|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ | | | | | | | | | | | | |
|  | | | | | |  | | |  | | | |
| ФГБОУ ВО «Пермский государственный национальный исследовательский университет» | | | | | | | | | | | | |
|  | | | | |  | | | | |  | | |
|  | | ОТЧЕТ  По разработке компилятора для языка Pascal  Вариант 11 | | | | | | | | |  | |
|  | | | |  | | | | | |  | | |
|  | Работу выполнил  студент гр. ПМИ-3  Субботин И.В.  (подпись)  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2022 | | | | | |  | Проверил  Преподаватель кафедры МОВС  Пономарев Ф.А.  (подпись)  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2022 | | | |  |
|  | | | | | | | | | | | | |
|  | | | Пермь 2022 | | | | | | |  | | |

СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 2](#_Toc101905891)

[1 Постановка задачи 3](#_Toc101905892)

[2 Структура компилятора 5](#_Toc101905893)

[3 Описание языка Pascal 8](#_Toc101905894)

[4 Модуль ввода-вывода 10](#_Toc101905895)

[4.1 Модуль ввода 10](#_Toc101905896)

[4.1.1 Проектирование 10](#_Toc101905897)

[4.1.2 Тестирование 10](#_Toc101905898)

[4.2 Модуль вывода 14](#_Toc101905899)

[5 Лексический анализатор 15](#_Toc101905900)

[5.1 Проектирование 15](#_Toc101905901)

[5.1.1 Токены 15](#_Toc101905902)

[5.1.2 Обработка ошибок 17](#_Toc101905903)

[5.1.3 Лексер 17](#_Toc101905904)

[5.2 Тестирование 19](#_Toc101905905)

[6 Синтаксический анализатор 23](#_Toc101905906)

[6.1 Используемые BNF 23](#_Toc101905907)

[6.2 Тестирование 24](#_Toc101905908)

[7 Семантический анализатор 27](#_Toc101905909)

[8 Генерация кода 28](#_Toc101905910)

1 Постановка задачи

Глобальное задание: написать компилятор для подмножества языка Паскаль. Задание разбивается на отдельные этапы:

1. Модуль ввода-вывода (8 баллов, оценивается совместно с лексическим анализатором).
2. Лексический анализатор (12 баллов, оценивается совместно с модулем ввода-вывода).
3. Синтаксический анализатор (12 баллов) с нейтрализацией синтаксических ошибок (8 баллов).
4. Семантический анализатор с нейтрализацией семантических ошибок (20 баллов).
5. Генерация кода (25 баллов).

Для получения минимального проходного балла необходимо реализовать указанные этапы для подмножества языка Паскаль, описанного далее в разделе «*Общая минимальная часть*».

Для получения 70% баллов за анализаторы и 100% баллов за генерацию кода необходимо дополнительно реализовать конструкции, описанные далее в разделе «*Общая дополнительная часть*».

Для получения 100% баллов за анализаторы необходимо дополнительно реализовать анализ конструкций, описанных далее в разделе «*Индивидуальная часть*», в соответствии с заданным вариантом.

В случае не сдачи студентом задания по генерации кода на последнем лабораторном занятии (или ранее), данное задание для данного студента заменяется теоретическим экзаменом, который оценивается исходя из максимума в 25 баллов.

Замена задания по генерации на теоретический экзамен возможна также в случае, если студента не устраивают баллы, полученные за задание по генерации. В этом случае набранные баллы обнуляются, и студент сдает экзамен на тех же условиях, что и студенты, не сдававшие задание по генерации кода вообще.

*Общая минимальная часть.* Основные разделы программы: раздел описания переменных, раздел операторов. Переменные стандартных типов (Boolean, integer, real, char). Числовые константы. Арифметическое выражение (в выражении допустимы только константы, переменные, операции +, –, \*, / и скобки). Оператор присваивания и составной оператор.

*Общая дополнительная часть.* Раздел описания типов. Выражение (полностью, включая арифметические, логические операции, сравнения и т.д., но только над константами и простыми переменными (не индексированные, не поля записи, не указатели)). Условный оператор (if). Оператор цикла с предусловием (while).

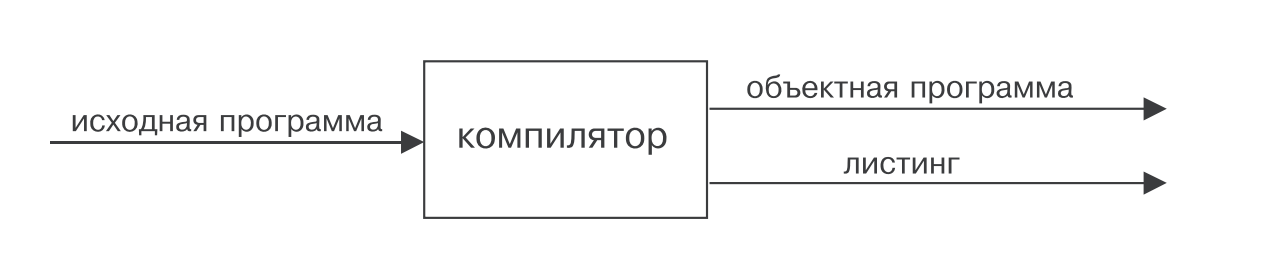
*Индивидуальная часть.*

Ссылочные типы данных. Описание функций. Вызов функции и указатели в выражении.

2 Структура компилятора

Компилятор – это программа, которая переводит программу на языке высокого уровня в эквивалентную программу на другом (объектном) языке. Обычно также выдает листинг, содержащий текст исходной программы и сообщения обо всех обнаруженных ошибках.

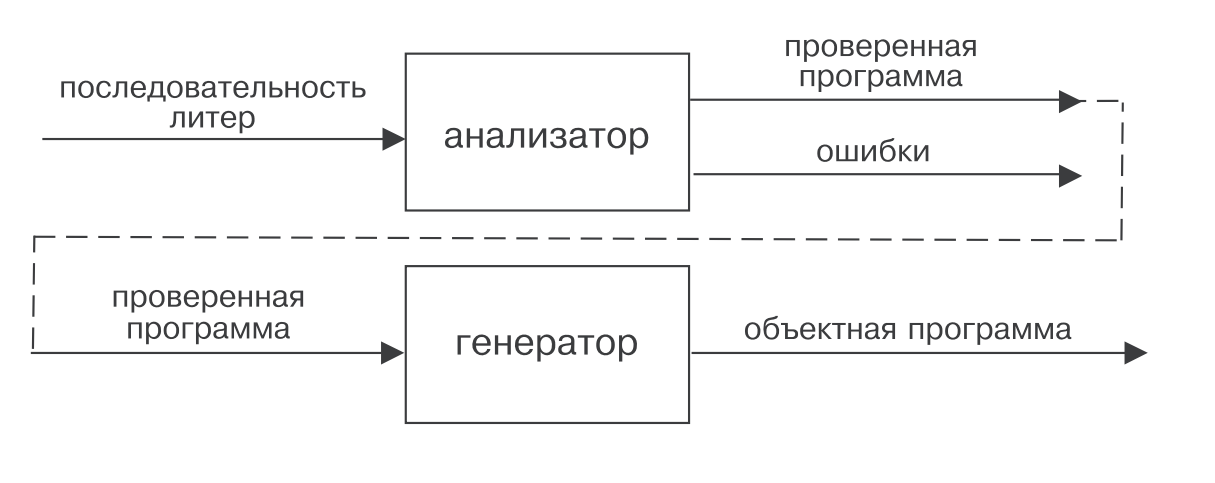
Компилятор можно представить в виде следующей схемы:



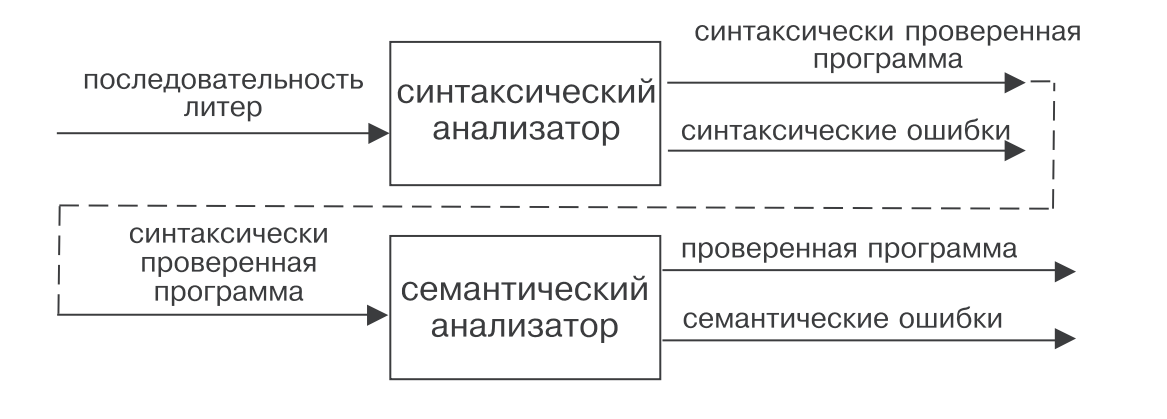
Работу компилятора можно разделить на 2 этапа:

1. Анализ – определение правильности исходной программы и формирование (в случае необходимости) сообщений об ошибках;
2. Синтез – генерация объектной программы; этот этап выполняется для программ, не содержащих ошибок.

Таким образом компилятор разбивается на следующие модули:



Анализатор в свою очередь можно разбиваться на синтаксический и семантический анализаторы:



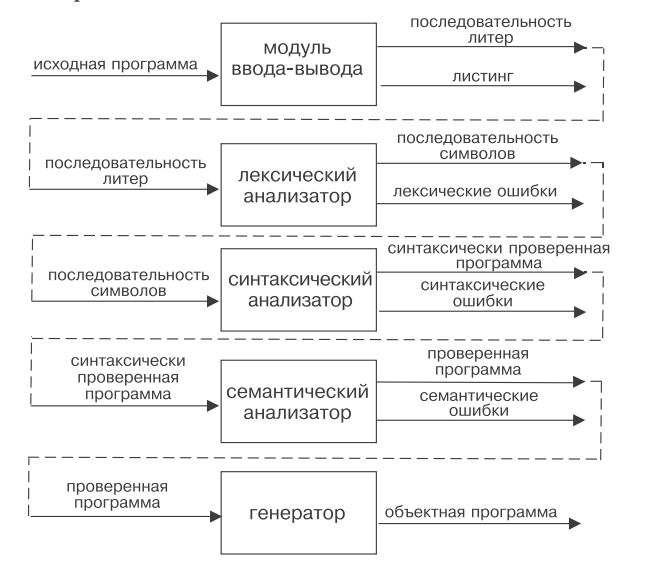
Синтаксический анализатор проверяет, удовлетворяет ли программа формальным правилам. Назначение же семантического анализатора состоит в том, чтобы выяснить, не нарушены ли неформальные правила описания языка.

Дальнейшее разбиение на модули выполняется внутри синтаксического анализатора.

Первый модуль – Лексический анализатор (Lexer) просматривает символы исходной программы и строит символы (лексемы) – идентификаторы, ключевые слова и константы.

Второй модуль – Синтаксический анализатор выполняет синтаксический анализ последовательности символов. На этом этапе символы рассматриваются как неделимые, и их представление как последовательности литер несущественно.

В результате получаем следующую структуру компилятора:



Для реализации компилятора необходимо реализовать каждый из модулей, представленных выше.

3 Описание языка Pascal

Будем рассматривать 3 типа токенов: ключевые слова, идентификаторы и константы.

Ключевые слова – это слова ЯП, которые имеют специальное, раз и навсегда закрепленное за ними значение. В программе нельзя использовать идентификаторы, совпадающие с ключевыми словами.

К ключевым словам будем также относить знаки операций, ограничители и специальные символы.

Идентификаторы – имена программ, модулей, функций, типов, переменных и констант. Идентификаторы начинаются с буквы (A-Z, a-z) или с символа нижнего подчеркивания и могут содержать буквы, символ нижнего подчеркивания и цифры. В языке Pascal идентификаторы нечувствительные к регистру.

Константы – это числа или строки, которые встречают в выражениях.

Ключевые слова:

|  |
| --- |
| program |
| begin |
| end |
| var |
| type |
| function |
| if |
| then |
| else |
| while |
| do |
| and |
| or |
| not |
| ^ |
| + |
| - |
| \* |
| / |
| ( |
| ) |
| := |
| . |
| , |
| : |
| ; |
| = |
| < |
| > |
| <= |
| >= |
| <> |

4 Модуль ввода-вывода

4.1 Модуль ввода

4.1.1 Проектирование

Для реализации модуля ввода создадим класс Reader, который будет содержать следующие поля и методы:

class Reader {

public:

Reader(std::istream& stream);

~Reader();

std::pair<char, TextPosition> get();

private:

std::istream& stream;

TextPosition pos;

};

stream хранит в себе ссылку на поток ввода, таким образом в качестве ввода могут использоваться различные источники (из файла, из строки, из консоли и т.д.).

pos хранит в себе информацию о текущей позиции в тексте и имеет следующее описание:

struct TextPosition {

int lineIndex;

int charIndex;

TextPosition();

TextPosition(int lineIndex, int charIndex);

};

То есть он хранит номер текущей строки и номер текущей позиции в этой строке.

Метод get() возвращает следующий символ из потока ввода и его позицию

4.1.2 Тестирование

Для тестирования рассмотрим пример паскаль программы и выведем все, что возвращает нам модуль ввода.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Видим, что в консоль программа вывелась такая же, как и была на входе.

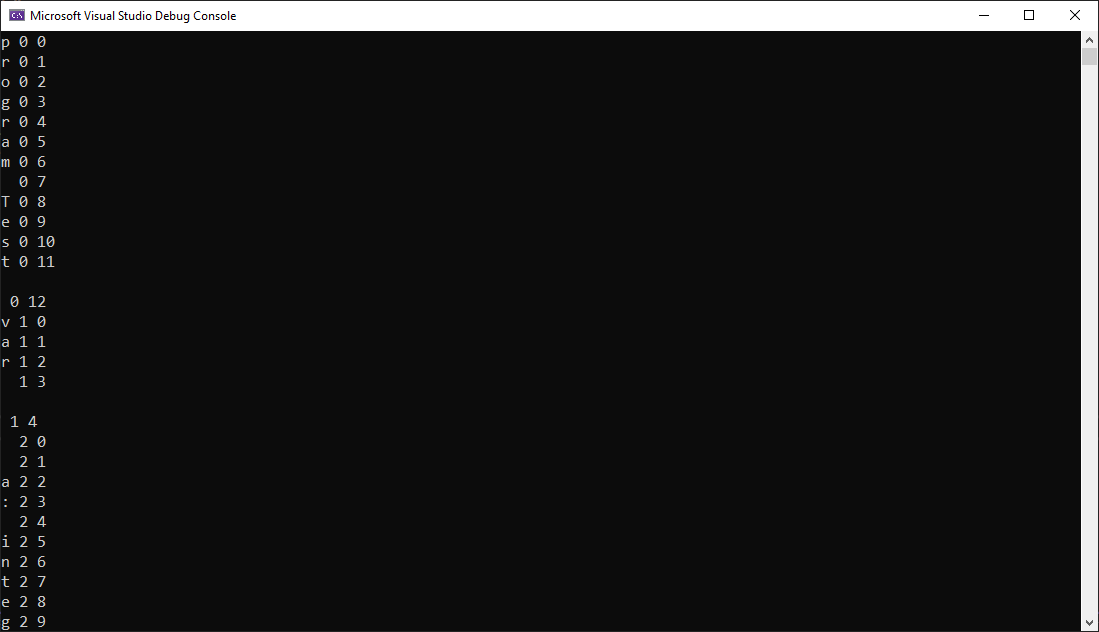
Теперь протестируем поток ввода из строки, для этого запишем следующую программу:

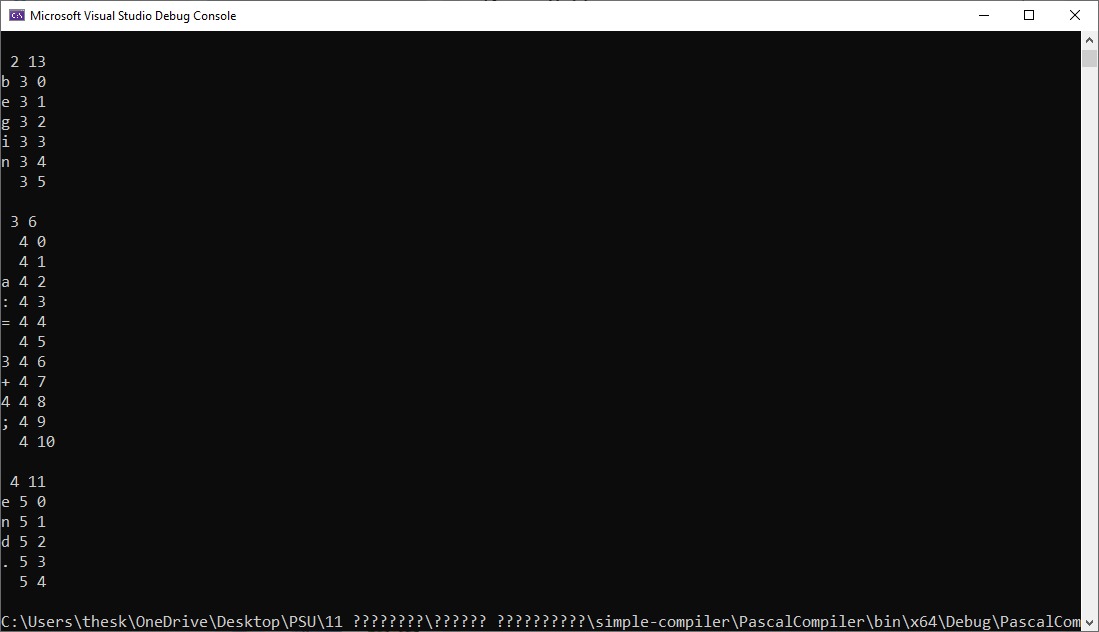
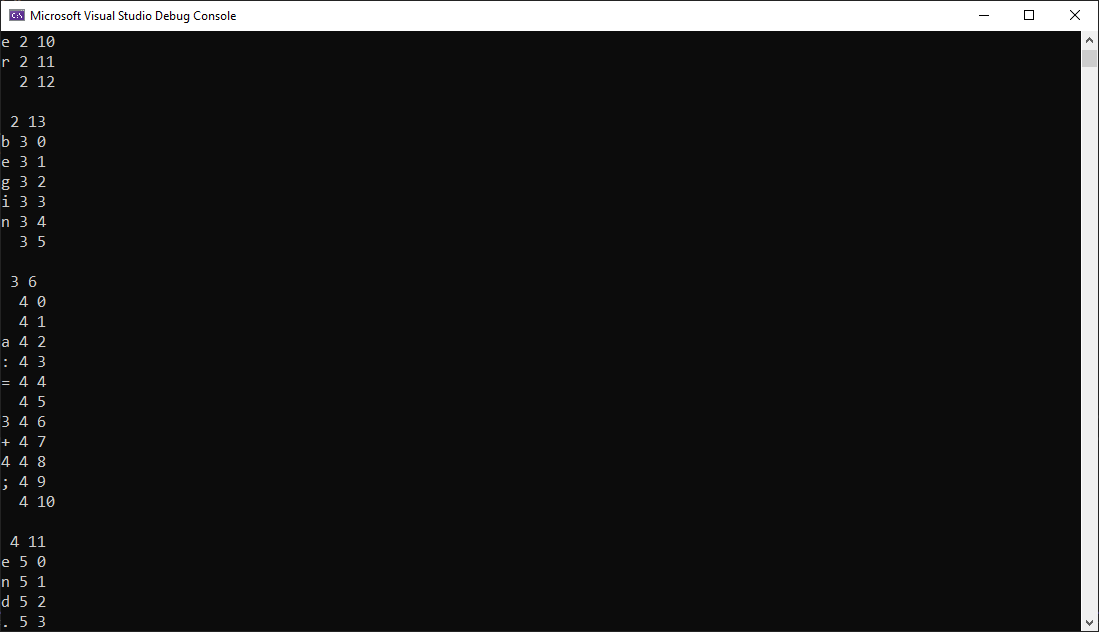
std::string s = "program Test\nvar \n a: integer \nbegin \n a:= 3+4; \nend.";

Результат:

Изображение выглядит как текст

Автоматически созданное описание

Теперь будем выводить позиции символов, получим следующий результат:



* 1. Модуль вывода

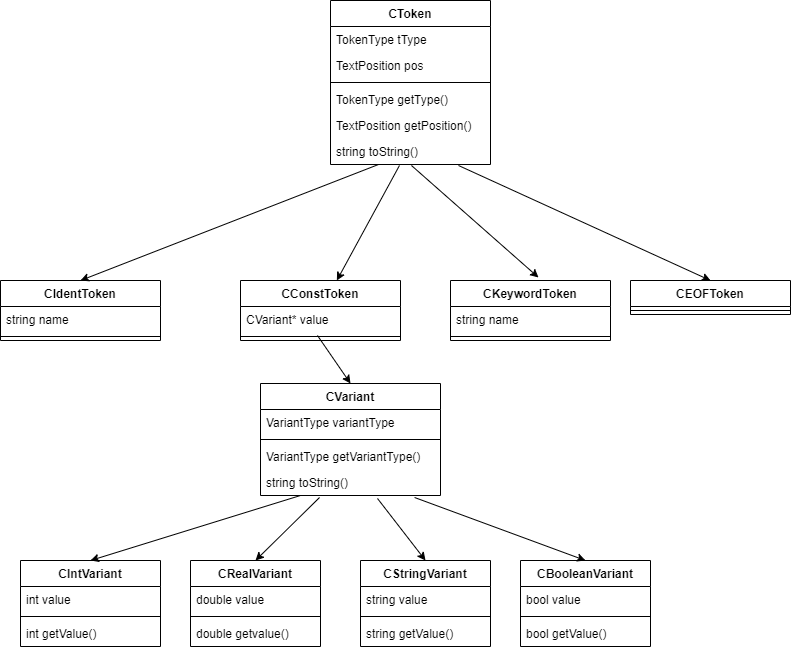
1. Лексический анализатор

5.1 Проектирование

5.1.1 Токены

Лексический анализатор формирует токены исходной программы, кроме того, он распознает и исключает комментарии, которые не нужны для дальнейшей трансляции.

Ниже представлена диаграмма классов токенов.



TokenType может быть одним из ttIdent, ttKeyword, ttConst, ttEOF – идентификатор, ключевое слово, константа и конец файла.

VariantType может быть одним из vtInt, vtReal, vtString, vtBoolean – integer, real, string и boolean.

Возможные ключевые слова и их обозначения представлены ниже:

programSy, // program

beginSy, // begin

endSy, // end

varSy, // var

typeSy, // type

functionSy, // function

ifSy, // if

thenSy, // then

elseSy, // else

whileSy, // while

doSy, // do

andSy, // and

orSy, // orSy

notSy, // not

pointerSy, // ^

plusSy, // +

minusSy, // -

multiplySy, // \*

divisionSy, // /

leftParSy, // (

rightParSy, // )

assignSy, // :=

dotSy, // .

commaSy, // ,

colonSy, // :

semicolonSy, // ;

equalSy, // =

lessSy, // <

greaterSy, // >

lessEqualSy, // <=

greaterEqualSy, // >=

notEqualSy, // <>

Для каждого ключевого слова хранится его строковое представление в словаре, а также для каждой строки хранится соответствующее ключевое слово. Таким образом возможно преобразование из строки в код ключевого слова и обратно.

5.1.2 Обработка ошибок

При возникновении ошибок во время лексического анализа, программа будет выбрасывать исключение определенного типа. На данном этапе используются следующие типы:

* IntegerOverflow – слишком длинное целое число
* UnknownSymbol – неизвестный символ
* IncorrectNumber – некорректное целое/вещественное число (когда в числе есть буквы или больше 1 точки)
* UnexptectedSymbol – непредвиденный символ, когда до начала комментария встретилась закрывающая скобка
* NoMatchingSymbol – при обработке комментария или строки, не нашлась соответствующая закрывающая скобка/ковычка

Реализован отдельный класс Error, который формирует сообщение об ошибке.

5.1.3 Лексер

Класс Лексера имеет следующую структуру:

class CLexer {

public:

CLexer(Reader\* reader);

std::unique\_ptr<CToken> getNextToken();

void getNextChar();

private:

std::unique\_ptr<Reader> reader;

char ch;

TextPosition pos;

std::string getName();

std::pair<VariantType, std::string> getNumber();

std::string getString();

void deleteComment();

bool isSpace(char ch);

bool isLetter(char ch);

bool isDigit(char ch);

bool isNumber(std::string &number);

};

Лексер хранит текущий символ ch и его позицию pos в тексте. Метод getNextChar() получает следующий символ и его позицию.

В классе Лексера описано несколько вспомогательных методов:

* isSpace(ch) – является ли символ ch пробельным
* isLetter(ch) – является ли символ ch буквой или нижним подчеркиванием
* isDigit(ch) – является ли символ ch цифрой
* IsNumber(number) – является ли строка number числом
* deleteComment() – Выделяет и удаляет комментарий
* getString() – выделяет константную строчку и возвращает ее
* getNumber() – выделяет число и возвращает его
* getName() – выделяет идентификатор или ключевое слово и возвращает его

Основной метод getNextToken() возвращает следующий токен, в зависимости от текущего символа выделяется тот или иной тип токена, а после этого возвращается.

* 1. Тестирование

Для тестирования лексического анализатора возьмём комплексную программу со всеми ключевыми словами, комментариями и идентификаторами.

Изображение выглядит как текст

Автоматически созданное описание

Результат выполнения программы:

Первая часть – до комментария

1:1: program [ ttKeyword ]

1:9: gcd [ ttIdent ]

1:12: ; [ ttKeyword ]

3:1: type [ ttKeyword ]

4:2: Bptr [ ttIdent ]

4:7: = [ ttKeyword ]

4:9: ^ [ ttKeyword ]

4:10: boolean [ ttIdent ]

4:17: ; [ ttKeyword ]

6:1: var [ ttKeyword ]

6:5: a [ ttIdent ]

6:6: , [ ttKeyword ]

6:7: b [ ttIdent ]

6:8: , [ ttKeyword ]

6:9: c [ ttIdent ]

6:10: : [ ttKeyword ]

6:11: integer [ ttIdent ]

6:18: ; [ ttKeyword ]

7:2: inpt [ ttIdent ]

7:6: : [ ttKeyword ]

7:8: string [ ttIdent ]

7:14: ; [ ttKeyword ]

8:2: d [ ttIdent ]

8:3: : [ ttKeyword ]

8:5: Bptr [ ttIdent ]

8:9: ; [ ttKeyword ]

9:2: e [ ttIdent ]

9:3: : [ ttKeyword ]

9:5: boolean [ ttIdent ]

9:12: ; [ ttKeyword ]

10:2: f [ ttIdent ]

10:3: , [ ttKeyword ]

10:4: h [ ttIdent ]

10:5: : [ ttKeyword ]

10:7: real [ ttIdent ]

10:11: ; [ ttKeyword ]

12:1: inpt [ ttIdent ]

12:5: : [ ttKeyword ]

12:6: string [ ttIdent ]

12:12: ; [ ttKeyword ]

Вторая часть – описание функции

19:1: function [ ttKeyword ]

19:10: GCD [ ttIdent ]

19:13: ( [ ttKeyword ]

19:14: m [ ttIdent ]

19:15: , [ ttKeyword ]

19:16: n [ ttIdent ]

19:17: : [ ttKeyword ]

19:18: integer [ ttIdent ]

19:25: ) [ ttKeyword ]

19:26: : [ ttKeyword ]

19:27: integer [ ttIdent ]

19:34: ; [ ttKeyword ]

20:2: begin [ ttKeyword ]

21:3: while [ ttKeyword ]

21:9: m [ ttIdent ]

21:10: <> [ ttKeyword ]

21:12: n [ ttIdent ]

21:14: do [ ttKeyword ]

21:17: . [ ttKeyword ]

22:3: if [ ttKeyword ]

22:6: m [ ttIdent ]

22:7: > [ ttKeyword ]

22:8: n [ ttIdent ]

22:10: then [ ttKeyword ]

22:15: m [ ttIdent ]

22:16: := [ ttKeyword ]

22:18: m [ ttIdent ]

22:19: - [ ttKeyword ]

22:20: n [ ttIdent ]

22:22: else [ ttKeyword ]

22:27: n [ ttIdent ]

22:28: := [ ttKeyword ]

22:30: n [ ttIdent ]

22:31: - [ ttKeyword ]

22:32: m [ ttIdent ]

22:33: ; [ ttKeyword ]

23:3: GCD [ ttIdent ]

23:6: := [ ttKeyword ]

23:8: m [ ttIdent ]

23:9: ; [ ttKeyword ]

24:2: end [ ttKeyword ]

24:5: ; [ ttKeyword ]

Третья часть – основная программа

26:1: begin [ ttKeyword ]

27:2: inpt [ ttIdent ]

27:7: := [ ttKeyword ]

27:10: Input 3 numbers ( string ) [ ttConst ]

27:27: ; [ ttKeyword ]

28:2: writeln [ ttIdent ]

28:9: ( [ ttKeyword ]

28:10: inpt [ ttIdent ]

28:14: ) [ ttKeyword ]

28:15: ; [ ttKeyword ]

30:2: e [ ttIdent ]

30:3: := [ ttKeyword ]

30:6: true ( boolean ) [ ttConst ]

30:10: ; [ ttKeyword ]

31:2: d [ ttIdent ]

31:4: := [ ttKeyword ]

31:7: @ [ ttKeyword ]

31:8: e [ ttIdent ]

31:9: ; [ ttKeyword ]

33:2: f [ ttIdent ]

33:4: := [ ttKeyword ]

33:7: 123.456700 ( real ) [ ttConst ]

33:15: ; [ ttKeyword ]

36:2: read [ ttIdent ]

36:6: ( [ ttKeyword ]

36:7: a [ ttIdent ]

36:8: , [ ttKeyword ]

36:9: b [ ttIdent ]

36:10: , [ ttKeyword ]

36:11: c [ ttIdent ]

36:12: ) [ ttKeyword ]

36:13: ; [ ttKeyword ]

38:2: writeln [ ttIdent ]

38:9: ( [ ttKeyword ]

38:10: The gcd of 3 numbers is: ( string ) [ ttConst ]

38:36: , [ ttKeyword ]

38:37: GCD [ ttIdent ]

38:40: ( [ ttKeyword ]

38:41: GCD [ ttIdent ]

38:44: ( [ ttKeyword ]

38:45: a [ ttIdent ]

38:46: , [ ttKeyword ]

38:47: b [ ttIdent ]

38:48: ) [ ttKeyword ]

38:49: , [ ttKeyword ]

38:50: c [ ttIdent ]

38:51: ) [ ttKeyword ]

38:52: ) [ ttKeyword ]

38:53: ; [ ttKeyword ]

40:2: a [ ttIdent ]

40:4: := [ ttKeyword ]

40:7: ( [ ttKeyword ]

40:8: a [ ttIdent ]

40:9: + [ ttKeyword ]

40:10: b [ ttIdent ]

40:11: ) [ ttKeyword ]

40:12: \* [ ttKeyword ]

40:13: c [ ttIdent ]

40:15: - [ ttKeyword ]

40:17: d [ ttIdent ]

40:18: / [ ttKeyword ]

40:20: ( [ ttKeyword ]

40:22: 312 ( integer ) [ ttConst ]

40:26: + [ ttKeyword ]

40:28: 40 ( integer ) [ ttConst ]

40:30: ) [ ttKeyword ]

40:31: ; [ ttKeyword ]

41:2: if [ ttKeyword ]

41:5: a [ ttIdent ]

41:7: = [ ttKeyword ]

41:9: 30 ( integer ) [ ttConst ]

41:12: and [ ttKeyword ]

41:16: b [ ttIdent ]

41:18: >= [ ttKeyword ]

41:21: 15 ( integer ) [ ttConst ]

41:24: or [ ttKeyword ]

41:27: c [ ttIdent ]

41:29: <= [ ttKeyword ]

41:32: 17 ( integer ) [ ttConst ]

41:35: xor [ ttKeyword ]

41:39: t [ ttIdent ]

41:41: <> [ ttKeyword ]

41:44: 20 ( integer ) [ ttConst ]

41:47: and [ ttKeyword ]

41:51: not [ ttKeyword ]

41:55: 12 ( integer ) [ ttConst ]

41:58: then [ ttKeyword ]

42:2: begin [ ttKeyword ]

43:3: writeln [ ttIdent ]

43:10: ( [ ttKeyword ]

43:11: Fine ( string ) [ ttConst ]

43:17: ) [ ttKeyword ]

43:18: ; [ ttKeyword ]

44:2: end [ ttKeyword ]

44:5: ; [ ttKeyword ]

45:1: end [ ttKeyword ]

45:4: . [ ttKeyword ]

6 Синтаксический анализатор

Синтаксический анализатор проверяет, удовлетворяет ли программа формальным правилам. Формальные правила языка можно описать при помощи BNF.

Все правила, описывающие синтаксис языка, имеют вид:

<S> ::= L

Где S – обозначение синтаксической конструкции, а L может состоять из символов языка, синтаксических конструкций и метасимволов «|» и «{}».

Основная идея написания синтаксического анализатора заключается в следующем: для каждого правила необходимо описать функцию, тело которой является результатом некоторого преобразования правой части.

procs() {

T(L)

}

6.1 Используемые BNF

<program> ::= program <identifier>;<block>.|<block>.

<block> ::= <typeDeclarationPart><varDeclarationPart>

<funcionDeclarationPart><statementPart>

<typeDeclarationPart> ::= <empty> | type <typeDeclaration>; {<typeDeclaration>;}

<typeDeclaration> ::=  <identifier> = <type>

<type> ::= <identifier> | <pointerType>

<pointerType> ::= ^<identifier>

<varDeclarationPart>::= <empty> | var <varDeclaration>; {<varDeclaration>;}

<varDeclaration> ::= <identifier> {,<identifier>} : <type>

<funcionDeclarationPart> ::= {<funcionDeclaration>;}

<funcionDeclaration> ::= <functionHeading> <block>

<functionHeading> ::= function <identifier>:<type>; |

        function <identifier>(<formalParameterSection>{;<formalParameterSection>}):<type>;

<formalParameterSection> ::= <empty> | <parameterGroup> | var <parameterGroup>

<parameterGroup> ::= <identifier>{,<identifier>}: <type>

<statementPart> ::= <compoundStatement>

<compoundStatement> ::= begin <empty> | <statement>{;<statement>} end

<statement> ::= <simpleStatement>|<structuredStatement>

<simpleStatement> ::= <assignmentStatement>|<procedureStatement>

<assignmentStatement> ::= <variable> := <expression>

<variable> ::= <identifier>{^}

<expression> ::= <simpleExpression>|<simpleExpression> <relationalOperator> <simpleExpression>

<relationalOperator> ::= <|>|>=|<=|=|<>

<simpleExpression> ::= <term>{<addingOperator><term>}

<addingOperator> ::= + | - | or

<term> ::= <factor>{<multiplyingOperator><factor>}

<multiplyingOperator> ::= \* | / | and

<factor> ::= <variable> | <unsignedConst> | (<expression>) | <functionDesignator> | <unaryOperator><factor>

<unaryOperator> ::= + | - | not

<unsignedConst> ::= <unsignedNumber> | <stringConst>

<unsignedNumber> ::= <unsignedInteger> | <unsignedReal>

<functionDesignator> ::= <identifier> | <identifier>(<actualParameter>{,<actualParameter>})

<procedureStatement> ::= <identifier> | <identifier>(<actualParameter>{,<actualParameter>})

<actualParameter> ::= <expression>|<empty>

<empty> ::=

<structuredStatement> ::= <compoundStatement>|<ifStatement>|<whileStatement>

<ifStatement> ::= if <expression> then <statement>| if<expression> then <statement> else <statement>

<whileStatement> ::= while <expression> do <statement>

6.2 Нейтрализация синтаксических ошибок

Синтаксический анализатор, описанный ранее будет сканировать программу только до первой встреченной ошибки. На практике такая организация работы неприемлема: вместо этого компилятор должен сформировать соответствующее сообщение об ошибке и продолжить анализ.

Для этого дополним синтаксический анализатор действиями, которые позволят продолжить анализ исходной программы после обнаружения ошибки. Основная идея состоит в следующем: после выявления ошибки будем пропускать один или несколько символов, чтобы найти символ, начиная с которого можно продолжить анализ.

Будем делать следующее: при обнаружении ошибки при анализе текущей конструкции, мы будем ее отлавливать, добавлять в список ошибок, а после этого переходить к следующему допустимому токену.

Мы будем хранить список acceptableTokens – символы, с которых может начинаться конструкция, которая следует непосредственно за текущей. Функция skipTo() будем пропускать токены, пока не встретит один из acceptableTokens.

Теперь конструкции, в которых возможны ошибки будут иметь следующий вид:

try {

//анализ текущей конструкции

}

catch (e) {

addError(e);

skipTo(acceptableTokens);

}

6.3 Тестирование

Для тестирования синтаксического анализатора возьмем синтаксически правильную программу, которая покрывает различные случаи.

program gcd;

type

Bptr = ^boolean;

IPtr = ^integer;

var a,b,c:integer;

inpt: string;

d: Bptr;

e: boolean;

f,h: real;

inpt:string;

{

function to find greatest common divisor

}

function GCD(m,n:integer):integer;

begin

while m<>n do

if m>n then m:=m-n else n:=n-m;

GCD:=m;

end;

var

v1:real;

v2: IPtr;

function t():real;

begin

t := 13;

end;

begin

inpt := 'Input 3 numbers';

writeln(inpt);

e:= true;

d := e;

f := 123.4567;

t:= ' 14214214214';

e^^ := 12 + -13\*14/(35-14.33)+GCD(3,5)/t;

read(a,b,c);

writeln('The gcd of 3 numbers is:',GCD(GCD(a,b),c));

a := (a+b)\*c - d/ ( 312 + 40);

if ---12 + 13 >= ((4\*5)+10 > 5) then

begin

writeln('Fine');

end;

end.

В результате синтаксического анализа ошибок не найдено.

Изображение выглядит как текст

Автоматически созданное описание

Рассмотрим теперь программу с ошибками, красным выделены места, в которых ошибки.

program gcd;

type

Bptr = ^boolean;

IPtr = ^integer

var a,b,c:integer;

inpt: string;

d: Bptr;

e boolean;

f,h: real;

inpt:string;

{

function to find greatest common divisor

}

function GCD(m,n:integer):integer;

begin

while m<>n do

if m>n then m:=m-n else n:=n-m;

kek-

end;

var

v1:real;

v2: IPtr;

function t():real;

begin

t := 13;

end;

begin

inpt := 'Input 3 numbers';

writeln(inpt);

e:= true;

d := e;

f := 123.4567;

t:= ' 14214214214';

e^^ := 12 + -13\*14/(35-14.33)+GCD(3,5)/t;

read(a,b,c);

writeln('The gcd of 3 numbers is:',GCD(GCD(a,b),c));

a := (a+b)\*c - d/ ( 312 + 40);

if ---12 + 13 >= ((4\*5)+10 > 5) then

begin

writeln('Fine');

end;

end.

Изображение выглядит как текст

Автоматически созданное описание

Как мы видим, синтаксический анализатор выявил все ошибки.

Теперь вместо программы с ошибками, напишем просто набор символов

var

grgr

gdf

2t2t3

gk

function igor

begin

end;

В результате синтаксического анализа получаем следующее:

Изображение выглядит как текст

Автоматически созданное описание

7 Семантический анализатор

Здесь описание семантического анализатора.

8 Генерация кода

Здесь генерация кода.