CISC106 Summer 2011 Project

Your job is to implement a graphical calculator application. Your application will evaluate arbitrary arithmetic expressions containing numbers, parentheses and the following operators: $+$ (plus), $-$ (minus), $*$ (times), $/$ (divided by) and $\hat{}$ (exponentiation). An exmample of such an expresson is $5 * (10 + 10)\hat{}2 - 100$. The calculator should follow the standard rules for order of operation. Your calculator will also include a "recall" feature, wherein it will display the last n successful calculations (for some $n > 3$, the value of which I'm leaving up to you. Make a reasonable choice.)

The graphical interface will be provided for you - your job is divided into three parts, as detailed below.

Evaluation Algorithm

This is the main part of the project, where you will implement the rules which cause an expression to be evaluated. The method you will use is called a *recursive descent* algorithm. A high-level overview of how it works is as follows:

- An *expression* consists of a *term* followed by an *addition continuation*
- An *addition continuation* consists of one of the following:
  - Either a $+$ or a $-$ followed by a *term* followed by another *addition continuation*
  - Nothing
- A *term* consists of a *factor* followed by a *multiplication continuation*
- A *multiplication continuation* consists of one of the following:
  - Either a $*$ or a $/$ followed by a *factor* followed by another *multiplication continuation*
  - Nothing
- A *factor* consists of a *unit* followed by an *exponential continuation*
- An *exponential continuation* consists of one of the following:
  - A $\hat{}$ followed by a *unit* followed by another *exponential continuation*
  - Nothing
- A *unit* consists of one of the following:
  - a number (either floating point or integer)
  - a $-$ followed by a number (again, either floating point or integer)
  - a $($ followed by an *expression* followed by a $)$
  - a $-$ followed by a $($ followed by an *expression* followed by a $)$

Each of the above elements (*expression*, *term*, *factor*, etc.) should be implemented in its own function which takes a *line* (described in the next section, the details of which are irrelevant to this section.) The continuation functions should also take an argument

for the left hand side of the operation (*i.e.* the value calculated prior to the call to the continuation.)

At this point, the name *recursive descent* should be starting to make sense. In the end, you will have a system of *mutually recursive* (meaning they call each other recurisvely) functions which all work together to compute the result of an arithmetic sentence.

My suggestion for implementing this would be start at the bottom (*unit*) and work your way up. In this way, you'll be able to write unit tests for each peice to be sure it works before using it to implement the next piece. [1]

String Processing

Since the input is in the form of a string, we need to turn it into symbols which can used in the evaluation. When you initially implement the first part of the project, you'll use the string processing functions provided by me in the *tokenizer* module. For this part of the project, you'll replace those functions with your own implementations. You'll be implementing two functions:

- a `next_symbol` function which takes a *line* (a list containing a string as its first element and a number representing an index into that string as its second element) $line$ and returns the string representation of the next *symbol* (*e.g.* $+$, $-$, $*$, $($, 99, 66.9) on $line$. Upon return, $line[1]$ (the index) should be set to the index of the first character in $line[0]$ (the string) past the returned symbol. If the end of $line$ has been reached (meaning there are no symbols left on $line$ when `next_symbol` is called) the string `'End'` should be returned and $line[1]$ set to $len(line[0])$. If an invalid symbol is read, then the string `'Error'` should be returned (and the value of $line[1]$ is *undefined* - I'd suggest just leaving it as whatever value it is when you realize you have an error.)
- a `look_ahead` function which takes a line $line$ and an integer k and returns the kth symbol on $line$ *without modifying* the value of $line[1]$. If there are less than k symbols available on $line$, it should return `'End'`. If $k < 1$ it should return `'Error'`.

I've already provided unit tests for these functions. You should just test your implementations with these tests to make sure they pass.

Recall Feature

The recall feature of the calculator requires the ability to play with lists, so it should be the last peice of the project you implement. Here you will be implementing the following functions:

- A `store_answer` function which takes a list of strings $answers$, a string $sentence$ which is an arithmetic expression and a number $result$ which is the result of $sentence$. The return value of `store_answer` will be a list satisfying one of the following:

[1] You're also be advised to read over the String Processing section before attempting to work on the Evaluation section.

  - If *result* is `'Err'`, then the return value should be *answers* unchanged.
  - If this *sentence*, *result* pair is already in *answers*, then the returned list should be a copy of *answers* with this *sentence*, *result* pair moved to the end of the list.
  - Otherwise, the returned list should be a copy of *answers* with a new string added to the end signifying that *result* is the answer to *sentence*. If the length of this new list would be greater than n, (where n is the maximum size of the recall list - a value you can decide upon based on the critera described at the top of this document) then you should remove items from the beginning of the list until its length is n.

- A `recall_string` function which takes a list of strings *answers* and returns a string containing all the elements in *answers*, separated by new lines (`'\n'`).

You should write unit tests for these functions which test *several* cases.

Collaboration

You are allowed (in fact I encourage it) to work in pairs for this project. If you decide to work with a partner, either you or your partner should send me an E-mail (and CC it to the other) letting me know the two of you will be working together.

Code sharing between teams is forbidden. Collaboration between teams is governed by the policy set forth in the course Syllabus.