# Лексический анализатор

## Постановка задачи:

Требуется написать лексический анализатор для языка паскаль.

Лексический анализатор – итератор по словам текста программы, параллельно переводящий слова в токены. Токен – более короткое (кодовое) представление слова программы.

## Проектирование

Компилятор состоит из трех модулей. Первый – главный, в котором объявлена главная функция main и класс компилятора. Второй – типы данных, в котором описаны все возможные типа данных, которые могут нам понадобиться: Токен, Константа и Паскалевские ключевые слова. Третий – стадии компиляции (процессоры), где описаны классы соответствующие каждой отдельной стадии компиляции. В данный момент там описан только Лексический анализатор.

На рисунке 2.1 представлена диаграмма классов для компилятора паскаля.

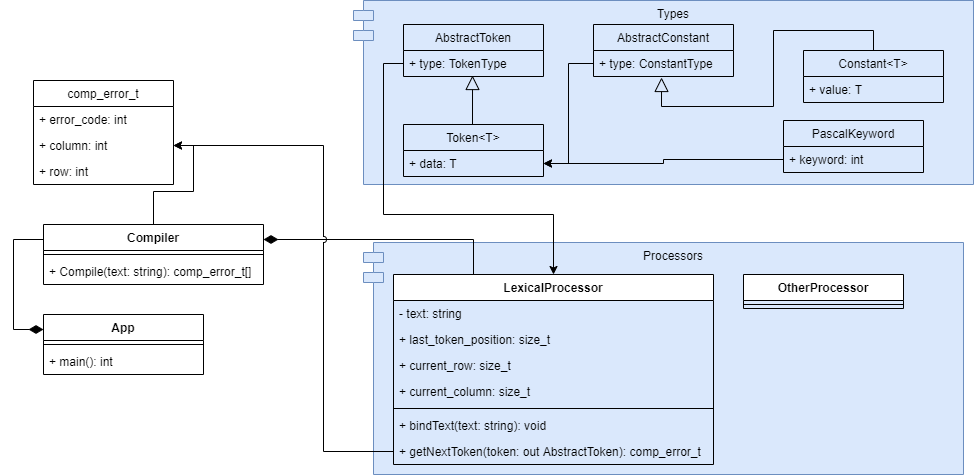


Рисунок 2.1 – Диаграмма классов компилятора Паскаля

Лексический анализатор построен в виде класса. К объекту лексического анализатора можно прикрепить текст программы и затем запустить последовательный обход токенов, с помощью метода getNextToken. Это основной метод, который выполняет всю основную работу по разбору (парсингу) текста, определению типа лексемы и формированию токена.

Интересным образом спроектирована система возврата ошибок. Лексический анализатор может вернуть одну ошибку за один разбор токена. Во время выполнения функции компиляции, компилятор следит чтобы ошибок не был. Если таковые будут, то он будет складывать их в специальный массив, который затем вернет управляющей функции main. В составе ошибки только позиция в тексте и код ошибки. Функция main, когда получит массив ошибок, будет последовательно разбирать коды ошибок и строить по ним текстовое описание, которое будет выводить на экран.

Также функция main ответственна за чтение текста программы из файла. Файл читается целиком (возможно это неправильно и хорошо бы читать файл по частям, но вроде оперативной памяти достаточно чтобы текст файла целиком поместить) в специально отведенный для этого буфер.

## Реализация

Все ключевые слова паскаля представлены в виде целочисленных кодов (идентификаторов), а чтобы быстро осуществлять перевод текстового представления в кодовое использован словарь (std::map), где по строке можно найти код символа.

Для того, чтобы в качестве токена могли выступать разнотипные данные (константы, паскалевские ключевые слова, строки), были использованы шаблоны языка С++. Притом, чтобы все токены были родственны друг другу, был создан общий класс-предок – абстрактный токен, от которого наследуются все остальные виды токенов. Абстрактный токен хранит только тип токена, чтобы по данному типу выполнять приведение указателя к нужному классу-предку.

Для улучшения читаемости кода разбора текста, были использован автомат, которые разбирает слово и в зависимости от встреченных символов переходит из состояние в состояние.

Всего выделено 6 состояний:

* Начало разбора,
* Разбор целого числа (первый символ - число),
* Разбор вещественного числа (можно прийти только после разбора целого числа),
* Разбор строковой константы (слово начинается с ‘ и заканчивается ‘),
* Разбор идентификатора (первый символ буква или \_),
* Конец разбора.

На рисунке 3.1 представлена схема реализованного автомата.

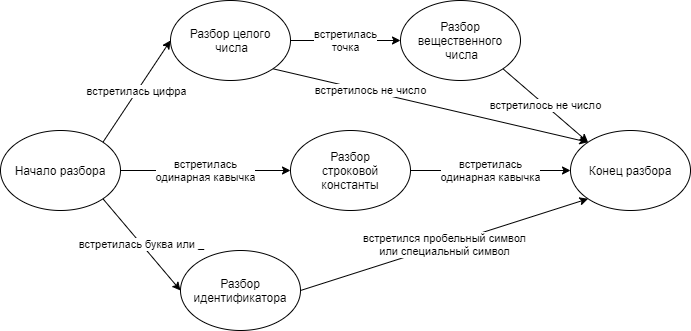


Рисунок 3.1 – Схема автомата разбора слов

Был обнаружен баг системы ввода вывода. Когда пытаешься определить размер файла (чтобы разом его считать), то размер определяется с учетом наличия в файле символа возврата каретки (‘\r’). Обычно этот символ ставится рядом с символом перехода на новую строку. Однако функция считывания текста из файла считывает без учета этого символа. Получается, что считанное количество символов меньше, чем в самом файле. Из-за этого может быть ряд проблем: в частности, лексический анализатор может найти идентификаторы даже после того, как разобрал весь осмысленный текст программы.

## Тестирование

На вход нашему лексическому анализатору подадим текст программы, представленный на рисунке 4.1.

program test;

var a,b,c: int;

begin

a := 10;

b := -10.01;

c := 'test123';

d[5] := +0.0;

d[3] := -1;

y = 2 \* f(x) - 1.5;

if (a <= b and b <> c) then

begin

x1 := map + 1.1;

end;

d := 1000000000000000000000000;//вызовет переполнение

end.

Рисунок 4.1 – Текст входной программы

На выходе должны получить полный список токенов и одну ошибку говорящую о переполнении целочисленной переменной.

Результат работы представлен на рисунке 4.2.

|  |  |  |
| --- | --- | --- |
| keyword token: 24  ident token: test  keyword token: 47  keyword token: 32  ident token: a  keyword token: 45  ident token: b  keyword token: 45  ident token: c  keyword token: 46  ident token: int  keyword token: 47  keyword token: 2  ident token: a  keyword token: 54  const token: int constant: 10  keyword token: 47  ident token: b  keyword token: 54  const token: real constant: -10.01  keyword token: 47  ident token: c  keyword token: 54  const token: string constant: test123  keyword token: 47  ident token: d | keyword token: 42  const token: int constant: 5  keyword token: 43  keyword token: 54  const token: real constant: 0  keyword token: 47  ident token: d  keyword token: 42  const token: int constant: 3  keyword token: 43  keyword token: 54  const token: int constant: -1  keyword token: 47  ident token: y  keyword token: 39  const token: int constant: 2  keyword token: 37  ident token: f  keyword token: 49  ident token: x  keyword token: 50  keyword token: 36  const token: real constant: 1.5  keyword token: 47  keyword token: 14  keyword token: 49  ident token: a  keyword token: 52  ident token: b  keyword token: 0  ident token: b | keyword token: 51  ident token: c  keyword token: 50  keyword token: 28  keyword token: 2  ident token: x1  keyword token: 54  ident token: map  keyword token: 35  const token: real constant: 1.1  keyword token: 47  keyword token: 9  keyword token: 47  ident token: d  keyword token: 54  keyword token: 47  keyword token: 9  keyword token: 44  ident token: efault  ERROR: 2(15, 30)// Ошибка с кодом 2 – переполнение числа |

Рисунок 4.2 – Результат работы лексического анализатор