CSE341 Spring '05 Due Wednesday, May 11
Assignment 4

Part I

Warm-up and Memoization

Warm-up

As a warm-up, write the following SCHEME functions:

1. Write the function `foldl` of the form `(foldl func initial lst)` which folds `func` over `lst` from left to right, using `initial` as the initial accumulator. `Func` should be a function of the form `(func item acc)`, where `item` is an item from the list and `acc` is the current accumulator. Be sure not to get confused with `foldr`; a good way to tell them apart is that `(foldl - 0 '(1 5))` yields `(5 - (1 - 0)) => 4`, while `(foldr - 0 '(1 5))` yields `(1 - (5 - 0)) => -4`.
2. Write a function `contains?` of the form `(contains? haystack needle)`, which returns true if the list `haystack` contains `needle` and false otherwise. Use the equivalence predicate `eqv?` to make comparisons. Use `foldl` in your solution.

Memoization

For the next few problems, consider the following situation: You have a nearly unlimited supply of 1, 7, and 11 (changed 5/8/05; originally was 13) cent stamps. You want to pay postage on a package with your stamps, where the cost of postage is x. You want to know the fewest number of individual stamps you can use to pay the exact postage. Let $f(x)$ be a function that gives you this number. Obviously, if the cost of postage, x, is zero, then $f(x) = 0$. For $x > 0$, there are several possibilities. You could assume you will use at least one 11-cent stamp, in which case you have $(x - 11)$ cents of postage left to pay. In that case, $f(x - 11)$ will give the minimum number of stamps needed for the rest of the postage, to which you add 1 for the 11-cent stamp you assumed. Likewise, you could also assume a 7-cent or 1-cent stamp. Obviously, you want to choose the one that gives you the smallest number of stamps, so you should choose the minimum of $f(x - 11)$, $f(x - 7)$, and $f(x - 1)$. This means that for $x > 0$, $f(x) = 1 + \min\{f(x - 11), f(x - 7), f(x - 1)\}$. If any of the terms in the minimum would call f with a value less than zero, you should exclude them (for instance, if x is 8, it doesn't make sense to guess that you might need an 11-cent stamp).

1. Write a recursive function `minimum-stamps` of the form `(minimum-stamps x)`, where `x` is the cost of postage, that returns the minimum number of stamps necessary to pay the postage using the algorithm above. Your function must be recursive and cannot use any additional data structures like a memoization table.
2. Write a function `minimum-stamps-memo` that is a memoized version of `minimum-stamps`. That is, it should maintain a table of previously computed answers. When called with a

value of `x` that has been seen before, it should retrieve the value from the table. When called with a value for which the answer is not in the table, it should calculate the answer and update the table. The table must persist across multiple calls to the function. Do *not* make the memo table a top-level binding. Do *not* use `minimum-stamps` in your solution. Use the built-in function `assoc` to simplify your task.

3. In a comment, answer the following questions:

   (a) How do `minimum-stamps` and `minimum-stamps-memo` compare in execution time for a very large value of `x`? How does the execution time of `minimum-stamps-memo` with a large value of `x` compare to the execution time of a second call to the function with the same `x`? Why is this?

   (b) If `minimum-stamps-memo` instead used `minimum-stamps` to calculate answers not in the memo table (which you shouldn't have done in your solution), how would it compare in terms of speed with your implementation of the function? Why?

4. **(Extra credit)** Write a function `minimum-stamps-general` of the form `(minimum-stamps-general denom x)`, where denom is a list of integer denominations of stamps available and `x` is the amount of postage to be paid. In particular, if the function were curried (which it isn't, so you don't need to write it this way), `(minimum-stamps-general '(11 7 1))` would be equivalent to `minimum-stamps`. Your function should be memoized, but keep in mind that a given memo table is only valid for a given set of denominations, so your memo table should not be saved across separate calls to the function. *Hint*: use a helper function, and make sure that the memo table persists across separate calls to the *helper*.

Part II

Infix Math Expressions

Don't Panic

For this part of the homework, you will write a few functions to convert simple math expressions in infix notation into equivalent Scheme code suitable for evaluation. Before you do, though, it's important to understand how Scheme programs are represented. Although the description of this part of the assignment is rather lengthy, the actual amount of code you have to write is minimal; it is more important that you understand the concepts involved, as they will play a large role in the second Scheme homework.

Programs as Data

A Scheme expression is represented as either an *atom* (number, string, symbol, etc.) or list of expressions, which could in turn be atoms or other lists. Consider this simple expression:

```
(foo (+ bar 42))
```

This expression consists of an outer list containing the symbol `foo` and an inner list containing the symbol `+`, the symbol `bar`, and the number `42`. Semantically speaking, SCHEME interprets this data structure as an application of the function `foo` to the result of applying the function `+` to `bar` and `42`. However, structurally it is nothing more than a list which we could create programmatically if we wished:

```
(list 'foo (list '+ 'bar 42))
```

Typing this expression in the DRSCHEME interactions window will confirm that it produces a list with the same structure (and printed representation, for that matter) as the first expression. Note that the single quote character in front of `foo`, `+`, and `bar` prevents Scheme from attempting to evaluate them as variables; instead, the symbols are inserting directly into the expression as arguments to `list`. It's often convenient to *quote* parts of a program this way to prevent evaluation:

```
(map (lambda (x) (* 2 x)) '(1 2 3 4))
```

In the above expression, the list `(1 2 3 4)` is quoted to prevent SCHEME from attempting to evaluate it as a function application. Instead, it is passed directly to `map`, and the result `(2 4 6 8)` is produced. This means we could rewrite the second expression in a more convenient form as follows:

```
'(foo (+ bar 42))
```

Note that everything within the list, including the symbols `foo`, `+`, and `bar`, are quoted and remain unevaluated. SCHEME provides a function, `eval`, which takes a SCHEME expression in list form and evaluates it, returning the result:

```
(eval '(* 2 (+ 1 1)))    =>   4
```

This provides a lot of flexibility: A SCHEME program can construct and evaluate new SCHEME expressions, or transform and analyze its own code. You will take advantage of this fact to write functions that take an expression in infix notation and convert it to an expression in prefix notation that SCHEME can evaluate.

Infix expressions

Representation

For convenience, we will define infix expressions in such a way that they can be written easily as a quoted list in Scheme. Predictably, an infix expression may consist of either an atom or a list. However, how we evaluate such an expression will differ from normal SCHEME semantics. A single atom by itself (which, for the purposes of this assignment, should only be a number) simply evaluates to itself:

```
4    =>   4
```

A list of expressions must be handled differently. In the simplest case, the list will be a *singleton* list containing only a single element, in which case it evaluates to the value of that element:

```
(5)   =>  5 ; Singleton list
((5)) =>  5 ; Singleton list of a singleton list
```

We see the first major difference immediately: extra parentheses do not matter as they normally do in SCHEME. The final case (and the most interesting to us) is a list of *tokens*, where each token is either an operator (one of `+`, `-`, `*`, and `/`), or another infix expression. Examples:

```
(1 + 1)       =>  2
(1 + 2 * 2)   =>  5
(2 * (1 + 1)) =>  4
```

The first example is simple enough: the first token is the atom 1, the second token is the operator +, and third token is also the atom 1. The second example is similar, but shows that expressions must be evaluated taking into account the normal order of arithmetic operations. In the third example, one of the tokens is another infix expression.

Evaluation

To evaluate such an expression, we would take a divide-and-conquer approach, where we split the expression into sub-expressions which are easier to evaluate and perform the appropriate operation with the results. Consider a simple case:

```
(1 + 2 * 2 + 3)
```

Because of the order of operations, the additions should occur *last*. Since we evaluate sub-expressions *first*, our sub-expressions become whatever is between the `+`'s:

```
ADD of (1) and (2 * 2) and (3)
```

Next, we evaluate the subexpressions. The first and the last are singleton lists; this happens because we split the expression (a list), into sub-expressions (which are also lists), so we end up with 1 and 3 being in lists by themselves. This might not seem obvious, but a natural implementation of a list-splitting function will do this, so be prepared for it. We extract out their elements, which are atoms, so no further evaluation is required:

```
ADD of 1 and (2 * 2) and 3
```

The second sub-expression is another infix expression. Again, we divide it into sub-expressions around the operator `*`:

```
ADD of 1 and (MULT of (2) and (2)) and 3
```

These sub-expressions are singleton lists, so we extract the elements and are left with:

```
ADD of 1 and (MULT of 2 and 2) and 3
```

Conversion

Keep in mind that your task is to convert, *not evaluate*, the infix expression. However, the pseudo-expression above rather naturally corresponds to the Scheme expression:

```
(+ 1 (* 2 2) 3)
```

Therefore, the general conversion algorithm is:

1. Find the operator in the list which should occur last in evaluation
2. Break the list into subexpressions around this operator
3. Create a SCHEME function application expression (which is just a list!) where the function is the operator used to break up the list, and the arguments are the results of converting the sub-expressions.

Provided functions

Your solution will need to detect when an expression is an atom or singleton list. You may use the following two functions for convenience when writing your code:

```
; Returns true if x is an atom
(define (atom? x)
  (not (list? x)))
; Returns true if x is a singleton list
(define (singleton? x)
  (and (list? x)
       (null? (cdr x))))
```

Problems

Only two:

1. Write a function `split-by` with the form `(split-by splitter expression)`, where `expression` is a list and `splitter` is a symbol. `Split-by` splits the list into sub-lists occurring between instances of `splitter` and returns a list of the sub-lists. You should *not* perform a split on any of the sub-lists. This is less complicated than it may sound; see below for example output. Hint: use a recursive helper function with at least one accumulator. You may use the built-in function `append` in your solution.
2. Write a function `infix->prefix` of the form `(infix->prefix expression)`, where `expression` is an expression in infix notation as described above. Your function should return a SCHEME expression (most likely a *list*) which produces the same answer as evaluating `expression` using normal order of operations. You need only handle the operators `+`, `-`, `*`, and `/`. You may use (and are encouraged to use) the built-in `map` function, plus any functions from part 1.

Example output

```
(split-by 'X '(one two X three four X five X six (seven X eight)))
     =>  ((one two) (three four) (five) (six (seven X eight)))
(infix->prefix '(1 + 1))                    => (+ 1 1)
(infix->prefix '(1 + 2 * (1 + 1)))          => (+ 1 (* 2 (+ 1 1)))
(infix->prefix '(1 / 2 * 1 / 3))            => (* (/ 1 2) (/ 1 3))
(infix->prefix 4)                           => 4
(infix->prefix '((((1))) + 2))              => (+ 1 2)
(infix->prefix '(1 + 2 + 3 - 4 - 5 + 6 + 7) => (+ 1 2 (- 3 4 5) 6 7)
```

Notes

- Be careful with spaces. `'(1+2)` is a list containing the symbol "1+2", whereas `'(1 + 2)` is what you probably wanted. Having a lot of spaces won't hurt you, but having too few will make SCHEME parse your input in ways you probably didn't intend. In general, you always need spaces to separate elements of a list.
- As shown above, SCHEME math functions can take any number of arguments, so you don't have to worry about this in your implementation.