|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Министерство науки и высшего образования РФ | | | | | | | | |
|  | | |  | | |  | | |
| Пермский государственный национальный  исследовательский университет | | | | | | | | |
|  | | |  | | |  | | |
|  | | ОТЧЁТ  по лабораторной работе “Разработка компилятора языка Pascal»  по дисциплине «Формальные грамматики и методы трансляции” | | | | |  | |
|  | | |  | | |  | | |
|  | Работу выполнил  студент гр. ПМИ-1,2  Валеев Р. Р. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_» \_\_\_\_\_\_\_\_ 2021 | | |  | Проверил  асис.кафедры МОВС  Пономарев Ф.А. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_» \_\_\_\_\_\_\_ 2021 | | |  |
|  |  | | |  |  | | |  |
| Пермь 2021 | | | | | | | | |

# Анализ

Для решения глобальной задачи – написание компилятора для подмножества языка Pascal, требуется решить несколько подзадач, а именно, спроектировать и разработать лексический, синтаксический и семантический анализаторы, а также генератор кода.

Следующим этапом станет разработка синтаксического анализатора. Синтаксический анализатор запрашивает токен, созданный лексическим анализатором, и проверяет соответствует ли он какому-либо формальному правилу.

Для задания синтаксиса языка программирования применяются формы Бэкуса-Наура (БНФ) или синтаксические диаграммы. Мы будем описывать синтаксические правила в форме БНФ.

# Проектирование

Перед началом разработки синтаксического анализатора было проведено начальное проектирование. На рисунке ниже представлена диаграмма классов, отражающая обновлённую структуру компилятора

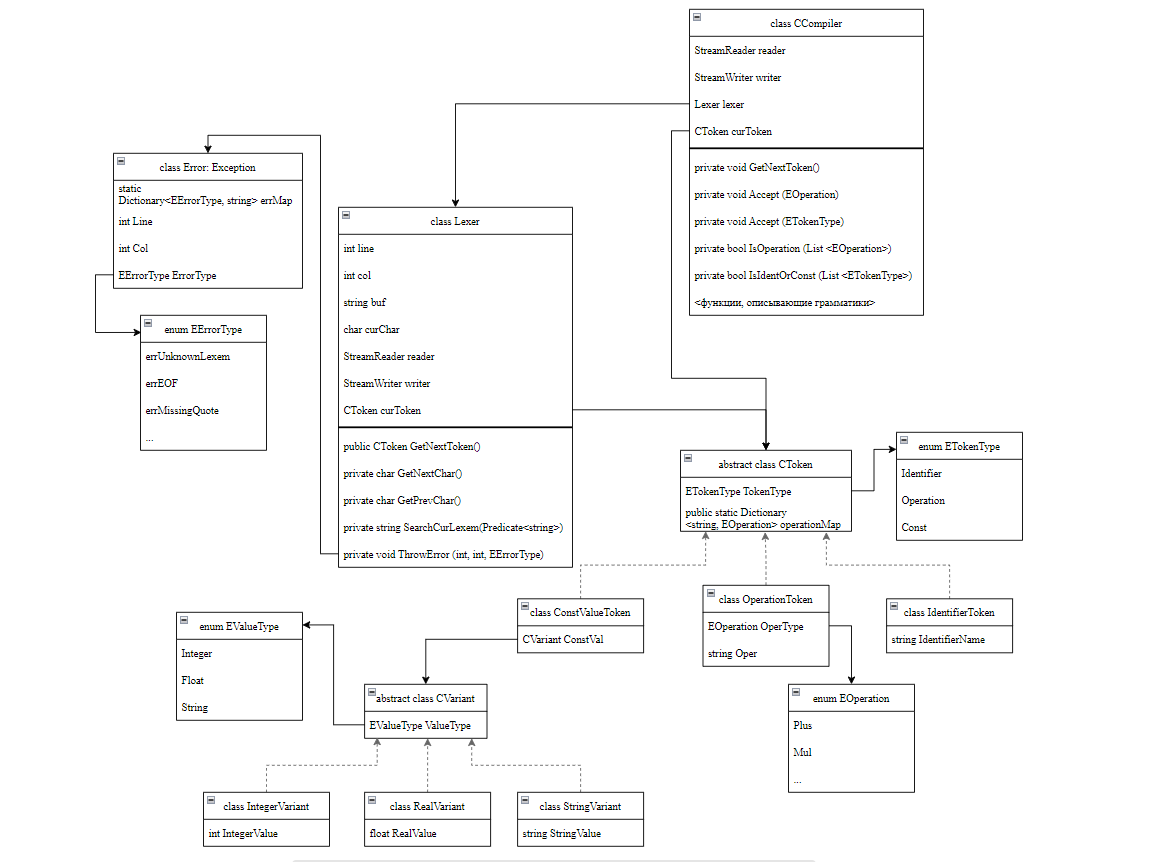


Рисунок 1 – Диаграмма классов

Класс CCompiler становится главным в иерархии. Именно он отвечает за процесс компиляции. Синтаксический анализатор не вынесен в отдельный класс, как, например, лексический анализатор. Синтаксический анализ происходит в самом классе компилятора CCompiler. Далее к синтаксическому анализу добавится семантический анализ, который будет происходить параллельно с синтаксическим. Для этого будут дорабатываться методы, реализованные для анализа синтаксических конструкций.

Все правила, описывающие синтаксис языка, представим в виде БНФ. Для каждой БНФ будет описана своя функция. Работа синтаксического анализатора начинается с вызова метода Program(), который соответствует синтаксическому правилу <программа>. В ходе выполнения данного метода происходит обращение к другим методам, например, Block(), TypePart(), VarPart() и т.д. Взаимодействие с лексическим анализатором происходит внутри метода GetNextToken(). Лексический анализатор считывает последовательность литер и формирует токен, который возвращается синтаксическому анализатору. Далее происходит сравнение полученного символа с ожидаемым символом в анализируемой конструкции. Если ожидаемый токен не совпадает с фактическим, то выдаётся сообщение об ошибке. На текущем этапе разработки процесс компиляции будет прекращаться при обнаружении ошибки.

# Разработка

Текущий токен будет храниться в атрибуте curToken класса CCompiler. Для получения следующего токена используется метод GetNextToken(). Он вызывает одноимённый метод класса Lexer. Полученный токен запоминается в curToken.

Метод Accept() осуществляет сравнение текущего токена с ожидаемым. Он является перегруженным. В качестве параметров он может принимать либо тип токена, либо тип операции. Если текущий и ожидаемый токены не совпадают, то генерируется исключение типа Error (тип исключения для обработки лексических и синтаксических ошибок). Для того чтобы по ожидаемому символу можно было определить код ошибки, коды символов и ошибок вида «ожидался символ (название символа)» совпадают.

В ходе разбора синтаксических конструкций в качестве следующего символа может ожидаться несколько вариантов. В этом случае вызываются методы IsOperation() или IsIdenOrConst() (в зависимости от ожидаемого символа). Их цель – проверить, что текущий токен входит в список ожидаемых символов.

Функции, описывающие грамматики, могут вызывать друг друга, а также проверять, что текущий токен совпадает с ожидаемым. Для примера рассмотрим, как осуществляется разбор следующих БНФ:

1. <раздел переменных> ::= var <описание однотипных переменных> ; { <описание однотипных переменных> ; } | <пусто>
2. <описание однотипных переменных> ::= <имя> { , <имя> } : <тип>

Код функции, осуществляющей разбор конструкции <раздел переменных>:

void VarPart()

{

if (IsOperation(new List<EOperation>() { EOperation.Var }))

{

GetNextToken();

do

{

SimilarTypeVarPart();

Accept(EOperation.Semicolon);

} while (IsIdentOrConst(new List<ETokenType>() { ETokenType.Identifier }));

}

}

Так как раздел описания переменных может отсутствовать в исходной программе, то сначала проверяется, чему равен текущий токен. Если текущий токен является символом ‘var’, то начинается разбор конструкции <раздел переменных>. Иначе, разбор прекращается. Так как уже известно, чему равен текущий символ, то просто считываем следующий токен. Далее хотя бы один раз мы точно попадём в конструкцию <описание однотипных переменных> и считаем символ ‘;’. Рассмотрим код функции, осуществляющий разбор БНФ <описание однотипных переменных>:

void SimilarTypeVarPart()

{

Accept(ETokenType.Identifier);

while (IsOperation(new List<EOperation>() { EOperation.Comma }))

{

GetNextToken();

Accept(ETokenType.Identifier);

}

Accept(EOperation.Colon);

Type();

}

Описание однотипных переменных начинается с идентификатора, после чего идут символ ‘:’ и тип (идентификатор). Проверяем, что текущий токен является идентификатором и считываем его. Пока встречаем символ ‘,’, то считываем два следующих токена, проверяя, что последний считанный токен является идентификатором. Как только встретили отличный от ‘,’ символ, проверяем, что текущий символ есть ‘:’, и считываем его. Далее происходит разбор конструкции <тип>, который сводится к тому, что следующим анализируемым символом должен быть идентификатор.

Окончив разбор конструкции <описание однотипных переменных>, происходит возврат к разделу переменных. Считывается символ ‘;’. Вызов функции для разбора конструкции <описание однотипных переменных> повторяется до тех пор, пока следующим токеном после символа ‘;’ является идентификатор.

# Тестирование

В этом разделе протестирована работа синтаксического анализатора на различных примерах.

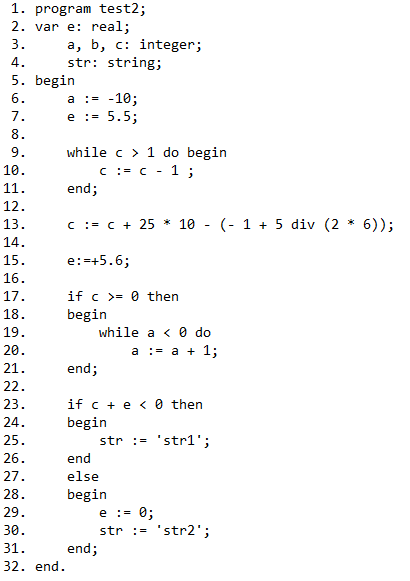
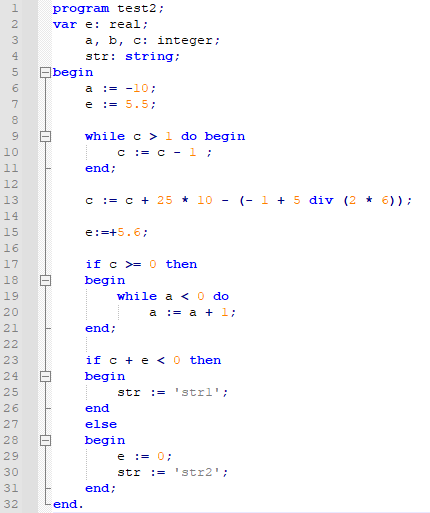


Рисунок 2 – Синтаксически правильная программа

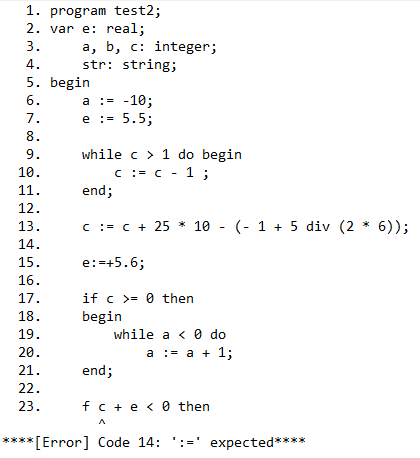
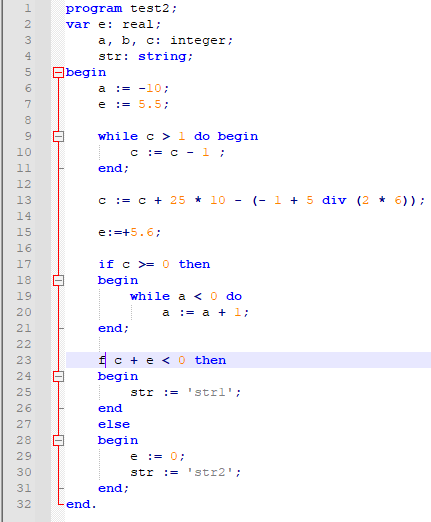


Рисунок 3 – Синтаксическая ошибка: «ожидался символ ‘:=’»

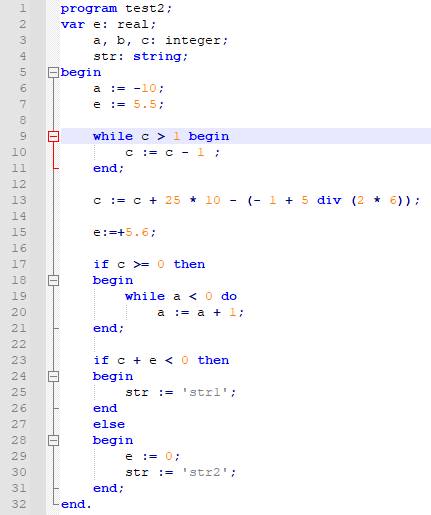
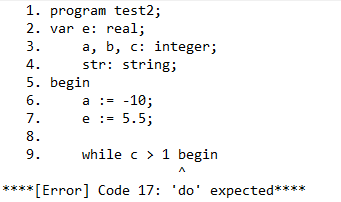


Рисунок 4 – Синтаксическая ошибка: «ожидался символ ‘do’»

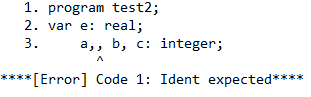
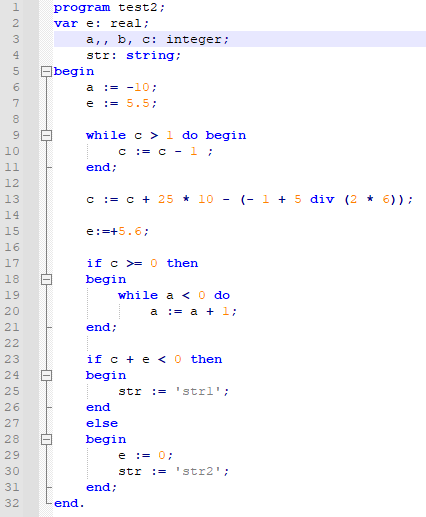


Рисунок 5 – Синтаксическая ошибка: «ожидался идентификатор»

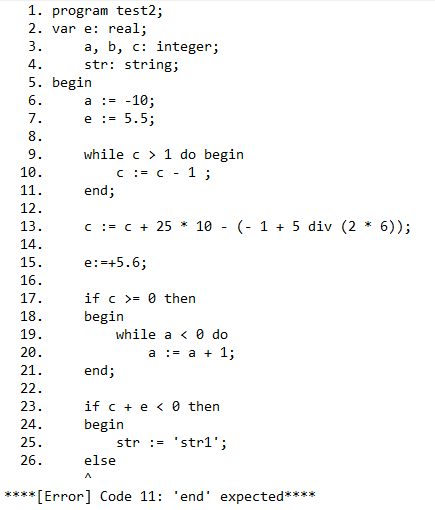
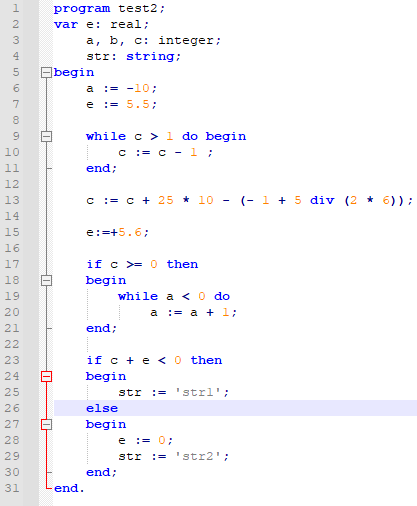


Рисунок 6 – Синтаксическая ошибка: «ожидался символ ‘end’»