НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

**Лабораторна робота №4**

з дисципліни **«**Системне програмування 2**»**

Виконав:

студент 3 курсу гр. ІО-83

Омелянський А. С.

Перевірив:

Павлов В. Г.

Київ 2020 р.

**Тема:** Вкладені конструкції. Оператори розгалуження. Тернарний оператор.

**Мета роботи**: застосування у компіляторі обробки вкладених конструкцій, ***conditional statements*** (таких як **if**) та ***conditional expressions*** (тернарних операторів типу **a ? b : c**).

**Варіант** – 20, умовний оператор: ternary

**Завдання**

1. Оновити лексер для обробки вкладених конструкцій, областей видимості та операторів розгалуження згідно варіанту.
2. Поновити парсер.
3. Удосконалити генератор коду.
4. Протестувати програму на всіх контрольних прикладах з попередніх робіт, нових прикладах, та внесіть їх результати у протокол. Програма повинна працювати для всіх коректних прикладів та видавати помилку для некоректних, що містять помилку. При обробці помилок повинно вказуватися місце в коді де сталась помилка.

**Висновок:**

В даній лабораторній роботі мною продовжено розширення функціоналу компілятору мови Python. В лексичний аналізатор додана обробка службових слів «if», «else», а в синтаксичному аналізатору розроблено клас Expr\_Ternary для реалізації обробки тернарного оператора. Результати роботи програми наведено нижче.

**Результат роботи програми:**

**1. Успішний тест для тернарного оператору**

**test1.py**

def main():

a = 5

b = 3 if a<3 else a+2

return b

**Результат виконання**

C:\prj\io8322\sysprog\lab\4-20-Java-IO-83-Omelyanskyi>PythonCompile.bat test1

[Compiler ] [INFO ] Python compiler

[Compiler ] [INFO ] Source file: test1.py

[Compiler ] [INFO ] Read 4 rows.

[Compiler ] [INFO ] Lexical analyzer starting.

[Compiler ] [INFO ] Lexical analyzer finished OK. [25 lexems]

TOKEN LIST:

-----------------------------------------------

[1,1][BLOCKINDENT] -> []

[1,1][DEF]

[1,5][ID] -> [main]

[1,9][BKTB]

[1,10][BKTE]

[1,11][COLON]

[2,1][BLOCKINDENT] -> [ ]

[2,5][ID] -> [a]

[2,7][EQUAL]

[2,9][CONSTINT] -> [5]

[3,1][BLOCKINDENT] -> [ ]

[3,5][ID] -> [b]

[3,7][EQUAL]

[3,9][CONSTINT] -> [3]

[3,11][IF]

[3,14][ID] -> [a]

[3,15][LESS]

[3,16][CONSTINT] -> [3]

[3,18][ELSE]

[3,23][ID] -> [a]

[3,24][PLUS]

[3,25][CONSTINT] -> [2]

[4,1][BLOCKINDENT] -> [ ]

[4,5][RETURN]

[4,12][ID] -> [b]

-----------------------------------------------

[Compiler ] [INFO ] Syntax analyzer starting.

[Compiler ] [INFO ] Syntax analyzer finished OK.

SYNTAX TREE:

-----------------------------------------------

[STMT\_PROGRAM]

{function}:

[STMT\_FUNCTION]

{nameFunction}:

[EXPR\_IDFUNCTION]: "main"

{bodyFunction}:

[STMT\_BLOCK]

{bodyBlock}:

[STMT\_SEQ]

{stmt1}:

[STMT\_SET]

{varName}:

[EXPR\_IDVAR]: "a"

{varValue}:

[EXPR\_CONSTINT]: "5"

{stmt2}:

[STMT\_SEQ]

{stmt1}:

[STMT\_SET]

{varName}:

[EXPR\_IDVAR]: "b"

{varValue}:

[EXPR\_TERNARY]:

{ifTrue}:

[EXPR\_CONSTINT]: "3"

{cond}:

[EXPR\_BINARYLESS]:

{expr1}:

[EXPR\_GETVARVALUE]:

[EXPR\_IDVAR]: "a"

{expr2}:

[EXPR\_CONSTINT]: "3"

{ifFalse}:

[EXPR\_BINARYPLUS]:

{expr1}:

[EXPR\_GETVARVALUE]:

[EXPR\_IDVAR]: "a"

{expr2}:

[EXPR\_CONSTINT]: "2"

{stmt2}:

[STMT\_SEQ]

{stmt1}:

[STMT\_RETURN]

{retValue}:

[EXPR\_GETVARVALUE]:

[EXPR\_IDVAR]: "b"

-----------------------------------------------

[Compiler ] [INFO ] Generate destination files

[Compiler ] [INFO ] Result file: test1.asm

[Compiler ] [INFO ] Python Compiler finished OK. [ 62 ms]

===========================================================

=== Execution test1.exe

===========================================================

7

**test1.asm**

.586

.model flat, stdcall

option casemap:none

include \masm32\include\windows.inc

include \masm32\include\kernel32.inc

include \masm32\include\masm32.inc

includelib \masm32\lib\kernel32.lib

includelib \masm32\lib\masm32.lib

NumbToStr PROTO :DWORD,:DWORD

main PROTO

.data

buff db 11 dup(?)

.code

start:

invoke main

invoke NumbToStr, ebx, ADDR buff

invoke StdOut,eax

invoke ExitProcess,0

main PROC

push ebp

mov ebp,esp

sub esp,8

L1:

push dword ptr 5

pop eax

mov [ebp-4],eax

push edx

mov eax,[ebp-4]

push eax

push dword ptr 3

pop edx

pop eax

cmp eax,edx

pop edx

jl L3

push dword ptr 0

jmp L4

L3:

push dword ptr 1

L4:

pop eax

cmp eax,0

je L5

push dword ptr 3

jmp L6

L5:

push edx

mov eax,[ebp-4]

push eax

push dword ptr 2

pop edx

pop eax

add eax,edx

pop edx

push eax

L6:

pop eax

mov [ebp-8],eax

mov eax,[ebp-8]

push eax

pop ebx

mov esp,ebp

pop ebp

ret

L2:

mov esp,ebp

pop ebp

main ENDP

NumbToStr PROC uses ebx x:DWORD,buffer:DWORD

mov ecx,buffer

mov eax,x

mov ebx,10

add ecx,ebx

LL1:

xor edx,edx

div ebx

add edx,48

mov BYTE PTR [ecx],dl

dec ecx

test eax,eax

jnz LL1

inc ecx

mov eax,ecx

ret

NumbToStr ENDP

end start

**2. Успішний тест для вкладеного тернарного оператору**

**test2.py**

def main():

a = 5

b = 3 if a<3 else 7 if a<2 else 9

return b

**Результат виконання**

C:\prj\io8322\sysprog\lab\4-20-Java-IO-83-Omelyanskyi>PythonCompile.bat test2

[Compiler ] [INFO ] Python compiler

[Compiler ] [INFO ] Source file: test2.py

[Compiler ] [INFO ] Read 4 rows.

[Compiler ] [INFO ] Lexical analyzer starting.

[Compiler ] [INFO ] Lexical analyzer finished OK. [29 lexems]

TOKEN LIST:

-----------------------------------------------

[1,1][BLOCKINDENT] -> []

[1,1][DEF]

[1,5][ID] -> [main]

[1,9][BKTB]

[1,10][BKTE]

[1,11][COLON]

[2,1][BLOCKINDENT] -> [ ]

[2,5][ID] -> [a]

[2,7][EQUAL]

[2,9][CONSTINT] -> [5]

[3,1][BLOCKINDENT] -> [ ]

[3,5][ID] -> [b]

[3,7][EQUAL]

[3,9][CONSTINT] -> [3]

[3,11][IF]

[3,14][ID] -> [a]

[3,15][LESS]

[3,16][CONSTINT] -> [3]

[3,18][ELSE]

[3,23][CONSTINT] -> [7]

[3,25][IF]

[3,28][ID] -> [a]

[3,29][LESS]

[3,30][CONSTINT] -> [2]

[3,32][ELSE]

[3,37][CONSTINT] -> [9]

[4,1][BLOCKINDENT] -> [ ]

[4,5][RETURN]

[4,12][ID] -> [b]

-----------------------------------------------

[Compiler ] [INFO ] Syntax analyzer starting.

[Compiler ] [INFO ] Syntax analyzer finished OK.

SYNTAX TREE:

-----------------------------------------------

[STMT\_PROGRAM]

{function}:

[STMT\_FUNCTION]

{nameFunction}:

[EXPR\_IDFUNCTION]: "main"

{bodyFunction}:

[STMT\_BLOCK]

{bodyBlock}:

[STMT\_SEQ]

{stmt1}:

[STMT\_SET]

{varName}:

[EXPR\_IDVAR]: "a"

{varValue}:

[EXPR\_CONSTINT]: "5"

{stmt2}:

[STMT\_SEQ]

{stmt1}:

[STMT\_SET]

{varName}:

[EXPR\_IDVAR]: "b"

{varValue}:

[EXPR\_TERNARY]:

{ifTrue}:

[EXPR\_CONSTINT]: "3"

{cond}:

[EXPR\_BINARYLESS]:

{expr1}:

[EXPR\_GETVARVALUE]:

[EXPR\_IDVAR]: "a"

{expr2}:

[EXPR\_CONSTINT]: "3"

{ifFalse}:

[EXPR\_TERNARY]:

{ifTrue}:

[EXPR\_CONSTINT]: "7"

{cond}:

[EXPR\_BINARYLESS]:

{expr1}:

[EXPR\_GETVARVALUE]:

[EXPR\_IDVAR]: "a"

{expr2}:

[EXPR\_CONSTINT]: "2"

{ifFalse}:

[EXPR\_CONSTINT]: "9"

{stmt2}:

[STMT\_SEQ]

{stmt1}:

[STMT\_RETURN]

{retValue}:

[EXPR\_GETVARVALUE]:

[EXPR\_IDVAR]: "b"

-----------------------------------------------

[Compiler ] [INFO ] Generate destination files

[Compiler ] [INFO ] Result file: test2.asm

[Compiler ] [INFO ] Python Compiler finished OK. [ 47 ms]

===========================================================

=== Execution test2.exe

===========================================================

9

**test2.asm**

.586

.model flat, stdcall

option casemap:none

include \masm32\include\windows.inc

include \masm32\include\kernel32.inc

include \masm32\include\masm32.inc

includelib \masm32\lib\kernel32.lib

includelib \masm32\lib\masm32.lib

NumbToStr PROTO :DWORD,:DWORD

main PROTO

.data

buff db 11 dup(?)

.code

start:

invoke main

invoke NumbToStr, ebx, ADDR buff

invoke StdOut,eax

invoke ExitProcess,0

main PROC

push ebp

mov ebp,esp

sub esp,8

L1:

push dword ptr 5

pop eax

mov [ebp-4],eax

push edx

mov eax,[ebp-4]

push eax

push dword ptr 3

pop edx

pop eax

cmp eax,edx

pop edx

jl L3

push dword ptr 0

jmp L4

L3:

push dword ptr 1

L4:

pop eax

cmp eax,0

je L5

push dword ptr 3

jmp L6

L5:

push edx

mov eax,[ebp-4]

push eax

push dword ptr 2

pop edx

pop eax

cmp eax,edx

pop edx

jl L7

push dword ptr 0

jmp L8

L7:

push dword ptr 1

L8:

pop eax

cmp eax,0

je L9

push dword ptr 7

jmp L10

L9:

push dword ptr 9

L10:

L6:

pop eax

mov [ebp-8],eax

mov eax,[ebp-8]

push eax

pop ebx

mov esp,ebp

pop ebp

ret

L2:

mov esp,ebp

pop ebp

main ENDP

NumbToStr PROC uses ebx x:DWORD,buffer:DWORD

mov ecx,buffer

mov eax,x

mov ebx,10

add ecx,ebx

LL1:

xor edx,edx

div ebx

add edx,48

mov BYTE PTR [ecx],dl

dec ecx

test eax,eax

jnz LL1

inc ecx

mov eax,ecx

ret

NumbToStr ENDP

end start

**3. Помилка: відсутній else для тернарного оператору**

**test3.py**

def main():

a = 5

b = 3 if 7 5

return b

**Результат виконання**

C:\prj\io8322\sysprog\lab\4-20-Java-IO-83-Omelyanskyi>PythonCompile.bat test3

[Compiler ] [INFO ] Python compiler

[Compiler ] [INFO ] Source file: test3.py

[Compiler ] [INFO ] Read 4 rows.

[Compiler ] [INFO ] Lexical analyzer starting.

[Compiler ] [INFO ] Lexical analyzer finished OK. [20 lexems]

TOKEN LIST:

-----------------------------------------------

[1,1][BLOCKINDENT] -> []

[1,1][DEF]

[1,5][ID] -> [main]

[1,9][BKTB]

[1,10][BKTE]

[1,11][COLON]

[2,1][BLOCKINDENT] -> [ ]

[2,5][ID] -> [a]

[2,7][EQUAL]

[2,9][CONSTINT] -> [5]

[3,1][BLOCKINDENT] -> [ ]

[3,5][ID] -> [b]

[3,7][EQUAL]

[3,9][CONSTINT] -> [3]

[3,11][IF]

[3,14][CONSTINT] -> [7]

[3,16][CONSTINT] -> [5]

[4,1][BLOCKINDENT] -> [ ]

[4,5][RETURN]

[4,12][ID] -> [b]

-----------------------------------------------

[Compiler ] [INFO ] Syntax analyzer starting.

SYNTAX TREE:

-----------------------------------------------

[STMT\_PROGRAM]

-----------------------------------------------

[Syntax ] [ERROR ] test3.py[3,16]->[ b = 3 if 7 5 ]

ERR: Token not "else".

[Compiler ] [ERROR ] Python Compiler finished ERROR. [ 47 ms]

**4. Помилка: відсутній вираз для істиного значення для тернарного оператору**

**test4.py**

def main():

a = 5

b = if 7 5

return b

**Результат виконання**

C:\prj\io8322\sysprog\lab\4-20-Java-IO-83-Omelyanskyi>PythonCompile.bat test4

[Compiler ] [INFO ] Python compiler

[Compiler ] [INFO ] Source file: test4.py

[Compiler ] [INFO ] Read 4 rows.

[Compiler ] [INFO ] Lexical analyzer starting.

[Compiler ] [INFO ] Lexical analyzer finished OK. [19 lexems]

TOKEN LIST:

-----------------------------------------------

[1,1][BLOCKINDENT] -> []

[1,1][DEF]

[1,5][ID] -> [main]

[1,9][BKTB]

[1,10][BKTE]

[1,11][COLON]

[2,1][BLOCKINDENT] -> [ ]

[2,5][ID] -> [a]

[2,7][EQUAL]

[2,9][CONSTINT] -> [5]

[3,1][BLOCKINDENT] -> [ ]

[3,5][ID] -> [b]

[3,7][EQUAL]

[3,9][IF]

[3,12][CONSTINT] -> [7]

[3,14][CONSTINT] -> [5]

[4,1][BLOCKINDENT] -> [ ]

[4,5][RETURN]

[4,12][ID] -> [b]

-----------------------------------------------

[Compiler ] [INFO ] Syntax analyzer starting.

SYNTAX TREE:

-----------------------------------------------

[STMT\_PROGRAM]

-----------------------------------------------

[Syntax ] [ERROR ] test4.py[3,9]->[ b = if 7 5 ]

ERR: Token not expression.

[Compiler ] [ERROR ] Python Compiler finished ERROR. [ 32 ms]

**Лістинг програми:**

**CompileException.java**

package edu.kpi.io8322.sysprog.lab.core;

import edu.kpi.io8322.sysprog.lab.PythonCompiler;

import edu.kpi.io8322.sysprog.lab.lexical.TokenInvalid;

import lombok.Getter;

@Getter

public class CompileException extends Exception {

private int row;

private int col;

public CompileException(int row, int col, String message, Throwable cause) {

super(message, cause);

this.row = row;

this.col = col;

}

public CompileException(TokenInvalid token) {

super(token.getErrMsg(), null);

this.row = token.getRow();

this.col = token.getCol();

}

@Override

public String toString() {

return PythonCompiler.app.getSrcname() +

"[" + row + "," + col + "]->[" + PythonCompiler.app.getSrclines().get(row - 1) + "]" + System.lineSeparator() + String.format("%1$50s %2$s", "ERR: ", getMessage());

}

}

**LexFabric.java**

package edu.kpi.io8322.sysprog.lab.lexical;

import java.util.HashMap;

import java.util.Map;

public class LexFabric {

private Map<LexTypeEnum, LexType> lexTypeMap;

private Map<Character, LexType\_symb> lexTypeSymbMap;

private Map<String, LexType\_keyword> lexTypeKeywordMap;

public LexFabric(LexicalAnalyzer lexicalAnalyzer) {

lexTypeMap = new HashMap<>();

lexTypeMap.put(LexTypeEnum.NONE, new LexType\_none());

lexTypeMap.put(LexTypeEnum.BLOCKINDENT, new LexType\_blockindent());

lexTypeMap.put(LexTypeEnum.BKTB, new LexType\_bktb());

lexTypeMap.put(LexTypeEnum.BKTE, new LexType\_bkte());

lexTypeMap.put(LexTypeEnum.COLON, new LexType\_colon());

lexTypeMap.put(LexTypeEnum.QUOTE1, new LexType\_quote1());

lexTypeMap.put(LexTypeEnum.QUOTE2, new LexType\_quote2());

lexTypeMap.put(LexTypeEnum.DEF, new LexType\_def());

lexTypeMap.put(LexTypeEnum.RETURN, new LexType\_return());

lexTypeMap.put(LexTypeEnum.CONSTINT, new LexType\_constint());

lexTypeMap.put(LexTypeEnum.CONSTCHAR, new LexType\_constchar());

lexTypeMap.put(LexTypeEnum.ID, new LexType\_id());

lexTypeMap.put(LexTypeEnum.NOT, new LexType\_not());

lexTypeMap.put(LexTypeEnum.PLUS, new LexType\_plus());

lexTypeMap.put(LexTypeEnum.MINUS, new LexType\_minus());

lexTypeMap.put(LexTypeEnum.EQUAL, new LexType\_equal());

lexTypeMap.put(LexTypeEnum.LESS, new LexType\_less());

lexTypeMap.put(LexTypeEnum.CONSTSTR, new LexType\_conststr());

lexTypeMap.put(LexTypeEnum.IF, new LexType\_if());

lexTypeMap.put(LexTypeEnum.ELSE, new LexType\_else());

lexTypeSymbMap = new HashMap<>();

lexTypeSymbMap.put(Character.valueOf('('), (LexType\_symb) getLexType(LexTypeEnum.BKTB));

lexTypeSymbMap.put(Character.valueOf(')'), (LexType\_symb) getLexType(LexTypeEnum.BKTE));

lexTypeSymbMap.put(Character.valueOf(':'), (LexType\_symb) getLexType(LexTypeEnum.COLON));

lexTypeSymbMap.put(Character.valueOf('\''), (LexType\_symb) getLexType(LexTypeEnum.QUOTE1));

lexTypeSymbMap.put(Character.valueOf('\"'), (LexType\_symb) getLexType(LexTypeEnum.QUOTE2));

lexTypeSymbMap.put(Character.valueOf('+'), (LexType\_symb) getLexType(LexTypeEnum.PLUS));

lexTypeSymbMap.put(Character.valueOf('-'), (LexType\_symb) getLexType(LexTypeEnum.MINUS));

lexTypeSymbMap.put(Character.valueOf('='), (LexType\_symb) getLexType(LexTypeEnum.EQUAL));

lexTypeSymbMap.put(Character.valueOf('<'), (LexType\_symb) getLexType(LexTypeEnum.LESS));

lexTypeKeywordMap = new HashMap<>();

lexTypeKeywordMap.put("def", (LexType\_keyword) getLexType(LexTypeEnum.DEF));

lexTypeKeywordMap.put("return", (LexType\_keyword) getLexType(LexTypeEnum.RETURN));

lexTypeKeywordMap.put("not", (LexType\_keyword) getLexType(LexTypeEnum.NOT));

lexTypeKeywordMap.put("if", (LexType\_keyword) getLexType(LexTypeEnum.IF));

lexTypeKeywordMap.put("else", (LexType\_keyword) getLexType(LexTypeEnum.ELSE));

}

public LexType getLexType(LexTypeEnum type) {

return lexTypeMap.get(type);

}

public LexType\_symb getLexType\_symb(char value) {

return lexTypeSymbMap.get(Character.valueOf(value));

}

public LexType\_keyword getLexType\_keyword(String value) {

return lexTypeKeywordMap.get(value);

}

}

**LexicalAnalyzer.java**

package edu.kpi.io8322.sysprog.lab.lexical;

import edu.kpi.io8322.sysprog.lab.PythonCompiler;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

import lombok.Setter;

import java.util.ArrayList;

import java.util.List;

@Getter

@Setter

public class LexicalAnalyzer {

private List<String> srclines;

private List<Token> tokenList;

private LexFabric lexFabric;

public LexicalAnalyzer(List<String> srclines) {

this.srclines = srclines;

this.lexFabric = new LexFabric(this);

}

public void exec() throws CompileException {

PythonCompiler.app.logInfo(null, null, "Lexical analyzer starting.");

tokenList = new ArrayList<>();

for (int row = 0; row < srclines.size(); row++) {

String line = srclines.get(row);

int block\_indent = 0;

while (block\_indent < line.length() && line.charAt(block\_indent) == ' ') {

block\_indent++;

}

if (block\_indent > line.length()) continue;

tokenList.add(new Token(lexFabric.getLexType(LexTypeEnum.BLOCKINDENT), row + 1, 1, line.substring(0, block\_indent)));

int col = block\_indent;

while (col < line.length()) {

if (line.charAt(col) == ' ') {

col++;

continue;

}

LexType lexType = lexFabric.getLexType\_symb(line.charAt(col));

if (lexType != null) {

Token token = new Token(lexType, row + 1, col + 1, null);

tokenList.add(token);

col++;

if (lexType.getType() == LexTypeEnum.QUOTE1 || lexType.getType() == LexTypeEnum.QUOTE2) {

int pos = -1;

switch(lexType.getType()){

case QUOTE1:

pos = line.indexOf('\'', col);

break;

case QUOTE2:

pos = line.indexOf('\"', col);

break;

}

if(pos<0){

TokenInvalid tokenInvalid = new TokenInvalid(lexFabric, row + 1, col + 1, null, "Not a closed quote.");

tokenList.add(tokenInvalid);

tokenInvalid.throwCompileException();

} else if(pos==col+1){

tokenList.add(new Token(lexFabric.getLexType(LexTypeEnum.CONSTCHAR), row + 1, col + 1, line.substring(col, pos)));

} else {

tokenList.add(new Token(lexFabric.getLexType(LexTypeEnum.CONSTSTR), row + 1, col + 1, line.substring(col, pos)));

}

tokenList.add(new Token(lexType, row+1, pos+1, null));

col = pos+1;

}

continue;

}

int next = searchToken(line, col);

if (col == next) {

TokenInvalid tokenInvalid = new TokenInvalid(lexFabric, row + 1, col + 1, line.substring(col, col + 1), "Bad symbol.");

tokenList.add(tokenInvalid);

tokenInvalid.throwCompileException();

}

String tokenValue = line.substring(col, next);

lexType = lexFabric.getLexType\_keyword(tokenValue);

if (lexType != null) {

tokenList.add(new Token(lexType, row + 1, col + 1, null));

col = next;

continue;

}

if (checkConstInt(tokenValue)) {

tokenList.add(new Token(lexFabric.getLexType(LexTypeEnum.CONSTINT), row + 1, col + 1, tokenValue));

col = next;

continue;

}

if (checkId(tokenValue)) {

tokenList.add(new Token(lexFabric.getLexType(LexTypeEnum.ID), row + 1, col + 1, tokenValue));

col = next;

continue;

}

TokenInvalid tokenInvalid = new TokenInvalid(lexFabric, row + 1, col + 1, tokenValue, "Bad identifier.");

tokenList.add(tokenInvalid);

tokenInvalid.throwCompileException();

}

}

PythonCompiler.app.logInfo(null, null, "Lexical analyzer finished OK. [" + tokenList.size() + " lexems]");

}

private int searchToken(String line, int start) {

int next = start;

while (next < line.length()) {

if ((line.charAt(next) >= 'A' && line.charAt(next) <= 'Z') ||

(line.charAt(next) >= 'a' && line.charAt(next) <= 'z') ||

(line.charAt(next) >= '0' && line.charAt(next) <= '9') ||

line.charAt(next) == '\_') {

next++;

} else {

return next;

}

}

return next;

}

private boolean checkConstInt(String value) {

if (value.length() > 10) return false;

if (value.startsWith("0x")) {

if(value.length()==2) return false;

for (int i = 2; i < value.length(); i++) {

if (!((value.charAt(i) >= '0' && value.charAt(i) <= '9') ||

(value.charAt(i) >= 'a' && value.charAt(i) <= 'f') ||

(value.charAt(i) >= 'A' && value.charAt(i) <= 'F'))) return false;

}

if (Long.parseLong(value.substring(2), 16) > Integer.MAX\_VALUE) return false;

} else {

for (int i = 0; i < value.length(); i++) {

if (value.charAt(i) < '0' || value.charAt(i) > '9') return false;

}

if (Long.parseLong(value, 10) > Integer.MAX\_VALUE) return false;

}

return true;

}

private boolean checkId(String value) {

if (!((value.charAt(0) >= 'A' && value.charAt(0) <= 'Z') || (value.charAt(0) >= 'a' && value.charAt(0) <= 'z')))

return false;

for (int i = 1; i < value.length(); i++) {

if (!((value.charAt(i) >= 'A' && value.charAt(i) <= 'Z') ||

(value.charAt(i) >= 'a' && value.charAt(i) <= 'z') ||

(value.charAt(i) >= '0' && value.charAt(i) <= '9') ||

value.charAt(i) == '\_'))

return false;

}

return true;

}

public void printTokenList(){

System.out.println("TOKEN LIST:");

System.out.println("-----------------------------------------------");

for(Token token: tokenList){

System.out.println(" "+token.toString());

}

System.out.println("-----------------------------------------------");

}

}

**LexType.java**

package edu.kpi.io8322.sysprog.lab.lexical;

import lombok.Getter;

import lombok.Setter;

@Getter

@Setter

public abstract class LexType {

private LexicalAnalyzer lexicalAnalyzer;

public abstract LexTypeEnum getType();

}

**LexType\_bktb.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_bktb extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.BKTB;

}

}

**LexType\_bkte.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_bkte extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.BKTE;

}

}

**LexType\_blockindent.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_blockindent extends LexType {

public LexTypeEnum getType(){

return LexTypeEnum.BLOCKINDENT;

}

}

**LexType\_colon.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_colon extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.COLON;

}

}

**LexType\_const.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public abstract class LexType\_const extends LexType {

}

**LexType\_constchar.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_constchar extends LexType\_const {

public LexTypeEnum getType(){

return LexTypeEnum.CONSTCHAR;

}

}

**LexType\_constint.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_constint extends LexType\_const {

public LexTypeEnum getType(){

return LexTypeEnum.CONSTINT;

}

}

**LexType\_conststr.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_conststr extends LexType\_const {

public LexTypeEnum getType(){

return LexTypeEnum.CONSTSTR;

}

}

**LexType\_def.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_def extends LexType\_keyword {

public LexTypeEnum getType(){

return LexTypeEnum.DEF;

}

}

**LexType\_else.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_else extends LexType\_keyword {

public LexTypeEnum getType(){

return LexTypeEnum.ELSE;

}

}

**LexType\_equal.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_equal extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.EQUAL;

}

}

**LexType\_id.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_id extends LexType {

public LexTypeEnum getType(){

return LexTypeEnum.ID;

}

}

**LexType\_if.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_if extends LexType\_keyword {

public LexTypeEnum getType(){

return LexTypeEnum.IF;

}

}

**LexType\_keyword.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public abstract class LexType\_keyword extends LexType {

}

**LexType\_less.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_less extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.LESS;

}

}

**LexType\_minus.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_minus extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.MINUS;

}

}

**LexType\_none.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_none extends LexType {

public LexTypeEnum getType(){

return LexTypeEnum.NONE;

}

}

**LexType\_not.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_not extends LexType\_keyword {

public LexTypeEnum getType(){

return LexTypeEnum.NOT;

}

}

**LexType\_plus.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_plus extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.PLUS;

}

}

**LexType\_quote1.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_quote1 extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.QUOTE1;

}

}

**LexType\_quote2.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_quote2 extends LexType\_symb {

public LexTypeEnum getType(){

return LexTypeEnum.QUOTE2;

}

}

**LexType\_return.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public class LexType\_return extends LexType\_keyword {

public LexTypeEnum getType(){

return LexTypeEnum.RETURN;

}

}

**LexType\_symb.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public abstract class LexType\_symb extends LexType {

}

**LexTypeEnum.java**

package edu.kpi.io8322.sysprog.lab.lexical;

public enum LexTypeEnum {

NONE, BLOCKINDENT, SYMB, BKTB, BKTE, COLON, QUOTE1, QUOTE2, KEYWORD, DEF, RETURN, CONST, CONSTINT, CONSTCHAR, ID,

NOT, PLUS,

EQUAL, MINUS, LESS, CONSTSTR,

IF, ELSE

}}

**Token.java**

package edu.kpi.io8322.sysprog.lab.lexical;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.Setter;

@Getter

@Setter

@AllArgsConstructor

public class Token {

private LexType lexType;

private int row;

private int col;

private String value;

@Override

public String toString(){

return "["+row+","+col+"]["+lexType.getType()+"]"+(value==null?"":" -> ["+value+"]");

}

public void generateCompileException(String errMsg) throws CompileException {

new TokenInvalid(this, errMsg).throwCompileException();

}

}

**TokenInvalid.java**

package edu.kpi.io8322.sysprog.lab.lexical;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

@Getter

public class TokenInvalid extends Token {

private Token real;

private String errMsg;

public TokenInvalid(Token real, String errMsg) {

super(real.getLexType(), real.getRow(), real.getCol(), real.getValue());

this.real = real;

this.errMsg = errMsg;

}

public TokenInvalid(LexFabric lexFabric, int row, int col, String value, String errMsg) {

super(lexFabric.getLexType(LexTypeEnum.NONE), row, col, value);

this.errMsg = errMsg;

}

public void throwCompileException() throws CompileException {

throw new CompileException(this);

}

@Override

public String toString(){

return "["+getRow()+","+getCol()+"]["+getLexType().getType()+".INVALID]"+(getValue()==null?"":" -> ["+getValue()+"]")+(errMsg==null?"":" // "+errMsg);

}

}

**Env.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

public class Env {

private static int SEQ\_ID = 1;

private List<Expr\_IdVar> varList;

protected int memOffsetPos;

protected Env prev;

private int id;

private int level;

public Env(Env prev) {

this.id = SEQ\_ID;

SEQ\_ID++;

memOffsetPos = 0;

this.prev = prev;

if (prev == null) {

level = 1;

} else {

level = prev.getLevel() + 1;

}

varList = new ArrayList<>();

}

public Env getPrev() {

return prev;

}

public int getId() {

return id;

}

public int getLevel() {

return level;

}

public int calcSubLevel(Env env) throws CompileException {

if (env.getId() == id) return 0;

if (prev == null) throw new CompileException(1, 0, "Error calc sublevel env.", null);

return prev.calcSubLevel(env) + 1;

}

public void putVar(Expr\_IdVar node) {

varList.add(node);

memOffsetPos += 4;

node.setMemOffset(memOffsetPos);

}

public Expr\_IdVar getVar(String key) {

for (Expr\_IdVar node : varList) {

if (node.getName().equals(key)) return node;

}

if (prev != null) return prev.getVar(key);

return null;

}

public int getMemBlockSize() {

return memOffsetPos;

}

public void genAllocMem(SyntaxAnalyzer prg) throws CompileException, IOException {

if (getMemBlockSize() > 0) {

prg.outWriteln("\tpush ebp");

prg.outWriteln("\tmov ebp,esp");

prg.outWriteln("\tsub esp," + getMemBlockSize());

}

}

public void genFreeMem(SyntaxAnalyzer prg, Env envCur) throws CompileException, IOException {

if (getMemBlockSize() > 0) {

prg.outWriteln("\tmov esp,ebp");

prg.outWriteln("\tpop ebp");

}

}

}

**Expr.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import java.io.IOException;

public abstract class Expr extends Node {

public Expr(int row, int col) {

super(row, col);

}

public void outPushValue(SyntaxAnalyzer prg) throws CompileException, IOException {

throw new CompileException(getRow(), getCol(), "outPushValue not support.", null);

}

}

**Expr\_Binary.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import lombok.Getter;

@Getter

public abstract class Expr\_Binary extends Expr {

private Expr expr1;

private Expr expr2;

public Expr\_Binary(int row, int col, Expr expr1, Expr expr2){

super(row, col);

this.expr1 = expr1;

this.expr2 = expr2;

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]: "+System.lineSeparator());

buf.append(indent+" {expr1}:"+System.lineSeparator());

getExpr1().printTree(buf, indent+" ");

buf.append(indent+" {expr2}:"+System.lineSeparator());

getExpr2().printTree(buf, indent+" ");

}

}

**Expr\_BinaryLess.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import java.io.IOException;

public class Expr\_BinaryLess extends Expr\_Binary {

public Expr\_BinaryLess(int row, int col, Expr expr1, Expr expr2){

super(row, col, expr1, expr2);

}

public NodeType getType(){

return NodeType.EXPR\_BINARYLESS;

}

@Override

public void outPushValue(SyntaxAnalyzer prg) throws CompileException, IOException {

prg.outWriteln("\tpush edx");

getExpr1().outPushValue(prg);

getExpr2().outPushValue(prg);

prg.outWriteln("\tpop edx");

prg.outWriteln("\tpop eax");

int labelTrue = prg.newLabel();

prg.outWriteln("\tcmp eax,edx");

prg.outWriteln("\tpop edx");

prg.outWriteln("\tjl " + prg.strLabel(labelTrue));

int labelAfter = prg.newLabel();

prg.outWriteln("\tpush dword ptr 0");

prg.outWriteln("\tjmp " + prg.strLabel(labelAfter));

prg.outWriteLabel(labelTrue);

prg.outWriteln("\tpush dword ptr 1");

prg.outWriteLabel(labelAfter);

}

}

**Expr\_BinaryMinus.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import java.io.IOException;

public class Expr\_BinaryMinus extends Expr\_Binary {

public Expr\_BinaryMinus(int row, int col, Expr expr1, Expr expr2){

super(row, col, expr1, expr2);

}

public NodeType getType(){

return NodeType.EXPR\_BINARYMINUS;

}

@Override

public void outPushValue(SyntaxAnalyzer prg) throws CompileException, IOException {

prg.outWriteln("\tpush edx");

getExpr1().outPushValue(prg);

getExpr2().outPushValue(prg);

prg.outWriteln("\tpop edx");

prg.outWriteln("\tpop eax");

prg.outWriteln("\tsub eax,edx");

prg.outWriteln("\tpop edx");

prg.outWriteln("\tpush eax");;

}

}

**Expr\_BinaryPlus.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import java.io.IOException;

public class Expr\_BinaryPlus extends Expr\_Binary {

public Expr\_BinaryPlus(int row, int col, Expr expr1, Expr expr2){

super(row, col, expr1, expr2);

}

public NodeType getType(){

return NodeType.EXPR\_BINARYPLUS;

}

@Override

public void outPushValue(SyntaxAnalyzer prg) throws CompileException, IOException {

prg.outWriteln("\tpush edx");

getExpr1().outPushValue(prg);

getExpr2().outPushValue(prg);

prg.outWriteln("\tpop edx");

prg.outWriteln("\tpop eax");

prg.outWriteln("\tadd eax,edx");

prg.outWriteln("\tpop edx");

prg.outWriteln("\tpush eax");;

}

}

**Expr\_Const.java**

package edu.kpi.io8322.sysprog.lab.syntax;

public abstract class Expr\_Const extends Expr {

public Expr\_Const(int row, int col){

super(row, col);

}

}

**Expr\_ConstInt.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import edu.kpi.io8322.sysprog.lab.lexical.Token;

import lombok.Getter;

import java.io.IOException;

@Getter

public class Expr\_ConstInt extends Expr\_Const {

private int value;

public Expr\_ConstInt(int row, int col, Token token) throws CompileException {

super(row, col);

switch (token.getLexType().getType()) {

case CONSTINT:

case CONSTSTR:

try {

if (token.getValue().startsWith("0x")) {

value = Integer.parseInt(token.getValue().substring(2), 16);

} else {

value = Integer.parseInt(token.getValue(), 10);

}

} catch (Throwable e) {

token.generateCompileException("Bad format number");

}

break;

case CONSTCHAR:

value = token.getValue().charAt(0);

break;

default:

token.generateCompileException("Not integer constant.");

}

}

public NodeType getType(){

return NodeType.EXPR\_CONSTINT;

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]: \""+value+"\""+System.lineSeparator());

}

@Override

public void outPushValue(SyntaxAnalyzer prg) throws CompileException, IOException {

prg.outWriteln("\tpush dword ptr "+value);

}

}

**Expr\_GetVarValue.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import java.io.IOException;

public class Expr\_GetVarValue extends Expr {

private Expr\_IdVar varName;

public Expr\_GetVarValue(int row, int col, Expr\_IdVar varName) {

super(row, col);

this.varName = varName;

}

public NodeType getType() {

return NodeType.EXPR\_GETVARVALUE;

}

public void printTree(StringBuilder buf, String indent) {

buf.append(indent + "[" + getType() + "]: " + System.lineSeparator());

varName.printTree(buf, indent + " ");

}

@Override

public void outPushValue(SyntaxAnalyzer prg) throws CompileException, IOException {

prg.outWriteln("\tmov eax,[ebp-" + varName.getMemOffset() + "]");

prg.outWriteln("\tpush eax");

}

}

**Expr\_Id.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import edu.kpi.io8322.sysprog.lab.lexical.LexTypeEnum;

import edu.kpi.io8322.sysprog.lab.lexical.Token;

import lombok.Getter;

@Getter

public abstract class Expr\_Id extends Expr {

private String name;

public Expr\_Id(Token token) throws CompileException {

super(token.getRow(), token.getCol());

if(token.getLexType().getType()!=LexTypeEnum.ID) token.generateCompileException("Token not identifier.");

this.name = token.getValue();

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]: \""+name+"\""+System.lineSeparator());

}

}

**Expr\_IdFunction.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import edu.kpi.io8322.sysprog.lab.lexical.Token;

public class Expr\_IdFunction extends Expr\_Id {

public Expr\_IdFunction(Token token) throws CompileException {

super(token);

}

public NodeType getType(){

return NodeType.EXPR\_IDFUNCTION;

}

}

**Expr\_IdVar.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import edu.kpi.io8322.sysprog.lab.lexical.Token;

public class Expr\_IdVar extends Expr\_Id {

public Expr\_IdVar(Token token) throws CompileException {

super(token);

}

public NodeType getType(){

return NodeType.EXPR\_IDVAR;

}

**Expr\_Ternary.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import java.io.IOException;

public class Expr\_Ternary extends Expr {

private Expr ifTrue;

private Expr cond;

private Expr ifFalse;

public Expr\_Ternary(int row, int col, Expr ifTrue, Expr cond, Expr ifFalse) {

super(row, col);

this.ifTrue = ifTrue;

this.cond = cond;

this.ifFalse = ifFalse;

}

public NodeType getType() {

return NodeType.EXPR\_TERNARY;

}

public void printTree(StringBuilder buf, String indent) {

buf.append(indent + "[" + getType() + "]: " + System.lineSeparator());

buf.append(indent + " {ifTrue}:" + System.lineSeparator());

ifTrue.printTree(buf, indent + " ");

buf.append(indent + " {cond}:" + System.lineSeparator());

cond.printTree(buf, indent + " ");

buf.append(indent + " {ifFalse}:" + System.lineSeparator());

ifFalse.printTree(buf, indent + " ");

}

@Override

public void outPushValue(SyntaxAnalyzer prg) throws CompileException, IOException {

cond.outPushValue(prg);

prg.outWriteln("\tpop eax");

int labelFalse = prg.newLabel();

prg.outWriteln("\tcmp eax,0");

prg.outWriteln("\tje " + prg.strLabel(labelFalse));

ifTrue.outPushValue(prg);

int labelAfter = prg.newLabel();

prg.outWriteln("\tjmp " + prg.strLabel(labelAfter));

prg.outWriteLabel(labelFalse);

ifFalse.outPushValue(prg);

prg.outWriteLabel(labelAfter);

}

}

**Expr\_Unary.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import lombok.Getter;

@Getter

public abstract class Expr\_Unary extends Expr {

private Expr expr;

public Expr\_Unary(int row, int col, Expr expr){

super(row, col);

this.expr = expr;

}

}

**Expr\_UnaryNot.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import java.io.IOException;

public class Expr\_UnaryNot extends Expr\_Unary {

public Expr\_UnaryNot(int row, int col, Expr expr){

super(row, col, expr);

}

public NodeType getType(){

return NodeType.EXPR\_UNARYNOT;

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]: "+System.lineSeparator());

getExpr().printTree(buf, indent+" ");

}

@Override

public void outPushValue(SyntaxAnalyzer prg) throws CompileException, IOException {

getExpr().outPushValue(prg);

prg.outWriteln("\tpop eax");

int labelFalse = prg.newLabel();

prg.outWriteln("\tcmp eax,0");

prg.outWriteln("\tje "+prg.strLabel(labelFalse));

int labelAfter = prg.newLabel();

prg.outWriteln("\tpush dword ptr 0");

prg.outWriteln("\tjmp "+prg.strLabel(labelAfter));

prg.outWriteLabel(labelFalse);

prg.outWriteln("\tpush dword ptr 1");

prg.outWriteLabel(labelAfter);

}

}

}

**Node.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import lombok.Getter;

@Getter

public abstract class Node {

private int row;

private int col;

public Node(int row, int col) {

this.row = row;

this.col = col;

}

public abstract NodeType getType();

public abstract void printTree(StringBuilder buf, String indent);

}

**NodeType.java**

package edu.kpi.io8322.sysprog.lab.syntax;

public enum NodeType {

NONE, EXPR, EXPR\_CONST, EXPR\_CONSTINT, EXPR\_ID, EXPR\_IDFUNCTION, EXPR\_IDVAR,

STMT, STMT\_RETURN, STMT\_BLOCK, STMT\_FUNCTION, STMT\_PROGRAM,

EXPR\_UNARY, EXPR\_BINARY, EXPR\_UNARYNOT, EXPR\_BINARYPLUS,

EXPR\_BINARYMINUS, EXPR\_BINARYLESS, STMT\_SET, STMT\_SEQ, EXPR\_GETVARVALUE,

EXPR\_TERNARY

}

**Stmt.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import java.io.IOException;

public abstract class Stmt extends Node {

public Stmt(int row, int col) {

super(row, col);

}

public void gen(SyntaxAnalyzer prg, int labelBegin, int labelAfter) throws CompileException, IOException {

}

}

**Stmt\_Block.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

import lombok.Setter;

import java.io.IOException;

@Getter

@Setter

public class Stmt\_Block extends Stmt {

private Env env;

private Stmt\_Seq body;

private String blockIndent;

public Stmt\_Block(int row, int col, Env env, String blockIndent) {

super(row, col);

this.env = env;

this.blockIndent = blockIndent;

}

public NodeType getType(){

return NodeType.STMT\_BLOCK;

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]"+System.lineSeparator());

if(body!=null){

buf.append(indent+" {bodyBlock}:"+System.lineSeparator());

body.printTree(buf, " "+indent);

}

}

@Override

public void gen(SyntaxAnalyzer prg, int labelBegin, int labelAfter) throws CompileException, IOException {

if(body!=null){

env.genAllocMem(prg);

int labelBlockBegin = prg.newLabel();

int labelBlockAfter = prg.newLabel();

prg.outWriteLabel(labelBlockBegin);

body.gen(prg, labelBlockBegin, labelBlockAfter);

prg.outWriteLabel(labelBlockAfter);

env.genFreeMem(prg, env);

}

}

}

**Stmt\_Function.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

import lombok.Setter;

import java.io.IOException;

@Getter

@Setter

public class Stmt\_Function extends Stmt {

private Expr\_IdFunction name;

private Stmt\_Block body;

public Stmt\_Function(int row, int col, Expr\_IdFunction name) {

super(row, col);

this.name = name;

}

public NodeType getType() {

return NodeType.STMT\_FUNCTION;

}

public void printTree(StringBuilder buf, String indent) {

buf.append(indent + "[" + getType() + "]" + System.lineSeparator());

buf.append(indent + " {nameFunction}:" + System.lineSeparator());

name.printTree(buf, " " + indent);

if (body != null) {

buf.append(indent + " {bodyFunction}:" + System.lineSeparator());

body.printTree(buf, " " + indent);

}

}

@Override

public void gen(SyntaxAnalyzer prg, int labelBegin, int labelAfter) throws CompileException, IOException {

prg.outWriteln(name.getName()+" PROC");

if(body!=null){

body.gen(prg, 0, 0);

}

prg.outWriteln(name.getName()+" ENDP");

}

}

**Stmt\_Program.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

import lombok.Setter;

import java.io.IOException;

@Getter

@Setter

public class Stmt\_Program extends Stmt {

private Stmt\_Function stmtFunction;

public Stmt\_Program() {

super(1, 1);

}

public NodeType getType(){

return NodeType.STMT\_PROGRAM;

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]"+System.lineSeparator());

if(stmtFunction!=null){

buf.append(indent+" {function}:"+System.lineSeparator());

stmtFunction.printTree(buf, " "+indent);

}

}

@Override

public void gen(SyntaxAnalyzer prg, int labelBegin, int labelAfter) throws CompileException, IOException {

prg.outWriteln(".586");

prg.outWriteln(".model flat, stdcall");

prg.outWriteln("option casemap:none");

prg.outWriteln("include \\masm32\\include\\windows.inc");

prg.outWriteln("include \\masm32\\include\\kernel32.inc");

prg.outWriteln("include \\masm32\\include\\masm32.inc");

prg.outWriteln("includelib \\masm32\\lib\\kernel32.lib");

prg.outWriteln("includelib \\masm32\\lib\\masm32.lib");

prg.outWriteln("NumbToStr PROTO :DWORD,:DWORD");

prg.outWriteln(stmtFunction.getName().getName()+" PROTO");

prg.outWriteln(".data");

prg.outWriteln("\tbuff db 11 dup(?)");

prg.outWriteln(".code");

prg.outWriteln("start:");

prg.outWriteln("\tinvoke main");

prg.outWriteln("\tinvoke NumbToStr, ebx, ADDR buff");

prg.outWriteln("\tinvoke StdOut,eax");

prg.outWriteln("\tinvoke ExitProcess,0");

stmtFunction.gen(prg, 0, 0);

prg.outWriteln("NumbToStr PROC uses ebx x:DWORD,buffer:DWORD");

prg.outWriteln("\tmov ecx,buffer");

prg.outWriteln("\tmov eax,x");

prg.outWriteln("\tmov ebx,10");

prg.outWriteln("\tadd ecx,ebx");

prg.outWriteln("LL1:");

prg.outWriteln("\txor edx,edx");

prg.outWriteln("\tdiv ebx");

prg.outWriteln("\tadd edx,48");

prg.outWriteln("\tmov BYTE PTR [ecx],dl");

prg.outWriteln("\tdec ecx");

prg.outWriteln("\ttest eax,eax");

prg.outWriteln("\tjnz LL1");

prg.outWriteln("\tinc ecx");

prg.outWriteln("\tmov eax,ecx");

prg.outWriteln("\tret");

prg.outWriteln("NumbToStr ENDP");

prg.outWriteln("end start");

}

}

**Stmt\_Return.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

import java.io.IOException;

@Getter

public class Stmt\_Return extends Stmt {

private Stmt\_Function stmtFunction;

private Expr retValue;

public Stmt\_Return(int row, int col, Stmt\_Function stmtFunction, Expr retValue) {

super(row, col);

this.stmtFunction = stmtFunction;

this.retValue = retValue;

}

public NodeType getType(){

return NodeType.STMT\_RETURN;

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]"+System.lineSeparator());

buf.append(indent+" {retValue}:"+System.lineSeparator());

retValue.printTree(buf, " "+indent);

}

@Override

public void gen(SyntaxAnalyzer prg, int labelBegin, int labelAfter) throws CompileException, IOException {

retValue.outPushValue(prg);

prg.outWriteln("\tpop ebx");

stmtFunction.getBody().getEnv().genFreeMem(prg, stmtFunction.getBody().getEnv());

prg.outWriteln("\tret");

}

}

**Stmt\_Seq.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

import java.io.IOException;

@Getter

public class Stmt\_Seq extends Stmt {

private Stmt stmt1;

private Stmt stmt2;

public Stmt\_Seq(int row, int col, Stmt stmt1, Stmt stmt2) {

super(row, col);

this.stmt1 = stmt1;

this.stmt2 = stmt2;

}

public NodeType getType(){

return NodeType.STMT\_SEQ;

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]"+System.lineSeparator());

if(stmt1!=null) {

buf.append(indent + " {stmt1}:" + System.lineSeparator());

stmt1.printTree(buf, " " + indent);

}

if(stmt2!=null) {

buf.append(indent + " {stmt2}:" + System.lineSeparator());

stmt2.printTree(buf, " " + indent);

}

}

@Override

public void gen(SyntaxAnalyzer prg, int labelBegin, int labelAfter) throws CompileException, IOException {

if(stmt1!=null) stmt1.gen(prg, labelBegin, labelAfter);

if(stmt2!=null) stmt2.gen(prg, labelBegin, labelAfter);

}

}

**Stmt\_Set.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import lombok.Getter;

import java.io.IOException;

@Getter

public class Stmt\_Set extends Stmt {

private Expr\_IdVar varName;

private Expr varValue;

public Stmt\_Set(int row, int col, Expr\_IdVar varName, Expr varValue) {

super(row, col);

this.varName = varName;

this.varValue = varValue;

}

public NodeType getType(){

return NodeType.STMT\_SET;

}

public void printTree(StringBuilder buf, String indent){

buf.append(indent+"["+getType()+"]"+System.lineSeparator());

if(varName!=null) {

buf.append(indent + " {varName}:" + System.lineSeparator());

varName.printTree(buf, " " + indent);

}

buf.append(indent + " {varValue}:" + System.lineSeparator());

varValue.printTree(buf, " " + indent);

}

@Override

public void gen(SyntaxAnalyzer prg, int labelBegin, int labelAfter) throws CompileException, IOException {

varValue.outPushValue(prg);

if(varName==null){

prg.outWriteln("\tpop eax");

} else {

varName.genSave(prg);

}

}

}

**SyntaxAnalyzer.java**

package edu.kpi.io8322.sysprog.lab.syntax;

import edu.kpi.io8322.sysprog.lab.PythonCompiler;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import edu.kpi.io8322.sysprog.lab.lexical.LexTypeEnum;

import edu.kpi.io8322.sysprog.lab.lexical.Token;

import java.io.IOException;

import java.io.Writer;

import java.util.\*;

public class SyntaxAnalyzer {

private List<Token> tokenList;

private Stmt\_Program root;

private int tokenIndexCur;

private Stmt\_Function functionCur;

private Deque<Stmt\_Block> blockStack;

private int labelIndexCur;

private Writer out;

private Env env;

public SyntaxAnalyzer(List<Token> tokenList) {

this.tokenList = tokenList;

}

public Stmt\_Program getRoot(){

return root;

}

public Token tokenCur() throws CompileException {

if (tokenIndexCur < tokenList.size())

return tokenList.get(tokenIndexCur);

throw new CompileException(1, 1, "Attempt to parse past end of file.", null);

}

public Token tokenPeek(int step) {

if (tokenIndexCur + step < tokenList.size())

return tokenList.get(tokenIndexCur + step);

return null;

}

public void tokenNext() {

tokenIndexCur++;

}

public Env newEnv() {

env = new Env(env);

return env;

}

public Env restorePrevEnv() {

env = env.getPrev();

return env;

}

public void exec() throws CompileException {

PythonCompiler.app.logInfo(null, null, "Syntax analyzer starting.");

tokenIndexCur = 0;

functionCur = null;

blockStack = new ArrayDeque<>();

env = null;

root = new Stmt\_Program();

if (tokenCur().getLexType().getType() != LexTypeEnum.BLOCKINDENT)

tokenCur().generateCompileException("Token not block indent.");

tokenNext();

root.setStmtFunction(parseFunction());

if(tokenPeek(0)!=null) tokenCur().generateCompileException("Bad token");

PythonCompiler.app.logInfo(null, null, "Syntax analyzer finished OK.");

}

public Stmt\_Function parseFunction() throws CompileException {

Token tokenDef = tokenCur();

if (tokenDef.getLexType().getType() != LexTypeEnum.DEF)

tokenDef.generateCompileException("Token not keyword \"def\".");

tokenNext();

Expr\_IdFunction nameFunction = new Expr\_IdFunction(tokenCur());

Stmt\_Function stmtFunction = new Stmt\_Function(tokenDef.getRow(), tokenDef.getCol(), nameFunction);

functionCur = stmtFunction;

blockStack.clear();

tokenNext();

if(tokenCur().getLexType().getType()!=LexTypeEnum.BKTB) tokenCur().generateCompileException("Not symbol \"(\".");

tokenNext();

if(tokenCur().getLexType().getType()!=LexTypeEnum.BKTE) tokenCur().generateCompileException("Not symbol \")\".");

tokenNext();

if(tokenCur().getLexType().getType()!=LexTypeEnum.COLON) tokenCur().generateCompileException("Not symbol \":\".");

tokenNext();

stmtFunction.setBody(parseBlock());

functionCur = null;

blockStack.clear();

return stmtFunction;

}

public Stmt\_Block parseBlock() throws CompileException {

if (tokenCur().getLexType().getType() != LexTypeEnum.BLOCKINDENT)

tokenCur().generateCompileException("Token not block indent.");

Stmt\_Block stmt\_block = new Stmt\_Block(tokenCur().getRow(), tokenCur().getCol(), newEnv(), tokenCur().getValue());

blockStack.addFirst(stmt\_block);

stmt\_block.setBody(parseSeq());

blockStack.removeFirst();

return stmt\_block;

}

public Stmt\_Seq parseSeq() throws CompileException {

if(tokenPeek(0)==null) return null;

if (tokenCur().getLexType().getType() != LexTypeEnum.BLOCKINDENT)

tokenCur().generateCompileException("Token not block indent.");

if(!tokenCur().getValue().equals(blockStack.peekFirst().getBlockIndent())) return null;

tokenNext();

return new Stmt\_Seq(tokenCur().getRow(), tokenCur().getCol(), parseStmt(),parseSeq());

}

public Stmt parseStmt() throws CompileException {

if(tokenCur().getLexType().getType()==LexTypeEnum.RETURN) return parseReturn();

if(tokenCur().getLexType().getType()==LexTypeEnum.ID) return parseIdLeft();

tokenCur().generateCompileException("Token not statement.");

return null;

}

public Stmt parseReturn() throws CompileException {

Token tokenReturn = tokenCur();

if(functionCur==null) tokenReturn.generateCompileException("Return without function.");

tokenNext();

return new Stmt\_Return(tokenReturn.getRow(),tokenReturn.getCol(), functionCur, parseExpr());

}

public Stmt parseIdLeft() throws CompileException {

Token tokenId = tokenCur();

if(tokenPeek(1)==null || tokenPeek(1).getLexType().getType()!=LexTypeEnum.EQUAL){

tokenId.generateCompileException("Not found symbol \".");

}

Expr\_IdVar varName = env.getVar(tokenId.getValue());

if(varName==null){

varName = new Expr\_IdVar(tokenId);

env.putVar(varName);

}

tokenNext();

tokenNext();

return new Stmt\_Set(tokenId.getRow(), tokenId.getCol(), varName, parseExpr());

}

public Expr parseTerm() throws CompileException {

if(tokenCur().getLexType().getType()==LexTypeEnum.BKTB){

tokenNext();

Expr expr = parseExpr();

if(tokenCur().getLexType().getType()!=LexTypeEnum.BKTE) tokenCur().generateCompileException("Token not \")\".");

tokenNext();

return expr;

}

if(tokenCur().getLexType().getType()==LexTypeEnum.QUOTE1 || tokenCur().getLexType().getType()==LexTypeEnum.QUOTE2){

Token quoteBegin = tokenCur();

tokenNext();

Expr\_ConstInt expr = new Expr\_ConstInt(tokenCur().getRow(), tokenCur().getCol(), tokenCur());

tokenNext();

if(tokenCur().getLexType().getType()!=quoteBegin.getLexType().getType()) tokenCur().generateCompileException("Not close quote.");

tokenNext();

return expr;

}

if(tokenCur().getLexType().getType()==LexTypeEnum.CONSTINT){

Expr\_ConstInt expr = new Expr\_ConstInt(tokenCur().getRow(), tokenCur().getCol(), tokenCur());

tokenNext();

return expr;

}

if(tokenCur().getLexType().getType()==LexTypeEnum.NOT){

Token token\_not = tokenCur();

tokenNext();

Expr expr = new Expr\_UnaryNot(token\_not.getRow(), token\_not.getCol(), parseExpr());

return expr;

}

if(tokenCur().getLexType().getType()==LexTypeEnum.ID){

Expr\_IdVar varName = env.getVar(tokenCur().getValue());

if(varName==null) tokenCur().generateCompileException("Variable \""+tokenCur().getValue()+"\" not defined.");

Expr expr = new Expr\_GetVarValue(tokenCur().getRow(), tokenCur().getCol(), varName);

tokenNext();

return expr;

}

tokenCur().generateCompileException("Token not expression.");

throw new RuntimeException();

}

public Expr parseExpr() throws CompileException {

Expr expr1 = parseExprPrior2();

if(tokenPeek(0)!=null){

if(tokenCur().getLexType().getType()==LexTypeEnum.IF){

tokenNext();

Expr cond = parseExprPrior2();

if(tokenCur().getLexType().getType()!=LexTypeEnum.ELSE)

tokenCur().generateCompileException("Token not \"else\".");

tokenNext();

Expr expr = new Expr\_Ternary(expr1.getRow(), expr1.getCol(), expr1, cond, parseExpr());

return expr;

}

}

return expr1;

}

public Expr parseExprPrior2() throws CompileException {

Expr expr1 = parseExprPrior3();

if(tokenPeek(0)!=null){

if(tokenCur().getLexType().getType()==LexTypeEnum.LESS){

tokenNext();

Expr expr = new Expr\_BinaryLess(expr1.getRow(), expr1.getCol(), expr1, parseExprPrior2());

return expr;

}

}

return expr1;

}

public Expr parseExprPrior3() throws CompileException {

Expr expr1 = parseTerm();

if (tokenPeek(0) != null) {

Expr expr = null;

if (tokenCur().getLexType().getType() == LexTypeEnum.PLUS) {

tokenNext();

expr = new Expr\_BinaryPlus(expr1.getRow(), expr1.getCol(), expr1, parseTerm());

} else if (tokenCur().getLexType().getType() == LexTypeEnum.MINUS) {

tokenNext();

expr = new Expr\_BinaryMinus(expr1.getRow(), expr1.getCol(), expr1, parseTerm());

}

if(expr!=null){

if(tokenPeek(0)==null) return expr;

switch(tokenCur().getLexType().getType()){

case PLUS:

tokenNext();

return new Expr\_BinaryPlus(expr.getRow(), expr.getCol(), expr, parseExprPrior3());

case MINUS:

tokenNext();

return new Expr\_BinaryMinus(expr.getRow(), expr.getCol(), expr, parseExprPrior3());

default:

return expr;

}

}

}

return expr1;

}

public void printTree() {

System.out.println("SYNTAX TREE:");

System.out.println("-----------------------------------------------");

StringBuilder bufPrint = new StringBuilder();

root.printTree(bufPrint, "");

System.out.println(bufPrint);

System.out.println("-----------------------------------------------");

}

public int newLabel() {

labelIndexCur++;

return labelIndexCur;

}

public String strLabel(int num) {

if(num>0) return "L"+num;

return null;

}

public void outWriteLabel(int num) throws IOException {

if (num > 0)

out.write("L" + num + ":" + System.lineSeparator());

}

public void outWriteln(String str) throws IOException {

if (str != null && !str.isEmpty()) out.write(str);

out.write(System.lineSeparator());

}

public void execOut(Writer out) throws CompileException, IOException {

this.out = out;

labelIndexCur = 0;

root.gen(this, 0, 0);

}

}

**PythonCompiler.java**

package edu.kpi.io8322.sysprog.lab;

import edu.kpi.io8322.sysprog.lab.core.CompileException;

import edu.kpi.io8322.sysprog.lab.lexical.LexicalAnalyzer;

import edu.kpi.io8322.sysprog.lab.syntax.SyntaxAnalyzer;

import lombok.Getter;

import java.io.\*;

import java.util.ArrayList;

import java.util.List;

@Getter

public class PythonCompiler {

private static final String LOG\_FORMAT = "[%1$-10s] [%2$-7s] %3$s";

public static PythonCompiler app;

private String srcname;

private String dstname;

private List<String> srclines;

private LexicalAnalyzer lexicalAnalyzer;

private SyntaxAnalyzer syntaxAnalyzer;

public PythonCompiler(String srcname) {

this.srcname = srcname;

if (srcname.endsWith(".py")) {

dstname = srcname.substring(0, srcname.length() - ".py".length()) + ".asm";

} else {

dstname = srcname + ".asm";

}

}

public int exec() {

logInfo(null, null, "Source file: " + srcname);

srclines = new ArrayList<>();

BufferedReader br = null;

try {

br = new BufferedReader(new FileReader(srcname));

String line = br.readLine();

while (line != null) {

srclines.add(line);

line = br.readLine();

}

br.close();

logInfo(null, null, "Read " + srclines.size() + " rows.");

} catch (Throwable e) {

logError(null, null, "Error read file " + srcname);

if (br != null) {

try {

br.close();

} catch (Throwable e1) {

}

}

return 1;

}

lexicalAnalyzer = new LexicalAnalyzer(srclines);

try {

lexicalAnalyzer.exec();

lexicalAnalyzer.printTokenList();;

} catch(CompileException e){

lexicalAnalyzer.printTokenList();;

logError("Lexical", null, e.toString());

return 1;

}

syntaxAnalyzer = new SyntaxAnalyzer(lexicalAnalyzer.getTokenList());

try {

syntaxAnalyzer.exec();

syntaxAnalyzer.printTree();

} catch(CompileException e){

syntaxAnalyzer.printTree();

logError("Syntax", null, e.toString());

return 1;

}

try {

logInfo(null, null, "Generate destination files");

StringWriter stringWriter = new StringWriter();

BufferedWriter writer = new BufferedWriter(stringWriter);

syntaxAnalyzer.execOut(writer);

writer.close();

String bodyResultFile = new String(stringWriter.getBuffer());

BufferedWriter writerFile = new BufferedWriter(new FileWriter(dstname));

writerFile.write(bodyResultFile);

writerFile.close();

} catch (Throwable e){

logError("Generator", null, e.toString());

return 1;

}

logInfo(null, null, "Result file: " + dstname);

return 0;

}

public void logInfo(String sourceClass, String sourceMethod, String msg) {

if (sourceClass == null) {

System.out.println(String.format(LOG\_FORMAT, "Compiler", "INFO", msg));

} else {

System.out.println(String.format(LOG\_FORMAT, sourceClass, "INFO", msg));

}

}

public void logError(String sourceClass, String sourceMethod, String msg) {

if (sourceClass == null) {

System.out.println(String.format(LOG\_FORMAT, "Compiler", "ERROR", msg));

} else {

System.out.println(String.format(LOG\_FORMAT, sourceClass, "ERROR", msg));

}

}

public static void main(String[] argc) {

if (argc.length < 1) {

System.out.println("Usage: PythonCompiler <source\_file>");

System.out.println();

System.exit(1);

}

PythonCompiler.app = new PythonCompiler(argc[0]);

long timeexec = System.currentTimeMillis();

PythonCompiler.app.logInfo(null, null, "Python compiler");

int exitcode = PythonCompiler.app.exec();

switch (exitcode) {

case 0:

PythonCompiler.app.logInfo(null, null, "Python Compiler finished OK. [ " + (System.currentTimeMillis() - timeexec) + " ms]");

break;

default:

PythonCompiler.app.logError(null, null, "Python Compiler finished ERROR. [ " + (System.currentTimeMillis() - timeexec) + " ms]");

break;

}

System.exit(exitcode);

}}