Homework Review

- Is reverse engineering possible?
- Syntax vs. Semantics!
- Stages of interpretation

Lists in Scheme

Remember how a singly-linked list works:

```
1 ———> 2 ———> 3
```

How can we make linked lists in Scheme?
Using and building lists

- null is an empty list.
- For an item a and list L, (cons a L) produces a list starting with a, followed by all the elements in L.
- (car L) produces the first thing in a non-empty list L.
- (cdr L) produces a list with the first item of L removed.
- DrScheme prints the list (cons 1 (cons 2 (cons 3 null))) as (1 2 3).
- Lists can be nested.

Exercises

Using only cons, null, car, and cdr,

1. Write an expression to produce the nested list (3 (4 5) 6).
2. Write a function (get2nd L) that returns the second element in the list L.
3. Using recursion, write a function split-digits that takes a number n and returns a list with the digits of n, in reverse.
   For example, (split-digits 413) should produce the list (3 1 4).

Useful list functions

- (list a b c ...) builds a list with the elements a, b, c, ...
- cXXXr, where X is a or d. A shortcut for long expressions like (cdr (car (car (cdr L)))). → (cdaadr L)
- (cons? L) — returns true iff L is a cons.
- (null? L) — returns true iff L is an empty list.
- (append L1 L2) — returns a list with the elements of L1, followed by those of L2.
  Can you write this function?
Scheme grammar

Here is a CFG for the Scheme syntax we have seen so far:

CFG for Scheme

\[
\text{exprseq} \rightarrow \text{expr} \mid \text{exprseq expr} \\
\text{expr} \rightarrow \text{atom} \mid (\text{exprseq}) \\
\text{atom} \rightarrow \text{identifier} \mid \text{number} \mid \text{boolean}
\]

This is incredibly simple!

Scheme is lists!

Everything in Scheme that looks like a list is a list. Scheme evaluates a list by using a general rule:

- First, turn a list of expressions \((e_1 \ e_2 \ e_3 \ldots)\) into a list of atoms \((a_1 \ a_2 \ a_3 \ldots)\) by recursively evaluating each \(e_1, e_2, \text{etc.}\)
- Then, apply the procedure \(a_1\) to the arguments \(a_2, a_3, \ldots\)

The only exceptions are special forms such as \text{define} and \text{cond} that do not evaluate all their arguments.

Scheme evaluation and unevaluation

We can use the built-in function \text{eval} to evaluate a Scheme expression within Scheme!

- Try \(\text{eval} (\text{list} + 1 2)\)

We can also ask Scheme \text{not} to evaluate an expression by using the (very) special form \text{quote}.

- Try \(\text{quote} (+ 1 2)\)

There is a convenient shortcut of \text{quote}: for example, \'(+ 1 2)\).
Symbols

An unevaluated identifier is called a **symbol**.
(Note: the predicate symbol? is useful here.)

Symbols are useful beyond evaluation and quoting.
We often use them like ENUMs in C++.
Examples: units, months, grades

Symbols are often used to **tag** data: (cons 10.3 'feet)

More exercises

1. Write a function (my-and a b) that works similar to the built-in and boolean function, but returns a symbol 'true or 'false as appropriate.

2. Write a function that takes a list of numbers and adds them up using the + function. (Hint: first build this expression using cons, then evaluate it using eval.)

3. Repeat #2 using the built-in apply function.