This problem set is due at 11:55pm on Thursday, September 17, 2020. Don’t go by the due date that you see on Blackboard because it is in ET. Go by the one given in this handout. 

Be sure to include a comment at the top of each file submitted that gives your name and email address. 

Submit on Blackboard your solution file named ps3.scm that contains all of your solutions for PS 3. Multiple submissions are allowed before the due date. 

Note: Please read the "Cooperation vs. Cheating" (i.e., "Academic Integrity") section of the syllabus before you start this problem set and future problem sets.

General Instructions

You are free to define auxiliary functions, as many as you want, and you may assume that the functions defined earlier "carry over" into any of the functions downstream in the entire problem set.

As usual all of your programs that you submit should run without any errors. If there is any error and the program crashes, you will get no credit.

In this problem set you are allowed to use set!, but not set-car!, set-cdr!, or begin unless I specifically mention that you are allowed to use them.

Problem 1

Suppose you are writing a program that will help our Admissions Office process the applications into the University. In this problem your program will open an input file containing applicant information and read it into a Scheme data structure. Once you have them read into a data structure, you can do some interesting operations with the data that you just read in.

See a sample input file named a0.txt in the [given] folder for PS 3. I added several input files of various sizes for your consumption if you need them. An entry (an application record) in the input file has the following attributes and is formatted as follows:

- The first line of an application record is the name of an applicant consisting of first name, middle name (or middle initial with a period), and last name in that order in a single line, all separated by a blank space.

- Next line is the street address in one line, city name in the next line, state in another line, and finally zip code in a separate line.

- The next line is a phone number consisting of the area code, prefix, and last four digits all separated by a blank space.

- The next line is the id number of the application.
• The next line is the intended major, e.g., Computer Science, Economics, etc. A multi-word name should be allowed.
• The next line is the applicant’s high school GPA, e.g., 3.96.
• The next line is an indication of whether the applicant is applying for a scholarship or not.
• If there is another application record in the file, it will follow next in the same format.

(a) Write a function named read-applications that takes a file name as its parameter and reads the data into a global variable named *student-records*. We can use that variable for further processing below.

(b) Now that we have read in all the application records into memory, find scholarship recipients. An applicant is considered a scholarship recipient if:

1. his/her high school GPA is higher than 3.8 and
2. his/her intended major is one of the following: Philosophy, History, or Sociology and
3. s/he is from Montana or Idaho, and
4. s/he is applying for scholarship.

Write a function named find-scholarship-recipients that takes one parameter (student records) and returns a list containing the full names of all the scholarship recipients. The result will be a list of lists.

(c) Write a function named average-gpa that takes two parameters, namely a state name and the student records, and returns the average GPA of all the applicants from a state. Call it with California as the state.

(d) Write a function named highest-gpa that takes a state name and student records as parameters, finds the applicant whose GPA is the highest among the applicants from the state, and returns the student’s name and his/her GPA in a list. Call it with Washington.

(e) Write a function named histogram that takes the student records and build a histogram of state names and the number of students from each state as an association list, e.g., the result will be of the form: ((Arizona 5) (California 20) ...). The list should not contain the states whose count is 0. Also the list should be in state name order, alphabetically.

Problem 2
In this problem we will write an interpreter for a small subset of an imperative programming language of the Algol family, such as Pascal or C. One example program in that language would look like this:

;;; This program computes y/x and quotient and remainder.

y = 10;
x = 3;
r = y;
q = 0;
while r > x do
    r = r - x;
    q = q + 1;
od

Although this example program does not use a conditional statement, the subset would also include conditionals. The language constructs that this subset supports include the following (***):

• constants
• variables
• addition operator
• subtraction operator
• equality testing operator
• skip as a null statement
• assignment statement
• sequence statement
• conditional statement (if...then...else)
• while loop statement (while exp do body)
• etc.

The language will not have types. That is, there won’t be any explicit type information as part of a program in this language.

To help you understand the overall structure of the interpreter, I am providing a subset of the interpreter implemented in the [given] folder (see given.scm). That simple subset is good enough to run the following program of three lines:

\[
x = 10 \\
y = 2 \\
z = x + y
\]

As you will see in given.scm, two of the functions there (assoc and update) have some code missing. You worked on similar functions in PS 2 so that will be helpful.

So, complete given.scm first and make it work. Understanding how given.scm works thoroughly will be necessary to complete the rest of the program.

How it works:

We are implementing an interpreter for an imperative programming language. The interpreter will be written in a functional language, Scheme, in our case. As I mentioned in class, an imperative language computes by effect, whereas a functional language computes by value. However, an imperative language also has the functional elements (i.e., expressions) in addition to statements which causes effects. In the example above \( x + y \) is an expression which produces a value. In the same example above \( z = x + y \) is an assignment statement which causes an effect, namely, \( z \) gets a value. The fact that \( z \) gets the value 12 needs to be stored somewhere. In our implementation it will be stored in the environment (*env*) in the form of an association list like this: \((z 12)\). Well, according to the example program above there should be two more pairs like this: \((x 10) (y 2) (z 12)\), not necessarily in that order.

So, let’s emphasize this in a different way. When you try to compute a value you will be reading values from the environment (the association list) to compute it. In given.scm the function val is the function that computes a value given an expression using the environment as the program memory. The environment is working as a stack frame. Note that we only have one stack frame in the stack since we only have one function in our simple program (we can view that one function as the main function in the program). The values are stored in the stack frame and val and effect access the stack frame.

When you try to execute a statement, such as an assignment statement, that produces an effect, you will be using the function called effect in given.scm.

So, given any language construct that you want to interpret (run or execute), you need to distinguish between value computation and effect causing. For the former, call val and for the latter call effect.

What to do - the plan:

Let’s use one of the three lines (repeated below) in the program given above and see what we would have to do in what order:
This is similar to the following in C or Java:

```c
int x = 10;
```

That is, this allocates a piece of memory (on the stack in our case) and puts 10 into the location. The difference is that we don’t use types in our program.

We will first represent this in a list form so that it will be easier to handle in Scheme. Furthermore we also have to tag each element so that we will be able to tell what we are dealing with later. So, we will represent Line A as follows:

```scheme
(ASSIGN x (QUOTE 10))
```

In this representation ASSIGN tells us that it is an assignment statement. The left hand side of the assignment is the variable x. The right hand side of the assignment is the expression (QUOTE 10) which means that a constant value like 10 is to be represented with the tag QUOTE as a constant. This sort of representation is called *abstract syntax* which is a useful technique when we are dealing with a language element in processing.

Let’s focus on (QUOTE 10). We know that given this representation, what we really need at some point is the number 10. How do we extract 10 out of (QUOTE 10)? Well, (cadr `(QUOTE 10)) will do, right? In given.scm we wrote a function named number that does just that.

When we try to evaluate (QUOTE 10), who should we ask to do it for us? The val function, right? Well, so we would call val like this: (val `(QUOTE 10) *env*). The function val can be called with any variety of expressions, (QUOTE 10) being one of the possibilities. Well, inside val we can tell which kind of expression it is dealing with. In the (QUOTE 10) case, the function const? is the one that identifies (QUOTE ...) kind of element. So, I gave this long explanation to show the relationships among three things that are all working together to deal with one element (QUOTE 10). Those three things are: (QUOTE 10), const?, and number.

For each language element that we have to process we will have to provide these things.

**What to do for each language element:**

We know what to do with constants represented as (QUOTE ...). I just explained it in the previous section.

We will have to represent a variable, say x, as (VAR x). Then, use var? and name. We call (VAR x) the *abstract syntax representation*; the function var? a *predicate* for (VAR ...); and the function name a *selector*. We just did that for (QUOTE 10) also. There, (QUOTE 10) was the abstract syntax representation for 10 that we want to interpret; const? was the predicate that identifies the element; and number was the selector that extracts the content in the abstract syntax representation. For each language element, we will have to do the same.

For `exp1 + exp2.given.scm` already handles everything. One thing I want you to notice is that this expression can be quite complex beacg each operand can itself be another complex expression.

For `expr1 = expr2 (equality testing expression), it is very similar to expr1 + expr2`. Perhaps, you can use (EQ e1 e2) as its abstract syntax. You can come up with appropriate names for its predicate and selector. You can add one for subtraction, e.g., `expr1 - expr2` too.

For `skip`, you may use (SKIP) as its abstract syntax. You would call `effect` to interpret, and it will do nothing and just return the current environment.

For `x = expr (assignment)` it is already done in given.scm.

For `S1; S2 (sequence), you can use (SEQ S1 S2)` as its abstract syntax. The interpretation (meaning) can be done by calling `effect` since it is an effect causing sequence of statements, right? The actual interpretation would be to call `(effect S1 env)` and use its resulting environment, say env2, to call `(effect S2 env2)` once more in that order. We are not using begin but function composition. What do I mean by function composition? Something like this: `(effect s2 (effect s1 env) env).right?
For `if exp then S1 else S2 fi` (conditional), you can use `(IF exp S1 S2)` as its abstract syntax. You can decide a predicate, e.g., `if?`. You will need three selectors: one that selects `exp`, second one that selects `S1`, and the third that selects `S2`. Should we use `val` or `effect` to interpret this one? Well, are we trying to get a value or are we going to cause effects with this conditional statement? Effect! Right? So, call `effect`. The branch in `effect` that handles conditional would be something like this: `(effect (if (val exp env) S1 S2) env)`. Note that as part of interpreting a conditional it also invokes `val` to evaluate the testing part of the conditional.

For `while exp do S od`, `(WHILE exp S)` can be the abstract syntax representation. Then the predicate may be `while?`. For selectors you will need one to extract the test part and another for body part. Interpreting a while loop would require a `letrec` construct of Scheme, which we have not studied although we did study `let`. I will give you an example of `letrec` in the FAQ or show you in the next recitation session. Anyway, you will need to do something like this to interpret a while statement: create a recursive function as part of a `letrec` construct. The recursive function would look something like the following:

```scheme
(letrec ((w (lambda (env)
          (if (val (wtest s) env)
              (w (effect (wbody s) env))
              env)))
    (w env))
```

Here the function `w` is called recursively within the `letrec`. Because of the scoping rules in `let` a recursive function would not work, thus the use of `letrec`. I will explain the difference between `let` and `letrec` separately.

**Putting it all together:**

Well, `given.scm` shows the overall structure. You just have to understand the structure and add more cases where appropriate to complete it all. The **How it works** section above gives you a good explanation of how you put it all together. If you need help, you know how to reach me.

My suggestion: incremental development!!! For each of the language constructs that you want to implement, add one construct at a time and make it work before you add another one. That way, if you run out of time, you will have a subset that works. It is much better to have a partial solution that works than the almost complete solution that does not run. You know that the program that crashes gets no credit.

**Testing:**

How do you know when you are done? Well, you should include at least those listed above (marked with (***)). The remainder and quotient program given at the beginning of this problem must run. Note that that program does not use any conditionals. So, you should come up with another test program that also uses a conditional in addition to other things listed in (***)