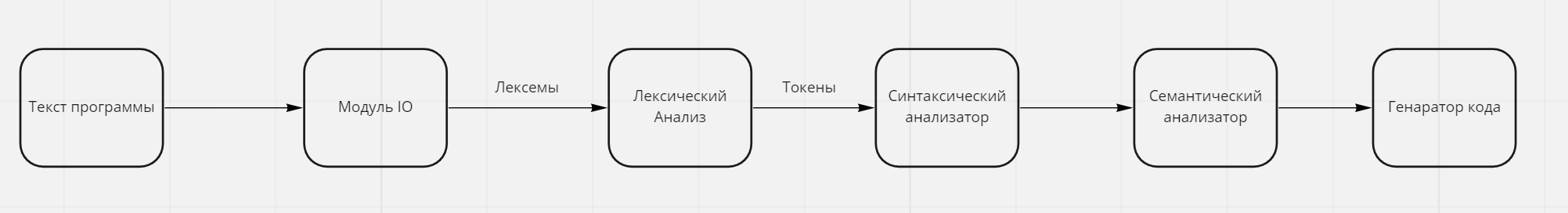
Глобальная цель: Написать компилятор языка Pascal

Постановка задачи:

Написать модуль ввода-вывода. Модуль должен считывать текст программы, разбивать его на лексемы(без разделения на виды)и иметь возможность вывести эти лексемы построчно.

Работа:

Изменим блок-схему архитектуры исходя из новых данных. Для начала создадим общую блок схему, которую после будем дробить на отдельные части. Компиляция программы проходит в несколько этапов, для каждого этапа будет своя блок-схема.



Кратко проанализируем задачи каждого из этапов.

1. Модуль IO. Должен считать текст программы и разбить его на лексемы(просто слова), по разделителям, вроде пробела, переноса строки и тд.
2. Лексический анализ. Разбивает полученные лексемы на токены. Токен в свою очередь представляет некий класс, в который преобразуется лексема, в нем уже есть информация о том что именно из себя представляет лексема(ключевое слово, переменная и тд.)
3. Синтаксический анализ. Проверка формальных правил
4. Семантический анализ. Проверка неформальных правил
5. Генератор. Генерация кода.

Теперь перейдем к написанию модуля IO. Блок-схему для этого модуля думаю нет смысла делать, здесь нужно просто исходя из текст программы разбить получить лексемы. Для этого нужно во первых определится с форматом хранения лексем, а во вторых с разделителями по которым будем разбивать текст программы на лексемы. По формату создать список ссылок на элемента теста программы или что-то вроде этого, так как в случае если программа большая хранить в памяти огромные куски кода—проблемно, но так как у нас упрощенная версия компилятора, думаю буду хранить просто список строк, если что потом можно усложнить, все-таки всегда в начале пишут упрощенный вариант, а уже после думают над улучшениями.

Что касается разделителей, здесь все не так просто. Проанализируем все возможные варианты:

1. Табуляция и пробел(В случае если не является частью строки-переменной)
2. Знак переноса строки \n, \r
3. Арифметические операции( они могут писаться слитно, плюс сами по себе являются лексемами), отдельно обратить внимание на :, <=,>=
4. () используется много где, содержимое скобок надо трактовать из смысла соседей, поэтому будем считать, что это одна лексема.

Кроме того возможен этот вариант, c кучей пробелов:  
program Hello;

begin

writeln ('Hello World')

end.

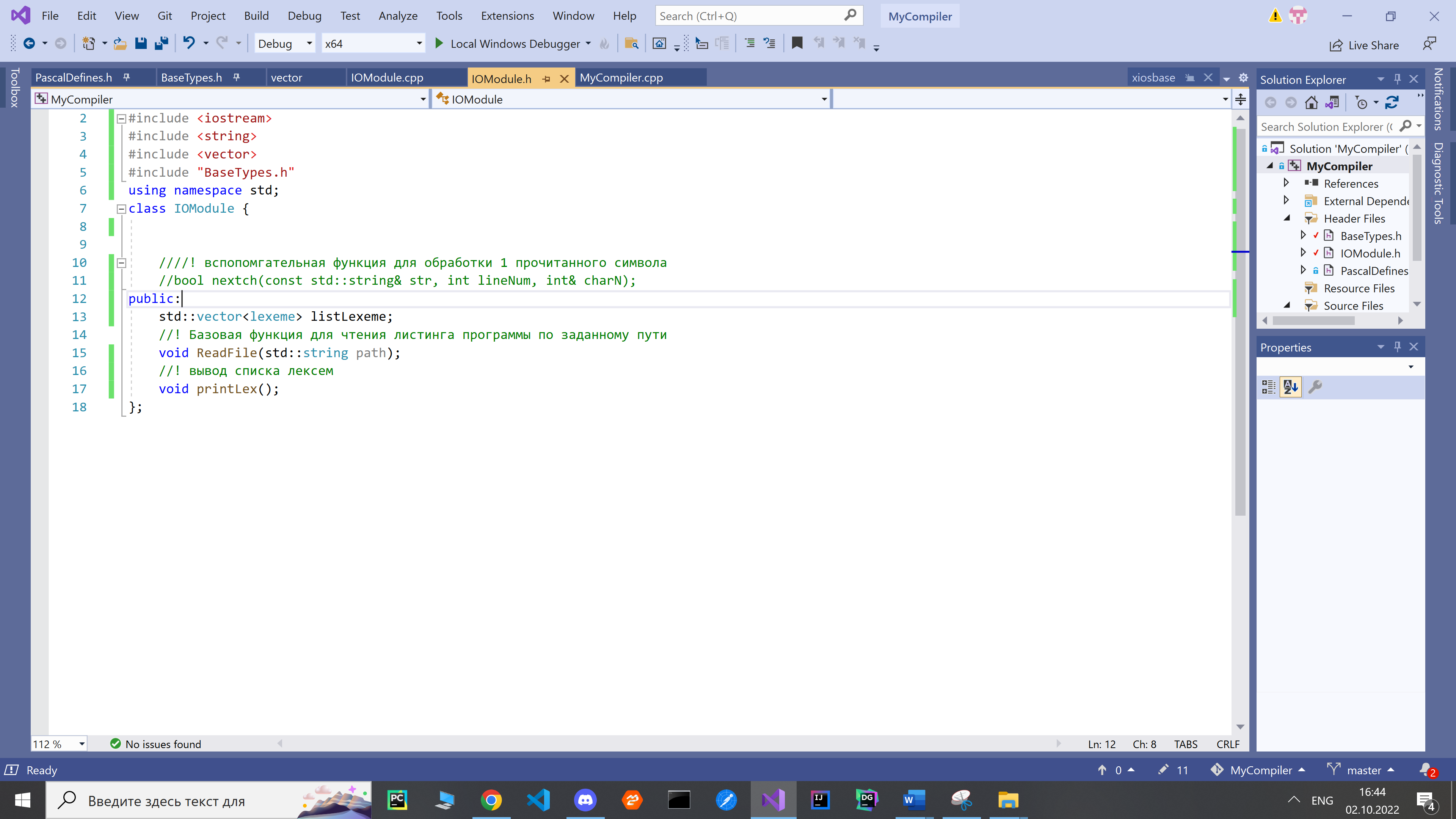
Еще интересные ситуации могут быть когда внутри арифметического выражения функцию Вызывают. Вообщем на данный момент сойдемся на том, что на этом этапе содержимое скобок всегда одна лексема(исключение комментарий).

1. “” – используются для строк, содержимое считать одной лексемой
2. Комментарии {} (\*\*), просто выкидываем, другие задачи этих скобок думаю просто игнорировать.
3. . и ; тоже разделители, но все-таки будем выделять их отдельными лексемами и думаю в будущем отнесем к ключевым словам.

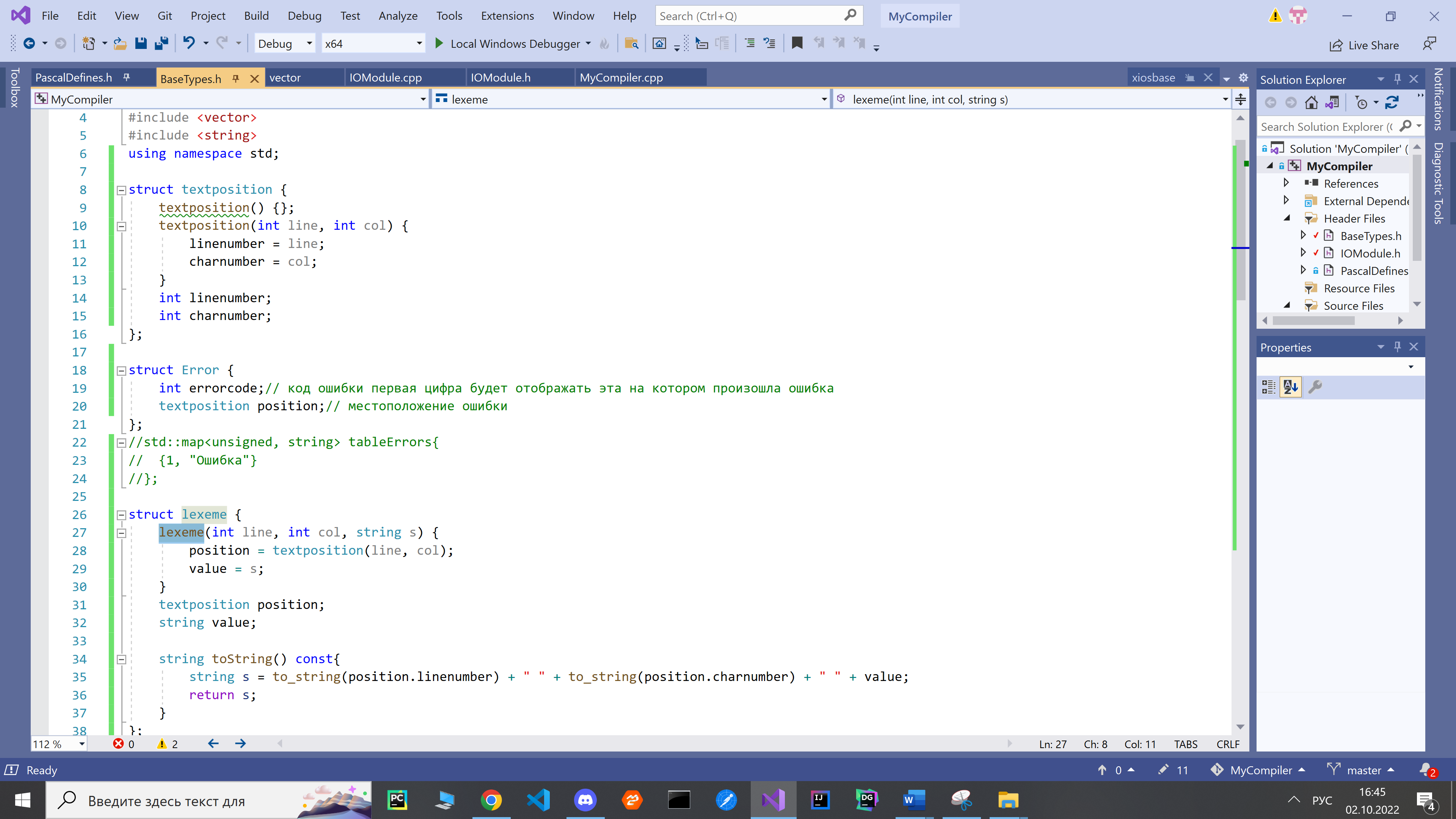
Также думаю на этом этапе есть смысл вывода ошибок в случае отсутствия закрывающего тега, остальные ошибки пока игнорируем и отсутствия = после :

Также думаю в формате хранения лексемы имеет смысл сразу хранить и позицию в коде.

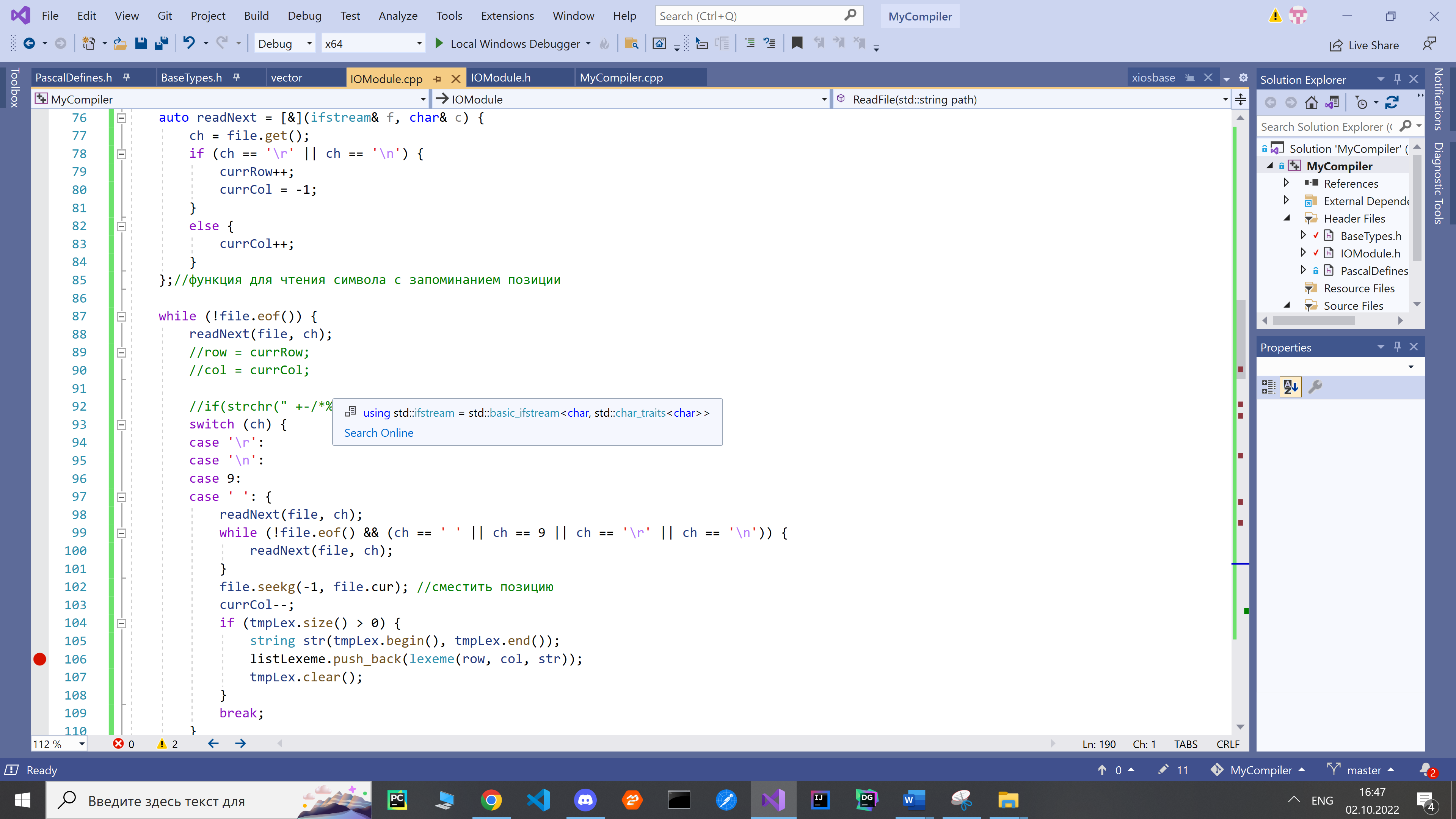
В результате написания кода у меня появился класс:



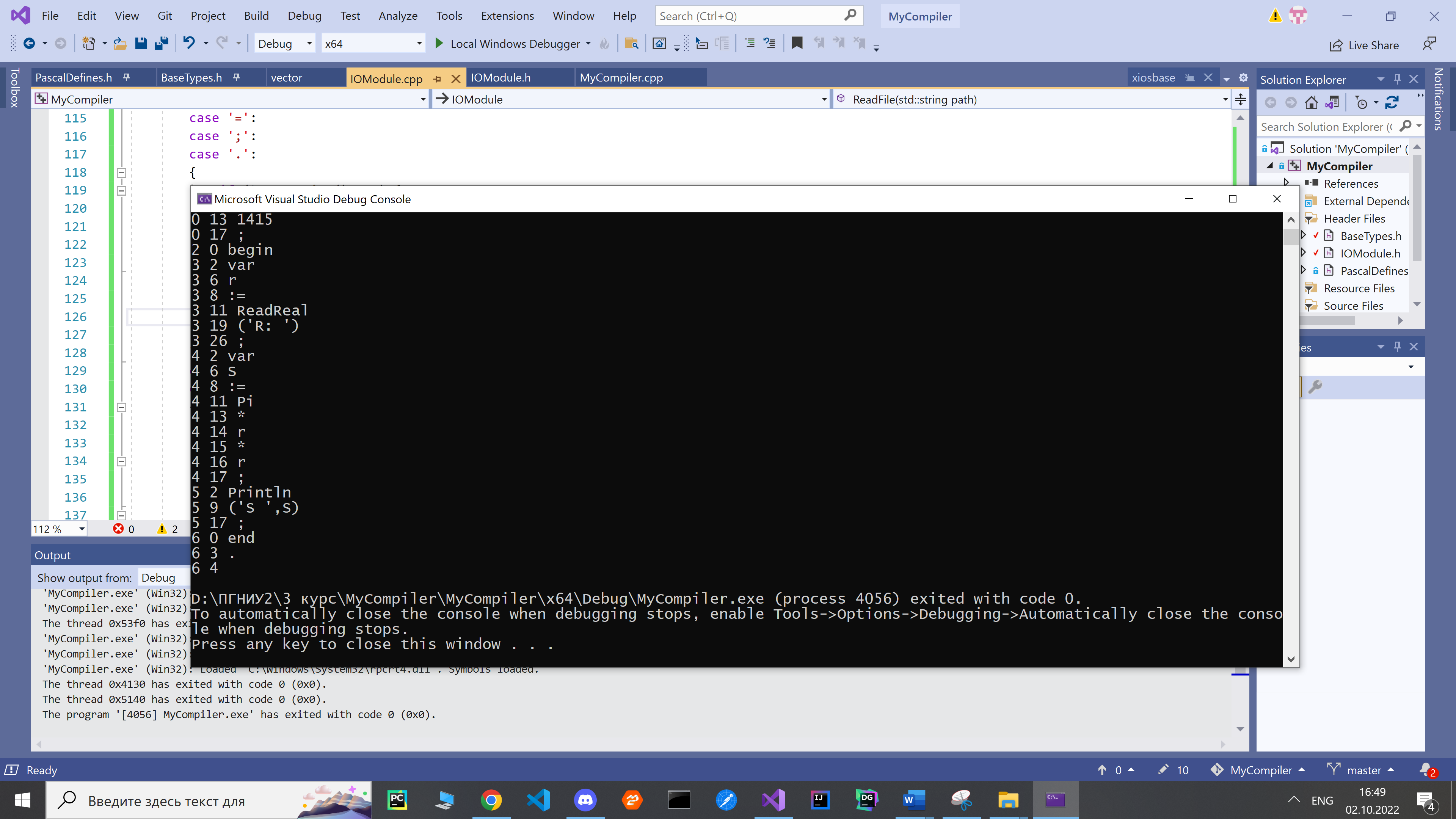
Также добавил файл с общими переменными, там же будет храниться таблица имен и тд



Сама функция чтения реализована просто через switch case, по итогу она заполняет поле-список лексем внутри класса IOModule.



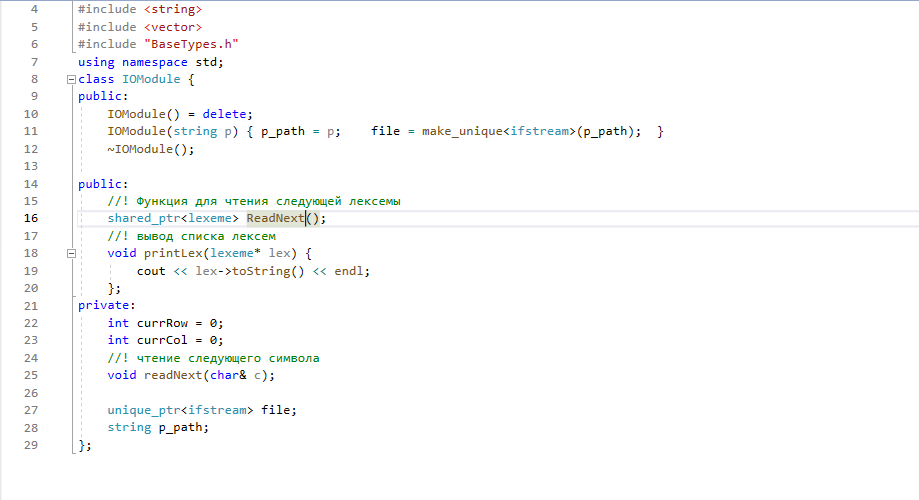
Также я добавил функцию вывода для модуля IO



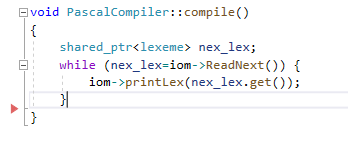
Далее нужно написать лексический анализ для улучшения обработки лексем, а также подумать как сделать таблицу ошибок, возможно сттоит использовать try catch.

**Пересмотр структуры компилятора.**  
После одной из пар, стало ясно, что компиляция должна производиться по мере чтения листинга, то есть после чтения лексемы она должна сразу проходить все этапы компиляции. Поэтому текущий модуль вввода/вывода был переписан на новый вариант.  
   
Также еще раз посмотрим на задание и сделаем упор в начале на создание минимальной части, поэтому временно вырежем остальное.

*Общая минимальная часть.* Основные разделы программы: раздел описания переменных, раздел операторов. Переменные стандартных типов (Boolean, integer, real, char). Числовые константы. Арифметическое выражение (в выражении допустимы только константы, переменные, операции +, –, \*, / и скобки). Оператор присваивания и составной оператор.

Модуль IO поменялся вот так(функция теперь возвращается следующую лексему по опрядку):  


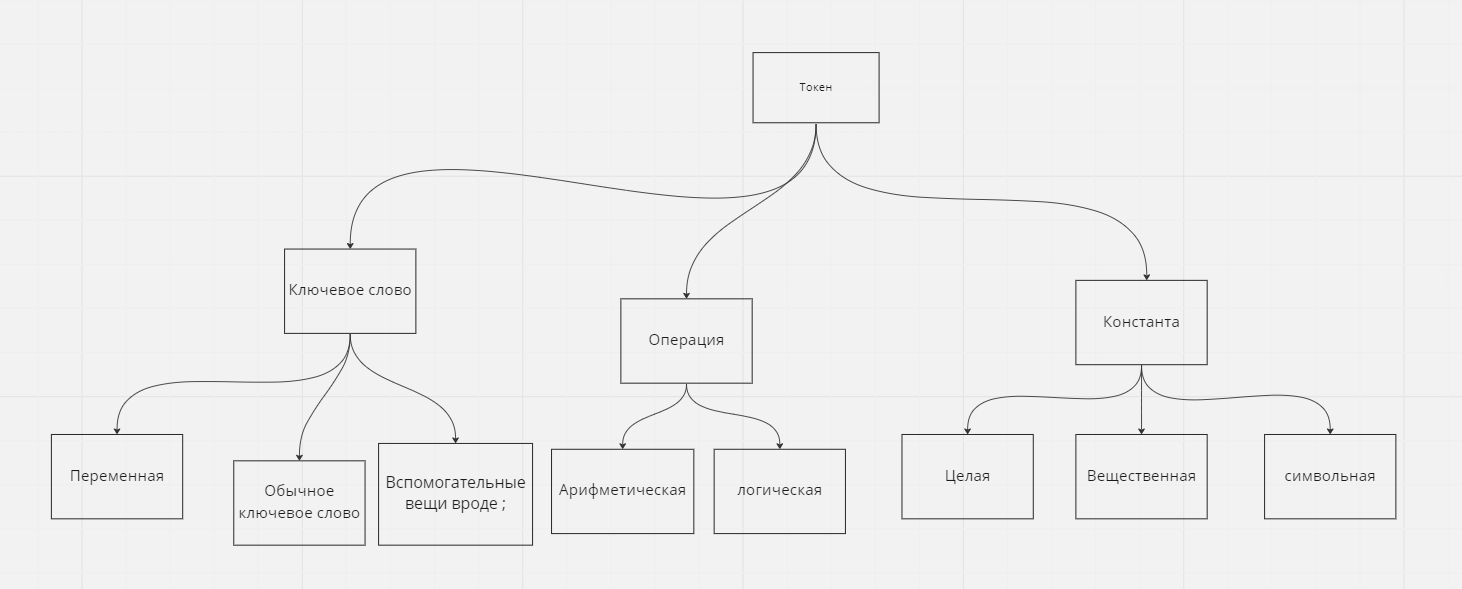
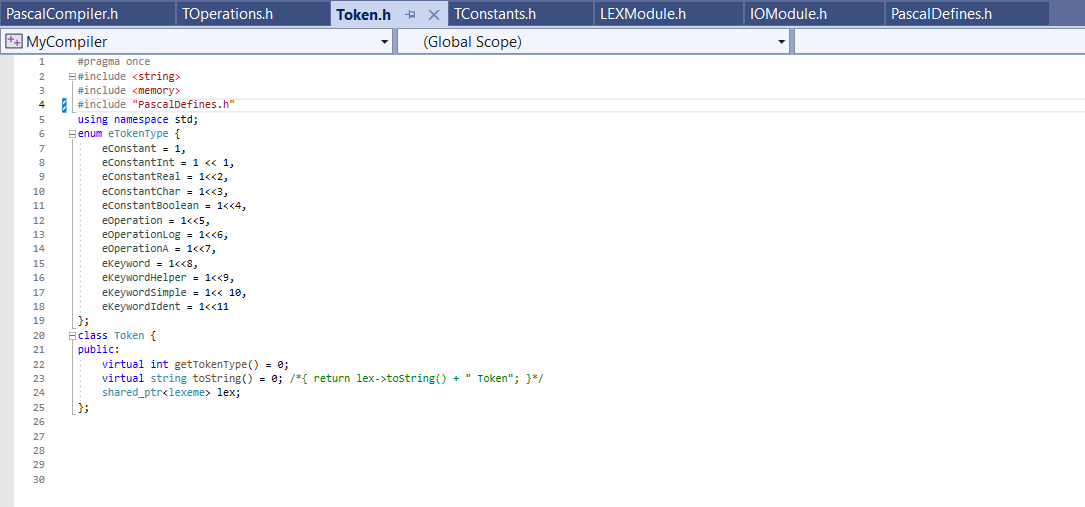
Кроме того было решение сделать отдельный класс для компилятора, может он и не обязателен, но в нем пока что только в функции compile буду последовательно запускать все этапы.  


Пока что compile выводит просто список лексем.  
  
**Лексический анализатор**

На прошлом этапе мы получили лексему, теперь необходимо преобразовать ее в некий токен, тип данных который будет понимать мой компилятор.  
Для этого определим какие виды Токенов будут присутствовать.  
Токен: Переменная/Ключевое слово, Константа, оператор(арифметика)

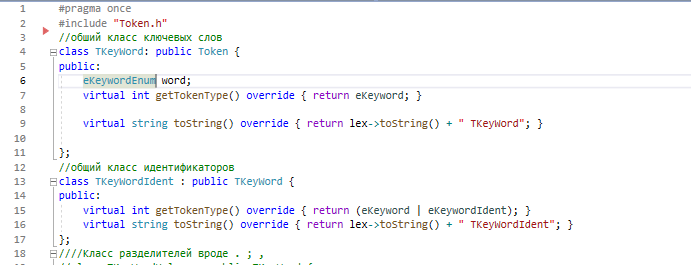
Возможно стоит оператор совместить с ключевыми словами тоже. Однако только они будут использоваться при польской записи, так что все-таки думаю их стоит держать отдельно, тем более вероятно в следующем шаге после := ожидается арифметическое выражение, а там могут использоваться идентификаторы, операторы, функции, константы, (понять переменная или идентификатор можно будет при анализе блока var, однако возможно их тоже стоит вынести в отдельный тип).   
Пока остановлюсь на таком наборе токенов.

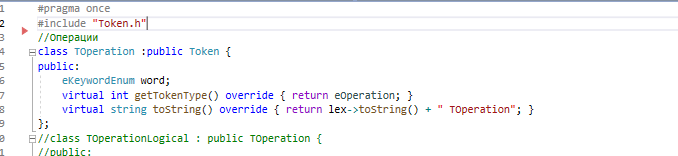
Я набросал такую схему:

  
Стоит заметить что лексические ошибки здесь тоже должны быть обработаны, но суть в том что раз каждая лексема проходит через все этапы, то на каком-то этапе должна формироваться таблица имен, поэтому здесь будет проверяться наличие идентификатора в таблице имен, которую формирует вероятно во время семантического анализа.  
Попробуем реализовать данный набор классов. Обход ошибок пока отложим, так как таблица имен формируется позже  
  
Во время реализации по началу хотел построить полную иерархию классов, но после понял, что вероятно подробное дробление не понадобится, поэтому я сделал в начале полную, а потом временно урезал.  
  
Класс токена:  


Я подумал, что возможно стоит использовать битовую маску, чтобы знать к каким именно типам токена относится объект, однако веротяно этот enum можно просто заменить на **dynamic\_cast** в нужном месте. В случае чего в будущем это вырежу.

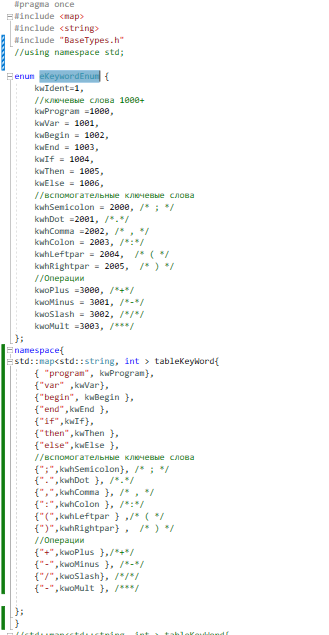
Для класса констант вышел такой набор:  


Ключевые слова:  


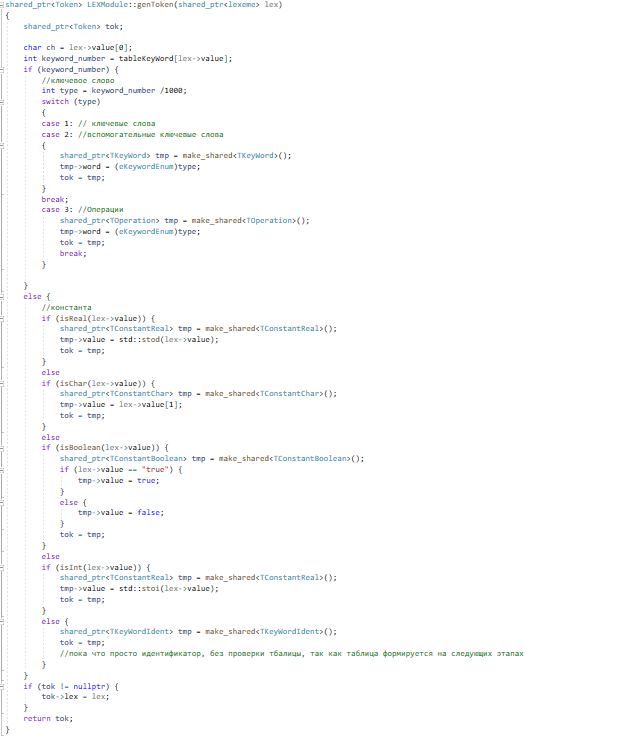
Операции:  


Также понадобиться таблица, чтобы понять вид ключевого слова.

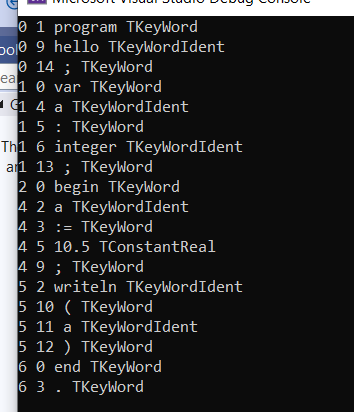
Для этого выделен отдельный хедер.



Соответственно для хранения был использован map. Enum тоже построен особым образом, по остатку от деления на 1000 можно понять подвид ключевого слова.  
  
Теперь, когда у нас есть нужна иерархия, надо преобразовать полученную лексему в соответствующий токен. Токены построены таким образом, что в случае чего можно поменять иерархию без особых проблем.  
  
сам класс лексера:  

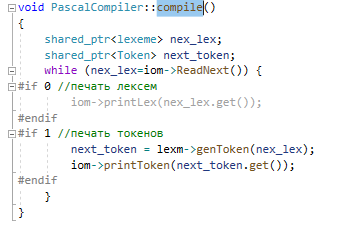

Функция преобразования лексемы в токен:  


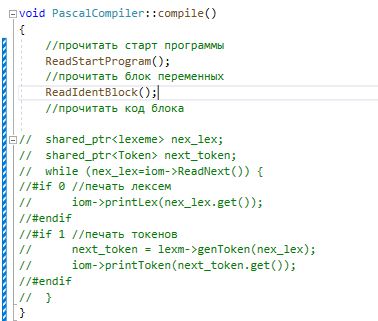
Теперь в базовом классе компилятора после получения лексемы можно сразу получить токен.  


Используя перегруженный метод toString получился такой вывод токенов:  
  
Пока что все неизвестное относиться к идентификаторам, потому что формирование таблицы имен пока не реализовано.  
Далее нужно сделать Синтаксический анализ.

**Синтаксический анализ.**

На этом этапе нужно проверить формальные правила языка. При подробно рассмотрении стало ясно, что вариант когда класс компилятора последовательно запускает все этапы не совсем удачный:



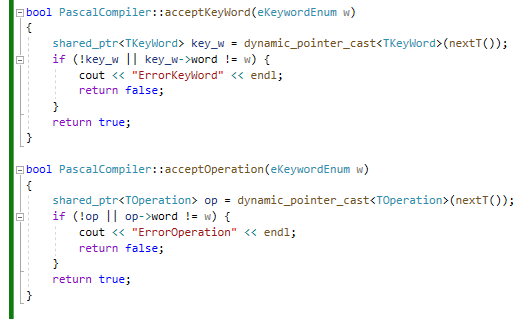
Если идти по этому пути, то надо запоминать кучу состояний у компилятора, а это не удобно. Поэтому думаю сделаю так, что лексер может дергать IO а класс коммпилятора будет отвечать в том числе и за синтаксис. Таким образом находясь в других методах можно будет дергать следующий токен без особых проблем.  
Поправим лексер.  
Теперь функция compile выглядит так:  


А в лексер добавлена функция, которая использует IO для создания токена.



Далее напишу чтение ключевых блоков в compile.

Решил все-таки вынеси переменные отдельным типом токенов.

Написал вспомогательные функции для чтения проверки токена.  


Возможно потом стоит добавить запоминание предыдущего токена в эти функции. Все-таки вероятно где-то понадобится смотреть последовательность токенов.

Теперь перывый блок старта программы выглядит так:



**Теперь напишем остальные функции.**

Перечислим основные правила, которые надо запрограммировать, правая часть правила должна быть какой-то функцией.

+1)<Программа > ::=program <имя>;<блок>.

+2)<блок>::=<раздел переменных><составной оператор(раздел операторов)>

+3)<раздел переменных>::=var {<перечисление переменных>;}

+4)<перечисление переменных>::=<имя перменной>{, <имя переменной>}:<тип>;

+5)<тип>::=integer | Boolean|char|real

+6) <составной оператор>::=Begin {<оператор>} end

+7)<оператор>::={<составной оператор> | <присваивание> | <условный оператор>};

+8)<присваивание>::=<имя переменной>:=<выражение>

+9)<выражение>::={{-}<операнд>{<операция><операнд>}}

+10)<логическое выражение>::=<лог операнд>| <выражение><лог операция> <выражение> | <логическое выражение>=<логическое выражение>//без связок

+11)<лог операнд> ::= <булевская переменная> | <булевская константа>

+12)<операнд>::=константа |переменная| ( <выражение>)

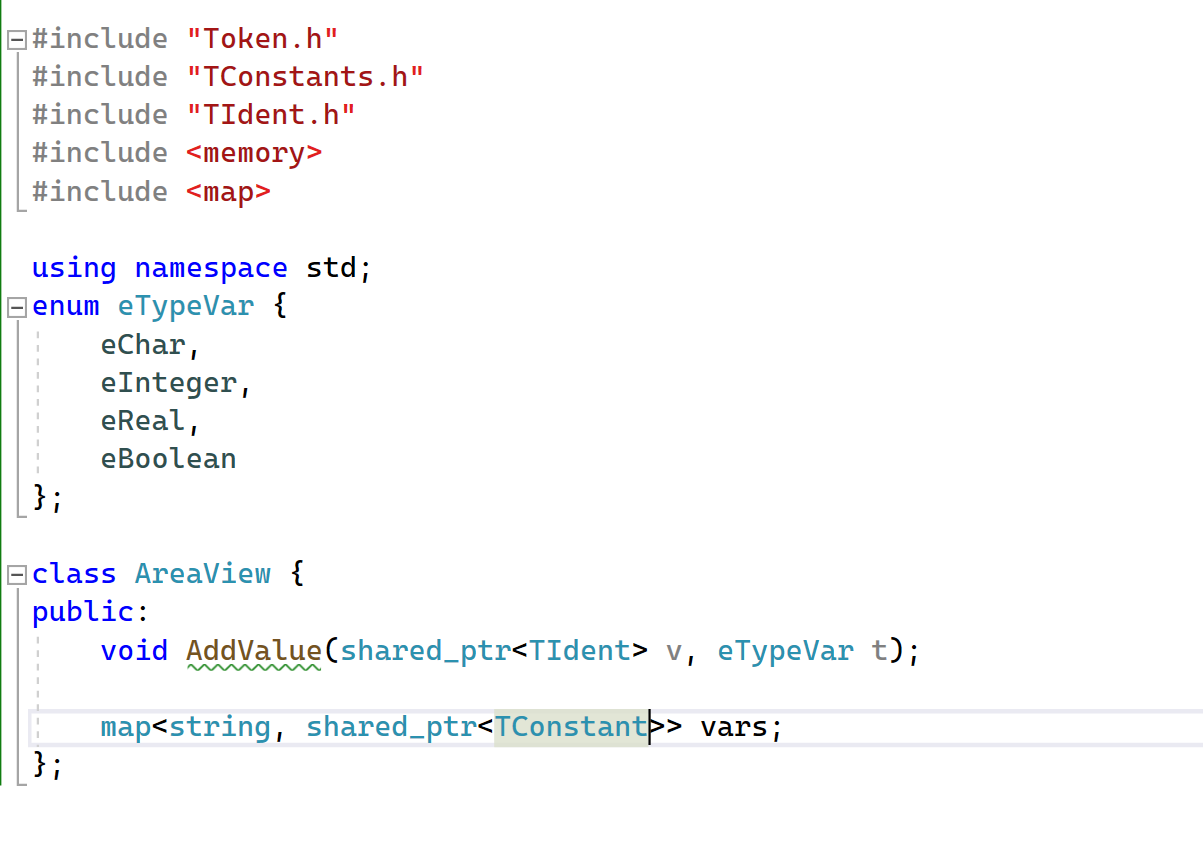
+13)<операция>::=+ | - |/ |\*

+14)<логическая операция>::=<|>|=

+15)<условный оператор>::=if <условие> then <оператор> {else <оператор>}

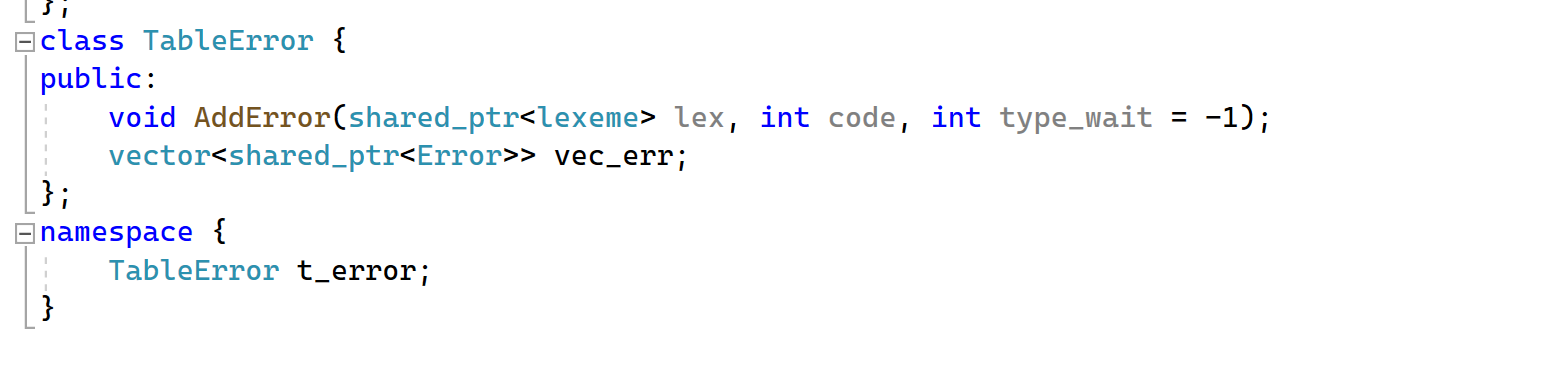
+16)<условие>::=<булевская переменная> | <логическое выражение>

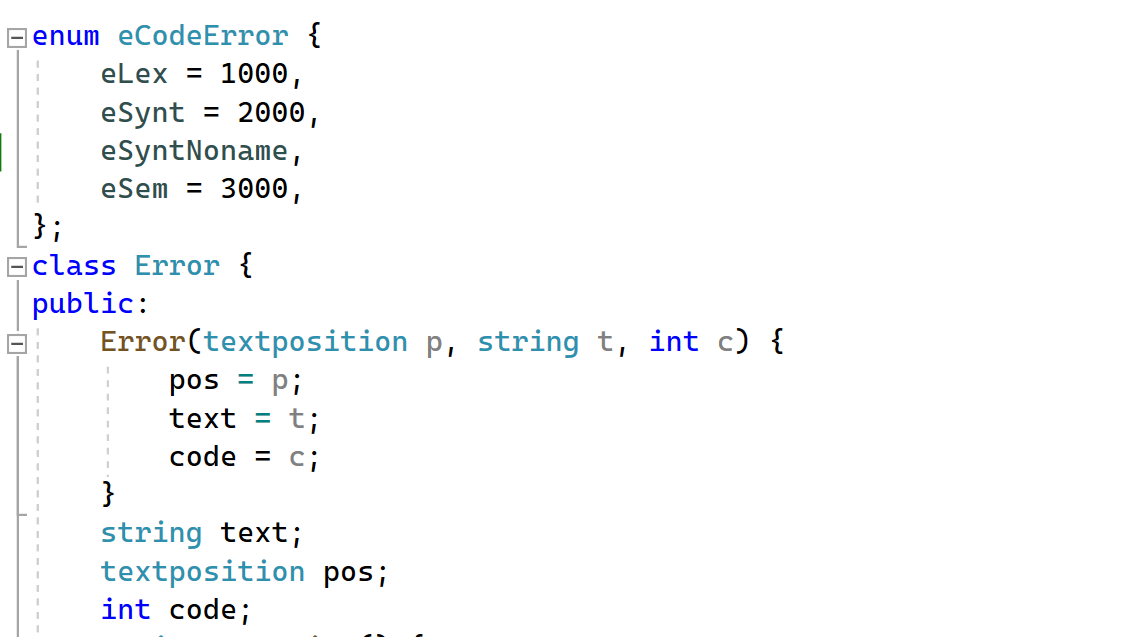


Во время реализации функций, стало ясно, что таблицу имен можно создать сразу после прочтения блока перменных, поэтому сразу сделаем и семантику.  


Как видно из рисунка, таблица имен из себя представляет map связанный по имени с константой.

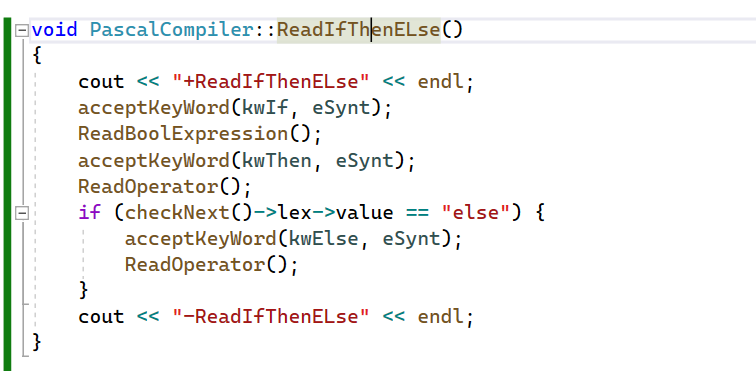
Также было решено сразу добавить нейтрализацию ошибок, для чего используется специальная таблица ошибок, в которую по мере необходимости добавляются новые.





Что касается семантики, то, так как мы не пишем функции, достаточно одной области видимости.

If else:



Цикл:= while <условие> do <оператор>

