|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 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](#_Toc90134096)

[Модуль ввода-вывода 4](#_Toc90134097)

[1. Описание 4](#_Toc90134098)

[2. Проектирование 4](#_Toc90134099)

[3. Реализация 5](#_Toc90134100)

[4. Тестирование 7](#_Toc90134101)

[Лексический анализатор 9](#_Toc90134102)

[1. Описание 9](#_Toc90134103)

[2. Проектирование 10](#_Toc90134104)

[3. Реализация 11](#_Toc90134105)

[4. Тестирование 15](#_Toc90134106)

[Синтаксический и семантический анализ 20](#_Toc90134107)

[1. Описание 20](#_Toc90134108)

# 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.

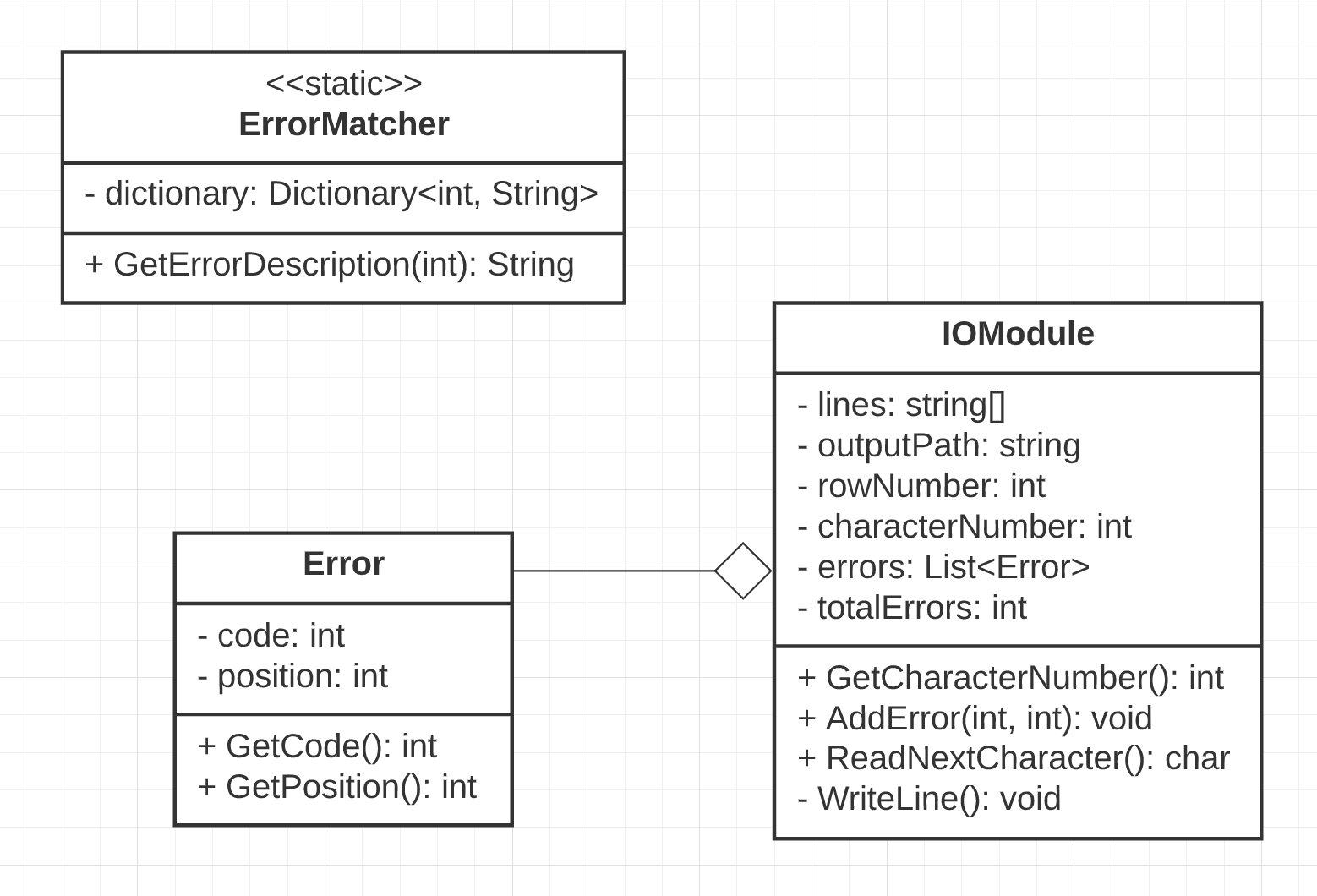
Помимо необходимой функциональности, я, в силу недостатка опыта проектирования, решил добавить в класс модуля ввода-вывода и работу с ошибками.

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.

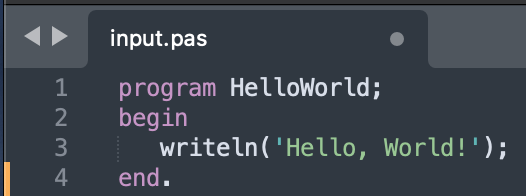
Как можно заметить, текущая строка с обнаруженными ошибками выводится при чтении первого символа следующей строки. Изначально я выводил эту информацию, когда доходил до последнего символа текущей строки. Далее, уже при тестировании лексического анализатора обнаружилась проблема такого подхода, заключающаяся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

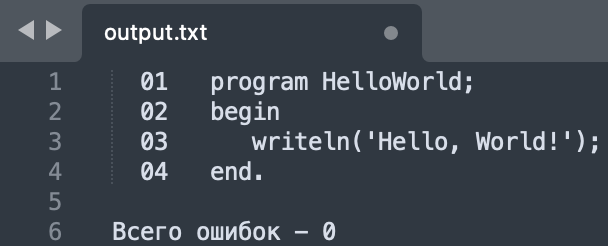
Метод **WriteLine** выводит в файл с листингом текущую строку программы с её номером. Далее, если список **errors** не пуст, увеличиваем счётчик ошибок **totalErrors** наколичество элементов в списке, после этого выводим все элементы, указывая коды и сообщения обнаруженных ошибок. Помимо этого, указывается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

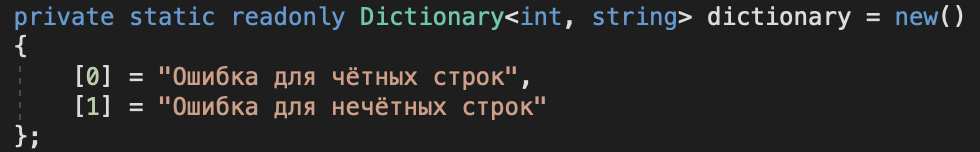
1. Для начала просто выведем содержимое входного файла в выходной на примере кода программы «Hello, World!».



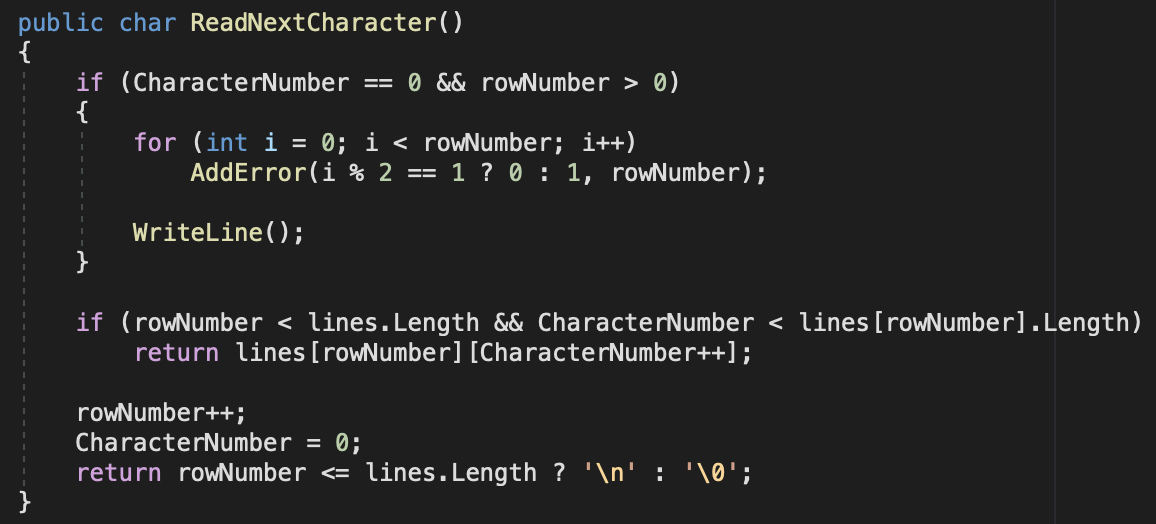


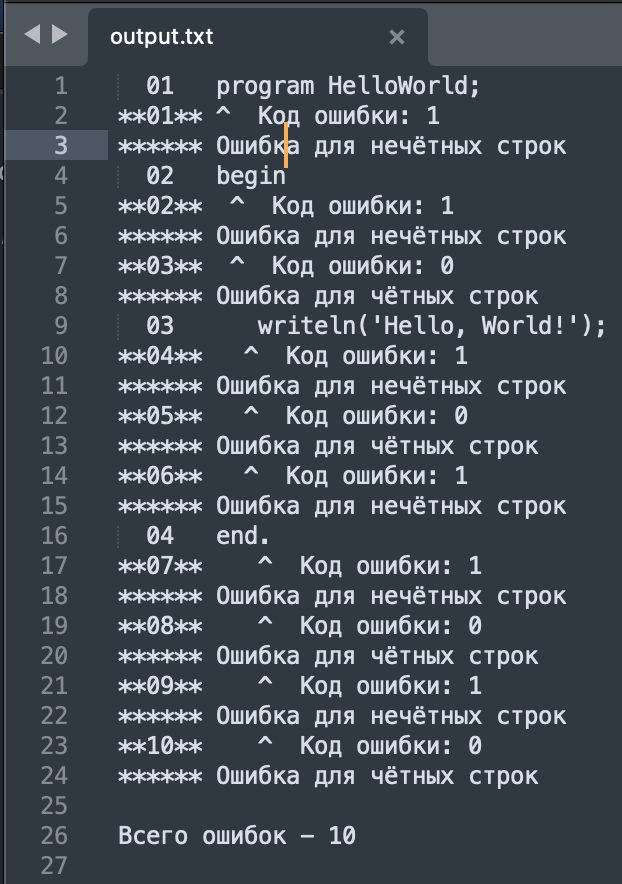
Текст исходной программы из исходного файла успешно был переписан в выходной. На удивление в нём не обнаружилось ни одной ошибки!

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.

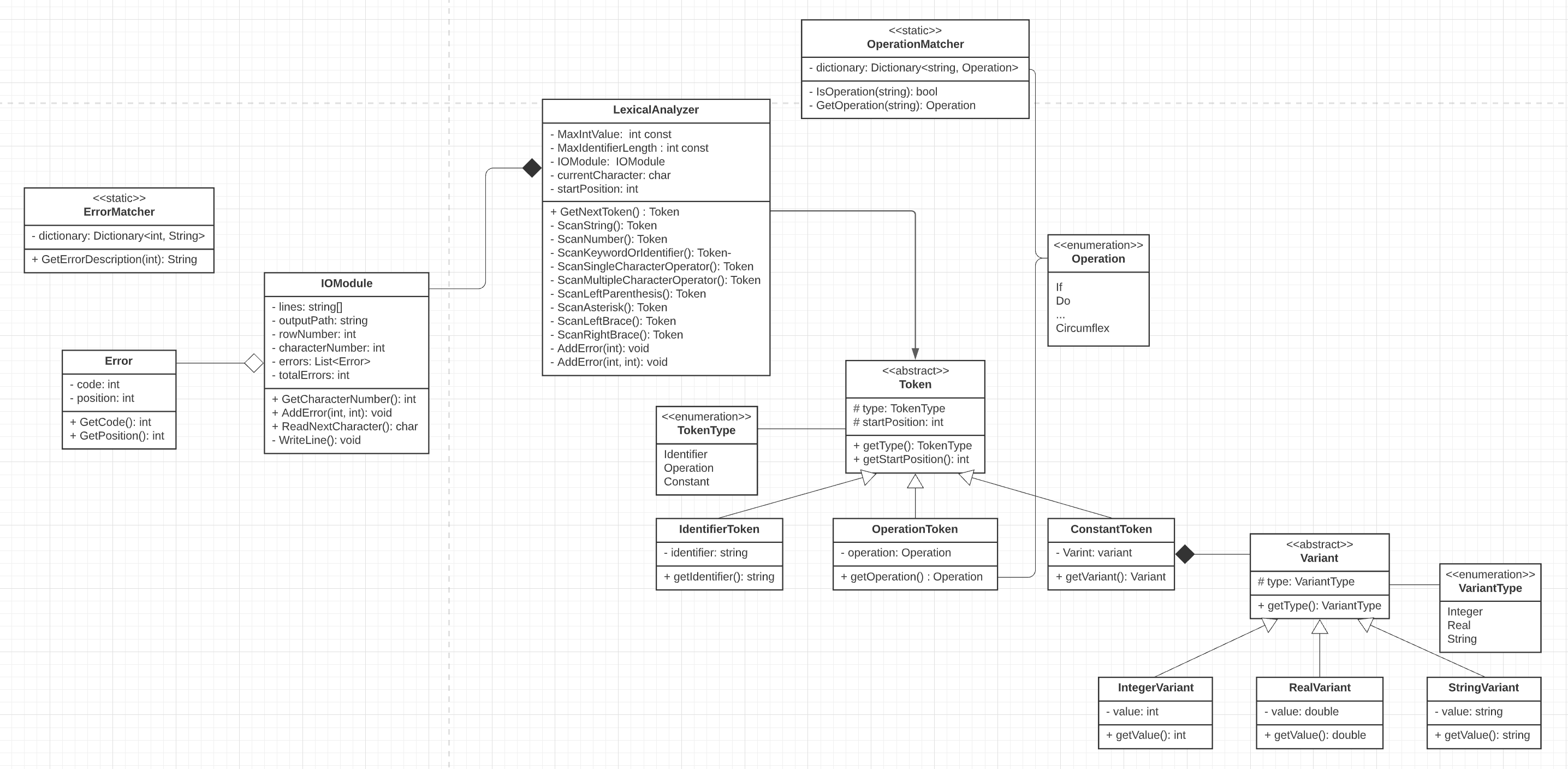
Для хранения значений констант в классе **ConstantToken** был создан абстрактный класс **Variant**.

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 «\*».

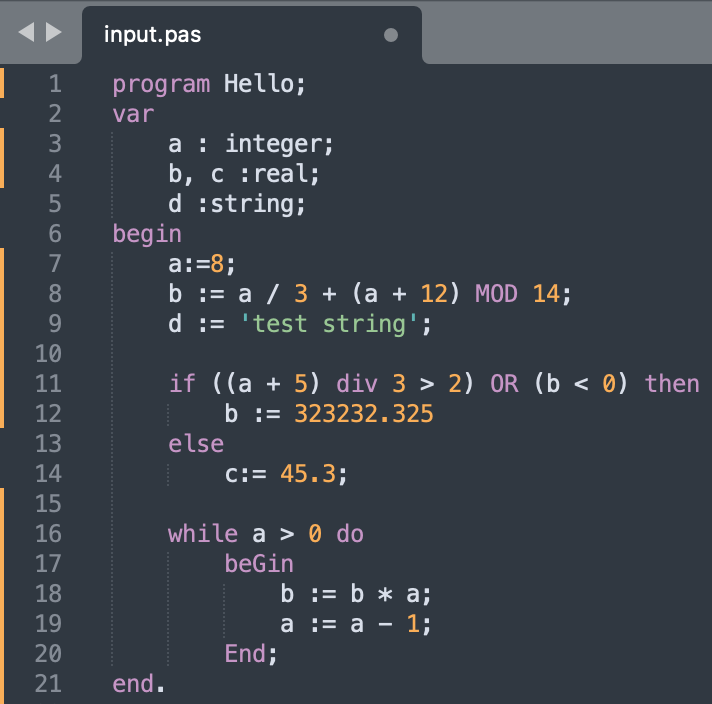
Стоит упомянуть, что во всех вышеописанных методах, в конструкторы классов, наследованных от **Token,** также передаётся позиция первой литеры этого токена в тексте исходной программы.

## 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:

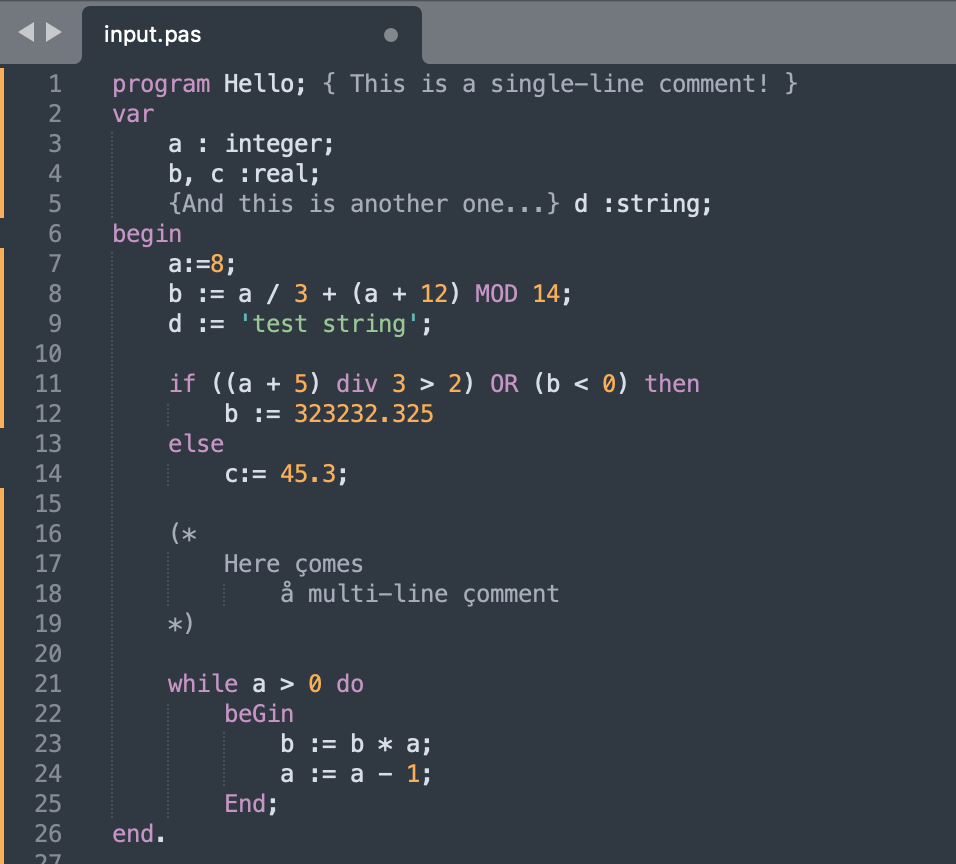




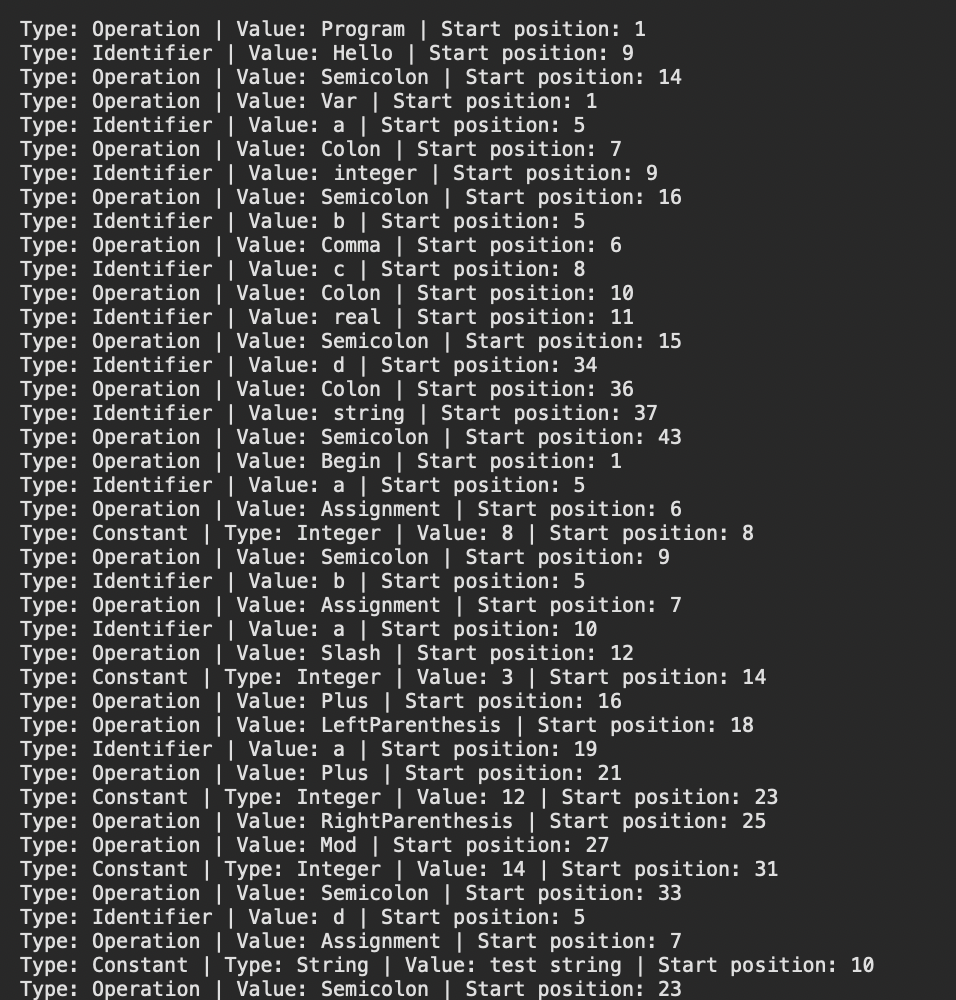
Как видно из результатов тестирования, лексический анализатор корректно определяет типы токенов, их значения, а также позицию первой литеры токена.

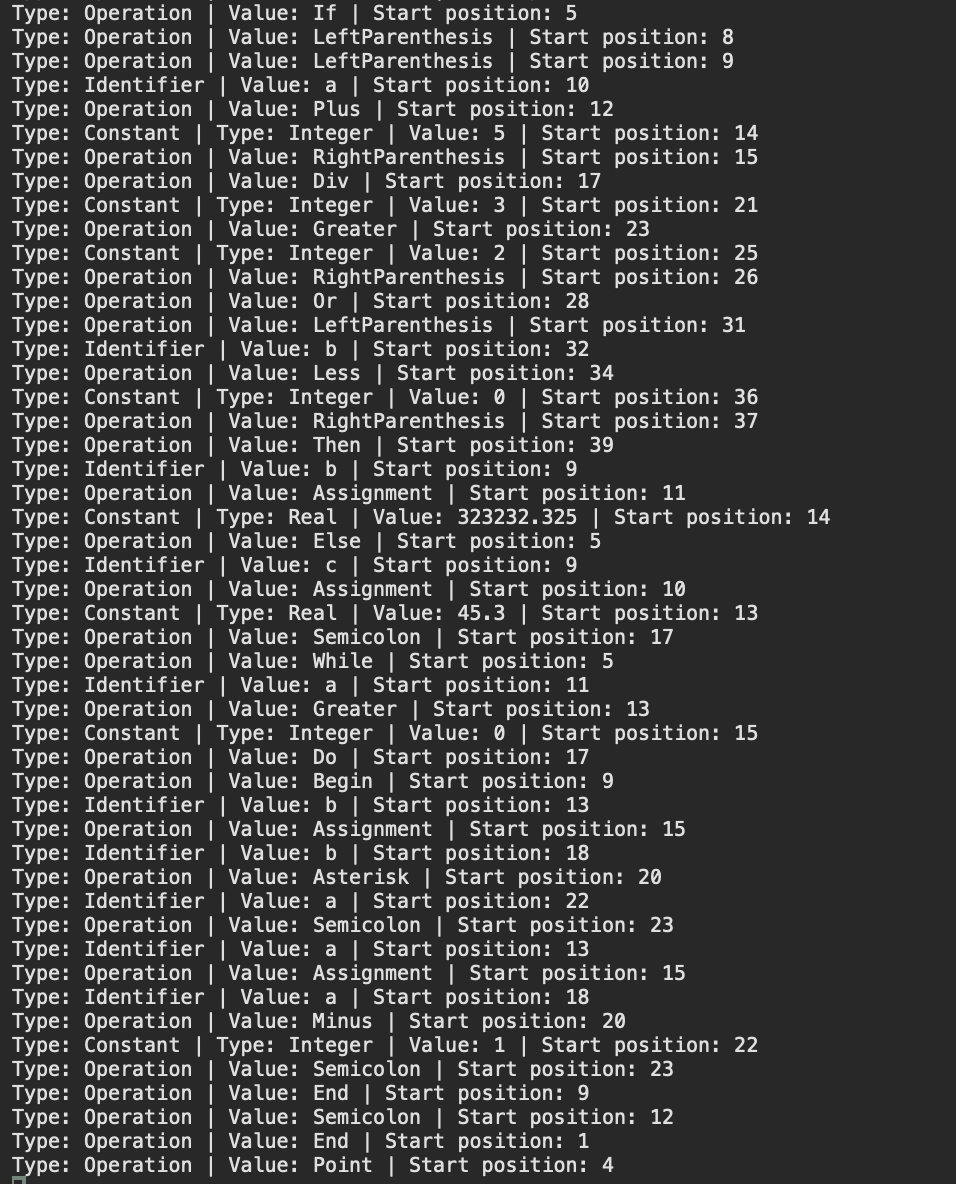
Хоть в этой программе используются и не все необходимые ключевые слова подмножества языка, дальнейшее тестирование можно пропустить в силу того, что токены строятся унифицировано в методах **ScanSingleCharacterOperator** и **ScanMultipleCharacterOperator**.

1. Добавим в программу однострочные и многострочные комментарии:



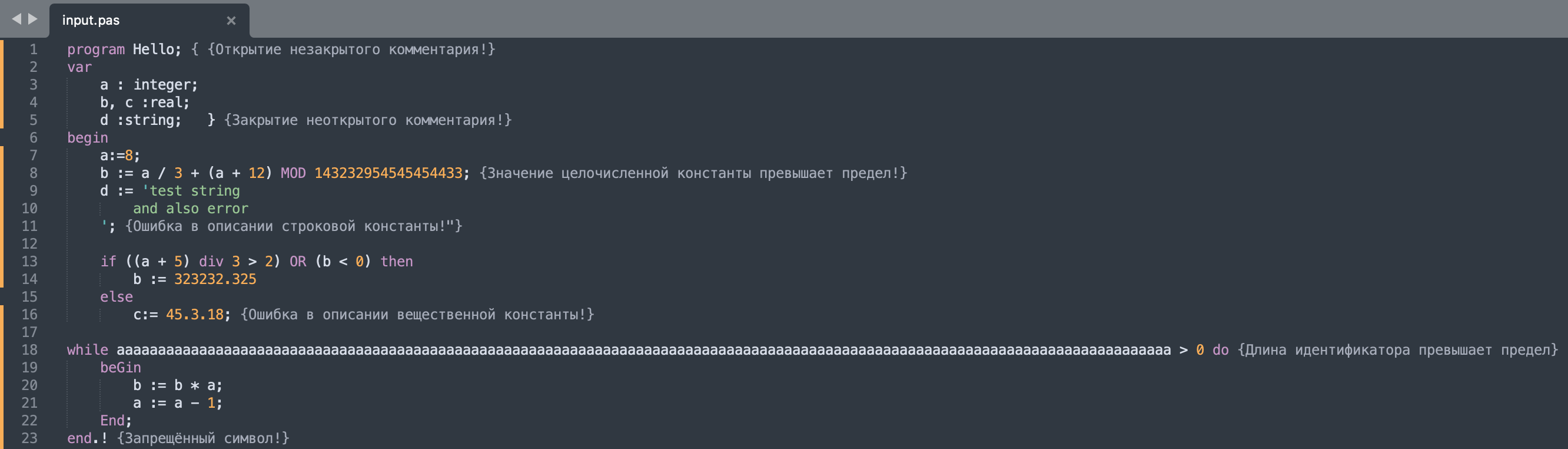
Результат вывода в консоль:



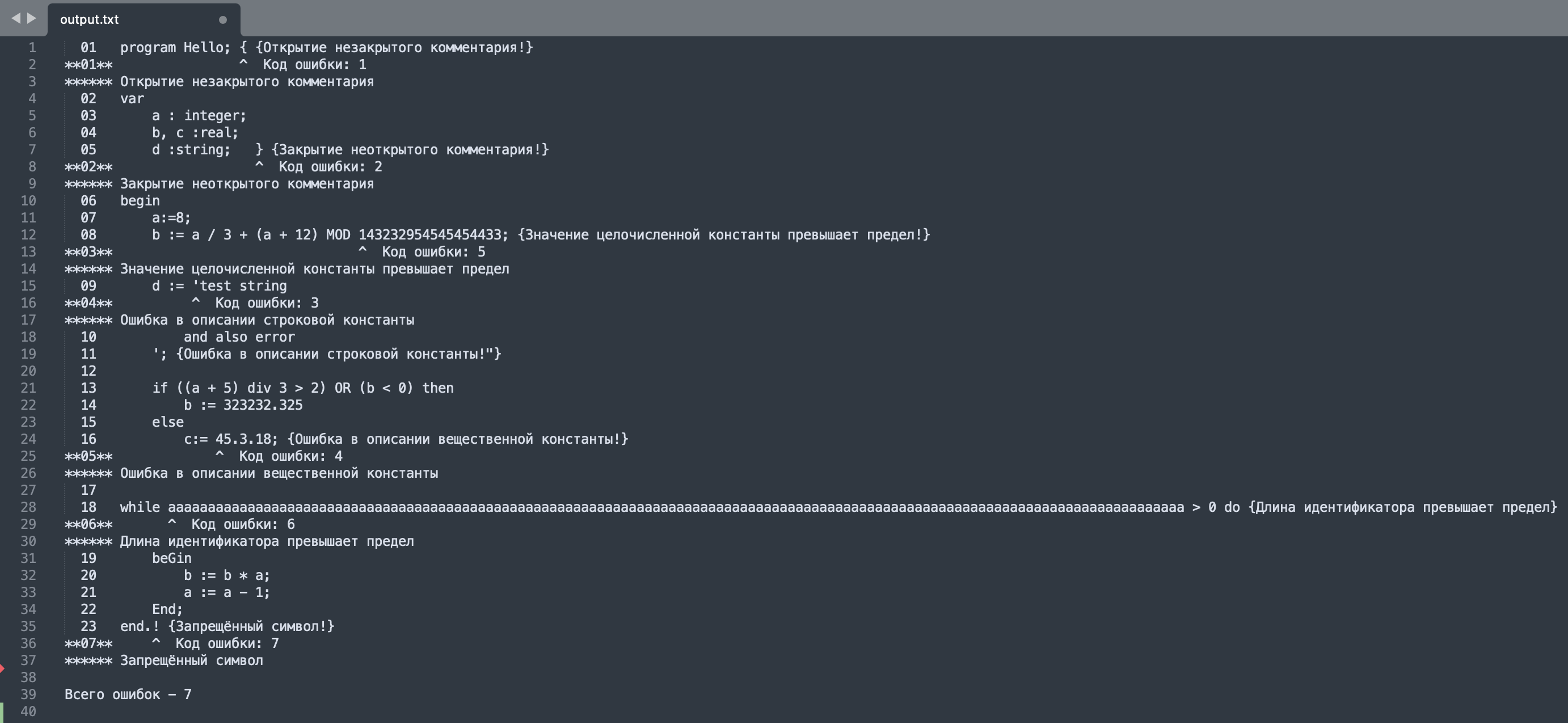


Как видно из результатов вывода в консоль, лексический анализатор корректно распознаёт и игнорирует как однострочные, так и многострочные комментарии.

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.

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

Для синтаксического анализа заданного подмножества языка Pascal необходимо реализовать следующие БНФ:

* <программа> ::= program <имя> ; <блок> .

# Реализация

# Тестирование