https://github.com/ComanacDragos/ToyLanguageCompiler

Statement: Considering a small programming language (that we shall call mini-langauge), write 3 small programs in this language.

Deliverables: p1.* , p2.*, and p3.* and p1err.* - small programs written in your programming language (p1, p2, p3 should be lexically correct; p1err should contain 2 types of lexical errors).

For example:

p1 and p2: compute de max/min of 3 numbers; verify if a number is prime, compute gcd of 2 numbers, compute the solutions for a 2nd order equation, aso

p3: compute **the sum of n numbers, computer the max/min of n numbers**

Statement: Considering a small programming language (that we shall call mini-language), you have to write a scanner (lexical analyser)

Task 1: Minilanguage Specification

Deliverables:

Lexic.txt (file containing mini language lexic description; see example)

token.in (containing the list of all tokens corresponding to the minilanguage)

Syntax.in - the syntactical rules of the language

Statement: Implement the Symbol Table (ST) as the specified data structure, with the corresponding operations

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”

Language specification

letter ::= "a"|"b"|...|"z"|"A"|...|"Z"

digit ::= "0"|"1"|"2"|...|"9"

non_zero_digit ::= "1"|"2"|...|"9"

symbols ::= "_"

unary_operator ::= "!"

binary_operator ::= "+" | "-" | "*" | "/" | "^" | "%"
| "and" | "or"
| ">" | "<" | ">=" | "<="| "!=" | "=="

operator ::= "=" | unary_operator | binary_operator

separators ::= "[" | "]" | "{" | "}" | ";" | "space" | "newline"

identifier ::= letter{letter|digit|symbol}\{0,255\} //at most 256 characters

number ::= non_zero_digit{digit}

const_int ::= ("+"|"-")?number | "0"

const_float ::= ("+"|"-")?(number|"0")"."(digit{digit})

const_character ::= """character"""

character ::= letter|digit|symbol

const_string ::= \"string\"

string ::= {character|\s}

const_bool ::= "true" | "false"

reserved_words ::= "if"
| "while"
| "bool"
| "char"
| "int"
| "string"
| "float"

// Syntax

program ::= statement_list

statement_list ::= statement | statement statement_list

statement ::= simple_statement | compund_statement

simple_statement ::= (assignment_statement
| iostatement

| declaration_statement)";"

compound_statement ::= if_statement | while_statement

simple_type ::= "bool"

| "char"

| "int"

| "string"

| "float"

array_type ::= simple_type"["number"]"

type ::= simple_type | array_type

constant ::= const_int

| const_float

| const_character

| const_string

| const_bool

expression ::= constant

| identifier

| identifier"["number"]"

| expression binary_operator expression

| unary_operator expression

| "("expression")"

declaration_statement ::= type identifier

| type identifier"="expression

iostatement ::= ("<<"identifier) | (">>"expression)

assignment_statement ::= identifier "=" expression

if_statement ::= if "("expression")" "{"statement_list"}" ["else" "{"statement_list"}"]

while_statement ::= while "("expression")" "{"statement_list"}"

Atom

---

identifier

constant

int

char

bool

string

float

>>

<<

while

if

else

and

or

!

+

-

*

/

%

>

<

>=

<=

!=

==

=

;

[

]

{

}

(

)

.

^

p1

#computes the maxium

int a=9;

int b=6;

if(a>b){

	>>"a is the maximum";

}else{

	>>"b is the maximum";

}

p2

#computes the gcd

int a=9;

int b=6;

while(a!=b){

```
        if (a>b){
                a=a-b;
        }
        if (a<b){
                b=b-a;
        }
}
>>a." is the gcd";
```

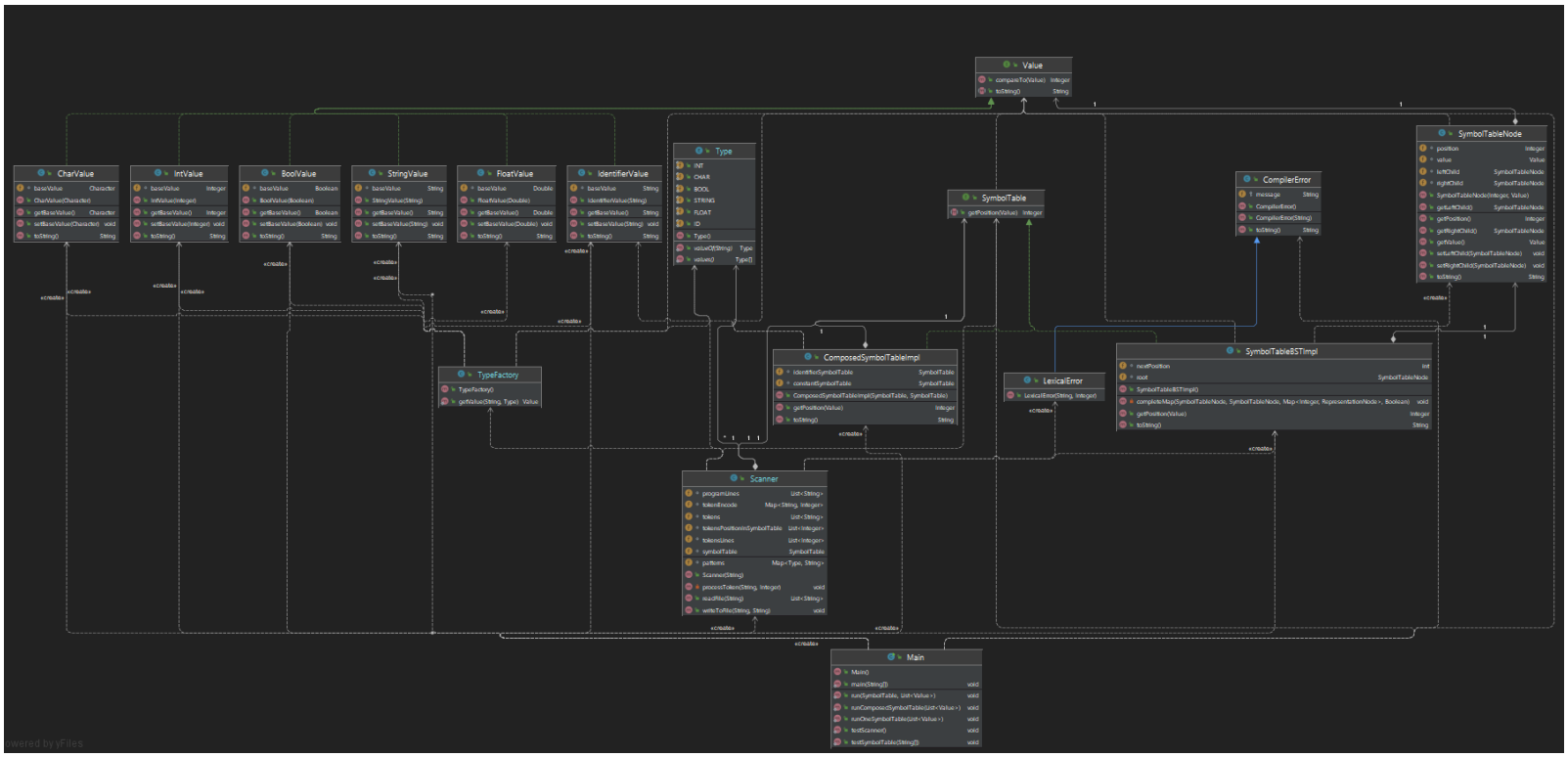
p3

```
#prints the square of the elements of an array
int[256] a;
int i=0;
int n;
<<n;
while (i<n){
        <<a[i];
        i=i+1;
}
i=0;
while (i<n){
        >>"square of",a[i]," is ",a[i]^2;
        i=i+1;
}
```

p4

```
1a=9;
@b=6;
if (a>b){
        >>"a is the maximum
}else{
        >>"b is the maximum"
}
```

Implementation



Symbol table details:

Value
// default implementation: compares lexicographically the string representation of the Value with
// the string representation of the otherValue. If the representations are equal 0 is returned
// If the Value representation is less than otherValue representation a negative value is returned
// otherwise a positive value is returned
default Integer compareTo(Value otherValue)

BoolValue, CharValue, FloatValue, IdentifierValue, IntValue, StringValue are implementing Value and each have a baseValue and their representation according to the specification

SymbolTableNode has the following attributes
    Integer position; // position in the symbol table
    Value value; // the value in the symbol table
    SymbolTableNode leftChild; // reference to the leftChild
    SymbolTableNode rightChild; // reference to the rightChild

public interface SymbolTable {
    /*
    Returns the position of the value if the value exists,
     otherwise it inserts the value and returns the new position

```
     */
    Integer getPosition(Value value);
}

public class SymbolTableBSTImpl implements SymbolTable and has the following attributes
    int nextPosition = 0; // represents the position of the next value to be inserted
    SymbolTableNode root; // represents the root of the tree


ComposedSymbolTableImpl implements SymbolTable and has 2 symbol tables
    SymbolTable identifierSymbolTable; // for identifiers
    SymbolTable constantSymbolTable; // for constants


Scanner details:

class Scanner

    //program split by newline

    List<String> programLines;

    //map which encodes each token that can appear in the program

    Map<String, Integer> tokenEncode;

    //tokens of the program -- first column of PIF

    List<String> tokens;

    //the position of each token in the symbol table -- second column in PIF

    List<Integer> tokensPositionInSymbolTable;

    //the line of each token in the program -- third line in PIF

    List<Integer> tokensLines;



    SymbolTable symbolTable = new SymbolTableBSTImpl();


    //patterns corresponding to each constant and ID

    Map<Type, String> patterns;
```

Receives the program and outputs the FIP and SymbolTable to a directory corresponding to the program name

- program and tokens are read from file
- each line is split by the set of simple operators and by the white spaces that are followed by at least 2 quotes
- empty lines are removed
- look ahead is applied to create composed tokens
- the token is processed
- FIP and Symbol table are written to files

public Scanner(String program)

Receives a token and a line

PIF is represented by the 3 lists: tokens, tokensLines, tokensPositionInSymbolTable

Classifies the token and adds it to the PIF otherwise it throws a LexicalError at the given line

- if the token is an operator separator or reserved word it is added to the PIF with the given line and the position -1
- if it is an id or a constant it is added to the PIF with the corresponding type (id or constant) and to the Symbol table according to the pattern that the token matches
- otherwise a lexical error is thrown

private void processToken(String token, Integer line)

//read the lines from a file

public List<String> readFile(String file)

//write to a file the content

public void writeToFile(String file, String content)

Types corresponding to the types of values in the symbol table

public enum Type

Type factory that generates the corresponding Value class given a token and a type

public class TypeFactory

Testing

Input:

```
int a=9;
int     b=6;
if(a>b){
   >>"a is the maximum";
}else{
   >>bbbb+"b is the maximum" ;
}


 >>0.0

   >>1.3
   <<=0.1
   >>+0.001
   >>-3
   >>-11111.1
   !=
   <=
   >=
   ==
   ^
   !a%bbb
-'a'
!"aa aa"
```

Output:

FIP:

token,position,line
int,-1,1
id,0,1
=,-1,1
constant,1,1
;,-1,1
int,-1,2
id,2,2
=,-1,2
constant,3,2
;,-1,2
if,-1,3
(,-1,3
id,0,3
>,-1,3
id,2,3
),-1,3
{,-1,3
>>,-1,4
constant,4,4
;,-1,4
},-1,5
else,-1,5
{,-1,5
>>,-1,6
id,5,6
+,-1,6
constant,6,6
;,-1,6
},-1,7
>>,-1,10
constant,7,10
>>,-1,12
constant,8,12
<<,-1,13
=,-1,13
constant,9,13
>>,-1,14
constant,10,14
>>,-1,15
constant,11,15

>>,-1,16
constant,12,16
!=,-1,17
<=,-1,18
>=,-1,19
==,-1,20
^,-1,21
!,-1,22
id,0,22
%,-1,22
id,13,22
-,-1,23
constant,14,23
!,-1,24
constant,15,24

ST:

position,value,parent,sibling
0,a,-1,-1
1,9,0,2
2,b,0,1
3,6,1,-1
4,"""a is the maximum""",3,-1
5,bbbb,2,-1
6,"""b is the maximum""",4,-1
7,0.0,6,15
8,1.3,7,11
9,0.1,8,-1
10,0.001,9,-1
11,-3,7,8
12,-11111.1,11,-1
13,bbb,5,-1
14,"'a'",12,-1
15,"""aa aa""",6,7

Error program:

```
a=9;
a+012
b='aa';
if (a>b){
   >>"a is the maximum
}else{
   >>"b is the maximum"
}
```

Output: Lexical error at line: 2 for token: '012'