УО «Белорусский государственный университет информатики и радиоэлектроники»

Кафедра ПОИТ

Отчет по лабораторной работе №3

по предмету

Компиляторные технологии

Вариант 5

Выполнил:

Наривончик А.М.

Группа 351004

Проверила:

Болтак С.В.

Минск 2024

**Задание:**

Разработать программное средство, проверяющее исходный код программы на соответствие грамматике (Python).

Лексический анализатор:

%option main

%{

#include <stdio.h>

#include <stdlib.h>

int line = 1;

void yyerror(char \*message);

%}

digit [0-9]

letter [\_a-zA-Z]

quote ['"]

tab [\t]

white\_space [ ]

unsigned {digit}+

signed -{unsigned}

arithmetic\_operator [-+\*/]

comparison\_sign <|>|>=|<=|==|!=

colon :

assignment =

bracket\_open [(]

bracket\_close [)]

identificator {letter}({letter}|{digit})\*

string {quote}({letter}|{white\_space})\*{quote}

new\_line [\n]

%%

if printf ("%s|%s|%d\n", "ttIF", yytext, line);

else printf ("%s|%s|%d\n", "ttELSE", yytext, line);

elif printf ("%s|%s|%d\n", "ttELIF", yytext, line);

while printf ("%s|%s|%d\n", "ttWHILE", yytext, line);

for printf ("%s|%s|%d\n", "ttFOR", yytext, line);

in printf ("%s|%s|%d\n", "ttIN", yytext, line);

and printf ("%s|%s|%d\n", "ttAND", yytext, line);

or printf ("%s|%s|%d\n", "ttOR", yytext, line);

match printf ("%s|%s|%d\n", "ttMATCH", yytext, line);

case printf ("%s|%s|%d\n", "ttCASE", yytext, line);

{bracket\_open} printf ("%s|%s|%d\n", "ttBRACKET\_OPEN", yytext, line);

{bracket\_close} printf ("%s|%s|%d\n", "ttBRACKET\_CLOSE", yytext, line);

{colon} printf ("%s|%s|%d\n", "ttCOLON", yytext, line);

{assignment} printf ("%s|%s|%d\n", "ttASSIGNMENT", yytext, line);

{unsigned} printf ("%s|%s|%d\n", "ttINT", yytext, line);

{signed} printf ("%s|%s|%d\n", "ttINT", yytext, line);

{arithmetic\_operator} printf ("%s|%s|%d\n", "ttARITHMETIC\_OPERATOR", yytext, line);

{comparison\_sign} printf ("%s|%s|%d\n", "ttCOMPARISON\_SIGN", yytext, line);

{identificator} printf ("%s|%s|%d\n", "ttIDENTIFICATOR", yytext, line);

{string} printf ("%s|%s|%d\n", "ttSTRING", yytext, line);

{tab} printf ("%s|%s|%d\n", "ttTAB", yytext, line);

{white\_space} {}

{new\_line} line++;

. yyerror("unrecognized symbol");

%%

void yyerror(char \*message) {

fprintf(stderr, "fatal error: line %d - %s - lexem %s\n", line, message, yytext);

}

**Грамматика языка:**

<Program> --> <Statement> <StatementSet> | ε

<StatementSet> --> <Statement> <StatementSet> | ε

<Statement> --> <IfStatement>

| <WhileStatement>

| <MatchStatement>

| <AssignmentStatement>

<IfStatement> --> ttIF <Condition> ttCOLON <Block> <IfTail>

<IfTail> --> <ElifBlock> <IfTail> | <ElseBlock> | ε

<ElifBlock> --> ttELIF <Condition> ttCOLON <Block>

<ElseBlock> --> ttELSE ttCOLON <Block>

<WhileStatement> --> ttWHILE <Condition> ttCOLON <Block>

<MatchStatement> --> ttMATCH <Expression> ttCOLON <CaseBlockSet>

<CaseBlockSet> --> <CaseBlock> <CaseBlockSet> | ε

<CaseBlock> --> ttTAB <CaseBlock>

| ttCASE <Expression> ttCOLON <Block>

<AssignmentStatement> --> ttIDENTIFICATOR ttASSIGNMENT <Expression>

<Operand> --> ttIDENTIFICATOR

| ttINT

| ttSTRING

| ttBRACKET\_OPEN <Expression> ttBRACKET\_CLOSE

<Expression> --> <Operand> <ArithmeticExpressionTail>

<ArithmeticExpressionTail> --> ttARITHMETIC\_OPERATOR <Operand> <ArithmeticExpressionTail> | ε

<Condition> --> <Expression> ttCOMPARISON\_SIGN <Expression>

| ttBRACKET\_OPEN <Condition> ttBRACKET\_CLOSE

<Block> --> <IndentedStatementSet>

<IndentedStatementSet> --> <IndentedStatement> <IndentedStatementSet> | ε

<IndentedStatement> --> ttTAB <IndentedStatement> | <Statement>

**Код программы:**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_TOKEN\_COUNT 256

////// EnumDefinitions //////

typedef enum TokenType {

ttASSIGNMENT,

ttARITHMETIC\_OPERATOR,

ttCOMPARISON\_SIGN,

ttCOLON,

ttTAB,

ttIF,

ttELSE,

ttELIF,

ttWHILE,

ttFOR,

ttIN,

ttIDENTIFICATOR,

ttSTRING,

ttINT,

ttAND,

ttOR,

ttMATCH,

ttCASE,

ttBRACKET\_OPEN,

ttBRACKET\_CLOSE,

ttNEW\_LINE,

ttEMPTY\_STR,

ttUNKNOWN,

ttPROGRAM\_END

} TokenType;

typedef TokenType\* PTokenType;

////// EnumDefinitions //////

////// StructDefinitions //////

typedef struct Node {

const char\* value;

char\* inSource;

int count;

struct Node\*\* nodes;

} Node;

typedef struct Token {

TokenType type;

char\* inSource;

int line;

} Token;

typedef Token\* PToken;

////// StructDefinitions //////

////// GlobalVariables //////

PToken arr;

int next;

int maxAvailable;

TokenType expectedToken;

Node\* tree;

////// GlobalVariables //////

////// NodeFunctions //////

Node\* createNode(const char\* value, char\* inSource, int count);

Node\* freeNode(Node\* curr);

////// NodeFunctions //////

//////// TreeFunctions //////

Node\* freeTree(Node\* curr);

//////// TreeFunctions //////

//////// PrintFunctions //////

void printNode(Node\* curr, int h);

void printTree(Node\* curr, int h);

//////// PrintFunctions //////

//// ParserFunctionsPrototypes //////

Node\* errorToken();

Node\* term(const TokenType expected);

Node\* program();

Node\* statement();

Node\* statementSet();

Node\* ifStatement();

Node\* ifTail();

Node\* elifBlock();

Node\* elseBlock();

Node\* whileStatement();

Node\* matchStatement();

Node\* caseBlockSet();

Node\* caseBlock();

Node\* assignmentStatement();

Node\* operand();

Node\* expression();

Node\* arithmeticExpressionTail();

Node\* condition();

Node\* block();

Node\* indentedStatementSet();

Node\* indentedStatement();

//// ParserFunctionsPrototypes //////

//// TokenFunctions //////

TokenType convertStringToToken(char\* str) {

if (!strcmp(str, "ttARITHMETIC\_OPERATOR")) {

return ttARITHMETIC\_OPERATOR;

}

if (!strcmp(str, "ttCOMPARISON\_SIGN")) {

return ttCOMPARISON\_SIGN;

}

if (!strcmp(str, "ttCOLON")) {

return ttCOLON;

}

if (!strcmp(str, "ttASSIGNMENT")) {

return ttASSIGNMENT;

}

if (!strcmp(str, "ttTAB")) {

return ttTAB;

}

if (!strcmp(str, "ttIF")) {

return ttIF;

}

if (!strcmp(str, "ttELSE")) {

return ttELSE;

}

if (!strcmp(str, "ttELIF")) {

return ttELIF;

}

if (!strcmp(str, "ttWHILE")) {

return ttWHILE;

}

if (!strcmp(str, "ttFOR")) {

return ttFOR;

}

if (!strcmp(str, "ttIN")) {

return ttIN;

}

if (!strcmp(str, "ttIDENTIFICATOR")) {

return ttIDENTIFICATOR;

}

if (!strcmp(str, "ttSTRING")) {

return ttSTRING;

}

if (!strcmp(str, "ttINT")) {

return ttINT;

}

if (!strcmp(str, "ttAND")) {

return ttAND;

}

if (!strcmp(str, "ttOR")) {

return ttOR;

}

if (!strcmp(str, "ttMATCH")) {

return ttMATCH;

}

if (!strcmp(str, "ttCASE")) {

return ttCASE;

}

if (!strcmp(str, "ttBRACKET\_OPEN")) {

return ttBRACKET\_OPEN;

}

if (!strcmp(str, "ttBRACKET\_CLOSE")) {

return ttBRACKET\_CLOSE;

}

if (!strcmp(str, "ttNEW\_LINE")) {

return ttNEW\_LINE;

}

if (!strcmp(str, "ttEMPTY\_STR")) {

return ttEMPTY\_STR;

}

return ttUNKNOWN;

}

char\* convertTokenToString(TokenType type) {

switch (type) {

case ttEMPTY\_STR:

return "";

case ttNEW\_LINE:

return "ttNEW\_LINE";

case ttIF:

return "ttIF";

case ttELSE:

return "ttELSE";

case ttELIF:

return "ttELIF";

case ttWHILE:

return "ttWHILE";

case ttFOR:

return "ttFOR";

case ttIN:

return "ttIN";

case ttIDENTIFICATOR:

return "ttIDENTIFICATOR";

case ttSTRING:

return "ttSTRING";

case ttINT:

return "ttINT";

case ttAND:

return "ttAND";

case ttOR:

return "ttOR";

case ttMATCH:

return "ttMATCH";

case ttCASE:

return "ttCASE";

case ttBRACKET\_OPEN:

return "ttBRACKET\_OPEN";

case ttBRACKET\_CLOSE:

return "ttBRACKET\_CLOSE";

case ttARITHMETIC\_OPERATOR:

return "ttARITHMETIC\_OPERATOR";

case ttCOMPARISON\_SIGN:

return "ttCOMPARISON\_SIGN";

case ttASSIGNMENT:

return "ttASSIGNMENT";

case ttCOLON:

return "ttCOLON";

case ttTAB:

return "ttTAB";

default:

return "ttUNKNOWN";

}

}

PToken getTokenFlowFromFile(char\* filePath) {

int i = 0;

char\* temp1 = (char\*)malloc(256 \* sizeof(char));

char\* temp2 = (char\*)malloc(256 \* sizeof(char));

int temp3;

FILE\* fInput;

fInput = fopen(filePath, "r");

arr = (PToken)malloc(sizeof(Token) \* MAX\_TOKEN\_COUNT);

while (!feof(fInput)) {

fscanf(fInput, "%255[^|]|%255[^|]|%d\n", temp1, temp2, &temp3);

arr[i].type = convertStringToToken(temp1);

arr[i].inSource = (char\*)malloc(256 \* sizeof(char));

strcpy(arr[i].inSource, temp2);

arr[i].line = temp3;

i++;

}

arr[i].type = ttPROGRAM\_END;

arr[i].inSource = "";

arr[i].line = -1;

fclose(fInput);

free(temp1);

free(temp2);

return arr;

}

////// TokenFunctions //////

//////// NodeFunctions //////

Node\* createNode(const char\* value, char\* inSource, int count) {

Node\* newNode;

newNode = (Node\*)malloc(sizeof(Node));

if (newNode == NULL) {

return newNode;

}

newNode->value = value;

newNode->inSource = inSource;

newNode->count = count;

if (count > 0) {

newNode->nodes = (Node\*\*)malloc(sizeof(Node\*) \* count);

for (int i = 0; i < count; i++) {

newNode->nodes[i] = NULL;

}

}

else {

newNode->nodes = NULL;

}

return newNode;

}

Node\* freeNode(Node\* curr) {

free(curr->nodes);

free(curr);

return NULL;

}

//////// NodeFunctions //////

//////// TreeFunctions //////

Node\* freeTree(Node\* curr) {

if (curr == NULL) {

return NULL;

}

else {

for (int i = 0; i < curr->count; i++) {

freeTree(curr->nodes[i]);

}

return freeNode(curr);

}

}

//////// TreeFunctions //////

//////// PrintFunctions //////

void printNode(Node\* curr, int h) {

int i;

for (i = 0; i < h - 1; i++) {

printf(" ");

}

if (h != 0) {

printf(" ");

}

if (curr->count == 0) {

if (curr->inSource) {

printf("<%s>------>(%s)\n", curr->value, curr->inSource);

}

else

printf("<%s>------>(%s)\n", curr->value, "eps");

}

else {

printf("<%s>\n", curr->value);

}

}

void printTree(Node\* curr, int h) {

if (curr != NULL) {

printNode(curr, h);

for (int i = 0; i < curr->count; i++) {

printTree(curr->nodes[i], h + 1);

}

}

}

//////// PrintFunctions //////

// ////// ParserFunctions //////

Node\* errorToken() {

printf("LINE: %d | EXPECTED token: %s | RECIEVED token: %s | value in file: %s.\n", arr[maxAvailable].line, convertTokenToString(expectedToken), convertTokenToString(arr[maxAvailable].type), arr[maxAvailable].inSource);

return NULL;

}

Node\* term(const TokenType expected) {

Node\* t = NULL;

if (expected == (arr[next].type)) {

t = createNode(convertTokenToString(arr[next].type), arr[next].inSource, 0);

if (arr[next].type != ttPROGRAM\_END) {

next += 1;

if (next > maxAvailable) {

maxAvailable = next;

}

}

}

else if (next == maxAvailable)

{

expectedToken = expected;

}

return t;

}

Node\* program() {

int save = next;

Node\* t1, \* t2, \* new;

next = save;

if ((t1 = statement()) && (t2 = statementSet())) {

new = createNode("Program", NULL, 2);

new->nodes[0] = t1;

new->nodes[1] = t2;

return new;

}

return createNode("Program", NULL, 0);

return NULL;

}

Node\* statement() {

int save = next;

Node\* t1, \* new;

next = save;

if ((t1 = ifStatement())) {

new = createNode("Statement", NULL, 1);

new->nodes[0] = t1;

return new;

}

next = save;

if ((t1 = whileStatement())) {

new = createNode("Statement", NULL, 1);

new->nodes[0] = t1;

return new;

}

next = save;

if ((t1 = matchStatement())) {

new = createNode("Statement", NULL, 1);

new->nodes[0] = t1;

return new;

}

next = save;

if ((t1 = assignmentStatement())) {

new = createNode("Statement", NULL, 1);

new->nodes[0] = t1;

return new;

}

return NULL;

}

Node\* statementSet() {

int save = next;

Node\* t1, \* t2, \* new;

next = save;

if ((t1 = statement()) && (t2 = statementSet())) {

new = createNode("StatementSet", NULL, 2);

new->nodes[0] = t1;

new->nodes[1] = t2;

return new;

}

return createNode("StatementSet", NULL, 0);

return NULL;

}

Node\* ifStatement() {

int save = next;

Node\* t1, \* t2, \* t3, \* t4, \* t5, \* new;

next = save;

if ((t1 = term(ttIF)) &&

(t2 = condition()) &&

(t3 = term(ttCOLON)) &&

(t4 = block()) &&

(t5 = ifTail())) {

new = createNode("ifStatement", NULL, 5);

new->nodes[0] = t1;

new->nodes[1] = t2;

new->nodes[2] = t3;

new->nodes[3] = t4;

new->nodes[4] = t5;

return new;

}

return NULL;

}

Node\* ifTail() {

int save = next;

Node\* t1, \* t2, \* new;

next = save;

if ((t1 = elifBlock()) && (t2 = ifTail())) {

new = createNode("IfTail", NULL, 2);

new->nodes[0] = t1;

new->nodes[1] = t2;

return new;

}

next = save;

if (t1 = elseBlock()) {

new = createNode("IfTail", NULL, 1);

new->nodes[0] = t1;

return new;

}

return createNode("IfTail", NULL, 0);

return NULL;

}

Node\* elifBlock() {

int save = next;

Node\* t1, \* t2, \* t3, \* t4, \* new;

next = save;

if ((t1 = term(ttELIF)) &&

(t2 = condition()) &&

(t3 = term(ttCOLON)) &&

(t4 = block())) {

new = createNode("ElifBlock", NULL, 4);

new->nodes[0] = t1;

new->nodes[1] = t2;

new->nodes[2] = t3;

new->nodes[3] = t4;

return new;

}

return NULL;

}

Node\* elseBlock() {

int save = next;

Node\* t1, \* t2, \* t3, \* new;

next = save;

if ((t1 = term(ttELSE)) &&

(t2 = term(ttCOLON)) &&

(t3 = block())) {

new = createNode("ElseBlock", NULL, 3);

new->nodes[0] = t1;

new->nodes[1] = t2;

new->nodes[2] = t3;

return new;

}

return NULL;

}

Node\* whileStatement() {

int save = next;

Node\* t1, \* t2, \* t3, \* t4, \* new;

next = save;

if ((t1 = term(ttWHILE)) &&

(t2 = condition()) &&

(t3 = term(ttCOLON)) &&

(t4 = block())) {

new = createNode("WhileStatement", NULL, 4);

new->nodes[0] = t1;

new->nodes[1] = t2;

new->nodes[2] = t3;

new->nodes[3] = t4;

return new;

}

return NULL;

}

Node\* matchStatement() {

int save = next;

Node\* t1, \* t2, \* t3, \* t4, \* new;

next = save;

if ((t1 = term(ttMATCH)) &&

(t2 = expression()) &&

(t3 = term(ttCOLON)) &&

(t4 = caseBlockSet())) {

new = createNode("MatchStatement", NULL, 4);

new->nodes[0] = t1;

new->nodes[1] = t2;

new->nodes[2] = t3;

new->nodes[3] = t4;

return new;

}

return NULL;

}

Node\* caseBlockSet() {

int save = next;

Node\* t1, \* t2, \* new;

next = save;

if ((t1 = caseBlock()) && (t2 = caseBlockSet())) {

new = createNode("CaseBlockSet", NULL, 2);

new->nodes[0] = t1;

new->nodes[1] = t2;

return new;

}

return createNode("CaseBlockSet", NULL, 0);

}

Node\* caseBlock() {

int save = next;

Node \*t1, \*t2, \*t3, \*t4, \*t5, \*new;

next = save;

if ((t1 = term(ttTAB)) &&

(t2 = caseBlock())) {

new = createNode("CaseBlock", NULL, 2);

new->nodes[0] = t1;

new->nodes[1] = t2;

return new;

}

next = save;

if ((t1 = term(ttCASE)) &&

(t2 = expression()) &&

(t3 = term(ttCOLON)) &&

(t4 = block())) {

new = createNode("CaseBlock", NULL, 4);

new->nodes[0] = t1;

new->nodes[1] = t2;

new->nodes[2] = t3;

new->nodes[3] = t4;

return new;

}

return NULL;

}

Node\* assignmentStatement() {

int save = next;

Node\* t1, \* t2, \* t3, \* new;

next = save;

if ((t1 = term(ttIDENTIFICATOR)) &&

(t2 = term(ttASSIGNMENT)) &&

(t3 = expression())) {

new = createNode("AssignmentStatement", NULL, 3);

new->nodes[0] = t1;

new->nodes[1] = t2;

new->nodes[2] = t3;

return new;

}

return NULL;

}

Node\* operand() {

PToken save = next;

Node\* t1, \* t2, \* t3, \* new;

next = save;

if ((t1 = term(ttIDENTIFICATOR))) {

new = createNode("Operand", NULL, 1);

new->nodes[0] = t1;

return new;

}

next = save;

if ((t1 = term(ttINT))) {

new = createNode("Operand", NULL, 1);

new->nodes[0] = t1;

return new;

}

next = save;

if ((t1 = term(ttSTRING))) {

new = createNode("Operand", NULL, 1);

new->nodes[0] = t1;

return new;

}

next = save;

if ((t1 = term(ttBRACKET\_OPEN)) &&

(t2 = expression()) &&

(t3 = term(ttBRACKET\_CLOSE))) {

new = createNode("Operand", NULL, 3);

new->nodes[0] = t1;

new->nodes[1] = t2;

new->nodes[2] = t3;

return new;

}

return NULL;

}

Node\* expression() {

int save = next;

Node\* t1, \* t2, \* new;

next = save;

if ((t1 = operand()) &&

(t2 = arithmeticExpressionTail())) {

new = createNode("Expression", NULL, 2);

new->nodes[0] = t1;

new->nodes[1] = t2;

return new;

}

return NULL;

}

Node\* arithmeticExpressionTail() {

int save = next;

Node\* t1, \* t2, \* t3, \* new;

next = save;

if ((t1 = term(ttARITHMETIC\_OPERATOR)) &&

(t2 = operand()) &&

(t3 = arithmeticExpressionTail())) {

new = createNode("ArithmeticExpressionTail", NULL, 3);

new->nodes[0] = t1;

new->nodes[1] = t2;

new->nodes[2] = t3;

return new;

}

return createNode("ArithmeticExpressionTail", NULL, 0);

return NULL;

}

Node\* condition() {

int save = next;

Node\* t1, \* t2, \* t3, \* new;

next = save;

if ((t1 = expression()) &&

(t2 = term(ttCOMPARISON\_SIGN)) &&

(t3 = expression())) {

new = createNode("Condition", NULL, 3);

new->nodes[0] = t1;

new->nodes[1] = t2;

new->nodes[2] = t3;

return new;

}

next = save;

if ((t1 = term(ttBRACKET\_OPEN)) &&

(t2 = condition()) &&

(t3 = term(ttBRACKET\_CLOSE))) {

new = createNode("Condition", NULL, 3);

new->nodes[0] = t1;

new->nodes[1] = t2;

new->nodes[2] = t3;

return new;

}

return NULL;

}

Node\* block() {

int save = next;

Node \*t1,\*new;

next = save;

if (t1 = indentedStatementSet()) {

new = createNode("Block", NULL, 1);

new->nodes[0] = t1;

return new;

}

return NULL;

}

Node\* indentedStatementSet() {

int save = next;

Node\* t1, \* t2, \* new;

next = save;

if ((t1 = indentedStatement()) && (t2 = indentedStatementSet())) {

new = createNode("IndentedStatementSet", NULL, 2);

new->nodes[0] = t1;

new->nodes[1] = t2;

return new;

}

return createNode("IndentedStatementSet", NULL, 0);

return NULL;

}

Node\* indentedStatement() {

int save = next;

Node\* t1, \* t2, \* new;

next = save;

if ((t1 = term(ttTAB)) && (t2 = indentedStatement())) {

new = createNode("IndentedStatement", NULL, 2);

new->nodes[0] = t1;

new->nodes[1] = t2;

return new;

}

if (t1 = statement()) {

new = createNode("IndentedStatement", NULL, 2);

new->nodes[0] = t1;

return new;

}

return NULL;

}

////// ParserFunctions //////

int main(void) {

char\* path;

PToken tokenFlow;

path = (char\*)malloc(sizeof(char) \* 256);

printf("Enter FilePath:\n");

scanf("%s", path);

tokenFlow = getTokenFlowFromFile(path);

next = 0;

maxAvailable = 0;

tree = program();

if (tree == NULL) {

printf("Unable To Parse");

}

else {

if (arr[next].type != ttPROGRAM\_END) {

printf("Syntax error\n");

errorToken();

}

else {

printTree(tree, 0);

}

}

fflush(stdin);

getchar();

free(tokenFlow);

free(path);

return 0;

}