|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 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](#_Toc90826900)

[Модуль ввода-вывода 4](#_Toc90826901)

[1. Описание 4](#_Toc90826902)

[2. Проектирование 4](#_Toc90826903)

[3. Реализация 5](#_Toc90826904)

[4. Тестирование 7](#_Toc90826905)

[Лексический анализатор 9](#_Toc90826906)

[1. Описание 9](#_Toc90826907)

[2. Проектирование 10](#_Toc90826908)

[3. Реализация 12](#_Toc90826909)

[4. Тестирование 15](#_Toc90826910)

[Синтаксический и семантический анализ 21](#_Toc90826911)

[1. Описание 21](#_Toc90826912)

[2. Проектирование 22](#_Toc90826913)

[3. Реализация 26](#_Toc90826914)

[4. Тестирование 30](#_Toc90826915)

# Модуль ввода-вывода

## 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.

## 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.

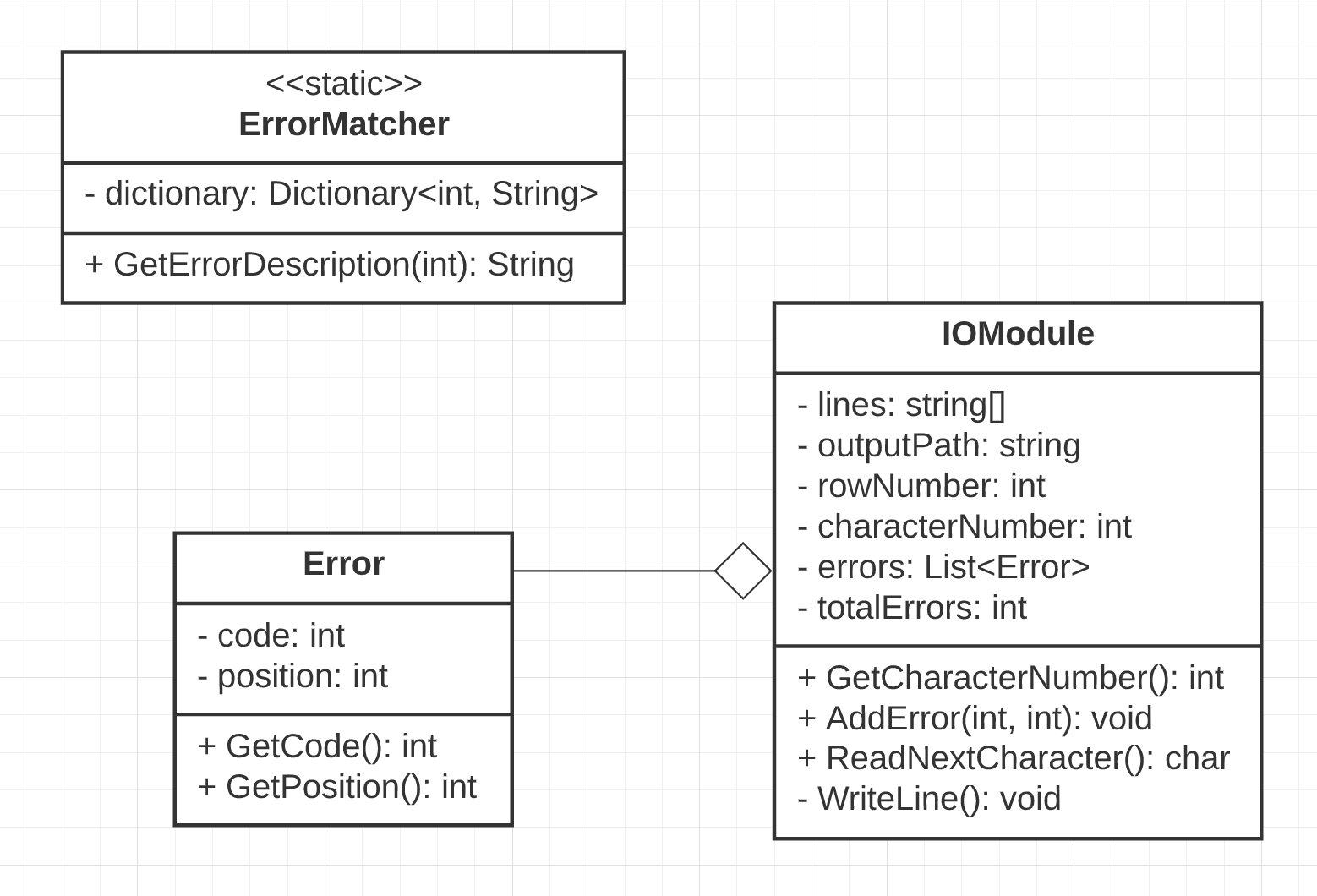
Помимо необходимой функциональности, я, в силу недостатка опыта проектирования, решил добавить в класс модуля ввода-вывода и работу с ошибками.

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**

## 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**. Метом работает следующим образом:

* Если значение **characterNumber** равно 0 и значение **rowNumber** больше 0, то вызываем метод **WriteLine**.
* Если **rowNumber** меньше длинны массива **lines** и **characterNumber** меньшедлины 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 длинной массива **lines**. Если оно получилось больше, то возвращаем символ конца файла, в противном случае символ переноса строки.

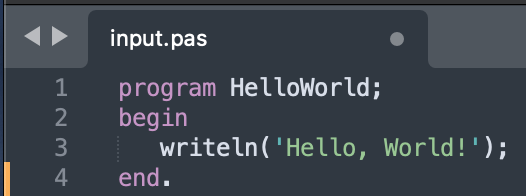
Как можно заметить, текущая строка с обнаруженными ошибками выводится при чтении первого символа следующей строки. Изначально я выводил эту информацию, когда доходил до последнего символа текущей строки. Далее, уже при тестировании лексического анализатора обнаружилась проблема такого подхода, заключающаяся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

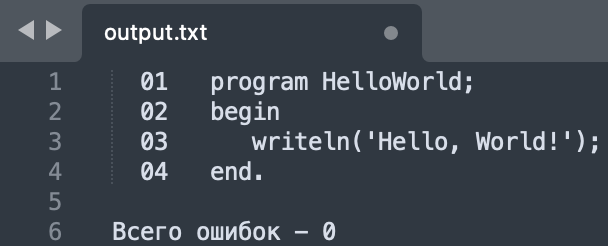
Метод **WriteLine** выводит в файл с листингом текущую строку программы с её номером. Далее, если список **errors** не пуст, увеличиваем счётчик ошибок **totalErrors** наколичество элементов в списке, после этого выводим все элементы, указывая коды и сообщения обнаруженных ошибок. Помимо этого, указывается 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.

## Тестирование

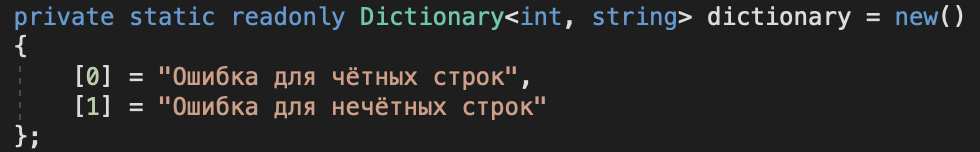
1. Для начала просто выведем содержимое входного файла в выходной на примере кода программы «Hello, World!».



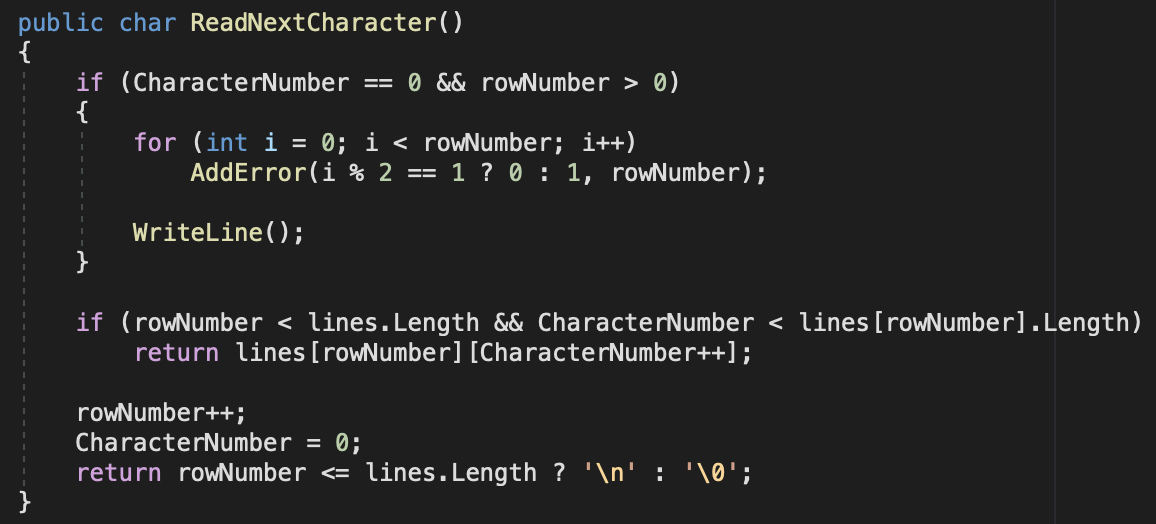


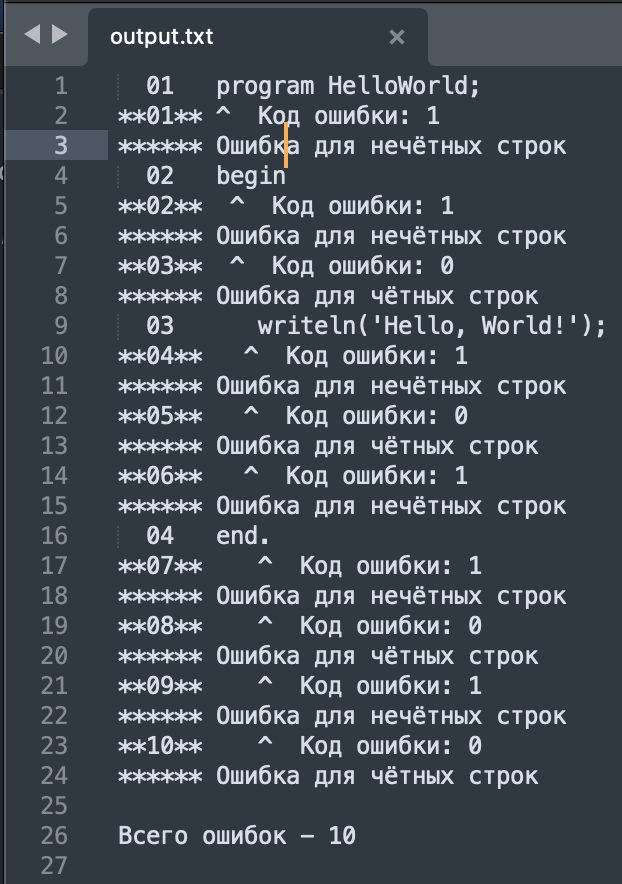
Текст исходной программы из исходного файла успешно был переписан в выходной. На удивление в нём не обнаружилось ни одной ошибки!

1. Теперь попробуем добавить в словарь в классе **ErrorMatcher** фиктивные ошибки.



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.





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

## 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.

## Проектирование

Вся логика лексического анализатора распложена 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.

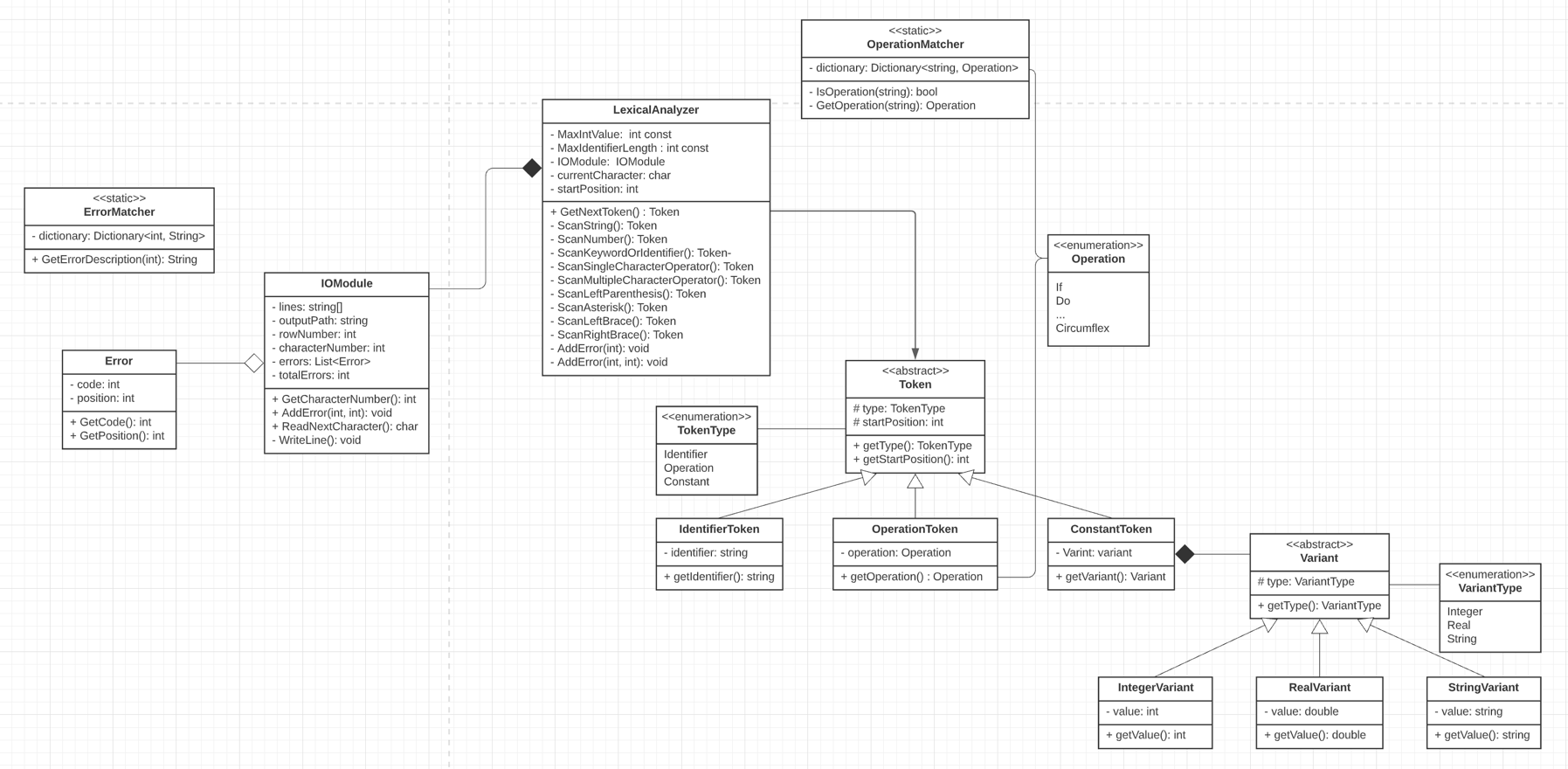
Для хранения операции в классе **OperationToken** было создано перечисление **Operation**, в которое входят все операции и ключевые слова языка.

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**

## 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 добававляем 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 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 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 «\*».

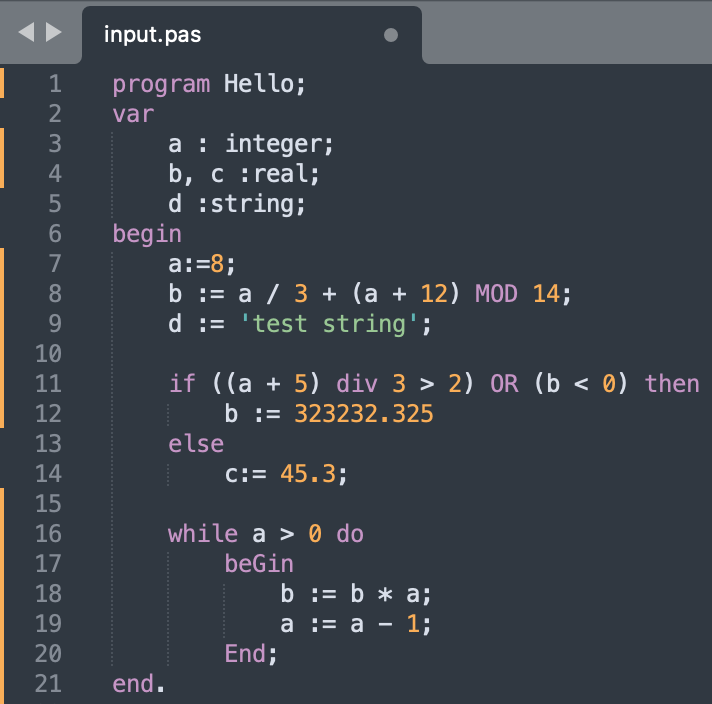
Стоит упомянуть, что во всех вышеописанных методах, в конструкторы классов, наследованных от **Token,** также передаётся позиция первой литеры этого токена в тексте исходной программы.

## 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.



Результат вывода в консоль:

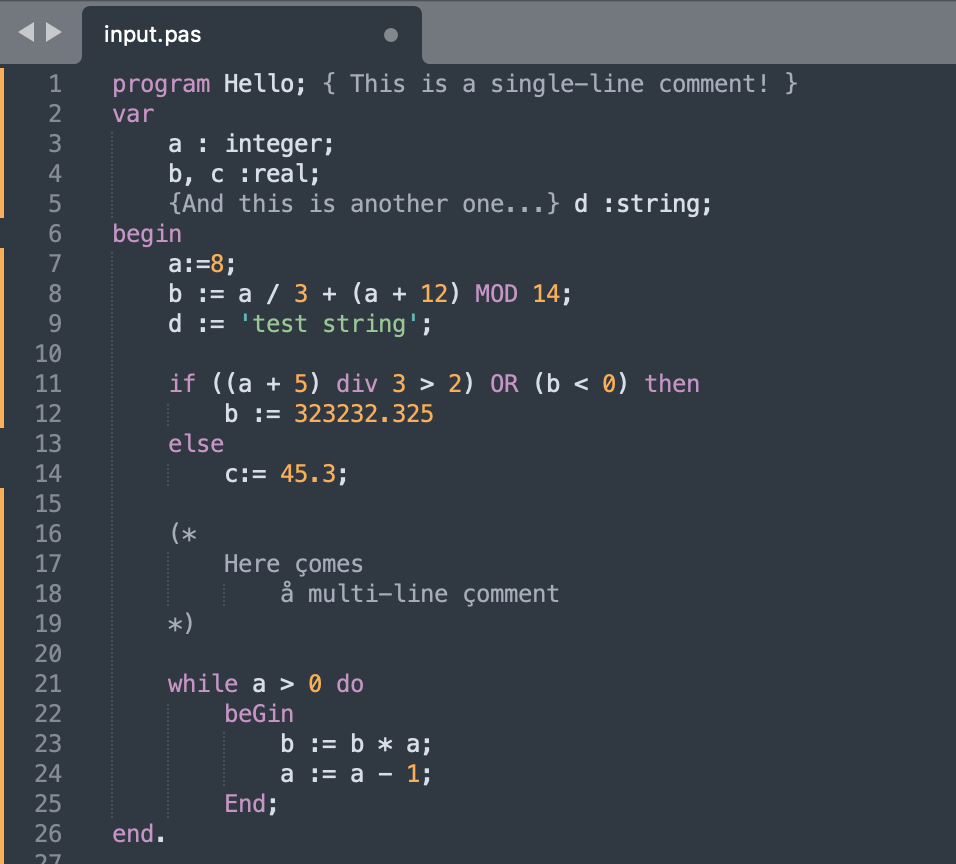




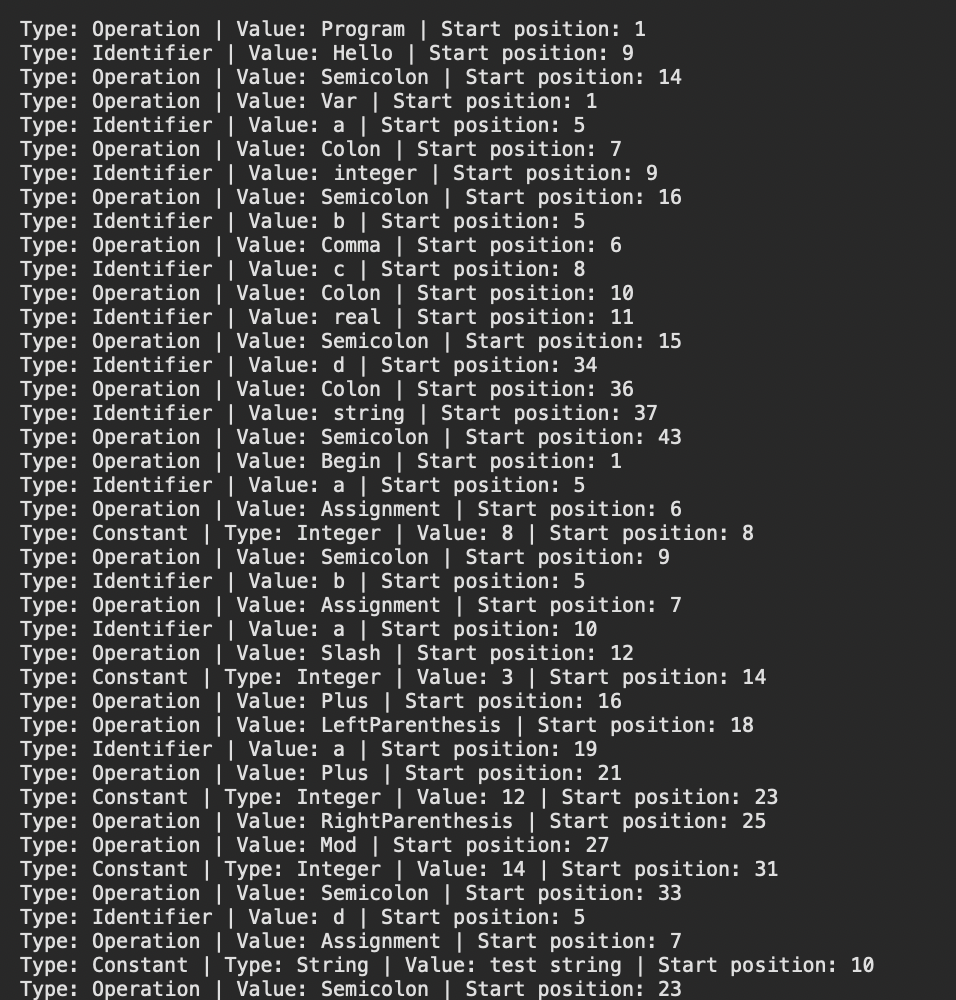
Как видно из результатов тестирования, лексический анализатор корректно определяет типы токенов, их значения, а также позицию первой литеры токена.

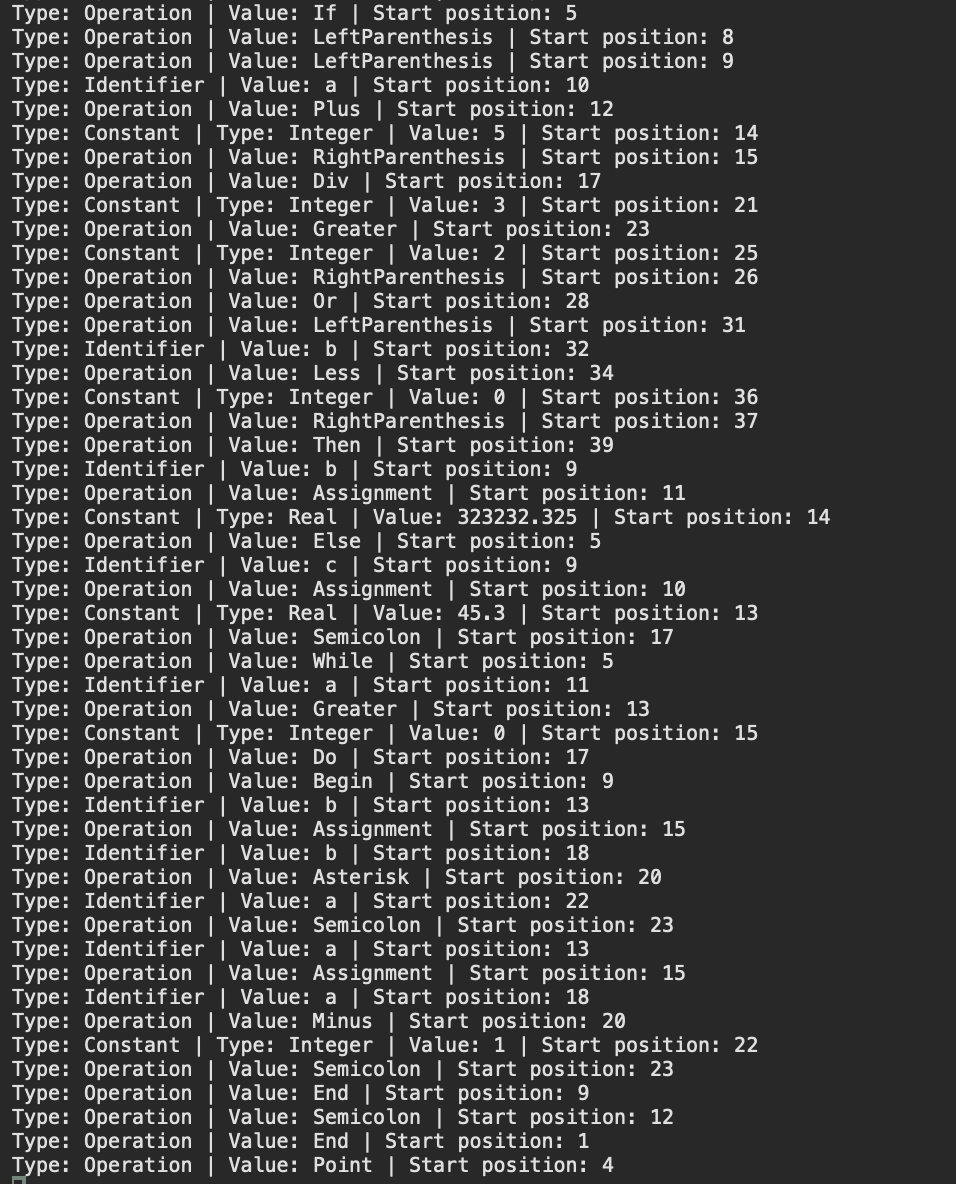
Хоть в этой программе используются и не все необходимые ключевые слова подмножества языка, дальнейшее тестирование можно пропустить в силу того, что токены строятся унифицировано в методах **ScanSingleCharacterOperator** и **ScanMultipleCharacterOperator**.

1. Добавим в программу однострочные и многострочные комментарии:

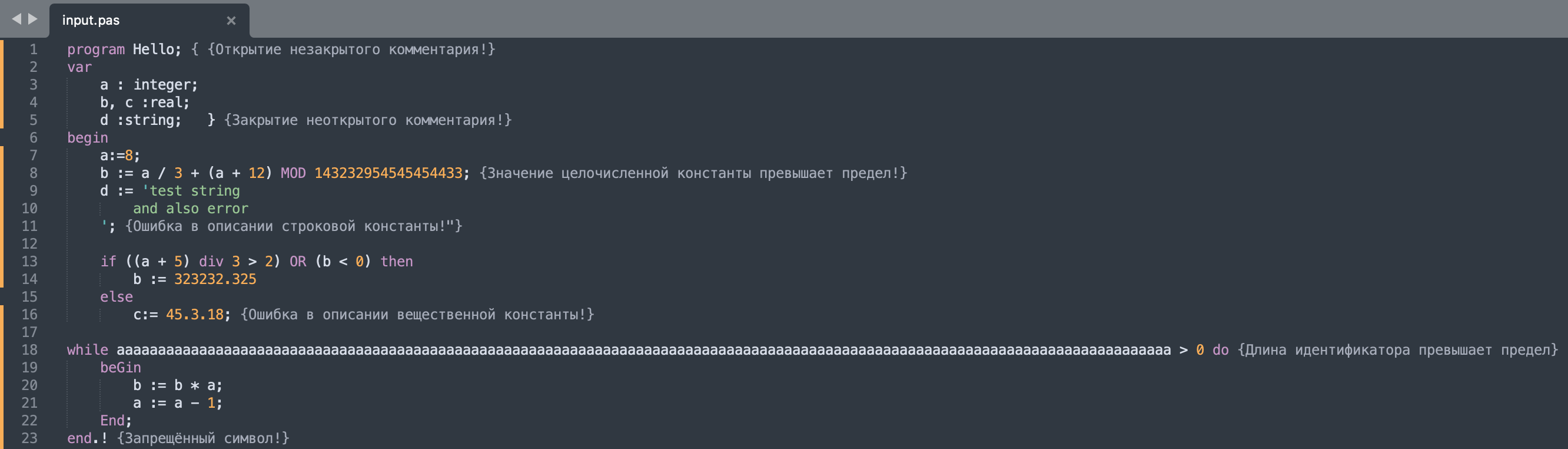


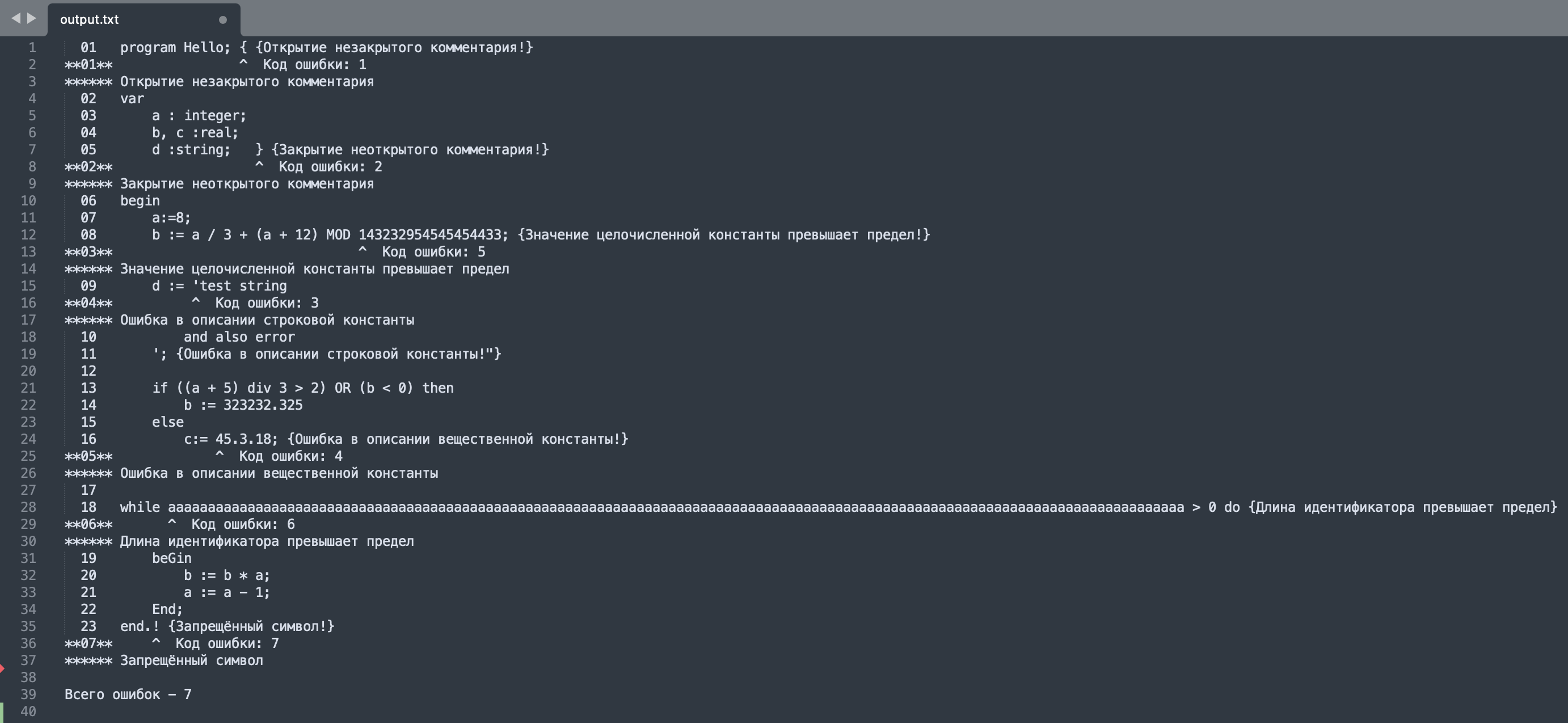
Результат вывода в консоль:





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

1. Теперь добавим в программу все описанные виды лексических ошибок:

Содержание выходного файла:

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

# Синтаксический и семантический анализ

## 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:

* 
* 
* 



* 
* 
* 
* 



* 
* 
* 



* 



* 



* 
* 
* 
* 
* 
*  
* 
* 

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.

## 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 анализатор 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. 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** необходимосоздать 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 идентификатор 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.

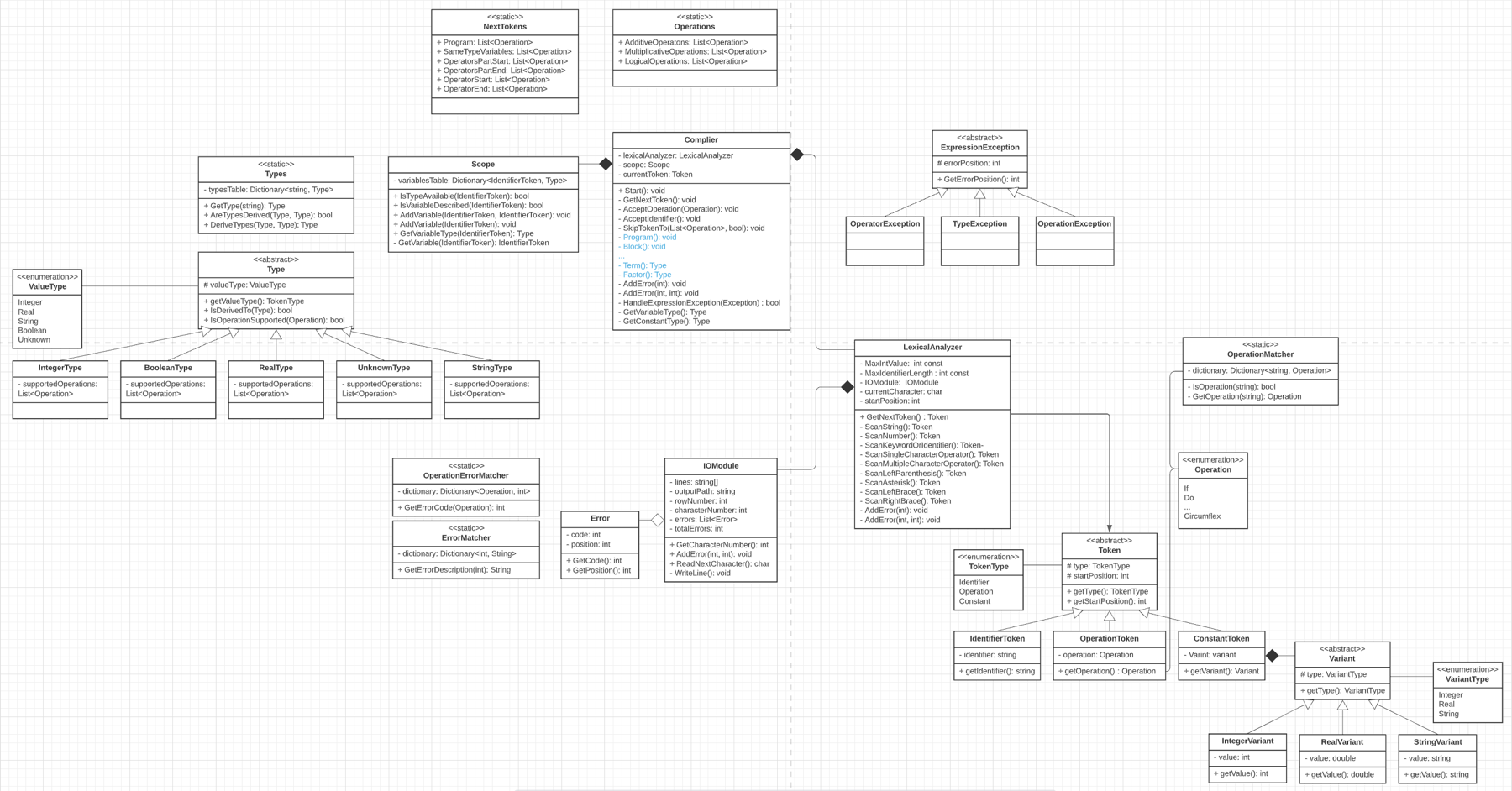
Для реализации семантического анализа также необходимо модифицировать методы, соответствующие БНФ. Например, в методе описания переменных добавлять переменные в поле **scope**, также проверять, не были ли переменные уже описаны, и если были добавлять ошибки, проверять тип на доступность и т.д.

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**

## 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 **SkitpTokensTo**, в качестве параметра которому передаётся один из списков в статическом классе **NextTokens**. Также один из параметров метода **SkitpTokensTo** булевский флажок. 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 закончится 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 вспомогетальные 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 **Faсtor** 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 типboolean, 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.

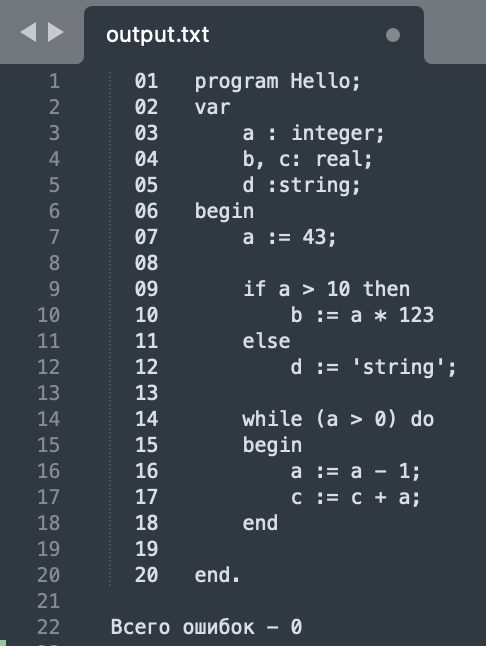
## Тестирование

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:

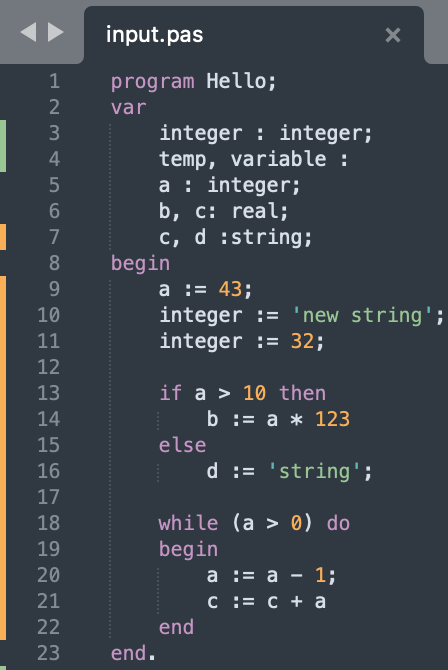


Содержимое выходного файла:

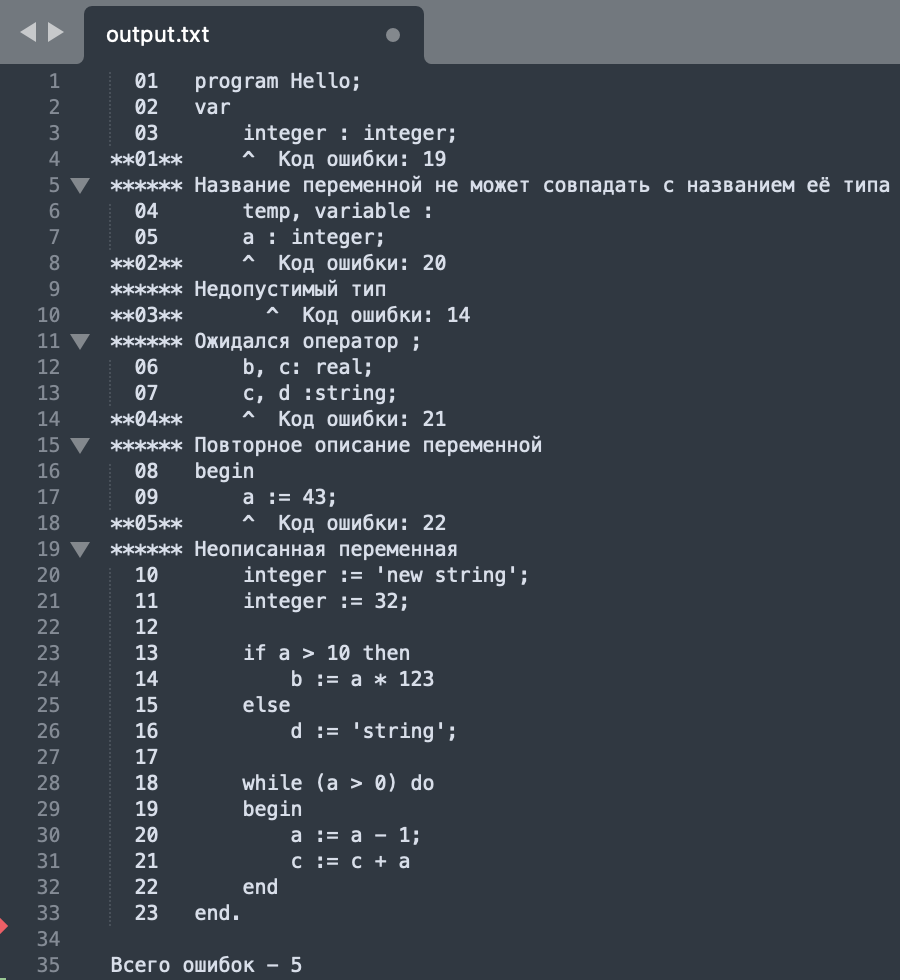


Как видно из выходного файла, компилятор успешно дообработал до конца программы, при этом не обнаружил ошибок там, где их нет!

1. 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:

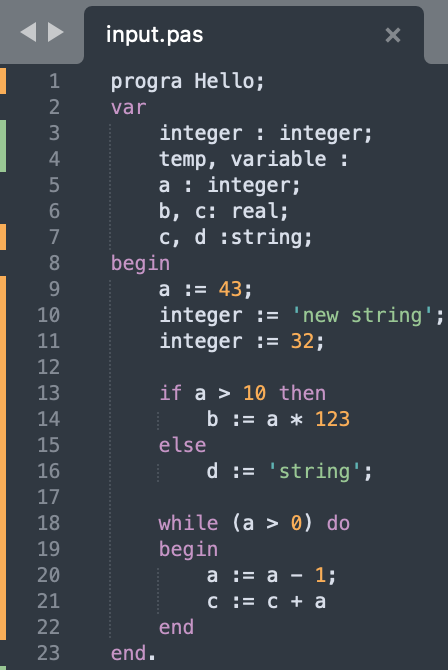


Содержимое выходного файла:

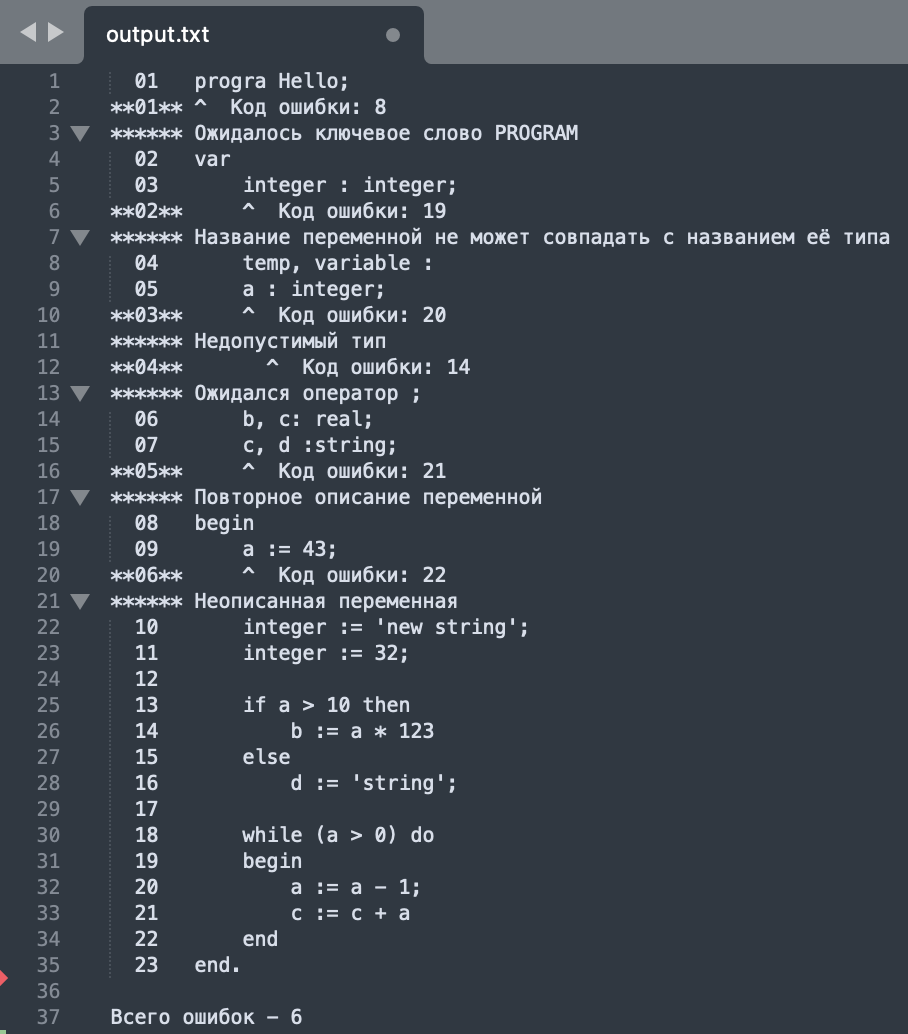


Как видно из выходного файла, все синтаксические и семантически ошибки успешно обнаружены и нейтрализованы. А именно на 3 строке переменная integer будет иметь неизвестный тип, т.к. при её описании использовался недопустимый тип. Далее, в 4-5 строках опять возникнет ошибка недопустимого типа, т.к. a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

1. Теперь вернёмся к первой строке программы и допустим опечатку в ключевом слове **Program**:

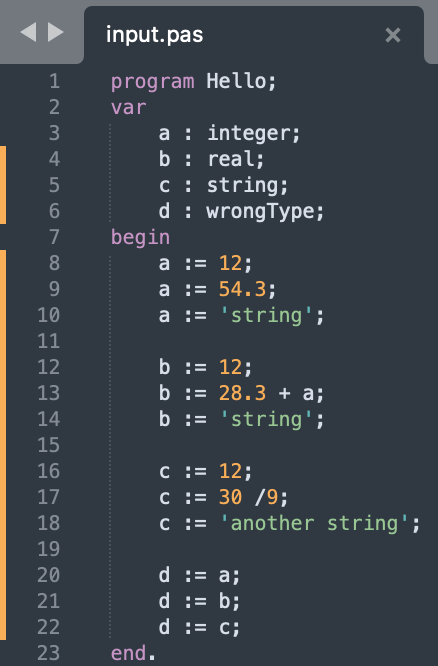


Содержимое выходного файла:

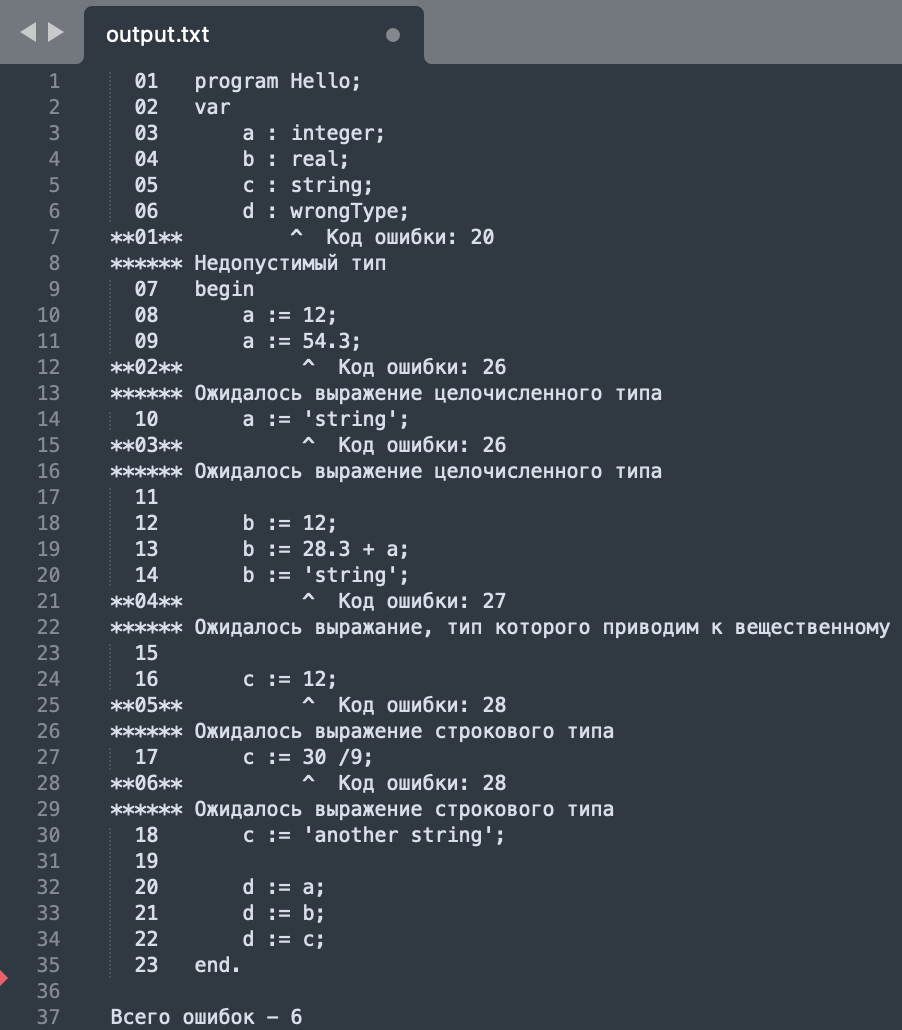


Можно заметить, что при нейтрализации синтаксической ошибки в первой строке токены были пропущены только до начала раздела переменных. Таким образом, было обнаружено максимальное количество ошибок.

1. Теперь проверим приводимость типов:

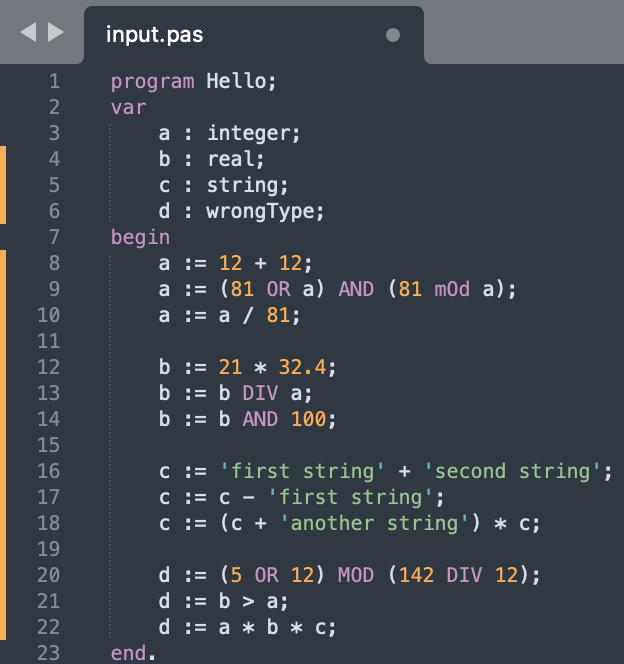


Содержимое выходного файла:

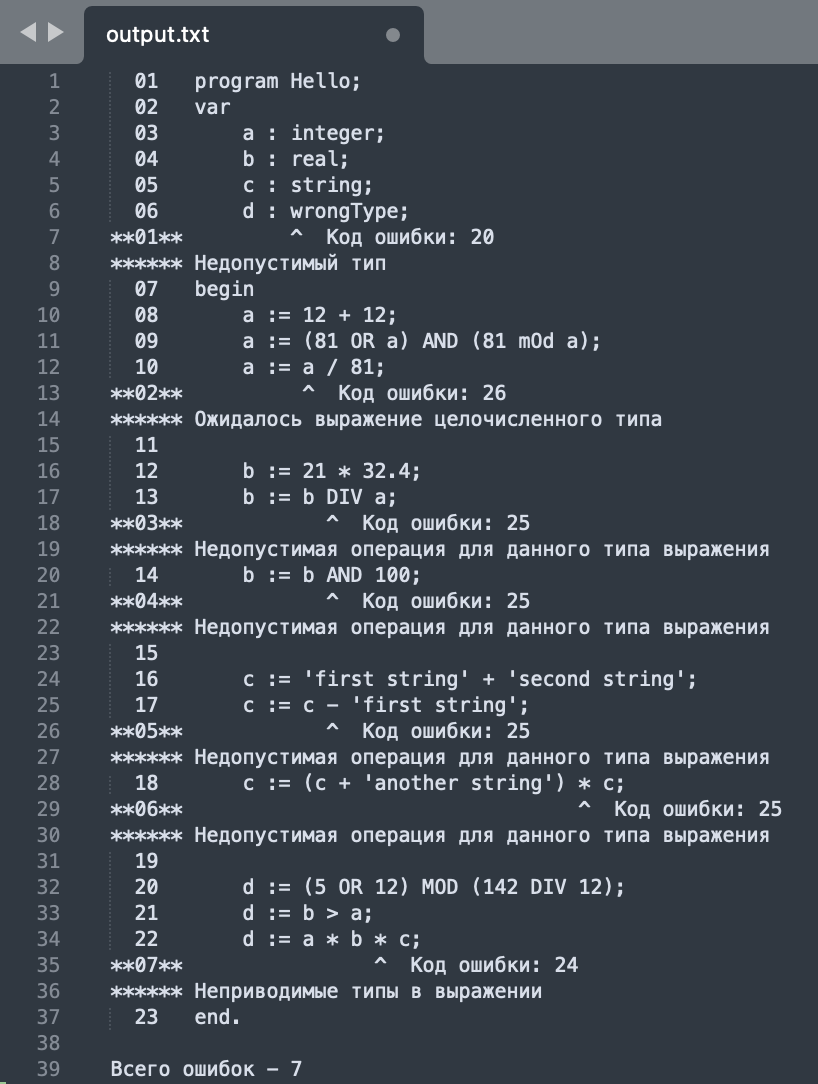


Как видно из сообщений об ошибках, переменной целочисленного типа можно присвоить только целочисленное значение, вещественной переменной – целочисленное или вещественное значение, а строковой переменной – только строку. Переменная d имеет неизвестный тип, поэтому она может принимать любые значения.

1. Теперь протестируем корректность операций, применяемых к разным типам данных:

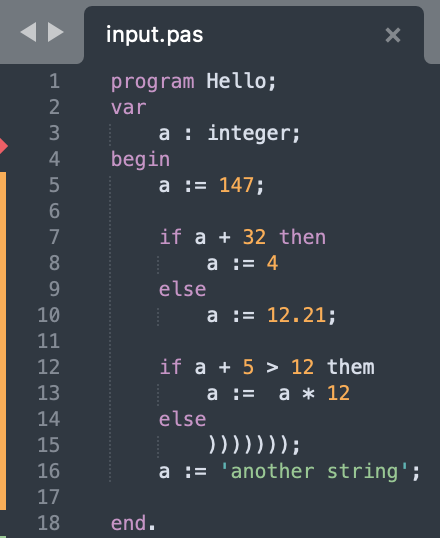


Содержимое выходного файла:

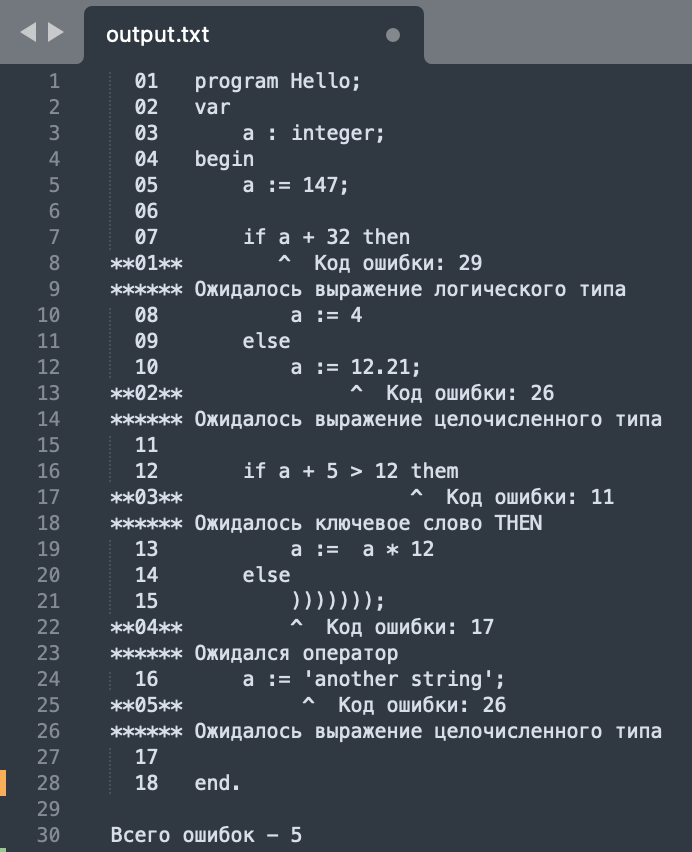


Как можно заметить из содержания файла, допустимость использования операций в зависимости от типов её операндов определяется корректно. К целочисленным операндам применимы любые операции, реализованные в заданном подмножестве языка. Для вещественных операндов применимы операции сложения, вычитания, умножения, 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.

1. Теперь проверим нейтрализацию синтаксических и семантических ошибок в условном операторе:

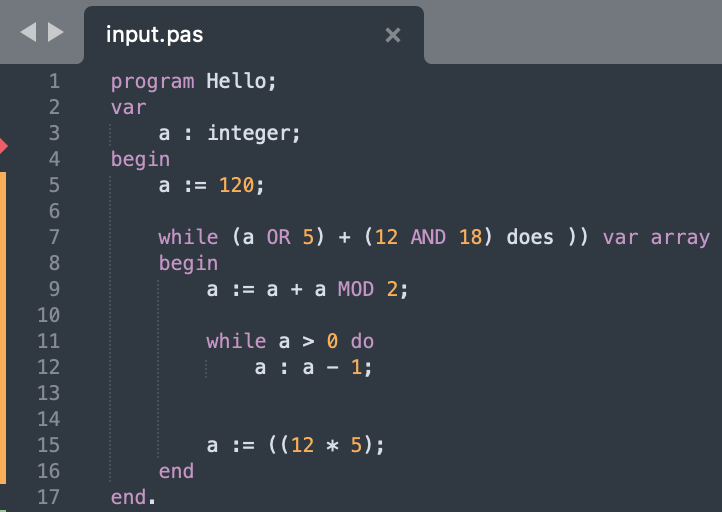


Содержимое выходного файла:

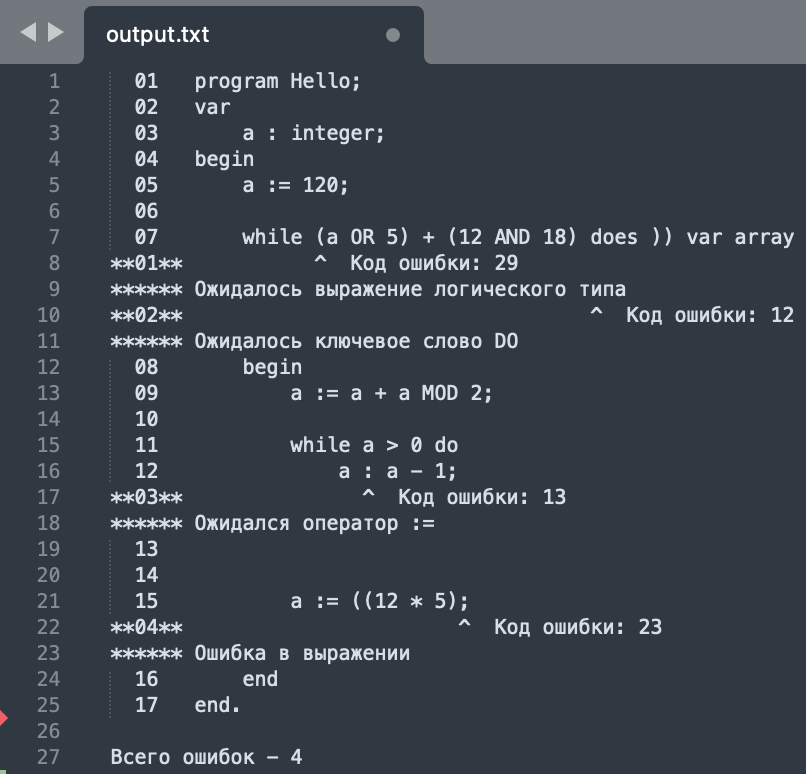


Как видно из сообщений об ошибках, синтаксические и семантические ошибки в условном операторе успешно находятся и нейтрализуются. В 7 строке была обнаружена семантическая ошибка, при её обнаружении нет необходимости пропускать какие-либо токены, так что компилятор сообщил об ошибке типа и продолжил работу 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.

1. Далее протестируем корректность работы оператора цикла:



Содержимое выходного файла:

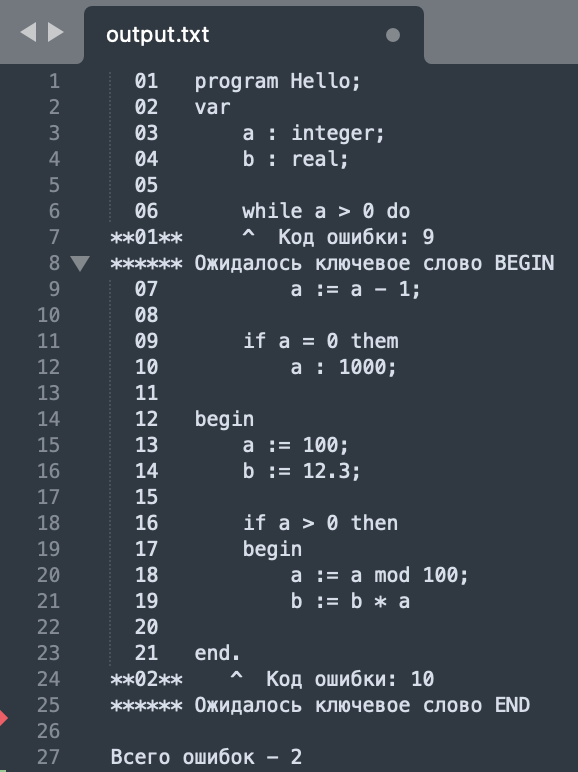


Как видно из сообщений об ошибках, все ошибки опять были корректно найдены и нейтрализованы. В 7 строке были найдены семантическая ошибка неправильного типа выражения, а также синтаксическая ошибка, в ходе нейтрализации которой были пропущены все токены до токена, с которого может начинаться оператор. 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.

1. Напоследок проверим корректность нейтрализации синтаксических ошибок в начале раздела операторов, а также в составном операторе:



Содержимое выходного файла:



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.