Министерство транспорта Российской Федерации

Федеральное агентство железнодорожного транспорта

Федеральное государственное бюджетное образовательное учреждение   
высшего образования

«Дальневосточный государственный университет путей сообщения»

Кафедра «Вычислительная техника и компьютерная графика»

Разработка транслятора

Отчет к лабораторным работам

ЛР 09.03.01.ЛИПО.01.04.000 – 943

Выполнил

студент П.А. Громов

подпись, дата

Проверил

преподаватель П.С. Тимош

подпись, дата

Хабаровск 2018

1. Цель работы

Основной целью лабораторных работ по разработке трансляторов является получение практических навыков, позволяющих разрабатывать трансляторы, как языков программирования, так специализированных языков САПР. Заданные для реализации языки подобны очень простым языкам программирования.

Входной язык

<Программа> ::= <Объявление переменных> <Описание вычислений>

<Описание вычислений> ::= <Список присваиваний>

<Объявление переменных> ::= Var <Список переменных> ;

<Список переменных> ::= <Идент> | <Идент> , <Список переменных>

<Список присваиваний> ::= <Присваивание>|

<Присваивание> <Список присваиваний>

<Присваивание> ::=<Идент> := <Выражение>;

<Выражение> ::= <Ун.оп><Подвыражение>|<Подвыражение>

<Подвыражение> :: = (<Выражение> ) <Операнд> |

<Подвыражение > <Бин.оп.> <Подвыражение>

<Ун.оп .>::= "-"

<Бин.оп.> ::= "-" | " + " | "\*" | "/"

<Операнд> ::= <Идент>|<Константа>

<Идент> ::= <Буква><Идент>|<Буква>

<Константа> ::= <Цифра><Константа>|<Цифра>

Выходной язык:

<Программа> ::= <Объявление переменных> <Описание вычислений>

<Описание вычислений> ::= [<Список присваиваний>]

<Объявление переменных> ::= Var <Список переменных>;

<Список переменных> ::= <Идент>|<Идент>, <Список переменных>

<Список присваиваний>::= <Присваивание> |

<Присваивание> <Список присваиваний>

<Присваивание> ::= <Идент> = <Выражение>;

<Выражение> ::= <Ун.оп.> <Подвыражение>|<Подвыражение>

<Подвыражение> :: = ( <Выражение> ) | <Операнд>|

<Подвыражение> <Бин.оп.> <Подвыражение>

<Ун.оп.> ::="-"

<Бин.оп.> ::="-" | "+" | "\*" | "/"

<Операнд> ::= <Идент><Константа>

<Идент> ::= <Буква><Идент>|<Буква>

<Константа> ::= <Цифра><Константа>|<Цифра>

1. Лексический анализатор

Порядок выполнения работы:

1. По варианту задания определить, какие классы лексем будут в вашем языке.
2. Составить контрольные примеры на реализуемом языке. Хотя бы один пример должен проверять поведение вашей программы при наличии недопустимых символов в транслируемом файле.
3. Запрограммировать и отладить модуль сканирования. Выполнить тестирование на контрольных примерах. Результатом работы должна быть таблица, содержащая лексемы и признаки их классов. Необходимо включить в результирующий файл информацию о номерах строк исходного текста транслируемой программы.

Классы лексем

|  |  |
| --- | --- |
| SERVICE\_WORD | Var |
| IDENT | a-zA-Z |
| CONST | 0-9 |
| BINARY | - |
| UNARY | +, -, \*, / |
| COMMA | , |
| SEMICOLON | ; |
| ASSIGMENT | : |
| BRACKET\_OPENING | ( |
| BRACKET\_CLOSING | ) |
| NONE |  |

1. Синтаксический анализатор

Порядок выполнения работы:

1. По варианту задания построить синтаксический граф для реализуемого языка. При этом преобразовать описание языка таким образом, чтобы оно учитывало приоритет операций в выражениях.
2. Составить контрольные примеры на реализуемом языке. Хотя бы один пример должен проверять поведение вашей программы при наличии синтаксических ошибок в контрольном примере.
3. Запрограммировать и отладить программу, производящую синтаксический анализ реализуемого языка. Выполнить тестирование на контрольных примерах. При этом пример пропускается через программу лексического анализа, а файл с лексемами является входным для программы синтаксического анализа. При необходимости доработать модуль сканирования.
4. Семантический анализатор

Порядок выполнения работы:

1. Для контрольных примеров на реализуемом языке вручную составить соответствующие им программы на целевом языке.
2. Запрограммировать и отладить программу, производящую семантический анализ реализуемого языка. Выполнить тестирование на контрольных примерах. При этом пример пропускается через программы лексического и синтаксического анализа. При необходимости доработать программы лексического и синтаксического анализа.
3. Тестирование
   1. Пример первый

Входные данные:

Var a, b, c;

a := -(((1\*5))) + 6 / 3;

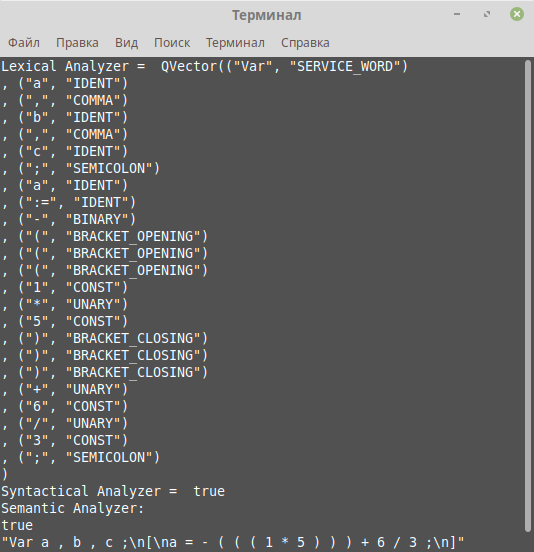


Рисунок 1 – Тест первый

Выходные данные:

Var a , b , c ;

[

a = - ( ( ( 1 \* 5 ) ) ) + 6 / 3 ;

]

* 1. Пример второй

Var

a, b, c, d;

q := 1;

w := 2;

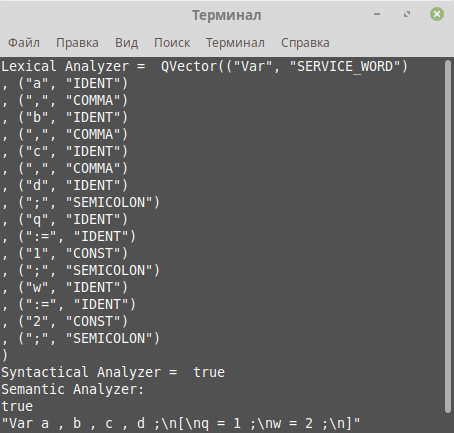


Рисунок 2 – Тест второй

Var a , b , c , d ;

[

q = 1 ;

w = 2 ;

]

* 1. Пример третий

Var .a1,a2:=Vara3a4=bc , qwe;

abc:=-532\*3/434+3;

qwe := abc ;

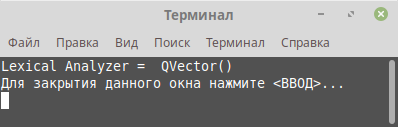


Рисунок 3 – Тест третий

Вывод

В ходе выполнения лабораторных работ, был написан транслятор, который переводит входной в выходной язык. Трансляция происходит путем трех анализов: лексического, синтаксического и семантического. Каждый из анализаторов, обладает своим функционалом и исполняет свою задачу.

Приложение А

(обязательное)

Lexem.h

#ifndef LEXEM\_H

#define LEXEM\_H

#include <QString>

#include <QDebugStateSaver>

class Lexem

{

public:

enum Types

{

SERVICE\_WORD,

IDENT,

CONST,

BINARY,

UNARY,

COMMA,

SEMICOLON,

ASSIGMENT,

BRACKET\_OPENING,

BRACKET\_CLOSING,

NONE

};

Lexem();

Lexem(QString name,

Types type);

QString **getName**() const;

Types **getType**() const;

QString **getTypesName**() const;

bool **operator** == (enum Types type);

bool **operator** == (const Lexem lexem);

bool **operator** != (enum Types type);

bool **operator** != (const Lexem lexem);

friend QDebug **operator** <<(QDebug debug, const Lexem lexem);

private:

QString name = "";

Types type = Lexem::NONE;

};

QDebug **operator** <<(QDebug debug, const Lexem lexem);

#endif // LEXEM\_H

Lexem.cpp

#include "lexem.h"

Lexem::**Lexem**():

name(), type(Lexem::NONE)

{

}

Lexem::**Lexem**(QString name, Lexem::Types type):

name(name), type(type)

{

}

QString Lexem::**getName**() const

{

return this->name;

}

Lexem::Types Lexem::**getType**() const

{

return this->type;

}

QString Lexem::**getTypesName**() const

{

switch (type)

{

case SERVICE\_WORD:

return "SERVICE\_WORD";

case IDENT:

return "IDENT";

case CONST:

return "CONST";

case BINARY:

return "BINARY";

case UNARY:

return "UNARY";

case COMMA:

return "COMMA";

case SEMICOLON:

return "SEMICOLON";

case ASSIGMENT:

return "ASSIGMENT";

case BRACKET\_OPENING:

return "BRACKET\_OPENING";

case BRACKET\_CLOSING:

return "BRACKET\_CLOSING";

default:

return "NONE";

}

}

bool Lexem::**operator** ==(Lexem::Types type)

{

return this->type == type;

}

bool Lexem::**operator** ==(const Lexem lexem)

{

return this->type == lexem.type;

}

bool Lexem::**operator** !=(Lexem::Types type)

{

return this->type != type;

}

bool Lexem::**operator** !=(const Lexem lexem)

{

return this->type != lexem.type;

}

QDebug **operator** <<(QDebug debug, const Lexem lexem)

{

QDebugStateSaver saver(debug);

debug.nospace() << "(" << lexem.getName() << ", " << lexem.getTypesName() << ")" << "\n";

return debug;

}

LexicalAnalyzer.h

#ifndef LEXICALANALYZER\_H

#define LEXICALANALYZER\_H

#include <QFile>

#include <QString>

#include <QVector>

#include <QDebug>

#include "lexem.h"

class LexicalAnalyzer

{

public:

LexicalAnalyzer();

LexicalAnalyzer(QString path);

void **readFileToString**();

void **setPath**(QString path);

QVector<Lexem> **run**();

QVector<Lexem> **getLexemBox**();

private:

bool **isConst**(QString word);

bool **isIdent**(QString word);

bool **isSigns**(QString word);

bool **isUnary**(QString word);

const QVector<QString> serviceWords = {"Var"};

const QVector<QString> unaryOperations = {"+", "-", "\*", "/"};

const QVector<QString> signs = {",", ";", ":", "(", ")"};

const QVector<QString> stringSymbols = {" ", "\n", "\r"};

bool printFlag = true;

int index = 0;

QString text;

QString path = "";

QVector<Lexem> lexemBox;

};

#endif // LEXICALANALYZER\_H

LexicalAnalyzer.cpp

#include "lexicalanalyzer.h"

LexicalAnalyzer::**LexicalAnalyzer**()

{

this->path = "";

}

LexicalAnalyzer::**LexicalAnalyzer**(QString path)

{

this->path = path;

}

bool LexicalAnalyzer::**isConst**(QString word)

{

foreach (QChar symbol, word)

if (!symbol.isDigit())

return false;

return true;

}

bool LexicalAnalyzer::**isIdent**(QString word)

{

foreach (QChar symbol, word)

if ((symbol < QChar('A') || symbol > QChar('Z')) &&

(symbol < QChar('a') || symbol > QChar('z')))

return false;

return true;

}

bool LexicalAnalyzer::**isUnary**(QString word)

{

return unaryOperations.contains(word);

}

bool LexicalAnalyzer::**isSigns**(QString word)

{

return signs.contains(word);

}

void LexicalAnalyzer::**readFileToString**()

{

QString string;

QFile file(path);

if (file.*open*(QIODevice::ReadOnly))

string.append(file.readAll());

this->text = string;

}

QVector<Lexem> LexicalAnalyzer::**getLexemBox**()

{

return this->lexemBox;

}

void LexicalAnalyzer::**setPath**(QString path)

{

this->path = path;

}

QVector<Lexem> LexicalAnalyzer::**run**()

{

readFileToString();

while (index < text.length())

{

Lexem::Types type;

QString word;

QString symbol = text.at(index);

// IDENT

if (isIdent(symbol))

{

while (true)

{

if (isIdent(text.at(index)))

{

word.append(text.at(index));

index++;

}

else

{

index--;

break;

}

}

symbol = word;

if (serviceWords.contains(symbol))

type = Lexem::SERVICE\_WORD;

else

type = Lexem::IDENT;

}

// CONST

else if (isConst(symbol))

{

while (true)

{

if (isConst(text.at(index)))

{

word.append(text.at(index));

index++;

}

else

{

index--;

break;

}

}

symbol = word;

type = Lexem::CONST;

}

// UNARY and BINARY OPERATIONS

else if (unaryOperations.contains(symbol))

{

if ((lexemBox.last().getName() == ":=") || (lexemBox.last().getName() == "("))

type = Lexem::BINARY;

else

type = Lexem::UNARY;

}

// SIGNS

else if (signs.contains(symbol))

{

if (text.at(index+1) == "=")

{

symbol.append(text.at(index+1));

index++;

}

if (symbol == ",")

type = Lexem::COMMA;

else if (symbol == ";")

type = Lexem::SEMICOLON;

else if (symbol == "(")

type = Lexem::BRACKET\_OPENING;

else if (symbol == ")")

type = Lexem::BRACKET\_CLOSING;

}

// CHECK (" ", "\n", "\r")

else if (stringSymbols.contains(symbol))

{

// empty

}

// ERROR

else

{

qDebug() << "Error";

lexemBox.clear();

break;

}

// Add LEXEM to box

if (symbol != ' ' && symbol != '\n' && symbol != '\r')

lexemBox.append(Lexem(symbol, type));

// NEXT ITERATOR

index++;

}

return lexemBox;

}

SyntacticalAnalyzer.h

#ifndef SYNTACTICALANALYZER\_H

#define SYNTACTICALANALYZER\_H

#include <QFile>

#include <QVector>

#include "lexicalanalyzer.h"

class SyntacticalAnalyzer

{

public:

SyntacticalAnalyzer();

SyntacticalAnalyzer(QVector<Lexem> lexemBox);

SyntacticalAnalyzer(LexicalAnalyzer lexicalAnalyzer);

bool **declarationOfVariables**();

bool **variableList**();

bool **subExpression**();

void **expression**();

void **setLexemBox**(QVector<Lexem> lexemBox);

void **setIterator**(QVector<Lexem>::iterator iterator);

private:

QVector<Lexem> lexemBox;

QVector<Lexem>::iterator iterator;

};

#endif // SYNTACTICALANALYZER\_H

SyntacticAnalyzer.cpp

#include "syntacticalanalyzer.h"

SyntacticalAnalyzer::**SyntacticalAnalyzer**()

{

}

SyntacticalAnalyzer::**SyntacticalAnalyzer**(QVector<Lexem> lexemBox):

lexemBox(lexemBox)

{

}

SyntacticalAnalyzer::**SyntacticalAnalyzer**(LexicalAnalyzer lexicalAnalyzer):

lexemBox(lexicalAnalyzer.getLexemBox())

{

}

bool SyntacticalAnalyzer::**declarationOfVariables**()

{

this->iterator = lexemBox.begin();

// первый блок - объявление переменных

if (\*iterator == Lexem::SERVICE\_WORD)

{

this->iterator++;

variableList();

this->iterator++;

// второй блок - описание вычислений

while (iterator != lexemBox.end())

{

if (\*iterator == Lexem::IDENT)

{

this->iterator++;

if (\*iterator == Lexem::ASSIGMENT)

{

this->iterator++;

expression();

}

}

this->iterator++;

}

return true;

}

else

return false;

}

bool SyntacticalAnalyzer::**variableList**()

{

if (\*iterator == Lexem::IDENT)

{

this->iterator++;

if(\*iterator == Lexem::COMMA) // ,

{

this->iterator++;

return variableList();

}

else if (\*iterator == Lexem::SEMICOLON) // ;

return true;

else

return false;

}

else

return false;

}

void SyntacticalAnalyzer::**expression**()

{

if (\*iterator == Lexem::UNARY)

this->iterator++;

subExpression();

}

bool SyntacticalAnalyzer::**subExpression**()

{

if (\*iterator == Lexem::BRACKET\_OPENING) // (

{

// вызываю выражение

this->iterator++;

expression();

this->iterator++;

if (\*iterator != Lexem::BRACKET\_CLOSING) // )

{

return false;

}

}

else if (\*iterator == Lexem::IDENT || \*iterator == Lexem::CONST)

{

this->iterator++;

if (\*iterator == Lexem::SEMICOLON)

{

return true;

}

else if (\*iterator == Lexem::BINARY)

{

this->iterator++;

return subExpression();

}

}

if(\*iterator == Lexem::SEMICOLON)

{

return true;

}

else

return false;

}

void SyntacticalAnalyzer::**setLexemBox**(QVector<Lexem> lexemBox)

{

this->lexemBox = lexemBox;

}

void SyntacticalAnalyzer::**setIterator**(QVector<Lexem>::iterator iterator)

{

this->iterator = iterator;

}

SemanticAnalyzer.h

#ifndef SEMANTICANALYZER\_H

#define SEMANTICANALYZER\_H

#include "lexicalanalyzer.h"

class SemanticAnalyzer

{

public:

SemanticAnalyzer();

SemanticAnalyzer(QVector<Lexem> lexemBox);

SemanticAnalyzer(LexicalAnalyzer lexicalAnalyzer);

void **toForm**();

private:

QVector<Lexem> lexemBox;

QVector<Lexem>::iterator iterator;

};

#endif // SEMANTICANALYZER\_H

SemanticAnalyzer.cpp

#include "semanticanalyzer.h"

SemanticAnalyzer::**SemanticAnalyzer**()

{

}

SemanticAnalyzer::**SemanticAnalyzer**(QVector<Lexem> lexemBox):

lexemBox(lexemBox)

{

}

SemanticAnalyzer::**SemanticAnalyzer**(LexicalAnalyzer lexicalAnalyzer):

lexemBox(lexicalAnalyzer.getLexemBox())

{

}

void SemanticAnalyzer::**toForm**()

{

QString text;

bool flag = false;

int counter = -1;

foreach (Lexem lexem, lexemBox)

{

if (lexem.getName() == ":=")

text.append("= ");

else if (lexem.getName() != ";")

text.append(lexem.getName()).append(" ");

else

{

text.append(";\n");

counter++;

if (counter == 0)

text.append("[\n");

}

}

text.append("]");

QFile file("/home/pagrom/university/linguisticProgramming/MainConsole/resourses/out\_1.txt");

if (file.*open*(QIODevice::WriteOnly | QIODevice::Text))

{

qDebug() << file.isOpen();

QTextStream writeStream(&file);

writeStream << text;

file.*close*();

}

qDebug() << text;

}

Main.cpp

#include <QCoreApplication>

#include <QFile>

#include <QDebug>

#include "lexicalanalyzer.h"

#include "syntacticalanalyzer.h"

#include "semanticanalyzer.h"

const QString PATH\_TO\_FILE = ":/input/resourses/test\_.txt";

int **main**(int argc, char \*argv[])

{

QCoreApplication app(argc, argv);

// INIT LEXICAL ANALYZER

LexicalAnalyzer lexicalAnalyzer(PATH\_TO\_FILE);

// INIT SYNTACTICAL ANALYZER

SyntacticalAnalyzer syntacticalAnalyzer(lexicalAnalyzer.run());

// INIT SEMANTIC ANALYZER

SemanticAnalyzer semanticAnalyzer(lexicalAnalyzer.run());

// DEBUG

qDebug() << "Lexical Analyzer = " << lexicalAnalyzer.getLexemBox();

qDebug() << "Syntactical Analyzer = " << syntacticalAnalyzer.declarationOfVariables();

qDebug() << "Semantic Analyzer: ";

semanticAnalyzer.toForm();

return app.exec();

}