**Дерево вывода**

<var> ::= <letter> | <var> <letter> | <var> <digit>

<digit> ::= 0|1|..|9

<letter> ::= a|A|b|B|...|z|Z

Example:

**a2b**

Порождающий:

дерево вывода спускается сверху вниз (от грамматики к входящей строке)

Распознающий: дерево поднимается вверх (от строки к грамматике)

**Проблема возвращения (**нисходящий разбор) - если по одной альтернативе не удалось вывести что-либо, надо возвращаться к другой альтернативе.

Чтобы избежать возвратов, вводится левая факторизация грамматики.

Похожие головы выносятся за скобки (как в <var>, одинаковое начало c <var> во втором и третьем правиле). Если вынесем за скобки получим:

<var> ::= <letter> | <var> (<digit> | <letter>)

эквивалентно

<var> ::= <letter> | <var> <digit> | <var> <letter>

**Зацикливание (леворекурсивные правила)**

Надо заменить все леворекурсивные правила на итеративные правила

<var> ::= <letter> { <digit> | <letter> }

эквивалентно

<var> ::= <letter> | <var> ( <digit> | <letter> )

**Примеры:**

**S ::= SSD | SSK | SK | zz | z | zzz**

S ::= S ( SD | SK | K ) | z ( z | zz | ^)

S ::= S ( S ( D | K ) | K ) | z ( z ( z | ^ ) | ^)

S ::= z ( z ( z | ^ ) | ^ ) **{ S ( D | K ) | K ) }**

**----**

**A ::= KD | KDD | Aab | Aba**

A ::= KD ( D | ^ ) | A( ab | ba )

A ::= KD (D | ^ ) **{ ab | ba }**

**Метод рекурсивного спуска**

Работает с расширенной нотацией грамматики.

Каждому нетерминалу ставится в соответствие строго одна подпрограмма, которая разбирает строку, начинающийся с этого нетерминала (строит часть дерева вывода, начиная с данного нетерминала).

Пример:

bool programa(int counter = 0)

{

if (SpOg(counter))

{

if (Lexemes[counter].Value.Equals("{")

{

++counter;

if (SpOp(counter))

{

if (Lexemes[counter].Equals("}");

{

counter++;

return true;

}

throw new Exception();

}

}

}

}

//Lexemes - твой жсон с лексемами