Homework 3: Arithmetic Interpreter

Due Thursday, Feb. 26 at 9:50 am

Tips:

- *You should spend a "medium" amount time thinking and a short amount time implementing the programming parts of this assignment. If you're spending a long time (e.g., more than 1 hour for part 1) thinking or implementing, then come see me and we'll try a different way of looking at the problems.*
- *See Homework 2 for instructions on how to run Dr. Scheme in the UNIX lab.*
- *Remember to acknowledge all assistance and state how long the assignment took to complete.*

1. Stream Programming

(20 points)

Implement procedures using map, foldl, and filter to:

1. Return the largest value in a list of positive integers.

2. Implement `(contains? L target)`, which returns true if any element of L is eq? to target.

3. Compute the length of a list (you may not call `length` or write your own recursive helper!)

4. Compute `(only-below x L)`, which creates a new list containing all elements of list of numbers L that are less than `x`.

5. Consider a list of students and their grades on some assignments, e.g.,

   '((bob 5 3 2) (sally 2 7 3) (ron 6 5 6))

   Implement a procedure `gpa` that given this input, returns a list of each student and their GPA. For the example case, a correct solution produces:

   (list (list 'bob 3 $\frac{1}{3}$) (list 'sally 4) (list 'ron 5 $\frac{2}{3}$))

   Note that for each entry in the input list, the `first` is the name and the `rest` is the list of grades, and recall that the `list` procedure constructs a list of its arguments. I recommend using a LET statement to take apart the list before processing it (see the following example). You may use the built-in `length` procedure here to compute the denominator.

6. Say that the instructor for a course marks excused absences with the symbol `'excused` in a grade list as above. These grades should not be counted in the GPA at all.

   a) Complete the following helper procedure `(remove-excuses L)` that removes excused absences from the grade list using `map` and `filter`.

```
(define (remove-excuses students)
  (????
   (lambda (s)
     (let ([name   (first s)]
           [grades (rest s)])
       (cons name  ????  )))
   students))
```

That is, the above procedure should remove all elements x for which `(not (eq? x 'excused))` is `#t`. Recall that the `cons` procedure constructs a list. It takes as its arguments the first element of the new list and a list of the rest of the elements. In debugging your solution, it is handy to know about the `print` and `display` procedures, which you can look up in the documentation. You may invoke either of those as many times as you want in a lambda, define-proc, or let expression immediately before the body expression. For example:

```
(define (hello x)
    (print (first x))
    (print (rest x))
  (cons 'hello x))
```

An example of remove-excuses operating correctly is:

> (remove-excuses '((bob 5 excused 2) (sally 2 7 3) (ron 6 excused excused)))
(list (list 'bob 5 2) (list 'sally 2 7 3) (list 'ron 6))

b) Write `(gpa-with-excuses L)` that uses `remove-excuses` and `gpa`. An example of this operating correctly is:

> (gpa-with-excuses '((bob 5 excused 2) (sally 2 7 3) (ron 6 excused excused)))
(list (list 'bob 3 1/2) (list 'sally 4) (list 'ron 6))

2. Iffy Parsing

(20 points)

A tokenizer breaks input source code into the individual pieces like operators and parentheses that form the input to a parser. A parser accepts this input and outputs a parse tree, where each node is an expression type and its children are the subexpressions. The parse tree is what the interpreter executes.

In Scheme, the built-in literal tokenizer-parser is called `read`. For our purposes we don't call it directly; instead we just type list literals like '(> (+ 1 2) 5). One way to write an interpreter is to define helper procedures that "take apart" (i.e., "parse") the lists, and use those helpers inside an interpreter that operates directly on list literals. This is a weak abstraction because everything has type list (or int, bool, or symbol). It has the advantage of not making us write a real parser. A stronger abstraction is to distinguish the list literals from the parse tree by writing a parser that converts each expression to its own class. PLAI Scheme includes the define-type and type-case special forms to help with that. In this question you'll implement a simple boolean-logic language the first way.

Consider the following language, which I'll call IFFY. This is Scheme-like, but IFFY is not Scheme. It contains only:

IFFY Expressions:

```
<BOOLEAN>   ::= #t | #f
<IF>        ::= ( if <EXP> <EXP> <EXP> )
<EXP>       ::= <BOOLEAN> | <IF>
```

IFFY Values:

```
<VALUE>     ::= <BOOLEAN>
```

Assume that the evaluation rules for these expressions are the same as they are for the equivalent expressions in Scheme.

7. You're going to write a weakly-typed parser (i.e., that does not use define-type and type-case). Determining if an expression is a BOOLEAN expression is easy. From Scheme, we get `boolean?`, the function that returns true if its argument is a BOOLEAN expression. We can abstract this lightly by:

   ```
   (define BOOLEAN? boolean?)
   ```

   so that our code is clear about when we're working with IFFY values and when we're working with Scheme values (even though we won't get an error if we mix them up).
   To parse the rest of the expressions, implement the following helper functions (don't bother with error cases):

   a. `IF?` Returns true if the argument is an IF expression (be careful about empty lists!)
   b. `IF-test` Returns the test subexpression from an IF expression
   c. `IF-consequent` Returns the test subexpression from an IF expression
   d. `IF-alternate` Returns the test subexpression from an IF expression

8. Implement `(eval exp)`, which reduces its <exp> expression argument to a <value> using the above helper methods. In the process, create and use a helper procedure, `(eval-IF t c a)`. This helper takes as arguments the test, consequent, and alternate sub-expressions of an IF expression. Don't bother with errors for malformed expressions. Here are some test cases:

   ```
   (eval '(if #t #t #f))                    →    true
   (eval '(if #f #t #f))                    →    false
   (eval '(if (if #t #t #f) #f #t))         →    false
   (eval '(if (if #t #f #t) (if #f #f #t) (if #f #t #t)))     →   true
   ```

3. Short Answer

(10 points)

9. Prove that COND is at least as powerful as IF in Scheme (i.e., that we could remove IF without decreasing the expressive power of the language)

10. a. Why is AND a special form in Scheme (and in most other languages)?

    b. Why is logical exclusive-or ("XOR", the ^ operator in Java and C++) not a special form in Scheme or Java?

11. What are the differences between a compiled and an interpreted language? Why is Java somewhere in between?

12. a) What are the advantages of distinguishing with different types (in the implementation language) the unparsed input, the expression domain, and the value domain when writing an interpreter?

    b) ...what are the disadvantages?

    c) Why does it, or does it not, make a difference whether we distinguish those three domains in the formal specification (e.g., BNF) of a language?

    d) Why is it a good convention to capitalize IF in "IF expression" for questions 7 and 8, for both English discussion and actual procedure names?

4. Challenges

(glory and experience, but no points awarded here)

13. For each of the following languages, is it usually compiled or interpreted?: C++, Scheme, ML, Python, Perl, Pascal, Matlab, sh, HTML, Postscript/PDF, ZIP (the compression file format). The last three are not what we usually think of as general-purpose programming languages. But they have grammars, a stream of instructions, and execute to compute values...so they *are* programming languages. You will need external resources (likely, on the web) to learn more about each of these languages—beware that not all of the resources you find will be correct, so think through what you read based on your understanding of compilers and interpreters.

14. Can the halting problem be solved for programs written in the IFFY language? Prove your result and explain how the incompleteness theorem applies to this case.

15. Give an example where foldl and foldr give different results on the same arguments.

16. Revisiting question 7, implement a strongly-typed parser `(parse s)`, using define-type. See pages 6 and 8 of the PLAI textbook for syntax guidance. I recommend saving this parser in a separate file from the one you wrote above. Here are some test cases:

```
> (parse '(if #t #f #t))
(IF (BOOLEAN true) (BOOLEAN false) (BOOLEAN true))

> (parse '(if (if #t #f #f) (if #t #f #t) #f))
(IF
 (IF (BOOLEAN true) (BOOLEAN false) (BOOLEAN false))
 (IF (BOOLEAN true) (BOOLEAN false) (BOOLEAN true))
 (BOOLEAN false))
```

17. Rewrite your solution to question 8 to be a strongly-typed evaluator that accepts parsed expressions. Use `type-case`. As before, make an eval-IF helper. You can use the same test cases as in part 2, except now you need to parse the input explicitly, e.g.,

```
(eval (parse '(if #t #t #f)))
```

18. Consider a linear-algebra package implemented in Scheme, where matrices are lists of rows (which are themselves lists of numbers), and column-vectors are lists of numbers, for example:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 3 \\ 2 & 1 & 1 \end{bmatrix}, \vec{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
(define A '((1 0 3)
            (2 1 1)))

(define x '(1 2 3))
```

Implement the following using map and foldl (i.e., no loop expressions or recursion):

a) `(dot x y)`, which computes the dot-product of two vectors implemented as lists, where the dot-product is given by: $x \bullet y = \sum x_i y_i$

b) `(mul M v)`, which computes the matrix-vector product where element *i* of the product is the dot product of row *i* of M with *v*. Use your dot-product helper function. As a test case, `(mul A x)` should return `(list 10 7)`.