Министерство образования Республики Беларусь Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №4

По теме “Семантический анализатор”

Выполнила: студентка гр. 053501 Шурко Т.А.

Проверил: ассистент кафедры информатики Гриценко Н. Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc131174104)

[2 Теория 4](#_Toc131174104)

[3 Программа и комментарии](#_Toc131174105) 5

[4 Демонстрация работы 6](#_Toc131174106)

[5 Демонстрация нахождения ошибок 7](#_Toc131174106)

[Вывод 9](#_Toc131174107)

[Приложение А Код программы 10](#_Toc131174107)

[Приложение Б Код программы 11](#_Toc131174107)

[Приложение В Листинг кода 12](#_Toc131174107)

[Приложение Г Текст программ 29](#_Toc131174107)

# **ЦЕЛЬ РАБОТЫ**

Освоение работы с существующими семантическими анализаторами.

Проведение семантического анализа; нахождение, обработка и вывод минимум 2 семантических ошибок.

В качестве анализируемого подмножества языка программирования будет использован язык программирования Python. Для написания анализатора использован язык программирования C++.

# **ТЕОРИЯ**

В процессе семантического анализа проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах для следующей стадии – генерации кода.

При семантическом анализе используются иерархические структуры, полученные во время синтаксического анализа для идентификации операторов и операндов выражений и инструкций. Важным аспектом семантического анализа является проверка типов, когда компилятор проверяет, что каждый оператор имеет операнды допустимого спецификациями языка типа. Например, определение многих языков программирования требует, чтобы при использовании действительного числа в качестве индекса массива генерировалось сообщение об ошибке. В то же время спецификация языка может позволить определенное насильственное преобразование типов, например, когда бинарный арифметический оператор применяется к операндам целого и действительного типов. В этом случае компилятору может потребоваться преобразование целого числа в действительное.

В большинстве языков программирования имеет место неявное изменение типов (иногда называемое приведением типов(coercion)). Реже встречаются языки, подобные Ada, в которых большинство изменений типов должно быть явным. В языках со статическими типами, например С, все типы известны во время компиляции, и это относится к типам выражений, идентификаторам и литералам. При этом неважно, насколько сложным является выражение: его тип может определяться во время компиляции за определенное количество шагов, исходя из типов его составляющих. Фактически, это позволяет производить контроль типов во время компиляции и находить заранее (в процессе компиляции, а не во время выполнения программы!) многие программные ошибки.

1. **ПРОГРАММА И КОММЕНТАРИИ**

Программа написана на с++. Семантический анализ представляет из себя дополнительные проверки при приведении и обработке типов данных. Реализация осуществлялась путем расширения класса Parser (Приложение В) программного продукта.

Для семантического анализа происходило построение дополнительной таблицы, в с++ аналогом таблицы является map (см. рисунок 3.1), с названиями переменных и их типами данных.

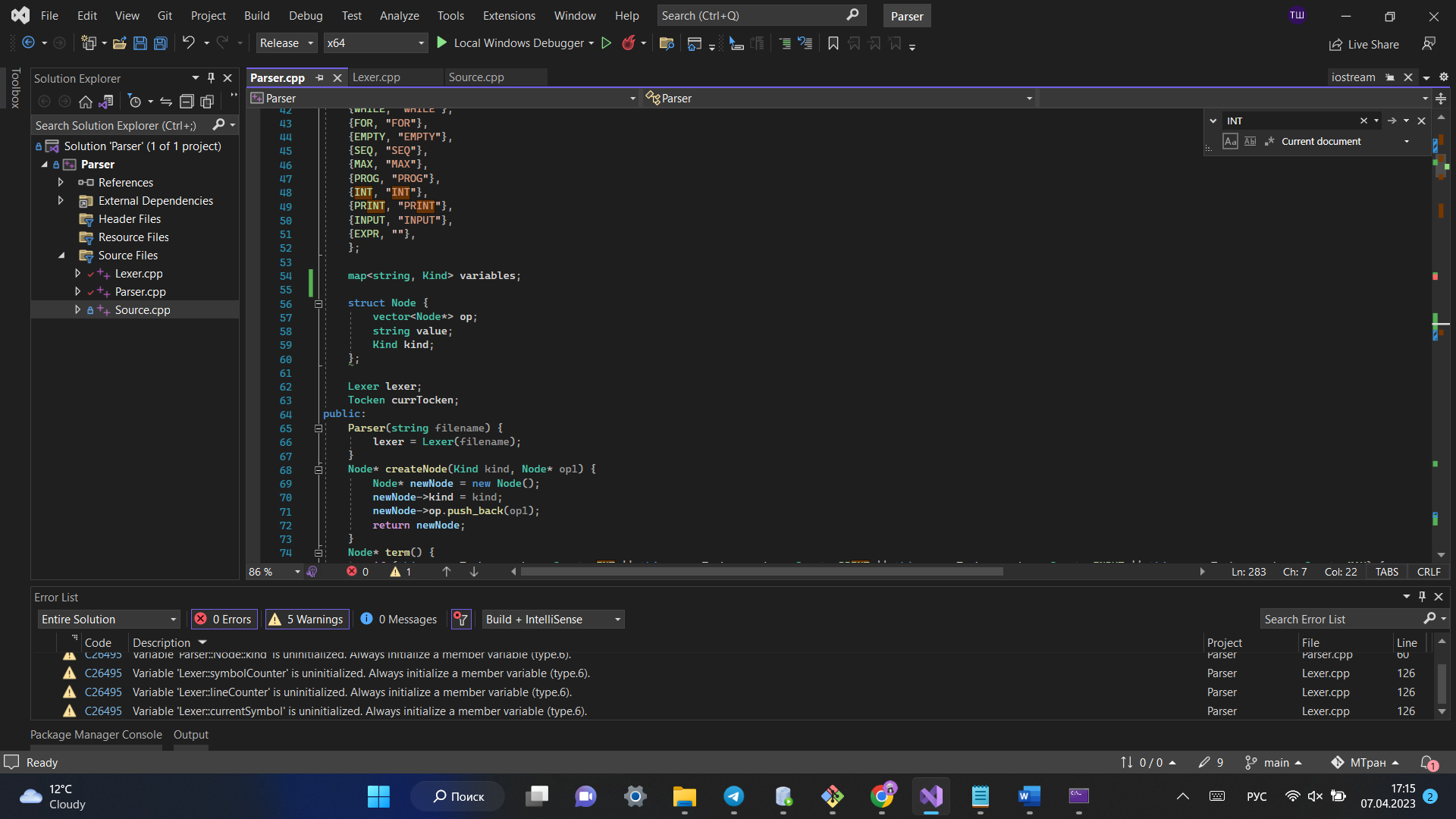


Рисунок 3.1 – Таблица переменных

При необходимости вычислений осуществляются проверки на типы данных, там где это необходимо. При обнаружении несоответствия типов или невозможности проведения операции между ними, программный продукт выводит подробную ошибку и заканчивает свою работу.

## **ДЕМОНСТРАЦИЯ РАБОТЫ**

Результат работы анализатора на рисунке 4.1 при наличии семантических ошибок в коде. При отсутствии каких-либо ошибок программный продукт продолжает свою работу.

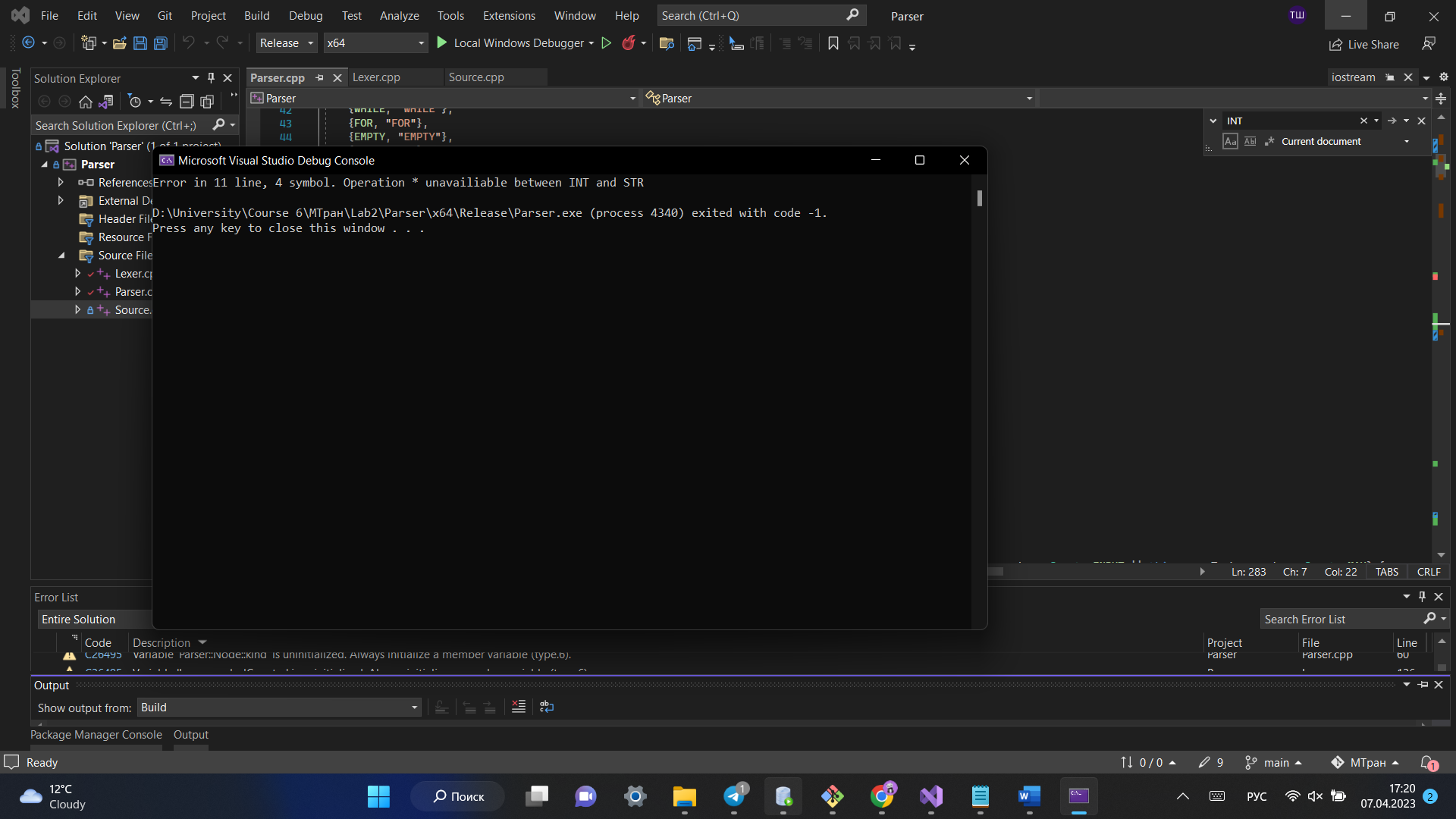


Рисунок 4.1 – Результат работы программного продукта

1. **ДЕМОНСТРАЦИЯ НАХОЖДЕНИЯ ОШИБОК**

На рисунке 5.1 семантическая ошибка в задаче 1 (приложение А): попытка провести недопустимую операцию между типами INT и STR. Одна из переменных имеет строковый тип, а другая целочисленный.

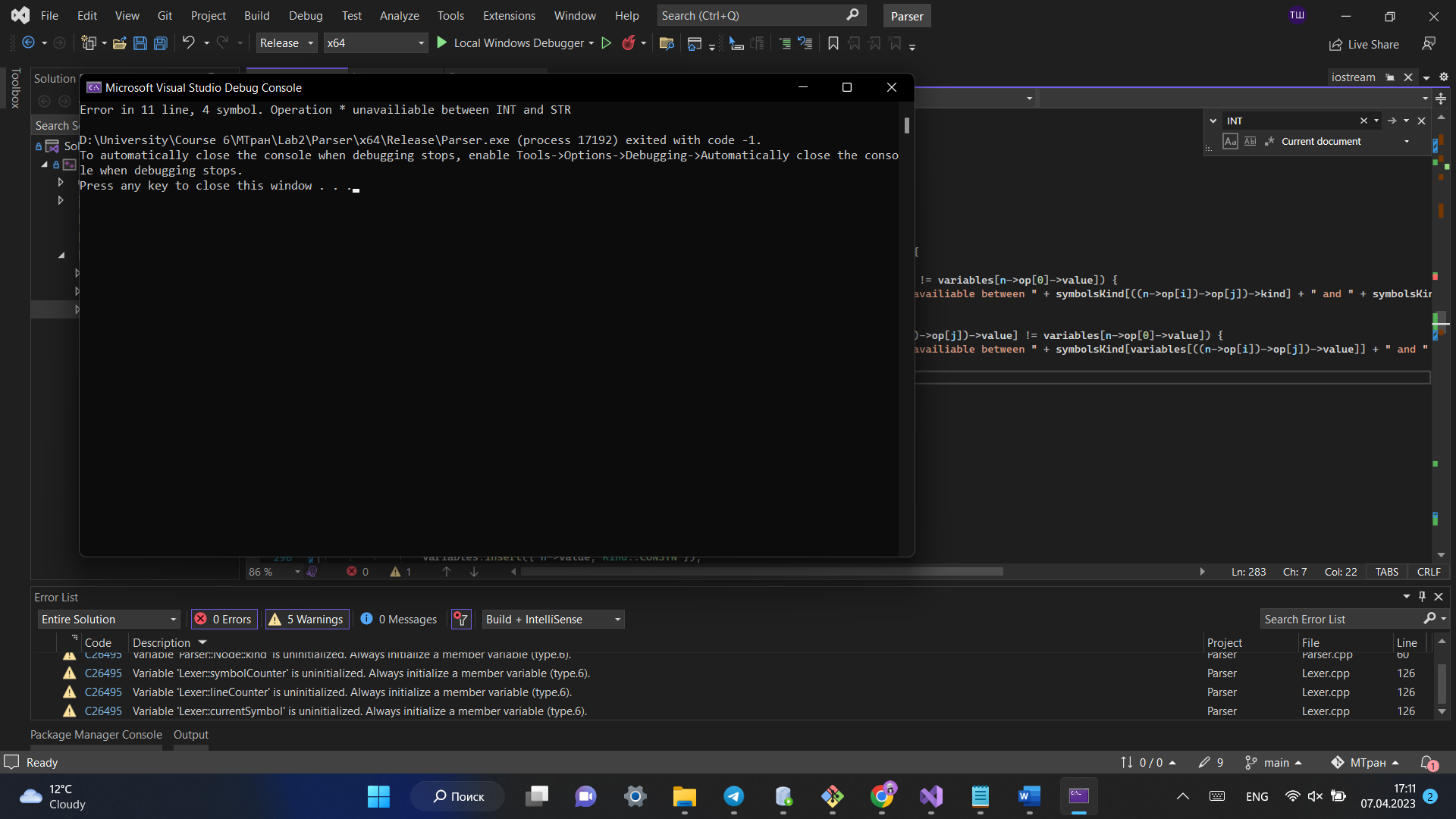


Рисунок 5.1 – Ошибка приведения типов

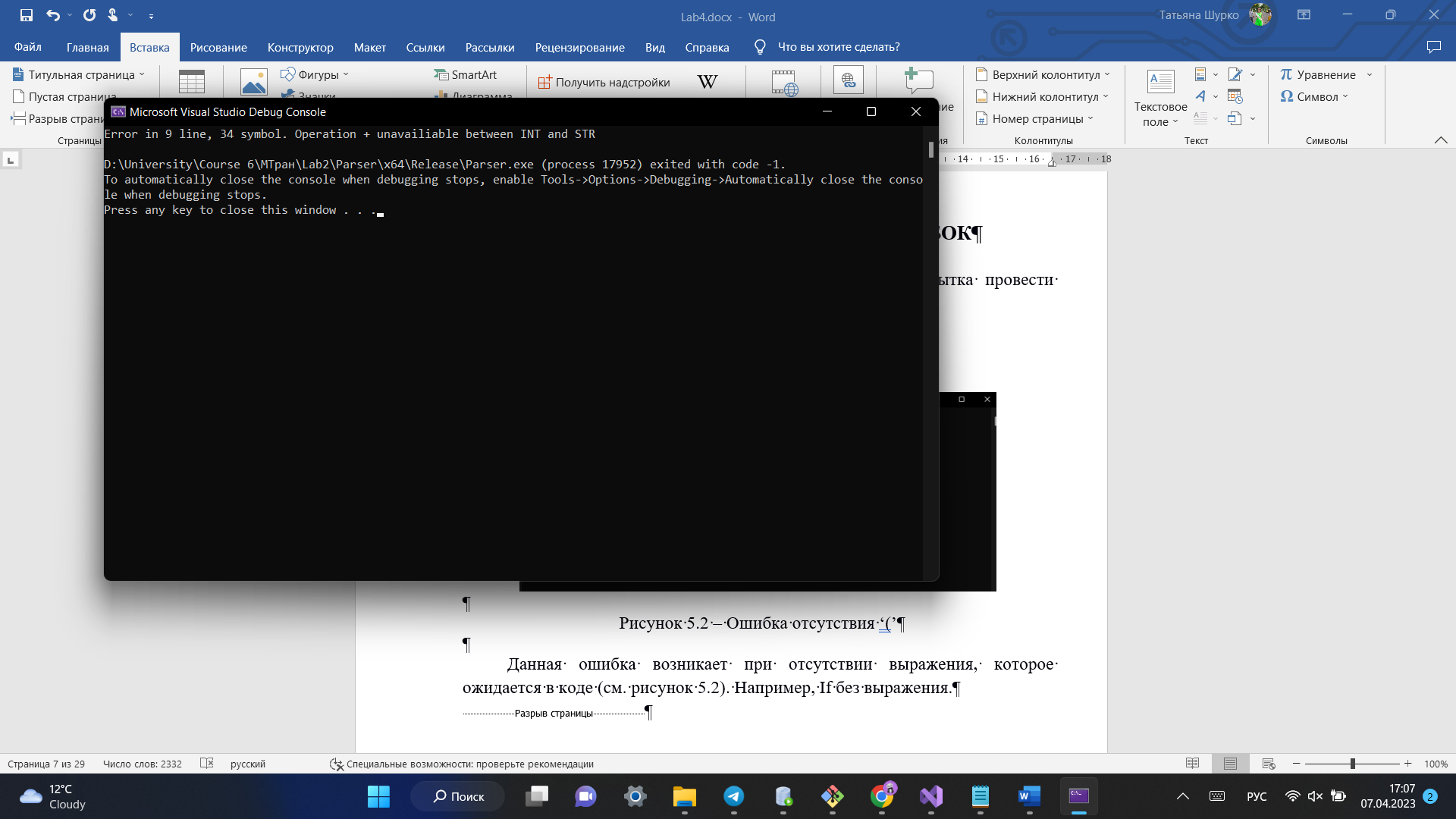


Рисунок 5.2 – Ошибка приведения типов

Данная ошибка возникает при отсутствии приведения введенного числа к типу int в задаче 1 (приложение Б), которое впоследствии прибавляется к целочисленному типу (см. рисунок 5.2). Исходный код задачи в приложении Г.

## **ВЫВОД**

В результате выполнения лабораторной работы была изучена теория о семантических анализаторах, разработан собственный семантический анализатор, выбранного подмножества языка программирования. Определены минимум 2 возможные семантические ошибки и показано их корректное выявление.

**ПРИЛОЖЕНИЕ А КОД ПРОГРАММЫ**

number = int(input("Enter a number: "))

factorial = 'how'

if number < 0:

print("Sorry, factorial does not exist for negative numbers")

elif number == 0:

print("The factorial of 0 is 1")

else:

for i in range(1, number + 1):

factorial = factorial\*i

print(factorial)

**ПРИЛОЖЕНИЕ Б КОД ПРОГРАММЫ**

number = input("Enter a number: ")

factorial = 1

if number < 0:

print("Sorry, factorial does not exist for negative numbers")

elif number == 0:

print("The factorial of 0 is 1")

else:

for i in range(1, number + 1):

factorial = factorial\*i

print(factorial)

**ПРИЛОЖЕНИЕ В ЛИСТИНГ КОДА**

#include <iostream>

#include <vector>

#include <map>

#include "Lexer.cpp"

using namespace std;

class Parser {

private:

enum Kind {

VAR, CONSTN, CONSTD, CONSTS,

ADD, SUB, MULTIPLY, DIVIDE,

PLUSEQ,

LT, LE, GT, GE, EQ,

SET, RANGE,

IF, ELIF, ELSE,

WHILE, FOR,

EMPTY, SEQ, PROG,

INT, PRINT, INPUT, MAX,

EXPR

};

std::map<Kind, string> symbolsKind = {

{VAR, ""},

{CONSTN, "INT"},

{CONSTD, "FLOAT"},

{CONSTS, "STR"},

{ADD, "+"},

{SUB, "-"},

{MULTIPLY, "\*"},

{DIVIDE, "/"},

{LT, "<"},

{LE, "<="},

{GT, ">"},

{GE, ">="},

{EQ, "="},

{SET, "="},

{RANGE, "RANGE"},

{IF, "IF"},

{ELIF, "ELIF"},

{ELSE, "ELSE"},

{WHILE, "WHILE"},

{FOR, "FOR"},

{EMPTY, "EMPTY"},

{SEQ, "SEQ"},

{MAX, "MAX"},

{PROG, "PROG"},

{INT, "INT"},

{PRINT, "PRINT"},

{INPUT, "INPUT"},

{EXPR, ""},

};

map<string, Kind> variables;

struct Node {

vector<Node\*> op;

string value;

Kind kind;

};

Lexer lexer;

Tocken currTocken;

public:

Parser(string filename) {

lexer = Lexer(filename);

}

Node\* createNode(Kind kind, Node\* op1) {

Node\* newNode = new Node();

newNode->kind = kind;

newNode->op.push\_back(op1);

return newNode;

}

Node\* term() {

if (this->currTocken.symb == Const::INT || this->currTocken.symb == Const::PRINT || this->currTocken.symb == Const::INPUT || this->currTocken.symb == Const::MAX) {

return this->parseFunctions();

} else if (this->currTocken.symb == Const::ID) {

Node\* nodeId = new Node();

nodeId->kind = VAR;

nodeId->value = currTocken.value;

this->currTocken = this->lexer.getNextTocken();

return nodeId;

} else if (this->currTocken.symb == Const::NUM) {

Node\* nodeNum = new Node();

nodeNum->kind = CONSTN;

nodeNum->value = this->currTocken.value;

this->currTocken = this->lexer.getNextTocken();

return nodeNum;

} else if (this->currTocken.symb == Const::DNUM) {

Node\* nodeDoubleNum = new Node();

nodeDoubleNum->kind = CONSTD;

nodeDoubleNum->value = this->currTocken.value;

this->currTocken = this->lexer.getNextTocken();

return nodeDoubleNum;

} else if (this->currTocken.symb == Const::STR) {

Node\* nodeStr = new Node();

nodeStr->kind = CONSTS;

nodeStr->value = this->currTocken.value;

this->currTocken = this->lexer.getNextTocken();

return nodeStr;

} else {

return this->parseParentExpression();

}

}

Node\* parseFunctions() {

if (this->currTocken.symb == Const::INT) {

Node\* nodeInt = new Node();

nodeInt->kind = INT;

this->currTocken = this->lexer.getNextTocken();

nodeInt->op.push\_back(this->parseParentExpression());

return nodeInt;

} else if (this->currTocken.symb == Const::PRINT) {

Node\* nodePrint = new Node();

nodePrint->kind = PRINT;

this->currTocken = this->lexer.getNextTocken();

nodePrint->op.push\_back(this->parseParentExpression());

return nodePrint;

} else if (this->currTocken.symb == Const::INPUT) {

Node\* nodeInput = new Node();

nodeInput->kind = INPUT;

this->currTocken = this->lexer.getNextTocken();

nodeInput->op.push\_back(this->parseParentExpression());

return nodeInput;

} else if (this->currTocken.symb == Const::MAX) {

Node\* nodeMAX = new Node();

nodeMAX->kind = MAX;

this->currTocken = this->lexer.getNextTocken();

if (this->currTocken.symb != Const::LPAR) {

this->showError("SYNTAX ERROR '(' EXPECTED!");

}

this->currTocken = this->lexer.getNextTocken();

nodeMAX->op.push\_back(this->parseArithmeticExpression());

if (this->currTocken.symb != Const::COMMA) {

this->showError("EXPECTED COMMA IN MAX EXPRESSION");

}

this->currTocken = this->lexer.getNextTocken();

nodeMAX->op.push\_back(this->parseArithmeticExpression());

if (this->currTocken.symb != Const::RPAR) {

this->showError("SYNTAX ERROR ')' EXPECTED!");

}

this->currTocken = this->lexer.getNextTocken();

return nodeMAX;

}

}

Node\* parseArithmeticExpression() {

Node\* node = this->term();

if (this->currTocken.symb == Const::PLUS || this->currTocken.symb == Const::MINUS ||

this->currTocken.symb == Const::MULTIPLY || this->currTocken.symb == Const::DIVIDE) {

Node\* arithmeticExpression = new Node();

arithmeticExpression->kind = EXPR;

arithmeticExpression->op.push\_back(node);

Kind kindTemp;

while (this->currTocken.symb == Const::PLUS || this->currTocken.symb == Const::MINUS ||

this->currTocken.symb == Const::MULTIPLY || this->currTocken.symb == Const::DIVIDE) {

switch (this->currTocken.symb) {

case Const::PLUS:

kindTemp = ADD;

break;

case Const::MINUS:

kindTemp = SUB;

break;

case Const::MULTIPLY:

kindTemp = MULTIPLY;

break;

case Const::DIVIDE:

kindTemp = DIVIDE;

break;

}

this->currTocken = this->lexer.getNextTocken();

Node\* n = new Node();

n->kind = kindTemp;

n->op.push\_back(this->term());

arithmeticExpression->op.push\_back(n);

}

return arithmeticExpression;

}

return node;

}

Node\* parseCompareExpression() {

this->parseQuote();

Node\* n = this->parseArithmeticExpression();

this->parseQuote();

if (this->currTocken.symb == Const::LESS) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeComp = new Node();

nodeComp->kind = LT;

nodeComp->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodeComp);

} else if (this->currTocken.symb == Const::GREATER) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeComp = new Node();

nodeComp->kind = GT;

nodeComp->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodeComp);

} else if (this->currTocken.symb == Const::GREATEREQUAL) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeComp = new Node();

nodeComp->kind = GE;

nodeComp->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodeComp);

} else if (this->currTocken.symb == Const::LESSEQUAL) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeComp = new Node();

nodeComp->kind = LE;

nodeComp->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodeComp);

} else if (this->currTocken.symb == Const::EQUALEQUAL) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeComp = new Node();

nodeComp->kind = EQ;

nodeComp->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodeComp);

} else if (this->currTocken.symb == Const::NOTEQUAL) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeComp = new Node();

nodeComp->kind = EQ;

nodeComp->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodeComp);

} else if (this->currTocken.symb == Const::PLUSEQUAL) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodePlusEq = new Node();

nodePlusEq->kind = PLUSEQ;

nodePlusEq->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodePlusEq);

}

return n;

}

Node\* parseExpression() {

if (this->currTocken.symb != Const::ID) {

return this->parseCompareExpression();

}

Node\* n = this->parseCompareExpression();

if (n->op.size() > 0 && n->kind == Kind::EXPR) {

for (int i = 0; i < n->op.size(); i++) {

if ((n->op[i])->kind == Kind::ADD || (n->op[i])->kind == Kind::SUB

|| (n->op[i])->kind == Kind::MULTIPLY || (n->op[i])->kind == Kind::DIVIDE) {

for (int j = 0; j < ((n->op[i])->op).size(); j++) {

if (((n->op[i])->op[j])->kind != Kind::VAR && ((n->op[i])->op[j])->kind != variables[n->op[0]->value]) {

this->showError("Operation " + symbolsKind[(n->op[i])->kind] + " unavailiable between " + symbolsKind[((n->op[i])->op[j])->kind] + " and " + symbolsKind[variables[n->op[0]->value]]);

}

else if (((n->op[i])->op[j])->kind == Kind::VAR && variables[((n->op[i])->op[j])->value] != variables[n->op[0]->value]) {

this->showError("Operation " + symbolsKind[(n->op[i])->kind] + " unavailiable between " + symbolsKind[variables[((n->op[i])->op[j])->value]] + " and " + symbolsKind[variables[n->op[0]->value]]);

}

}

//

}

}

}

if (n->kind == VAR && this->currTocken.symb == Const::EQUAL) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeExpr = new Node();

nodeExpr->kind = SET;

nodeExpr->op.push\_back(this->parseExpression());

if (nodeExpr->op.size() > 0 && nodeExpr->op[0]->kind == Kind::INT) {

variables.insert({ n->value, Kind::CONSTN });

}

else if (nodeExpr->op.size() > 0 && (nodeExpr->op[0]->kind == Kind::CONSTD || nodeExpr->op[0]->kind == Kind::CONSTN || nodeExpr->op[0]->kind == Kind::CONSTS)) {

variables.insert({ n->value, nodeExpr->op[0]->kind });

}

}

return n;

}

void parseQuote() {

if (this->currTocken.symb == Const::QUOTE) {

this->currTocken = this->lexer.getNextTocken();

}

}

Node\* parseParentExpression() {

if (this->currTocken.symb != Const::LPAR) {

this->showError("SYNTAX ERROR '(' EXPECTED!");

}

this->currTocken = this->lexer.getNextTocken();

this->parseQuote();

Node\* n = this->parseExpression();

this->parseQuote();

if (this->currTocken.symb != Const::RPAR) {

this->showError("SYNTAX ERROR ')' EXPECTED!");

}

this->currTocken = this->lexer.getNextTocken();

return n;

}

Node\* parseStatementFor() {

if (this->currTocken.symb != Const::RANGE) {

showError("Syntax error! Operator range expected!");

}

Node\* nodeFor = new Node();

nodeFor->kind = RANGE;

this->currTocken = this->lexer.getNextTocken();

if(this->currTocken.symb != Const::LPAR) {

this->showError("SYNTAX ERROR '(' EXPECTED!");

}

this->currTocken = this->lexer.getNextTocken();

Node\* n = this->parseExpression();

nodeFor->op.push\_back(n);

if (this->currTocken.symb == Const::COMMA) {

this->currTocken = this->lexer.getNextTocken();

Node\* n = this->parseExpression();

nodeFor->op.push\_back(n);

}

if (this->currTocken.symb != Const::RPAR) {

this->showError("SYNTAX ERROR ')' EXPECTED!");

}

this->currTocken = this->lexer.getNextTocken();

return nodeFor;

}

Node\* parseStatement() {

Node\* n = new Node();

if (this->currTocken.symb == Const::IF) {

n->kind = Kind::IF;

//if node

Node\* ifNode = new Node();

ifNode->kind = Kind::IF;

this->currTocken = this->lexer.getNextTocken();

//parse expression

if (this->currTocken.symb == Const::LPAR) {

ifNode->op.push\_back(this->parseParentExpression());

}

else {

ifNode->op.push\_back(this->parseExpression());

}

//checks if a colon is present

if (this->currTocken.symb != Const::COLON) {

this->showError("EXPECTED COLON AFTER IF STATEMENT!");

}

//parse statement

this->currTocken = this->lexer.getNextTocken();

while (this->currTocken.symb == Const::TAB) {

this->currTocken = this->lexer.getNextTocken();

ifNode->op.push\_back(this->parseStatement());

}

//append to node

n->op.push\_back(ifNode);

} else if(this->currTocken.symb == Const::ELIF){

//same for (might be more than one)

while (this->currTocken.symb == Const::ELIF) {

//creating node for elif

Node\* elifNode = new Node();

elifNode->kind = Kind::ELIF;

this->currTocken = this->lexer.getNextTocken();

//parse expression

if (this->currTocken.symb == Const::LPAR) {

elifNode->op.push\_back(this->parseParentExpression());

}

else {

elifNode->op.push\_back(this->parseExpression());

}

//checks if a colon is present

if (this->currTocken.symb != Const::COLON) {

this->showError("EXPECTED COLON AFTER IF STATEMENT!");

}

//parse statement

this->currTocken = this->lexer.getNextTocken();

while (this->currTocken.symb == Const::TAB) {

this->currTocken = this->lexer.getNextTocken();

elifNode->op.push\_back(this->parseStatement());

}

//append to node

n->op.push\_back(elifNode);

}

//same for else

}

else if (this->currTocken.symb == Const::ELSE) {

//new node for else

Node\* elseNode = new Node();

elseNode->kind = Kind::ELSE;

this->currTocken = this->lexer.getNextTocken();

//checks if a colon is present

if (this->currTocken.symb != Const::COLON) {

this->showError("EXPECTED COLON AFTER IF STATEMENT!");

}

//parse statement

this->currTocken = this->lexer.getNextTocken();

while (this->currTocken.symb == Const::TAB) {

this->currTocken = this->lexer.getNextTocken();

elseNode->op.push\_back(this->parseStatement());

}

//append to node

n->op.push\_back(elseNode);

}

else if (this->currTocken.symb == Const::WHILE) {

n->kind = Kind::WHILE;

this->currTocken = this->lexer.getNextTocken();

//parse expression(might be in brackets)

if (currTocken.symb == Const::LPAR) {

n->op.push\_back(this->parseParentExpression());

}

else {

n->op.push\_back(this->parseExpression());

}

//checks if a colon is present

if (this->currTocken.symb != Const::COLON) {

this->showError("EXPECTED COLON AFTER IF STATEMENT!");

}

this->currTocken = this->lexer.getNextTocken();

while (this->currTocken.symb == Const::TAB) {

this->currTocken = this->lexer.getNextTocken();

}

n->op.push\_back(this->parseStatement());

}

else if (this->currTocken.symb == Const::FOR) {

n->kind = Kind::FOR;

this->currTocken = this->lexer.getNextTocken();

//cannot be in brackets

//maybe there shoud be different parse function for expr

n->op.push\_back(this->term());

variables.insert({ n->op[0]->value, Kind::CONSTN });

if (this->currTocken.symb != Const::IN) {

showError("Syntax error. Operator IN expected!");

}

this->currTocken = this->lexer.getNextTocken();

n->op.push\_back(this->parseStatementFor());

this->currTocken = this->lexer.getNextTocken();

while (this->currTocken.symb == Const::TAB) {

this->currTocken = this->lexer.getNextTocken();

n->op.push\_back(this->parseStatement());

}

}

else if (this->currTocken.symb == Const::SEMICOLON) {

//; do nothing

n->kind = EMPTY;

this->currTocken = this->lexer.getNextTocken();

}

else if (this->currTocken.symb == Const::LBRA) {

// block { }

n->kind = EMPTY;

this->currTocken = this->lexer.getNextTocken();

while (this->currTocken.symb != Const::RBRA) {

n->kind = SEQ;

n->op.push\_back(n);

n->op.push\_back(parseStatement());

}

this->currTocken = this->lexer.getNextTocken();

}

else {

n->kind = EXPR;

if (this->currTocken.symb == Const::TAB) {

while (this->currTocken.symb == Const::TAB) {

this->currTocken = this->lexer.getNextTocken();

}

}

n->op.push\_back(this->parseExpression());

}

return n;

}

Node\* Parse() {

currTocken = this->lexer.getNextTocken();

Node\* node = createNode(PROG, this->parseStatement());

while (this->currTocken.symb != Const::EF) {

node->op.push\_back(this->parseStatement());

}

cout << "Show tree: " << endl;

showTree(node);

return node;

}

string tabs = "\t";

void showTree(Node\* root) {

if (root->value != "") {

cout << tabs << root->value << endl;

}

for (int i = 0; i < root->op.size(); i++) {

cout << tabs << symbolsKind[root->kind] << endl;

tabs += '\t';

if (root->op.size() == 0) {

}

showTree(root->op[i]);

tabs.pop\_back();

}

}

void showError(string message) {

cout << "Error in "<< lexer.getLine() << " line, " <<lexer.getSymbolCounter() << " symbol. " << message << endl;

exit(-1);

}

int getLine() {

return lexer.getLine();

}

int getSymbolCounter() {

return lexer.getSymbolCounter();

}

};

**ПРИЛОЖЕНИЕ Г ТЕКСТ ПРОГРАММ**

1. Нахождения n чисел Фибоначчи, введенного пользователем

n = int(input("Enter positive number: "))  
  
if n <= 0:  
 print("You enter negative number or 0")  
else:  
 num\_1 = 0  
 num\_2 = 1  
   
 print(num\_2)  
   
 for i in range(n-1):  
 res = num\_1 + num\_2  
 num\_1 = num\_2  
 num\_2 = res  
   
 print(num\_2)

1. Дана последовательность натуральных чисел, завершающаяся числом 0. Определите, какое наибольшее число подряд идущих элементов этой последовательности равны друг другу.

prev = -1  
curr\_rep\_len = 0  
max\_rep\_len = 0  
  
element = int(input())  
  
while element != 0:  
 if prev == element:  
 curr\_rep\_len += 1  
 else:  
 prev = element  
 max\_rep\_len = max(max\_rep\_len, curr\_rep\_len)  
 curr\_rep\_len = 1  
   
 element = int(input())  
   
max\_rep\_len = max(max\_rep\_len, curr\_rep\_len)  
print(max\_rep\_len)

1. Вычислить факториал введенного пользователем числа.

number = int(input("Enter a number: "))  
factorial = 1  
  
if number < 0:  
 print("Sorry, factorial does not exist for negative numbers")  
elif number == 0:  
 print("The factorial of 0 is 1")  
else:  
 for i in range(1, number + 1):  
 factorial = factorial\*i  
 print(factorial)