Project (2): Implementing Simple Compiler

Required:

It is required to implement compiler basic functions, your implementation should include 3 modules:

1. Lexical analysis
2. Parsing
3. Code generation.

Input: A file containing source code in high level language

```
PROGRAM   BASICS
VAR
     X,Y,A,B,C,Z
BEGIN
     READ(X,Y,Z,B)
     A = X+B;
     C = X+ Z;
     C = C * B;
     Z =  A+B+C+Y;
     WRITE(A,C,Z)
END.
```

Output: Assembly code for the source code

Lexical Analysis

1. The output of the lexical analyzer is a stream of tokens. You will use token coding schemes described in the reference and tutorials.
2. For recognizing (classifying) tokens, you can use either regular expressions or finite automatas tabular implementation.

Tokens Coding

PROGRAM	1
VAR	2
BEGIN	3
END	4
END.	5
FOR	6
READ	7

WRITE	8
TO	9
DO	10
;	11
:=	12
+	13
FOR	14
(	15
)	16
id	17
*	18

Parsing

1. You are required to implement the top-down (recursive descent parsing) algorithm to give the parse tree for each line.
2. The parse tree provides the ordering of statements executions and arithmetic operations.
3. You can choose to implement other parsing techniques as shift-reduce algorithm.
4. You should implement the grammar provided in lectures and the reference. You can implement any grammar of your choice.

Grammar

<prog> ::= PROGRAM <prog-name> VAR <id-list> BEGIN <stmt-list> END.
<prog-name> ::= id
<id-list> ::= id | <id-list>, id
<stmt-list> ::= <stmt> | <stmt-list> ; <stmt>
<stmt> ::= <assign> | <read> | <write> | <for>
<assign> ::= id := <exp>
<exp> ::= <factor> + <factor> | <factor> * factor>
<factor> ::= id | (<exp>)
<read> ::= READ (<id-list>)
<write> ::= WRITE (<id-list>)
<for> ::= FOR <index-exp> DO <body>
<index-exp> ::= id := <exp> to <exp>
<body> ::= <stmt> | BEGIN <stmt-list> END

Note: implementing <for>, <index-exp> & <body> is bonus.

Recursive descent parser

For every nonterminal, there is a procedure whose job is to recognize that nonterminal in the input stream. The return value of the procedure is a parse tree for a segment of the program, with the nonterminal at the root. The procedure first decides which production to use (by examining the next input token). It then has a step for each symbol in the right side of the production. If the symbol is a terminal, the procedure matches it with the next input token. If the symbol is a nonterminal, it calls the procedure associated with that nonterminal, receiving a tree in return. When it finishes these steps, it can construct a tree node whose children are the terminals it has matched from the input and the subtrees received from calls to the various nonterminal procedures. The procedure exits with this tree node as its return value.

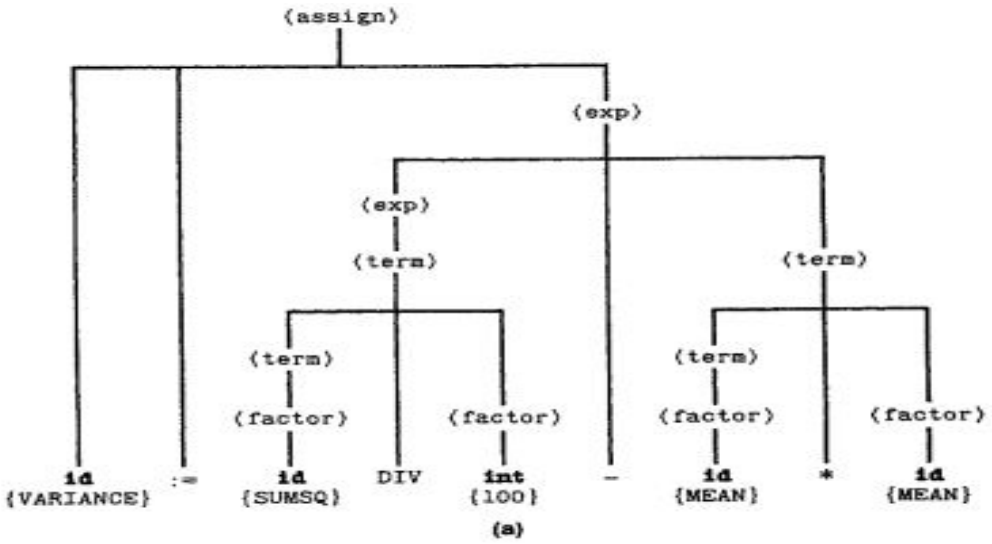
Procedure example for <EXP> rule: (you can find more procedures for other rules in the textbook)

```
procedure EXP
       begin
           FOUND := FALSE
           if TERM returns success then
              begin
                  FOUND := TRUE
                  while ((TOKEN = 16 {+}) or (TOKEN = 17 {-}))
                      and ( FOUND = TRUE ) do
                         begin
                             advance to next token
                             if TERM returns failure then
                                FOUND := FALSE
                         end {while}
              end {if TERM}
           if FOUND = TRUE then
              return success
           else
              return failure
       end {EXP}
```

Note: You can use backtracking in parsing, or other parsers as shift reduce parser.

Code Generation

You should implement your own procedures for code generation.
Each grammar rule will have its own code generations routine.
The generated code will be in SIC/XE.



(a)

```
<assign> ::= id := <exp>

                GETA ( <exp> )
                generate [STA    S(id)]
                REGA := null


<exp> ::= <term>

                S(<exp>) := S(<term>)
                if S(<exp>) = rA then
                    REGA := <exp>


<exp>1 ::= <exp>2 + <term>

                if S(<exp>2) = rA then
                    generate [ADD    S(<term>)]
                else if S(<term>) = rA then
                    generate [ADD    S(<exp>2)]
                else
                    begin
                        GETA (<exp>2)
                        generate [   ADD    S(<term>)]
                    end
                S(<exp>1) := rA
                REGA := <exp>1
```

Final Notes

- You compiler should be invoked from command line as this:
  compiler prog1.txt
- Your output should be an assembly file *prog1.asm*
- You should print the compile time on the output screen
- Generating object code from the assembly file (SIC/XE) is bonus (5 marks)
- Keep the parsing technique as simple as possible (remember parsing just determines the order of execution of statements)
- Extra implemented grammar rules bonus (5 marks)
- You are required to deliver the project according to the following milestones:

	Deliveries	Date
1st Milestone	Report containing RE or for all tokens in the language and description of lexical analysis implementation.	28/4/2018
2nd Milestone	Lexical analysis & Parsing implementation (even if incomplete)	5/5/2018
Final Delivery	Full source code + test cases	12/5/2018

Email: hazemahmed@alexu.edu.eg

References

1. http://web.stanford.edu/class/cs143/
2. http://ccl.pku.edu.cn/doubtfire/NLP/Parsing/Parsing_Algorithm/Top-down/Top-down%20Parsing.htm
3. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-035-computer-language-engineering-spring-2010/
4. https://github.com/ShivamSarodia/ShivyC (reference implementation in python)
5. https://legacy.python.org/workshops/1998-11/proceedings/papers/aycock-little/aycock-little.html