**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Пермский государственный национальный исследовательский университет»**

**Формальные грамматики и методы трансляции**

**«Компилятор для Pascal»**

Работу выполнил студент группы

ПМИ-1 **Орлов А.В.**

**Пермь 2023**

Компилятор состоит из нескольких модулей:

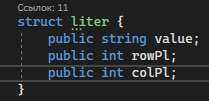
1. Модуль Ввода-вывода
2. Лексический анализатор
3. Синтаксический анализатор
4. Нейтрализация ошибок
5. Семантический анализатор

Давайте пройдемся по каждому и посмотрим что они делают.

1. Модуль Ввода-вывода:

При создании объекта класса Ввода-вывода(InOut), мы передаем путь к файлу, содержащему текст программы на языке Pascal, с которого модуль будет постепенно считывать по строке. После этого, так же по требованию, будет выполнено разделение на отдельные символы - «литеры».

Каждая найденная литера будет записана в переменную «l» специального типа liter:



В которой хранится строковое значение литеры и ее положение в программе (т.е. в каком столбце и строке ее начало).

Также стоит упомянуть что на этом этапе пропускаются не нужные для дальнейшего анализа части программы, такие как комментарии, табы и пробелы.

Полученную литеру мы передаем в лексический анализатор.

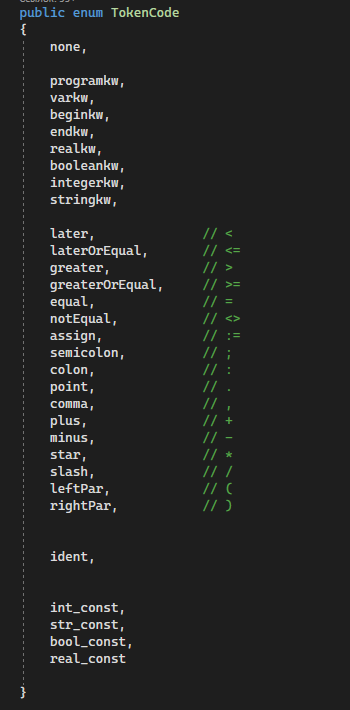
1. Лексический анализатор

В лексическом анализаторе, мы проверяем к какому типу принадлежит эта лексема, это может быть «Оператор», «Ключевое слово», «Идентификатор» или «Константа».

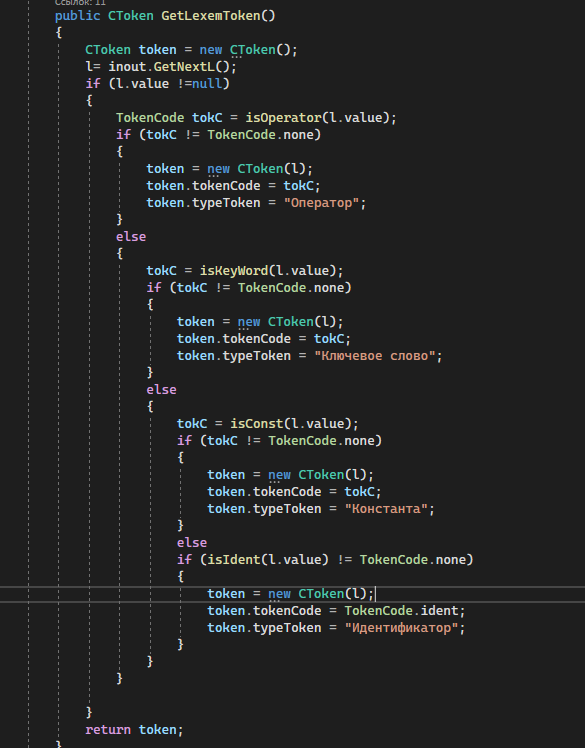
Для этого предварительно создаем класс CToken со структурой:



где в будет хранится тип токена, его индивидуальный код и литера из которой получился данный токен.

Коды токенов удобно хранить в перечеслимом типе данных enum:

Сразу же при проверке мы запоминаем какому именно коду токена соответствует литера, возвращаем его из функции и если он не ноль, присваиваем в соответствующее поле.

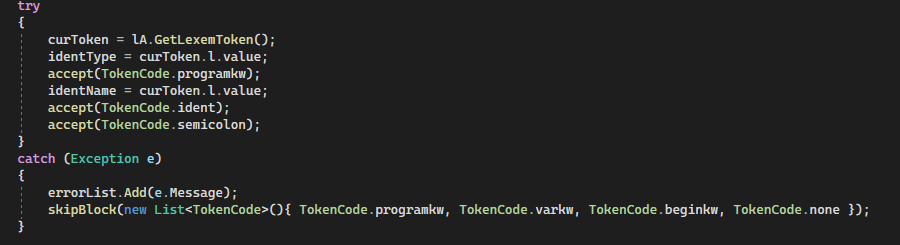


В результате этот токен попадает уже в модуль синтаксического анализатора.

1. Синтаксический анализатор

Данный модуль отвечает за выявление основных конструкций языка.

Начиная с самого начала программы, мы получаем токены из лексического анализатора и для каждого блока (programBlock, varBlock, mainBlock) проверяем, соответствует ли порядок этих токенов правилам языка.

Например:

Здесь мы проверяем порядок токенов для блока Program, то есть чтобы у нас пришло ключевое слово program, потом название программы и точка с запятой.

Функция accept, принимает в параметрах нужный в данный момент код токена, а затем проверяет соответствует ли текущий токен (хранящийся в поле класс curToken) нужному, в противном случае вызывает ошибку.



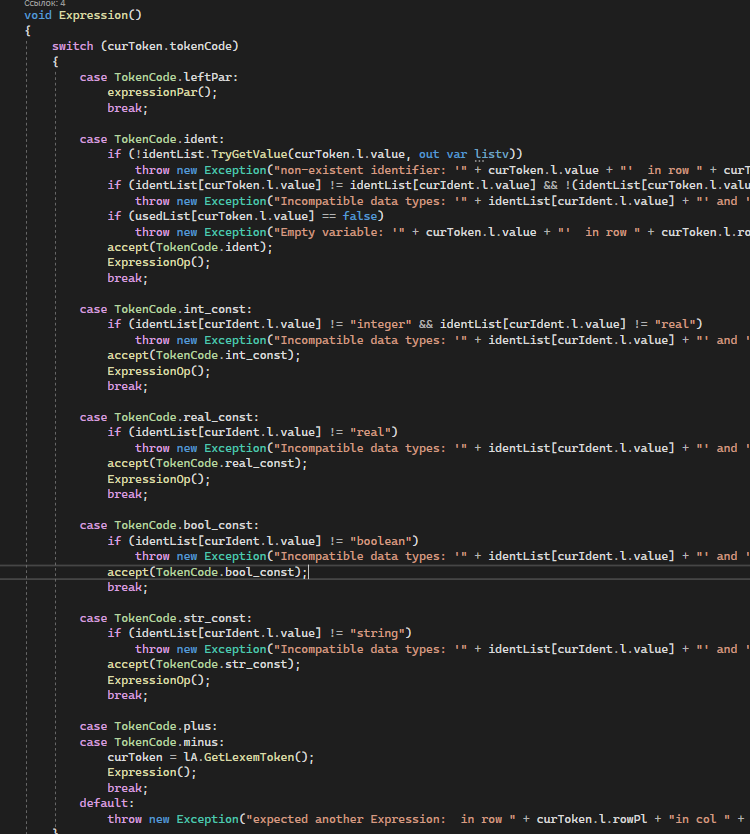
Также здесь проверяется на наличие записей после точки.

Для хранения всех найденных ошибок, мы создаем лист,



Как можно заметить выше, мы вызываем accept в блоке try а в catch обрабатываем эту ошибку и записываем ее в наш лист.

Еще как пример БНФ для выражений:



\*Это момент после того как пришел идентификатор и оператор присвоить.

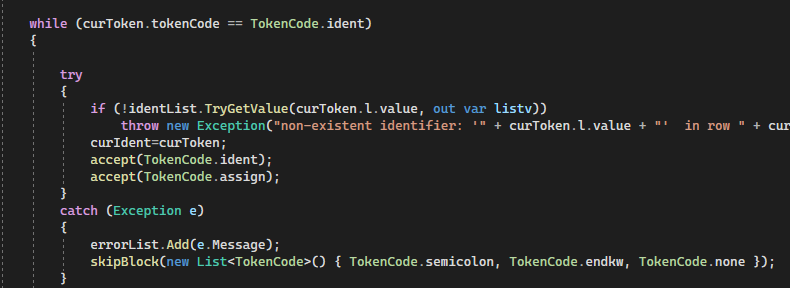
Мы смотрим что у нас идет дальше и в зависимости от этого вызываем определенные ветки БНФ. Например если пришел int\_const, то мы вызовем ветку, где буду ожидаться какие либо операторы мат операций (+, -, \*, /) или ожидается конец выражения.

Таким образом мы проходимся по всей программе, собирая ошибки.

1. Нейтрализация ошибок

Когда у нас встречается определенная ошибка, нам было бы полезно пропустить небольшую часть кода, а затем продолжить анализ, для определения других ошибок. Для этого и существует такой блок как нейтрализация ошибок.

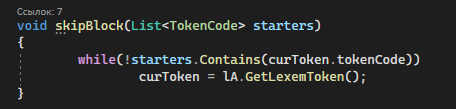
Пример:



\*действие происходит в основном блоке программы после begin

Тут мы ожидаем идентификатор и оператор присвоить.

Если нам пришло что то другое то мы вызываем skipBlock которая будет получать следующий токен, пока не дойдет до ближайшей точки с запятой или конца программы (токен код none нужен, чтобы если больше вообще ничего не встретится то работа компилятора корректно завершилась). Когда он дошел до ближайшей точки с запятой, со следующей строки можно смело начинать анализ уже следующих конструкций.

Сама функция skipBlock:

1. Семантический анализатор

Этот модуль дополняет семантический анализатор, а если конкретно то здесь мы добавляем проверки контекстных условий и смысловых конструкций нашей программы.

Первым делом создадим таблицу идентификаторов, (\*у нас она будет одна, т.к. при этих условиях нет разных областей видимости) где будем хранить название и тип каждого приходящего идентификатора. Она создается для того чтобы контролировать неверные присвоения по типу

«boolean := string»

И контролировать объявление идентификатора с уже существующим именем.



Я буду хранить эту таблицу в словаре, чтобы в дальнейшем удобно было обращаться к типу идентификатора через ключ (название)

Создадим еще один словарь, но тут будем хранить уже не тип идентификатора, а информацию о том, присваивали ли идентификатору какое то значение.

Тут контролируется момент присвоения в какой либо переменной пустого значения.



Все ошибки также через catch добавляются в лист ошибок и вызывается «скип-функция»

Также имеется обработка ошибки, когда пытаются присвоить какое то значение не существующей переменной.

