

Languages and Paradigms Chapter 4 Project

Q1) Given the following BNF:

```
<program> -> <stmt>{<stmt>}
<stmt> -> <decl> | <assign>
<decl> -> int <intIds> | float <floatIds>
<intIds> -> <singleInt> | <singleInt>, <intIds>  ≡ <intIdent> → <singleInt>{<singleInt>}
<singleInt> -> <ident> | <ident> = <intLit>
<assign> -> <ident> = <expr>
<expr> → <term> {(+ | -) <expr>}                ≡ <exp> → <term> | <term> (+ | -) <expr>
<term> → <factor> {( * | / ) <term>}              ≡ <term> → <factor> | <factor> ( * | / ) <term>
<factor> → <ident> | <intLit> | ( <expr> )
<ident> → <firstLetter> | <firstLetter>{<Rest>}
<firstLetter> → A | ... | Z | a | ... | z | _
<Rest> → <firstLetter> | <Digit>
<Digit> → 0 | 1 | ... | 9
<intLit> → <Digit>{<Digit>}
```

Attached to this assignment is the parser program (Q1Tokenizer and Q1Parser.cpp);

Q1Tokenizer checks the syntax of a source code program. It reads the source file one word at a time, then converts the word into tokens while the Q1Parser checks the syntax based on the BNF rules above.

Create a folder on the Desktop called **TokenizerParser**. Download the Q1Tokenizer.cpp and Q1Parser.cpp from the Canvas and save it into the folder TokenizerParser.

Then **create a new file** called **prg.in**, then type the following code and save it:

```
int A, B, C, D=2
A = 3 + 5
B = A * 10 + 3
C = 5 + A / D + 3
```

The first line is a declaration statement that defines four variables A, B, C and D. D is initialized to value 2; all other variables are initialized to zero by default. The next three lines are assignment expressions for three variables A, B and C with some value after computing the expression on the right side of the assignment operator.

Now, let us look at the lexical analyzer (Q1tokenizer.cpp) that converts the source program to tokens. The main function is called tokenizer (). Every time you call the function tokenizer() you will read a word from the source program; then convert it into the corresponding token. For

example, the source program in 'prg.in', the first token is int declaration keywords. The second token is identifier A.

The set of tokens are defined using enum type at the top of the parser.cpp program:

```
enum Tokens { LETTER, DIGIT, INT_LIT=10, FLOAT_LIT, IDENT, ASSIGN_OP=20, ADD_OP, SUB_OP,  
MULT_OP, DIV_OP, LEFT_PAREN, RIGHT_PAREN, PERIOD, INT_KEYWORD=40, COMMA, DECL,  
ENDFILE=80, UNKNOWN=99 };
```

Then, there is the symbol table structure, that stores every variable, type and value used in the declaration statement of the program. This is not used in the tokenizer but used in the parser.

The next function is prt, this function helps in printing the tokens as string. It consists of a switch-case for every token and print the token as string to the display.

The next function is the errMsg which displays on the screen the error messages found and increment the error variable to keep track of the number of errors. Then call the lexical analyzer function to get a new token.

The function addChar adds a character to the lexeme variable. The lexeme variable collects all characters for one word.

The function getChar reads a character from the file and returns it. It also increments the line variable if the character read is '\n' (i.e., new line).

The function getNonBlank() calls getChar function to return a character and skips the white spaces (space, tab and new lines).

The function lookupKeywords determines the keywords found in the lexeme; we have two keywords 'int' and 'float' if found then the corresponding token is set 'INT_KEYWORD' or 'FLOAT_KEYWORD'. In addition, we have plus, minus, multiply and divide operators, comma, equal sign, open and close parentheses.

Finally, the tokenizer function which read a character and determine the token as either IDENT, DECL, ADD_OP, SUB_OP, MULT_OP, DIV_OP, EQUAL, COMMA, RIGHT_PAREN, LEFT_PAREN, DIGIT, end of file.

Now, let us test the tokenizer. Type the code in figure 1 inside the function main.

```

/* main driver */
int main() {
    infp.open("./prg2.in");
    if (!infp) {
        cout << "ERROR - cannot open file \n";
        errors++;
    }
    else {
        nextChar = ' ';
        do {
            nextToken = tokenizer();
        } while (nextToken != ENDFILE);
    }
    cout << "Total number of errors: " << errors << endl;
    return 0;
}

```

Figure 1, main code to test the lexical analyzer (tokenizer)

The code will loop until the end of file reached converting every word to a token. It will call the function tokenizer, then call the function prt to display the tokens found, see **figure 2**.

To run the Q1Tokenizer.cpp, open a new terminal, go to the folder where the Q1Tokenizer exist by typing at the terminal:

cd Desktop

cd TokenizerParser

g++ ./Q1Tokenizer.cpp

To run the Q1Tokenizer:

Windows: ./a.exe

MacOS: ./a.out

// ----- End of Tokenizer -----

The next part of the program is the **parser** (or syntax analyzer); which starts with the symbol table; we used a linear symbol table that takes $O(n)$ for searching for a variable (identifier).

There is addSymbolTable, searchSymbolTable and printSymbolTable; then names are self-explanatory.

The next part is calling function based on the BNF, for example expr function calls term and whenever there is an add operator or subtract operator it will calls term and loops back.

Similarly, the term, factor, assign, declaration (decl), stmt, intVar, and intIdent.

Let us test the parser, replace the main with the following code in figure 2.

```
Token read: 43 DECL Lexeme: int
Token read: 12 <IDENT> Lexeme: A
Token read: 42 <COMMA> Lexeme: ,
Token read: 12 <IDENT> Lexeme: B
Token read: 42 <COMMA> Lexeme: ,
Token read: 12 <IDENT> Lexeme: C
Token read: 42 <COMMA> Lexeme: ,
Token read: 12 <IDENT> Lexeme: D
Token read: 23 Lexeme: =
Token read: 10 <INT_LIT> Lexeme: 2
Token read: 12 <IDENT> Lexeme: A
Token read: 23 Lexeme: =
Token read: 10 <INT_LIT> Lexeme: 3
Token read: 21 <ADD_OP> Lexeme: +
Token read: 10 <INT_LIT> Lexeme: 5
Token read: 12 <IDENT> Lexeme: B
Token read: 23 Lexeme: =
Token read: 12 <IDENT> Lexeme: A
Token read: 24 <MULT_OP> Lexeme: *
Token read: 10 <INT_LIT> Lexeme: 10
Token read: 21 <ADD_OP> Lexeme: +
Token read: 10 <INT_LIT> Lexeme: 3
Token read: 12 <IDENT> Lexeme: C
Token read: 23 Lexeme: =
Token read: 10 <INT_LIT> Lexeme: 5
Token read: 21 <ADD_OP> Lexeme: +
Token read: 12 <IDENT> Lexeme: A
Token read: 25 <DIV_OP> Lexeme: /
Token read: 12 <IDENT> Lexeme: D
Token read: 21 <ADD_OP> Lexeme: +
Token read: 10 <INT_LIT> Lexeme: 3
Token read: 80 <ENDOFFILE> Lexeme: EOF
Total number of errors: 0
```

Figure 2. Output of tokenizer for the prg.in source code.

```

int main() {
    infp.open(".\\prg2.in");
    if (!infp) {
        cout << "ERROR - cannot open file \n";
        errors++;
    }
    else {
        nextChar = ' ';
        Tokens nextToken = tokenizer();
        do {
            nextToken = stmt (nextToken);
            if (errors > 10) break;
        } while (nextToken != ENDFILE);
    }
    cout << "Total number of errors: " << errors << endl;
    return 0;
}

```

Figure 3. Test code for parser.

To run the program Q1Parser.cpp, open a terminal and change the folder to the Desktop folder then to the TokenizerParser folder:

cd Desktop

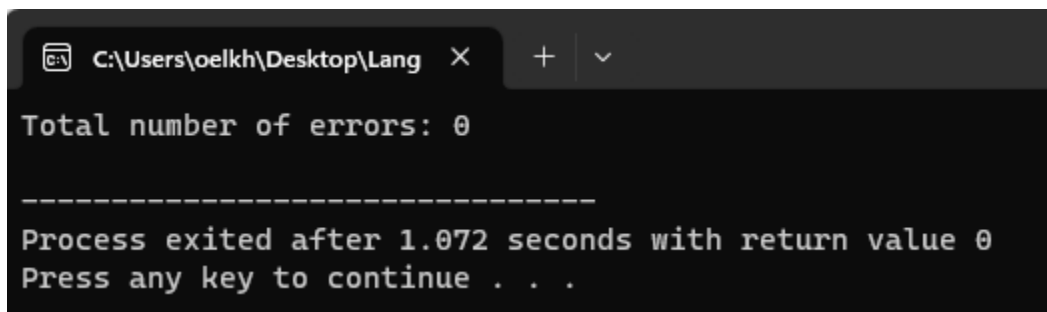
cd TokenizerParser

g++ Q1Parser.cpp

Windows: a.exe

MacOS: a.out

You should see no errors on the screen as in **figure 4**.



```

C:\Users\oelkh\Desktop\Lang >
Total number of errors: 0
-----
Process exited after 1.072 seconds with return value 0
Press any key to continue . . .

```

Figure 4. No error message is displayed when testing parser.

Change prg.in to the following code:

```
int A, B, C, D=2
A = 3 + 5
B = E * 10 + B
C = 5 + A / D + 3
```

Now, go back to the parser and run it, you see the following errors occurring:

```
Error at line: 3: Unknown identifier: E
Total number of errors: 1
```

Change the prg.in program to:

```
int A, B, C, D=2
A = 3 + 5
B = A * 10 ++
C = 5 + A / D + 3
```

Run the parser, and you will see the error:

```
Error at line: 3: missing operand
Total number of errors: 1
```

Change the program 'prg.in' to:

```
int A, B, C, D=2
A = 3 + 5
B = A * 10 +
C = 5 + A / D + 3
```

Run the parser and you will see the following:

```
Error at line: 4: Identifier expected, found =
Error at line: 4: Identifier expected, found 5
Error at line: 4: Identifier expected, found +
Error at line: 4: equal sign expected, found /
Error at line: 4: equal sign expected, found +
Error at line: 5: Identifier expected, found 3
Total number of errors: 6
```

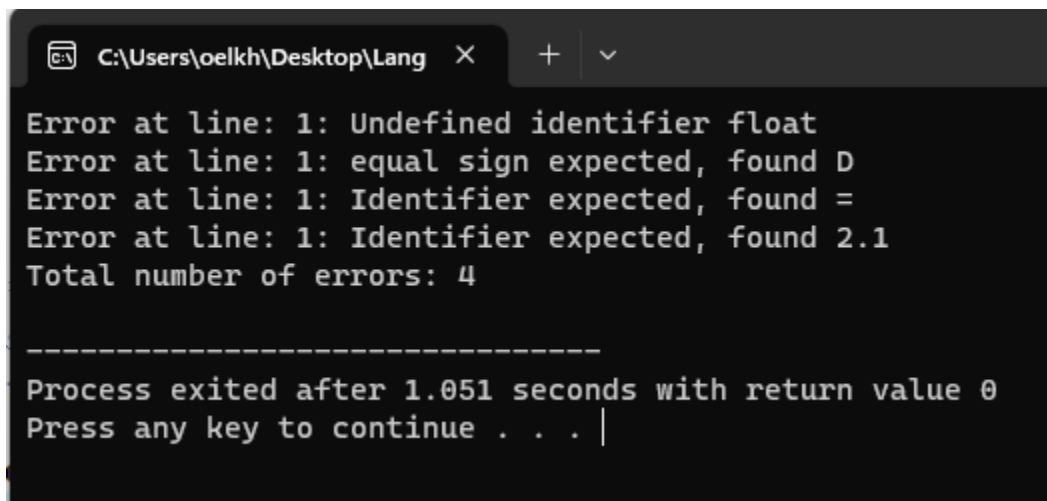
Can you explain why it says Identifier expected, found =? In other words, why identifier expected?

Your answer: There should be a variable name or some sort of identifier after the plus sign, such as an int; this aligns with the syntax of the BNF and language. However, since there is no identifier, the next symbol is an equals sign, which is not an identifier

Now change the prg.in to the following code:

```
int A, B, C
float D=2.1
A = 3 + 5
B = A * 10 + 3
C = 5 + A / D + 3
```

Here we are using floating number 2.1 to initialize D. when you run the parser you will get the following error message:



```
C:\Users\oelkh\Desktop\Lang X + v
Error at line: 1: Undefined identifier float
Error at line: 1: equal sign expected, found D
Error at line: 1: Identifier expected, found =
Error at line: 1: Identifier expected, found 2.1
Total number of errors: 4

-----
Process exited after 1.051 seconds with return value 0
Press any key to continue . . . |
```

Why do you think the error message: equal sign expected, found D?

Answer: Float is not recognized as a word in the language, so the syntax ignores it. It instead goes to D, in which there is no comma. The syntax assumes an equal sign should be there.

The program is missing some code to handle **float** type declaration and numbers. Follow the following steps to make the parser program handle the float numbers:

- Add a new token `FLOAT_KEYWORD` in the Token enum.
- Add a case in the `prt` function to display the `<FLOAT_KEYWORD>` as string.
- Add your code in the `lookupKeywords` function to check if the lexeme contains 'float' type and set the `FLOAT_KEYWORD` token.
- In the `stmt ()` function, replace the `'if (nextToken == INT_KEYWORD) declaration()'` with `'if (nextToken == INT_KEYWORD || nextToken == FLOAT_KEYWORD) declaration()'`.

Now run the parser again, you should see no errors.

C:\Users\oelkh\Desktop\Lang X

+

▼

Total number of errors: 0

Process exited after 1.072 seconds with return value 0

Press any key to continue . . .

Q2) write a tokenizer (Q2Tokenizer.cpp) and parser (Q2Parser.cpp) for the following BNF:

$S \rightarrow \langle A \rangle \langle C \rangle \langle B \rangle \mid \langle A \rangle$

$\langle A \rangle \rightarrow a \langle A \rangle \mid a$

$\langle B \rangle \rightarrow b \langle B \rangle \mid b$

$\langle C \rangle \rightarrow c$

Test your tokenizer and parser on the following programs in **figure 5**:

aacbb

aaabb

Figure 5, program test for Q2

Run the Q2Tokenizer and Q2Parser for the above program and make screenshots.

The first thing you need to do is to change the enum to include the tokens which are: A, B, C, ENDFILE=80, UNKNOWN=99. Then, the prt function should change to display the tokens. After that the lookupKeyword should change to get the token that corresponds to every symbol 'a', 'b' or 'c'. Finally, the tokenizer must be changed to every case: a's, b's and c.

Hint about tokenizer:

To read the token $\langle A \rangle$:

```
if (nextChar == 'a') {  
    addChar(nextChar);  
    nextChar = getChar();  
    nextToken = A;  
}
```

You will need to do the same for B. For C you will check only for the one c that appears.

What to submit?

- Create a github repository and upload all your programs, screenshots and this document with your answers.
- Submit on Canvas the github repository link.