Практическое занятие №2

Густов Владимир Владимирович
gutstuf@gmail.com

Цитата дня: Слишком рано, чтобы просыпаться, слишком поздно, чтобы спать. (с) Егор Летов

Курсовик

- Команды?
- Варианты?
- Части: лексический, синтаксический, генератор кода.

Repeat it

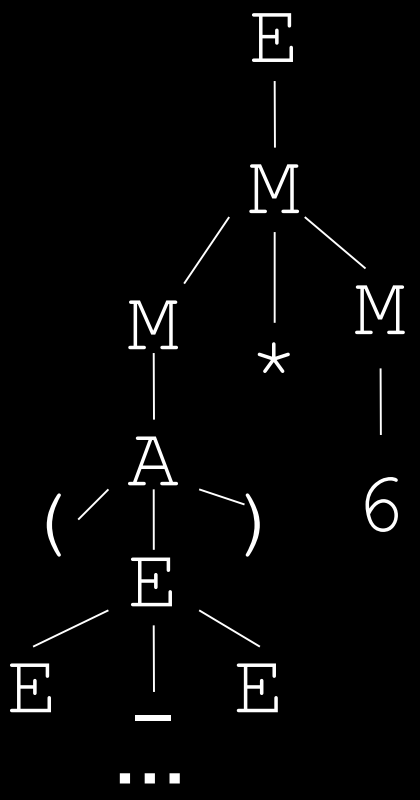
```
<Exp> ::= <Ter> | <Exp> + <Ter> | <Exp> - <Ter>
<Ter> ::= <Mul> | <Ter> * <Mul> | <Ter> / <Mul>
<Mul> ::= Id | ( <Exp> )
```

1) `4 / 2 * 5;`
2) `4 / 2 + 5 * 4;`
3) `4 + 2 * 5 + 4;`
4) `4 + 2 * (5 + 4);`

(4-2-5)*6

неоднозначность

```
E -> E + E | E - E | M
M -> M * M | M / M | A
A -> id | num | ( E )
```

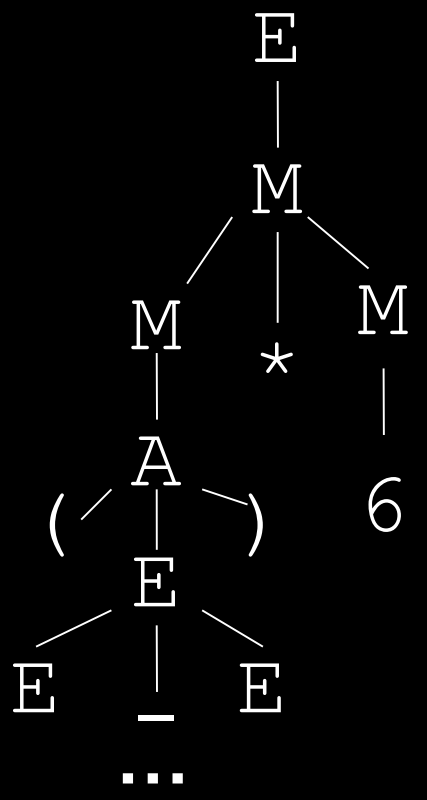


(левая) рекурсия

(4-2-5)*6

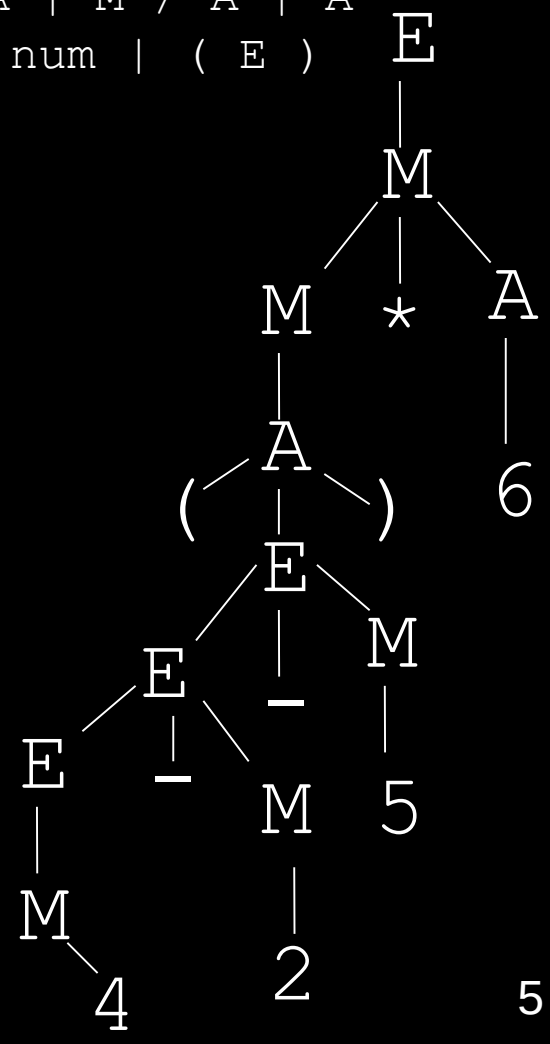
неоднозначность

```
E -> E + E | E - E | M
M -> M * M | M / M | A
A -> id | num | ( E )
```



(левая) рекурсия

```
E -> E + M | E - M | M
M -> M * A | M / A | A
A -> id | num | ( E )
```



Было:

```
E ::= E + M | E - M | M
M ::= M * A | M / A | A
A ::= var | num | (E)
```

Стало:

```
E  ::=   M E'
E' ::= + M E' | - M E' | empty
M  ::=   A M'
M' ::= * A M' | / A M' | empty
A  ::= var | num | (E)
```

Нисходящий/Восходящий анализ

Нисходящий анализ – производит просмотр входной цепочки символов слева направо и генерирует левосторонний вывод. Строит дерево вывода от корня к листьям. Используется алгоритм с подбором альтернатив.

Восходящий анализ – производит просмотр входной цепочки слева направо и генерацию правостороннего вывода. Строит дерево вывода от листьев к корню. Используется алгоритм сдвиг-свёртка.

Нисходящий анализ

Последовательное раскрытие нетерминалов начиная со стартового символа грамматики с подменой продукций. Чаще всего проход реализуется с помощью рекурсивных вызовов, либо конечного автомата.

Восходящий анализ

Операция сдвига сохраняет значение терминального символа в стеке парсера и запрашивает следующий токен.

Операция свёртки извлекает из стека парсера 1 или несколько символов находящихся в продукции справа и заменяет их продукцией слева.

[3] + 2 - 4

```
E ::= E + A | E - A | A
A ::= var | num | (E)
```

Входной токен: 3

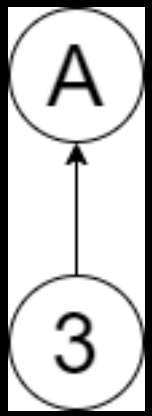
Стек парсера: пуст

Для упращения: num – это терминал, включает все числа от 0 до 9

3 [+] 2 - 4

Проверяем вершину стека и исходя из грамматики производим свёртку: извлекаем токен num из вершины стека, сравниваем с грамматикой и помещаем в вершину продукцию A.

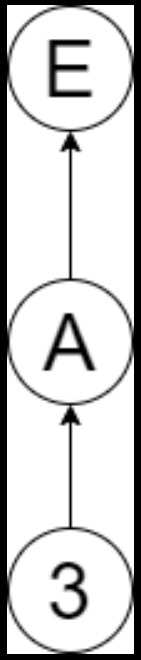
```
A ::= var | num | (E)
```



Входной токен: +

Стек парсера: A

3 [+] 2 - 4

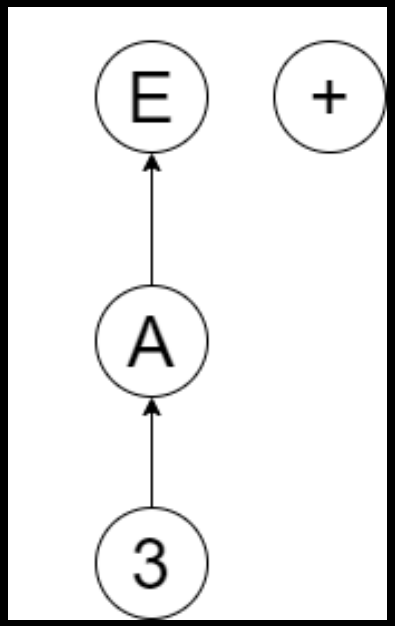


Входной токен: +

Стек парсера: E

3 + [2] - 4

Производим операцию сдвига – добавляем входящий токен в стек.

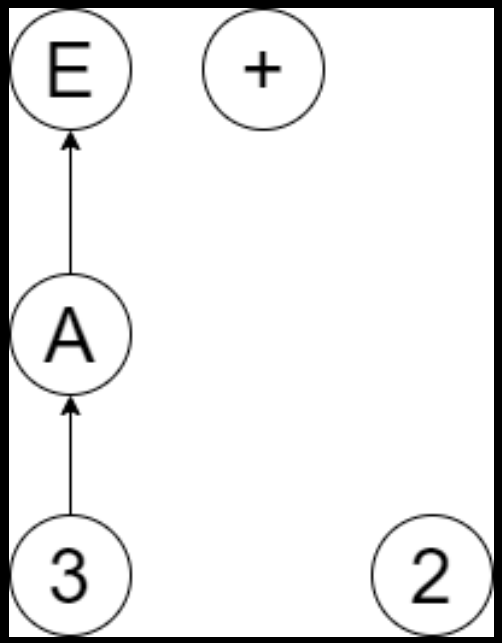


Входной токен: 2

Стек парсера: E +

3 + 2 [-] 4

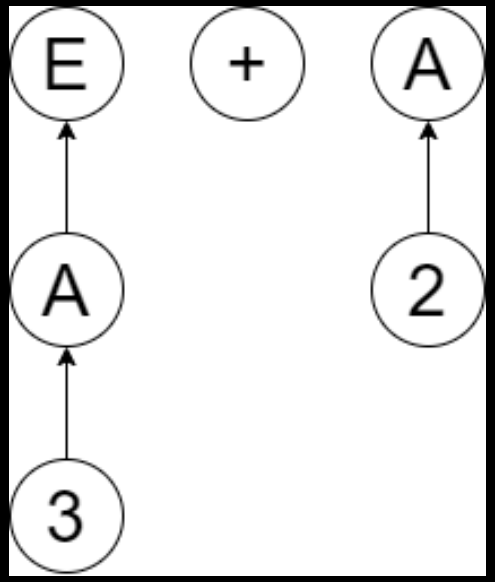
Т.к. текущее значение стека соответствует середине продукции **E ::= E + A**, то мы выполняем ещё одну операцию сдвига.



Входной токен: -

Стек парсера: E + 2

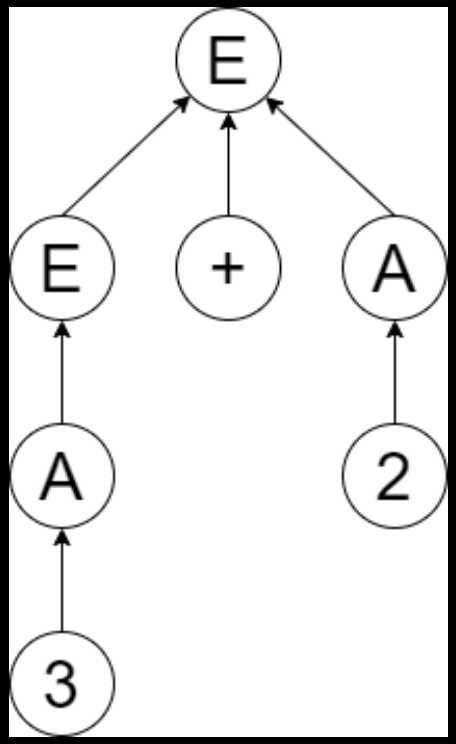
3 + 2 [-] 4



Входной токен: -

Стек парсера: E + A

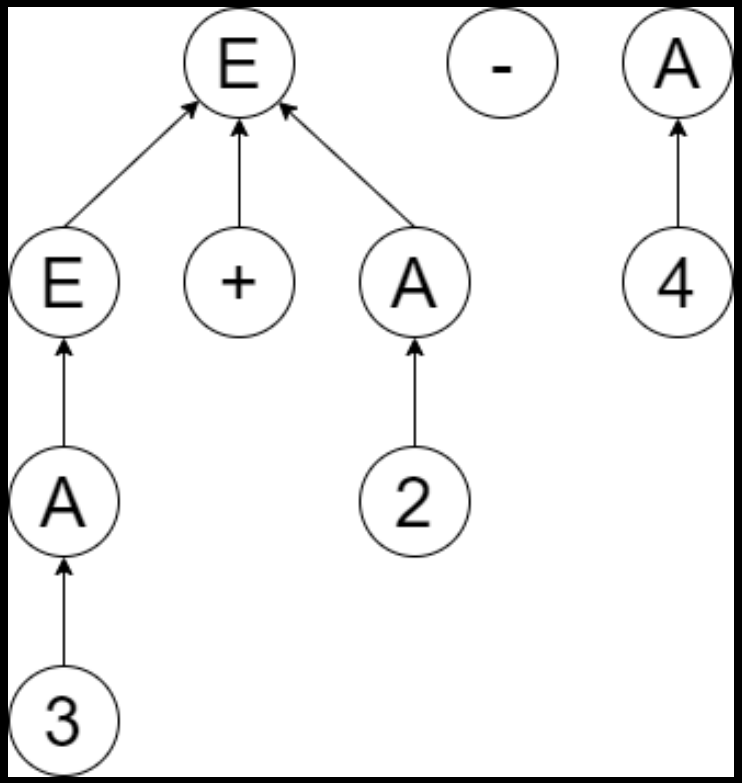
3 + 2 [-] 4



Входной токен: -

Стек парсера: E

3 + 2 – 4

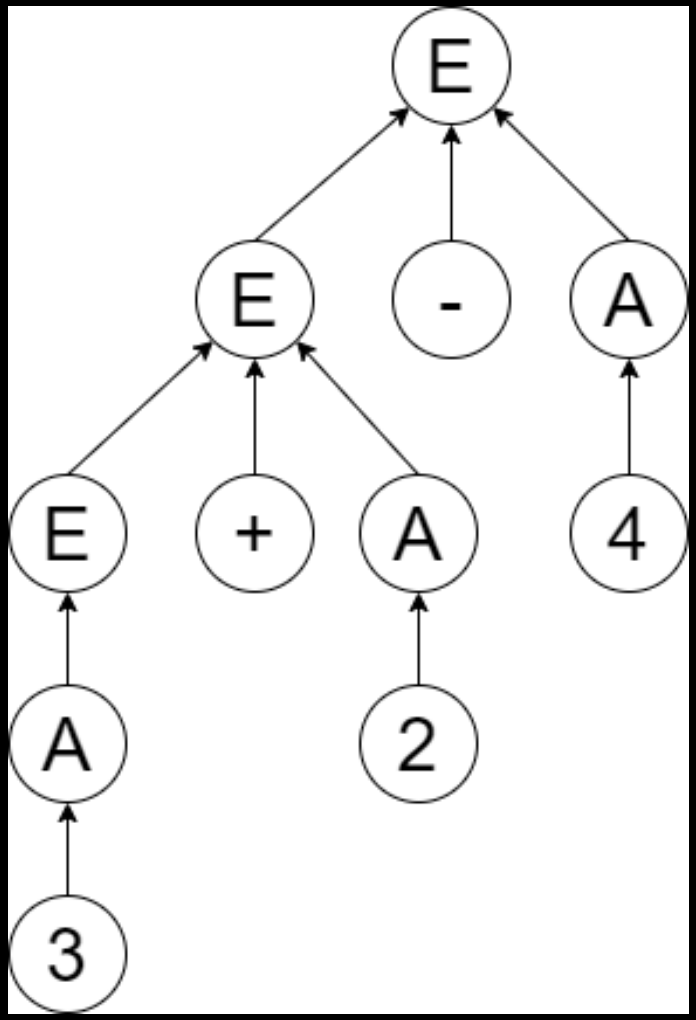


Входной токен: EOF

Стек парсера: E - A

3 + 2 – 4

```
E ::= E + A | E - A | A
A ::= var | num | (E)
```



Входной токен: EOF

Стек парсера: E

LL/LR парсер

LL(k)-грамматика	LR(k)-грамматика
L - Разбор слева на право L - Левосторонний вывод	L - Разбор слева на право R - Правосторонний вывод
- Обход дерева сверху вниз (top-down) от корня к листьям	- Обход дерева снизу вверху (bottom-up) от листьев к корню
- Разбор производится посредством «предсказания» и сопоставления продукции	- Разбор производится посредством сдвига и «уменьшения» продукции
- Необходимо избегать/избавляться от левой рекурсии	- Леворекурсивна сама по себе, не подходит для нисходящего анализа
- k – количество лексем необходимых для определения продукции	
- Необходимо избегать неоднозначности (возможности постройки нескольких деревьев)	

Get/Peek/Match

Get – последовательно читает символы из потока, сдвигает курсор, формирует лексемы (токены).

Peek – получает из потока лексему без сдвига курсора (позволяет «заглянуть вперёд»).

Match – сопоставляет полученный токен с правилами грамматики и выполняет соответствующую продукцию.

Ассоциативность арифметических операций

Ассоциативность определяет, какая часть выражения должна быть вычислена первой.
Например, в зависимости от ассоциативности оператора «–», для выражения 5 – 2 – 1 мы можем получить как 2, так и 4.

Ассоциативность операторов выражается с помощью правил грамматики (очерёдности продукции).

Все операторы с двумя операндами (кроме оператора присваивания) – левоассоциативные, т.е. они обрабатывают выражения слева направо.

Ассоциативность арифметических операций

a – a – a

```
E ::= M - E
M ::= a
```

```
E ::= E - M
M ::= a
```

```
E ::= TM
M ::= -TM | e
T ::= a
```

Ассоциативность арифметических операций

a – a – a

```
E ::= M - E
M ::= a
```

+ нет левой рекурсии
- правоассоциативна

```
E ::= E - M
M ::= a
```

+ левоассоциативна
- есть левая рекурсия

```
E ::= TM
M ::= -TM | e
T ::= a
```

+ левоассоциативна
+ нет левой рекурсии
- больше продукций

Старшинство операторов

Старшинство операторов определяется очерёдностью их объявления в продукциях грамматики. Чем «старше» оператор, тем ниже/позже он объявляется в грамматике.

Т.е. для того, чтобы указать старшинство оператора «*», над оператором «+», необходимо, чтобы в грамматике оператор «+» предшествовал оператору «*». Например:

```
(1) <E> ::= <E> + <T>
(2) <T> ::= <M> * <T>
(3) <M> ::= num | var
```

Продукция (1) указывает, что <T>, содержащий умножение, расположен справа от оператора “+” и должен вычисляться до выполнения “+”.

Ошибки

Лексические: неверно записанные идентификаторы, ключевые слова;

Синтаксические: ошибки в арифм. выражениях (несбалансированные скобки), ошибки конструкций;

Семантические: несоответствие типов, необъявленная переменная, использование зарезервированного идентификатора;

Логические: бесконечная рекурсия, ошибки алгоритмов;

Восстановление

Режим паники: считываем все последующие токены до тех пор, пока не дойдём до «синхронизирующего» токена, т.е. пропускаем все лексемы

Уровень фразы: производится локальная корректировка, т.е. добавляется / заменяется синхронизирующий токен (,/;)

Ошибка производства: расширение грамматики включая в неё распространённые ошибочные конструкции.

<program> ::= ***program*** <identifier> ; <block> .

<identifier> ::= <letter > {<letter or digit>}

<block> ::= <variable declaration part> <statement part>

<variable declaration part> ::= <empty> | ***var*** <variable declaration> {; <variable declaration>} ;

<variable declaration> ::= <identifier> {,<identifier>} : <type>

<statement part> ::= <compound statement>

<compound statement> ::= ***begin*** <statement> {; <statement> } ***end;***

1

```
program 1codechef;
begin
end.
```

2

```
Program codechef;
Begin
end.
```

3

```
PrAgram codechef;
Begin
end.
```

4

```
program codechef;
  var a : bool;
begin
  a := 0;
end.
```

5

```
program codechef;
  var a : integer;
begin
  a := false;
end.
```

https://www.codechef.com/ide

Compile time

```
program hmm;
begin
     var a : integer;
     var c, d, e : integer;
     a := 2;
     d := 3;

     if a <> d then
        writeln(a);
    else
        writeln(d);
end.
```

Ссылки

- грамматика Pascal в БНФ [1];
- грамматика Pascal в БНФ [2];
- ассоциативность операций [1];
- ассоциативность операций [2];
- LR парсер (видео);