, so it happens linearly with n. This linear procedure called a linear amount of times results in quadrating time, which means it runs in n^2. In practice, this means that if you double n, the running time of the program increases four-fold.

4.1.2 Notation

We do not really use numbers when we talk about the running time of a program. We don't even care about (in the case of polynomials) all of the various terms. In fact, all we really care about in polynomials is the degree. This makes sense because all we're really worried about are cases when n is really large. When this is the case, the highest degree so dominates the result that nothing else matters.

It would be convenient to be able to talk about families of running time functions like linear or quadratic. To do this, we can use something called "theta notation" which uses a capital Greek theta (Θ).

Saying that $f = \Theta(g)$ means that the two functions belong to the same "family"—that is, they are within a constant of each other. $25n^2 + 3n = \Theta(n^2)$ since they are both quadratic functions.

4.1.3 Families

Most functions encountered in programming fall into a few types of families:

1. $\Theta(1)$
2. $\Theta(\log n)$
3. $\Theta(n)$
4. $\Theta(n \log n)$
5. $\Theta(n^2)$
6. $\Theta(n^3)$

7. $\Theta(2^n)$
8. $\Theta(n!)$
9. $\Theta(n^n)$

4.2 Memory Usage

A program's efficiency is not only measured based on running time; the amount of memory used is also important. Memory usage is particularly stressed in SICP because LISP had the reputation of using much too much memory, primarily because it does everything with recursion, having no explicit iteration commands. There is some optimization, of course, but the reputation still persists.

4.2.1 A Simple Example: Count

Take the `count` procedure. This counts the number of elements in a sentence or a word by `bf`-ing throughout the word or sentence. Here is how `count` works:

```
(count '(she loves you)) ..->(+1 (count '(loves you))) ...->(+1
(count '(you))) ....->(+1 (count '())) ....<-1 ...<-2 ..<-3
```

This is a recursive process. Recursive processes take $\Theta(n)$ memory space.

Now compare this to a different, iterative process:

```
(count '(she loves you)) ..->(iter '(she loves you) 0) ..->(iter
'(loves you) 1) ..->(iter '(you) 2) ..->(iter '() 3) ..<-3
```

This iterative process runs with constant ($\Theta(1)$) memory usage. Note that the running time is not constant; rather, it is the same as the running time of the recursive process ($\Theta(n)$). The iterative process is more space efficient but runs in the same time.

The iterative process is tail-call optimized, which is what makes it iterative. This means that if you trace the procedure, it will not be able to optimize in the same way and will look like a recursive process. Tail-call optimization is part of the Scheme standard, which means that any conforming implementation of Scheme can optimize appropriate recursive procedures into iterative ones.

In practice, almost all algorithms run with, at most, $\Theta(n)$ memory usage. Using more memory than that is probably a bad sign. Note that this is all relative: the $\Theta(1)$ for the iterative process is *in addition to* the $\Theta(n)$ required to store the data (sentence) itself.

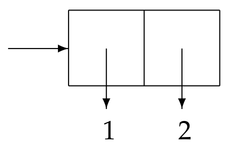
5 Data

A program is generally concerned with information; this can be acted upon, as shown earlier, and can be stored in different ways, as will be shown now.

5.1 Pairs

The most fundamental structure for storing data in Scheme is the *pair*. The pair is just what it sounds like: two pieces of data together. This is a very simple data type, but it is not as limited as it seems. Ultimately, combinations of pairs can be used to make up much more complex data structures.

A pair can be represented very simply using something called a "box and pointer diagram". Pairs are simply boxes divided into two halves with an arrow pointing to it. Here is an example for the pair `(1 . 2)`:



5.1.1 Working with Pairs

Working with pairs requires the use of several built-in procedures. The three most basic ones are: `cons`, `car` and `cdr`. The first one, `cons`, creates the pair; it is the *cons*tructor. The other two respectively return the first and second element of the pair respectively.

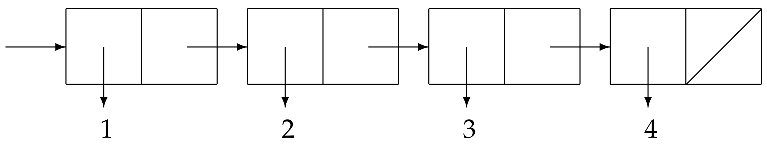
This is all that you need to know to work with pairs; they are really very simple. Of course, there is more to this: pairs a merely the building blocks of much more complex data structures.

5.2 Lists

The next most simple data structure is the *list*. A list is merely a particular arrangement of pairs. The idea is simple: the first element of each pair is whatever element makes up that part of the list; the second element of the pair is *the rest of the list*.

This can be thought of as a simple linked list: each element is actually the element and a pointer to the next element, which has a pointer to the next element *ad nauseum*. At the very end of the list, the second element of the last pair is `nil` which signifies that the list is ended.

Lists are very simple to represent using box and pointer diagrams; each pair is drawn as normally with the arrows connecting things as expected. There is one note: by convention, the last pair in the list with a `cdr` of `nil` is drawn as having a slash through the last box. Here is the list `(1 2 3 4)`:



5.2.1 List Constructors

You can operate on lists with nothing more than just `cons`, `car` and `cdr`.This is not particularly difficult:

```
(cons 'a (cons 'b nil)) (a b)
```

We get the list of two elements, 'a and 'b here.

This is a nice way to create lists, but it gets unwieldly for longer lists. In order to simplify creating lists like this; there is a nice constructor for lists called `list`. The following snippet is equivalent to the previous:

```
(list 'a 'b) (a b)
```

This is much easier to type and read than the first version.

However in the grand scheme of things, the `list` constructor is of limited utility. Another, much more useful, constructor is called `append`. This takes two lists and appends one to the other. This returns a new list with the two lists turned into one. For example:

```
(append (list 'a 'b) (list 'c 'd)) (a b c d)
```

Another very useful constructor is, once again, `cons`. The `cons` procedure can prepend elements to a list. This allows you to easily build a list up by adding things to the front. Here is an example:

```
(cons 'a (list 'b 'c 'd)) (a b c d)
```

However, `cons` does not always behave well. The main problem is trying to prepend a list: this makes the *entire* list the first element. That is,

```
(cons (list 'a 'b) (list 'c 'd)) ((a b) c d)
```

This is not usually the result that people want! This is something that is worth being careful with.

5.2.2 General Utilities

There are multiple convenient procedures already defined to work with lists. These procedures are pretty self-evident, but very useful, so they deserve mention here.

The first is `null?`. This returns whether the given list is the empty list or not. It is particularly useful for finding the ends of lists. It is used as follows:

```
(null? '()) #t (null? (list 1 2 3)) #f (null? (cdr (cdr '(1 2)))) #t
(null? 5) #f
```

The last case is worth noting: any argument at all that isn't the empty list results in `#f`, even if the argument isn't a list at all. This means that passing arguments to `null?` should not result in errors regardless of the arguments' types.

Another procedure is `length`. This, much like it sounds, gets the length of the list. It works like this:

```
(length '(1 2 3 4)) 4 (length '('(1 2) '(3 4))) 2
```

The last case shows that each sub-list only counts as one element. It is also worth noting that, with Scheme lists, `length` runs in *linear* time because it has to `cdr` through the entire list to count all of the elements.

5.2.3 Higher-Order Utilities

There are also some utility higher-order procedures for lists. These are equivalent to procedures created for words and sentences. The `every` procedure is the same as `map`; `keep` is the same as `filter` and `accumulate` is the same as, well, `accumulate`.

The `map` procedure takes a procedure and maps it to the list; that is, it applies the procedure to each element in the list. This is similar to the for-each loop constructs in other languages, but more functional.

The `filter` procedure selects items from a list based on a given criteria. The criteria is defined by a predicate, that is a function that, given an element, returns either true or false. Given a list and a predicate, filter will return a new list with only the elements selected by the given predicate.

The `accumulate` procedure works exactly the same way as it was defined earlier.

5.2.4 List Selectors

The most basic list selectors are, naturally, `car` and `cdr`. The first selects the first element of the list; the latter selects the rest of the list. They can be combined to find arbitrary elements in the list; for example, the following snippet gets the fifth element in a list:

```
(car (cdr (cdr (cdr (cdr list)))))
```

This can sometimes get inconvenient; at the very least, it will involve a lot of typing.

As ever, there is a procedure that makes life easier: Programmers do not like to type too much. This is called `list-ref`. This procedure takes a list and a number n and returns the n^{th} element of the list. Note that, like any self-respecting programming language, Scheme likes 0-based indexing. This means that:

```
(list-ref list 4)
```

is exactly the same as the previous snippet. An easy way to remember what number is needed is to count the number of `cdrs` needed to get to that element.

5.3 Calc

Calc is a very simple program that can do menial arithmetic supplied to it in prefix notation. Calc is, ultimately, a very simple, limited implementation of Scheme with a ton of details simplified out.

Like any interpreter, calc runs what is called a “read, eval, print” loop. This loop does exactly what you think it does: it reads in input, evaluates it and prints out the answer.

The main difference between calc and Scheme from a lower-level view is that it does not have procedures as first-class citizens: Procedures in calc can only be called, they cannot be passed around.

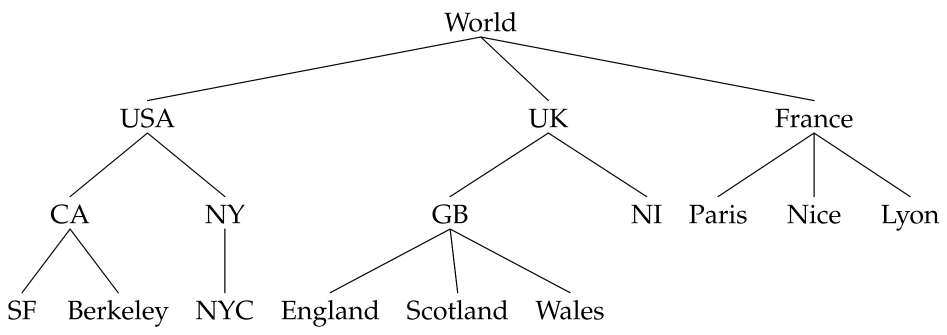
The interesting part of calc is found in the `calc-eval` procedure. This is a recursive procedure; however, it is a very different approach to recursion. Instead of simply calling `calc-eval` from itself, it is mapped to a list internally. This means that at each level there can be more than one recursive calls; in fact, there is a recursive call for each of the elements of the remaining list.

This is a very powerful approach to dealing with two-dimensional lists. Such recursion scales with the list size, letting a simple recursive call cover all of the list elements. This also shows why lisp function calls are written the way they are: in practice, the function call is just a list. Having functions like this makes implementing a Scheme interpreter in Scheme almost trivial.

5.4 Trees

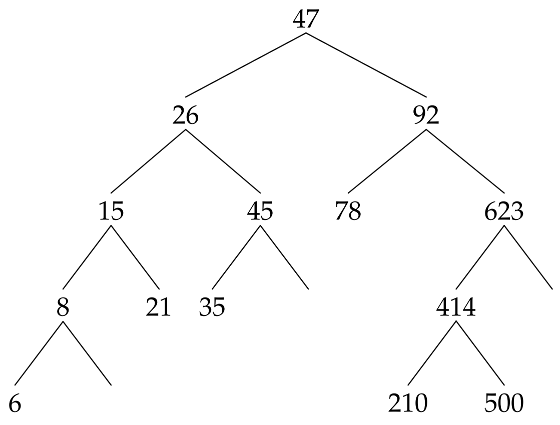
A tree is one kind of data structure that is more complex than a simple list. Most intuitively, trees can be used to describe hierarchies. These hierarchies can be very complex: each “branch” of the tree can be different, go to different depths and generally have its own properties. There can be an arbitrary amount of branches in most types of trees. Each element is called a “node”, with each node on the end being called a “leaf”.

Here is a simple example of a tree with a variable amount of children on each node:



5.4.1 Binary Trees

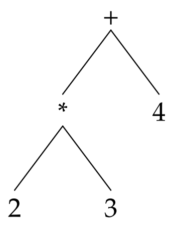
A special type of tree is a binary tree where each node can have no children, a right child, a left child or both a right and left child. A binary tree can be used to represent a binary search, creating a binary search tree. Using this tree lets you search for elements in $\log n$ time. It looks like this:



Note that some of the nodes only have one actual child.

5.4.2 Expressions

Trees are particularly useful because they can represent complex expressions like `(+ (* 2 3) 4)`. Interpreters actually do this internally when evaluating various expressions. The given expression would be represented as:



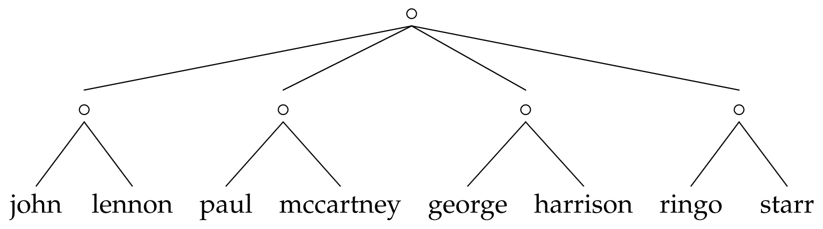
5.4.3 Trees vs Lists

Remember that a tree is an abstraction; it is not merely a special case of list. In practical terms, this means that you should *not* use `car` and `cdr` on trees!

Trees as described now will be generally referred to as Trees with a capital "t". This is to differentiate actual, abstract Trees from various tree like structures. To illustrate the point, take a list of lists:

```
((john lennon) (paul mccartney) (george harrison)
 (ringo starr))
```

This is just a list, it is *not* a Tree but it is a tree. Here is how this list could be drawn:



Note how the nodes all either have a datum or children; none of the nodes of this tree have both a datum and children. This is not a Tree because it is not abstracted as such; it is a tree because, like any deep list, it is sometimes convenient to think about it much like a tree.

Basically, thinking about lists like this lets us use the same techniques on them as we would on Trees; however, we should always be aware of the difference. The "how" may be the same but the "what" isn't": a tree may be the same as a Tree from a practical standpoint, but it is very different from a conceptual standpoint.

Here is a summary of basic tree-like structures:

	N Children	2 Children
ADT	Tree	Binary Tree
List Structure	Deep List Recursion	`car` and `cdr` recursion

5.4.4 Searching

It is often very convenient to search through. There are, basically, two approaches to searching through trees: depth-first and breadth-first. The two methods basically work like they sound.

Depth-first search goes down the list branch by branch back-tracking up the tree when it runs out of nodes in any particular branch. This is really the natural way to go through a tree as it is a hierarchical structure.

Breadth-first search is somewhat more difficult to implement than depth-first search. The basic idea is to search through the tree side-ways instead of vertically. The way to approach this is to use a queue of tasks. The queue is made up of nodes to work on along with all of their children. Each time a single node is looked through, all of its children get enqueued. This way each level is read through before continuing on to the next level.

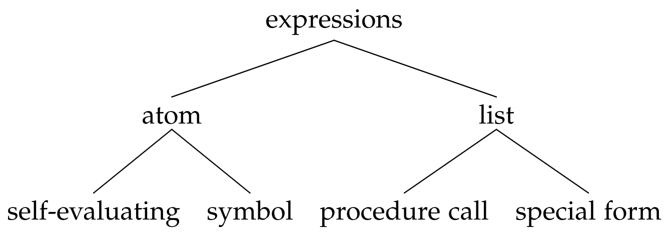
Why would we differentiate between the two? In practice, a depth-first search is often the way to go simply because it is the more simple approach. However, there are cases when breadth-first search is the better option. The canonical example is a chess game: the tree that represents all of the possible games is so large that generating the entire tree is impractical. Here, a breadth-first search lets you get the best possible position for all the data you do manage to generate.

6 A Scheme Interpreter

We are now going to look at the implementation of a relatively simple Scheme interpreter in Scheme. The interpreter we are looking at is, itself, written in Scheme. Even real Scheme interpreters are largely implemented in Scheme, so what we are looking at is not completely unrealistic.

6.1 Expression Types

There are, broadly speaking, four types of possible expressions. They are divided into two categories: atoms and lists. There are atoms and lists. The atoms are self-evaluating expressions (numbers, booleans, strings...) or symbols. The lists are either procedure calls or special forms.



The self-evaluating expressions are just themselves; this is very simple. The symbols can be variable or procedure names or global values like keywords. The lists are slightly more complicated. To evaluate a list, we first need to look at the first element. If the first element is a *keyword*, then the list is a special form, which means it has to be treated differently. However, if the first element is just a procedure, it can simply be called with the rest of the list as arguments.

The symbols are basically names. They stand for some other expression. When it gets time to evaluate a symbol, the interpreter needs to look up the value of the symbol and substitute it in the symbol's place. This is, ultimately simple, with one main complication: variable scope. Some symbols have different meanings depending on their locations; some symbols may even simply not be defined anywhere except certain procedures.

Procedure calls are not very complicated: they're exactly what they sound like. When a procedure call is found, it is the first element of the list. The interpreter then evaluates all of the following elements of list and passes them to the procedure being called as arguments.

The last type is very similar to a procedure call; the main difference is that if the procedure is actually a special form, the arguments do not get evaluated immediately. Instead, the interpreter evaluates the arguments as is necessary for the special form in question.

6.2 Tagged Data

Data floating around in Scheme is "tagged". This basically means that each little datum carries some meta-data related to it's type. A very simple implementation of this would just create a pair for each datum with the `car` of the pair being a type identifier and the `cdr` being the value of the type.

In practice, Scheme does not use such a naïve implementation of tagged data; however, the idea behind it is the same—the difference lies solely in boring operational details.

The natural advantage of tagged data is that it is much easier to write code. There is no need to worry about the type of variables or arguments; it is very easy to write heterogenous lists containing different types of values... In all, it makes life easier for the programmer. That said, there is a disadvantage—this way is inherently slower than a more statically-typed system.