

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФГАОУ ВО «Пермский государственный  
национальный исследовательский университет»

ОТЧЕТ

по заданию «Разработка компилятора языка программирования Pascal»  
по дисциплине «Формальные грамматики и методы трансляции»

Работу выполнил  
Студент гр.ПМИ-1,2  
Проскуряков К.А.\_\_\_\_\_  
<<\_\_>>\_\_\_\_\_2021

Проверил  
Ассистент кафедры МОВС  
Пономарёв Ф.А.\_\_\_\_\_  
<<\_\_>>\_\_\_\_\_2021

Пермь 2021

## Постановка задачи

Необходимо написать компилятор для подмножества языка Pascal. Компилятор состоит из модуля ввода-вывода, лексического анализатора, синтаксического анализатора и семантического анализатора. Необходимо также реализовать нейтрализацию синтаксических и семантических ошибок.

В подмножество языка Pascal входит:

- Раздел описания переменных;
- Раздел операторов;
- Переменные и константы типов `integer`, `real`, `string`;
- Выражение (арифметические и логические операции над константами и переменными);
- Оператор присваивания;
- Составной оператор;
- Условный оператор (`if`);
- Оператор цикла с предусловием (`while`);

## Оглавление

Постановка задачи.....	2
Модуль ввода-вывода .....	4
1. Описание.....	4
2. Проектирование.....	4
3. Реализация .....	5
4. Тестирование .....	7
Лексический анализатор .....	9
1. Описание.....	9
2. Проектирование.....	10
3. Реализация .....	12
4. Тестирование .....	15
Синтаксический и семантический анализ.....	21
1. Описание.....	21
2. Проектирование.....	22
3. Реализация .....	26
4. Тестирование .....	30

## Модуль ввода-вывода

### 1. Описание

Необходимо разработать модуль ввода-вывода. Этот модуль должен получать на вход путь до файла с исходным кодом на языке Pascal и путь до файла, в который необходимо будет вывести листинг исходной программы и информацию об обнаруженных в ней ошибках. Также модуль должен реализовывать возможность получения следующего непечатаемого символа исходного текста программы.

### 2. Проектирование

Модуль ввода-вывода представляет из себя класс **IOModule** с публичным конструктором, принимающим пути до файлов, а также публичным методом **ReadNextCharacter**, возвращающим следующую непечатаемую литеру.

Помимо необходимой функциональности, я, в силу недостатка опыта проектирования, решил добавить в класс модуля ввода-вывода и работу с ошибками.

Ошибка описывается классом **Error**, имеющим два поля – код ошибки и её позиция, а также публичный конструктор, принимающий код и позицию. Помимо конструктора, в классе есть два публичных геттера для доступа к значениям полей.

Таким образом, в класс **IOModule** был добавлен публичный метод **AddError**, принимающий код ошибки и её позицию, создающий экземпляр класса **Error**, и добавляющий его в список ошибок текущей строки.

Поскольку мы ходим выводить пользователю не просто код ошибки, а какое-то осмысленное сообщение, для получения сообщения ошибки из её кода был разработан статический класс **ErrorMatcher**, имеющий единственный публичный метод, возвращающий сообщение ошибки по её коду.

Примерная диаграмма классов модуля представлена на рисунке 1.

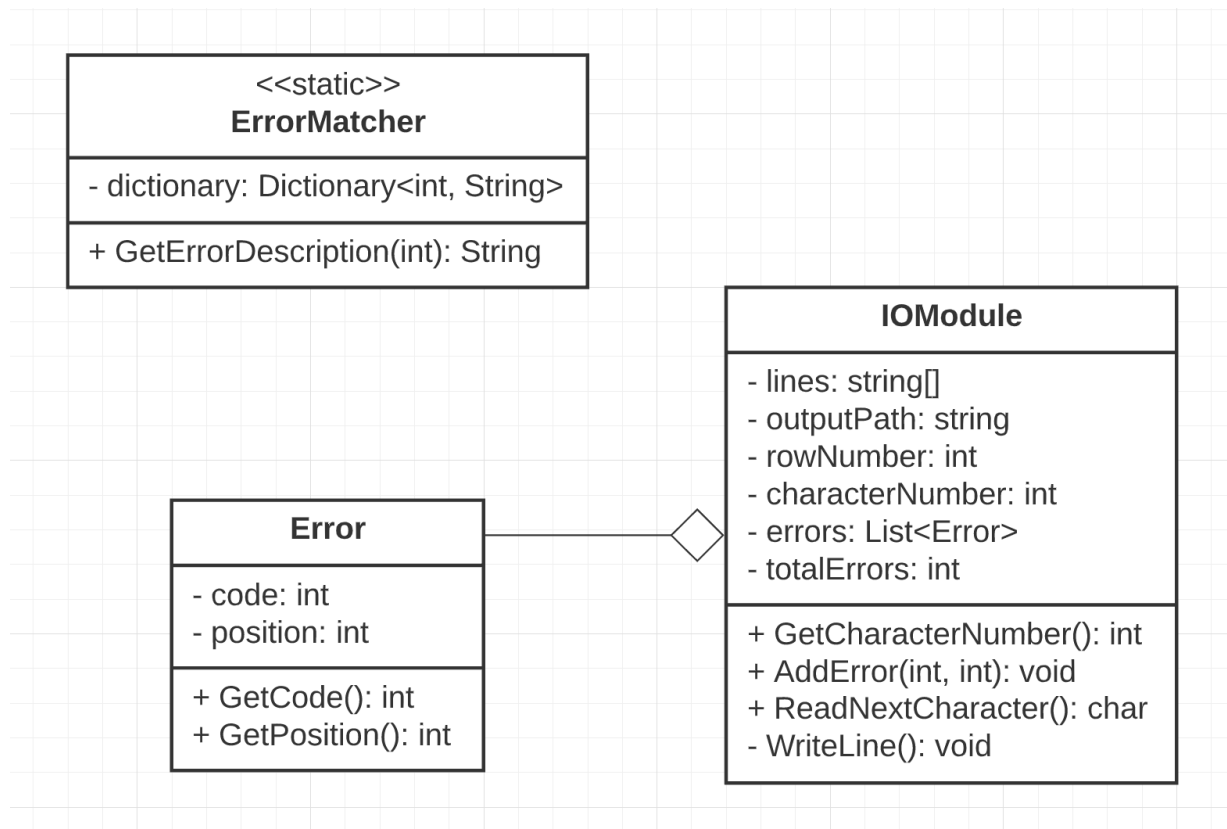


Рисунок 1 – диаграмма классов модуля ввода-вывода

### 3. Реализация

Начнём рассмотрение реализации с класса **IOModule**. Как уже было сказано в предыдущем пункте, этот класс имеет публичный конструктор, принимающий пути до файла с исходным текстом программы, а также путь до файла, в который необходимо вывести листинг программы со всеми обнаруженными ошибками.

Поскольку компилятору всё равно придётся пройти по всему тексту программы, я решил сразу в конструкторе прочитать весь текст по строкам в массив **lines**. Таким образом, избавившись от необходимости хранить путь до исходного файла.

Чтение следующей литеры реализовано в методе **ReadNextCharacter**. В этом методе используются вспомогательные переменные **rowNumber** и **characterNumber**. Метод работает следующим образом:

- Если значение **characterNumber** равно 0 и значение **rowNumber** больше 0, то вызываем метод **WriteLine**.
- Если **rowNumber** меньше длины массива **lines** и **characterNumber** меньше длины текущей строки, возвращаем символ в этой строке с индексом **characterNumber**, а также увеличить значение **characterNumber** на 1.
- Иначе присваиваем **characterNumber** значение 0, увеличиваем значение **rowNumber** на 1. Затем сравниваем его с длиной массива **lines**. Если оно получилось больше, то возвращаем символ конца файла, в противном случае символ переноса строки.

Как можно заметить, текущая строка с обнаруженными ошибками выводится при чтении первого символа следующей строки. Изначально я выводил эту информацию, когда доходил до последнего символа текущей строки. Далее, уже при тестировании лексического анализатора обнаружилась проблема такого подхода, заключающаяся в неправильном определении строки обнаруженной ошибки.

Метод **WriteLine** выводит в файл с листингом текущую строку программы с её номером. Далее, если список **errors** не пуст, увеличиваем счётчик ошибок **totalErrors** на количество элементов в списке, после этого выводим все элементы, указывая коды и сообщения обнаруженных ошибок. Помимо этого, указывается место в строке, где была совершена ошибка. После этого очищаем список ошибок. Если только что выведенная строка оказалась последней, то выводим итоговое количество ошибок.

В статическом классе **ErrorMatcher** содержится словарь, у которого ключ – код ошибки, а значение – соответствующее сообщение. Таким образом, метод **GetErrorDescription** просто возвращает значение по переданному ключу. В данный момент словарь пуст, но при разработке следующих модулей он будет заполняться информацией о соответствующих ошибках.

## 4. Тестирование

- 1) Для начала просто выведем содержимое входного файла в выходной на примере кода программы «Hello, World!».

```
input.pas
1  program HelloWorld;
2  begin
3      writeln('Hello, World!');
4  end.
```

```
output.txt
1      01  program HelloWorld;
2      02  begin
3      03      writeln('Hello, World!');
4      04  end.
5
6  Всего ошибок – 0
```

Текст исходной программы из исходного файла успешно был переписан в выходной. На удивление в нём не обнаружилось ни одной ошибки!

- 2) Теперь попробуем добавить в словарь в классе **ErrorMatcher** фиктивные ошибки.

```
private static readonly Dictionary<int, string> dictionary = new()
{
    [0] = "Ошибка для чётных строк",
    [1] = "Ошибка для нечётных строк"
};
```

Далее немного изменим код метода **ReadNextCharacter** таким образом, чтобы он в каждую строку вставлял столько фиктивных ошибок, какой порядковый номер этой строки. Код ошибки также будет зависеть от чётности/нечётности текущей строки. Позиция ошибки будет определяться номером строки.

```

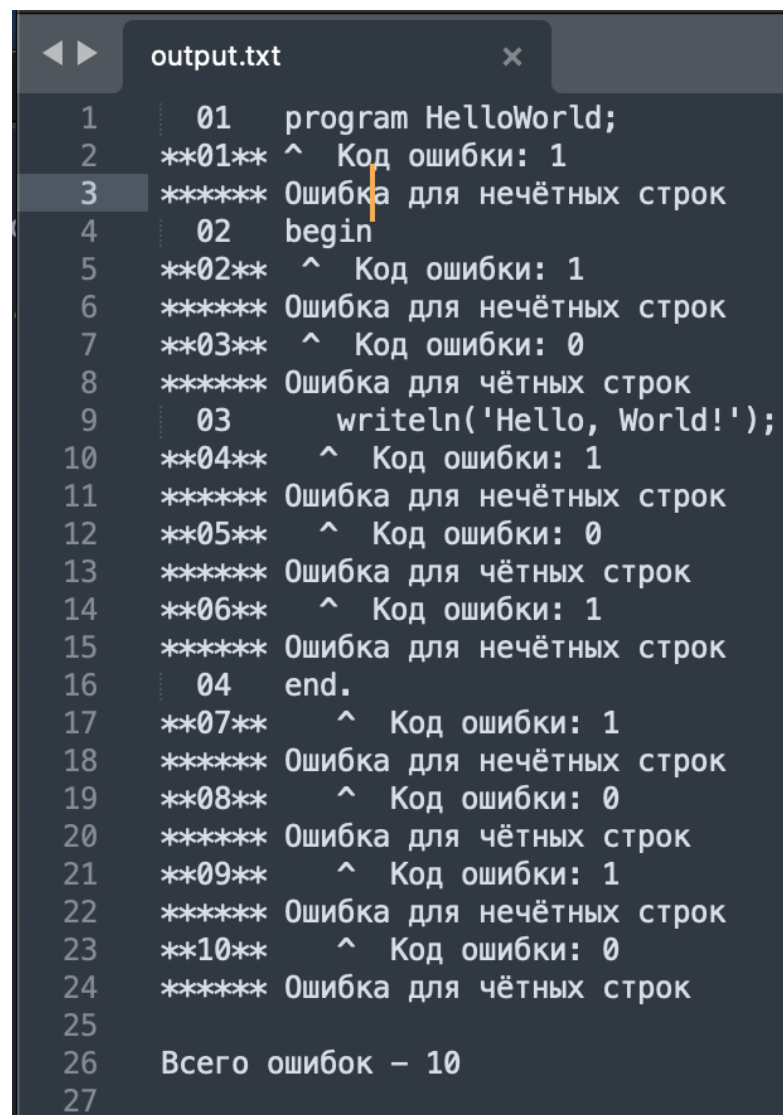
public char ReadNextCharacter()
{
    if (CharacterNumber == 0 && rowNumber > 0)
    {
        for (int i = 0; i < rowNumber; i++)
            AddError(i % 2 == 1 ? 0 : 1, rowNumber);

        WriteLine();
    }

    if (rowNumber < lines.Length && CharacterNumber < lines[rowNumber].Length)
        return lines[rowNumber][CharacterNumber++];

    rowNumber++;
    CharacterNumber = 0;
    return rowNumber <= lines.Length ? '\n' : '\0';
}

```



```

1      01  program HelloWorld;
2      **01** ^ Код ошибки: 1
3      ***** Ошибка для нечётных строк
4      02  begin
5      **02** ^ Код ошибки: 1
6      ***** Ошибка для нечётных строк
7      **03** ^ Код ошибки: 0
8      ***** Ошибка для чётных строк
9      03      writeln('Hello, World!');
10     **04** ^ Код ошибки: 1
11     ***** Ошибка для нечётных строк
12     **05** ^ Код ошибки: 0
13     ***** Ошибка для чётных строк
14     **06** ^ Код ошибки: 1
15     ***** Ошибка для нечётных строк
16     04  end.
17     **07** ^ Код ошибки: 1
18     ***** Ошибка для нечётных строк
19     **08** ^ Код ошибки: 0
20     ***** Ошибка для чётных строк
21     **09** ^ Код ошибки: 1
22     ***** Ошибка для нечётных строк
23     **10** ^ Код ошибки: 0
24     ***** Ошибка для чётных строк
25
26     Всего ошибок – 10
27

```

Как видно из содержания выходного файла, модуль успешно справляется с переводом кодов ошибок в текстовые сообщения, а также корректно подсчитывает номера ошибок и их итоговое количество.



# Лексический анализатор

## 1. Описание

Лексический анализатор должен содержать метод обращения к модулю ввода-вывода для получения непрочитанных литер и формирования из них лексем. Также модуль должен сообщать модулю ввода-вывода об обнаруженных лексических ошибках.

Необходимо реализовать поддержку следующих типов лексем:

- Идентификатор (например, названия переменных, процедур и т.д.);
- Операция (в том числе ключевые слова);
- Константы (целочисленные, числа с плавающей точкой и строки)

К лексическим ошибкам я отнёс:

- Открытие незакрытого комментария (когда однострочный комментарий не закрывается на той же строке, на которой был открыт, или не закрывается до конца файла вовсе, либо многострочный комментарий не закрывается до конца файла);
- Закрытие неоткрытого комментария (когда лексический анализатор встречает лексемы закрытия комментариев. Т.к. при встрече лексемы, открывающей комментарий, все лексемы до закрытия комментария включительно должны быть проигнорированы);
- Ошибка в описании строковой константы (когда строковая константа не закрывается до конца файла, либо она закрывается не на той же строке в исходном тексте, на которой была открыта);
- Ошибка в описании вещественной константы (когда при формировании лексемы вещественной константы в ней встречается несколько символов «.», либо этот символ встречен один раз, но в самом конце лексемы);
- Значение целочисленной константы превышает предел (в моей реализации максимальное значение целочисленной константы – 32767);

- Длина идентификатора превышает предел (в моей реализации максимальная длина идентификатора – 127 символов);
- Запрещённый символ (когда при чтении следующего символа модуль ввода-вывода вернул символ, запрещённый в языке Pascal. Например, символы !, ?, & и т.д.);

Возможно, первые 2 ошибки не стоит относить к лексическим, но все комментарии должны быть проигнорированы именно при работе лексического анализатора и не могут встретиться на следующих этапах анализа. Поэтому я решил отнести эти ошибки именно к лексическим.

## 2. Проектирование

Вся логика лексического анализатора расположена в классе **LexicalAnalyzer**. В качестве одного из полей этого класса выступает реализованный в прошлом разделе модуль ввода-вывода. Поэтому у класса **LexicalAnalyzer** есть публичный конструктор, который принимает пути до входного и выходного файлов, которые затем будут переданы в конструктор класса **IOModule**. Помимо конструктора, этот класс содержит публичный метод **GetNextToken**, возвращающий следующую лексему (токен).

Сам токен представляет из себя абстрактный класс **Token**, от которого уже наследуются классы токена-идентификатора **IdentifierToken**, токена-операции **OperationToken** и токена-константы **ConstantToken**.

Для хранения типа токена в базовом классе было создано перечисление **TokenType**. Также, на этапе проектирования синтаксического анализа я решил модифицировать класс **Token**, добавив в него позицию первой литеры этого токена в строке.

Для хранения операции в классе **OperationToken** было создано перечисление **Operation**, в которое входят все операции и ключевые слова языка.

Для хранения значений констант в классе **ConstantToken** был создан абстрактный класс **Variant**.

От этого класса наследуются 3 производных: класс целочисленной константы **IntegerVariant**, класс вещественной константы **RealVariant** и класс строковой константы **StringVariant**.

Для хранения типа константы в базовом классе было создано перечисление **VariantType**.

В ходе построения токенов в методе **GetNextToken** придётся проверять, является ли текущая строка оператором, а также получать значение перечисления **Operation** по строке. В этих целях был создан статический класс **OperationMatcher**.

Примерная диаграмма классов, включая классы модуля ввода-вывода представлена на рисунке 2.

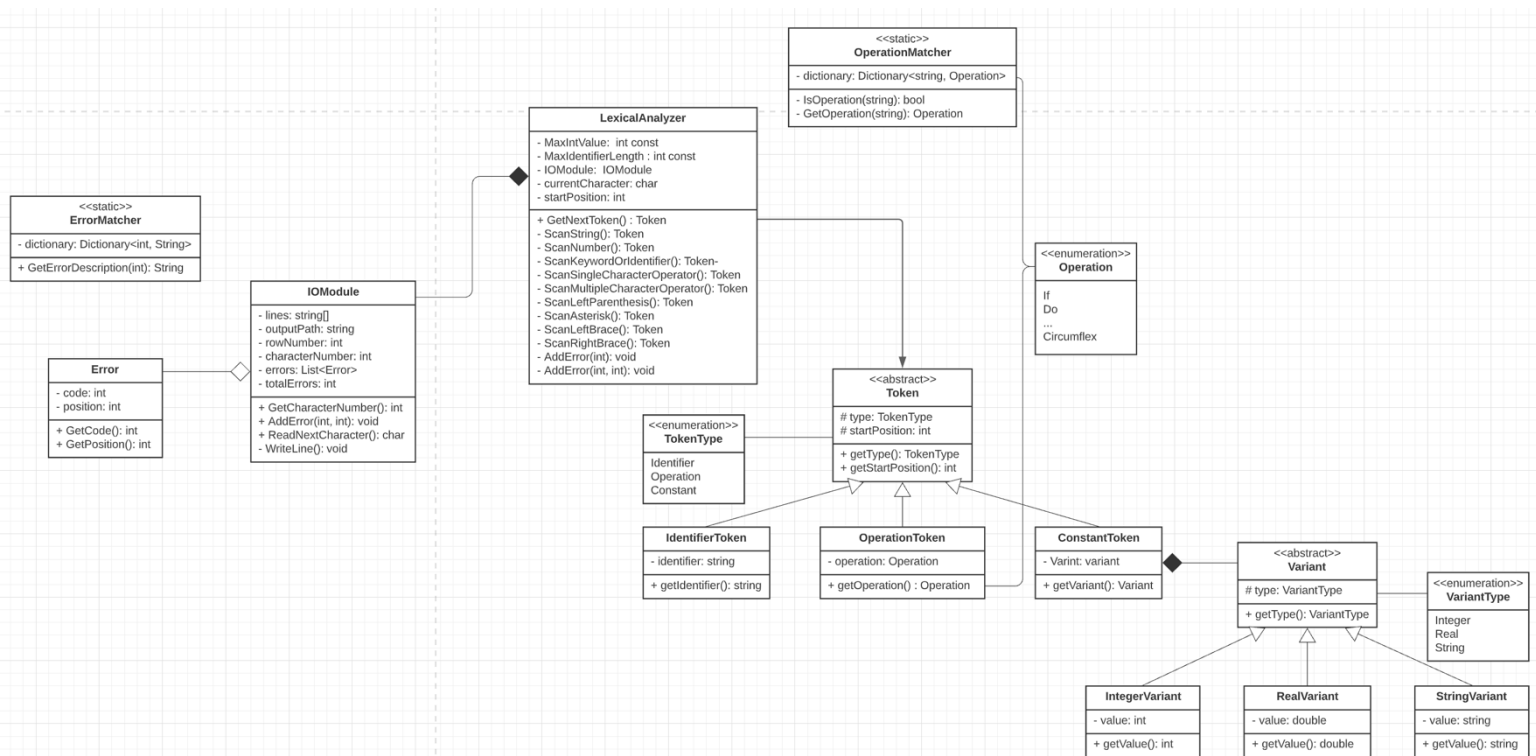


Рисунок 2 – текущая диаграмма классов компилятора

### 3. Реализация

#### Алгоритм построения токена

До тех пор, пока текущая литера является пробелом, символом табуляции (`\t`), либо символом переноса строки (`\n`), запрашиваем у модуля ввода-вывода следующую литеру.

Далее действуем в зависимости от полученной литеры:

- Если текущая литера является терминальным нулём (`\0`), то возвращаем `null`.
- Если текущая литера является символом, открывающим строку, вызываем метод сканирования строки **ScanString**. Этот метод запрашивает у модуля ввода-вывода и запоминает все литеры до тех пор, пока не дойдёт до закрытия строки или конца текста программы. Если метод дошёл до конца текста программы, либо если в полученной строке имеются символы переноса строки, добавляем ошибку, связанную с неправильным описанием строковой константы. Возвращаем **ConstantToken**, хранящий полученную строку.
- Если текущая литера является цифрой, то вызываем метод сканирования числа **ScanNumber**. В этом методе запрашиваются и запоминаются в строку литеры до тех пор, пока они являются цифрами либо символом «`.`». Далее смотрим, содержится ли в полученной строке символ «`.`». Если содержится, то пытаемся методом **TryParse** класса **double** преобразовать строку в вещественное число, если преобразование получилось, необходимо также проверить, что литера «`.`» находилась не на последней позиции в строке. Если оба условия выполняются, значит ошибки нет и возвращаем **ConstantToken**, хранящий полученное значение. Иначе добавляем ошибку описания вещественной константы и возвращаем **ConstantToken** со значением `0.0`. (Потому что в любом

случае, из-за наличия ошибок в тесте программы это значение не будет использовано). Если же символа «.» в строке не содержалось, пытаемся методом **TryParse** уже из класса **int** преобразовать строку в целочисленное значение. Если преобразование удалось, необходимо проверить, не превышает ли полученное значение 32767. Если превышает, необходимо сформировать ошибку превышения целочисленной константой предела и вернуть **ConstantToken**, передав в конструктор 0 (по той же причине, что и в случае вещественной константы). Иначе возвращаем **ConstantToken**, передав в конструктор полученное целочисленное значение.

- Если текущая литера является буквой, либо символом «\_», то вызываем метод сканирования идентификатор и ключевых слов **ScanKeywordOrIdentifier**. В этом методе запрашиваются и запоминаются в строку литеры до тех пор, пока они являются буквами, цифрами либо символом нижнего подчеркивания. Когда условие перестаёт выполняться, передаём полученную строку в метод **IsOperation** класса **OperationMatcher**, если переданная строка является оператором, обращаемся уже к методу **GetOperation** того же класса и возвращаем **OperationToken**, хранящий полученное значение. Иначе имеем дело с идентификатором и необходимо проверить его длину. Если его длина превосходит 127 символов, формируем соответствующую ошибку, но в любом случае возвращаем **IdentifierToken**, передав в конструктор полученное значение.
- Если текущая литера принадлежит следующему набору: «)», «;», «=», «,», «^», «]», «+», «-», «/», «{» «}», то вызываем метод **ScanSingleCharacterOperator**. В этом методе необходимо отдельно обработать литеры «{» и «}», вызвав для них методы **ScanLeftBrace** и **ScanRightBrace** соответственно. Поскольку ни один из составных

оператор не начинается с литер, входящих в этот набор, можно однозначно определить коды этих операторов, вызвав метод **GetOperation** из класса **OperationMatcher**, после чего вернуть **OperationToken** с полученным значением.

- В методе **ScanLeftBrace** запрашиваем литеры у модуля ввода-вывода и пропускаем их до тех пор, пока не встретим «}», символ перевода строки, либо не дойдём до конца файла. Если встретили литеру «}», то запрашиваем у модуля ввода-вывода следующий символ и вызываем метод **GetNextToken**. Иначе формируем сообщением об ошибке. Далее смотрим, если дошли до конца файла, то возвращаем **null**, иначе вызываем метод **GetNextToken**.
- В методе **ScanRightBrace** просто формируем ошибку незакрытого комментария, запрашиваем у модуля ввода-вывода следующую литеру и вызываем метод **GetNextToken**.
- Если текущая литера принадлежит набору – «(», «\*», «<», «>», «:», «.», то вызываем метод **ScanMultipleCharacterOperator**. В этом методе смотрим, если получили литеру «(» или «\*» вызываем методы **ScanLeftParenthesis** и **ScanAsterisk** соответственно. В противном случае запоминаем текущую литеру, читаем следующую, объединяем их в строку и передаём в метод **IsOperation** класса **OperationMatcher**, если переданная строка является оператором, читаем следующую литеру и возвращаем **OperationToken**, передав в конструктор полученную операцию, иначе возвращаем **OperationToken**, представляющий операцию предыдущего прочитанного символа.
- В методе **ScanLeftParenthesis** сначала читаем следующую литеру, если она отличается от «\*», возвращаем **OperationToken**, хранящий информацию об операции «(». Иначе запрашиваем у модуля ввода-вывода следующие литеры, пока не получим, что соседние литеры

равны «\*» и «)» либо не дойдём до конца файла. Если дошли до конца файла, то формируем ошибку незакрытого комментария и возвращаем `null`. Иначе читаем следующую литеру и вызываем метод **GetNextToken**.

- В методе **ScanAsterisk** запрашиваем у модуля ввода-вывода следующую литеру. Если получаем «)», то формируем сообщение об ошибке незакрытого комментария, читаем следующую литеру и вызываем метод **GetNextToken**. Иначе возвращаем **OperationToken**, хранящий информацию о лексеме «\*».

Стоит упомянуть, что во всех вышеописанных методах, в конструкторы классов, наследованных от **Token**, также передаётся позиция первой литеры этого токена в тексте исходной программы.

#### 4. Тестирование

Необходимо протестировать правильность построения токенов, корректность обработки комментариев, а также работу с лексическими ошибками.

Проверка правильности построения токенов будет осуществляться выводом информации о них в консоль. Для этого был переопределён метод **ToString** в производных классах класса **Token**.

- 1) Для тестирования правильности построения токенов была написана программа на языке Pascal, включающая в себя все элементы подмножества языка, компилятор для которого необходимо написать.

```
input.pas
1  program Hello;
2  var
3      a : integer;
4      b, c : real;
5      d : string;
6  begin
7      a:=8;
8      b := a / 3 + (a + 12) MOD 14;
9      d := 'test string';
10
11      if ((a + 5) div 3 > 2) OR (b < 0) then
12          b := 323232.325
13      else
14          c:= 45.3;
15
16      while a > 0 do
17          begin
18              b := b * a;
19              a := a - 1;
20          end;
21  end.
```

Результат вывода в консоль:

```
Type: Operation | Value: Program | Start position: 1 | 1 строка
Type: Identifier | Value: Hello | Start position: 9
Type: Operation | Value: Semicolon | Start position: 14
Type: Operation | Value: Var | Start position: 1 | 2 строка
Type: Identifier | Value: a | Start position: 5
Type: Operation | Value: Colon | Start position: 7
Type: Identifier | Value: integer | Start position: 9
Type: Operation | Value: Semicolon | Start position: 16
Type: Identifier | Value: b | Start position: 5
Type: Operation | Value: Comma | Start position: 6
Type: Identifier | Value: c | Start position: 8
Type: Operation | Value: Colon | Start position: 10
Type: Identifier | Value: real | Start position: 11
Type: Operation | Value: Semicolon | Start position: 15
Type: Identifier | Value: d | Start position: 5
Type: Operation | Value: Colon | Start position: 7
Type: Identifier | Value: string | Start position: 8
Type: Operation | Value: Semicolon | Start position: 14
Type: Operation | Value: Begin | Start position: 1 | 5 строка
Type: Identifier | Value: a | Start position: 5
Type: Operation | Value: Assignment | Start position: 6
Type: Constant | Type: Integer | Value: 8 | Start position: 8 | 6 строка
Type: Operation | Value: Semicolon | Start position: 9
Type: Identifier | Value: b | Start position: 5
Type: Operation | Value: Assignment | Start position: 7
Type: Identifier | Value: a | Start position: 10
Type: Operation | Value: Slash | Start position: 12
Type: Constant | Type: Integer | Value: 3 | Start position: 14
Type: Operation | Value: Plus | Start position: 16
Type: Operation | Value: LeftParenthesis | Start position: 18
Type: Identifier | Value: a | Start position: 19
Type: Operation | Value: Plus | Start position: 21
Type: Constant | Type: Integer | Value: 12 | Start position: 23
Type: Operation | Value: RightParenthesis | Start position: 25
Type: Operation | Value: Mod | Start position: 27
Type: Constant | Type: Integer | Value: 14 | Start position: 31
Type: Operation | Value: Semicolon | Start position: 33
Type: Identifier | Value: d | Start position: 5
Type: Operation | Value: Assignment | Start position: 7
Type: Constant | Type: String | Value: test string | Start position: 10
Type: Operation | Value: Semicolon | Start position: 23 | 8 строка
Type: Operation | Value: Semicolon | Start position: 23 | 9 строка
```



Type: Operation	Value: If	Start position: 5	
Type: Operation	Value: LeftParenthesis	Start position: 8	
Type: Operation	Value: LeftParenthesis	Start position: 9	
Type: Identifier	Value: a	Start position: 10	
Type: Operation	Value: Plus	Start position: 12	
Type: Constant	Type: Integer	Value: 5	Start position: 14
Type: Operation	Value: RightParenthesis	Start position: 15	
Type: Operation	Value: Div	Start position: 17	
Type: Constant	Type: Integer	Value: 3	Start position: 21
Type: Operation	Value: Greater	Start position: 23	Строка 11
Type: Constant	Type: Integer	Value: 2	Start position: 25
Type: Operation	Value: RightParenthesis	Start position: 26	
Type: Operation	Value: Or	Start position: 28	
Type: Operation	Value: LeftParenthesis	Start position: 31	
Type: Identifier	Value: b	Start position: 32	
Type: Operation	Value: Less	Start position: 34	
Type: Constant	Type: Integer	Value: 0	Start position: 36
Type: Operation	Value: RightParenthesis	Start position: 37	
Type: Operation	Value: Then	Start position: 39	
Type: Identifier	Value: b	Start position: 9	Строка 12
Type: Operation	Value: Assignment	Start position: 11	
Type: Constant	Type: Real	Value: 323232.325	Start position: 14
Type: Operation	Value: Else	Start position: 5	Строка 13
Type: Identifier	Value: c	Start position: 9	
Type: Operation	Value: Assignment	Start position: 10	Строка 14
Type: Constant	Type: Real	Value: 45.3	Start position: 13
Type: Operation	Value: Semicolon	Start position: 17	
Type: Operation	Value: While	Start position: 5	
Type: Identifier	Value: a	Start position: 11	Строка 16
Type: Operation	Value: Greater	Start position: 13	
Type: Constant	Type: Integer	Value: 0	Start position: 15
Type: Operation	Value: Do	Start position: 17	
Type: Operation	Value: Begin	Start position: 9	Строка 17
Type: Identifier	Value: b	Start position: 13	
Type: Operation	Value: Assignment	Start position: 15	
Type: Identifier	Value: b	Start position: 18	
Type: Operation	Value: Asterisk	Start position: 20	Строка 18
Type: Identifier	Value: a	Start position: 22	
Type: Operation	Value: Semicolon	Start position: 23	
Type: Identifier	Value: a	Start position: 13	
Type: Operation	Value: Assignment	Start position: 15	
Type: Identifier	Value: a	Start position: 18	Строка 19
Type: Operation	Value: Minus	Start position: 20	
Type: Constant	Type: Integer	Value: 1	Start position: 22
Type: Operation	Value: Semicolon	Start position: 23	
Type: Operation	Value: End	Start position: 9	Строка 20
Type: Operation	Value: End	Start position: 12	
Type: Operation	Value: End	Start position: 1	Строка 21
Type: Operation	Value: Point	Start position: 4	

Как видно из результатов тестирования, лексический анализатор корректно определяет типы токенов, их значения, а также позицию первой литеры токена.

Хоть в этой программе используются и не все необходимые ключевые слова подмножества языка, дальнейшее тестирование можно пропустить в силу того, что токены строятся унифицировано в методах **ScanSingleCharacterOperator** и **ScanMultipleCharacterOperator**.

2) Добавим в программу однострочные и многострочные комментарии:

```
input.pas
1  program Hello; { This is a single-line comment! }
2  var
3      a : integer;
4      b, c : real;
5      {And this is another one...} d : string;
6  begin
7      a:=8;
8      b := a / 3 + (a + 12) MOD 14;
9      d := 'test string';
10
11      if ((a + 5) div 3 > 2) OR (b < 0) then
12          b := 323232.325
13      else
14          c:= 45.3;
15
16      (*
17          Here comes
18          a multi-line comment
19      *)
20
21      while a > 0 do
22          begin
23              b := b * a;
24              a := a - 1;
25          end;
26  end.
27
```

Результат вывода в консоль:

```
Type: Operation | Value: Program | Start position: 1
Type: Identifier | Value: Hello | Start position: 9
Type: Operation | Value: Semicolon | Start position: 14
Type: Operation | Value: Var | Start position: 1
Type: Identifier | Value: a | Start position: 5
Type: Operation | Value: Colon | Start position: 7
Type: Identifier | Value: integer | Start position: 9
Type: Operation | Value: Semicolon | Start position: 16
Type: Identifier | Value: b | Start position: 5
Type: Operation | Value: Comma | Start position: 6
Type: Identifier | Value: c | Start position: 8
Type: Operation | Value: Colon | Start position: 10
Type: Identifier | Value: real | Start position: 11
Type: Operation | Value: Semicolon | Start position: 15
Type: Identifier | Value: d | Start position: 34
Type: Operation | Value: Colon | Start position: 36
Type: Identifier | Value: string | Start position: 37
Type: Operation | Value: Semicolon | Start position: 43
Type: Operation | Value: Begin | Start position: 1
Type: Identifier | Value: a | Start position: 5
Type: Operation | Value: Assignment | Start position: 6
Type: Constant | Type: Integer | Value: 8 | Start position: 8
Type: Operation | Value: Semicolon | Start position: 9
Type: Identifier | Value: b | Start position: 5
Type: Operation | Value: Assignment | Start position: 7
Type: Identifier | Value: a | Start position: 10
Type: Operation | Value: Slash | Start position: 12
Type: Constant | Type: Integer | Value: 3 | Start position: 14
Type: Operation | Value: Plus | Start position: 16
Type: Operation | Value: LeftParenthesis | Start position: 18
Type: Identifier | Value: a | Start position: 19
Type: Operation | Value: Plus | Start position: 21
Type: Constant | Type: Integer | Value: 12 | Start position: 23
Type: Operation | Value: RightParenthesis | Start position: 25
Type: Operation | Value: Mod | Start position: 27
Type: Constant | Type: Integer | Value: 14 | Start position: 31
Type: Operation | Value: Semicolon | Start position: 33
Type: Identifier | Value: d | Start position: 5
Type: Operation | Value: Assignment | Start position: 7
Type: Constant | Type: String | Value: test string | Start position: 10
Type: Operation | Value: Semicolon | Start position: 23
```

```

Type: Operation | Value: If | Start position: 5
Type: Operation | Value: LeftParenthesis | Start position: 8
Type: Operation | Value: LeftParenthesis | Start position: 9
Type: Identifier | Value: a | Start position: 10
Type: Operation | Value: Plus | Start position: 12
Type: Constant | Type: Integer | Value: 5 | Start position: 14
Type: Operation | Value: RightParenthesis | Start position: 15
Type: Operation | Value: Div | Start position: 17
Type: Constant | Type: Integer | Value: 3 | Start position: 21
Type: Operation | Value: Greater | Start position: 23
Type: Constant | Type: Integer | Value: 2 | Start position: 25
Type: Operation | Value: RightParenthesis | Start position: 26
Type: Operation | Value: Or | Start position: 28
Type: Operation | Value: LeftParenthesis | Start position: 31
Type: Identifier | Value: b | Start position: 32
Type: Operation | Value: Less | Start position: 34
Type: Constant | Type: Integer | Value: 0 | Start position: 36
Type: Operation | Value: RightParenthesis | Start position: 37
Type: Operation | Value: Then | Start position: 39
Type: Identifier | Value: b | Start position: 9
Type: Operation | Value: Assignment | Start position: 11
Type: Constant | Type: Real | Value: 323232.325 | Start position: 14
Type: Operation | Value: Else | Start position: 5
Type: Identifier | Value: c | Start position: 9
Type: Operation | Value: Assignment | Start position: 10
Type: Constant | Type: Real | Value: 45.3 | Start position: 13
Type: Operation | Value: Semicolon | Start position: 17
Type: Operation | Value: While | Start position: 5
Type: Identifier | Value: a | Start position: 11
Type: Operation | Value: Greater | Start position: 13
Type: Constant | Type: Integer | Value: 0 | Start position: 15
Type: Operation | Value: Do | Start position: 17
Type: Operation | Value: Begin | Start position: 9
Type: Identifier | Value: b | Start position: 13
Type: Operation | Value: Assignment | Start position: 15
Type: Identifier | Value: b | Start position: 18
Type: Operation | Value: Asterisk | Start position: 20
Type: Identifier | Value: a | Start position: 22
Type: Operation | Value: Semicolon | Start position: 23
Type: Identifier | Value: a | Start position: 13
Type: Operation | Value: Assignment | Start position: 15
Type: Identifier | Value: a | Start position: 18
Type: Operation | Value: Minus | Start position: 20
Type: Constant | Type: Integer | Value: 1 | Start position: 22
Type: Operation | Value: Semicolon | Start position: 23
Type: Operation | Value: End | Start position: 9
Type: Operation | Value: Semicolon | Start position: 12
Type: Operation | Value: End | Start position: 1
Type: Operation | Value: Point | Start position: 4

```

Как видно из результатов вывода в консоль, лексический анализатор корректно распознаёт и игнорирует как однострочные, так и многострочные комментарии.

3) Теперь добавим в программу все описанные виды лексических ошибок:

[illegible]

## Содержание выходного файла:

```

1      01 program Hello; { {Открытие незакрытого комментария!}
2      **01**                ^ Код ошибки: 1
3      ***** Открытие незакрытого комментария
4      02 var
5          03     a : integer;
6          04     b, c :real;
7          05     d :string; } {Закрытие неоткрытого комментария!}
8      **02**                ^ Код ошибки: 2
9      ***** Закрытие неоткрытого комментария
10     06 begin
11         07     a:=8;
12         08     b := a / 3 + (a + 12) MOD 143232954545454433; {Значение целочисленной константы превышает предел!}
13     **03**                ^ Код ошибки: 5
14     ***** Значение целочисленной константы превышает предел
15     09     d := 'test string
16     **04**                ^ Код ошибки: 3
17     ***** Ошибка в описании строковой константы
18         10         and also error
19         11         '; {Ошибка в описании строковой константы!"}}
20         12
21         13         if ((a + 5) div 3 > 2) OR (b < 0) then
22             14             b := 323232.325
23             15         else
24                 16             c:= 45.3.18; {Ошибка в описании вещественной константы!}
25     **05**                ^ Код ошибки: 4
26     ***** Ошибка в описании вещественной константы
27     17
28     18 while aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa > 0 do {Длина идентификатора превышает предел}
29     **06**                ^ Код ошибки: 6
30     ***** Длина идентификатора превышает предел
31         19         beGin
32             20             b := b * a;
33             21             a := a - 1;
34             End;
35         23 end.! {Запрещённый символ!}
36     **07**                ^ Код ошибки: 7
37     ***** Запрещённый символ
38
39     Всего ошибок – 7
40

```

Как видно из содержания выходного файла, лексический анализатор корректно обнаруживает все лексические ошибки, при этом игнорирует их внутри комментариев.

# Синтаксический и семантический анализ

## 1. Описание

Синтаксический анализ необходим для проверки соответствия текста программы формальным правилам описания языка. Для этого удобно воспользоваться формами Бэкуса-Наура, сопоставив каждой конструкции форму.

Всё подмножество языка Pascal, компилятор для которого необходимо реализовать можно описать следующими формами Бэкуса-Наура:

- $\langle \text{программа} \rangle ::= \text{program } \langle \text{имя} \rangle ; \langle \text{блок} \rangle .$
- $\langle \text{блок} \rangle ::= \langle \text{раздел переменных} \rangle \langle \text{раздел операторов} \rangle$
- $\langle \text{раздел переменных} \rangle ::= \text{var } \langle \text{описание однотипных переменных} \rangle ;$   
 $\{ \langle \text{описание однотипных переменных} \rangle ; \} | \langle \text{пусто} \rangle$
- $\langle \text{описание однотипных переменных} \rangle ::= \langle \text{имя} \rangle \{ , \langle \text{имя} \rangle \} : \langle \text{тип} \rangle$
- $\langle \text{раздел операторов} \rangle ::= \langle \text{составной оператор} \rangle$
- $\langle \text{составной оператор} \rangle ::= \text{begin } \langle \text{оператор} \rangle \{ ; \langle \text{оператор} \rangle \} \text{end}$
- $\langle \text{оператор} \rangle ::= \langle \text{оператор присваивания} \rangle | \langle \text{пустой оператор} \rangle |$   
 $\langle \text{составной оператор} \rangle | \langle \text{выбирающий оператор} \rangle | \langle \text{оператор цикла} \rangle$
- $\langle \text{оператор присваивания} \rangle ::= \langle \text{переменная} \rangle : \langle \text{выражение} \rangle$
- $\langle \text{переменная} \rangle ::= \langle \text{имя} \rangle$
- $\langle \text{выражение} \rangle ::= \langle \text{простое выражение} \rangle |$   
 $\langle \text{простое выражение} \rangle \langle \text{операция отношения} \rangle \langle \text{простое выражение} \rangle |$
- $\langle \text{операция отношения} \rangle ::= = | < | < = | > | > =$
- $\langle \text{простое выражение} \rangle ::= \langle \text{знак} \rangle \langle \text{слагаемое} \rangle \{ \langle \text{аддитивная операция} \rangle \langle \text{слагаемое} \rangle \}$
- $\langle \text{аддитивная операция} \rangle ::= + | - | \text{or}$
- $\langle \text{слагаемое} \rangle ::= \langle \text{множитель} \rangle \{ \langle \text{мультипликативная операция} \rangle \langle \text{множитель} \rangle \}$
- $\langle \text{мультипликативная операция} \rangle ::= * | / | \text{div} | \text{mod} | \text{and}$
- $\langle \text{множитель} \rangle ::= \langle \text{переменная} \rangle | ( \langle \text{выражение} \rangle ) | \langle \text{константа} \rangle$

- *< константа > ::= < число > | < строка >*
- *< пустой оператор > ::= < пусто >*
- *< выбирающий оператор > ::= < условный оператор >*
- *< условный оператор > ::= if < выражение > then < оператор > |  
if < выражение > then < оператор > else < оператор >*
- *< оператор цикла > ::= < цикл с предусловием >*
- *< цикл с предусловием > ::= while < выражение > do < оператор >*

При анализе могут возникнуть синтаксические ошибки, например, если в тексте исходной программы пропущена точка с запятой. Если не заняться нейтрализацией таких ошибок, то компилятор будет корректно работать до первой обнаруженной ошибки. Основная идея нейтрализации синтаксических ошибок – пропустить какую-то часть программы до момента, с которого можно будет продолжить дальнейший анализ.

Помимо формальных правил описания языка, есть и неформальные. Например, в одной области видимости идентификатор не может быть описан более одного раза, каждому прикладному вхождению идентификатора должно найтись определяющее вхождение и т.д. Проверку таких правил осуществляет семантический анализатор.

## 2. Проектирование

Реализация синтаксического и семантического анализаторов расположена в классе **Compiler**. Одним из полей этого класса является класс **LexicalAnalyzer**, поэтому у класса **Compiler** есть публичный конструктор, принимающий пути до входного и выходного файлов, которые затем будут переданы в конструктор класса **LexicalAnalyzer**. Помимо конструктора, в классе есть один публичный метод **start**, который начинает работу компилятора.

Также в классе есть приватный метод **GetNextToken**, который вызывает одноимённый метод лексического анализатора и присваивает результат в

поле **currentToken** и **AddError**, которые вызывают соответствующие методы лексического анализатора.

Ещё одним из полей класса **Compiler** является экземпляр класса **Scope**. Этот класс используется для семантического анализа. Он представляет из себя словарь для хранения имён переменных, содержащий **IdentifierToken** в качестве ключа, и **Type** в качестве значения. Также в этом классе имеются метод проверки доступности типа, метод проверки, не была ли переменная уже описана, а также методы добавления новой переменной в зависимости от её типа.

Класс **Type** является абстрактным, его единственным полем является значение перечисления **ValueType**, созданного специально для идентификации типов. Также в классе абстрактные методы. Первый метод принимает другой экземпляр класса **Type** и проверяет, приводим ли текущий тип к переданному. Второй метод принимает значение перечисления **Operation** и проверяет, поддерживает ли тип эту операцию.

От класса **Type** наследуются следующие классы: **IntegerType**, **RealType**, **StringType**, **BooleanType** и **UnkownType**. Во всех классах, за исключением последнего, в качестве единственного собственного поля содержится список с операциями, поддерживаемыми данным типом.

Класс **UnkownType** создан для нейтрализации семантических ошибок. Он используется в качестве типа необъявленной переменной, либо если одна переменная была объявлена несколько раз с разными типами. Особенность этого типа заключается в том, что он приводим ко всем остальным типам, а все остальные типы, в свою очередь, приводимы, к нему. Также этот тип поддерживает все операции.

Для упрощения работы с типами из класса **Compiler** был создан статический класс **Types**. Этот класс содержит словарь, в качестве ключей которого выступают строки, а в качестве значений – экземпляры иерархии класса **Type**. Этот словарь изначально инициализирован всеми типами.





проверять, не были ли переменные уже описаны, и если были добавлять ошибки, проверять тип на доступность и т.д.

Также необходимо переработать метод, советующий оператору присваивания. Необходимо проверять, чтобы используемая переменная была описана, а тип выражения после оператора был приводим к типу переменной.

Необходимо также проверять, чтобы тип данных выражения, используемого в условном и циклическом операторах, был логический.

Самым большим переработкам подверглись методы, отвечающие за БНФ «Выражение», «Простое выражение», «Слагаемое», «Множитель». В них должна проводится проверка типов на приводимость, а также используемые операции на легитимность. Также эти методы должны возвращать тип получившегося выражения. В этих методах могут возникнуть разнообразные ошибки, связанные с неприводимостью типов, неверной операцией, синтаксическими ошибками в выражении.

Для более удобной спецификации этих ошибок, а также позиции в строке, на которой они возникли был создан абстрактный класс исключения **ExpressionException**, хранящий позицию возникновения ошибки. От этого класса наследуются классы **OperatorException**, **TypeException** и **OperationException**

```

classDiagram
    class Compiler {
        <<static>>
        NextTokens
        + Program: List<Operation>
        + SameTypeVariables: List<Operation>
        + OperatorPartStart: List<Operation>
        + OperatorPartEnd: List<Operation>
        + OperatorStart: List<Operation>
        + OperatorEnd: List<Operation>
    }

    class LexicalAnalyzer {
        <<static>>
        LexicalAnalyzer
        + scope: Scope
        + currentToken: Token
        + Start(): void
        + GetNextToken(): Token
        + AcceptOperation(): void
        + AcceptIdentifier(): void
        + SkipTokenToLastOperation(): bool
        + Program(): void
        + Block(): void
        + Term(): Type
        + Factor(): Type
        + AddError(msg): void
        + AddErrorMsg(msg): void
        + HandleExpressionException(Exception): bool
        + GetVariableType(): Type
        + GetConstantType(): Type
    }

    class ExpressionException {
        <<abstract>>
        # errorPosition: int
        + GetErrorPosition(): int
    }

    class OperatorException {
        <<abstract>>
    }

    class TypeException {
        <<abstract>>
    }

    class OperationException {
        <<abstract>>
    }

    class Types {
        <<static>>
        + typesTable: Dictionary<string, Type>
        + GetType(string): Type
        + AreTypesDerive(Type, Type): bool
        + DeriveTypes(Type, Type): Type
    }

    class Scope {
        <<static>>
        + variablesTable: Dictionary<IdentifierToken, Type>
        + IsTypeAvailable(IdentifierToken): bool
        + IsVariableDescribed(IdentifierToken): bool
        + AddVariables(IdentifierToken, IdentifierToken): void
        + AddVariable(IdentifierToken): void
        + GetVariableType(IdentifierToken): Type
        + GetVariable(IdentifierToken): IdentifierToken
    }

    class Type {
        <<abstract>>
        # valueType: ValueType
        + GetValueType(): TokenType
        + IsDerivedType(): bool
        + IsOperationSupported(Operation): bool
    }

    class RealType {
        <<static>>
        + supportedOperations: List<Operation>
    }

    class UnknownType {
        <<static>>
        + supportedOperations: List<Operation>
    }

    class StringType {
        <<static>>
        + supportedOperations: List<Operation>
    }

    class OperationErrorMatcher {
        <<static>>
        + dictionary: Dictionary<Operation, int>
        + GetErrorCode(Operation): int
    }

    class ErrorMatcher {
        <<static>>
        + dictionary: Dictionary<int, string>
        + GetErrorDescription(int): string
    }

    class IOModule {
        + lines: string[]
        + outputPath: string
        + lineNumber: int
        + characterNumber: int
        + error: List<Error>
        + GetCharacterNumber(): int
        + AddError(msg, int): void
        + ReadNextCharacter(): char
        + WriteLine(): void
    }

    class Error {
        + code: int
        + position: int
        + GetCode(): int
        + GetPosition(): int
    }

    class LexicalAnalyzer {
        + MaxInValue: int const
        + MaxIdentifierLength: int const
        + IOModule: IOModule
        + currentCharacter: char
        + startPosition: int
        + GetNextToken(): Token
        + ScanString(): Token
        + ScanNumber(): Token
        + ScanKeywordOrIdentifier(): Token
        + ScanSingleCharacterOperator(): Token
        + ScanMultipleCharacterOperator(): Token
        + ScanLeftParenthesis(): Token
        + ScanRightParenthesis(): Token
        + ScanAsterisk(): Token
        + ScanAmpersand(): Token
        + ScanRightShift(): Token
        + ScanRightShift(): Token
        + AddError(msg): void
        + AddErrorMsg(msg): void
    }

    class OperationMatcher {
        <<static>>
        + dictionary: Dictionary<string, Operation>
        + IsOperation(string): bool
        + GetOperation(string): Operation
    }

    class Token {
        <<abstract>>
        # type: TokenType
        # startPosition: int
        + GetType(): TokenType
        + GetStartPosition(): int
    }

    class IdentifierToken {
        <<enumeration>>
        TokenType
        + identifier: string
        + GetIdentifier(): string
    }

    class OperationToken {
        <<enumeration>>
        TokenType
        + operation: Operation
        + GetOperation(): Operation
    }

    class ConstantToken {
        <<enumeration>>
        TokenType
        + constant: Variant
        + GetVariant(): Variant
    }

    class Variant {
        <<abstract>>
        # type: VariantType
        + GetType(): VariantType
    }

    class IntegerVariant {
        <<enumeration>>
        VariantType
        + value: int
        + GetValue(): int
    }

    class RealVariant {
        <<enumeration>>
        VariantType
        + value: double
        + GetValue(): double
    }

    class StringVariant {
        <<enumeration>>
        VariantType
        + value: string
        + GetValue(): string
    }

    Compiler * LexicalAnalyzer
    Compiler * ExpressionException
    Compiler * OperatorException
    Compiler * TypeException
    Compiler * OperationException
    Compiler * Types
    Compiler * Scope
    Compiler * Type
    Compiler * RealType
    Compiler * UnknownType
    Compiler * StringType
    Compiler * OperationErrorMatcher
    Compiler * ErrorMatcher
    Compiler * IOModule
    Compiler * Error
    Compiler * LexicalAnalyzer
    Compiler * OperationMatcher
    Compiler * Token
    Compiler * IdentifierToken
    Compiler * OperationToken
    Compiler * ConstantToken
    Compiler * Variant
    Compiler * IntegerVariant
    Compiler * RealVariant
    Compiler * StringVariant
  
```

The diagram illustrates the architecture of a compiler, organized into several main components and their interactions:

- Compiler** (Central Component):
  - Manages **NextTokens** and various operation lists (Program, SameTypeVariables, OperatorPartStart/End, OperatorStart/End).
  - Coordinates with the **LexicalAnalyzer** and **ExpressionException**.
  - Manages **Types**, **Scope**, and **Type** information.
  - Handles **OperationErrorMatcher**, **ErrorMatcher**, and **IOModule**.
  - Manages **Error** objects.
- LexicalAnalyzer**:
  - Processes input into tokens, managing **MaxInValue**, **MaxIdentifierLength**, and the **IOModule**.
  - Scans for various tokens: strings, numbers, keywords, operators, parentheses, asterisks, and ampersands.
  - Reports errors via **AddError** and **AddErrorMsg**.
- ExpressionException** and **OperationException**:
  - Abstract base classes for handling errors during expression evaluation and operation matching.
- Types** and **Type**:
  - Types** manages a table of types and provides methods to get types, check derivation, and derive types.
  - Type** is an abstract base class for **RealType**, **UnknownType**, and **StringType**, defining methods for value types, derived types, and operation support.
- OperationMatcher** and **ErrorMatcher**:
  - OperationMatcher** maps operation strings to their corresponding **Operation** objects.
  - ErrorMatcher** maps error codes to their corresponding error descriptions.
- IOModule** and **Error**:
  - IOModule** handles file input/output, including line and character numbers.
  - Error** objects track error codes and positions.
- Token** and **Variant**:
  - Token** is an abstract base class for **IdentifierToken**, **OperationToken**, and **ConstantToken**.
  - Variant** is an abstract base class for **IntegerVariant**, **RealVariant**, and **StringVariant**.

### 3. Реализация

В методе описания одноптипных переменных я завожу список **IdentifierToken** для сохранения переменных. Далее последовательно,

пока идут идентификаторы, разделённые оператором «**,**» я добавляю их в список. Эта последовательность должна закончиться оператором «**:**», после которого должно идти название типа. Если встречаются какие-то синтаксические ошибки, я бросаю исключение и пропускаю токены до «**;**», оператора **begin** или «**.**». Если синтаксических ошибок нет, необходимо добавлять переменные в словарь с переменными, одновременно проверяя наличие семантических ошибок. Изначально проверяется тип, если он недоступен, то все переменные будут добавлены в словарь с неизвестным типом. Также необходимо проверять, переменные на повторное описание, совпадение названия переменной с названием типа. Если переменная описывается несколько раз с разными типами, необходимо также изменить её тип на неизвестный в таблице переменных.

В методе раздела описания операторов мы ожидаем в начале увидеть ключевое слово **begin**, если встретили другой токен, то добавляем ошибку и пропускает токены либо до ключевого слова **begin**, либо до конца программы. Далее вызывается метод **Operator**.

Реализация синтаксического анализа и нейтрализации ошибок в этом методе вызвали у меня больше всего затруднений. Если встретили идентификатор, вызываем метод **AssignmentOperator**, если ключевое слово **begin**, то метод **CompoundOperator**, если ключевые слова **if** или **while** – соответствующие им методы. В противном случае – пропускаем токены до оператора «**;**», либо ключевого слова **end**.

В методе **CompoundOperator** изначально принимается ключевое слово **begin**, далее вызывается метод **Operator**, пока текущий токен равен «**;**», принимаем этот токен и опять вызываем метод **Operator**. В конце работы методы принимаем ключевое слово **end**.

В методе оператора присваивания изначально принимается **IdentifierToken**. Проверяется, была ли такая переменная описана, если нет, она добавляется в словарь переменных с неизвестным типом. Далее

запоминается тип переменной, после чего принимается оператор присваивания и вызывается метод **Expression**. Если вместо оператора присваивания был получен другой токен, либо если из выражения прилетело исключение, происходит нейтрализация, в ходе которой пропускаются все токены до «;» или **end**. Иначе проверяется тип, полученный из выражения на приводимость к типу переменной. Если типы не приводимы, формируется сообщение об ошибке.

В методе **IfOperator** принимается ключевое слово **if**, далее вызывается метод **Expression**, если он вернул не логический тип, необходимо добавить сообщение об ошибке, далее принимается ключевое слово **then**. Если произошли какие-либо синтаксические ошибки, то пропускаем токены до идентификатора или ключевого слова из набора **begin, if, while, end**. После этого вызывается метод **Operator**. Если текущий токен равняется ключевому слову **else**, то принимаем его и опять вызываем метод **Operator**.

Работа метода **whileOperator** схожа с работой первой половины предыдущего метода, только за место ключевого слова **then** принимается ключевое слово **do**.

Для реализации выражения были разработаны методы **Expression, SimpleExpression, Term, Factor, IsAdditiveOperation, IsMultiplicativeOperation, IsLogicalOperation**, соответствующие БНФ. Помимо них были реализованы вспомогательные методы: **IsOperation, GetVariableType, GetConstantType**.

Последние три метода, соответствующие БНФ, просто вызывают метод **IsOperation** с соответствующими параметрами. Этот метод проверяет, принадлежит ли операция текущего токена переданному набору.

Метод **Factor** возвращает тип текущего множителя. Если текущий токен соответствует переменной, то он вызывает метод **GetVariableType**, если константа – **GetConstantType**, иначе он должен принять операцию «(»,

после чего вызвать метод **Expression**, после чего принять операцию «)».

Если произошла синтаксическая ошибка – бросаем соответствующее исключение, иначе – возвращаем значение, которое было получено при вызове соответствующего метода.

В методе **GetVariableType** проверяется, была ли такая переменная уже описана, если нет, то добавляет её с неизвестным типом. Метод в любом случае возвращает тип переменной.

Метод **Term** вызывает метод **Factor**, запоминая полученное значение, далее, пока текущий токен является мультипликативной операцией (метод **IsMultiplicativeOperation**), запоминает операцию, далее опять вызывает метод **Factor**, запоминая полученное значение. Если типы неприводимы, либо полученный в ходе приведения тип не поддерживает текущую мультипликативную операцию – бросаем исключение. Иначе возвращаем полученный в ходе приведения тип.

Метод **SimpleExpression** схож с предыдущим методом, однако вместо метода **Factor** он вызывает метод **Term**, а вместо мультипликативной операции проверяет аддитивную (метод **IsAdditiveOperation**).

Метод **Expression** сначала вызывает метод **SimpleExpression**, запоминая полученный тип, далее проверяет, является ли текущий токен операцией отношения (метод **IsLogicalOperation**), если является, снова вызывает метод **SimpleExpression**, запоминая полученный тип. Если типы приводимы, возвращает тип `boolean`, иначе - бросает ошибку неприводимости типов.

В классе **Scope** содержится словарь, где ключ – **IdentifierToken**, а значение – экземпляр класса **Type**.

При проверке допустимости типа в методе **IsTypeAvailable** необходимо проверить, что имя типа – одно из следующих значений `integer`, `real`, `string`, а также название ни одной из переменных не совпадает с названием переданного типа.

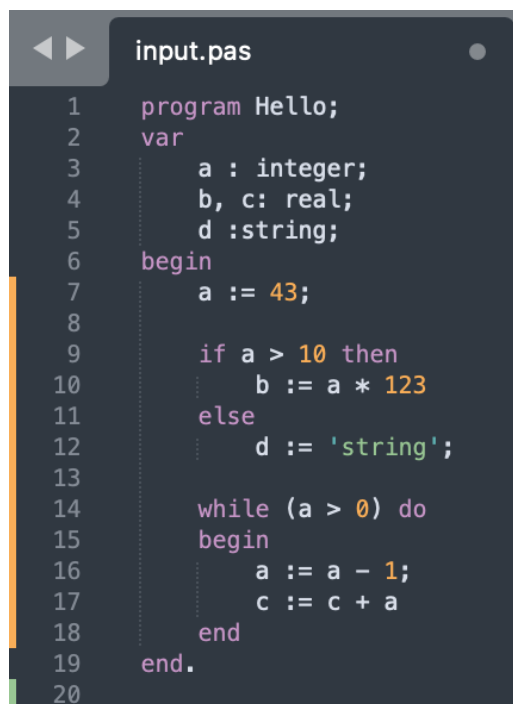
При проверке, описана ли переменная в методе **IsVariableDescribed**, просто проходимся по всем ключам и смотрит идентификаторы.

При добавлении новой переменной в методе **AddVariable** сначала проверяем, не была ли переменная уже описана с другим типом. Если была – удаляем её из словаря и добавляем новую с неизвестным типом. Иначе просто добавляем переменную с переданным типом.

#### 4. Тестирование

При тестировании необходимо проверить корректность работы компилятора на программе без ошибок, а также обнаружение и нейтрализацию синтаксических и семантических ошибок.

1) Пример корректной программы на языке Pascal:



```
input.pas
1  program Hello;
2  var
3      a : integer;
4      b, c: real;
5      d :string;
6  begin
7      a := 43;
8
9      if a > 10 then
10         b := a * 123
11     else
12         d := 'string';
13
14     while (a > 0) do
15     begin
16         a := a - 1;
17         c := c + a
18     end
19 end.
```

Содержимое выходного файла:

```
output.txt
1      01  program Hello;
2      02  var
3      03      a : integer;
4      04      b, c: real;
5      05      d :string;
6      06  begin
7      07      a := 43;
8      08
9      09      if a > 10 then
10     10          b := a * 123
11     11      else
12     12          d := 'string';
13     13
14     14      while (a > 0) do
15     15          begin
16     16              a := a - 1;
17     17              c := c + a;
18     18          end
19     19
20     20  end.
21
22  Всего ошибок - 0
```

Как видно из выходного файла, компилятор успешно дообработал до конца программы, при этом не обнаружил ошибок там, где их нет!

- 2) Теперь попробуем изменить раздел описания переменных таким образом, чтобы в нём возникли как синтаксические, так и семантические ошибки. В результате такого изменения, семантические ошибки появятся и в разделе операторов. Также добавим операторы присваивания в разделе операторов:

```
input.pas
1  program Hello;
2  var
3      integer : integer;
4      temp, variable :
5      a : integer;
6      b, c: real;
7      c, d :string;
8  begin
9      a := 43;
10     integer := 'new string';
11     integer := 32;
12
13     if a > 10 then
14         b := a * 123
15     else
16         d := 'string';
17
18     while (a > 0) do
19         begin
20             a := a - 1;
21             c := c + a
22         end
23     end.
```

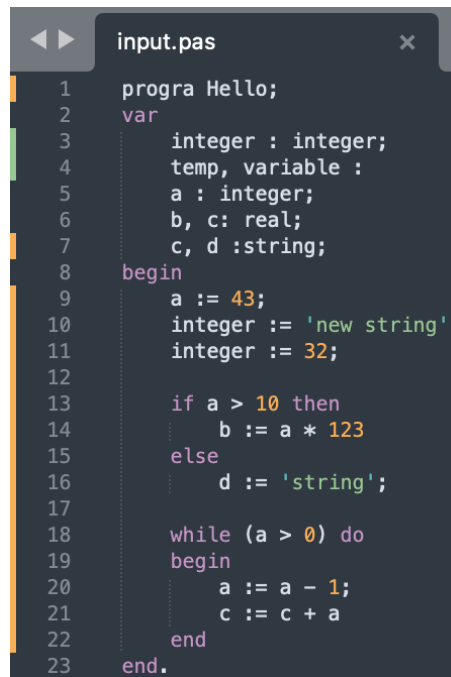
Содержимое выходного файла:

```
output.txt
1      01  program Hello;
2      02  var
3      03      integer : integer;
4      **01**      ^ Код ошибки: 19
5      ***** Название переменной не может совпадать с названием её типа
6      04      temp, variable :
7      05      a : integer;
8      **02**      ^ Код ошибки: 20
9      ***** Недопустимый тип
10     **03**      ^ Код ошибки: 14
11     ***** Ожидался оператор ;
12     06      b, c: real;
13     07      c, d :string;
14     **04**      ^ Код ошибки: 21
15     ***** Повторное описание переменной
16     08      begin
17     09      a := 43;
18     **05**      ^ Код ошибки: 22
19     ***** Неописанная переменная
20     10      integer := 'new string';
21     11      integer := 32;
22     12
23     13      if a > 10 then
24     14      b := a * 123
25     15      else
26     16      d := 'string';
27     17
28     18      while (a > 0) do
29     19      begin
30     20      a := a - 1;
31     21      c := c + a
32     22      end
33     23  end.
34
35  Всего ошибок - 5
```

Как видно из выходного файла, все синтаксические и семантически ошибки успешно обнаружены и нейтрализованы. А именно на 3 строке переменная `integer` будет иметь неизвестный тип, т.к. при её описании использовался недопустимый тип. Далее, в 4-5 строках опять возникнет ошибка недопустимого типа, т.к. `a` не является допустимым типом. Помимо этого, в 5 строке возникнет синтаксическая ошибка и при её нейтрализации будут пропущены все токены до токена «;» в конце строки. В 7 строке переменная `c` будет описана во второй раз с типом, отличным от первого описания, поэтому её тип переменной поменяется на неизвестный. В 9 строке возникнет ошибка неописанной переменной. Это связано с тем, что описание переменной `a` было интерпретировано как тип переменных с прошлой строки. Таким образом, переменная так и не была описана. После встречи неописанной переменной, она была добавлена в таблицу идентификаторов с неизвестным типом. Также в 10 и 11 строках видно, что переменная `integer` имеет неизвестный тип и может принимать любое значение.

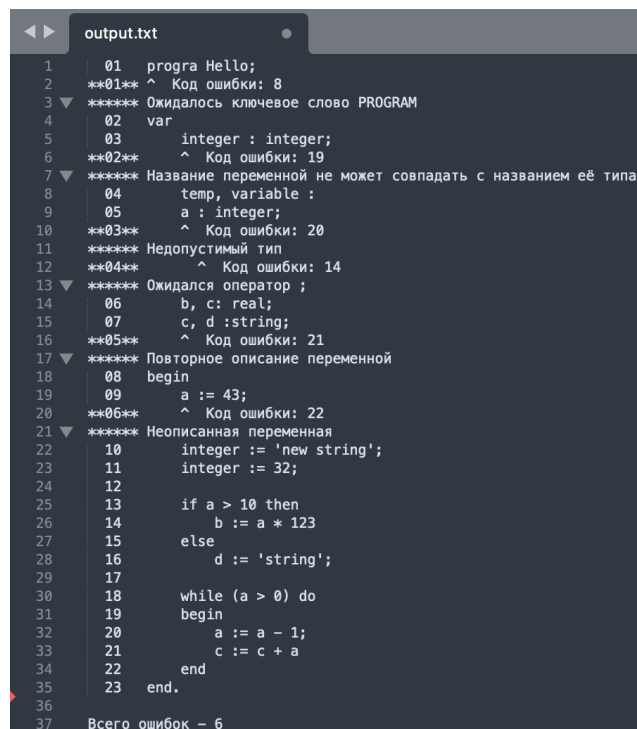


3) Теперь вернёмся к первой строке программы и допустим опечатку в ключевом слове **Program**:



```
1  progra Hello;
2  var
3      integer : integer;
4      temp, variable :
5      a : integer;
6      b, c: real;
7      c, d :string;
8  begin
9      a := 43;
10     integer := 'new string';
11     integer := 32;
12
13     if a > 10 then
14         b := a * 123
15     else
16         d := 'string';
17
18     while (a > 0) do
19     begin
20         a := a - 1;
21         c := c + a
22     end
23 end.
```

Содержимое выходного файла:



```
1  01  progra Hello;
2  **01** ^ Код ошибки: 8
3  ***** Ожидалось ключевое слово PROGRAM
4  02  var
5  03      integer : integer;
6  **02** ^ Код ошибки: 19
7  ***** Название переменной не может совпадать с названием её типа
8  04      temp, variable :
9  05      a : integer;
10 **03** ^ Код ошибки: 20
11 ***** Недопустимый тип
12 **04** ^ Код ошибки: 14
13 ***** Ожидался оператор ;
14 06      b, c: real;
15 07      c, d :string;
16 **05** ^ Код ошибки: 21
17 ***** Повторное описание переменной
18 08  begin
19 09      a := 43;
20 **06** ^ Код ошибки: 22
21 ***** Неописанная переменная
22 10      integer := 'new string';
23 11      integer := 32;
24 12
25 13      if a > 10 then
26 14          b := a * 123
27 15      else
28 16          d := 'string';
29 17
30 18      while (a > 0) do
31 19      begin
32 20          a := a - 1;
33 21          c := c + a
34 22      end
35 23  end.
36
37  Всего ошибок - 6
```

Можно заметить, что при нейтрализации синтаксической ошибки в первой строке токены были пропущены только до начала раздела переменных. Таким образом, было обнаружено максимальное количество ошибок.

#### 4) Теперь проверим приводимость типов:

```
input.pas
1  program Hello;
2  var
3      a : integer;
4      b : real;
5      c : string;
6      d : wrongType;
7  begin
8      a := 12;
9      a := 54.3;
10     a := 'string';
11
12     b := 12;
13     b := 28.3 + a;
14     b := 'string';
15
16     c := 12;
17     c := 30 / 9;
18     c := 'another string';
19
20     d := a;
21     d := b;
22     d := c;
23 end.
```

Содержимое выходного файла:

```
output.txt
1  01 program Hello;
2  02 var
3  03   a : integer;
4  04   b : real;
5  05   c : string;
6  06   d : wrongType;
7  **01**      ^ Код ошибки: 20
8  ***** Недопустимый тип
9  07 begin
10 08   a := 12;
11 09   a := 54.3;
12 **02**      ^ Код ошибки: 26
13 ***** Ожидалось выражение целочисленного типа
14 10   a := 'string';
15 **03**      ^ Код ошибки: 26
16 ***** Ожидалось выражение целочисленного типа
17 11
18 12   b := 12;
19 13   b := 28.3 + a;
20 14   b := 'string';
21 **04**      ^ Код ошибки: 27
22 ***** Ожидалось выражение, тип которого приводим к вещественному
23 15
24 16   c := 12;
25 **05**      ^ Код ошибки: 28
26 ***** Ожидалось выражение строкового типа
27 17   c := 30 / 9;
28 **06**      ^ Код ошибки: 28
29 ***** Ожидалось выражение строкового типа
30 18   c := 'another string';
31 19
32 20   d := a;
33 21   d := b;
34 22   d := c;
35 23 end.
36
37 Всего ошибок - 6
```

Как видно из сообщений об ошибках, переменной целочисленного типа можно присвоить только целочисленное значение, вещественной переменной — целочисленное или вещественное значение, а строковой

переменной – только строку. Переменная `d` имеет неизвестный тип, поэтому она может принимать любые значения.

5) Теперь протестируем корректность операций, применяемых к разным типам данных:

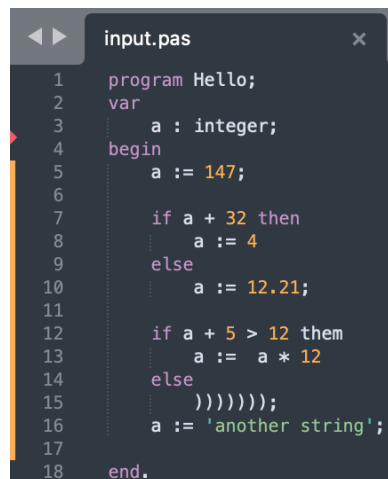
```
input.pas
1  program Hello;
2  var
3      a : integer;
4      b : real;
5      c : string;
6      d : wrongType;
7  begin
8      a := 12 + 12;
9      a := (81 OR a) AND (81 mod a);
10     a := a / 81;
11
12     b := 21 * 32.4;
13     b := b DIV a;
14     b := b AND 100;
15
16     c := 'first string' + 'second string';
17     c := c - 'first string';
18     c := (c + 'another string') * c;
19
20     d := (5 OR 12) MOD (142 DIV 12);
21     d := b > a;
22     d := a * b * c;
23 end.
```

Содержимое выходного файла:

```
output.txt
1  01 program Hello;
2  02 var
3  03   a : integer;
4  04   b : real;
5  05   c : string;
6  06   d : wrongType;
7  **01**      ^ Код ошибки: 20
8  ***** Недопустимый тип
9  07 begin
10 08   a := 12 + 12;
11 09   a := (81 OR a) AND (81 mod a);
12 10   a := a / 81;
13 **02**      ^ Код ошибки: 26
14 ***** Ожидалось выражение целочисленного типа
15 11
16 12   b := 21 * 32.4;
17 13   b := b DIV a;
18 **03**      ^ Код ошибки: 25
19 ***** Недопустимая операция для данного типа выражения
20 14   b := b AND 100;
21 **04**      ^ Код ошибки: 25
22 ***** Недопустимая операция для данного типа выражения
23 15
24 16   c := 'first string' + 'second string';
25 17   c := c - 'first string';
26 **05**      ^ Код ошибки: 25
27 ***** Недопустимая операция для данного типа выражения
28 18   c := (c + 'another string') * c;
29 **06**      ^ Код ошибки: 25
30 ***** Недопустимая операция для данного типа выражения
31 19
32 20   d := (5 OR 12) MOD (142 DIV 12);
33 21   d := b > a;
34 22   d := a * b * c;
35 **07**      ^ Код ошибки: 24
36 ***** Неприводимые типы в выражении
37 23 end.
38
39 Всего ошибок – 7
```

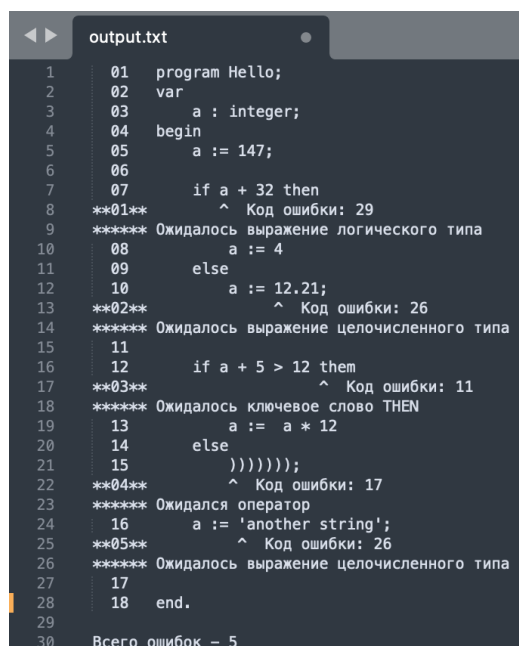
Как можно заметить из содержания файла, допустимость использования операций в зависимости от типов её операндов определяется корректно. К целочисленным операндам применимы любые операции, реализованные в заданном подмножестве языка. Для вещественных операндов применимы операции сложения, вычитания, умножения, а также вещественного деления. Для строк допустима только операция сложения. Логические операции применимы для любых типов, если операнды приводимы. Переменная неизвестного типа поддерживает все операции.

- б) Теперь проверим нейтрализацию синтаксических и семантических ошибок в условном операторе:



```
1  program Hello;
2  var
3      a : integer;
4  begin
5      a := 147;
6
7      if a + 32 then
8          a := 4
9      else
10         a := 12.21;
11
12         if a + 5 > 12 then
13             a := a * 12
14         else
15             ))))));
16         a := 'another string';
17
18     end.
```


Содержимое выходного файла:



```
1  01  program Hello;
2  02  var
3  03      a : integer;
4  04  begin
5  05      a := 147;
6  06
7  07      if a + 32 then
8  **01**      ^ Код ошибки: 29
9  ***** Ожидалось выражение логического типа
10 08          a := 4
1109      else
1210          a := 12.21;
13**02**      ^ Код ошибки: 26
14***** Ожидалось выражение целочисленного типа
1511
1612          if a + 5 > 12 then
17**03**      ^ Код ошибки: 11
18***** Ожидалось ключевое слово THEN
1913              a := a * 12
2014          else
2115              ))))));
22**04**      ^ Код ошибки: 17
23***** Ожидался оператор
2416          a := 'another string';
25**05**      ^ Код ошибки: 26
26***** Ожидалось выражение целочисленного типа
2717
2818      end.
29
30 Всего ошибок - 5
```

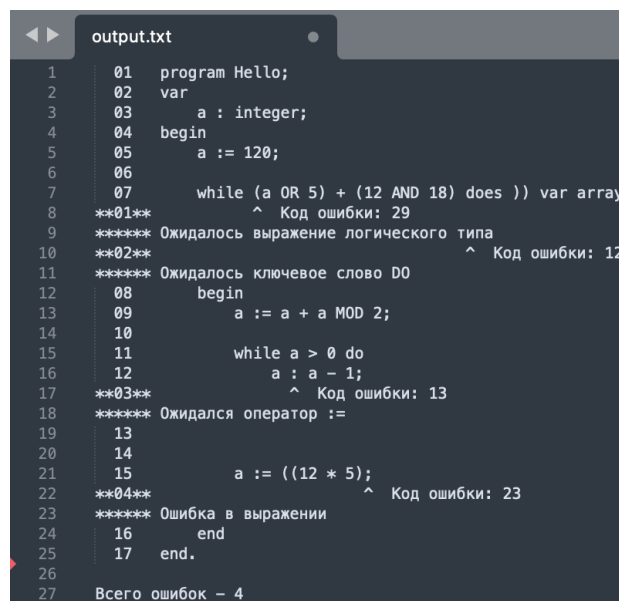
Как видно из сообщений об ошибках, синтаксические и семантические ошибки в условном операторе успешно находятся и нейтрализуются. В 7 строке была обнаружена семантическая ошибка, при её обнаружении нет необходимости пропускать какие-либо токены, так что компилятор сообщил об ошибке типа и продолжил работу с того же токена. В 12 же строка произошла синтаксическая ошибка и при нейтрализации токены будут пропускать до тех пор, пока не встретится токен, соответствующий началу оператора. В этом случае - идентификатор для оператора присваивания. В 15 строке опять произошла синтаксическая ошибка, которая будет обработана таким же образом, как и предыдущая.

7) Далее протестируем корректность работы оператора цикла:



```
1  program Hello;
2  var
3      a : integer;
4  begin
5      a := 120;
6
7      while (a OR 5) + (12 AND 18) does )) var array
8      begin
9          a := a + a MOD 2;
10
11          while a > 0 do
12              a : a - 1;
13
14
15          a := ((12 * 5);
16      end
17  end.
```

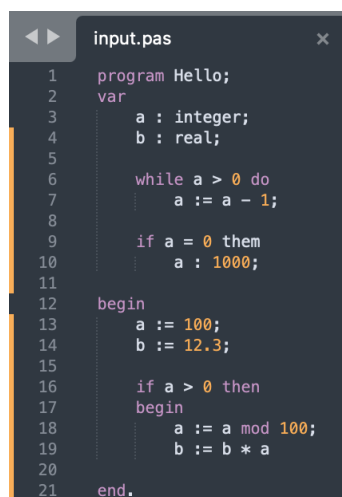
Содержимое выходного файла:



```
1  01  program Hello;
2  02  var
3  03      a : integer;
4  04  begin
5  05      a := 120;
6  06
7  07      while (a OR 5) + (12 AND 18) does )) var array
8  **01**      ^ Код ошибки: 29
9  ***** Ожидалось выражение логического типа
10 **02**      ^ Код ошибки: 12
11 ***** Ожидалось ключевое слово DO
12 08      begin
13 09          a := a + a MOD 2;
14 10
15 11          while a > 0 do
16 12              a : a - 1;
17 **03**      ^ Код ошибки: 13
18 ***** Ожидался оператор :=
19 13
20 14
21 15          a := ((12 * 5);
22 **04**      ^ Код ошибки: 23
23 ***** Ошибка в выражении
24 16      end
25 17  end.
26
27  Всего ошибок - 4
```

Как видно из сообщений об ошибках, все ошибки опять были корректно найдены и нейтрализованы. В 7 строке были найдены семантическая ошибка неправильного типа выражения, а также синтаксическая ошибка, в ходе нейтрализации которой были пропущены все токены до токена, с которого может начинаться оператор. В 12 строке была обнаружена ещё одна синтаксическая ошибка в операторе присваивания. В этом случае токены должны были пропускать либо до точки с запятой, либо до ключевого слова **end**. В 15 строке была найдена последняя ошибка, заключающаяся в неправильной расстановке скобок в выражении.

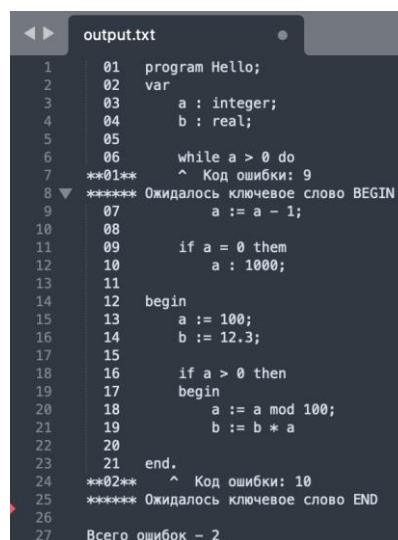
- 8) Напоследок проверим корректность нейтрализации синтаксических ошибок в начале раздела операторов, а также в составном операторе:



```

input.pas
1  program Hello;
2  var
3      a : integer;
4      b : real;
5
6      while a > 0 do
7          a := a - 1;
8
9      if a = 0 then
10         a : 1000;
11
12     begin
13         a := 100;
14         b := 12.3;
15
16         if a > 0 then
17             begin
18                 a := a mod 100;
19                 b := b * a
20
21     end.
  
```

Содержимое выходного файла:



```

output.txt
1  01 program Hello;
2  02 var
3  03     a : integer;
4  04     b : real;
5  05
6  06     while a > 0 do
7  07         a := a - 1;
8  08
9  09     if a = 0 then
10  10         a : 1000;
11  11
12  12     begin
13  13         a := 100;
14  14         b := 12.3;
15  15
16  16         if a > 0 then
17  17             begin
18  18                 a := a mod 100;
19  19                 b := b * a
20  20
21  21     end.
22
23
24  01 program Hello;
25  02 var
26  03     a : integer;
27  04     b : real;
28  05
29  06     while a > 0 do
30  07         a := a - 1;
31  08
32  09     if a = 0 then
33  10         a : 1000;
34  11
35  12     begin
36  13         a := 100;
37  14         b := 12.3;
38  15
39  16         if a > 0 then
40  17             begin
41  18                 a := a mod 100;
42  19                 b := b * a
43  20
44  21     end.
45
46
47  01 program Hello;
48  02 var
49  03     a : integer;
50  04     b : real;
51  05
52  06     while a > 0 do
53  07         a := a - 1;
54  08
55  09     if a = 0 then
56  10         a : 1000;
57  11
58  12     begin
59  13         a := 100;
60  14         b := 12.3;
61  15
62  16         if a > 0 then
63  17             begin
64  18                 a := a mod 100;
65  19                 b := b * a
66  20
67  21     end.
68
69
70  01 program Hello;
71  02 var
72  03     a : integer;
73  04     b : real;
74  05
75  06     while a > 0 do
76  07         a := a - 1;
77  08
78  09     if a = 0 then
79  10         a : 1000;
80  11
81  12     begin
82  13         a := 100;
83  14         b := 12.3;
84  15
85  16         if a > 0 then
86  17             begin
87  18                 a := a mod 100;
88  19                 b := b * a
89  20
90  21     end.
91
92
93  01 program Hello;
94  02 var
95  03     a : integer;
96  04     b : real;
97  05
98  06     while a > 0 do
99  07         a := a - 1;
100 08
101 09     if a = 0 then
102 10         a : 1000;
103 11
104 12     begin
105 13         a := 100;
106 14         b := 12.3;
107 15
108 16         if a > 0 then
109 17             begin
110 18                 a := a mod 100;
111 19                 b := b * a
112 20
113 21     end.
114
115
116 01 program Hello;
117 02 var
118 03     a : integer;
119 04     b : real;
120 05
121 06     while a > 0 do
122 07         a := a - 1;
123 08
124 09     if a = 0 then
125 10         a : 1000;
126 11
127 12     begin
128 13         a := 100;
129 14         b := 12.3;
130 15
131 16         if a > 0 then
132 17             begin
133 18                 a := a mod 100;
134 19                 b := b * a
135 20
136 21     end.
137
138
139 01 program Hello;
140 02 var
141 03     a : integer;
142 04     b : real;
143 05
144 06     while a > 0 do
145 07         a := a - 1;
146 08
147 09     if a = 0 then
148 10         a : 1000;
149 11
150 12     begin
151 13         a := 100;
152 14         b := 12.3;
153 15
154 16         if a > 0 then
155 17             begin
156 18                 a := a mod 100;
157 19                 b := b * a
158 20
159 21     end.
160
161
162 01 program Hello;
163 02 var
164 03     a : integer;
165 04     b : real;
166 05
167 06     while a > 0 do
168 07         a := a - 1;
169 08
170 09     if a = 0 then
171 10         a : 1000;
172 11
173 12     begin
174 13         a := 100;
175 14         b := 12.3;
176 15
177 16         if a > 0 then
178 17             begin
179 18                 a := a mod 100;
180 19                 b := b * a
181 20
182 21     end.
183
184
185 01 program Hello;
186 02 var
187 03     a : integer;
188 04     b : real;
189 05
190 06     while a > 0 do
191 07         a := a - 1;
192 08
193 09     if a = 0 then
194 10         a : 1000;
195 11
196 12     begin
197 13         a := 100;
198 14         b := 12.3;
199 15
200 16         if a > 0 then
201 17             begin
202 18                 a := a mod 100;
203 19                 b := b * a
204 20
205 21     end.
206
207
208 01 program Hello;
209 02 var
210 03     a : integer;
211 04     b : real;
212 05
213 06     while a > 0 do
214 07         a := a - 1;
215 08
216 09     if a = 0 then
217 10         a : 1000;
218 11
219 12     begin
220 13         a := 100;
221 14         b := 12.3;
222 15
223 16         if a > 0 then
224 17             begin
225 18                 a := a mod 100;
226 19                 b := b * a
227 20
228 21     end.
229
230
231 01 program Hello;
232 02 var
233 03     a : integer;
234 04     b : real;
235 05
236 06     while a > 0 do
237 07         a := a - 1;
238 08
239 09     if a = 0 then
240 10         a : 1000;
241 11
242 12     begin
243 13         a := 100;
244 14         b := 12.3;
245 15
246 16         if a > 0 then
247 17             begin
248 18                 a := a mod 100;
249 19                 b := b * a
250 20
251 21     end.
252
253
254 01 program Hello;
255 02 var
256 03     a : integer;
257 04     b : real;
258 05
259 06     while a > 0 do
260 07         a := a - 1;
261 08
262 09     if a = 0 then
263 10         a : 1000;
264 11
265 12     begin
266 13         a := 100;
267 14         b := 12.3;
268 15
269 16         if a > 0 then
270 17             begin
271 18                 a := a mod 100;
272 19                 b := b * a
273 20
274 21     end.
275
276
277 01 program Hello;
278 02 var
279 03     a : integer;
280 04     b : real;
281 05
282 06     while a > 0 do
283 07         a := a - 1;
284 08
285 09     if a = 0 then
286 10         a : 1000;
287 11
288 12     begin
289 13         a := 100;
290 14         b := 12.3;
291 15
292 16         if a > 0 then
293 17             begin
294 18                 a := a mod 100;
295 19                 b := b * a
296 20
297 21     end.
298
299
300 01 program Hello;
301 02 var
302 03     a : integer;
303 04     b : real;
304 05
305 06     while a > 0 do
306 07         a := a - 1;
307 08
308 09     if a = 0 then
309 10         a : 1000;
310 11
311 12     begin
312 13         a := 100;
313 14         b := 12.3;
314 15
315 16         if a > 0 then
316 17             begin
317 18                 a := a mod 100;
318 19                 b := b * a
319 20
320 21     end.
321
322
323 01 program Hello;
324 02 var
325 03     a : integer;
326 04     b : real;
327 05
328 06     while a > 0 do
329 07         a := a - 1;
330 08
331 09     if a = 0 then
332 10         a : 1000;
333 11
334 12     begin
335 13         a := 100;
336 14         b := 12.3;
337 15
338 16         if a > 0 then
339 17             begin
340 18                 a := a mod 100;
341 19                 b := b * a
342 20
343 21     end.
344
345
346 01 program Hello;
347 02 var
348 03     a : integer;
349 04     b : real;
350 05
351 06     while a > 0 do
352 07         a := a - 1;
353 08
354 09     if a = 0 then
355 10         a : 1000;
356 11
357 12     begin
358 13         a := 100;
359 14         b := 12.3;
360 15
361 16         if a > 0 then
362 17             begin
363 18                 a := a mod 100;
364 19                 b := b * a
365 20
366 21     end.
367
368
369 01 program Hello;
370 02 var
371 03     a : integer;
372 04     b : real;
373 05
374 06     while a > 0 do
375 07         a := a - 1;
376 08
377 09     if a = 0 then
378 10         a : 1000;
379 11
380 12     begin
381 13         a := 100;
382 14         b := 12.3;
383 15
384 16         if a > 0 then
385 17             begin
386 18                 a := a mod 100;
387 19                 b := b * a
388 20
389 21     end.
390
391
392 01 program Hello;
393 02 var
394 03     a : integer;
395 04     b : real;
396 05
397 06     while a > 0 do
398 07         a := a - 1;
399 08
400 09     if a = 0 then
401 10         a : 1000;
402 11
403 12     begin
404 13         a := 100;
405 14         b := 12.3;
406 15
407 16         if a > 0 then
408 17             begin
409 18                 a := a mod 100;
410 19                 b := b * a
411 20
412 21     end.
413
414
415 01 program Hello;
416 02 var
417 03     a : integer;
418 04     b : real;
419 05
420 06     while a > 0 do
421 07         a := a - 1;
422 08
423 09     if a = 0 then
424 10         a : 1000;
425 11
426 12     begin
427 13         a := 100;
428 14         b := 12.3;
429 15
430 16         if a > 0 then
431 17             begin
432 18                 a := a mod 100;
433 19                 b := b * a
434 20
435 21     end.
436
437
438 01 program Hello;
439 02 var
440 03     a : integer;
441 04     b : real;
442 05
443 06     while a > 0 do
444 07         a := a - 1;
445 08
446 09     if a = 0 then
447 10         a : 1000;
448 11
449 12     begin
450 13         a := 100;
451 14         b := 12.3;
452 15
453 16         if a > 0 then
454 17             begin
455 18                 a := a mod 100;
456 19                 b := b * a
457 20
458 21     end.
459
460
461 01 program Hello;
462 02 var
463 03     a : integer;
464 04     b : real;
465 05
466 06     while a > 0 do
467 07         a := a - 1;
468 08
469 09     if a = 0 then
470 10         a : 1000;
471 11
472 12     begin
473 13         a := 100;
474 14         b := 12.3;
475 15
476 16         if a > 0 then
477 17             begin
478 18                 a := a mod 100;
479 19                 b := b * a
480 20
481 21     end.
482
483
484 01 program Hello;
485 02 var
486 03     a : integer;
487 04     b : real;
488 05
489 06     while a > 0 do
490 07         a := a - 1;
491 08
492 09     if a = 0 then
493 10         a : 1000;
494 11
495 12     begin
496 13         a := 100;
497 14         b := 12.3;
498 15
499 16         if a > 0 then
500 17             begin
501 18                 a := a mod 100;
502 19                 b := b * a
503 20
504 21     end.
505
506
507 01 program Hello;
508 02 var
509 03     a : integer;
510 04     b : real;
511 05
512 06     while a > 0 do
513 07         a := a - 1;
514 08
515 09     if a = 0 then
516 10         a : 1000;
517 11
518 12     begin
519 13         a := 100;
520 14         b := 12.3;
521 15
522 16         if a > 0 then
523 17             begin
524 18                 a := a mod 100;
525 19                 b := b * a
526 20
527 21     end.
528
529
530 01 program Hello;
531 02 var
532 03     a : integer;
533 04     b : real;
534 05
535 06     while a > 0 do
536 07         a := a - 1;
537 08
538 09     if a = 0 then
539 10         a : 1000;
540 11
541 12     begin
542 13         a := 100;
543 14         b := 12.3;
544 15
545 16         if a > 0 then
546 17             begin
547 18                 a := a mod 100;
548 19                 b := b * a
549 20
550 21     end.
551
552
553 01 program Hello;
554 02 var
555 03     a : integer;
556 04     b : real;
557 05
558 06     while a > 0 do
559 07         a := a - 1;
560 08
561 09     if a = 0 then
562 10         a : 1000;
563 11
564 12     begin
565 13         a := 100;
566 14         b := 12.3;
567 15
568 16         if a > 0 then
569 17             begin
570 18                 a := a mod 100;
571 19                 b := b * a
572 20
573 21     end.
574
575
576 01 program Hello;
577 02 var
578 03     a : integer;
579 04     b : real;
580 05
581 06     while a > 0 do
582 07         a := a - 1;
583 08
584 09     if a = 0 then
585 10         a : 1000;
586 11
587 12     begin
588 13         a := 100;
589 14         b := 12.3;
590 15
591 16         if a > 0 then
592 17             begin
593 18                 a := a mod 100;
594 19                 b := b * a
595 20
596 21     end.
597
598
599 01 program Hello;
600 02 var
601 03     a : integer;
602 04     b : real;
603 05
604 06     while a > 0 do
605 07         a := a - 1;
606 08
607 09     if a = 0 then
608 10         a : 1000;
609 11
610 12     begin
611 13         a := 100;
612 14         b := 12.3;
613 15
614 16         if a > 0 then
615 17             begin
616 18                 a := a mod 100;
617 19                 b := b * a
618 20
619 21     end.
620
621
622 01 program Hello;
623 02 var
624 03     a : integer;
625 04     b : real;
626 05
627 06     while a > 0 do
628 07         a := a - 1;
629 08
630 09     if a = 0 then
631 10         a : 1000;
632 11
633 12     begin
634 13         a := 100;
635 14         b := 12.3;
636 15
637 16         if a > 0 then
638 17             begin
639 18                 a := a mod 100;
640 19                 b := b * a
641 20
642 21     end.
643
644
645 01 program Hello;
646 02 var
647 03     a : integer;
648 04     b : real;
649 05
650 06     while a > 0 do
651 07         a := a - 1;
652 08
653 09     if a = 0 then
654 10         a : 1000;
655 11
656 12     begin
657 13         a := 100;
658 14         b := 12.3;
659 15
660 16         if a > 0 then
661 17             begin
662 18                 a := a mod 100;
663 19                 b := b * a
664 20
665 21     end.
666
667
668 01 program Hello;
669 02 var
670 03     a : integer;
671 04     b : real;
672 05
673 06     while a > 0 do
674 07         a := a - 1;
675 08
676 09     if a = 0 then
677 10         a : 1000;
678 11
679 12     begin
680 13         a := 100;
681 14         b := 12.3;
682 15
683 16         if a > 0 then
684 17             begin
685 18                 a := a mod 100;
686 19                 b := b * a
687 20
688 21     end.
689
690
691 01 program Hello;
692 02 var
693 03     a : integer;
694 04     b : real;
695 05
696 06     while a > 0 do
697 07         a := a - 1;
698 08
699 09     if a = 0 then
700 10         a : 1000;
701 11
702 12     begin
703 13         a := 100;
704 14         b := 12.3;
705 15
706 16         if a > 0 then
707 17             begin
708 18                 a := a mod 100;
709 19                 b := b * a
710 20
711 21     end.
712
713
714 01 program Hello;
715 02 var
716 03     a : integer;
717 04     b : real;
718 05
719 06     while a > 0 do
720 07         a := a - 1;
721 08
722 09     if a = 0 then
723 10         a : 1000;
724 11
725 12     begin
726 13         a := 100;
727 14         b := 12.3;
728 15
729 16         if a > 0 then
730 17             begin
731 18                 a := a mod 100;
732 19                 b := b * a
733 20
734 21     end.
735
736
737 01 program Hello;
738 02 var
739 03     a : integer;
740 04     b : real;
741 05
742 06     while a > 0 do
743 07         a := a - 1;
744 08
745 09     if a = 0 then
746 10         a : 1000;
747 11
748 12     begin
749 13         a := 100;
750 14         b := 12.3;
751 15
752 16         if a > 0 then
753 17             begin
754 18                 a := a mod 100;
755 19                 b := b * a
756 20
757 21     end.
758
759
760 01 program Hello;
761 02 var
762 03     a : integer;
763 04     b : real;
764 05
765 06     while a > 0 do
766 07         a := a - 1;
767 08
768 09     if a = 0 then
769 10         a : 1000;
770 11
771 12     begin
772 13         a := 100;
773 14         b := 12.3;
774 15
775 16         if a > 0 then
776 17             begin
777 18                 a := a mod 100;
778 19                 b := b * a
779 20
780 21     end.
781
782
783 01 program Hello;
784 02 var
785 03     a : integer;
786 04     b : real;
787 05
788 06     while a > 0 do
789 07         a := a - 1;
790 08
791 09     if a = 0 then
792 10         a : 1000;
793 11
794 12     begin
795 13         a := 100;
796 14         b := 12.3;
797 15
798 16         if a > 0 then
799 17             begin
800 18                 a := a mod 100;
801 19                 b := b * a
802 20
803 21     end.
804
805
806 01 program Hello;
807 02 var
808 03     a : integer;
809 04     b : real;
810 05
811 06     while a > 0 do
812 07         a := a - 1;
813 08
814 09     if a = 0 then
815 10         a : 1000;
816 11
817 12     begin
818 13         a := 100;
819 14         b := 12.3;
820 15
821 16         if a > 0 then
822 17             begin
823 18                 a := a mod 100;
824 19                 b := b * a
825 20
826 21     end.
827
828
829 01 program Hello;
830 02 var
831 03     a : integer;
832 04     b : real;
833 05
834 06     while a > 0 do
835 07         a := a - 1;
836 08
837 09     if a = 0 then
838 10         a : 1000;
839 11
840 12     begin
841 13         a := 100;
842 14         b := 12.3;
843 15
844 16         if a > 0 then
845 17             begin
846 18                 a := a mod 100;
847 19                 b := b * a
848 20
849 21     end.
850
851
852 01 program Hello;
853 02 var
854 03     a : integer;
855 04     b : real;
856 05
857 06     while a > 0 do
858 07         a := a - 1;
859 08
860 09     if a = 0 then
861 10         a : 1000;
862 11
863 12     begin
864 13         a := 100;
865 14         b := 12.3;
866 15
867 16         if a > 0 then
868 17             begin
869 18                 a := a mod 100;
870 19                 b := b * a
871 20
872 21     end.
873
874
875 01 program Hello;
876 02 var
877 03     a : integer;
878 04     b : real;
879 05
880 06     while a > 0 do
881 07         a := a - 1;
882 08
883 09     if a = 0 then
884 10         a : 1000;
885 11
886 12     begin
887 13         a := 100;
888 14         b := 12.3;
889 15
890 16         if a > 0 then
891 17             begin
892 18                 a := a mod 100;
893 19                 b := b * a
894 20
895 21     end.
896
897
898 01 program Hello;
899 02 var
900 03     a : integer;
901 04     b : real;
902 05
903 06     while a > 0 do
904 07         a := a - 1;
905 08
906 09     if a = 0 then
907 10         a : 1000;
908 11
909 12     begin
910 13         a := 100;
911 14         b := 12.3;
912 15
913 16         if a > 0 then
914 17             begin
915 18                 a := a mod 100;
916 19                 b := b * a
917 20
918 21     end.
919
920
921 01 program Hello;
922 02 var
923 03     a : integer;
924 04     b : real;
925 05
926 06     while a > 0 do
927 07         a := a - 1;
928 08
929 09     if a = 0 then
930 10         a : 1000;
931 11
932 12     begin
933 13         a := 100;
934 14         b := 12.3;
935 15
936 16         if a > 0 then
937 17             begin
938 18                 a := a mod 100;
939 19                 b := b * a
940 20
941 21     end.
942
943
944 01 program Hello;
945 02 var
946 03     a : integer;
947 04     b : real;
948 05
949 06     while a > 0 do
950 07         a := a - 1;
951 08
952 09     if a = 0 then
953 10         a : 1000;
954 11
955 12     begin
956 13         a := 100;
957 14         b := 12.3;
958 15
959 16         if a > 0 then
960 17             begin
961 18                 a := a mod 100;
962 19                 b := b * a
963 20
964 21     end.
965
966
967 01 program Hello;
968 02 var
969 03     a : integer;
970 04     b : real;
971 05
972 06     while a > 0 do
973 07         a := a - 1;
974 08
975 09     if a = 0 then
976 10         a : 1000;
977 11
978 12     begin
979 13         a := 100;
980 14         b := 12.3;
981 15
982 16         if a > 0 then
983 17             begin
984 18                 a := a mod 100;
985 19                 b := b * a
986 20
987 21     end.
988
989
990 01 program Hello;
991 02 var
992 03     a : integer;
993 04     b : real;
994 05
995 06     while a > 0 do
996 07         a := a - 1;
997 08
998 09     if a = 0 then
999 10         a : 1000;
1000 11
1001 12     begin
1002 13         a := 100;
1003 14         b := 12.3;
1004 15
1005 16         if a > 0 then
1006 17             begin
1007 18                 a := a mod 100;
1008 19                 b := b * a
1009 20
1010 21     end.
1011
1012
1013 01 program Hello;
1014 02 var
1015 03     a : integer;
1016 04     b : real;
1017 05
1018 06     while a > 0 do
1019 07         a := a - 1;
1020 08
1021 09     if a = 0 then
1022 10         a : 1000;
1023 11
1024 12     begin
1025 13         a := 100;
1026 14         b := 12.3;
1027 15
1028 16         if a > 0 then
1029 17             begin
1030 18                 a := a mod 100;
1031 19                 b := b * a
1032 20
1033 21     end.
1034
1035
1036 01 program Hello;
1037 02 var
1038 03     a : integer;
1039 04     b : real;
1040 05
1041 06     while a > 0 do
1042 07         a := a - 1;
1043 08
1044 09     if a = 0 then
1045 10         a : 1000;
1046 11
1047 12     begin
1048 13         a := 100;
1049 14         b := 12.3;
1050 15
1051 16         if a > 0 then
1052 17             begin
1053 18                 a := a mod 100;
1054 19                 b := b * a
1055 20
1056 21     end.
1057
1058
1059 01 program Hello;
1060 02 var
1061 03     a : integer;
1062 04     b : real;
1063 05
1064 06     while a > 0 do
1065 07         a := a - 1;
1066 08
1067 09     if a = 0 then
1068 10         a : 1000;
1069 11
1070 12     begin
1071 13         a := 100;
1072 14         b := 12.3;
1073 1
```

Как видно из выходного файла, все ошибки опять были найдены и нейтрализованы корректно. Начиная с 6 строки, компилятор понял, что мы вышли из раздела переменных и зашли в раздел операторов. Этот раздел должен начинаться с ключевого слова **Begin**. Поэтому формируется советующая ошибка и в ходе нейтрализации пропускаются все токены, пока не будет найдено ключевое слово **Begin**, либо программа не закончится. На 21 строке обнаруживается ещё одна синтаксическая ошибка. Дело в том, что составной оператор в тела условного оператора не был закрыт, и ключевое слово **end**, советующее закрытию раздела операторов было интерпретировано как закрытие составного оператора в теле условного оператора. После которого уже ожидалось **end** для закрытия раздела операторов.