Lab4

https://github.com/915-Nichifor-Dragos/FLCD/tree/master/Lab3/lab-3

P1.txt

```
int a, b, c;


read(a);

read(b);

read(c);


int maximum = a;


if (b > maximum) {
        maximum = b;
}


if (c > maximum) {
        maximum = c;
}


write(maximum);
```

P2.txt

```
# Prime number


int x;

bool isPrime = true;


read(x);


if (x < 2 or x > 2 and x % 2 == 0) {
	isPrime = false;
}


int d = 3;


while (d * d <= x) {
	if (x % d == 0) {
		isPrime = false;
	}
	d = d + 2;
}


write(isPrime);
```

P3.txt

```
# Sum of n numbers

numbers = list;
int x, n;
int sum = 0, count = 0;

write("How many numbers will you sum up?")

read(n);

while (count < n) {
        read(x);
        numbers.append(x);
        count = count + 1;
}

int index = 0;

while (index < numbers.size()) {
        sum = sum + numbers[i];
        index = index + 1;
}

write(sum);
```

P1err.txt

```
# Maximum of 3 numbers


# A variable name should not start with a digit or a special character

# $ does not a variable name, an operator or an identifier


int 1a, b, c;


read(a);

read(b);

read(c);


int maximum = a;


if (b > maximum) {

        maximum $ b;

}


if (c > maximum) {

        maximum = c;

}


write(maximum);
```

Lexic.txt

Alphabet:

- Upper (A-Z) and lower case letters (a-z) of the English alphabet
- Underline character '_'
- Decimal digits (0-9)

Lexic:

- special symbols:
  - operators: + - * / % < <= = >= == != and or
  - separators: [ ] { } : ; , space " ' ( )
  - reserved words: function int string bool list and or read write if else while true false
- identifiers: a sequence of letters and digits, such that the first character is

a letter or underscore with the rule being:

identifier = ("_" | letter){letter|digit|"_"}

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

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

- constants:

integer_constant = "0" | ["+"|"-"]nonzero-digit{digit}

nonzero-digit = "1" | "2" | ... | "9"

string_constant = """{letter|digit|"_"|" "}"""

char-constant = "'"(letter|digit|special_character)"'"

special_character = "+" | "-" | "*" | "<" | ">" | ...

Syntax.txt

program = function IDENTIFIER "{" stmtlist "}"

declaration = "type" IDENTIFIER

decllist = declaration | declaration "," decllist

type1 = "int" | " "string" | "bool"

arraydecl = "list"

type = type1 | arraydecl

stmtlist = stmt | stmt ";" stmtlist

stmt = simplstmt | structstmt

simplstmt = assignstmt | iostmt | declaration

assignstmt = IDENTIFIER "=" expression

iostmt = "read" "(" IDENTIFIER ")" | "write" "(" IDENTIFIER ")"

structstmt = stmtlist | ifstmt | whilestmt

ifstmt = "if(" condition "){" stmtlist "}" ["else" "{" stmtlist "}"]

whilestmt = "while(" condition "){" stmtlist "}"

expression = expression "+" term | expression "-" term | term

term = term "*" factor | term "/" factor | term "%" factor | factor

factor = "(" expression ")" | IDENTIFIER | CONST

condition = expression RELATION expression

RELATION = "<" | "<=" | "==" | "!=" | ">=" | ">"

Token.txt

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x

y

z

[

]

{

}

(

)

,

;

:

\n

'

"

_

0

1

2

3

4

5

6

7

8

9

+

-

*

/

%

<

>

==

>=

<=

!=

=

int

string

bool

read

write

and

or

true

false

if

else

while

list

function

.append

.remove

.size

Symbol Table:

The Symbol Table is a class composed of three separate hash tables, each designed for specific data types: identifiers, integer constants, and string constants. These hash tables are represented as lists, where each position within the list can store multiple values that hash to the same position. The hash tables have a predefined size. Each element within the Symbol Table is located using a pair of indices, with the first index indicating the list in which the element is stored and the second index indicating the element's position within that list. The hash function used for determining the position of elements in the hash tables differs depending on the data type:

Operations:

addIdentifier(name: string): (int, int) - Adds an identifier to the Symbol Table and returns its position.

addIntConstant(constant: int): (int, int) - Adds an integer constant to the Symbol Table and returns its position.

addStringConstant(constant: string): (int, int) - Adds a string constant to the Symbol Table and returns its position.

hasIdentifier(name: string): boolean - Checks if the given identifier exists in the Symbol Table.

hasIntConstant(constant: int): boolean - Checks if the given integer constant exists in the Symbol Table.

hasStringConstant(constant: string): boolean - Checks if the given string constant exists in the Symbol Table.

getPositionIdentifier(name: string): (int, int) - Retrieves the position of the identifier in the Symbol Table.

getPositionIntConstant(constant: int): (int, int) - Retrieves the position of the integer constant in the Symbol Table.

getPositionStringConstant(constant: string): (int, int) - Retrieves the position of the string constant in the Symbol Table.

toString() (overridden method) - Provides a string representation of the entire Symbol Table.

Hash Table:

The Hash Table is a generic implementation used within the Symbol Table for managing the storage and retrieval of elements. It includes the following operations:

Operations:

hash(key: int): int - Computes the position in the Symbol Table for an integer constant based on the modulo operation with the size of the list.

hash(key: string): int - Computes the position in the Symbol Table for a string constant or identifier based on the sum of the ASCII codes of their characters, followed by a modulo operation with the size of the list.

getSize(): int - Returns the size of the Hash Table.

getHashValue(key: T): int - Returns the corresponding position in the Symbol Table based on the type of the provided 'key.'

add(key: T): (int, int) - Adds the 'key' to the Hash Table and returns its position if the operation is successful; otherwise, throws an exception.

contains(key: T): boolean - Checks if the given 'key' is present in the Hash Table.

getPosition(key: T): (int, int) - Retrieves the position in the Symbol Table of the given 'key' if it exists; otherwise, returns (-1, -1).

toString() (overridden method) - Provides a string representation of the Hash Table.

This structure allows for efficient management and retrieval of identifiers, integer constants, and string constants within the Symbol Table by utilizing separate hash tables for each data type.

Scanner Class:

The Scanner class is responsible for performing lexical analysis on a given program. It reads the characters from the input program, identifies the tokens, checks for reserved words, and generates a Program Internal Form (PIF) along with maintaining a Symbol Table (ST).

Fields

- _program: Stores the input program string to be scanned.
- _index: Tracks the current index within the program during scanning.
- _currentLine: Keeps track of the current line being scanned in the program.
- _tokens: List storing symbols allowed within the language's syntax rules.
- _reservedWords: List storing reserved words.
- _symbolTable: Object maintaining identifiers, string constants, and int constants.
- _PIF: List holding the Program Internal Form data.

Constructor

- `Scanner()`: Initializes the Scanner class, sets up symbol tables, PIF, and loads tokens and reserved words from the 'token.in' file.

Methods

- `SetProgram(string program)`: Sets the program content to be scanned.
- `Scan(string programFileName)`: Initiates the scanning process on the given program file.
- `ReadTokensAndReservedWords()`: Reads and categorizes symbols into tokens and reserved words.
- `SkipSpaces()`: Skips spaces and updates the current line.
- `SkipComments()`: Skips comments in the program.
- `TreatStringConstant()`: Identifies and processes string constants in the program.
- `TreatIntConstant()`: Identifies and processes integer constants in the program.
- `CheckIfIdentifierIsValid(string possibleIdentifier)`: Checks if an identifier is valid and not a reserved word.
- `TreatIdentifier()`: Identifies and processes identifiers in the program.
- `TreatFromTokenList()`: Checks and processes tokens from the token list.

- `NextToken()`: Analyzes and classifies the next token in the program.

Error Handling

- Handles and throws `ScannerException` in case of lexical errors during token analysis.

- Provides error messages specifying the line and index of the error.

Output

- Generates output files "PIF" (Program Internal Form) and "ST" (Symbol Table) in the 'outputs' directory.

Usage

1. Create a `Scanner` object.

2. Set the program content using `SetProgram()`.

3. Scan the program using `Scan(programFileName)`.

Regex

`Regex("^\"[a-zA-z0-9_ ?:*^+=.!]*\""); - checks that the variable contains only numbers, letters and the specified characters (for string)`

`Regex("^\"[^\"]"); - checks that the variable is enclosed in “ “ (for string)`

`Regex("^([+-]?[1-9][0-9]*|0)"); - checks that the variable optionally starts with + or -, that it has only digits and that the first digit does not start with 0`

`Regex("^([a-zA-Z_][a-zA-Z0-9_]*)"); - checks that the variable starts with a letter or _ and does not allow non-alphanumeric characters`

Documentation:

https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/

https://research.cs.vt.edu/AVresearch/hashing/strings.php

https://learn.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex?view=net-7.0

https://www.bytehide.com/blog/regex-csharp