Project 1: Scheme & Natural Language Parsing

Educational Objectives: After completion of this assignment, the student should be able to

- Use the Scheme cond-else and if special forms
- Use Scheme car, cdr, cons to inspect, manipulate, and construct lists
- Use lambda abstraction to define functions in Scheme
- Write recursive functions in Scheme
- Implement recursive descent parsers in Scheme for small LL(1) grammars

Operational Objectives: Create the file `proj1.scm` containing various scheme functions including a small declarative natural language sentence parser.

Deliverables: One file `proj1.scm`.

Procedural Requirements:

1. Create your scheme functions in the file `proj1.scm` as required below.

2. Copy the submit script `LIB/proj1/proj1submit.sh` into your project directory. Change permissions to executable ["`chmod 700 *.sh`"].

3. Turn in one file `proj1.scm` using the `proj1submit.sh` submit script.

   Warning: *Submit scripts do not work on the `program` and `linprog` servers. Use `shell.cs.fsu.edu` to submit projects. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.*

Technical Requirements and Specifications

1. Write scheme functions as follows:

```
BOR  ; "Binary OR" takes two arguments x,y, returns x OR y; non-recursive
     ; examples:
     ; (BOR #t #f) Value: #t
     ; (BOR #f #f) Value: #f

BAND ; "Binary AND" takes two arguments x,y, returns x AND y; non-recursive
     ; examples:
     ; (BAND #t #f) Value: #f
     ; (BAND #t #t) Value: #t

pos? ; "positive?" takes one argument, returns true if positive number, false otherwise
     ; examples:
     ; (pos? 1)   Value: #t
     ; (pos? -2)  Value: #f
     ; (pos? 0)   Value: #f

in?  ; takes two arguments item, list; returns true if item is in list, false otherwise
     ; examples:
     ; (in? 'x '(b c x d))    Value: #t
     ; (in? 'x '(b c d e f)) Value: #f

reduce ; takes two arguments binary op, list; returns result of applying op to entire list
     ; examples:
     ; (reduce + '(1 2 3 4 5))          Value: 15
     ; (reduce + '(2))                  Value: 2
     ; (reduce BOR '(#t #t #f #t #f))   Value: #t
     ; (reduce BAND '(#t #t #f #t #f))  Value: #f
     ; (reduce BAND '(#t))              Value: #t
     ; (reduce BAND '(#f))              Value: #f
     ; (reduce BOR '(#f))               Value: #f
     ; (reduce BOR '(#t))               Value: #t

filter ; takes two arguments pred, list; returns list of items passing pred test
     ; examples:
     ; (filter pos? '(1 3 -3 5 -6 7))  Value: (1 3 5 7)
```

```
; (filter pos? '(-1 -2 -3))       Value: ()
; (filter pos? '(1 2 3 4))        Value: (1 2 3 4)
```

Be sure to thoroughly test each of these as you produce them.

2. Write five Scheme functions that take a word (represented by a Scheme atom) as an argument and return either `#t` (= true) or `#f` (= false) depending on the grammatical classification of the word in five categories: determiner, noun, verb, adjective, and preposition. The five functions should be named `det?`, `noun?`, `verb?`, `adj?`, and `prep?` respectively. You may assume that the vocabulary is limited to the following words:

```
determiners:  a an the
nouns:        apple bicycle car cow dog fox motorcycle path pie road truck
adjectives:   black brown fast hairy hot quick red slow
verbs:        commutes destroys drives eats jumps makes occupies rides stops travels walks
prepositions: around at of on over to under
```

   Run Scheme, load your functions, and enter these:

```
(reduce BOR (map det? '(hot red car)))
(reduce BOR (map det? '(the red car is a hot dog)))
```

   What are the results? Explain in some detail how the results are calculated.

   Write a function named `OK` that returns `#t` when no more than 25% of the words in a sentence are adjectives and `#f` otherwise. For example:

```
1 ]=> (load "proj1")
2 ]=> (OK '(a hairy red dog occupies the hot red car))
;Value: #f
3 ]=> (OK '(a red car rides the road))
;Value: #t
```

3. Write new Scheme functions `det`, `noun`, `verb`, `adj`, and `prep`:

```
det  ; one argument - a list
     ; returns the cdr of the list if the first word in the list is a determiner
     ; otherwise, it returns an empty list '()

noun ; one argument - a list
     ; returns the cdr of the list if the first word in the list is a noun
     ; otherwise, it returns an empty list '()

verb ; one argument - a list
     ; returns the cdr of the list if the first word in the list is a verb
     ; otherwise, it returns an empty list '()

adj  ; one argument - a list
     ; returns the cdr of the list if the first word in the list is an adjective
     ; otherwise, it returns an empty list '()
     ;
prep ; one argument - a list
     ; returns the cdr of the list if the first word in the list is a preposition
     ; otherwise, it returns an empty list '()
     ;
     ; examples:
     ; (adj '(hairy red dog))        ; Value: (red dog)
     ; (det '(red car))              ; Value: ()
     ; (adj (adj '(hairy red dog)))  ; Value: (dog)
     ; (noun (det '(a dog $)))       ; Value: ($)
```

   (Note that '`$`' has no special meaning in Scheme. It is used above to have a non-empty `cdr` to return.)

4. Using the technology developed above, write a natural language parser in Scheme that implements the following simple grammar for declarative sentences:

```
<sentence>     ->  <nounphrase1> <verbphrase>
<nounphrase1>  ->  [<det>] <nounphrase2>
<nounphrase2>  ->  <adj> <nounphrase2>
<nounphrase2>  ->  <noun>
```

```
<verbphrase>   ->  <verb> [<preposition>] [<nounphrase1>]
```

This entails writing new Scheme functions sentence, nounphrase1, nounphrase2, and verbphrase. These functions should return the tail of a list if parsing is successful and #f otherwise. The following are examples of processing using these functions:

```
(nounphrase1 '(a dog PASSED TEST 1)) ; (passed test 1)
(nounphrase1 '(red dog PASSED TEST 2)) ; (passed test 2)
(nounphrase1 '(dog PASSED TEST 3)) ; (passed test 3)
(nounphrase1 '(a red dog PASSED TEST 4)) ; (passed test 4)
(sentence '(the red dog rides a hot car PASSED TEST 5)) ; (passed test 5)
(sentence '(a rides car FAILED TEST 6)) ; #f
(sentence '(the hairy hot red dog eats a red hot hot hot apple PASSED TEST 7)) ; (passed test 7)
(sentence '(the hairy hot red dog walks)) ; ()
(sentence '(an apple FAILED TEST 9)) ; #f
(sentence '(a red dog FAILED TEST 10)) ; #f
(sentence '(dog eats)) ; ()
(sentence '(the quick red fox jumps over the brown cow PASSED TEST 12)) ; (passed test 12)
(sentence '(fox jumps over the brown cow PASSED TEST 13))  ; (passed test 13)
(sentence '(fox jumps cow PASSED TEST 14)) ; (passed test 14)
```

Alternative: Instead of returning #f when parsing fails, you may choose to (consistently) return the tail of the list with #f added at the head, thus providing information where parsing failed. For example:

```
(sentence '(a red dog FAILED TEST 10)) ; (#f failed test 10)
```

Hints

- A confusing point is that Scheme makes little distinction between `()` [the empty list] and `#f` [the false value].
- Here is a small grammar and parser for it:

```
;
; <bits> -> 0 <bits>
; <bits> -> 1 <bits>
; <bits> -> empty
;

(define bits
  (lambda (arg)
    (cond
      ((equal? arg ())                     #t)
      ((equal? '0 (car arg)) (bits (cdr arg)))
      ((equal? '1 (car arg)) (bits (cdr arg)))
      (else                      (cons #f arg))
    )
  )
)
```

It is remarkable how simple this is to code and actually make work. Test it with these inputs:

```
(bits '(0 0 1 1 1))
(bits '(0 1 0 1 x 0))
```