CS 335 - Homework 2

Deadline:

Due by 11:59 pm on Friday, February 11

How to submit:

Submit your files for this homework using `~st10/335submit` on nrs-labs, with a homework number of `2`

Purpose:

To practice with leftmost derivations, rightmost derivations, and proving that a CFG is ambiguous; to start getting familiar with Scheme R5RS dialect as implemented in DrRacket, including practice with writing arithmetic expressions, manipulating lists, and writing recursive functions.

Important notes:

- Download an appropriate version of DrRacket (http://racket-lang.org/) and install it, or work in a campus lab that already has it installed (which *should* include BSS 313, BSS 317, LIB 121, LIB 122).
- Start it up, and remember to set the language appropriately:
  - under the Language menu, select "Choose Language...", select **R5RS** from the Legacy Languages section, and click OK.
  - NOTE: the language change does NOT take effect until you then click the Run button in the top right corner above the definitions window (next-to-last button, directly to the left of the Stop button).
  - In a campus lab, you'll probably have to set the language every time you start up DrRacket - on a home version, it will probably start up with whatever language version you were using the last time you existed DrRacket.
- Note that, under the File menu, you can Save your Definitions window (the upper window) contents to a file; under "Save Other", you can also save the contents of the Interactions window (the lower window).
- Before each Scheme function, include comment lines that include at least the name of the function, and a purpose statement (clearly describing what it expects and what it returns).
- Put a blank line before each function definition and after each function definition -- use additional blank lines are needed for readability.
- Start each function's body on a different line than the function header, indented for readability.
- Use hard returns rather than letting lines of code wrap to the next line.

The Problems:

Problem 1

For this problem, type your answers into a plain text file, `335hw2-1.txt`, or saved as a PDF, `335hw2-1.pdf`. Include your name as the first thing in this file, and clearly label each answer with the letter of the question part it is for.

Recall the CFG G used on Homework 1, Problem 3:

A -> DAD | B

B -> mCh | hCm

C -> DCD | D | ε

D -> m | h | p

1 part a

Recall that a leftmost derivation is one in which, at each step, a production is applied to the leftmost nonterminal [Hopcroft and Ullman, "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley, 1979, p. 87]

Give a leftmost derivation for `hhmmm`.

1 part b

Give a rightmost derivation for `hhmmm`.

1 part c

Also recall that a CFG G is said to be ambiguous if the language generated by that CFG (sometimes called L(G) for short) includes even one word *w* for which:

- *w* has more than one parse tree, OR
- *w* has more than one leftmost derivation, OR
- *w* has more than one rightmost derivation

Is the CFG G given in this problem ambiguous? If so, prove it; if not, explain why you think it is not ambiguous. (If you choose to do this via parse trees, you can put "see pdf" here and scan your parse tree drawing, save it as a PDF file `prob1-c.pdf`, and submit that along with your file containing the rest of your answers.)

Problem 2

Start a Scheme file `335hw2.scm` or `33hw2.ss`. The first thing in this file should be comments containing at least:

- your name
- `CS 335 - Homework 2`
- and the last-modified date

Now follow that with a comment containing `Problem 2`, followed by your answers for this problem.

(adapted from MacLennan Exercise 2-39, p. 89)

As we mentioned in class, Scheme does not need a concept of operator precedence - since Lisp syntax is essentially an abstract syntax tree, what operators are applied to what is not ambiguous. (That's why, in the translation of a high-level language, expressions may be parsed into the non-ambiguous abstract syntax tree form as part of the translation process!)

Rewrite each of the following expressions as equivalent Scheme expressions - that is, include an equivalent Scheme expression for each of these within your `.scm`/`.ss` file. (Your Scheme expressions must perform all of the same operations as these expressions ... the idea here is to practice writing particular expressions using this particular syntax.) Assume that ** is exponentiation.

2 part a

```
((1 - (2 * 3)) + (4 / 5))
```

2 part b

```
(((1 - 2) - 3) - 4)
```

2 part c

```
((1 / 2) / 3)
```

2 part d

```
((1 / 2) * 3)
```

2 part e

Note: Scheme does have a built-in function, `expt`, for raising a number to a power...

```
(2 * (3 ** 4))
```

2 part f

```
(3 + (1 / 4)) - 5
```

Problem 3

Put a comment containing `Problem 3`, followed by your answers for this problem.

For some list-expressions practice...

Consider the list `(1 2 3 4 5)`

(and remember: because of Scheme's eager evaluation, you need to precede this list with a ' when it is used in expressions, so Scheme will not try to treat 1 as a function to be applied to arguments 2, 3, 4, and 5, which the poor number 1 is not).

```
    '(1 2 3 4 5)          ; is actually the same, under the hood, as...
    (list 1 2 3 4 5)      ; and
    (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 '())))))
```

(I strongly encourage you to try out the above expressions in DrRacket, and verify for yourself that all evaluate to the same list).

As a convenience, Scheme also provides an `append` function that expects two lists and produces the result of putting all of the contents of those two lists into a list -- that is,

```
    (append '(1 2 3) '(4 5 6 7))
```

...results in the list `(1 2 3 4 5 6 7)`. (Compare this to the result of using `cons` with the same two lists as its arguments -- the result is different...)

3 part a

Write three different Scheme expressions whose value is a list consisting of 3 strings of your choice. (Remember: a string literal in Scheme is similar to what you are probably used to from C++ or Java, surrounded by double-quotes.)

Now, use any of the versions of this list that you wish in your answers to parts b through e below, and:

3 part b

... write a Scheme expression using that list whose value will be the first string in that list.

3 part c

... write a Scheme expression whose value will be the list of the second and third strings in that list.

3 part d

... write a Scheme expression whose value will be the second string in that list.

3 part e

... write a Scheme expression whose value will be the third string in that list.

It turns out that, in Scheme, you can give a name to a value with the `define` function:

```
(define MAX-FRUIT-PIECES 5)
MAX-FRUIT-PIECES    ; now is a simple expression whose value is 5
```

(If we are attempting to be somewhat pure in our functional programming, we will not change the value of `MAX-FRUIT-PIECES` to something else later on in our Scheme program... 8-))

Now consider the list whose name is `cs-phone` :

```
(define cs-phone '((tuttle 3381) (burgess 3331)
                   (burroughs 3337) (dixon 3530)
                   (amoussou 3380)))
```

Type or paste the above define expression for `cs-phone` into your definitions window, and:

3 part f

... write a Scheme expression, using `cs-phone`, whose value will be the list `(tuttle 3381)`

3 part g

... write a Scheme expression, using `cs-phone`, whose value will be `3331`

3 part h

... write a Scheme expression, using cs-phone, whose value will be

```
((rizzardi 4951) (tuttle 3381) (burgess 3331)
 (burroughs 3337) (dixon 3530)(amoussou 3380))
```

3 part i

... write a Scheme expression, using `cs-phone`, whose value will be

```
((tuttle 3381) (burgess 3331)(burroughs 3337)
 (dixon 3530)(amoussou 3380)(tuttle 3381)(burgess 3331)
 (burroughs 3337) (dixon 3530)(amoussou 3380))
```

Problem 4

Put a comment containing `Problem 4`, followed by your answers for this problem.

As a Scheme function warm-up, here is a classic recursive function, to get your feet wet: write a function `len` that expects a list and returns the number of top-level elements in that list. (By top-level, one means that if the list contains lists, each such element list counts as 1 element in the overall list -- that is,

```
     (len '(1 (2 3) (4 (5 6) 7)))
```

...should be 3.

`len` must determine this in a recursive fashion (you may not call the built-in length function instead... 8-))

For example:

`(len '(1 2 3))` should return 3

`(len '((1 2) 3 (4 5 6) 7))` should return 4

`(len '())` should return 0

To provide some quick'n'sleazy testing, after your function insert the following (feel free to insert additional testing calls if you'd like):

```
(display "testing len - #t's mean passed:\n")
(equal? (len '(1 2 3))
        3)
(equal? (len '((1 2) "hi there" ('a 5 (17) 'b) #t))
        4)
(equal? (len '())
        0)
```

As you can see, this is going to show the value of the string to the screen, followed by `#t` or `#f` for each call to `equal?` comparing a call to your `len` function to the expected return value for that call. When you click the Run button, if your `len` is working, you should see the following in your Interactions window:

```
testing len - #t's mean passed:
#t
#t
#t
```

Problem 5

Put a comment containing `Problem 6`, followed by your answers for this problem.

Now that you have function `len`, you should be able to use it to good effect in a (non-recursive) function `get-4th` that expects a list, and if there are at least 4 elements in the list it returns the 4th element -- otherwise it returns the empty set.

After your function insert the following (feel free to insert additional testing calls if you'd like):

```
(display "testing get-4th - #t's mean passed:\n")

(equal? (get-4th '())

        '())

(equal? (get-4th '(hi there))

        '())

(equal? (get-4th '(1 2 3 4))

        4)

(equal? (get-4th '(do re mi fa sol la ti do))

        'fa)
```

Problem 6

Put a comment containing `Problem 6`, followed by your answers for this problem.

Write a recursive function `build-it` that expects an integer (a list size) and a value, and that returns a list containing that many of that value. That is,

```
> (build-it 4 #t)

(#t #t #t #t)

> (build-it 2 '(hi there))

((hi there)(hi there))

> (build-it 0 42)

()
```

After your function insert the following (feel free to insert additional testing calls if you'd like):

```
(display "testing build-it - #t's mean passed:\n")
(equal? (build-it 4 #t)
        '(#t #t #t #t))
(equal? (build-it 2 '(hi there))
        '((hi there) (hi there)))
(equal? (build-it 0 42)
        '())
```