МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра информационных систем и цифровых технологий

Работа допущена к защите

\_\_\_\_\_\_\_\_\_\_\_\_\_\_Руководитель

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

**КУРСОВОЙ ПРОЕКТ**

по дисциплине «Теория языков программирования и методы трансляции»

на тему: «Компилятор для подмножества языка Python»

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Музалевский Н.С.

Шифр 190139

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Группа 92ПГ

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Гордиенко А.П.

Оценка: «\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_» Дата \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Орел 2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра информационных систем и цифровых технологий

УТВЕРЖДАЮ:

\_\_\_\_\_\_\_\_\_\_\_\_Зав. кафедрой

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2022г.

**ЗАДАНИЕ**

**на курсовой проект**

по дисциплине «Теория языков программирования и методы трансляции»

Студент Музалевский Н.С., Шифр 190139

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Группа 92ПГ

1 Тема курсовой работы

«Компилятор для подмножества языка Python»

2 Срок сдачи студентом законченной работы «26» мая 2022

3 Исходные данные

Для подмножества языка Python сделать:

Определение переменных целого, вещественного, булевого типа; определение массивов и классов; определение функций и процедур; команды присваивания, условий, цикла, ввода-вывода, блока команд, вызова процедур и функций; использование таких операций как: обращение к элементу массива, обращение к полю класса, арифметические операции, операции сравнения, логические операции.

Для реализации использовать лексический анализатор на основе конечных автоматов, синтаксический анализатор методом рекурсивного спуска.

4 Содержание курсовой работы

Лексический анализ

Синтаксический анализ

Абстрактное синтаксическое дерево

5 Отчетный материал курсовой работы

Пояснительная записка курсового проекта, программа на съемном носителе

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Гордиенко А.П.

Задание принял к исполнению: «11» марта 2022

Подпись студента\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc104391117)

[ЛЕКСИЧЕСКИЙ АНАЛИЗ 5](#_Toc104391118)

[СИНТАКСИЧЕСКИЙ АНАЛИЗ 6](#_Toc104391119)

[АБСТРАКТНОЕ СИНТАКСИЧЕСКОЕ ДЕРЕВО 7](#_Toc104391120)

[ЗАКЛЮЧЕНИЕ 8](#_Toc104391121)

[СПИСОК ЛИТЕРАТУРЫ 9](#_Toc104391122)

[ПРИЛОЖЕНИЕ А (обязательное) РЕАЛИЗАЦИЯ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON(lexer.h) 10](#_Toc104391123)

[ПРИЛОЖЕНИЕ Б (обязательное) РЕАЛИЗАЦИЯ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON(lexer.c) 11](#_Toc104391124)

[ПРИЛОЖЕНИЕ В (обязательное) ФОРМАЛЬНАЯ ГРАММАТИКА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON 21](#_Toc104391125)

[ПРИЛОЖЕНИЕ Г (обязательное) ФРАГМЕНТЫ РЕАЛИЗАЦИИ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON(parser.h) 23](#_Toc104391126)

[ПРИЛОЖЕНИЕ Д (обязательное) ФРАГМЕНТЫ РЕАЛИЗАЦИИ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON(parser.с) 25](#_Toc104391127)

[ПРИЛОЖЕНИЕ Е (обязательное) ФРАГМЕНТЫ РЕАЛИЗАЦИИ ПОСТРОЕНИЯ АБСТРАКТНОГО СИНТАКСИЧЕСКОГО ДЕРЕВА(tree.h) 45](#_Toc104391128)

[ПРИЛОЖЕНИЕ Ё (обязательное) ФРАГМЕНТЫ РЕАЛИЗАЦИИ ПОСТРОЕНИЯ АБСТРАКТНОГО СИНТАКСИЧЕСКОГО ДЕРЕВА(tree.c) 46](#_Toc104391129)

# ВВЕДЕНИЕ

В настоящие время сферу информационных технологий невозможно представить без компиляторов. Именно они позволили достигнуть текущего уровня развития сферы информационных технологий в целом и программирования в частности. Ведь писать код на каком-то С# или Python в миллион раз быстрее и легче чем на машинном коде. Компилятор — программа, переводящая написанный на языке программирования текст в набор машинных кодов [1].

Важность данной темы для программиста заключается в том, что глубокое понимание того, как работает компилятор, и его реализация, даёт программисту более широкое понимание синтаксиса языка и почему некоторые вещи сделаны именно так, например во многих языках программирования нельзя начинать имя переменой с цифры, так как если добавить возможность начинать с цифры сложность конечного автомата вырастет в несколько порядков и в этом нет какого-то особого смысла.

Объектом нашей работы является разработка компилятора для подмножества императивного языка Python. Предметом нашей работы является процесс создания лексического и синтаксического анализаторов, как составляющих частей компилятора для подмножества императивного языка Python.

Цель нашей работы: приобрести навыки разработки компиляторов на примере реализации компилятора для подмножества императивного языка Python.

Для достижения цели нам необходимо решить соответствующие задачи:

1) описание и построение лексического анализатора;

2) описание и построение синтаксического анализатора;

3) описание и построение абстрактного синтаксического дерева.

# ЛЕКСИЧЕСКИЙ АНАЛИЗ

Ключевое\_слово не([A-Za-z] | [0-9] | \_)

([A-Za-z] | \_) ([A-Za-z0-9]|\_)\*

[0-9]\*(.[0-9]+)?

# СИНТАКСИЧЕСКИЙ АНАЛИЗ

# АБСТРАКТНОЕ СИНТАКСИЧЕСКОЕ ДЕРЕВО

# ЗАКЛЮЧЕНИЕ

В результате выполнения курсового проекта были описаны принципы построения лексического анализатора, синтаксического анализатора и абстрактного синтаксического дерева для подмножества языка Python.

В ходе данной работы был реализован лексический анализатор на основе конечных автоматов, синтаксический анализатор методом рекурсивного спуска, а также абстрактное синтаксическое дерево.

В результате выполнения работы были получены знания о принципах работы лексического анализатора и синтаксического анализатора, а также закреплены на практике в результате их реализации на языке Си. Мы узнали, как лексический анализатор разбивает исходную последовательность на последовательность лексем(токенов), далее, как синтаксический анализатор проверяет последовательность токенов на соответствие с грамматикой языка Python и строит на основе рекурсивного спуска абстрактное синтаксическое дерево.

Исходя из результатов работы можно сделать вывод, что все поставленные задачи были выполнены, и, следовательно, цель данного курсового проекта была достигнута.

# СПИСОК ЛИТЕРАТУРЫ

1. Компилятор [Электронный ресурс]. - Режим доступа: https://ru.wikipedia.org/wiki/Компилятор (дата обращения: 12.04.2022).
2. Лексический анализ [Электронный ресурс]. - Режим доступа: https://ru.wikipedia.org/wiki/Лексический\_анализ (дата обращения: 19.04.2022).
3. Синтаксический анализ [Электронный ресурс]. - Режим доступа: https://ru.wikipedia.org/wiki/Синтаксический\_анализ (дата обращения: 25.04.2022).
4. Абстрактное синтаксическое дерево [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Абстрактное\_синтаксическое\_дерево (дата обращения: 09.05.2022).
5. Метод рекурсивного спуска [Электронный ресурс]. - Режим доступа: https://ru.wikipedia.org/wiki/Метод\_рекурсивного\_спуска (дата обращения: 10.05.2022).
6. Конечный автомат [Электронный ресурс]. - Режим доступа: https://ru.wikipedia.org/wiki/Конечный\_автомат (дата обращения: 11.05.2022).
7. Python [Электронный ресурс]. - Режим доступа: https://ru.wikipedia.org/wiki/Python (дата обращения: 11.05.2022).

# ПРИЛОЖЕНИЕ А (обязательное) РЕАЛИЗАЦИЯ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON(lexer.h)

#ifndef LEXER\_H

#define LEXER\_H

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <ctype.h>

#include "token.h"

enum number {

numbers,

one\_dot\_numbers,

finish\_number,

no\_number

};

bool IsNumber(char\* str);

void FinishToken(enum TokenType type, int\* tokenQuantity);

struct Token\* Lexer(char\* content,int \* tokenQuantity);

bool CompareStrings(char\* str);

bool CompareStringCheck(char\* str);

#endif

# ПРИЛОЖЕНИЕ Б (обязательное) РЕАЛИЗАЦИЯ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON(lexer.c)

#include "lexer.h"

char\* fileContent;

int currentTokenLength = 0;

int idx = 0;

int lines = 0;

struct Token \* tokenS;

void FinishToken(enum TokenType type, int \* tokenQuantity)

{

struct Token\* temp = malloc((\*tokenQuantity + 1)\*sizeof(struct Token));

for (int i = 0; i < \*tokenQuantity; i++)

{

temp[i] = tokenS[i];

}

temp[\*tokenQuantity].type = type;

char buffer[currentTokenLength + 1];

memcpy(buffer, &fileContent[idx - currentTokenLength], currentTokenLength);

buffer[currentTokenLength] = '\0';

temp[\*tokenQuantity].value = malloc(strlen(buffer) + 1);

strcpy(temp[\*tokenQuantity].value, buffer);

temp[\*tokenQuantity].pos = idx - currentTokenLength;

temp[\*tokenQuantity].line = lines;

if (tokenS)

{

free(tokenS);

}

tokenS = temp;

(\*tokenQuantity)++;

currentTokenLength = 0;

}

struct Token \* Lexer(char\* content, int \* tokenQuantity)

{

enum TokenType state = Delimiter;

tokenS = malloc(0);

fileContent = content;

bool io\_Rooted = false;

bool identificatorFirst = true;

while (idx < strlen(fileContent))

{

switch (state)

{

case Delimiter:

if ((idx + 1 < strlen(fileContent) && fileContent[idx] == '\r'&& fileContent[idx+1] =='\n'))

{

lines++;

idx+=2;

currentTokenLength+=1;

FinishToken(Delimiter, tokenQuantity);

state = Delimiter;

}

else if ( fileContent[idx] == '\n' || fileContent[idx] == '\0')

{

lines++;

idx++;

currentTokenLength++;

FinishToken(Delimiter, tokenQuantity);

state = Delimiter;

}

else

{

state = Tab;

}

break;

case Tab:

if(CompareStrings("\t"))

{

FinishToken(Tab, tokenQuantity);

state = Delimiter;

}

else

{

state = Comma;

}

break;

case Comma:

if (CompareStrings(","))

{

FinishToken(Comma, tokenQuantity);

state = Delimiter;

}

else

{

state = Dot;

}

break;

case Dot:

if (CompareStrings("."))

{

FinishToken(Dot, tokenQuantity);

state = Delimiter;

}

else

{

state = DoubleDot;

}

break;

case DoubleDot:

if (CompareStrings(":"))

{

FinishToken(DoubleDot, tokenQuantity);

state = Delimiter;

}

else

{

state = Bool;

}

break;

case Bool:

if (CompareStringCheck("true")||CompareStringCheck("false"))

{

FinishToken(Bool, tokenQuantity);

state = Delimiter;

}

else

{

state = Logical;

}

break;

case Logical:

if (CompareStringCheck("and")||CompareStringCheck("or")||CompareStringCheck("not"))

{

FinishToken(Logical, tokenQuantity);

state = Delimiter;

}

else

{

state = OpenBraces;

}

break;

case OpenBraces:

if (CompareStrings("["))

{

FinishToken(OpenBraces, tokenQuantity);

state = Delimiter;

}

else

{

state = CloseBraces;

}

break;

case CloseBraces:

if (CompareStrings("]"))

{

FinishToken(CloseBraces, tokenQuantity);

state = Delimiter;

}

else

{

state = OpenBracket;

}

break;

case OpenBracket:

if (CompareStrings("("))

{

FinishToken(OpenBracket, tokenQuantity);

state = Delimiter;

}

else

{

state = CloseBracket;

}

break;

case CloseBracket:

if (CompareStrings(")"))

{

FinishToken(CloseBracket, tokenQuantity);

state = Delimiter;

}

else

{

state = Comparison;

}

break;

case Comparison:

if (CompareStrings("!=")||CompareStrings("==")||CompareStrings("<=")||CompareStrings(">=")||CompareStrings(">")||CompareStrings("<"))

{

FinishToken(Comparison, tokenQuantity);

state = Delimiter;

}

else

{

state = Assignment;

}

break;

case Assignment:

if (CompareStrings("=")||CompareStrings("+=")||CompareStrings("-=")||CompareStrings("\*=")||CompareStrings("/=")||CompareStrings("\%="))

{

FinishToken(Assignment, tokenQuantity);

state = Delimiter;

}

else

{

state = MathSign;

}

break;

case MathSign:

if (CompareStrings("+") || CompareStrings("-") || CompareStrings("\*") || CompareStrings("/") || CompareStrings("\%"))

{

FinishToken(MathSign, tokenQuantity);

state = Delimiter;

}

else

{

state = If;

}

break;

case If:

if (CompareStringCheck("if"))

{

FinishToken(If, tokenQuantity);

state = Delimiter;

}

else

{

state = Else;

}

break;

case Else:

if (CompareStringCheck("else"))

{

FinishToken(Else, tokenQuantity);

state = Delimiter;

}

else

{

state = For;

}

break;

case For:

if (CompareStringCheck("for"))

{

FinishToken(For, tokenQuantity);

state = Delimiter;

}

else

{

state = While;

}

break;

case While:

if (CompareStringCheck("while"))

{

FinishToken(While, tokenQuantity);

state = Delimiter;

}

else

{

state = Input;

}

break;

case Input:

if (CompareStringCheck("input"))

{

FinishToken(Input, tokenQuantity);

state = Delimiter;

}

else

{

state = Output;

}

break;

case Output:

if (CompareStringCheck("print"))

{

FinishToken(Output, tokenQuantity);

state = Delimiter;

}

else

{

state = Class;

}

break;

case Class:

if (CompareStringCheck("class"))

{

FinishToken(Class, tokenQuantity);

state = Delimiter;

}

else

{

state = Def;

}

break;

case Def:

if (CompareStringCheck("def"))

{

FinishToken(Def, tokenQuantity);

state = Delimiter;

}

else

{

state = In;

}

break;

case In:

if (CompareStringCheck("in"))

{

FinishToken(In, tokenQuantity);

state = Delimiter;

}

else

{

state = Return;

}

break;

case Return:

if (CompareStringCheck("return"))

{

FinishToken(Return, tokenQuantity);

state = Delimiter;

}

else

{

state = Variable;

}

break;

case Variable:

if (isalpha(fileContent[idx]) || CompareStrings("\_") || (!identificatorFirst && isdigit(fileContent[idx])))

{

identificatorFirst = false;

currentTokenLength++;

idx++;

}

else if (!identificatorFirst)

{

identificatorFirst = true;

FinishToken(Variable, tokenQuantity);

state = Delimiter;

}

else

{

state = Number;

}

break;

case Number:

if (IsNumber(fileContent))

{

FinishToken(Number, tokenQuantity);

state = Delimiter;

}

else

{

state = Error;

}

break;

case Error:

if(fileContent[idx]!=' ')

{

printf("\nLexical error on the line %d - unreserved character: %c ",lines+1 , fileContent[idx]);

}

idx++;

state = Delimiter;

default:

break;

}

}

FinishToken(Delimiter, tokenQuantity);

FinishToken(END, tokenQuantity);

return tokenS;

}

bool CompareStrings(char\* str)

{

if (idx + strlen(str)-1 < strlen(fileContent))

{

for (int i = 0; i < strlen(str); i++)

{

if (fileContent[idx + i] != str[i])

{

return false;

}

}

idx += strlen(str);

currentTokenLength += strlen(str);

return true;

}

return false;

}

bool CompareStringCheck(char\* str)

{

if (idx + strlen(str)-1 < strlen(fileContent))

{

for (int i = 0; i < strlen(str); i++)

{

if (fileContent[idx + i] != str[i])

{

return false;

}

}

if(idx + strlen(str) < strlen(fileContent))

{

if(isalpha(fileContent[idx + strlen(str)]) || isdigit(fileContent[idx + strlen(str)]))

{

return false;

}

}

idx += strlen(str);

currentTokenLength += strlen(str);

return true;

}

return false;

}

bool IsNumber(char\* str)

{

enum number stage = numbers;

bool RootNumber = false;

while (true)

{

switch (stage)

{

case numbers:

{

if (idx < strlen(str))

{

if (isdigit(str[idx]))

{

RootNumber = true;

currentTokenLength++;

idx++;

}

else if (str[idx] == '.' && RootNumber)

{

currentTokenLength++;

idx++;

stage = one\_dot\_numbers;

}

else if ((str[idx] != ' ' || str[idx] != '\n' || str[idx] != '\r') && RootNumber)

{

stage = finish\_number;

}

else

{

stage = no\_number;

}

}

else

{

stage = finish\_number;

}

break;

}

case one\_dot\_numbers:

{

if (idx < strlen(str) && str[idx] != ' ' && str[idx] != '\n' && str[idx] != '\r')

{

if (isdigit(str[idx]))

{

currentTokenLength++;

idx++;

}

else

{

stage = no\_number;

}

}

else

{

stage = finish\_number;

}

break;

}

case no\_number:

{

return false;

}

case finish\_number:

{

return true;

}

}

}

}

# ПРИЛОЖЕНИЕ В (обязательное) ФОРМАЛЬНАЯ ГРАММАТИКА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON

«значение» → «переменная»"."«переменная» | «переменная»“[“«арифметическое выражение»”]“ | «переменная» | «число» | «булевые значения»

«идентификатор» → «переменная»"."«переменная» | «переменная» “[“

«арифметическое выражение»”]“ | «переменная»

«знак сравнения» → “>” | “<” | “==“ | “!=” | “>=”| “<=”

«арифметический знак» → “+” | “/” | “-“ | “\*” | “%” | “\*\*” | “//”

«логический знак» → “or” | “and“

«булевые значения» → “true“ | “fasle“

«возврат» → "return" («значение» "\n" | "\n")

«операция присваивания» → «идентификатор» "=" «арифметическое выражение» "\n" | «идентификатор» "=" «команда чтения» "\n" | «идентификатор» "=" «вызов функции» "\n" | | «идентификатор» "=" «определение массива» "\n"

«определение массива» → "[" [«аргументы»] "]"

«определение класса» → “class” «переменная» “(“ “)” “:” “\n” «блок»

«определение функций» → "def" «переменная» "(" «аргументы для определения функции» ")" ":" "\n" «блок»

«аргументы для определения функции» → «переменная» [“,” | «аргумент для определения функции»] | эпсилон

«аргумент для определения функции» → «переменная» [“,” | «аргумент для определения функции»]

«вызов функции» → «переменная» "(" «аргументы» ")" "\n"

«аргументы» → «идентификатор» [“,” | «аргумент»] | | эпсилон

«аргумент» → «идентификатор» [“,” | «аргумент»]

«условный оператор if» → “if” «логическая операция» ":" "\n"

«последовательность» [“else” ":" "\n" «блок»]

«оператор цикла while» → “while” «логическая операция» ":" "\n"

«блок»

«оператор цикла for» → “for” «значение» “in” «значение» ":" "\n"

«блок»

«логическая операция» → [“(“] «логическая операция» [«логический знак» «логическая операция»] [“)“]

«логическое выражение» → [“(“] «арифметическое выражение» [«знак сравнения» «логическое выражение»] [“)“]

«арифметическое выражение» →

[“(“]«значение» [«арифметический знак» «арифметическое выражение»] [“)“]

«команда ввода» → “input” “(“ “)” “\n”

«команда вывода» → “print” “(“ «арифметическое выражение»” “)” “\n”

«обычное утверждение» → «условный оператор if» | «оператор цикла while» | «оператор цикла for» | «операция присваивания» | «команда ввода» | «команда вывода» | «определение класса» | «определение функции» | «вызов функции» | “\n”

«утверждение класса» → «операция присваивания» | «определение функции» | “\n”

«утверждение функции» → «обычное утверждение» | «возврат»

«блок» → [«таб»] «обычное утверждение» «блок» | [«таб»] «утверждение класса» «блок» | [«таб»] «утверждение функции» «блок» | эпсилон

# ПРИЛОЖЕНИЕ Г (обязательное) ФРАГМЕНТЫ РЕАЛИЗАЦИИ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON(parser.h)

#ifndef PARSER\_H

#define PARSER\_H

#include <stdbool.h>

#include "token.h"

#include "tree.h"

enum StateBlock

{

CommonBlock,

ClassBlock,

DefBlock,

};

bool Block();

bool Statement();

bool StatementDef();

bool StatementClass();

bool Condition();

bool WhileLoop();

bool ForLoop();

bool OutputCommand();

bool InputCommand(struct Node\* inputNode);

bool AssignmentOperation();

bool ArrayDefinition(struct Node\* arrayFunctionDefinition);

bool FunctionDefinition();

bool FunctionCall(struct Node\* functionCallNode);

bool Return1();

bool LogicalOperationMain(struct Node\* arifNode);

bool LogicalOperation(int\* bracketCountDifference, struct Node\* arifNode);

bool LogicalExpression(int\* bracketCountDifference, struct Node\* arifNode);

bool ArithmeticExpressionLO(int\* bracketCountDifference, struct Node\* arifNode);

bool ArithmeticExpressionMain(bool open, struct Node\* arifNode);

bool ArithmeticExpression(int\* bracketCount, struct Node\* arifNode);

bool ArgumentsFunctionDefinition(struct Node\* argumentsFunctionNode);

bool ArgumentFunctionDefinition(struct Node\* argumentFunctionNode);

bool Arguments(struct Node\* argumentsNode);

bool Argument(struct Node\* argumentNode);

bool ClassDefinition();

bool Value(struct Node\* valueNode);

bool Identificator(struct Node\* identificatorNode);

bool Is(enum TokenType type);

void Parser(struct Token\* token,int tokenQuantity);

#endif

# ПРИЛОЖЕНИЕ Д (обязательное) ФРАГМЕНТЫ РЕАЛИЗАЦИИ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА PYTHON(parser.с)

#include "parser.h"

int tempCurrentToken;

int currentToken;

int tokenLength;

int tabCount = 0;

int needTabCount = 0;

struct Token \* tokens;

enum StateBlock sequence = CommonBlock;

enum StateBlock prevBlock = CommonBlock;

struct Node root;

struct Node \* current;

void InitFirstTree()

{

enum NonTerminalType RootTerm = BlockTerm;

root = \*NewNodeTerminal(RootTerm);

current = &root;

}

void ErrorRecovery()

{

while (currentToken < tokenLength)

{

if (tokens[currentToken].type != Delimiter)

{

currentToken++;

}

else

{

currentToken++;

break;

}

}

}

void Parser(struct Token \* token, int tokenQuantity)

{

InitFirstTree();

tokens = token;

tokenLength = tokenQuantity;

currentToken = 0;

printf("\nSTART\_PARSER\n");

if (Block())

{

printf("\nSuccessfully (A good program without errors well done)\n");

}

else

{

printf("\nSyntax error on the line: %d\n",tokens[currentToken].line+1);

}

Print2D(&root);

printf("\nEND\_PARSER\n");

}

bool Block()

{

currentToken = tempCurrentToken;

if(currentToken < tokenLength)

{

//printf("\nnew Block token - %s #%d\n", NameType(tokens[currentToken].type), currentToken);

}

if(currentToken >= tokenLength)

{

tabCount -= 1;

return true;

}

else if (tokens[currentToken].type == END)

{

tabCount -= 1;

return true;

}

else

{

tempCurrentToken = currentToken;

if (needTabCount != 0)

{

int currentTab = 0;

while (Is(Tab))

{

currentTab += 1;

}

tempCurrentToken = currentToken;

if (currentTab != needTabCount)

{

if (currentTab > needTabCount)

{

printf("\nError - incorrect number of tabs(a lot of tabs)\n");

}

else

{

printf("\nError - incorrect number of tabs(small number of tabs)\n");

}

return false;

}

needTabCount = 0;

}

for (int i = 0; i < tabCount; i++)

{

if (!Is(Tab))

{

current = current->parent;

tempCurrentToken = currentToken;

tabCount -= 1;

return true;

}

}

currentToken = tempCurrentToken;

if (sequence == CommonBlock)

{

if (Statement() && Block())

{

sequence = CommonBlock;

needTabCount = 0;

return true;

}

}

else if (sequence == ClassBlock)

{

if (StatementClass() && Block())

{

prevBlock = CommonBlock;

sequence = CommonBlock;

needTabCount = 0;

return true;

}

}

else if (sequence == DefBlock)

{

if (StatementDef() && Block())

{

sequence = prevBlock;

needTabCount = 0;

return true;

}

}

return false;

}

printf("\nerror Block\n");

return false;

}

bool StatementDef()

{

if (Statement())

{

return true;

}

else

{

if (Return1())

{

currentToken = tempCurrentToken;

return true;

}

return false;

}

}

bool Return1()

{

if (Is(Return))

{

struct Node\* returnNode = AddChild(tokens[tempCurrentToken - 1], current);

if (Is(Delimiter) || (Value(returnNode) && Is(Delimiter)))

{

return true;

}

}

return false;

}

bool StatementClass()

{

tempCurrentToken = currentToken;

if (Is(Delimiter))

{

return true;

}

if (AssignmentOperation())

{

currentToken = tempCurrentToken;

return true;

}

else

{

if (FunctionDefinition())

{

currentToken = tempCurrentToken;

return true;

}

}

}

bool Statement()

{

tempCurrentToken = currentToken;

if(Is(Delimiter))

{

return true;

}

if (Condition())

{

currentToken = tempCurrentToken;

return true;

}

else

{

tempCurrentToken = currentToken;

if (AssignmentOperation())

{

currentToken = tempCurrentToken;

return true;

}

else

{

tempCurrentToken = currentToken;

if (InputCommand(current))

{

currentToken = tempCurrentToken;

return true;

}

else

{

tempCurrentToken = currentToken;

if (OutputCommand())

{

currentToken = tempCurrentToken;

return true;

}

else

{

if(WhileLoop())

{

currentToken = tempCurrentToken;

return true;

}

else

{

if(ForLoop())

{

currentToken = tempCurrentToken;

return true;

}

else

{

if(FunctionDefinition())

{

currentToken = tempCurrentToken;

return true;

}

else

{

if(FunctionCall(current))

{

currentToken = tempCurrentToken;

return true;

}

else

{

if (ClassDefinition())

{

currentToken = tempCurrentToken;

return true;

}

}

}

}

}

}

}

}

}

return false;

}

bool ClassDefinition()

{

if (Is(Class))

{

struct Node\* funcNode = NewNode(tokens[tempCurrentToken - 1]);

if (Is(Variable))

{

struct Node\* funcNodetemp = AddChild(tokens[tempCurrentToken - 1], funcNode);

if (Is(OpenBracket))

{

if (Is(CloseBracket))

{

if (Is(DoubleDot))

{

if (Is(Delimiter))

{

tabCount += 1;

needTabCount = tabCount;

sequence = ClassBlock;

enum NonTerminalType body = BlockTerm;

struct Node \* returnCurrent = current;

AddChildNode(funcNode, current);

current = AddNextTerminal(body, funcNodetemp);

if (Block())

{

current = returnCurrent;

currentToken = tempCurrentToken;

return true;

}

}

}

}

}

}

}

}

bool FunctionDefinition()

{

tempCurrentToken = currentToken;

if(Is(Def))

{

struct Node\* funcNode = NewNode(tokens[tempCurrentToken - 1]);

if(Is(Variable))

{

struct Node\* funcNodetemp = AddChild(tokens[tempCurrentToken - 1], funcNode);

if (Is(OpenBracket))

{

enum NonTerminalType param = Params;

struct Node \* params = AddNextTerminal(param, funcNodetemp);

if (ArgumentsFunctionDefinition(params))

{

if (Is(CloseBracket))

{

if (Is(DoubleDot))

{

if (Is(Delimiter))

{

tabCount += 1;

needTabCount = tabCount;

prevBlock = sequence;

sequence = DefBlock;

enum NonTerminalType body = BlockTerm;

struct Node\* returnCurrent = current;

AddChildNode(funcNode, current);

current = AddNextTerminal(body, funcNodetemp);

if (Block())

{

current = returnCurrent;

currentToken = tempCurrentToken;

return true;

}

}

}

}

}

}

}

}

return false;

}

bool ArrayDefinition(struct Node\* ArrayDefinition)

{

if (Is(OpenBraces))

{

struct Node\* arrayDefinition = NewNode(tokens[tempCurrentToken - 1]);

enum NonTerminalType ParamsTerm = Params;

struct Node\* paramsNode = AddChildTerminal(ParamsTerm, arrayDefinition);

if (Arguments(paramsNode))

{

if (Is(CloseBraces))

{

AddChildNode(arrayDefinition, ArrayDefinition);

AddChild(tokens[tempCurrentToken - 1], arrayDefinition);

return true;

}

}

}

}

bool FunctionCall(struct Node \* functionCallNode)

{

if(Is(Variable))

{

enum NonTerminalType funcTerm = Function;

struct Node\* functionCallTemp = NewNodeTerminal(funcTerm);

struct Node\* functionCallChild = AddChild(tokens[tempCurrentToken - 1], functionCallTemp);

enum NonTerminalType params = Params;

functionCallChild = AddNextTerminal(params, functionCallChild);

if(Is(OpenBracket))

{

if (Arguments(functionCallChild))

{

if (Is(CloseBracket))

{

if (Is(Delimiter))

{

AddChildNode(functionCallTemp, functionCallNode);

currentToken = tempCurrentToken;

return true;

}

}

}

}

}

return false;

}

bool ArgumentsFunctionDefinition(struct Node \* argumentsFunctionNode)

{

if (Is(Variable))

{

struct Node\* argFuncTemp = NewNode(tokens[tempCurrentToken-1]);

if (Is(Comma))

{

struct Node\* commaNode = AddNext(tokens[tempCurrentToken - 1], argFuncTemp);

if (ArgumentFunctionDefinition(commaNode))

{

AddChildNode(argFuncTemp, argumentsFunctionNode);

return true;

}

else

{

return false;

}

}

else

{

AddChildNode(argFuncTemp, argumentsFunctionNode);

return true;

}

}

else

{

return true;

}

}

bool ArgumentFunctionDefinition(struct Node\* argumentFunctionNode)

{

if (Is(Variable))

{

struct Node\* argFuncTemp = NewNode(tokens[tempCurrentToken - 1]);

if (Is(Comma))

{

struct Node\* commaNode = AddNext(tokens[tempCurrentToken - 1], argFuncTemp);

if (ArgumentFunctionDefinition(commaNode))

{

AddChildNode(argFuncTemp, argumentFunctionNode);

return true;

}

else

{

return false;

}

}

else

{

AddChildNode(argFuncTemp, argumentFunctionNode);

return true;

}

}

else

{

return false;

}

}

bool Arguments(struct Node\* argumentsNode)

{

if (Value(argumentsNode))

{

if (Is(Comma))

{

struct Node \* commaNode = AddChild(tokens[tempCurrentToken - 1], argumentsNode);

if (Argument(commaNode))

{

return true;

}

else

{

return false;

}

}

else

{

return true;

}

}

else

{

return true;

}

}

bool Argument(struct Node\* argumentNode)

{

if (Value(argumentNode))

{

if (Is(Comma))

{

struct Node \* commaNode = AddChild(tokens[tempCurrentToken - 1], argumentNode);

if (Argument(commaNode))

{

return true;

}

else

{

return false;

}

}

else

{

return true;

}

}

else

{

return false;

}

}

bool AssignmentOperation()

{

tempCurrentToken = currentToken;

struct Token assignment;

assignment.value = "=";

assignment.type = Assignment;

struct Node\* assignmentNode = NewNode(assignment);

if (Identificator(assignmentNode))

{

if (Is(Assignment))

{

AddChildNode(assignmentNode, current);

currentToken = tempCurrentToken;

if (ArithmeticExpressionMain(false, assignmentNode) && Is(Delimiter))

{

currentToken = tempCurrentToken;

return true;

}

tempCurrentToken = currentToken;

if (InputCommand(assignmentNode))

{

currentToken = tempCurrentToken;

return true;

}

tempCurrentToken = currentToken;

if (FunctionCall(assignmentNode))

{

currentToken = tempCurrentToken;

return true;

}

tempCurrentToken = currentToken;

if (ArrayDefinition(assignmentNode) && Is(Delimiter))

{

currentToken = tempCurrentToken;

return true;

}

}

}

return false;

}

bool OutputCommand()

{

tempCurrentToken = currentToken;

if (Is(Output))

{

struct Node\* outputNode = NewNode(tokens[currentToken]);

if (Is(OpenBracket))

{

if (ArithmeticExpressionMain(true, outputNode))

{

if (Is(Delimiter))

{

AddChildNode(outputNode, current);

currentToken = tempCurrentToken;

}

return true;

}

}

}

return false;

}

bool InputCommand(struct Node \* inputNode)

{

if (Is(Input))

{

if (Is(OpenBracket))

{

if (Is(CloseBracket))

{

if (Is(Delimiter))

{

AddChild(tokens[tempCurrentToken - 4], inputNode);

return true;

}

}

}

}

return false;

}

bool LogicalOperationMain(struct Node\* arifNode)

{

int bracketCountDifference = 0;

if (Is(OpenBracket))

{

bracketCountDifference++;

}

if (LogicalOperation(&bracketCountDifference, arifNode))

{

if (Is(CloseBracket))

{

bracketCountDifference--;

}

if (bracketCountDifference == 0)

{

return true;

}

else

{

printf("\ncountbracket:! %d !\n", bracketCountDifference);

printf("\nError - the difference in the number of opening brackets to the closing ones: %d\n", tokens[tempCurrentToken].pos);

}

}

return false;

}

bool LogicalOperation(int \* bracketCountDifference, struct Node\* arifNode)

{

if (Is(OpenBracket))

{

(\*bracketCountDifference)++;

}

struct Node\* valueNode = NewNode(tokens[currentToken]);

if (LogicalExpression(bracketCountDifference, valueNode))

{

if (Is(Logical))

{

struct Node\* childLog = AddChild(tokens[tempCurrentToken - 1], arifNode);

AddChildNode(valueNode->childs, childLog);

if (LogicalOperation(bracketCountDifference, childLog))

{

currentToken = tempCurrentToken;

}

else

{

return false;

}

}

else

{

AddChildNode(valueNode->childs, arifNode);

}

if (Is(CloseBracket))

{

(\* bracketCountDifference)--;

}

return true;

}

return false;

}

bool LogicalExpression(int \* bracketCountDifference, struct Node\* arifNode)

{

if (Is(OpenBracket))

{

(\*bracketCountDifference)++;

}

if (ArithmeticExpressionLO(bracketCountDifference, arifNode))

{

if(Is(Comparison))

{

struct Node\* childComp = AddChild(tokens[tempCurrentToken - 1], arifNode);

if(LogicalExpression(bracketCountDifference, childComp))

{

currentToken = tempCurrentToken;

}

else

{

return false;

}

}

if (Is(CloseBracket))

{

(\*bracketCountDifference)--;

}

return true;

}

return false;

}

bool ArithmeticExpressionLO(int \* bracketCountDifference, struct Node\* arifNode)

{

int temp = tempCurrentToken;

if (ArithmeticExpression(bracketCountDifference, arifNode))

{

return true;

}

tempCurrentToken = temp;

return false;

}

bool ArithmeticExpressionMain(bool open, struct Node\* arifNode)

{

int temp = tempCurrentToken;

int bracketCountDifference = open;

if (ArithmeticExpression(&bracketCountDifference, arifNode))

{

if (bracketCountDifference == 0)

{

return true;

}

else

{

printf("\ncountbracket:%d\n", bracketCountDifference);

printf("\nError - the difference in the number of opening brackets to the closing ones: %d\n", tokens[tempCurrentToken].pos);

}

}

tempCurrentToken = temp;

return false;

}

struct Node\* bracketNode;

bool ArithmeticExpression(int \* bracketCount, struct Node\* arifNode)

{

if (Is(OpenBracket))

{

(\*bracketCount)++;

}

struct Node\* valueNode=NewNode(tokens[currentToken]);

if (Value(valueNode))

{

if (Is(MathSign))

{

struct Node\* mathNode = AddChild(tokens[tempCurrentToken-1],arifNode);

AddChildNode(valueNode->childs, mathNode);

if(ArithmeticExpression(bracketCount, mathNode))

{

currentToken = tempCurrentToken;

}

}

else

{

AddChildNode(valueNode->childs, arifNode);

}

if (Is(CloseBracket))

{

(\*bracketCount)--;

if ((\*bracketCount) < 0)

{

printf("Error - missing opening parenthesis before: %d", tokens[tempCurrentToken].pos);

return false;

}

}

return true;

}

return false;

}

bool Value(struct Node\* valueNode)

{

int temp = tempCurrentToken;

if (Is(Variable) && Is(Dot) && Is(Variable))

{

struct Node\* temp = AddChild(tokens[tempCurrentToken - 3], valueNode);

struct Node\* forNext = AddChild(tokens[tempCurrentToken - 2], temp);

AddNext(tokens[tempCurrentToken - 1], forNext);

return true;

}

tempCurrentToken = temp;

if (Is(Variable) && Is(OpenBraces) )

{

if (ArithmeticExpressionMain(false, valueNode) && Is(CloseBraces))

{

return true;

}

}

tempCurrentToken = temp;

if (Is(Variable) || Is(Number) || Is(Bool))

{

AddChild(tokens[tempCurrentToken - 1], valueNode);

return true;

}

return false;

}

bool Identificator(struct Node\* identificatorNode)

{

int temp = tempCurrentToken;

if (Is(Variable) && Is(Dot) && Is(Variable))

{

struct Node\* temp = AddChild(tokens[tempCurrentToken - 3], identificatorNode);

struct Node\* forNext = AddChild(tokens[tempCurrentToken - 2], temp);

AddNext(tokens[tempCurrentToken - 1], forNext);

return true;

}

tempCurrentToken = temp;

if (Is(Variable) && Is(OpenBraces) && ArithmeticExpressionMain(false, identificatorNode) && Is(CloseBraces))

{

return true;

}

tempCurrentToken = temp;

if (Is(Variable))

{

AddChild(tokens[tempCurrentToken - 1], identificatorNode);

return true;

}

return false;

}

bool Condition()

{

int tempCurrentToken1 = currentToken;

tempCurrentToken = currentToken;

if (Is(If))

{

struct Node\* ifNode = NewNode(tokens[currentToken]);

enum NonTerminalType lo = LogicalOperationTerm;

struct Node\* loNode = AddChildTerminal(lo, ifNode);

if (LogicalOperationMain(loNode) && Is(DoubleDot) && Is(Delimiter))

{

tabCount += 1;

needTabCount = tabCount;

enum NonTerminalType body = BlockTerm;

struct Node\* returnCurrent = current;

AddChildNode(ifNode, current);

current = AddNextTerminal(body, loNode);

if (Block())

{

current = returnCurrent;

currentToken = tempCurrentToken;

if (Is(Else))

{

struct Node\* elseNode = NewNode(tokens[currentToken]);

if(Is(DoubleDot) && Is(Delimiter))

{

tabCount += 1;

needTabCount = tabCount;

enum NonTerminalType body = BlockTerm;

returnCurrent = current;

AddChildNode(elseNode, current);

current = AddChildTerminal(body, elseNode);

if (Block())

{

current = returnCurrent;

currentToken = tempCurrentToken;

}

}

}

return true;

}

}

}

else

{

currentToken = tempCurrentToken1;

return false;

}

}

bool WhileLoop()

{

if(Is(While))

{

struct Node\* whileNode = NewNode(tokens[currentToken]);

enum NonTerminalType lo = LogicalOperationTerm;

struct Node\* loNode = AddChildTerminal(lo, whileNode);

if(LogicalOperationMain(loNode))

{

if(Is(DoubleDot))

{

if(Is(Delimiter))

{

tabCount += 1;

needTabCount = tabCount;

enum NonTerminalType body = BlockTerm;

struct Node\* returnCurrent = current;

AddChildNode(whileNode, current);

current = AddNextTerminal(body, loNode);

if(Block())

{

current = returnCurrent;

currentToken = tempCurrentToken;

return true;

}

}

}

}

}

}

bool ForLoop()

{

tempCurrentToken = currentToken;

if(Is(For))

{

struct Node\* forNode = NewNode(tokens[currentToken]);

if(Identificator(forNode))

{

if(Is(In))

{

if(Identificator(forNode))

{

if(Is(DoubleDot))

{

if(Is(Delimiter))

{

tabCount += 1;

needTabCount = tabCount;

enum NonTerminalType body = BlockTerm;

struct Node\* returnCurrent = current;

AddChildNode(forNode, current);

current = AddChildTerminal(body, forNode);

if(Block())

{

current = returnCurrent;

currentToken = tempCurrentToken;

return true;

}

}

}

}

}

}

}

}

bool Is(enum TokenType type)

{

if (tokens[tempCurrentToken].type == type)

{

tempCurrentToken++;

return true;

}

return false;}

# ПРИЛОЖЕНИЕ Е (обязательное) ФРАГМЕНТЫ РЕАЛИЗАЦИИ ПОСТРОЕНИЯ АБСТРАКТНОГО СИНТАКСИЧЕСКОГО ДЕРЕВА(tree.h)

#include<stdio.h>

#include<malloc.h>

#include "token.h"

#define COUNT 5

struct Node

{

enum NonTerminalType type;

struct Token data;

struct Node\* parent;

struct Node\* next;

struct Node\* childs;

};

struct Node\* NewNode(struct Token data);

struct Node\* NewNodeTerminal(enum NonTerminalType data);

struct Node\* AddChild(struct Token data, struct Node\* node);

struct Node\* AddChildNode(struct Node\* child, struct Node\* node);

struct Node\* AddChildTerminal(enum NonTerminalType data, struct Node\* node);

void AddParent(struct Token data, struct Node\* node);

struct Node\* AddNext(struct Token data, struct Node\* node);

struct Node\* AddNextTerminal(enum NonTerminalType data, struct Node\* node);

void PrintTree(struct Node \*root, int space);

void Print2D(struct Node \*root);

# ПРИЛОЖЕНИЕ Ё (обязательное) ФРАГМЕНТЫ РЕАЛИЗАЦИИ ПОСТРОЕНИЯ АБСТРАКТНОГО СИНТАКСИЧЕСКОГО ДЕРЕВА(tree.c)

#include "tree.h"

struct Node\* NewNode(struct Token data)

{

struct Node\* node = malloc(sizeof(struct Node));

node->data = data;

node->type = Terminal;

node->next = node->childs = NULL;

return node;

}

struct Node\* NewNodeTerminal(enum NonTerminalType data)

{

struct Node\* node = malloc(sizeof(struct Node));

node->type = data;

node->next = node->childs = NULL;

return node;

}

struct Node \* AddChild(struct Token data, struct Node\* node)

{

if (node == NULL)

{

node = malloc(sizeof(struct Node));

node->data = data;

node->type = Terminal;

node->next = node->childs = NULL;

return node;

}

else

{

if (node->childs == NULL)

{

node->childs = malloc(sizeof(struct Node));

node->childs->data = data;

node->childs->type = Terminal;

node->childs->next = node->childs->childs = NULL;

node->childs->parent = node;

return node->childs;

}

struct Node\*\* temp = &node->childs;

while ((\*temp)->next != NULL)

{

temp = (&((\*temp)->next));

}

if ((\*temp)->next == NULL)

{

(\*temp)->next = malloc(sizeof(struct Node));

(\*temp)->next->parent = node;

(\*temp)->next->type = Terminal;

(\*temp)->next->data = data;

(\*temp)->next->next = (\*temp)->next->childs = NULL;

return (\*temp)->next;

}

}

}

struct Node\* AddChildTerminal(enum NonTerminalType data, struct Node\* node)

{

if (node == NULL)

{

node = malloc(sizeof(struct Node));

node->type = data;

node->next = node->childs = NULL;

return node;

}

else

{

if (node->childs == NULL)

{

node->childs = malloc(sizeof(struct Node));

node->childs->type = data;

node->childs->next = node->childs->childs = NULL;

node->childs->parent = node;

return node->childs;

}

struct Node\*\* temp = &node->childs;

while ((\*temp)->next != NULL)

{

temp = (&((\*temp)->next));

}

if ((\*temp)->next == NULL)

{

(\*temp)->next = malloc(sizeof(struct Node));

(\*temp)->next->parent = node;

(\*temp)->next->type = data;

(\*temp)->next->next = (\*temp)->next->childs = NULL;

return (\*temp)->next;

}

}

}

struct Node\* AddChildNode(struct Node\* child, struct Node\* node)

{

if (node == NULL)

{

node = malloc(sizeof(struct Node));

node->childs = child;

child->parent = node;

return child;

}

else

{

if (node->childs == NULL)

{

struct Node\*\* temp = &child;

while ((\*temp)->next != NULL)

{

(\*temp)->next->parent = node;

temp = (&((\*temp)->next));

}

node->childs = malloc(sizeof(struct Node));

node->childs = child;

node->childs->parent = node;

return child;

}

struct Node\*\* temp2 = &node->childs;

while ((\*temp2)->next != NULL)

{

temp2 = (&((\*temp2)->next));

}

if ((\*temp2)->next == NULL)

{

struct Node\*\* temp1 = &child;

while ((\*temp1)->next != NULL)

{

(\*temp1)->next->parent = node;

temp1 = (&((\*temp1)->next));

}

(\*temp2)->next = malloc(sizeof(struct Node));

(\*temp2)->next = child;

(\*temp2)->next->parent = node;

return child;

}

}

}

void AddParent(struct Token data, struct Node\* node)

{

if (node == NULL)

{

node = malloc(sizeof(struct Node));

node->data = data;

node->next = node->childs = NULL;

}

else

{

if (node->parent == NULL)

{

node->parent = malloc(sizeof(struct Node));

node->parent->data = data;

node->parent->next = NULL;

node->parent->childs = node;

return;

}

struct Node\*\* temp = &node->childs;

while ((\*temp)->next != NULL)

{

temp = (&((\*temp)->next));

}

if ((\*temp)->next == NULL)

{

(\*temp)->next->parent = node;

(\*temp)->next = malloc(sizeof(struct Node));

(\*temp)->next->data = data;

(\*temp)->next->next = (\*temp)->next->childs = NULL;

}

}

}

struct Node \* AddNext(struct Token data, struct Node\* node)

{

if (node == NULL)

{

node = malloc(sizeof(struct Node));

node->data = data;

node->type = Terminal;

node->next = node->childs = NULL;

return node;

}

else

{

if (node->next == NULL)

{

node->next = malloc(sizeof(struct Node));

node->next->data = data;

node->next->type = Terminal;

node->next->parent = node->parent;

node->next->next = node->next->childs = NULL;

return node->next;

}

struct Node\*\* temp = &node->next;

while ((\*temp)->next != NULL)

{

temp = (&((\*temp)->next));

}

if ((\*temp)->next == NULL)

{

(\*temp)->next = malloc(sizeof(struct Node));

(\*temp)->next->parent = node->parent;

(\*temp)->next->data = data;

(\*temp)->next->type = Terminal;

(\*temp)->next->next = (\*temp)->next->childs = NULL;

return (\*temp)->next;

}

}

}

struct Node\* AddNextTerminal(enum NonTerminalType data, struct Node\* node)

{

if (node == NULL)

{

node = malloc(sizeof(struct Node));

node->type = data;

node->next = node->childs = NULL;

return node;

}

else

{

if (node->next == NULL)

{

node->next = malloc(sizeof(struct Node));

node->next->type = data;

node->next->parent = node->parent;

node->next->next = node->next->childs = NULL;

return node->next;

}

struct Node\*\* temp = &node->next;

while ((\*temp)->next != NULL)

{

temp = (&((\*temp)->next));

}

if ((\*temp)->next == NULL)

{

(\*temp)->next = malloc(sizeof(struct Node));

(\*temp)->next->parent = node->parent;

(\*temp)->next->type = data;

(\*temp)->next->next = (\*temp)->next->childs = NULL;

return (\*temp)->next;

}

}

}

void PrintTree(struct Node \*root, int space)

{

if (root == NULL)

return;

for (int i = 0; i < space; i++)

printf(" ");

if (root->type == Terminal)

{

if (root->childs == NULL || root->data.type == Variable)

{

printf("%s - %s\n", NameType(root->data.type), ((\*root).data.value));

}

else

{

printf("%s\n", NameType(root->data.type));

}

}

else

{

printf("%s\n", NameNonTerminalType(root->type));

}

PrintTree(root->childs, space + COUNT);

PrintTree(root->next, space);

}

void Print2D(struct Node \*root)

{

printf("\nTree: \n\n");

PrintTree(root, 0);

printf("\nTree end; \n");

}