COL 765	September 25, 2021
Assignment 2	
Instructor: Subodh Sharma	**Due: Oct 8, 2021**

Problem 1:

The goal in this problem is to create a big integer module in Haskell from scratch. Note that big integer arithmetic is widely used in multi-precision libraries in high-performance computing, as well as in cryptography (e.g., RSA). Your specific tasks include the implementation of addition and multiplication of two large integers. While the addition logic is the same as what is taugh in high school, the multiplication logic is going to be different.

The multiplication of two $n-$digit numbers by the usual long multiplication algorithm (column-wise) that is taught in schools takes about $O(n^2)$ single-digit multiplications. According to the Wikipedia, Karatsuba's algorithm (discovered by Anatoly Karatsuba in 1960) reduces the complexity of multiplication of two $n-$digit numbers to at most $n^{lg_2 3}$ single-digit multiplications. Note that you may have to take special care for overflow.

- Note that you are not allowed to use existing libraries for big integers.
- Also, the input and output numbers to these functions are strings of digits. Therefore, define auxillary functions `fromString: string -> [int]` and `toString: [int] -> string` for purposes of input and output, respectively.
- Your solution should be parametric in the base B in which the lists of digits have to be manipulated.
- The Karatsuba multiplication function takes two numbers as input and returns the result of multiplication. The function to compute Karatsuba multiplication should have signature `karatsuba: [int] -> [int] -> [int]`
- If the input string is not a valid number, your program should raise an exception of type `exception string Invalid_Input_exception`.
- Submit a file named `<EntryNumber>-q1.hs`

Problem 2:

Your task is to write a lexer and parser for a 2-sorted algebra $\Omega_{\mathbb{BZ}} = \langle \{\mathbb{B}, \mathbb{Z}\}, \mathcal{F} \rangle$.

Language Specification

- Integer arithmetic operators are: `PLUS, MINUS, TIMES, NEGATE`(unary operator), `EQUALS, LESSTHAN, GREATERTHAN`. Boolean arithmetic operators are: `NOT, AND, OR, XOR, IMPLIES, EQUALS`. The associativity of boolean operators is defined as: `NOT, IMPLIES` are right-to-left associative and the rest are left-to-right associative. Assume the precedence to be the following: `NOT > AND = OR = XOR = EQUALS > IMPLIES`. For integer operators use the usual precedence and associativity rules (refer to C programming language rules). Use the suitable (with same name) token types for the above operators when implementing a lexer specification file.

- Constants `TRUE, FALSE` representing boolean 1 and 0 respectively. Use token type `CONST` for constants.
- Use token types LPAREN and RPAREN for left and right parenthesis respectively, which help resolve the order of evaluation over different operations.
- Each program should end with a token type EOF.
- Any other string containing only lower and upper case English alphabets is a variable. Use token type ID for variable identifiers.
- The expression grammar also has:
  - support for `if` exp `then` exp `else` exp `fi`.
  - support for `let` var = exp `in` exp `end` to create temporary identifier bindings. Here *exp* is either a valid formula or a valid integer arithmetic expression.

Tasks to be performed

- Download *Alex*: Alex is a tool for generating lexical analysers in Haskell – given a description of the tokens to be recognised in the form of regular expressions, it generates a Haskell module containing the code for scanning the program text and classying it into a stream of tokens. It is similar to the tool lex or flex for C/C++. The command is: `cabal install alex`
- Implement a `lexer.x` file, where the token specification is provided. Refer to *Alex* documentation https://www.haskell.org/alex/ on example lexer specifications. HINT: use a "posn" wrapper which provides line and column number information of tokens in the input text.
- Implement a *recursive descent parser* **from scratch** for the sought language (do not use Happy parser generator for Haskell). You will have to import the lexer module in your parser implementation so that you can utilize the `main lexing routines` (look at the generated lexer.hs for main lexing functions) while parsing.

Output Format and Submission Instructions

- We will compile your submission by running `make` command and the run the executable as `./a2` ⟨`filename`⟩. The executable `a2` should produce the output of the lexer followed by a newline, then the parser's output. Lexer output should be a comma-separated list (enclosed in square brackets) of tokens in order of their appearance in the input file. Each token in output should be of the form `<token type> space <actual token in the input file enclosed in double quotes>`. The output of the parser should be the preorder traversal of the generated parse tree. The preorder traversal should be a comma-separated list of each node's representation. Use the production rules to represent a non-terminal node, and `<token type> space <actual token in the input file>` to represent the terminal nodes in preorder traversal.
- Whenever an invalid token is encountered, lexer should generate Unknown `Token:<line no>:<column number>:<token>` error. Here `<line no>` and `<column number>` start from 1, and `<token>` is the invalid token. If the input is not syntatically correct according to the specifications, the parser should generate `Syntax Error:<line no>:<column number>:<production rule>` error. Here `<production rule>` is the production rule where syntax did not match.
- Submit a zip file named `<EntryNumber>-q2.zip`. On unzipping the file, it should produce the files lexer.x, parser.hs, ebnf.txt (a text representation of your language's BNF representation). You may additionally provide an associated `README.md` describing any aspect of your implementation.

Examples
Some examples of valid and invalid programs are as follows:

- `(xyz IMPLIES FALSE) OR TRUE AND if A then b else c` is a valid statement having identifiers `xyz, A, b` and `c`. The lexer output for this example should be as follows: `[LPAREN ‘‘(‘‘, ID ‘‘xyz’’, IMPLIES ‘‘IMPLIES’’, CONST ‘‘FALSE’’, RPAREN, ‘‘)’’, OR ‘‘OR’’, CONST ‘‘TRUE’’, AND ‘‘AND’’, IF ‘‘if’’, ID ‘‘A’’, THEN ‘‘then’’, ID ‘‘b’’, ELSE ‘‘else’’, ID ‘‘c’’]`
- If line 5 of input file is $a \mid b$; lexer should result in error `Unknown token:5:3:|`.
- `if x EQUALS y z then TRUE else a XOR b` is an invalid statement. Observe that all the tokens in this expression are valid; therefore the lexer should produce the correct output. However, the parser should generate an error. If this statement is in line 5, the parser error should be `Syntax Error:5:15:<concerned production rule>`
- Consider a program `1+2*3`. A possible parser output (depending on the type names you choose) can be: `BinExp (AddExp(Num 1, BinExp (MulExp (Num 2, Num 3))))`.

Important Notes

- Do not change any of the names given in this document. You are not even allowed to change upper-case letters to lower-case letters or vice-versa.
- Follow the input/output specification as given. A part of this assignment will be auto-grade. In case of mismatch, you will be awarded zero marks.
- Take care of extra whitespace characters.
- You create new token types if required. Make sure that all the terminals use the token types specified in the document.
- We will create a piazza post titled “A2 Queries”. If you have doubts, please post **only** in this thread. The queries outside this thread or over email will not be entertained.