|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | Министерство науки и высшего образования РФ | | | | | | |  | | |
|  | | | | |  | | |  | | | | |
|  | | ФГБОУ ВПО «Пермский государственный национальный исследовательский университет» | | | | | | | | |  | |
|  | | | | |  | | |  | | | | |
|  | | | | Отчет  по индивидуальной работе «Создание Pascal компилятора» по дисциплине «Формальные грамматики и методы трансляции» | | | | |  | | | |
|  | | | | |  | | |  | | | | |
|  | Работу выполнил  студент гр. ПМИ-3, 3 курс  Власов А.А. \_\_\_\_\_\_\_\_\_\_\_\_  «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_2021 | | | | |  | Проверил  ассистент кафедры МОВС  Пономарев Ф. А. \_\_\_\_\_\_\_\_\_  «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_2021 | | | | |  |
|  | | | | |  | | |  | | | | |
|  | | | | | Пермь 2021 | | |  | | | | |

Содержание

[Модуль ввода-вывода 4](#_Toc90318758)

[Постановка задачи 4](#_Toc90318759)

[Проектирование 4](#_Toc90318760)

[Тесты 5](#_Toc90318761)

[Лексический анализатор 6](#_Toc90318762)

[Постановка задачи 6](#_Toc90318763)

[Проектирование 6](#_Toc90318764)

[Тесты 6](#_Toc90318765)

[Синтаксический анализатор 8](#_Toc90318766)

[Постановка задачи 8](#_Toc90318767)

[Проектирование 8](#_Toc90318768)

[БНФ 8](#_Toc90318769)

[Реализация 9](#_Toc90318770)

[Тестирование 10](#_Toc90318771)

[Минимальная допустимая программа: 10](#_Toc90318772)

[Раздел переменных: 10](#_Toc90318773)

[Раздел операторов, составной оператор: 11](#_Toc90318774)

[Оператор присваивания: 11](#_Toc90318775)

[Условный оператор: 12](#_Toc90318776)

[Цикл с предусловием: 12](#_Toc90318777)

[Выражение, оставшиеся операторы: 12](#_Toc90318778)

[Нейтрализация ошибок 13](#_Toc90318779)

[Семантический анализатор 14](#_Toc90318780)

[Постановка задачи 14](#_Toc90318781)

[Проектирование 14](#_Toc90318782)

[Таблица идентификаторов 14](#_Toc90318783)

[Проверка на соответствие типов 14](#_Toc90318784)

[Нейтрализация ошибок. 15](#_Toc90318785)

[Тесты 15](#_Toc90318786)

[Добавление writeln() 17](#_Toc90318787)

[Генерация кода 17](#_Toc90318788)

[Постановка задачи 17](#_Toc90318789)

[Проектирование 17](#_Toc90318790)

[Разбор выражения и присваивание 18](#_Toc90318791)

[Вывод в консоль, цикл, условный оператор 18](#_Toc90318792)

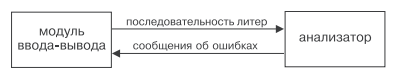
[Тесты 18](#_Toc90318793)

[P.S. Особенности моей написанной программы 19](#_Toc90318794)

# Модуль ввода-вывода

## Постановка задачи

Модуль ввода-вывода считывает последовательность литер исходной программы с внешнего устройства и передает их анализатору. Анализатор проверяет, удовлетворяет ли эта последовательность литер правилам описания языка, и формирует (в случае необходимости) сообщения об ошибках. Их взаимодействие можно описать в виде следующей схемы



## Проектирование

Исходный текст программы будем считывать по 1 строке из файла, далее из буфера считываются отдельные литеры, за это отвечает метод **getNextLetter**. Также модуль ввода-вывода должен принимать, обрабатывать и хранить информацию об ошибках, их местоположении относительно кода исходной программы. На рисунке 1 представлена схема класса **InputOutputModule**. Все ошибки будем хранить в виде списка, запоминая их позицию в тексте исходной программы. Сообщения об ошибках будем выводить в конце работы компилятора.

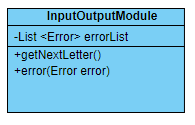


Рис.1

Для ошибки создадим отдельный класс Error (рисунок 2).

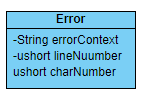


Рис.2

Также на основе WPF создадим простой UI компилятора (рисунок 3), для удобства дальнейшей работы и тестирования. При создании UI я вдохновлялся (копировал) программой PascalABC.NET 3.8.2.

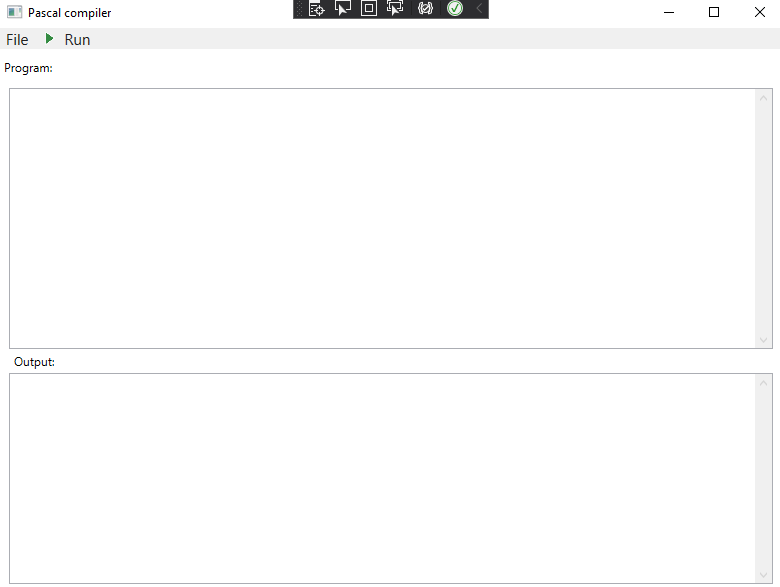
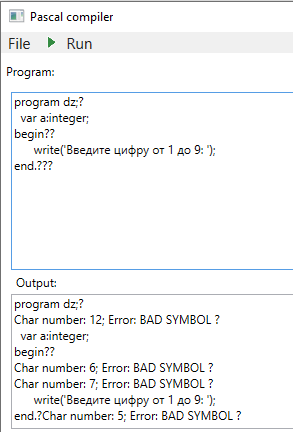
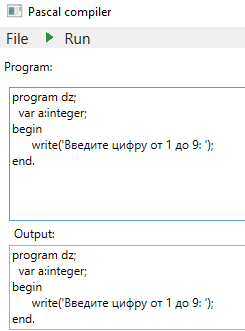


Рис.3

## Тесты

После написания программы нам необходимо проверить на работоспособность 2 основные функции getNextLetter и error. Для этого в окно результата будем выводить литеры, полученные с помощью getNextLetter, и в это же окно будем выводить информацию об ошибках в строке, т.к. модуль ввода вывода сам никак не может определить ошибки, то мы искусственно будем вызывать ошибки при обнаружении символа <?>. Результаты:

Вывод: модуль ввода-вывода разработан, работает корректно.

# Лексический анализатор

## Постановка задачи

Лексический анализатор формирует символы исходной программы и строит их внутреннее представление. Он получает от модуля ввода-вывода поток литер, формирует из них контекстные лексемы (токены, которые содержат тип лексемы). Далее (в зависимости от типа, полученного токена) обрабатывает и передаёт их в следующие модули компилятора.

## Проектирование

Мы выделим 3 типа токенов (Класс CToken), это: ttOper (операторы и ключевые слова), ttIdent (идентификаторы констант, типов, переменных, процедур и функций) и ttValue (значения констант).

ttOper будет хранить тип оператора или ключевого слова, которое он представляет.

ttIdent содержит в себе только имя пользовательского идентификатора.

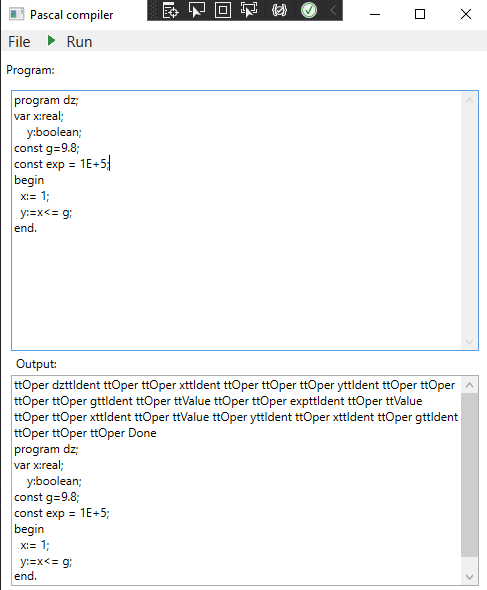
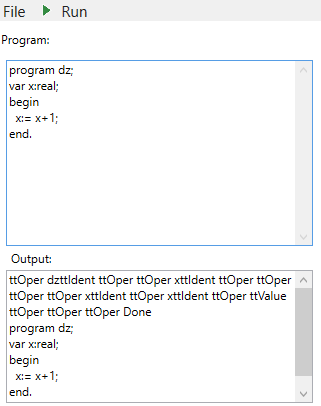
ttValue содержит значение константы, а также её тип (целое/вещественное/символ/строка).

Лексический анализатор также полностью исключает все комментарии, пробелы и прочие разделители, системные символы, а также определяет переполнения для целых/вещественных чисел. Сам же лексический анализатор устроен крайне просто, он запрашивает литеры у модуля ввода-вывода, и в зависимости от того что получил (через оператор switch), возвращает определённый токен (функция GetNextToken).

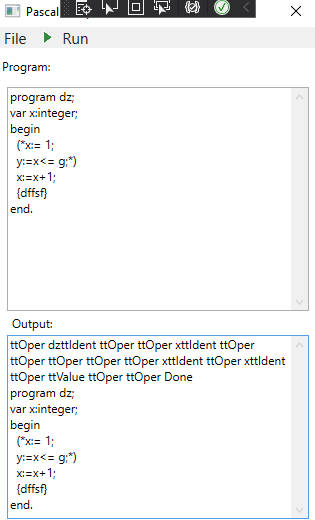
## Тесты

Если ошибок не обнаружено будем выводить тип получаемого токена, если есть ошибки, то исходный код с комментариями об ошибке.

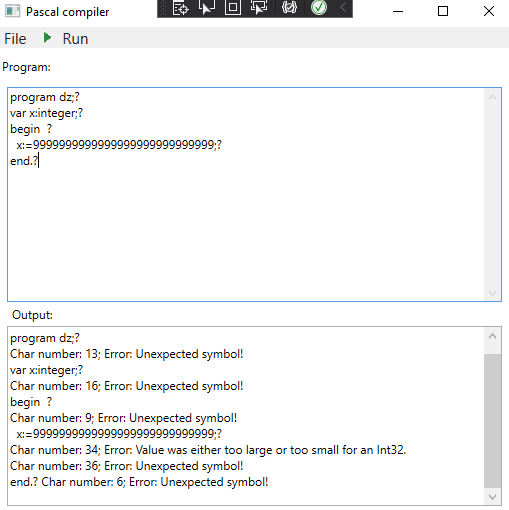
Нет ошибок:



Обработка комментариев, без них ровно 17 токенов:



Ошибки:



# Синтаксический анализатор

## Постановка задачи

Синтаксический анализатор проверяет соответствует ли написанный код установленному синтаксису языка программирования. Для задания синтаксиса широко применяются формальные правила формы Бэкуса—Наура, а также синтаксические диаграммы, между собой они отличаются лишь визуально.

## Проектирование

С технической точки зрения синтаксический анализатор рекурсивно вызывает функции, где каждая функция соответствует определённой БНФ. Начинаем с главной функции program. Ниже я приведу все БНФ, которые нам потребуется реализовать, согласно заданию, в виде функций.

### БНФ

<программа>::=program <имя>;<блок>.

<имя>::=<буква>{<буква>|<цифра>}

<блок>::=<раздел переменных><раздел операторов>

<константа>::=<число без знака>|<знак><число без знака>|<строка>

<число без знака>::=<целое без знака>|<вещественное без знака>

<целое без знака>::=<цифра>{<цифра>}

<вещественное без знака>::=<целое без знака>.<цифра>{<цифра>}|<целое без знака>.<цифра>{<цифра>}E<порядок>|<целое без знака>E<порядок>

<порядок>::=<целое без знака>|<знак><целое без знака>

<знак>::=+|-

<строка>::='<символ>{<символ>}'

<тип>::=<простой тип>

<простой тип>::=<имя типа>

<имя типа>::=<имя>

<раздел переменных>::= var <описание однотипных переменных>;{<описание однотипных переменных>;}|<пусто>

<описание однотипных переменных>::=<имя>{,<имя>}:<тип>

<раздел операторов>::=<составной оператор>

<оператор>::=<непомеченный оператор>

<непомеченный оператор>::=<простой оператор>|<сложный оператор>

<простой оператор>::=<оператор присваивания>|<пустой оператор>

<оператор присваивания>::=<переменная>:=<выражение>

<переменная>::=<полная переменная>

<полная переменная>::=<имя переменной>

<имя переменной>::=<имя>

<выражение>::=<простое выражение>[<операция отношения><простое выражение>]

<операция отношения>::==|<>|<|<=|>=|>

<простое выражение>::=<знак><слагаемое>{<аддитивная операция><слагаемое>}

<аддитивная операция>::=+|-|or

<слагаемое>::=<множитель>{<мультипликативная операция><множитель>}

<мультипликативная операция>::=\*|/|div|mod|and

<множитель>::=<переменная>|<константа без знака>|(<выражение>)|not <множитель>

<константа без знака>::=<число без знака>|<строка>

<пустой оператор>::=<пусто>

<пусто>::=

<сложный оператор>::=<составной оператор>|<выбирающий оператор>|<оператор цикла>

<составной оператор>::= begin <оператор>{;<оператор>} end

<выбирающий оператор>::=<условный оператор>

<условный оператор>::= if <выражение> then <оператор>[else <оператор>]

<оператор цикла>::=<цикл с предусловием>

<цикл с предусловием>::= while <выражение> do <оператор>

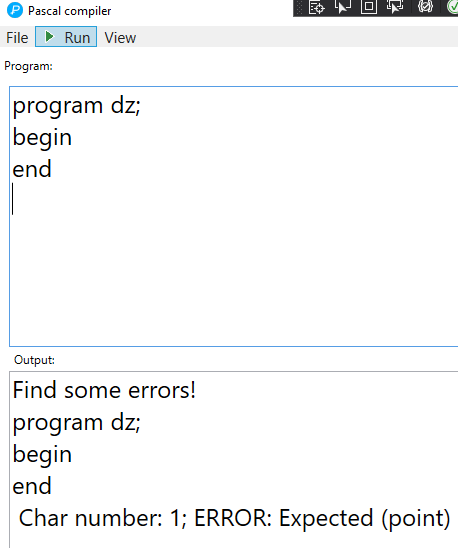
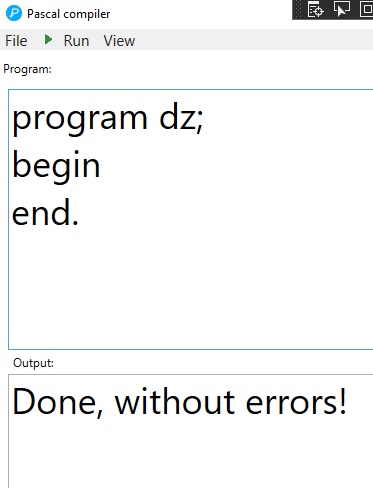
## Реализация

Реализуем функцию accept(expected token), которая будет считывать следующий токен и проверять является ли он токен expected token' ом, в случае несоответствия выдаём ошибку. Все БНФ реализуем в прямом их представлении, скобочки ({}, []) означают что конструкция может быть сколь угодно раз, или не быть вообще. В некоторых БНФ присутствует знак |, он означает <или>. Чтобы понять какую конструкцию мы будем считывать дальше мы считаем следующий токен, с его помощью мы однозначно можем определить тип следующей синтаксической конструкции и приступить к анализу. В результате мы начинаем с БНФ program, и далее, последовательно, в зависимости от написанного кода, “ацептим” допустимые синтаксические конструкции, в противном случаем выдаём ошибку.

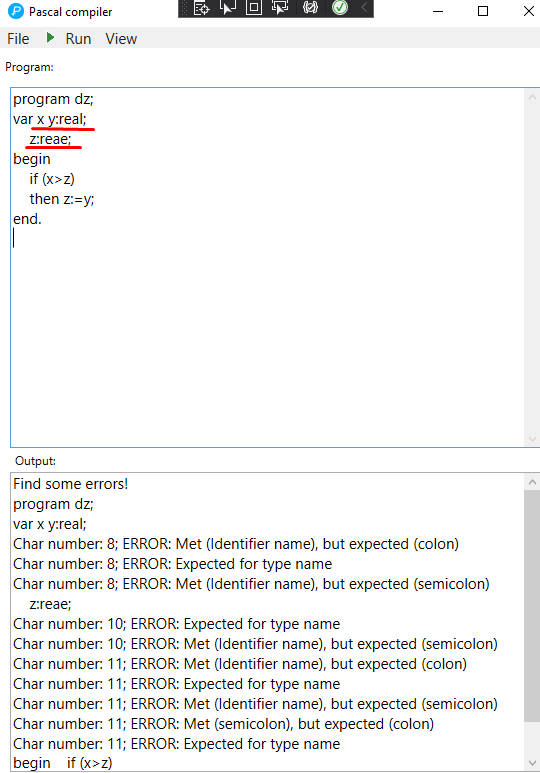
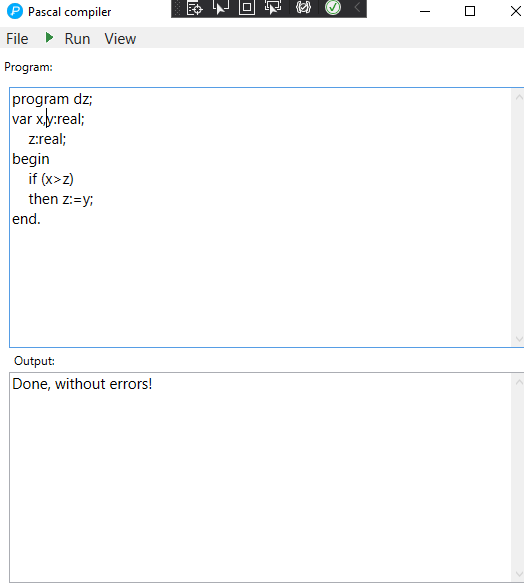
## Тестирование

Постараемся протестировать каждый блок по-отдельности:

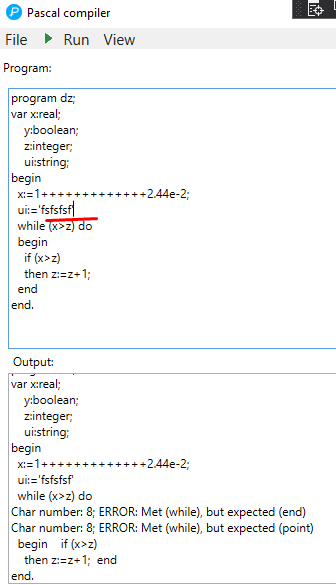
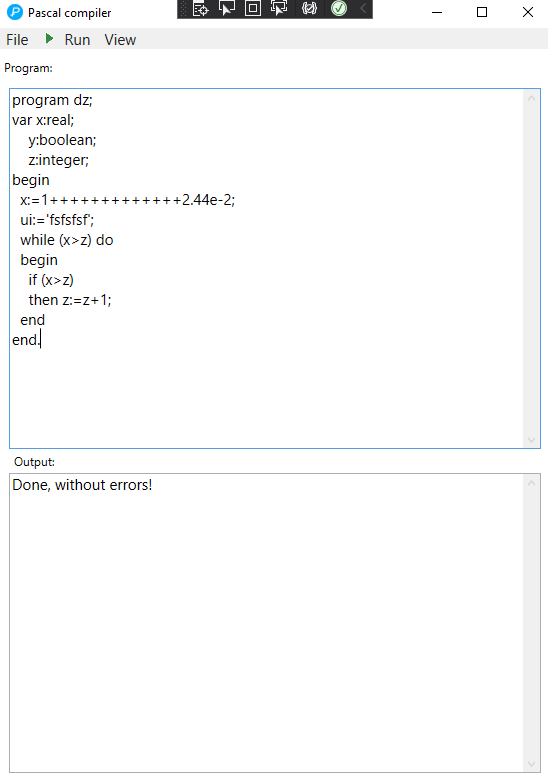
### Минимальная допустимая программа:



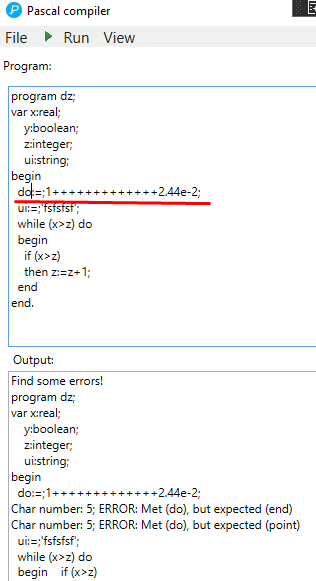
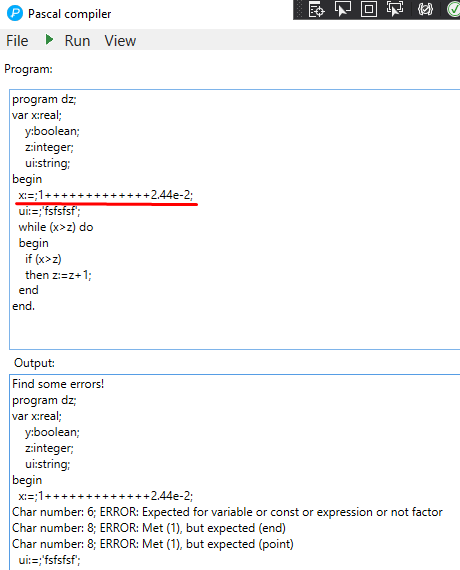
### Раздел переменных:



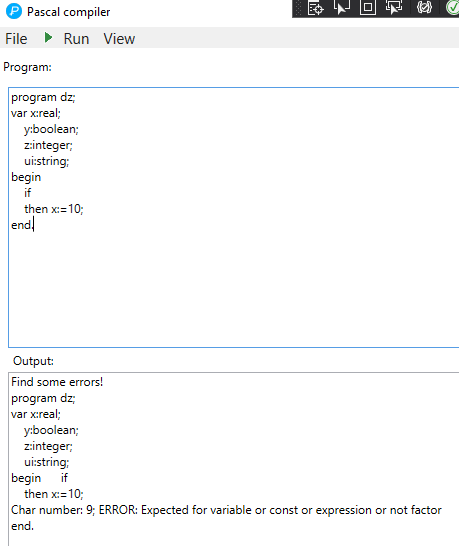
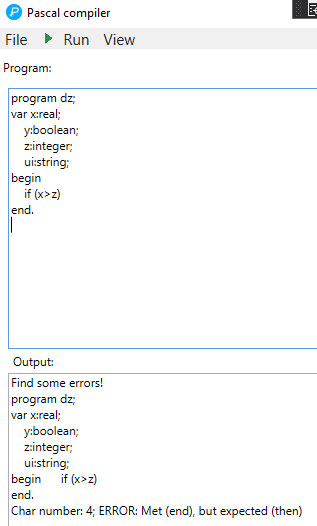
### Раздел операторов, составной оператор:



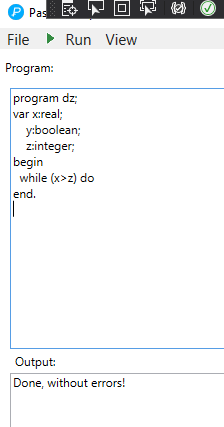
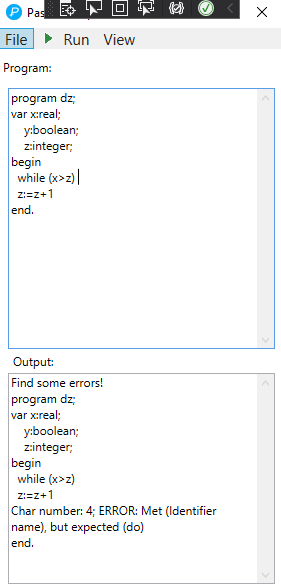
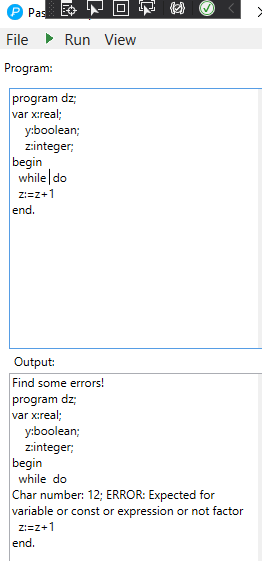
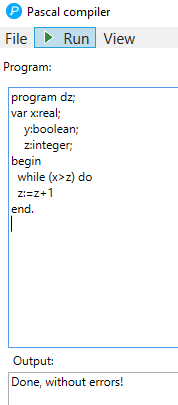
### Оператор присваивания:



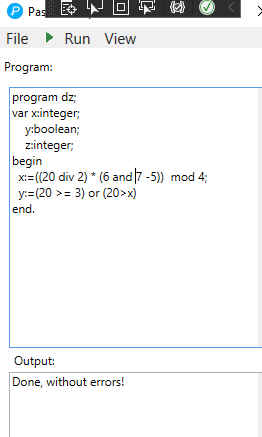
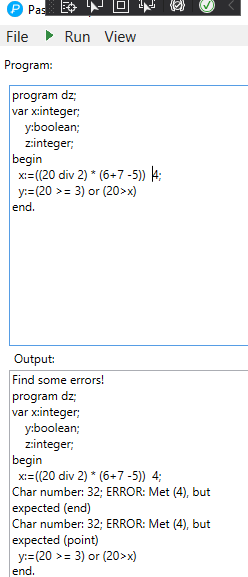
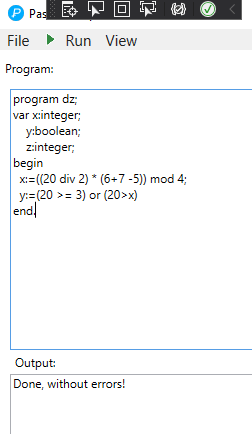
### Условный оператор:



### Цикл с предусловием:

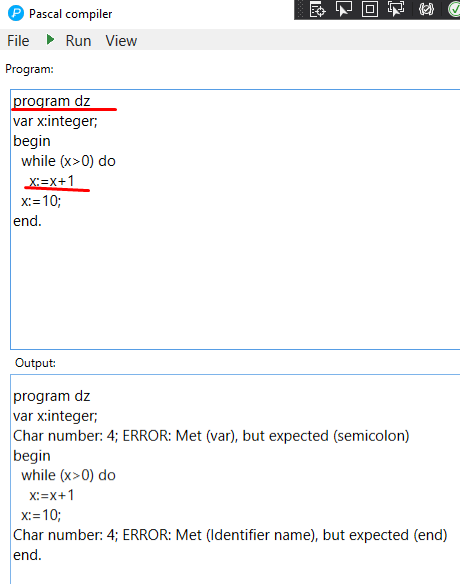
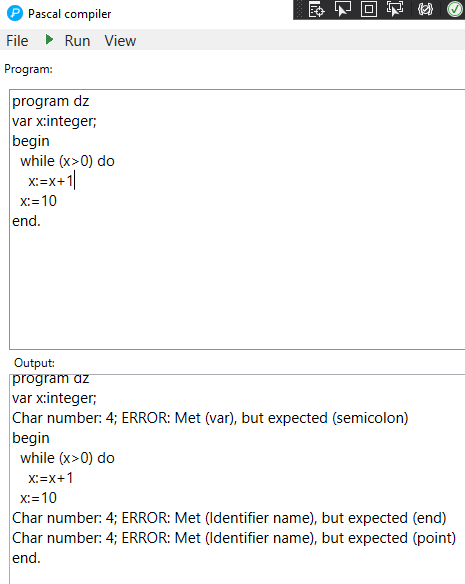


### Выражение, оставшиеся операторы:



## Нейтрализация ошибок

Чтобы обнаружить максимально кол-во синтаксических ошибок за один проход компилятора, разработаем нейтрализацию синтаксических ошибок. Я выбрал следующий принцип: при обнаружении синтаксической ошибки, мы будем пропускать все токены, пока не встретим ближайшее ключевое слово (program, var, begin, if, else, then, end, while, do). Чтобы наведённых ошибок было минимально кол-во, токены, при операции пропуска, будем обрабатывать согласно структурам БНФ, но не будем выдавать ошибку, если она присутствует (а она будет присутствовать). Как только будет встречено ключевое слово, синтаксический анализатор продолжает искать ошибки в штатном режиме. Ниже результат до/после.



# Семантический анализатор

## Постановка задачи

Семантический анализатор проверяет соответствует ли написанная программа неформальным правилам, которые невозможно описать и проверить с помощь БНФ. Он обнаруживает следующие типы ошибок:

1. Использование неописанного идентификатора
2. Повторно описанный идентификатор
3. Несоответствие типов
4. Неправильный способ использования
5. Ошибки с параметрами функций, процедур, индексов массивов, указателей и т.д.

Нам же, ввиду общего дополнительного задания, искать придётся только первые 3 типа ошибок. Также из-за задания у нас будет одна единая область действия, а идентификаторы могут быть только переменные.

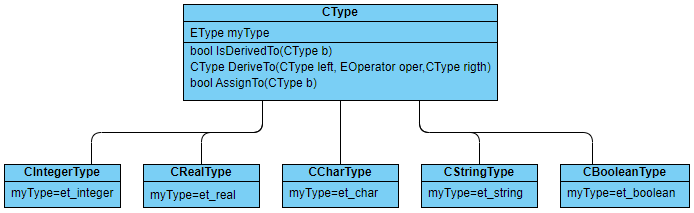
## Проектирование

### Таблица идентификаторов

В разделе описания переменных нам нужно будет создать и заполнить таблицу идентификаторов, в которой мы будем хранить имя идентификатора и его тип. Для этого будем использовать структуру Dictionary, где ключом будет имя идентификатора, а значением его тип идентификатора. При описании новых переменных будем последовательно добавлять их в таблицу, выдавая ошибку при повторном описании идентификатора. При прохождении по БНФ, каждый раз при обнаружении переменной будем сверяться с таблицей, чтобы проверить, описана ли данная переменная.

### Проверка на соответствие типов

В языке программирования тип – это множество значений и операций над этими значениями. Поэтому для добавления стандартных типов мы должны определить какие значения и операции они поддерживают. Разработаем следующее дерево классов:



Для каждого типа определим свой набор доступных операций, а также результат выполнения данных операций. Также изменим следующие БНФ:

1) <переменная>

2) <выражение>

3) <простое выражение>

4) <слагаемое>

5) <множитель>

Теперь они в качества результата возвращают CType ожидаемого значения, или же null, если допущена синтаксическая или семантическая ошибка. На основе этих результатов мы будем проверять соответствие типов. Для следующих операция существую ограничения:

1) сложение определено только int\real + int\real или char\string + char\string

2) разность, умножение, деление определены только для int\real

3) операции div, mod только для int

4) операции not, and, or только для bool

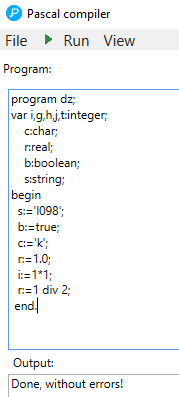
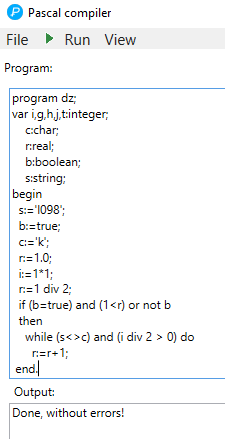
Помимо этого, нужно проверять соответствие типов при встрече оператора присваивания. В конструкциях if, while условие определяется выражением, значение которого должно быть типа bool.

## Нейтрализация ошибок.

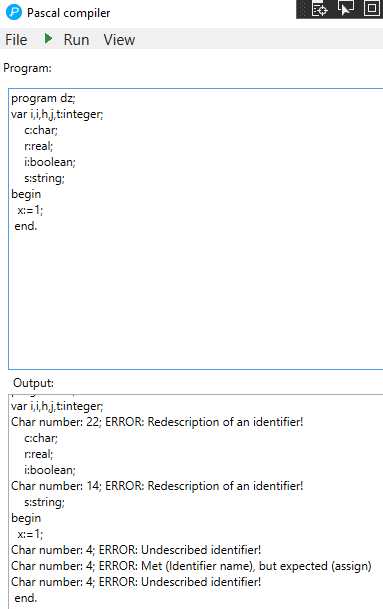
Т.к. из-за семантических ошибок не нарушается работа БНФ, мы продолжаем просматривать их и обнаруживать следующие ошибки. Количество наведённых ошибок при этом будет около нулевым.

## Тесты

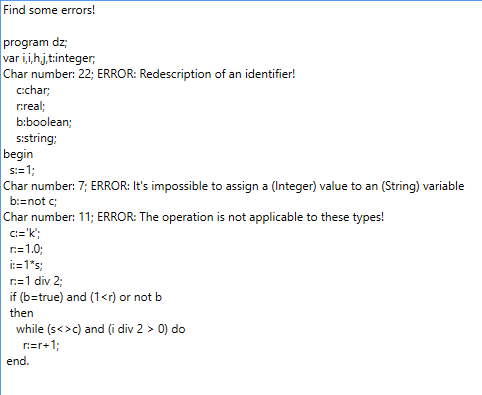
Без ошибок:

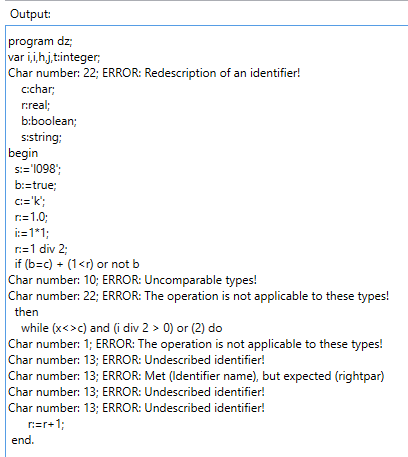
Неописанные и повторно описанные переменные:



Несоответствие типов:



Операторы if, while:



# Добавление writeln()

Для добавления оператора writeln добавим новый тип доступного оператора в EOperator, добавим новый тип доступного токена в лексический анализатор и изменим следующие БНФ:

1. <оператор вывода>::= **writeln** (<переменная>)
2. <простой оператор>::=<оператор присваивания>|<пустой оператор>|<оператор вывода>

**Генерация кода**

Постановка задачи

Мы будем делать транслятор из языка Pascal в IL коды для .NET. Инструкции языка Pascal напрямую транслируем в IL код.

Проектирование

Сама трансляция очень похожа на создание обратной польской записи используя стек. Сначала мы создаём динамическую сборку, куда помещаем динамические модули, типы и методы. В рамках нашего задания нам хватит всего 1 метода. После записи всех инструкций (IL кодов) сохраняем динамическую сборку в dll файл на диск. Основные задачи: правильный разбор выражения, присваивание, консольный вывод, цикл с предусловием, условный оператор.

Разбор выражения и присваивание

Мы разбираем выражение ещё на этапе синтаксического анализа, когда пробегаем по БНФ, но нужно лишь добавить вывод констант, переменных и операций в динамическую сборку.

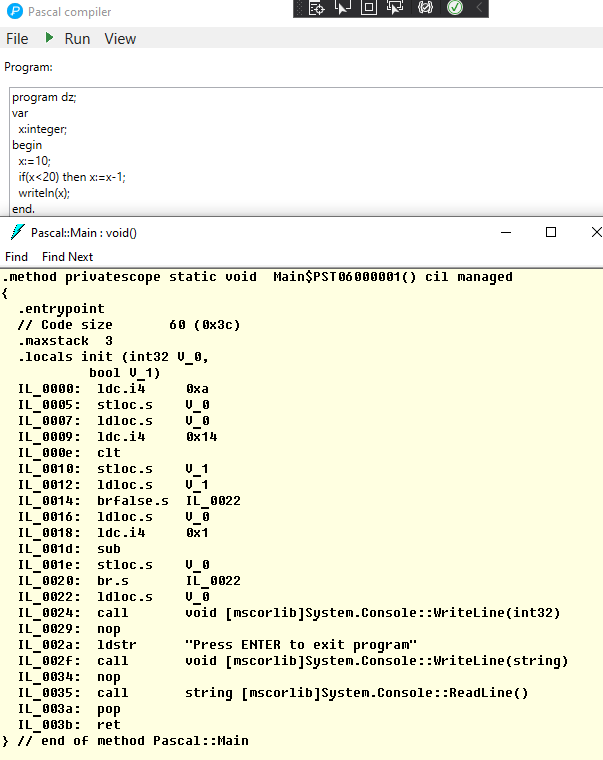
Перед присваиванием мы должны объявить локальные переменные, поочерёдно добавляем все используя таблицу идентификаторов.

Вывод в консоль, цикл, условный оператор

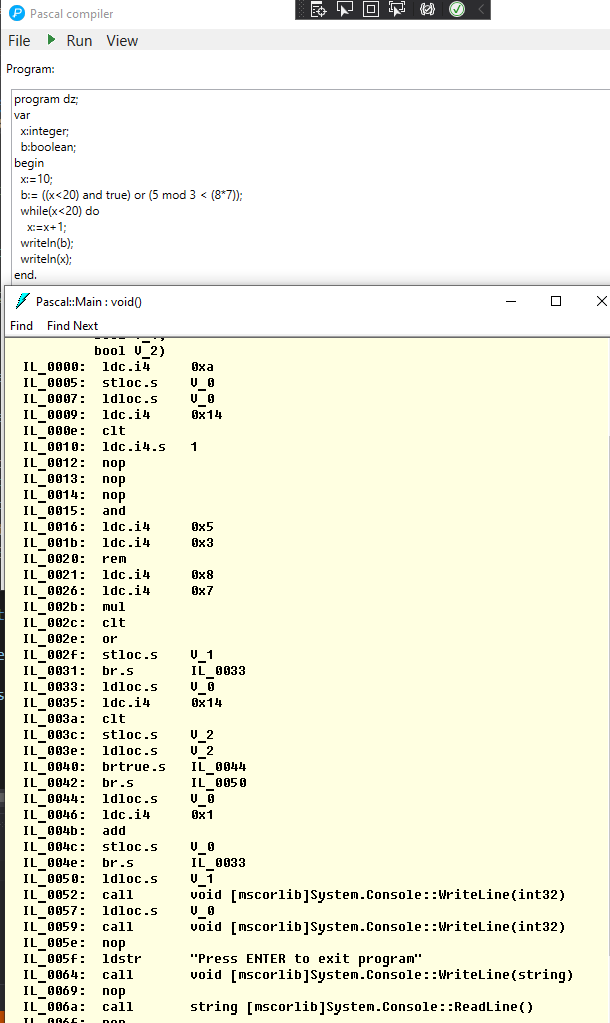
Вывод реализуется так, что мы помещаем на стек необходимую переменную и для неё вызываем метод writeline. Цикл и условный оператор реализуются похоже, с помощью метки label мы помечаем инструкции, которые будем выполнять исходя из условия, как только условие выполняется мы сразу же перепрыгиваем на следующие инструкции.

## Тесты

If, writeln:



While,разбор выражений:



# P.S. Особенности моей написанной программы

Версия: .NET FrameWork 4.8. Программа представляет WPF приложение с понятным интерфейсом. На верхней панели активна только кнопка Run, чтобы запустить компиляцию, и кнопка View что увеличить\уменьшить масштаб написанного кода. Если после компиляции не возникнет ошибок, то программа создаст dll файл с IL кодами, в окошке output будет показан путь до этого файла. Если же возникнут ошибки, то dll файл будет создан и удалён, в окошке output будет выведен листинг со всеми обнаруженными ошибками в программе. Нужный путь для создания dll файла вручную устанавливается в классе MaimWindow.xaml.cs(да, знаю, косяк). После работы dll файл можно посмотреть с помощью ildasm.exe, а также с помощью функции dump из ildasm.exe создать .il файл, который уже запустить через ilasm.exe чтобы получить готовый исполняемый exe. Для удобства, после трансляции и исполнения pascal кода, в exe я добавил ожидание нажатия enter, чтобы можно было увидеть результат работы. Источники, которыми я пользовался:

https://docs.microsoft.com/ru-ru/dotnet/

<http://www.cyberguru.ru/microsoft-net/net-framework/language-compiler.html?showall=1>

<https://habr.com/ru/post/351498/>

Спасибо за внимание ☺.