This will be a small review sheet that covers the first few topics of CS3S. However, to tell the truth, you might find the SPC web site more helpful. We highly recommend you to the SPC web site — there's a great online review on the bottom of `http://www-inst.eecs.berkeley.edu/~selfpace/`. This will try to cover misconceptions and problems that people have been running into.

1 Expressions

Scheme programs are built out of expressions. For example:

```
    42
    (* 5 6)
    'i-am-a-quoted-symbol
    (= a b)
```

are all considered expressions. You might not expect it, but `+` and the other math symbols are also valid expressions:

```
STk> +
#[subr +]
STk> *
#[subr *]
STk> /
#[subr /]
```

We find that numbers are "self-evaluating" — the value of `42` is `42`. Simple enough. It's sorta like a mathematician who starts off saying $0 = 0$, but we find that this rule keeps Scheme's evaluation system simple.

Let's take a slightly detailed look at what happens when we have parentheses. In the case of `(* 5 6)`, Scheme will take the following steps to evaluate this expression:

- Evaluate the value of `*`, `5`, and `6`. `5` and `6` are self evaluating, so we stop there. `*` is a primitive function that Scheme picks up. Scheme now knows about these three elements.
- Assume the first item is a function, and apply it on the rest of the list. This means that `*` will be applied on `5` and `6`.

You should notice that there's something weird happening; isn't this definition circular? We're defining evaluation in terms of evaluation!

This rule is necessary though. What happens here?

```
    (* 2 (+ 1 20))
```

In order to compute this expression, we need to apply evaluation on `*`, `2`, and `(+ 1 20)`. Ok, let's look at `(+ 1 20)`. `1` and `20` evaluate to themselves, and `+` evaluates to a primitive function. Eventually, we "hit rock bottom" at Scheme primitives. If we follow the rules of evaluation long enough, we'll eventually get an answer.

Note — wrapping expressions with regular parentheses in attempts to get it into a list doesn't quite work:

```
STk> ( (list 1 2) )

*** Error:
eval: bad function in : ((list 1 2))
```

This is because of the rules above on evaluation: `(list 1 2)` evaluates to the list, and then the outer pair of parentheses tells Scheme to apply a function with no arguments. The next section will elaborate on this.

2 Quotation

Now that we've talked about evaluation, it's time to bend its rules with quotation. You can quote something using the quote symbol (`'`) or with `quote`.

Let's say we want to make a function, `greeting`, that takes a name and returns a nice greeting:

```
(define (greeting name)
  (list 'hello name))
```

Let's try it out:

```
STk> (greeting 'carol)
(hello carol)
STk> (greeting 'clancy)
(hello clancy)
```

Let's try something else:

```
STk> (greeting ernie)

*** Error:
unbound variable: ernie
```

Ah! According to evaluation rules, Scheme will try to figure out what `greeting` and `ernie` are. It knows about `greeting`, since we just defined it, but gets lost at `ernie`.

What happens if we change the definition of `greeting` like this?

```
(define (bad-greeting name)
  '(hello name))

STk> (bad-greeting 'voldemort)
(hello name)
```

Ooops.

- When we quote an atom (symbol or number), Scheme will evaluate it as itself.
- When we quote a list, Scheme will return a list with each element itself quoted.

Another example:

```
STk> '(enumerate and (list (a (b) c)))
(enumerate and (list (a (b) c)))
```

This actually deals with some issues. First, you can make nested lists very easily with quotation. Next, since Scheme isn't really looking up the value of things, we can use words like `list`. Because we quote `list`, Scheme will treat it just like any other quoted symbol.

This brings up the question: what happens when we do this?

```
STk> ('list 1 2 3)

*** Error:
eval: bad function in : ((quote list) 1 2 3)
```

We find out two things here — `'list` is just shorthand for the longer expression, `(quote list)`. Second of all, `'list` evaluates to the symbol "list", and not the list function, which is why this breaks.[1]

3 List manipulation

Let's talk about list manipulation a little bit. You might have heard that Scheme is a Lisp-like language. What does this mean? A small excerpt from the Jargon file (`http://www.tuxedo.org/~esr/jargon`):

> LISP n. [from 'LISt Processing language', but mythically from 'Lots of Irritating Superfluous Parentheses'] AI's mother tongue, a language based on the ideas of (a) variable-length lists and trees as fundamental data types, and (b) the interpretation of code as data and vice-versa.

So Scheme makes it easy to do stuff with lists. What sort of stuff? Obviously, we'd like to look at individual elements. Let's look at the two primary functions for getting at list elements, `car` and `cdr`.

The `car` of a list will give us its *first* element:

[1]However, it *is* possible to force evaluation of a symbol with the `eval` function: `(eval '(list 1 2))` does work. We won't cover this in 3S, but it's nice to know that it works.

```
STk> car
#[subr car]
STk> (car '(Jack and Jill))
jack
STk> (car '((Herimone) and Ron))
(herimone)
STk> (car '((Batman (and (Robin)))))
(batman (and (robin)))
```

The `cdr` of a list will give us the *rest* of the list.

```
STk> (cdr '(Donald E Knuth))
(e knuth)
STk> (cdr '((William Richard) Stevens))
(stevens)
STk> (cdr '((John McCarthy)))
()
```

Notice that if a list has one element, `cdr` will still give us the rest of the list — the empty list.

With these two functions, we can define convenient functions to reach the first, second, and third elements of a list:

```
(define (first l) (car l))
(define (second l) (car (cdr l)))
(define (third l) (car (cdr (cdr l))))
```

... and so on.[2]

Other important functions include `list`, `cons`, and `append`, which are list building functions. Other useful functions include `reverse` and `length`, which have obvious and not-so-obvious uses.

The two functions that seems to catch people off guard are `cons` and `append`. `cons` takes in 2 elements. For 3S, we'll assume that the second element is always a list. Take a look:

```
STk> (cons 'la '(vals))
(la vals)
STk> (cons '(le) '((miserables)))
((le) (miserables))
```

We can imagine the first argument squeezing into the second element.

Now let's look at `append`. `append` only works with lists. Take a look:

```
STk> (append 'foo '(bar))

*** Error:
```

[2]Scheme does provide these functions: `cadr`, and `caddr` to do the same thing as our `second` and `third`.

```
append: argument is not a list: foo
STk> (append 'foo 'bar)

*** Error:
append: argument is not a list: foo


STk> (append '(foo) '(bar))
(foo bar)
```

Append will put two lists together, and breaks otherwise. [3]

Using `list` is pretty simple: it takes in one or more arguments, and makes a list of those elements.

```
STk> (list 'olympic 'gold)
(olympic gold)
```

What catches people, initially, is the fact that it's perfectly legal to make a list with one element:

```
STk> (list 'supercalifragilistexpialidocious)
(supercalifragilistexpialidocious)
STk> (list '(list))
((list))
```

Time to start plugging advertisements. If you have any questions, you can always contact us through the newsgroup, `ucb.class.cs3s`. Details on doing this are on the SPC url above. However, if you like human contact, come by to the SPC, and we'll be glad to talk with you. Also, do come by and get the work done as quickly as you can — you do *not* want to be at the SPC during the last few weeks. Trust us: it's not fun.

Let's end with a nice quote by the late Richard Stevens:

> "This process of digging up details and learning how things work leads down many side streets and to many dead ends, but it is fundamental (I think) to understanding something new. Many times in my books I have set out to write how something works, thinking I know how it works, only to write some test programs that lead me to things that I never knew. I try to convey some of these missteps in my books, as I think seeing the wrong solution to a problem (and understanding why it is wrong is often as informative as seeing the correct solution."

Good luck!

[3] We are sorta lying through our teeth: `(append '(foo) 'bar)` $\Longrightarrow$ `(foo . bar)` doesn't technically break! (Neither does `(cons 'foo 'bar)`!) However, this is CS61A material — you don't need to worry about it for now.