Formal Languages and Compiler Design - Lab3

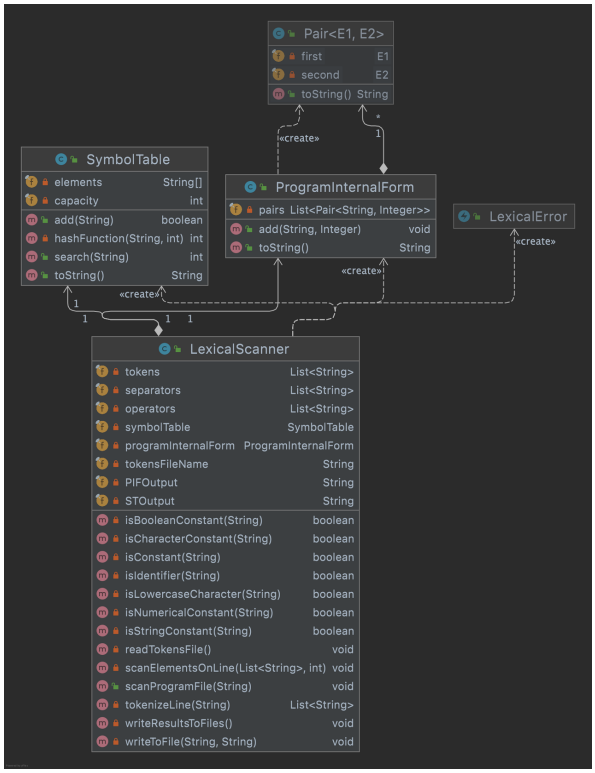
Requirement

Statement: Implement a scanner (lexical analyzer): Implement the scanning algorithm and use ST from lab 2 for the symbol table.

Input: Programs p1/p2/p3/p1err and token.in (see Lab 1a)
Output: PIF.out, ST.out, message “lexically correct” or “lexical error + location”

Design

Class Diagram



SymbolTable

Representation

The Symbol Table is represented as a Hash Table.
The SymbolTable class contains an array of String and a given capacity.

```
public class SymbolTable {
    private String[] elements;
    private int capacity;
}
```

Collision resolution

The collision resolution method used is Open addressing with Linear Probing. The hash function encodes the string as the sum of the ASCII codes of the characters and then computes the value as a linear probing hash function.

```
h(k, i) = (sumOfASCIICodes(k) % capacity + i) % capacity, i = 0, ..., capacity - 1
```

Methods

- add

```
public boolean add(String key)
```

    - verifies if the key already exists and returns false if it exists
    - searches for an empty position to insert the key using the hash function
    - returns true if the element was added
- search

```
public int search(String key)
```

    - searches the key of the positions computed with the hash function
    - returns the position of the key or -1 if the key does not exist in the table

ProgramInternalForm

The ProgramInternalForm contains a list of pairs.

```
public class ProgramInternalForm {
    private final List<Pair<String, Integer>> pairs;
}
```

(token, 0) is added in case of reserved words, operators and separators (tokens from token.in (http://token.in/) file) or (token, positionInSymbolTable) in case of identifier or constant.

```
public void add(String token, Integer positionInSymbolTable)
```

LexicalScanner

```
public class LexicalScanner {
    private final List<String> tokens;
    private final List<String> separators;
    private final List<String> operators;
    private final SymbolTable symbolTable;
    private final ProgramInternalForm programInternalForm;

    private final String tokensFileName = "src/files/token.in";
    private final String PIFOutput = "src/files/PIF.out";
    private final String STOutput = "src/files/ST.out";
}
```

The method `scanProgramFile(fileName)` with the file name of the program as an argument can be called in order to run the scanning algorithm.

It will first call `tokenizeLine(line)` in order to detect the tokens and then it will call `scanElementsOnLine(elementsOnLine, lineNumber)` for detecting possible lexical errors.

The last method will throw a `LexicalError` indicating the line number and the token in case an illegal token is found. In this situation, the error message will be added to the list of lexical errors and printed at the end. Otherwise, after all the lines have been tokenized and scanned, the message "Lexically correct" will be printed in the console.

After the program has been analysed, the PIF.out and ST.out files are created.

Besides the main methods, the LexicalScanner also has the following functions:

- readTokensFile() - reads the file with the tokens
- isIdentifier(String element) - checks if the given string is a valid identifier
- isCharacterConstant(String element) - checks if the given string is a character constant
- isStringConstant(String element) - checks if the given string is a string constant
- isNumericalConstant(String element) - checks if the given string is a numerical constant
- isLowercaseCharacter(String element) - checks if the given string is a lowercase character
- writeToFile(String fileName, String content) - writes content to fileName
- writeResultsToFiles() - writes the ST and PIF to ST.out and PIF.out

Lexic

- Alphabet:
  - letters of the English alphabet (A-Z, a-z)
  - digits (0-9)
  - _ (underscore)
- Lexic:
  - special symbols
    - operators: + - * / % < <= > >= = != := AND OR
    - separators: [] {} ; : ' '
    - reserved words: ARRAY CHAR INT IF ELSE THEN DO WHILE READ WRITE
  - identifiers
    - identifier ::= (letter | _) {letter | digit | _ }
    - letter ::= "A" | "B" | ... | "Z"
    - digit ::= "0" | "1" | ... | "9"
  - constants
  - const ::= no_const | character_const | string_const | boolean_const
    - integers
      - no_const ::= sign? non_zero_digit {digit} | "0"
      - non_zero_digit ::= "1" | "2" | ... | "9"
      - digit ::= "0" | non_zero_digit
      - sign ::= "-"

- character
  - character_const ::= ‘letter’ | ‘digit’ | ‘ ’
- string
  - string_const ::= “string”
  - string ::= char{char} | “” (empty string)
  - char ::= letter | digit | ‘ ’
- boolean
  - boolean_const ::= “true” | “false”

Implementation

Github Link

Tests

p1 - max of 3 numbers

```
START
 A: INT; B: INT; C: INT; MX1: INT; MX: INT;
 BEGIN
  READ (A);
  READ (B);
  READ (C);

  IF A>B THEN
   MX1 := A;
  ELSE
   MX1 := B;
  IF C > MX1 THEN
   MX := C;
  ELSE
   MX := MX1;
  WRITE (MX);
 END
ENDPRG
```

p2 - gcd of 2 numbers

```
START
 A: INT; B: INT; AUX: INT; R: INT;

 BEGIN
  READ (A);
  READ (B);

  IF A > B THEN
   BEGIN
    AUX := A;
    A := B;
    B := AUX;
```

```
   END

  WHILE R != 0 DO
   BEGIN
    R := B % A;
    A := B;
    B := R;
   END

  WRITE (A);
 END
ENDPRG
```

p3 - sum of N numbers

```
START
 N: INT; SUM: INT; I: INT; X: INT;

 BEGIN
  READ (N);
  SUM := 0;
  I := 0;

  WHILE I < N DO
   BEGIN
    READ (X);
    SUM := SUM + X;
    I := I + 1;
   END

  WRITE (SUM);
 END
ENDPRG
```

p1err - verify if number is prime

```
START
 #
 A := 'G
 B := -0
 C := 6-9

 a: INT; I: INT; Result: BOOLEAN;

 BEGIN
  READ (a);
  Result := True;
  I := 2;

  WHILE I < (a / 2) DO
   BEGIN
    IF a % I = 0 THEN
     Result := false;
```

```
    I := I + 1;
   END

  WRITE (Result);
 END
ENDPRG
```

Errors:

- Lexical error at line 2: Invalid token #
- Lexical error at line 3: Invalid token 'G
- Lexical error at line 4: Cannot have constant -0
- Lexical error at line 7: Identifier starts with lowercase letter a
- Lexical error at line 10: Identifier starts with lowercase letter a
- Lexical error at line 14: Identifier starts with lowercase letter a
- Lexical error at line 16: Identifier starts with lowercase letter a