

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`);

Помимо этого, необходимо написать генератор кода, который будет создавать инструкции языка MSIL, соответствующие полученной на вход программе на подмножестве языка Pascal. После чего необходимо получить соответствующий исполняемый файл.

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
Генератор кода	40
1. Описание.....	40
2. Проектирование.....	40
3. Реализация	41
4. Тестирование	43

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 непрочитанного 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 непротоптанную 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.

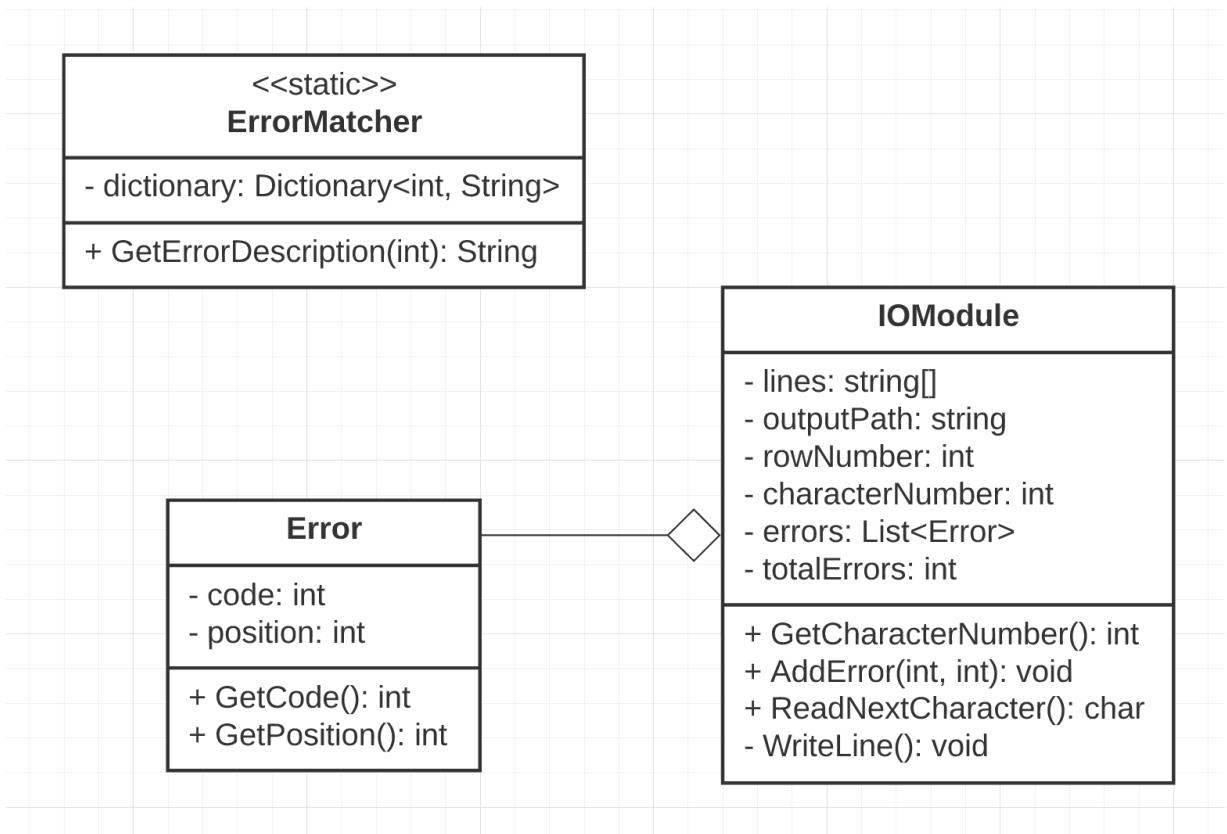


Рисунок 1 – диаграмма классов модуля ввода-вывода

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 пройтись 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 `GetErrorMessage` 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!».

The screenshot shows two terminal windows side-by-side. The left window is titled 'input.pas' and contains the following code:

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

The right window is titled 'output.txt' and contains the following output:

```
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 && lineNumber > 0)
    {
        for (int i = 0; i < lineNumber; i++)
            AddError(i % 2 == 1 ? 0 : 1, lineNumber);

        WriteLine();
    }

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

    lineNumber++;
    CharacterNumber = 0;
    return lineNumber <= 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 односторочный 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 буквы 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.

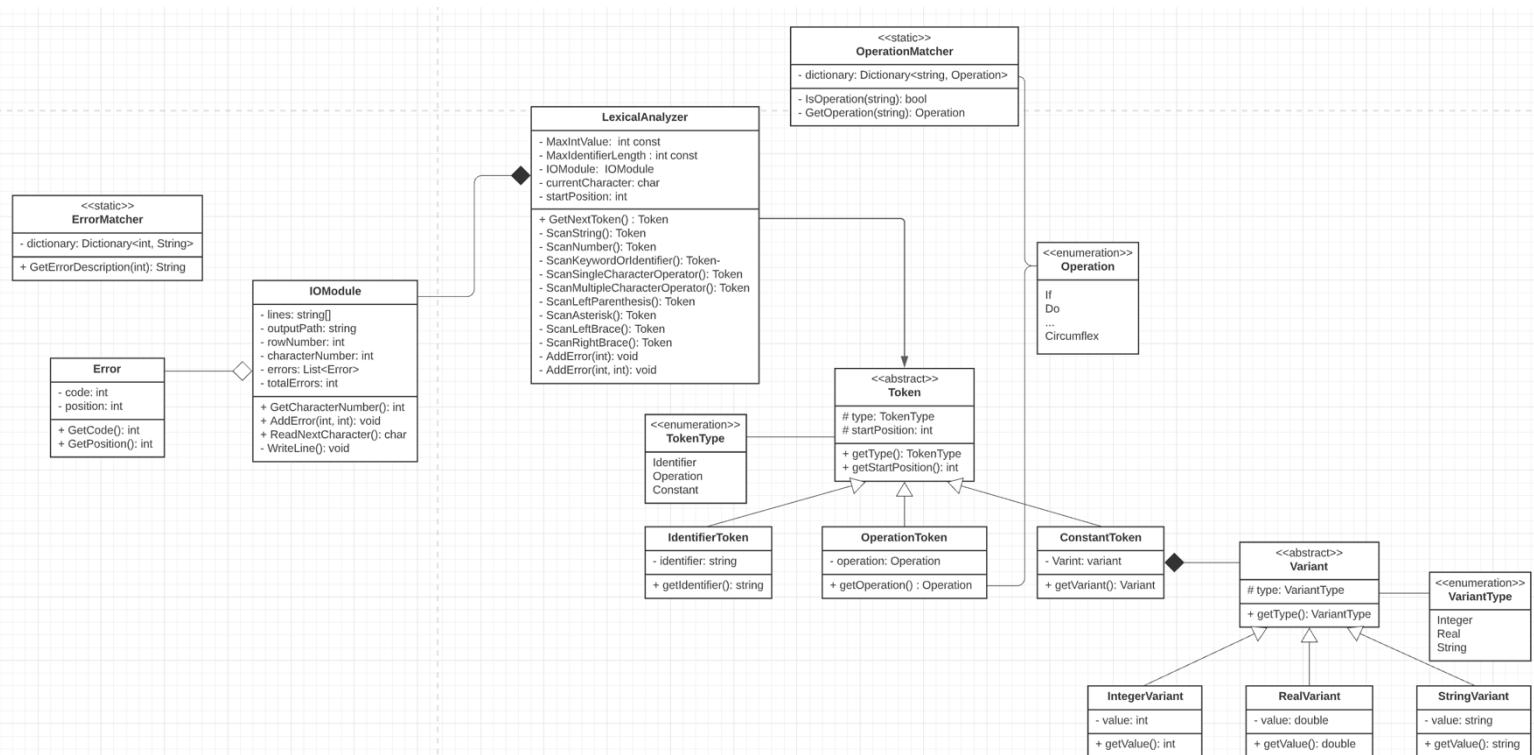


Рисунок 2 – текущая диаграмма классов компилятора

3. Реализация

Алгоритм построения токена

До тех пор, пока текущая литерра 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 литерра является терминальным нулём (`\0`), то возвращаем `null`.
- Если текущая литерра 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 литерра 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 литерра «.» 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 teste 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 подчёркивания. Когда условие перстаёт выполняется, 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 литерра 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 лексическим 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.

```
program Hello;
var
    a : integer;
    b, c :real;
    d :string;
begin
    a:=8;
    b := a / 3 + (a + 12) MOD 14;
    d := 'test string';
if ((a + 5) div 3 > 2) OR (b < 0) then
    b := 323232.325
else
    c:= 45.3;
while a > 0 do
    begin
        b := b * a;
        a := a - 1;
    end;
end.
```

Результат вывода в консоль:

Type	Value	Start position	Row
Type: Operation	Value: Program	Start position: 1	1 строка
Type: Identifier	Value: Hello	Start position: 9	2 строка
Type: Operation	Value: Semicolon	Start position: 14	3 строка
Type: Operation	Value: Var	Start position: 1	4 строка
Type: Identifier	Value: a	Start position: 5	5 строка
Type: Operation	Value: Colon	Start position: 7	6 строка
Type: Operation	Value: integer	Start position: 9	7 строка
Type: Identifier	Value: b	Start position: 5	8 строка
Type: Operation	Value: Comma	Start position: 6	9 строка
Type: Identifier	Value: c	Start position: 8	10 строка
Type: Operation	Value: Colon	Start position: 10	11 строка
Type: Identifier	Value: real	Start position: 11	12 строка
Type: Operation	Value: Semicolon	Start position: 15	13 строка
Type: Identifier	Value: d	Start position: 5	14 строка
Type: Operation	Value: Colon	Start position: 7	15 строка
Type: Identifier	Value: string	Start position: 8	16 строка
Type: Operation	Value: Semicolon	Start position: 14	17 строка
Type: Operation	Value: Begin	Start position: 1	18 строка
Type: Identifier	Value: a	Start position: 5	19 строка
Type: Operation	Value: Assignment	Start position: 6	20 строка
Type: Constant	Type: Integer Value: 8	Start position: 8	21 строка
Type: Operation	Value: Semicolon	Start position: 9	22 строка
Type: Identifier	Value: b	Start position: 5	23 строка
Type: Operation	Value: Assignment	Start position: 7	24 строка
Type: Identifier	Value: a	Start position: 10	25 строка
Type: Operation	Value: Slash	Start position: 12	26 строка
Type: Constant	Type: Integer Value: 3	Start position: 14	27 строка
Type: Operation	Value: Plus	Start position: 16	28 строка
Type: Operation	Value: LeftParenthesis	Start position: 18	29 строка
Type: Identifier	Value: a	Start position: 19	30 строка
Type: Operation	Value: Plus	Start position: 21	31 строка
Type: Constant	Type: Integer Value: 12	Start position: 23	32 строка
Type: Operation	Value: RightParenthesis	Start position: 25	33 строка
Type: Operation	Value: Mod	Start position: 27	34 строка
Type: Constant	Type: Integer Value: 14	Start position: 31	35 строка
Type: Operation	Value: Semicolon	Start position: 33	36 строка
Type: Identifier	Value: d	Start position: 5	37 строка
Type: Operation	Value: Assignment	Start position: 7	38 строка
Type: Constant	Type: String Value: test string	Start position: 10	39 строка
Type: Operation	Value: Semicolon	Start position: 23	40 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
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 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 односторонние 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.
```

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:

Содержание выходного файла:

```
▶ output.txt

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
0 06 begin
1 07     a:=8;
2 08     b := a / 3 + (a + 12) MOD 143232954545454433; {Значение целочисленной константы превышает предел!}
3 **03** ^ Код ошибки: 5
4 ***** Значение целочисленной константы превышает предел
5 09     d := 'test string
6 **04** ^ Код ошибки: 3
7 ***** Ошибка в описании строковой константы
8 10         and also error
9 11     '; {Ошибка в описании строковой константы!"}
0 12
1 13     if ((a + 5) div 3 > 2) OR (b < 0) then
2 14         b := 323232.325
3 15     else
4 16         c:= 45.3.18; {Ошибка в описании вещественной константы!}
5 **05** ^ Код ошибки: 4
6 ***** Ошибка в описании вещественной константы
7 17
8 18     while aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa > 0 do {Длина идентификатора превышает предел}
9 **06** ^ Код ошибки: 6
0 ***** Длина идентификатора превышает предел
1 19         begin
2 20             b := b * a;
3 21             a := a - 1;
4 22         end;
5 23     end.! {Запрещённый символ!}
6 **07** ^ Код ошибки: 7
7 ***** Запрещённый символ
8
9 Всего ошибок - 7
```

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:

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

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

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 методы **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 `UnknownType`. Во всех классах, за исключением последнего, в качестве единственного собственного поля содержится список с операциями, поддерживаемыми данным типом.

Класс `UnknownType` 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.

Также в классе содержатся методы, упрощающие работы с типами. Первый метод возвращает значение словаря по переданному ключу. Второй метод получает два типа и проверяет их на взаимную приводимость. Третий метод получает на вход два приводимых типа и приводит их к одному типу.

Для реализации синтаксического анализа в классе **Compiler** необходимо создать методы, соответствующие описанным БНФ. Так, для формы «Программа» будет создан метод **Program**. В этом методе в самом начале будет приниматься ключевое слово **Program**, далее будет приниматься идентификатор, соответствующий названию программы, далее будет вызван метод **Block**, после чего опять потребуется принять операцию – «.».

Для принятия операторов и идентификаторов используются методы **AcceptOperation** и **AcceptIdentifier**. Если значение текущего токена отличается от того, что эти методы ожидали получить, они должны бросить ошибку.

Для обработки синтаксических ошибок необходимо модифицировать методы, соответствующие БНФ. Они должны обрабатывать полученные ошибки и пропускать токены до тех пор, пока не дойдут до такого токена, с которого можно будет вернуться к анализу следующих БНФ.

Для этого необходимо реализовать метод **SkipTokensTo**, который будет получать список токенов и пропускать текущие до тех пор, пока не наткнётся на какой-то из списка.

Для удобного хранения этих списков создаётся абстрактный класс **NextTokens**. Этот класс содержит списки токенов, которые должны следовать после текущей формы.

Для реализации семантического анализа также необходимо модифицировать методы, соответствующие БНФ. Например, в методе описания переменных добавлять переменные в поле **scope**, также

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**

Диаграмма классов, включая классы модуля ввода-вывода и лексического анализатора представлена на рисунке 3 (синим выделены методы, реализующие формы Бэкуса-Наура).

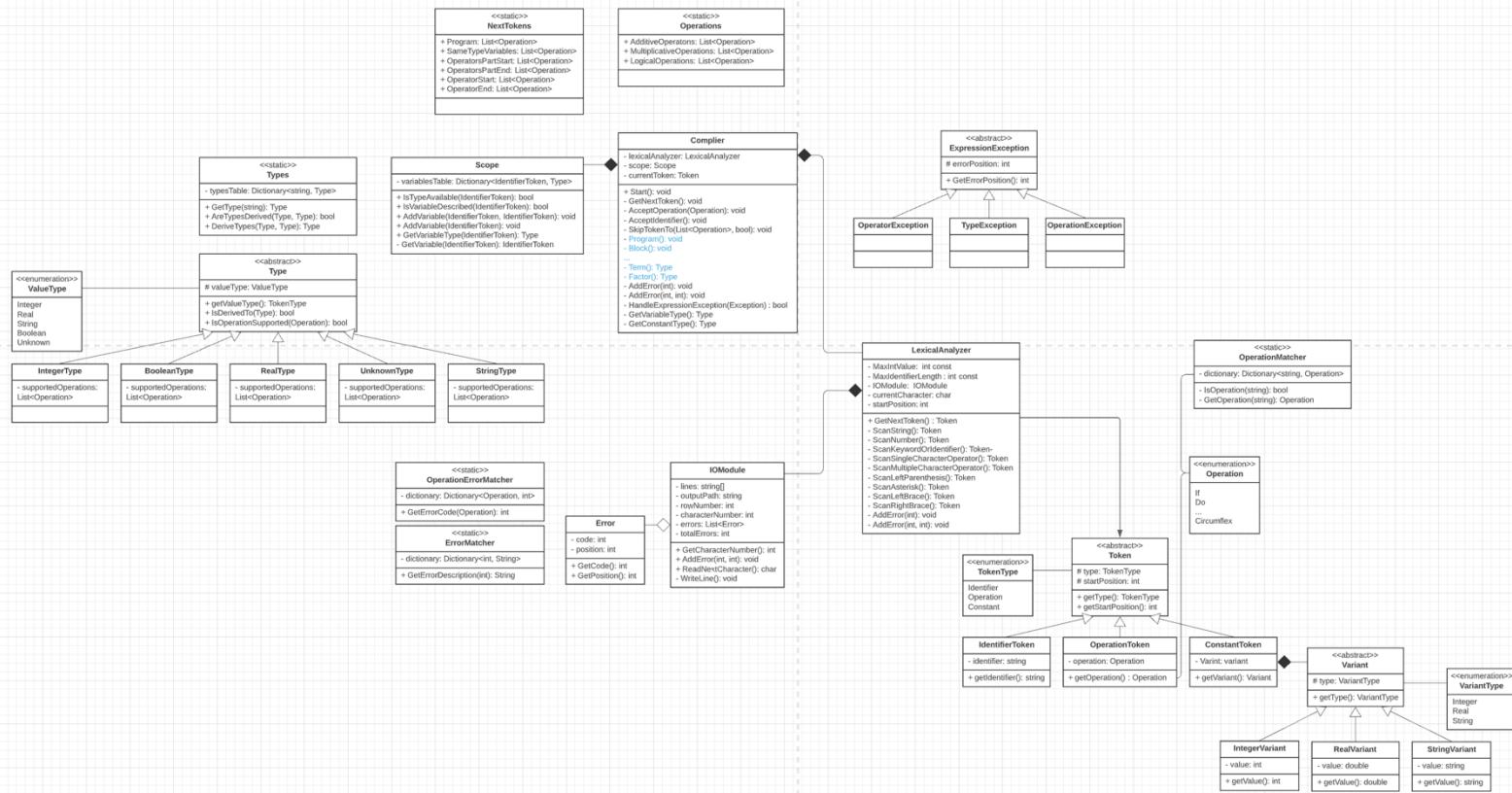


Рисунок 3 – текущая диаграмма классов компилятора

3. Реализация

Начнём с реализации класса **Compiler**. Все методы, соответствующие БНФ, реализованы следующим образом: все конструкции, в которых могут возникнуть синтаксические ошибки, заключаются в блок **try**. А в блоке **catch** происходит обработка возникших ошибок, и в случае, если это необходимо, вызываем метод **SkipTokensTo**, в качестве параметра которому передаётся один из списков в статическом классе **NextTokens**. Также один из параметров метода **SkipTokensTo** булевский флаг **block**. Мне пришлось реализовать это таким образом, потому что идентификатор не является операцией, но во многих формах необходимо пропускать токены до определённых операций, либо до идентификатора.

В методе описания однотипных 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 пропускаем 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**, запоминания 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 следующий значений **integer**, **real**, **string**, а также название ни одной из переменных не совпадает с названием переданного типа.

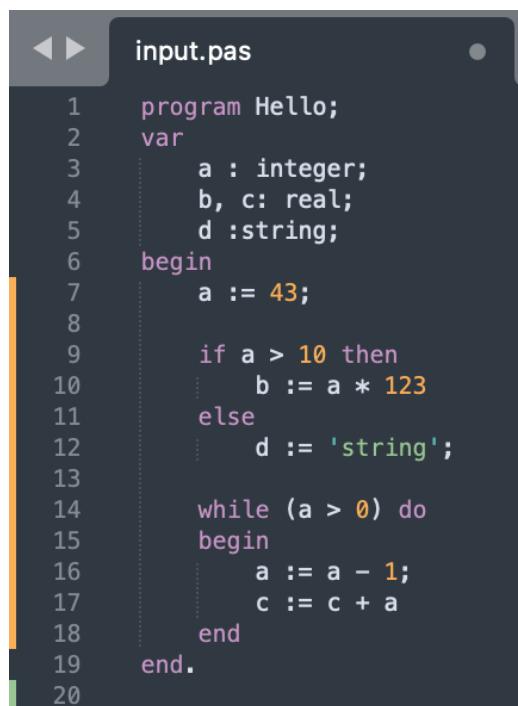
При проверке, описана ли переменная в методе **IsVariableDescribed**, просто проходимся по всем ключам и смотрит идентификаторы.

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

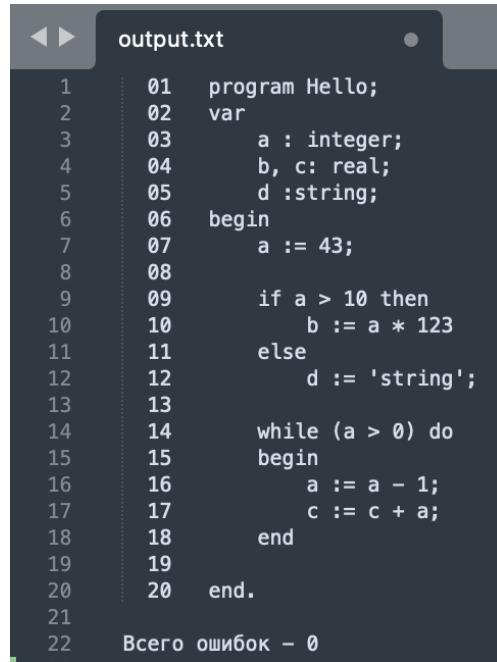
При тестировании необходимо проверить корректность работы компилятора на программе без ошибок, а также обнаружение и нейтрализацию синтаксических и семантических ошибок.

- 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.
20
```

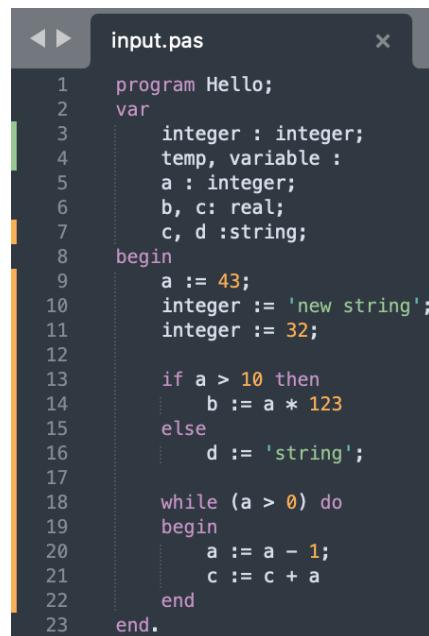
Содержимое выходного файла:



```
01  program Hello;
02  var
03      a : integer;
04      b, c: real;
05      d :string;
06  begin
07      a := 43;
08
09      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
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:



```
program Hello;
var
    integer : integer;
    temp, variable :
    a : integer;
    b, c: real;
    c, d :string;
begin
    a := 43;
    integer := 'new string';
    integer := 32;

    if a > 10 then
        b := a * 123
    else
        d := 'string';

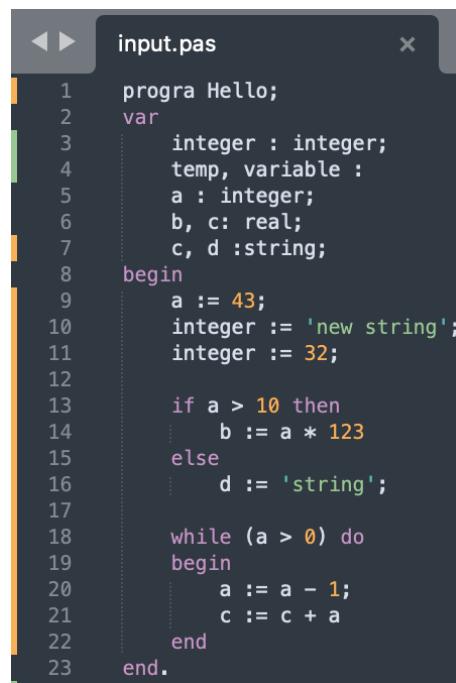
    while (a > 0) do
        begin
            a := a - 1;
            c := c + a
        end
end.
```

Содержимое выходного файла:

```
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
```

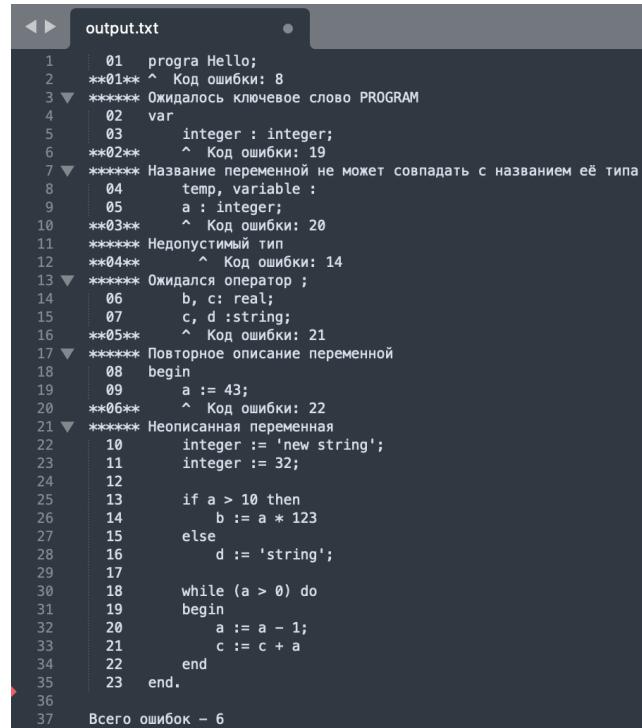
Как видно из выходного файла, все синтаксические и семантические ошибки успешно обнаружены и нейтрализованы. А именно на 3 строке переменная `integer` будет иметь неизвестный тип, т.к. при её описании использовался недопустимый тип. Далее, в 4-5 строках опять возникнет ошибка недопустимого типа, т.к. `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.
```

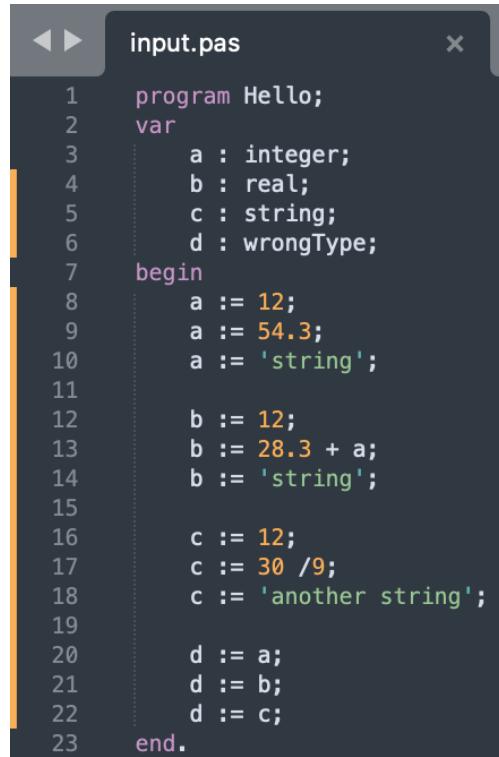
Содержимое выходного файла:



```
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
25  13      if a > 10 then
26  14          b := a * 123
27  15      else
28  16          d := 'string';
29
30  18      while (a > 0) do
31  19      begin
32  20          a := a - 1;
33  21          c := c + a
34  22      end
35
36  23 end.
37 Всего ошибок - 6
```

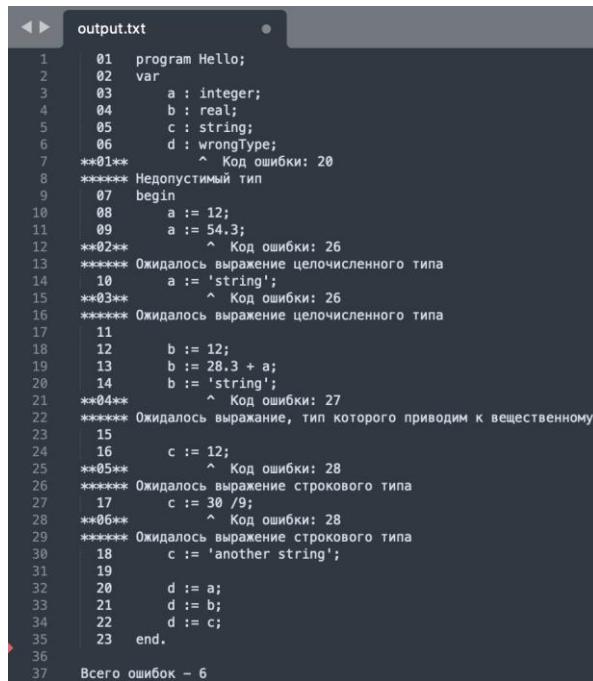
Можно заметить, что принейтрализации синтаксической ошибки в первой строке токены были пропущены только до начала раздела переменных. Таким образом, было обнаружено максимальное количество ошибок.

4) Теперь проверим приводимость типов:



```
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:

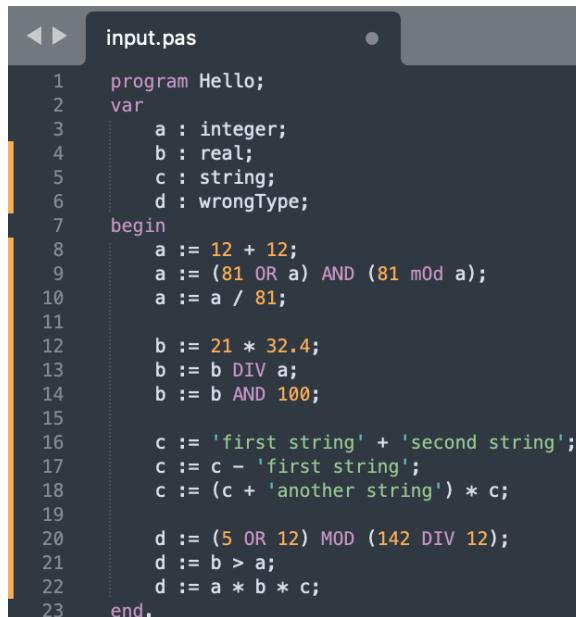


```
1 01 program Hello;
2 02 var
3 03     a : integer;
4 04     b : real;
5 05     c : string;
6 06     d : wrongType;
7 **01**     ^ Код ошибки: 20
8 ***** Недопустимый тип
9 07 begin
10 08     a := 12;
11 09     a := 54.3;
12 **02**     ^ Код ошибки: 26
13 ***** Ождалось выражение целочисленного типа
14 10     a := 'string';
15 **03**     ^ Код ошибки: 26
16 ***** Ождалось 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 ***** Ождалось выражение, тип которого приводим к вещественному
23 15
24 16     c := 12;
25 **05**     ^ Код ошибки: 28
26 ***** Ождалось выражение строкового типа
27 17     c := 30 / 9;
28 **06**     ^ Код ошибки: 28
29 ***** Ождалось 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

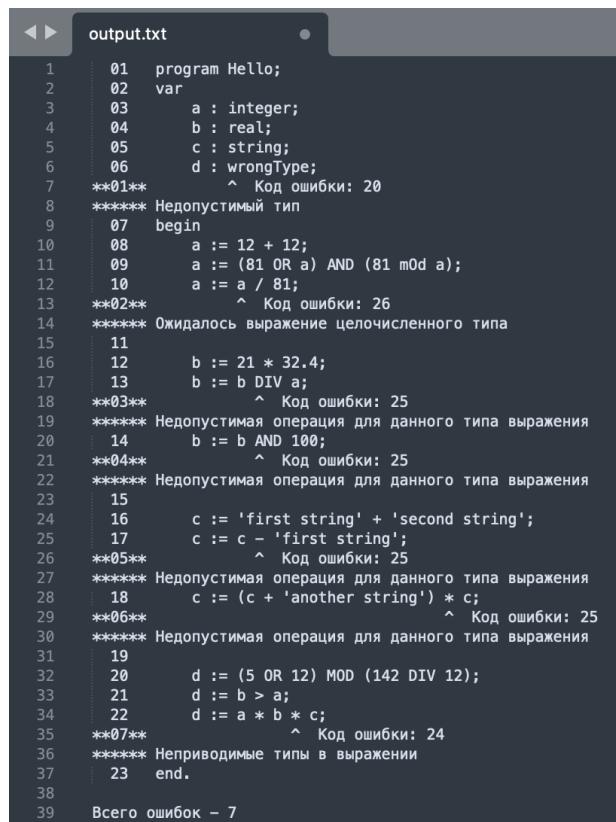
переменной – только строку. Переменная `d` имеет неизвестный тип, поэтому она может принимать любые значения.

- 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
***** Недопустимый тип
8 begin
9
10    a := 12 + 12;
11    a := (81 OR a) AND (81 mOd a);
12    a := a / 81;
13 **02**          ^ Код ошибки: 26
***** Ождалось выражение целочисленного типа
14
15
16    b := 21 * 32.4;
17    b := b DIV a;
18 **03**          ^ Код ошибки: 25
***** Недопустимая операция для данного типа выражения
19    b := b AND 100;
20 **04**          ^ Код ошибки: 25
***** Недопустимая операция для данного типа выражения
21
22
23
24    c := 'first string' + 'second string';
25    c := c - 'first string';
26 **05**          ^ Код ошибки: 25
***** Недопустимая операция для данного типа выражения
27    c := (c + 'another string') * c;
28 **06**          ^ Код ошибки: 25
***** Недопустимая операция для данного типа выражения
29
30
31
32    d := (5 OR 12) MOD (142 DIV 12);
33    d := b > a;
34    d := a * b * c;
35 **07**          ^ Код ошибки: 24
***** Неприводимые типы в выражении
36
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 operandам применимы любые операции, реализованные в заданном подмножестве языка. Для вещественных operandов применимы операции сложения, вычитания, умножения, а также вещественного деления. Для строк допустима только операция сложения. Логические операции применимы для любых типов, если operandы приводимы. Переменная неизвестного типа поддерживает все операции.

- 6) 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  ***** Ождалось выражение логического типа
10 08          a := 4
11 09      else
12 10          a := 12.21;
13  **02**           ^ Код ошибки: 26
14  ***** Ождалось выражение целочисленного типа
15 11
16 12      if a + 5 > 12 then
17  **03**           ^ Код ошибки: 11
18  ***** Ождалось ключевое слово THEN
19 13          a := a * 12
20 14      else
21 15          ))))))));
22  **04**           ^ Код ошибки: 17
23  ***** Ождался оператор
24 16          a := 'another string';
25  **05**           ^ Код ошибки: 26
26  ***** Ождалось выражение целочисленного типа
27 17
28 18  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         a := ((12 * 5);
15     end
16
17 end.

```

Содержимое выходного файла:

```

01  program Hello;
02  var
03      a : integer;
04  begin
05      a := 120;
06
07      while (a OR 5) + (12 AND 18) does )) var array
08      ^
09      **01** Ожидалось выражение логического типа
10      ****02** Ожидалось ключевое слово DO
11      ^
12      **03** Ожидалось ключевое слово begin
13      ^
14      **04** Ожидался оператор :=
15      ^
16      **05** Ошибка в выражении
17      ^
18      ****06** Ожидалось оператор :=
19
20
21      ^
22      **07** Ошибочка в выражении
23      ^
24      ****08** Ожидалось ключевое слово end
25      ^
26
27      ****09** Ожидалось ключевое слово end.

```

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 присваивая. 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:

```

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:

```

1  01  program Hello;
2  02  var
3  03      a : integer;
4  04      b : real;
5  05
6  06      while a > 0 do
7  ***01** ^ Код ошибки: 9
8  ***** Ожидалось ключевое слово BEGIN
9  07      a := a - 1;
10
11 09      if a = 0 then
12 10      a : 1000;
13
14 12 begin
15 13      a := 100;
16 14      b := 12.3;
17
18 16      if a > 0 then
19 17      begin
20 18          a := a mod 100;
21 19          b := b * a
22
23 21 end.
24 ***02** ^ Код ошибки: 10
25 ***** Ожидалось ключевое слово END
26
27 Всего ошибок - 2

```

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 теле 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.

Генератор кода

1. Описание

Необходимо создать генератор кода, который будет строить инструкции языка MSIL, соответствующие этой программе. После чего будет создаваться файл с разрешением exe.

Также необходимо в заданное подмножество языка добавить поддержку процедуры `writeln` для того, чтобы была возможность убедиться в корректности работы генератора, запустив получившийся исполняемый файл.

Стоит отметить, что генератор, как ни странно, правильно работает только на полностью корректной программе, в которой не содержатся лексические, синтаксические, а также семантические ошибки.

2. Проектирование

Для генерации кода нет необходимости создавать новые классы, достаточно модифицировать уже имеющиеся.

Во-первых, необходимо модифицировать метод `Main` в классе `Program` таким образом, чтобы в нём определялась динамическая сборка с одним модулем. Этот модуль содержит один тип, который имеет единственный публичный и статический метод, соответствующий переданной на вход программе. После работы компилятора, когда все MSIL коды будут уже записаны, необходимо сохранить полученную сборку с расширением exe.

Для добавления MSIL инструкций необходимо модифицировать класс `Compiler`, добавив в качестве одного из полей экземпляр класса `ILGenerator`. Этот экземпляр будет получен в ходе описания метода в сборке, после чего будет передан в конструктор класса `Compiler`.

Далее, для возможности проверки корректности работы генератора необходимо добавить в заданное подмножество языка процедуру `writeln`, которая выводит полученный аргумент в консоль.

Необходимо создать метод **DeclareVariable** в классе **Compiler**, в котором будут происходить декларации переменных, вызывая соответствующий метод **ILGenerator** с переданным типом.

Поскольку в генераторе работа с переменными идёт по их индексам (т.е. порядковыми номерами, в котором они были объявлены) необходимо добавить в класс **Scope** метод, возвращающий индекс переменной по переданному **IdentifierToken**.

После чего нужно модифицировать методы условного оператора и оператора цикла таким образом, чтобы в этих методах появилась возможность передачи управления в зависимости от выполнимости соответствующих условий.

Также необходимо модифицировать методы, вызывающиеся при встрече выражения таким образом, чтобы в них записывались инструкции по вычислению выражения. Для этого необходимо организовать работу со стеком, в который буду заноситься встреченные константы и переменные, а при встрече операций выполнять их для двух верхних элементов стека и заносить результат опять на вершину. Таким образом вычисляются выражения, записанные в постфиксной форме.

3. Реализация

Метод **DeclareVariable** вызывается для каждой описываемой переменной. В этом методе вызывается метод **DeclareLocal** класса **ILGenerator**, в который передаётся тип описываемой переменной.

В методе **Operator** теперь если текущий токен является идентификатором не сразу вызывается метод оператора присваивания, а проверяется значение идентификатора. Если значением будет названием процедуры **writeln**, то вызывается соответствующий метод, иначе – метод оператора присваивания.

В методе **WriteLn** сначала принимается «(», затем вызывается метод **Expression** и запоминается его тип, далее принимается «)». После чего

объявляется объект класса `MethodInfo`. Далее этому объекту присваивается метод `WriteLine` класса `Console`, с единственным параметром типа, соответствующего типу выражения.

Метод оператора присваивания был изменён следующим образом: после того, как выражение справа от оператора присваивания было вычислено, его значение извлекается из стека и сохраняется в локальную переменную, стоящую слева от оператора.

Для модифицирования метода `IfOperator` были использованы метки. Если в ходе вычисления выражения на вершине стека оказалось значение `false`, то управление переходит к метке, расположенной после вызова метода `Operator` после ключевого слова `then`, перед проверкой текущего токена на значение ключевого слова `else`. Иначе перед этой проверкой управление передаётся в метку, расположенную после этой проверки и вызова оператора после неё.

В методе оператора цикла также были использованы метки. Если в ходе вычисления условия в операторе на вершине стека оказалось значение `true`, то вызывается метод `Operator`, соответствующий телу цикла, после чего управление передаётся к метке, расположенной перед проверкой условия цикла. Если же условие оказалось ложным – управление переходит к метке, расположенной в конце метода.

Для добавления MSIL инструкций операций в выражениях был создан вспомогательный метод `EmitOperation`. Этот метод просто принимает на вход операцию (аддитивную, мультипликативную или операцию отношения), после чего добавляет в поток инструкций соответствующие ей MSIL инструкции из класса `OpCodes`. Записанные инструкции выполняются следующим образом: с вершины стека снимаются два верхних значения, для них выполняются соответствующие инструкции, после чего результат кладётся обратно на вершину стека, что соответствует постфиксной записи выражений.

В методах `Expression`, `SimpleExpression` и `Term` после вызова соответствующих методов для левой и правой частей вызывается метод `EmitOperation` с операцией, которой связаны эти части.

Стоит упомянуть, что здесь есть одно исключение. Дело в том, что строки являются ссылочным типом и в стек помещаются ссылки на них. И при попытке сложения строк будет происходить сложение ссылок, что повлечёт за собой ошибку. Я решил это следующим образом: если в методе `SimpleExpression` текущая операция – сложение, а тип у operandов строковый, то вместо метода `EmitOperation` я вызываю метод `ConcatStrings`.

В этом методе я инициализирую объект класса `MethodInfo` методом `Concat` класса `string`, после чего он применяется к двум верхним элементам стека, а результат кладётся на вершину.

В методе `GetConstantType`, вызываемом из метода `Factor`, если текущий токен является константой, я также помещаю полученное значение константы на стек.

А в методе `GetVariableType` находится и загружается на стек переменная, соответствующая названию текущего `IdentifierToken`.

4. Тестирование

Для проверки корректности записываемых инструкций языка MSIL я воспользовался сайтом [SharpLab](#). На этом сайте я создавал программы на языке C#, идентичные тем, что я создавал на заданном подмножестве языка Pascal. После чего я, с помощью программы IL DASM получал инструкции на языке MSIL моих программ на подмножестве языке Pascal и сравнивал полученные результаты.

- 1) Начнём с чего-нибудь простого. Например, сразу в разделе операторов вызовем процедуру `WriteLn` с разными типами аргументов:

The screenshot shows a code editor window with a dark theme. The file is named "input.pas". The code contains a single program block:

```
1 program Hello;
2 begin
3     writeln(17 * 3 div 4);
4     writeln(23.5 / 10);
5     writeln('Hello ' + 'World!');
6     writeln(100 <> 32)
7 end.
```

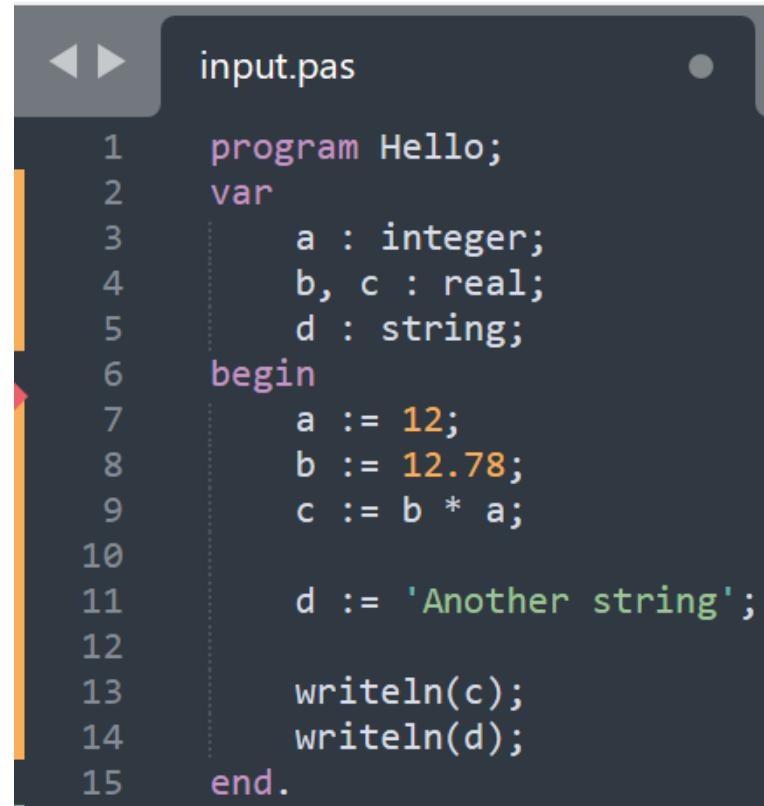
Полученные инструкции на языке MSIL:

```
.method public static void Main() cil managed
{
    .entrypoint
    // Размер кода:      83 (0x53)
    .maxstack 2
    IL_0000: ldc.i4    0x11
    IL_0005: ldc.i4    0x3
    IL_000a: mul
    IL_000b: ldc.i4    0x4
    IL_0010: div
    IL_0011: call      void [mscorlib]System.Console::WriteLine(int32)
    IL_0016: ldc.r8    23.5
    IL_001f: ldc.i4    0xa
    IL_0024: div
    IL_0025: call      void [mscorlib]System.Console::WriteLine(float64)
    IL_002a: ldstr     "Hello "
    IL_002f: ldstr     "World!"
    IL_0034: call      string [mscorlib]System.String::Concat(string,
                                         string)
    IL_0039: call      void [mscorlib]System.Console::WriteLine(string)
    IL_003e: ldc.i4    0x64
    IL_0043: ldc.i4    0x20
    IL_0048: ceq
    IL_004a: ldc.i4.0
    IL_004b: ceq
    IL_004d: call      void [mscorlib]System.Console::WriteLine(bool)
    IL_0052: ret
} // end of method Program::Main
```

Результат запуска исполняемого файла из командной строки:

```
C:\Users\Justk>C:\Users\Justk\Documents\GitHub\PascalCompiler\PascalCompiler\bin\Debug\Program.exe
12
2,35
Hello World!
True
```

2) Теперь попробуем добавить переменных разных типов в программу:



```
program Hello;
var
  a : integer;
  b, c : real;
  d : string;
begin
  a := 12;
  b := 12.78;
  c := b * a;
  d := 'Another string';
  writeln(c);
  writeln(d);
end.
```

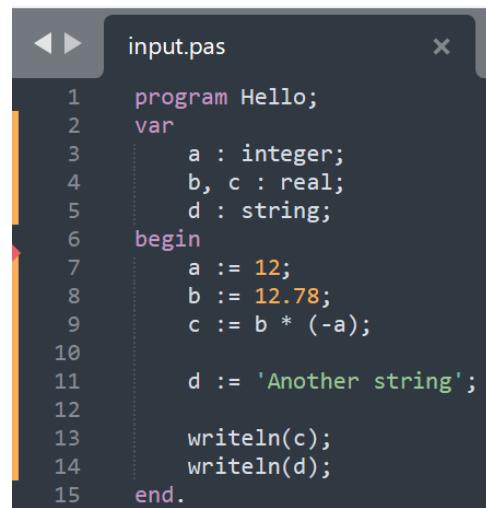
Полученные инструкции на языке MSIL:

```
.method public static void Main() cil managed
{
    .entrypoint
    // Размер кода:      79 (0x4f)
    .maxstack 2
    .locals init ([0] int32 V_0,
                 [1] float64 V_1,
                 [2] float64 V_2,
                 [3] string V_3)
    IL_0000: ldc.i4    0xc
    IL_0005: stloc     V_0
    IL_0009: nop
    IL_000a: nop
    IL_000b: ldc.r8    12.77999999999999
    IL_0014: stloc     V_1
    IL_0018: nop
    IL_0019: nop
    IL_001a: ldloc     V_1
    IL_001e: nop
    IL_001f: nop
    IL_0020: ldloc     V_0
    IL_0024: nop
    IL_0025: nop
    IL_0026: mul
    IL_0027: stloc     V_2
    IL_002b: nop
    IL_002c: nop
    IL_002d: ldstr     "Another string"
    IL_0032: stloc     V_3
    IL_0036: nop
    IL_0037: nop
    IL_0038: ldloc     V_2
    IL_003c: nop
    IL_003d: nop
    IL_003e: call      void [mscorlib]System.Console::WriteLine(float64)
    IL_0043: ldloc     V_3
    IL_0047: nop
    IL_0048: nop
    IL_0049: call      void [mscorlib]System.Console::WriteLine(string)
    IL_004e: ret
} // end of method Program::Main
```

Результат запуска исполняемого файла из командной строки:

```
C:\Users\Justk>C:\Users\Justk\Documents\GitHub\PascalCompiler\PascalCompiler\bin\Debug\Program.exe
153,36
Another string
```

3) На этапе тестирования генератора кода, когда я заметил, что неправильно реализовал метод, соответствующий БНФ простого выражения, в моей реализации перед первым слагаемым не могло быть знака, теперь я это исправил. Изменим выражения в коде, чтобы это проверить:



```
program Hello;
var
  a : integer;
  b, c : real;
  d : string;
begin
  a := 12;
  b := 12.78;
  c := b * (-a);
  d := 'Another string';
  writeln(c);
  writeln(d);
end.
```

Полученные инструкции на языке MSIL:

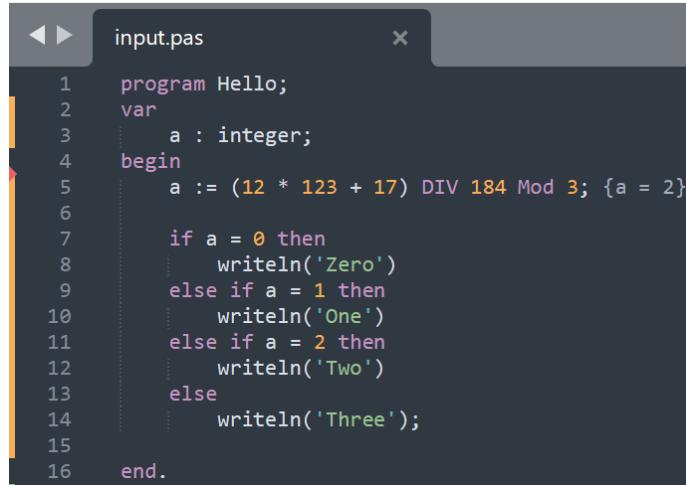
```
.method public static void Main() cil managed
{
    .entrypoint
    // Размер кода:      85 (0x55)
    .maxstack 3
    .locals init ([0] int32 V_0,
                 [1] float64 V_1,
                 [2] float64 V_2,
                 [3] string V_3)
    IL_0000: ldc.i4    0xc
    IL_0005: stloc     V_0
    IL_0009: nop
    IL_000a: nop
    IL_000b: ldc.r8    12.77999999999999
    IL_0014: stloc     V_1
    IL_0018: nop
    IL_0019: nop
    IL_001a: ldloc     V_1
    IL_001e: nop
    IL_001f: nop
    IL_0020: ldloc     V_0
    IL_0024: nop
    IL_0025: nop
    IL_0026: ldc.i4    0xFFFFFFFF
    IL_002b: mul
    IL_002c: mul
    IL_002d: stloc     V_2
    IL_0031: nop
    IL_0032: nop
    IL_0033: ldstr     "Another string"
    IL_0038: stloc     V_3
    IL_003c: nop
    IL_003d: nop
    IL_003e: ldloc     V_2
    IL_0042: nop
    IL_0043: nop
    IL_0044: call      void [mscorlib]System.Console::WriteLine(Float64)
    IL_0049: ldloc     V_3
    IL_004d: nop
    IL_004e: nop
    IL_004f: call      void [mscorlib]System.Console::WriteLine(string)
    IL_0054: ret
} // end of method Program::Main
```

Результат запуска исполняемого файла из командной строки:

```
C:\Users\Justk>C:\Users\Justk\Documents\GitHub\PascalCompiler\PascalCompiler\bin\Debug\Program.exe
-153,36
Another string
```

Как видно из результатов тестирования, теперь выражения работают корректно, если перед первым слагаемым в простом выражении стоит знак.

4) Теперь проверим корректность работы с условным оператором if:



```
program Hello;
var
  a : integer;
begin
  a := (12 * 123 + 17) DIV 184 Mod 3; {a = 2}
  if a = 0 then
    writeln('Zero')
  else if a = 1 then
    writeln('One')
  else if a = 2 then
    writeln('Two')
  else
    writeln('Three');
end.
```

Полученные инструкции на языке MSIL:

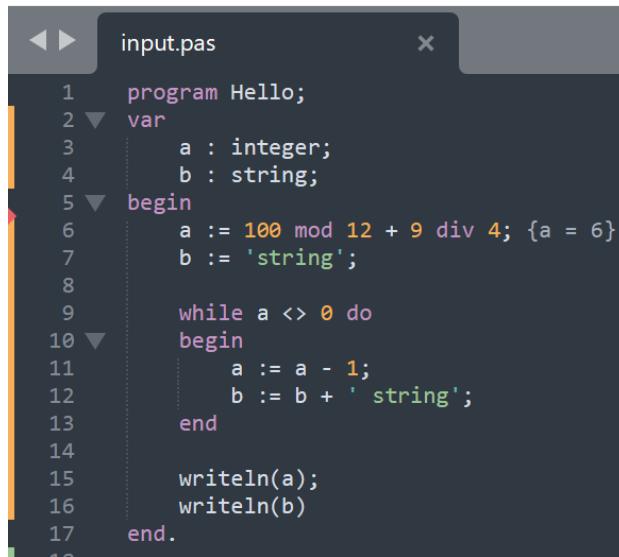
```
.method public static void Main() cil managed
{
  .entrypoint
  // Размер кода:      121 (0x79)
  .maxstack 7
  .locals init ([0] int32 v_0)
  IL_0000: ldc.i4    0xc
  IL_0005: ldc.i4    0x7b
  IL_000a: nul
  IL_000b: ldc.i4    0xb8
  IL_0010: div
  IL_0011: ldc.i4    0x3
  IL_0016: rem
  IL_0017: stloc    v_0
  IL_001b: nop
  IL_001c: nop
  IL_001d: ldloc    v_0
  IL_0021: nop
  IL_0022: nop
  IL_0023: ldc.i4    0x0
  IL_0028: ceq
  IL_002a: brfalse.s IL_0038
  IL_002c: ldstr    "Zero"
  IL_0031: call     void [mscorlib]System.Console::WriteLine(string)
  IL_0036: br.s     IL_0078
  IL_0038: ldloc    v_0
  IL_003c: nop
  IL_003d: nop
  IL_003e: ldc.i4    0x1
  IL_0043: ceq
  IL_0045: brfalse.s IL_0053
  IL_0047: ldstr    "One"
  IL_004c: call     void [mscorlib]System.Console::WriteLine(string)
  IL_0051: br.s     IL_0078
  IL_0053: ldloc    v_0
  IL_0057: nop
  IL_0058: nop
  IL_0059: ldc.i4    0x2
  IL_005e: ceq
  IL_0060: brfalse.s IL_006e
  IL_0062: ldstr    "Three"
  IL_0067: call     void [mscorlib]System.Console::WriteLine(string)
  IL_006c: br.s     IL_0078
  IL_006e: ldstr    "Four"
  IL_0073: call     void [mscorlib]System.Console::WriteLine(string)
  IL_0078: ret
} // end of method Program::Main
```

Результат запуска исполняемого файла из командной строки:

```
C:\Users\Justk>C:\Users\Justk\Documents\GitHub\PascalCompiler\PascalCompiler\bin\Debug\Program.exe  
Two
```

Как видно из результатов в командной строке, работа условного оператора корректна. Также из инструкций на языке MSIL можно заметить, что работа с метками происходит корректно.

5) Теперь проведём тесты для оператора цикл:



```
input.pas
1  program Hello;
2  var
3      a : integer;
4      b : string;
5  begin
6      a := 100 mod 12 + 9 div 4; {a = 6}
7      b := 'string';
8
9      while a <> 0 do
10     begin
11         a := a - 1;
12         b := b + ' string';
13     end
14
15     writeln(a);
16     writeln(b)
17 end.
```

Полученные инструкции на языке MSIL:

```
.method public static void Main() cil managed
{
    .entrypoint
    // Размер кода:      123 (0x7b)
    .maxstack 4
    .locals init ([0] int32 V_0,
                 [1] string V_1)
    IL_0000: ldc.i4    0x64
    IL_0005: ldc.i4    0xc
    IL_000a: rem
    IL_000b: ldc.i4    0x9
    IL_0010: ldc.i4    0x4
    IL_0015: div
    IL_0016: add
    IL_0017: stloc    V_0
    IL_001b: nop
    IL_001c: nop
    IL_001d: ldstr     "string"
    IL_0022: stloc    V_1
    IL_0026: nop
    IL_0027: nop
    IL_0028: ldloc    V_0
    IL_002c: nop
    IL_002d: nop
    IL_002e: ldc.i4    0x0
    IL_0033: ceq
    IL_0035: ldc.i4.0
    IL_0036: ceq
    IL_0038: brfalse.s IL_0064
    IL_003a: ldloc    V_0
    IL_003e: nop
    IL_003f: nop
    IL_0040: ldc.i4    0x1
    IL_0045: sub
    IL_0046: stloc    V_0
    IL_004a: nop
    IL_004b: nop
    IL_004c: ldloc    V_1
    IL_0050: nop
    IL_0051: nop
```

```

IL_0052: ldstr      " string"
IL_0057: call       string [mscorlib]System.String::Concat(string,
                                         string)
IL_005c: stloc      V_1
IL_0060: nop
IL_0061: nop
IL_0062: br.s       IL_0028
IL_0064: ldloc      V_0
IL_0068: nop
IL_0069: nop
IL_006a: call       void [mscorlib]System.Console::WriteLine(int32)
IL_006f: ldloc      V_1
IL_0073: nop
IL_0074: nop
IL_0075: call       void [mscorlib]System.Console::WriteLine(string)
IL_007a: ret
} // end of method Program::Main

```

Результат запуска исполняемого файла из командной строки:

```
C:\Users\Justk>C:\Users\Justk\Documents\GitHub\PascalCompiler\PascalCompiler\bin\Debug\Program.exe
0
string string string string string string
```

Как видно из инструкций MSIL и результатов в командной строке, оператор цикла также работает корректно.

6) Объединим операторы цикла и условный операторов в одной программе:

```

program Hello;
var
  a : integer;
begin
  a := 100 mod 12 + 9 div 4; {a = 6}
  while a >= 0 do
    begin
      writeln(a);
      if a mod 2 = 0 then
        writeln('This number is even')
      else
        writeln('And this is an odd one');
      a := a - 1;
    end;
end.

```

Полученные инструкции на языке MSIL:

```
.method public static void Main() cil managed
{
    .entrypoint
    // Размер кода:      122 (0x7a)
    .maxstack 5
    .locals init ([0] int32 V_0)
    IL_0000: ldc.i4    0x64
    IL_0005: ldc.i4    0xc
    IL_000a: rem
    IL_000b: ldc.i4    0x9
    IL_0010: ldc.i4    0x4
    IL_0015: div
    IL_0016: add
    IL_0017: stloc    V_0
    IL_001b: nop
    IL_001c: nop
    IL_001d: ldloc    V_0
    IL_0021: nop
    IL_0022: nop
    IL_0023: ldc.i4    0x0
    IL_0028: clt
    IL_002a: ldc.i4.0
    IL_002b: ceq
    IL_002d: brfalse.s IL_0079
    IL_002f: ldloc    V_0
    IL_0033: nop
    IL_0034: nop
    IL_0035: call      void [mscorlib]System.Console::WriteLine(int32)
    IL_003a: ldloc    V_0
    IL_003e: nop
    IL_003f: nop
    IL_0040: ldc.i4    0x2
    IL_0045: rem
    IL_0046: ldc.i4    0x0
    IL_004b: ceq
    IL_004d: brfalse.s IL_005b
    IL_004f: ldstr     "This number is even"
    IL_0054: call      void [mscorlib]System.Console::WriteLine(string)
    IL_0059: br.s      IL_0065
    IL_005b: ldstr     "And this is an odd one"
    IL_0060: call      void [mscorlib]System.Console::WriteLine(string)
    IL_0065: ldloc    V_0
    IL_0069: nop
    IL_006a: nop
    IL_006b: ldc.i4    0x1
    IL_0070: sub
    IL_0071: stloc    V_0
    IL_0075: nop
    IL_0076: nop
    IL_0077: br.s      IL_001d
    IL_0079: ret
} // end of method Program::Main
```

Результат запуска исполняемого файла из командной строки:

```
C:\Users\Justk>C:\Users\Justk\Documents\GitHub\PascalCompiler\PascalCompiler\bin\Debug\Program.exe
6
This number is even
5
And this is an odd one
4
This number is even
3
And this is an odd one
2
This number is even
1
And this is an odd one
0
This number is even
```

Как видно из результатов тестирования, программа работает корректно, генерируется код, соответствующий исходной программе, при запуске исполняемого файла видим то, что и ожидали увидеть.

Стоит отметить, что у генератора есть ограничения. Не рассматривается случай деления на 0 – при встрече данной операции в выражении, программа просто упадёт. А также есть проблема с приводимостью типов `integer` и `real` в выражениях. Если выражение типа `real` стоит слева от операции, а выражение типа `integer` – справа, то никаких проблем не возникнет и самый первый тест это подтверждает. Но если поменять типы местами, то при операциях на вершине стека будет лежать переменная типа `real`, а за ней переменная типа `integer`. Я не придумал как приводить вторую переменную к типу `real`, и оставил невозможность вычисления таких выражений в качестве особенности моего генератора...