Lexical Analysis

Student: Alexandrescu Andrei-Robert, 931/1

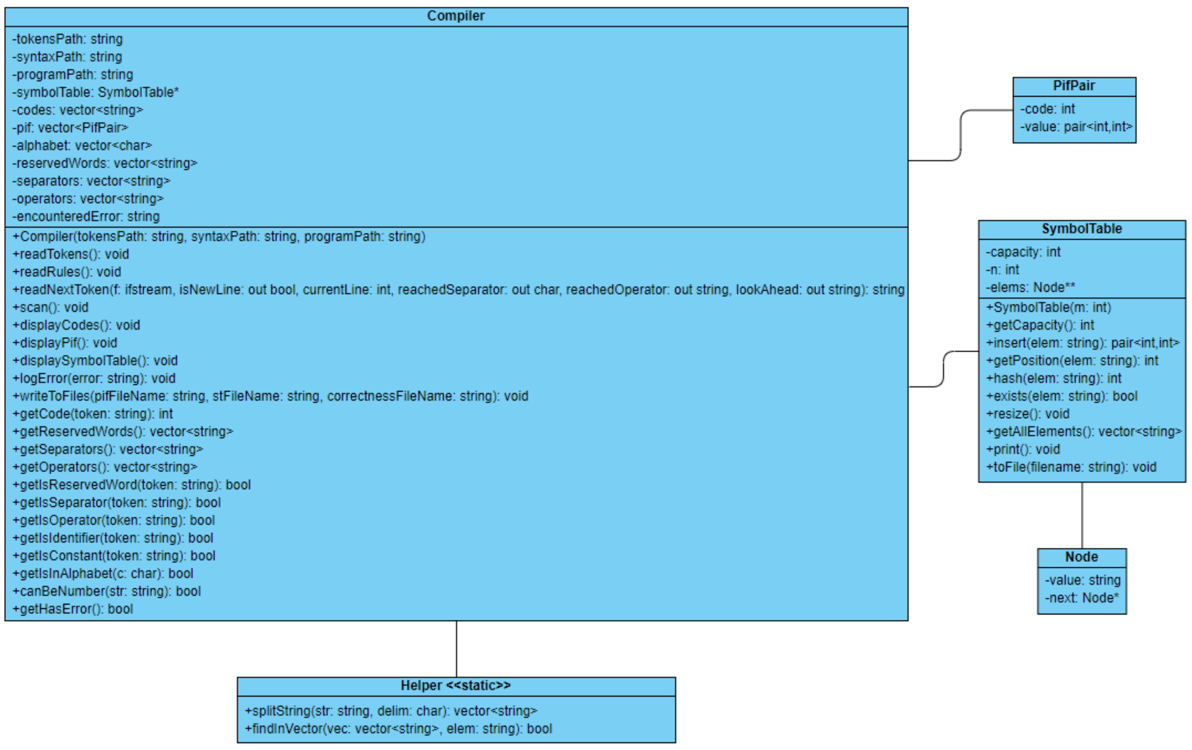
Implementation: [Repository Link]

1. Requirements and Specifications

Considering a small programming language (that we shall call mini-language), you have to write a scanner (lexical analyser). Implement the Symbol Table (ST) as the specified data structure, with the corresponding operations. Implement a scanner (lexical analyzer): Implement the scanning algorithm and use ST from lab 2 for the symbol table.

The SymbolTable is implemented over a hashmap using separate chaining for conflict resolution.

2. Class Diagram


Compiler
-tokensPath: string
-syntaxPath: string
-programPath: string
-symbolTable: SymbolTable*
-codes: vector<string>
-pif: vector<PifPair>
-alphabet: vector<char>
-reservedWords: vector<string>
-separators: vector<string>
-operators: vector<string>
-encounteredError: string
+Compiler(tokensPath: string, syntaxPath: string, programPath: string)
+readTokens(): void
+readRules(): void
+readNextToken(f: ifstream, isNewLine: out bool, currentLine: int, reachedSeparator: out char, reachedOperator: out string, lookAhead: out string): string
+scan(): void
+displayCodes(): void
+displayPif(): void
+displaySymbolTable(): void
+logError(error: string): void
+writeToFiles(pifFileName: string, stFileName: string, correctnessFileName: string): void
+getCode(token: string): int
+getReservedWords(): vector<string>
+getSeparators(): vector<string>
+getOperators(): vector<string>
+getIsReservedWord(token: string): bool
+getIsSeparator(token: string): bool
+getIsOperator(token: string): bool
+getIsIdentifier(token: string): bool
+getIsConstant(token: string): bool
+getIsInAlphabet(c: char): bool
+canBeNumber(str: string): bool
+getHasError(): bool
PifPair
-code: int
-value: pair<int,int>
SymbolTable
-capacity: int
-n: int
-elems: Node**
+SymbolTable(m: int)
+getCapacity(): int
+insert(elem: string): pair<int,int>
+getPosition(elem: string): int
+hash(elem: string): int
+exists(elem: string): bool
+resize(): void
+getAllElements(): vector<string>
+print(): void
+toFile(filename: string): void
Node
-value: string
-next: Node*
Helper <<static>>
+splitString(str: string, delim: char): vector<string>
+findInVector(vec: vector<string>, elem: string): bool

3. Particular aspects

3.1 Token detection

The tokens are detected and split in the *readNextToken* method. This method takes as input:

- *f*: the stream of data (from the file)
- *isNewLine*: an output boolean variable that signals whether the '\n' character was found. This is used for keeping track of the current line read so that the error messages are as explicit as possible
- *currentLine*
- *reachedSeparator* and *reachedOperator* are two output character variables that hold the encountered separator or operator. This is needed in case we have something like: "A+B". If we encounter the +, we want to keep track of the token read so far (which is A) and also the + operator, but deal with it the next iteration. Thus, we store it in this special variable. The same for separators like ; , : and others.
- lookAhead: used for storing the lookahead character if necessary. This is used in case we have reached the "+" or "-", "<" or ">" characters. A lexical error is thrown in case the "+" or "-" is followed by a '0' character. We perform lookahead for '<' and '>' in case they are followed by '='.

The *readNextToken* method is trying to detect whether the token that is currently read is a constant (string, character, number), between square brackets (for array indexing) or an operator or a separator. Further details:

- for string constants we must check for " characters. The complete check is performed in the *getIsConstant* method, but the tokenizing process stops at the second " reached (if there is one);
- similar to string constants, char constant checking works the same, with the exception that char constructs must have length 3 (two ' characters and the character itself). This checking is performed in the *getIsConstant* method;
- detecting array indexing is simple since the square brackets are considered to be separators and, once a separator is read by *readNextToken*, it is immediately returned;
- operator detection is straightforward, performed using the *getIsOperator* method, the only exceptions being the lookahead cases (for +0, -0, <=, >=). Note that we do not use = or == symbols, but the triple equality for checking (===), thus there is no need to lookahead for this symbol. The symbols = and == are not valid tokens, so there can be no misunderstanding when reading the === token;
- separators are simply checked using the *getIsSeparator* method;

3.2 Analysis process

Similar to the one from the lecture. Some particularities are:

- if the variables *reachedSeparator* or *reachedOperator* contain some value (that means we have encountered a separator/operator at the previous step), the next token will be read at the next loop iteration. The current iteration will consider the value from *reachedSeparator*/*reachedOperator* respectively.

4. Tests

4.1 Sample program (Without lexical errors)

```
START
DEF N:NUMBER
READ N
DEF NUMBERS:ARRAY[N] OF NUMBER; MAXNUM:NUMBER; IDX:NUMBER
ASSIGN MAXNUM:UNDEFINED
ASSIGN IDX:-5
WHILE IDX<N
STARTWHILE
    READ NUMBERS[IDX]
    ASSIGN IDX:(IDX-1)
FINISHWHILE
ASSIGN IDX:0
WHILE IDX <= N
STARTWHILE
    IF MAXNUM === UNDEFINED || NUMBERS[IDX] > MAXNUM
    STARTIF
        ASSIGN MAXNUM:NUMBERS[IDX]
    FINISHIF
    ASSIGN IDX:(IDX+1)
FINISHWHILE
PRINT("The maximum number is " + MAXNUM)
FINISH
```

Remarks:

- **DEF N:NUMBER** is analyzed into 4 tokens: **DEF**, **N**, **:**, **NUMBER**. Note that there is no need for spaces between the tokens **N**, **:** and **NUMBER**
- **DEF NUMBERS:ARRAY[N]** is analyzed into the corresponding tokens: **DEF**, **NUMBERS**, **:** , **ARRAY**, **[**, **N**, **]**
- **ASSIGN IDX:-5** is tokenized into: **ASSIGN**, **IDX**, **:**, **-5**
- **ASSIGN IDX:(IDX-1)** is tokenized into: **ASSIGN**, **IDX**, **:**, **(**, **IDX**, **-**, **1**, **)**
- **WHILE IDX** <= **N** is analyzed into: **WHILE**, **IDX**, <=, **N** using the lookahead method for obtaining the <= token
- **PRINT("The maximum number is " + MAXNUM)** is analyzed into: **PRINT**, **(** **“The maximum number is ”**, +, **MAXNUM**, **)**

4.2 Sample program (With lexical errors)

Note: the program stops at the first error

```
START
DEF N$:NUMBER
READ 2N
DEF NUMBERS:ARRAY[N] OF NUMBER; MAXNUM:NUMBER; IDX:NUMBER
ASSIGN MAXNUM:UNDEFINED
```

```
6. ASSIGN IDX:-0
7. WHILE IDX<N
8. STARTWHILE
9.     READ NUMBERS[IDX]
10.    ASSIGN IDX:(IDX+1)
11.FINISHWHILE
12.ASSIGN IDX:0
13.WHILE IDX < N
14.STARTWHILE
15.    IF MAXNUM === UNDEFINED || NUMBERS[IDX] > MAXNUM
16.    STARTIF
17.        ASSIGN MAXNUM:NUMBERS[IDX]
18.    FINISHIF
19.    ASSIGN IDX:(IDX+1)
20.FINISHWHILE
21.PRINT(‘The maximum number is ‘ + MAXNUM)
22.FINISH
```

Errors:

- [LEXICAL ERROR at line 2: token `N$` could not be classified.]
- [LEXICAL ERROR at line 3: token `2N` could not be classified.]
- [LEXICAL ERROR at line 6: token `-0` is invalid.]
- [LEXICAL ERROR at line 21: token `'The` invalid character format.] in case ‘’ marks are used for writing a constant string
- [LEXICAL ERROR at line 21: token `'The` is missing a closing '.] in case the quote is not closed
- [LEXICAL ERROR at line 21: token `"The maximum number is  ` could not be classified.] in case the double quote is not closed