Лексема – это какая-то отдельная фраза из языка, ключевое слово, отдельный символ, с которым работает автомат.

Форма Грейбах отличается от обычной грамматики отличается тем, что он используется в генераторе и в начале правил всегда идет лексема.

В ***main*** мы изначально запрашиваем источник: имя файла, из которого будем читать. Далее мы полностью прочитываем файл: мы открываем поток для чтения, создаем переменную с именем *program\_text* и читаем файл до конца через *getline* (маркер до конца ‘\0’).

Программа кидает исключения. Если в программе случается какая-то ошибка: неправильно сама программа задана, встретилась какая-то неизвестная лексема или случилось что-то еще – все это уходит в блок *catch* в форме *exception*.

Далее создаем экземпляр анализатора, который принимает на вход текст программы и запускаем его. Потом создаем генератор, который принимает в себя лексемы, которые возвращает анализатор и запускаем его. Потом создаем экземпляр интерпретатора, который принимает таблицы переменных и ОПС, которые возвращает генератор и запускаем его.

\*показываю *ParserGenerator.h*\* показываем, что у него есть функция *run*, функция *getdata* и конструктор.

*InterpreteException*. В нем мы берем стандартное исключение и наследуем от него структуру – наше исключение. Мы создаем структуру *debugInfo*, которая хранит строку и позицию в этой строке для того, чтобы определить, в каком месте программа работает некорректно. Исключение хранит в себе какую-то строчку и у него есть метод *what*, который ее возвращает. И добавлен конструктор, который принимает строку с исключением и *debugInfo*. Дальше он их просто компанует. Файл *InterpreteException.h* в принципе можно не показывать.

Объяснение Костюку:

Начинаем с **лексического анализатора**. Вот у нас есть первая структура – типы лексем. Служебные слова, *varname* (имя переменной), *intnum*, *floatnum*, скобки разные и так далее. Еще есть финишная и ошибочная лексемы. Их нет в списке, но они нужны в программе просто для реализации.

У нас нет унарного минуса. Понять это можно только по грамматике, так как минус в лексеме – просто минус.

Смысл лексического анализатора – он просто проверяет, что мы что-то ввели и переводит набор символов в лексемы. Например, не удобно по букве считывать название переменной или служебное слово.

Дальше идет структура лексемы, которая содержит тип и какое-то значение. Программа, когда читает исходный файл, она записывает в *value* все, что читает. Для дебага, например, и для переменных или для чисел. Если мы читаем какое-то число, нам мало знать, что это число, и нужно знать его значение. То же самое для переменной.

Дальше класс ***Analyzer***, который анализирует входной текст и возвращает лексемы. У него есть метод *Run*, который есть у всех структур генератора, интерпретатора и анализатора. *GetData* тоже есть у всех, кроме интерпретатора и конструктор. Конструктор принимает строку, которая затем перемещается в приватное поле *program\_text*. Дальше мы поддерживаем текущий индекс и *debugInfo*. И есть *data* – вектор лексем. Мы не знаем, сколько их всего, поэтому удобно использовать динамический массив.

Метод ***Run***. Мы хотели прочитать все лексемы. Мы делаем цикл: пробегаемся по всему тексту программы. Дальше, пока пробельный символ: если просто пробел, увеличиваем позицию; если перевод строки, увеличиваем строку и позицию считаем первой. Если табуляция – 4 пробела. Дальше смотрим, если встретили # - символ, который комментирует строку, и мы не дочитали файл до конца – мы увеличиваем строку на 1 (комментируем строку до конца). Мы перейдем на 1 позицию в следующей строке. Теперь нам нужно сменить индекс позиции. То есть, когда мы встретили # - читаем файл до перевода строки, никак его не анализируя. Если закончился текущий индекс, т.е. программа закончилась кучей пробелов – делаем break.   
Основные действия: текущую лексему мы присваиваем nextlexeme, который считает, что у нас начинается лексема. Он нам возвращает лексему, записываем в нее info и смотрим: если лексема имеет тип error, то мы кидаем исключение – ошибка в анализаторе, неизвестная лексема, и говорим ее value. Лексемы он забракует, если пишем числа и потом буквы, или любой символ не лексему.

Метод ***NextLexeme*** – вызывается, когда мы обработали все комментарии и пробелы и стоит символ, который должен быть лексемой. Мы его читаем, создаем переменную *result*, в которую будем возвращать. У нее изначальный тип *error* и value *curchar*. Если встретили букву, то тип лексемы будет переменная (*varname*). Цикл будет до упора читать буквы цифры и увеличивать *value*. Дальше проверки на служебные слова – если value *define* – то *define*.

Читаем цифры – по дефолту *int*, читаем цифры, если встречаем букву – ошибка. Если точка, то *floatnum*. Если после вещественного числа буквы – ошибка.

Теперь рассматриваем генератор ***ParserGenerator***. Вход генератора – лексемы (*input\_data*). Магазин – стек (в обратном порядке), состоящий из *magazine items*. Дальше правила перехода – КС грамматика языка. Дальше generator – стек, состоящий из *generator* *tasks*. И ОПС (*interpretData*) – массив *OpsItem*. *Map* – контейнер, как и вектор, который содержит в себе пары ключ-значение (название переменной – значение переменной). Пример: « int\_table[var\_name] = 5; » или «arrayint\_table[var\_name][3] = 5;».   
На каждом генераторе выполняется какой-то шаг(*task*) генератора (из стека *generator*). В магазине и генераторе всегда одинаковое количество элементов. Для поддержания этого правила есть пустые действия, а в языке есть пустой нетерминал – *Z* – его аналог. Смысл генератора – сгенерировать ОПС.

ОПС – это бесскобочная запись (еще с метками).

Task1-Task11:

Первые 5 тасков из лекции Костюка. Они нужны для расставления меток в ОПС. Следующие 4 запускаются, когда мы встречаем int, float, massint и massfloat. Они меняют таблицу. Таск 10 меняет переменные int и float, таск 11 меняет massint и massfloat.

Тип переменной определяется в генераторе. Он заглядывает в таблицы и там ищет переменную.

DoTask смотрит текущий таск и empty, то ничего не делает, если переменная – смотрит лексему, значение, берет имя, смотрит таблицы, берет тип.