Scanner

Requirement:

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

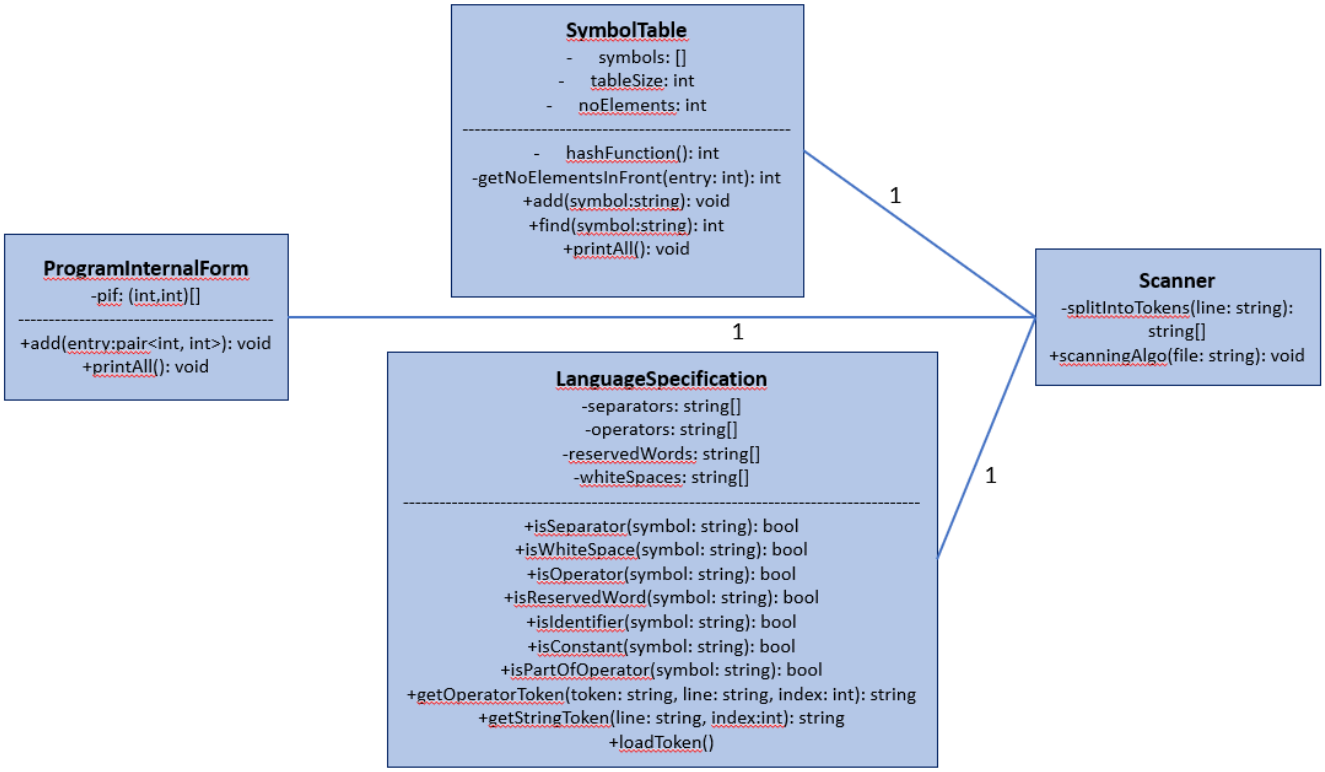
Input: Programs p1/p2/p3/p1err and token.in (see Lab 1a)

Output: PIF.out, ST.out, message “lexically correct” or “lexical error + location”

Details:

- ST.out should give information about the data structure used in representation
- If there exists an error the program should give a description and the location (line and token)

Class Diagram:



Language specification:

1. Alphabet:
    a. upper (A-Z) and lower (a-z) case letters of the English alphabet
    b. Decimal digits (0-9)
2. Lexic:
    a. Operators: + - * := < <= > >= = != ++ --
    b. Separators: space ; ( ) { } [ ] ,
    c. Reserved words: if and or not read write const array int char string for div mod true false
    d. Identifiers: a sequence of letters and digits, without _, such that the first character is a letter, having no more than 100 characters
        i. identifier ::= letter | letter {(letter | digit)}
        ii. letter ::= "A" | "B" | . ..| "Z" | "a" | "b" | . ..| "z"
        iii. digit ::= "0" | "1" |...| "9"
    e. Constants:
        i. Int
            1. number ::= nonZeroDigit{digit}
            2. nonZeroDigit ::= "1" |...| "9"
            3. sign ::= + | -
            4. zero ::= “0”
            5. int ::= [sign] number | zero
        ii. Char
            1. character ::= ‘symbol’
        iii. String
            1. string ::= “{symbol}”
            2. symbol ::= A | B | … | Z | a | b | … | z |0 | … | 9 | space | # | & | ^ | %
        iv. Bool
            1. bool ::= true | false
3. Syntax

The words - predefined tokens are specified between " and " :

Program ::= declarationList “;” statementList “;”

declarationList ::= declaration | declaration “;” declarationList

declaration ::= typeSimple (IDENTIFIER | assignment) | “;” declaration

typeSimple ::= “int” | “char” | “bool” | “string”

arrayDeclaration ::= “array” “[” CONSTANT “]” typeSimple

type ::= typeSimple | arrayDeclaration

expression ::= expression “+” term | expression “-” term | term

term ::= term “*” factor | term “DIV” factor | term “MOD” factor | factor

factor ::= “(” expression “)” | IDENTIFIER | CONSTANT | IDENTIFIER “[” CONSTANT “]”

statementList ::= statement | statement “;” statementList

statement ::= simpleStatement | structStatement

simpleStatement ::= assignment | inOutStmt

assignment ::= IDENTIFIER “:=” expression

inOutStmt ::= “READ” “(” IDENTIFIER “)”|“WRITE” “(” IDENTIFIER “)”|“WRITE” “(” CONSTANT“)”

structStatement ::= ifStmt | forStmt

ifStmt ::= “IF” “(” condition “)” “{” statement “}” [“ELSE” “{” statement “}”]

condition ::= expression RELATION expression

forStmt ::= “FOR” “(”assignment “;” condition “;” stepForStmt “;” “)” “{” statement “}”

stepForStmt ::= (IDENTIFIER “++”) | (IDENTIFIER “--”) | assignment

RELATION ::= “<” | “<=” | “>” | “>=” | “=” | “!=”

Implementation details:

Github: ***https://github.com/adabirtocian/Scanner_FLCD/tree/lexical-analyzer-lab3***

Symbol Table:

Data structure: ***Hash table with separate chaining***

- Collisions are solved by having linked lists, so the symbols that hash on the same value are added in a vector one after the other.
- Hash function: sum of ascii codes modulo hash table size

Constructor

Initial size of the hash table is 100

void add(string symbol)

- Precondition: symbol should be string
- Postcondition: symbol is added if not already in the symbol table
- Checks is the symbol was already added and adds it if not

int find(string symbol)
- Precondition: symbol should be string
- Postcondition: -1 if the symbol is not in the symbol table; index where it is found
- Return the exact position inside the symbol table
- For each chain there are maximum 10 symbols that can be added so the real position is computed based on the hash value and the index in the linked list

Scanner:

- Extracting tokens
    - Take each line at a time.
    - Each line is taken character by character
    - Spaces and tabs are excluded
    - If a separator is found, it adds it in the tokens list
    - If an operator is found, it checks for composed operators and it adds it in the tokens list
    - If a reserved word is found, it adds it in the tokens list
    - If a string is found (between “”), it adds it in the tokens list
    - If a char is found (between ‘’), it adds it in the tokens list
- Scanning algo
    - Reads one line at a time from the file
    - Tokenizes the line and returns a list of tokens
    - Process each token and completes the pif and symbol table
    - Throws error at the first lexical error encountered

Testing:

Input:

```
int n, sumNums:=0, a;
a:=-9+7-(-3);
a:=+0;
a:=-0;
read(n);
array[100] int nums;
for (int i:=0; i<n; i++)
{
        read(nums[i]);
        sumNums := sumNums + nums[i];
}
int averageNums := sumNums div n;
write(averageNums);
```

Output:

Lexically correct

Program internal form	Symbol table
int -- -1	=====2=====
id -- 100	(-9, 20)
, -- -1	
id -- 600	=====5=====
:= -- -1	(i, 50)
constant -- 480	
, -- -1	=====10=====
id -- 970	(n, 100)
; -- -1	
id -- 970	=====45=====
:= -- -1	(100, 450)
constant -- 20	
+ -- -1	=====48=====
constant -- 550	(0, 480)
- -- -1	
(-- -1	=====50=====
constant -- 960	(averageNums, 500)
) -- -1	
; -- -1	=====51=====
id -- 970	(nums, 510)
:= -- -1	
constant -- 480	=====55=====

; -- -1 id -- 970 := -- -1 constant -- 480 ; -- -1 read -- -1 (-- -1 id -- 100) -- -1 ; -- -1 array -- -1 [-- -1 constant -- 450] -- -1 int -- -1 id -- 510 ; -- -1 for -- -1 (-- -1 int -- -1 id -- 50 := -- -1 constant -- 480 ; -- -1 id -- 50 < -- -1 id -- 100 ; -- -1 id -- 50 ++ -- -1) -- -1 { -- -1 read -- -1 (-- -1 id -- 510 [-- -1 id -- 50] -- -1) -- -1 ; -- -1 id -- 600 := -- -1 id -- 600 + -- -1	(7, 550) =====60===== (sumNums, 600) =====96===== (-3, 960) =====97===== (a, 970)

id -- 510 [-- -1 id -- 50] -- -1 ; -- -1 } -- -1 int -- -1 id -- 500 := -- -1 id -- 600 div -- -1 id -- 100 ; -- -1 write -- -1 (-- -1 id -- 500) -- -1 ; -- -1	

Input:

```
int  a, b, c_num;
read(a);
read(b);
read(c_num);
if ( b > a and b > c_num)
{
        write(b);
}
else if (a > b and a > c_num)
{
        write(a);
}
else(c_num > a and c_num > b)
{
        write(c_num);
}
```

Output:

Lexical error at line 1 token c_num

Program internal form	Symbol table
int -- -1 id -- 970 , -- -1 id -- 980 , -- -1	=====97===== (a, 970) =====98===== (b, 980)

Input:

```
int n;
bool isprime:=true;
read(n);
for(int i:=2; i <= n div 2; i++)
{
        if (n mod i = 0)
        {
                isprime:=false;
        }
}
if(isprime)
{
        write("prime”);
}
else
{
        write("not prime");
}
```

Output:

Lexically correct

Program internal form	Symbol table
int -- -1 id -- 100 ; -- -1 bool -- -1 id -- 610 := -- -1 true -- -1 ; -- -1 read -- -1 (-- -1 id -- 100) -- -1 ; -- -1 for -- -1 (-- -1 int -- -1 id -- 50 := -- -1 constant -- 500 ; -- -1	=====5===== (i, 50) =====9===== ("prime", 90) =====10===== (n, 100) =====48===== (0, 480) =====50===== (2, 500) =====61===== (isprime, 610) =====78===== ("not prime", 780)

id -- 50 <= -- -1 id -- 100 div -- -1 constant -- 500 ; -- -1 id -- 50 ++ -- -1) -- -1 { -- -1 if -- -1 (-- -1 id -- 100 mod -- -1 id -- 50 = -- -1 constant -- 480) -- -1 { -- -1 id -- 610 := -- -1 false -- -1 ; -- -1 } -- -1 } -- -1 if -- -1 (-- -1 id -- 610) -- -1 { -- -1 write -- -1 (-- -1 constant -- 90) -- -1 ; -- -1 } -- -1 { -- -1 write -- -1 (-- -1 constant -- 780) -- -1 ; -- -1 } -- -1	

Input:

```
int n;
bool isprime:=true;
read(n);
for(int i:=2; i <= n div 2; i++)
{
        if (n mod i = 0)
        {
                isprime:=false;
        }
}
if(isprime)
{
        write("prime);
}
else
{
        write("not prime");
}
```

Output:

Lexical error at line 13 token "prime);"

Program internal form	Symbol table
int -- -1 n -- 100 ; -- -1 bool -- -1 isprime -- 610 := -- -1 true -- -1 ; -- -1 read -- -1 (-- -1 n -- 100) -- -1 ; -- -1 for -- -1 (-- -1 int -- -1 i -- 50 := -- -1	=====5===== (i, 50) =====10===== (n, 100) =====48===== (0, 480) =====50===== (2, 500) =====61===== (isprime, 610)

2 -- 500 ; -- -1 i -- 50 <= -- -1 n -- 100 div -- -1 2 -- 500 ; -- -1 i -- 50 ++ -- -1) -- -1 { -- -1 if -- -1 (-- -1 n -- 100 mod -- -1 i -- 50 = -- -1 0 -- 480) -- -1 { -- -1 isprime -- 610 := -- -1 false -- -1 ; -- -1 } -- -1 } -- -1 if -- -1 (-- -1 isprime -- 610) -- -1 { -- -1 write -- -1 (-- -1	