|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | 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 3](#_Toc89024307)

[Постановка задачи 3](#_Toc89024308)

[Проектирование 3](#_Toc89024309)

[Тесты 4](#_Toc89024310)

[Лексический анализатор 5](#_Toc89024311)

[Постановка задачи 5](#_Toc89024312)

[Проектирование 5](#_Toc89024313)

[Тесты 5](#_Toc89024314)

[Синтаксический анализатор 7](#_Toc89024315)

[Постановка задачи 7](#_Toc89024316)

[Проектирование 7](#_Toc89024317)

[БНФ 7](#_Toc89024318)

[Реализация 8](#_Toc89024319)

[Тестирование 9](#_Toc89024320)

[Минимальная допустимая программа: 9](#_Toc89024321)

[Раздел переменных: 9](#_Toc89024322)

[Раздел операторов, составной оператор: 10](#_Toc89024323)

[Оператор присваивания: 10](#_Toc89024324)

[Условный оператор: 11](#_Toc89024325)

[Цикл с предусловием: 11](#_Toc89024326)

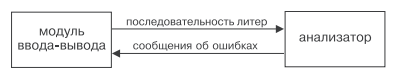
[Выражение, оставшиеся операторы: 11](#_Toc89024327)

[Нейтрализация ошибок 12](#_Toc89024328)

# 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.

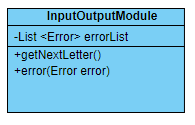


Рис.1

Для ошибки создадим отдельный класс Error (рисунок 2).

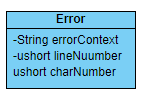


Рис.2

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.

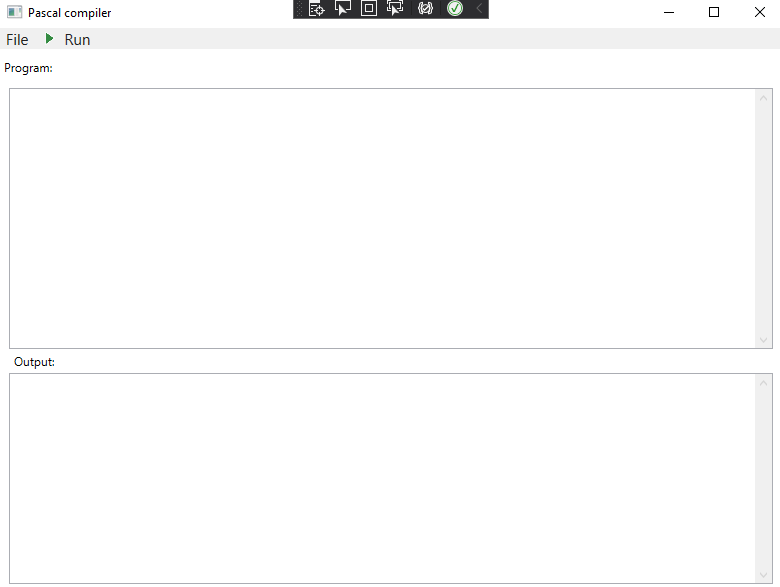
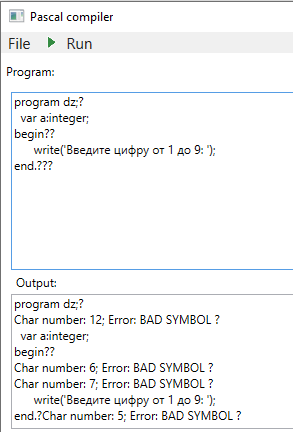
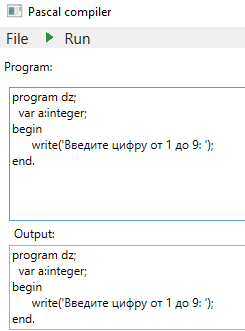


Рис.3

## 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.

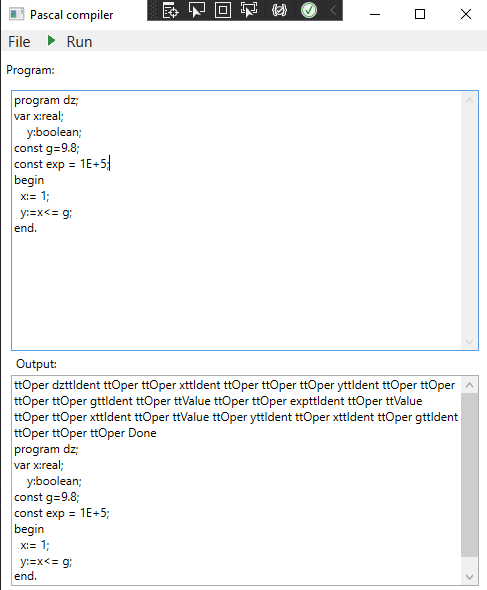
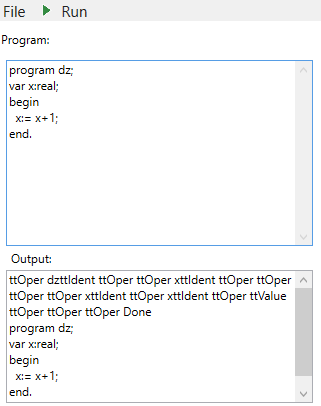
ttValue содержит значение константы, а также её тип (целое/вещественное/символ/строка).

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).

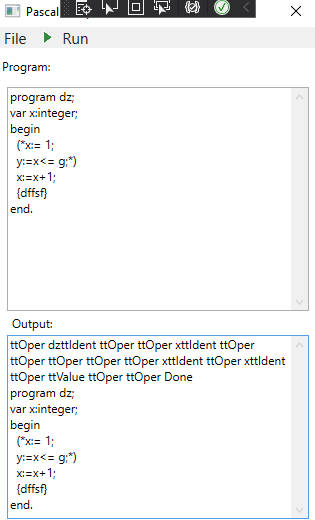
## Тесты

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

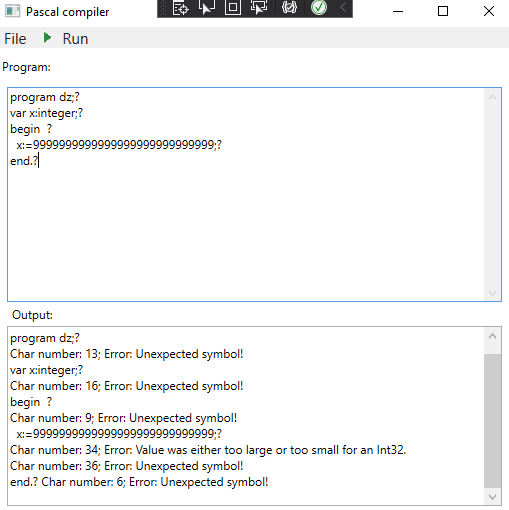
Нет ошибок:



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>

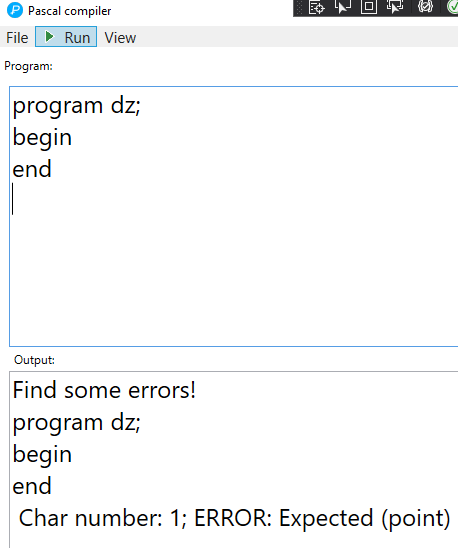
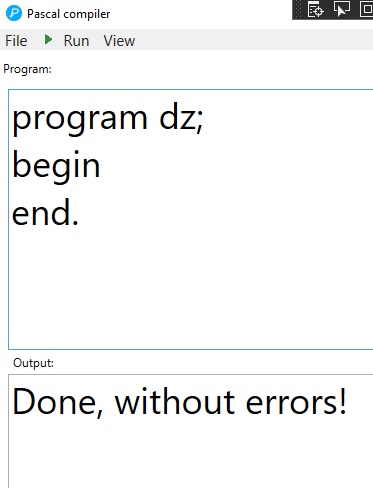
## Реализация

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.

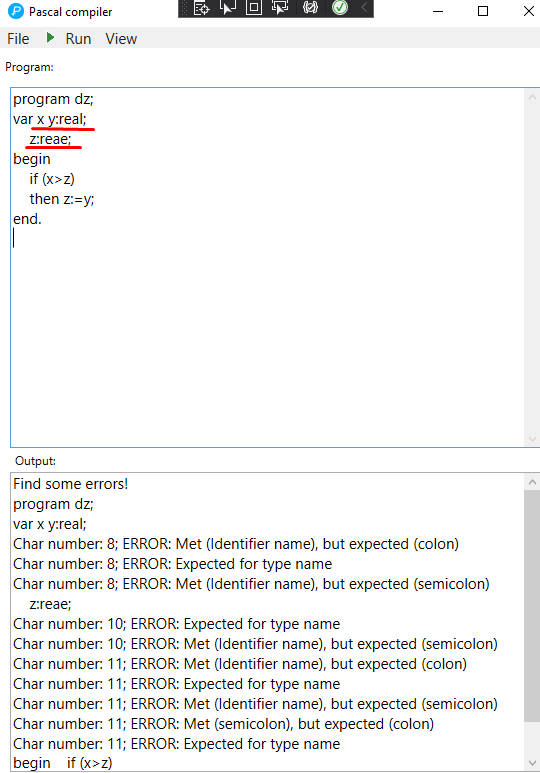
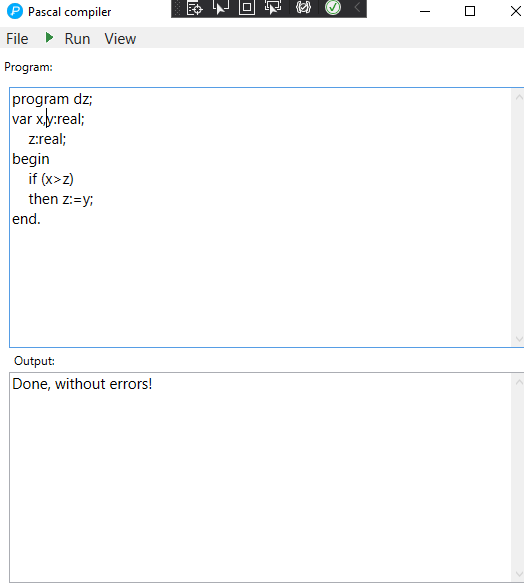
## Тестирование

Постараемся протестировать каждый блок по-отдельности:

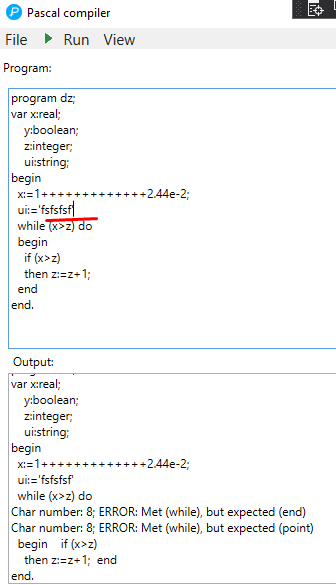
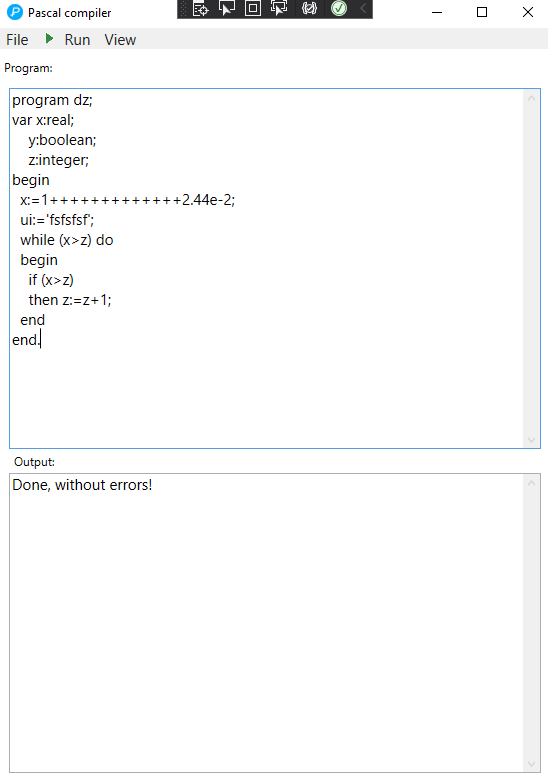
### Минимальная допустимая программа:



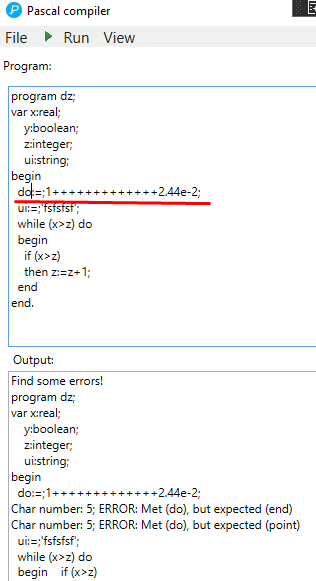
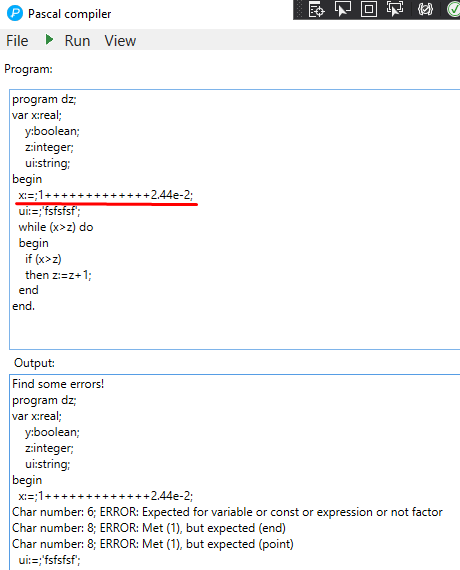
### Раздел переменных:



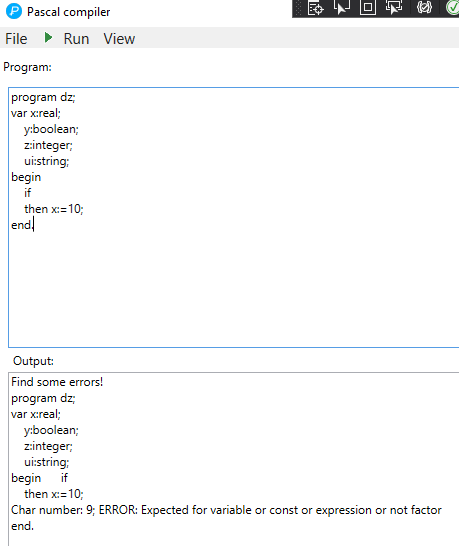
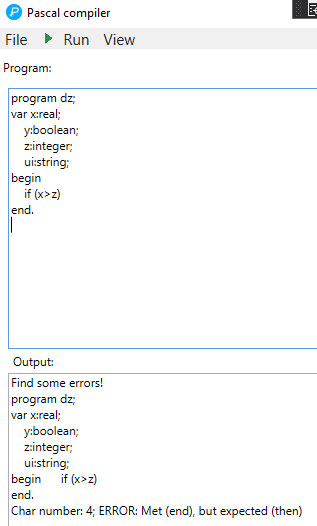
### Раздел операторов, составной оператор:



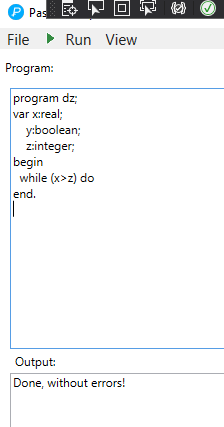
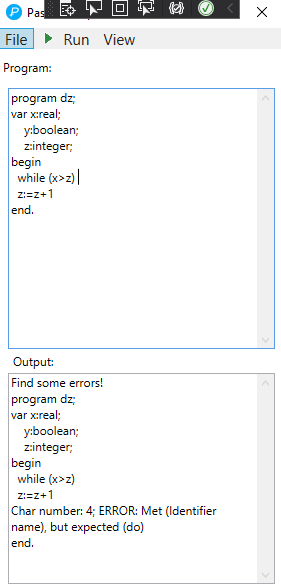
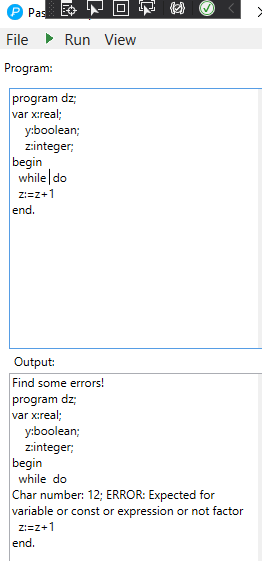
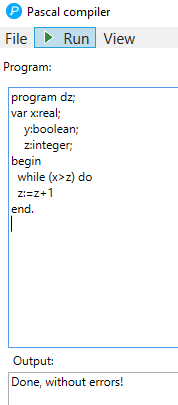
### Оператор присваивания:



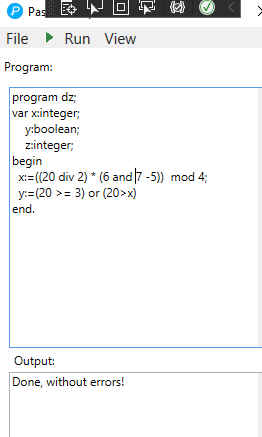
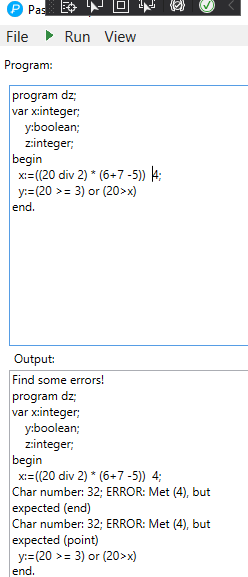
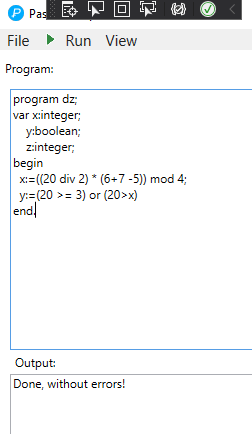
### Условный оператор:



### Цикл с предусловием:



### Выражение, оставшиеся операторы:



## Нейтрализация ошибок