CSE 341: Homework 7

due 12/1/99

You may work in groups of two for this assignment. A solo attempt will be graded the same as that of a group. As usual, turn in one hard copy and one electronic copy of your solutions per group.

Part I: (Not to be turned in.)

No part I.

Part II: (Not to be turned in.)

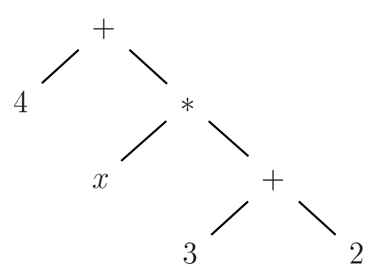
No part II.

Part III: (Due Wednesday, 12/1/99. Turn this in.)

Complete the following. If you encounter an error in the input to any of your functions, you should use the built in `error` function to announce what type of error occurred. **You must turn in output of your code on a demonstrative set of test cases.**

For this assignment you will be implementing a parser for infix arithmetic expressions with variables. `4 + x * (3 + 2)` is an example expression. You will also be implementing a simplifier that would simplify the above expression to `4 + x * 5`.

Your parser will consist of two stages, first the *tokenizing* or *lexical analysis*, and then the actual *parsing* of tokens. The result of running your parser on `4 + x + (3 + 2)` should be the abstract syntax tree:



1. (5 points) Implement the lexical analyzer. Write the function

   `tokens :: [char]->[token]`

   The function `tokens` takes a string as input and returns a list of tokens as output. You will need do define the type of `token`. The idea is that you want to turn the character string into a list of data that is easer to manipulate. This is the process of lexical analysis. For example, if we defined `token` as

   `token ::= RightParen | LeftParen | NumToken num | ...`

   then we would expect output:

```
Miranda tokens "(2 3)"
[LeftParen, NumToken 2, NumToken 3, RightParen]
```

You need to complete the definition of the `token` type and implement `tokens`.

2. (15 points) The next step is writing the parser which will take a list of tokens and return an expression. You goal is to write the function

   ```
   parse :: [token]->expr
   ```

   where `expr` is a user defined type that will hold a single expression. You will have to define `expr` so as to be able to represent expressions trees as shown above.

   Hint: There are many ways to write a parser. A *shift-reduce* parser is a stack based parser that at each step will either *reduce* tokens and expressions already on the stack or it will shift the first unread token off the list of yet unparsed tokens onto the top of the stack. The stack here should be able to contain either tokens or expressions. When you run out of tokens to shift and you cannot make any more reductions you are done. You can return your stack as the parsed answer. Note that if there is a parse error (i.e. bad input, for example `"(1)("`, where the extra parentheses is a parse error) then you will be left with un-reduced tokens on the stack in addition to expressions. A correct parse will leave only a single item on the stack.
   It is recommended that you use a shift-reduce parser in your implementation of `parse`. To do so you will want to define a new data type that can be either a `token` or an `expr` define the shift-reduce parser as

   ```
   tokexpr ::= Tok token | Expr expr
   parse_shift_reduce :: [tokexpr]->[token]->[tokexpr]
   ```

   This is a function that takes as arguments the stack and the list of unused tokens (in that order) and returns the final stack. It remains for you to call `parse_shift_reduce` in `parse` and massage the output into the desired form.

   Parsing is not an easy thing. For infix arithmetic, `*` and `/` have higher precedence than `+` and `-`. Operators of equal precedence are left associative. That is `5 + 6 - 7` is the same as `(5 + 6) - 7`. You do not have to handle unary minus. For partial credit (10 points), you may turn in a parser that does not apply proper precedence rules; however, you must state this and explain what precedence rules you are using.

3. (5 points) Implement `simplify` that takes in an expression and returns a simplified expression. Use all *constant folding* rules that you know. In particular, expressions that have no variables should always be simplified to a expression that represents a single number.

   ```
   simplify :: expr->expr
   ```

4. (5 points) Implement the function `display` to take a parsed expression and turn it back into a fully parenthesized string. That is:

   ```
   display :: expr->[char]
   ```

Also implement the function `transform` that will take a string, convert it into an expression, simplify the expression, and return the expression fully parenthesized and as a string value.

```
transform :: [char]->[char]
```

Some examples of `transform` in action are:

```
Miranda transform "1 + 2 * 3 + 4"
"11"
Miranda transform "x + y * z + w"
"((x + (y * z)) + w)"
Miranda transform "1 + 2 + x + y + z"
"(((3 + x) + y) + z)"
```