# Programming Assignment 2

We are going to write a ***Scheme*** program to simulate the behavior of a push down automaton utilizing an LL(1) parse table to parse statements from and LL(1) language. The language is the "calculator language" described in the handout you received in class. This language is defined by the 19 productions you see on page number 67 of the handout. We didn't explain in this course how this is done but you should take it on faith that we can write a program that reads the productions for an LL(1) grammar and produces such a table that can be used by the pushdown automaton to parse the language. The table for our calculator language is shown on page number 68 of your handout. A trace of the parse of the program

```
read A
read B
sum := A + B
write sum
write sum / 2
```

is shown in Figure 2.13 of your handout.

*Conventions used in the program*

Unlike the program shown above we will limit the names of identifiers in our version of the calculator language to be only *single capital letters*. For example, we would need to substitute a variable named C in the above program for "sum" as we can't have multi-character ids. This is just to simplify the program. In reality a parser wouldn't ever see strings of characters like "sum" or "read" it would just see a "token" for ID or the READ token not individual letters. Our machine isn't being fed a stream of such tokens from a lexical analyzer, since we didn't write one, so we will just have to deal with recognizing "r e a d" as the "token" for the terminal symbol read in our grammar. To make things as simple as possible you can assume that the input will always have at least one space between the stings in the input to make it easy for you to recognize them.

To simplify your output rather than producing the three columns as shown in figure 2.13 we will produce *three separate files*, one for each of the columns. The names of these files should be: *"parsestack", "inputstream", and "comment"* ; just like the headings for figure 2.13 only without any spaces or capital letters in the names. As a further simplification in the input stream column (i.e. file) you should only show the next input symbol not a sequence of symbols as appears in the figure. So if *read* is the next thing in the input show *read* and not *read A read B...* like they have in the figure. Your comment column (file) should be like the figure with the word *match* or *predict*. Match says what was matched like "match id." Predict should depart from the way it is shown in the figure and just say *predict 7* where 7 is the production number predicted. In other words don't write out the right hand side of the production. Conform to the use of the special symbol $$ to mean the bottom of the stack.

Please remember that your programs will be tested by a script so you must conform exactly the all the file naming and output formatting conventions. Also, please note that this program is to be written in R5RS Scheme NOT Racket. If you are using the Windows DrRacket program be sure while you are experimenting that you select the proper language. Click Languages and select Other Languages->R5Rs. The script that tests your programs will invoke the plt-r5rs interpreter on Linux. If you want to check it using this just log into any of the lab machines in Linux and run it against you source file from the command line, i.e. plt-r5rs filename. Don't put a "#lang r5rs" line at the beginning of your program, plt-r5rs doesn't like that and besides it only does r5rs so there is not need for it.

Use the file name input for the file that contains the program. I will place a $$ at the end of the input put to make it easy to match the accepting condition.  All of our literals will be simple integers, no floating point numbers.  Note that I will place at least one space character between every token to make it easy for you to recognize them in the input.

```
read A
read B
C := A + B
write C
write C / 2
```