

# Методы трансляции

Михайлов Максим

2 января 2022 г.

## Оглавление

Лекция 1	2 сентября	2
1	Введение	2
2	$LL(k)$ , FIRST, FOLLOW	4
Лекция 2	9 сентября	6
2.1	Вычисление FIRST	6
2.2	Вычисление FOLLOW	7
2.3	Доказательство теоремы о характеристизации $LL(1)$	7
2.4	Проблемы грамматик	8
2.4.1	Левая рекурсия	8
2.4.2	Правое ветвление	8
2.4.3	Пример	9
3	Построение парсеров	9
Лекция 3	16 сентября	11

# Лекция 1

## 2 сентября

### 1 Введение

Этот курс — про парсеры. Рассмотрим их работу в общем случае.

1. На вход подается строка.
2. Строка разбивается на неделимые блоки (*лексемы или токены*) лексическим анализом.
3. Последовательность токенов с учетом синтаксиса языка переводится в дерево разбора путем синтаксического разбора (*парсинга*).
4. Дерево разбора не есть самоцель, дерево переводится с учетом семантики языка в искомый результат.

Адепты *architecture-driven* подхода могут захотеть разделить семантику и синтаксис, однако это проблематично. Рассмотрим арифметические выражения как пример.

Токены арифметических выражений это  $+$ ,  $\cdot$ ,  $($ ,  $)$ ,  $n$ , где  $n$  — число. Синтаксис задается следующей контекстно-свободной грамматикой:

- $E \rightarrow n$
- $E \rightarrow (E)$
- $E \rightarrow E + E$
- $E \rightarrow E \cdot E$

Однако, эта грамматика не однозначна, и выражение  $2 + 2 \cdot 2$  можно разобрать по-разному, из-за чего невозможно навесить семантику. Таким образом, синтаксис нужно задавать с учетом семантики:

- $E \rightarrow T$

- $E \rightarrow T + E$
- $T \rightarrow F$
- $T \rightarrow F \cdot T$
- $F \rightarrow n$
- $F \rightarrow (E)$

Но с такой грамматикой операции правоассоциативные и семантику не получится навесить с добавлением вычитания. В правильной грамматике нужно переставить местами аргументы второго правила.

Рассмотрим, как мы будем писать калькулятор арифметических выражений по дереву разбора. Наивный подход — обойти дерево DFS-ом и рассматривать детей вершины, в которой мы находимся. Однако, таким образом информация о синтаксисе описывается в двух сущностях — в парсере и в калькуляторе. Это неудобно, поэтому часто парсинг и вычисления комбинируются в один шаг без построения дерева разбора. На примере арифметических выражений:

- $E_0.val = T.val$
- $E_0.val = E_1.val + T.val$
- $E_0.val = E_1.val - T.val$
- $\vdots$

Такой подход называется **синтаксически управляемая трансляция**.

Итого существуют четыре подхода дизайну систем парсинга в зависимости от сложности задачи:

1. Ad hoc: без теории, наивно.
2. Parser + walker: Парсер производит дерево разбора и walker его обходит.
3. Синтаксически управляемая трансляция.
4. Декомпозиция задач.

Этот курс рассматривает второй и третий подходы.

Рассмотрим пример калькулятора арифметических выражений:

```
int expr():  
    r = term()  
    nexttoken()  
    while token == '+':  
        nexttoken()  
        t = term()
```

```
    r += t

int term():
    r = factor()
    nexttoken()
    while token == '*':
        nexttoken()
        f = factor()
        r += f

int factor():
    if token == '(':
        nexttoken()
        r = expr()
        assert token == ')'
        nexttoken()
    else # token == 'n'
        r = tokenval()
        nexttoken()
```

Какая связь между этим кодом и грамматикой арифметических выражений? Оказывается, весьма близкая и код можно получить из нее.

## 2 $LL(k)$ , FIRST, FOLLOW

**Определение** (контекстно-свободная грамматика).

- Алфавит  $\Sigma$  — множество токенов
- Нетерминалы  $N$
- Стартовый нетерминал  $S \in N$
- Правила  $P \subset N \times (N \cup \Sigma)^*$

**Определение.**  $\langle A, \alpha \rangle \in P \Leftrightarrow A \rightarrow \alpha$

**Определение.**  $\alpha \Rightarrow \beta$  — из  $\alpha$  выводится за один шаг  $\beta$ , если:

- $\alpha = \alpha_1 A \alpha_2$
- $\beta = \alpha_1 \xi \alpha_2$
- $A \rightarrow \xi \in P$

**Определение** (язык грамматики).  $L(\Gamma) = \{x \mid S \Rightarrow^* x\}$ ,  $x \in \Sigma^*$ , где  $\Rightarrow^*$  есть замыкание отношения  $\Rightarrow$ .

**Определение.** Грамматика **однозначна**, если для любого слова из языка есть только одно дерево разбора и **неоднозначна** иначе.

*Примечание.* Здесь и далее буквы из конца латинского алфавита обозначают нетерминалы, а буквы греческого алфавита — строки из терминалов и/или нетерминалов.

**Определение.**  $\Gamma \in LL(1)$ , если из выполнения следующих двух условий:

- $S \Rightarrow^* xA\alpha \Rightarrow x\xi\alpha \Rightarrow^* xcy$
- $S \Rightarrow^* xA\beta \Rightarrow x\eta\beta \Rightarrow^* xcz$

следует  $c \in \Sigma$ , или  $c = \varepsilon$ , или  $y = \varepsilon$ , или  $z = \varepsilon$ , тогда  $\xi = \eta$ .

**Определение.**  $\Gamma \in LL(k)$ , если из выполнения следующих двух условий:

- $S \Rightarrow^* xA\alpha \Rightarrow x\xi\alpha \Rightarrow^* xcy$
- $S \Rightarrow^* xA\beta \Rightarrow x\eta\beta \Rightarrow^* xcz$

следует  $c \in \Sigma^k$ , или  $c \in \Sigma^{\leq k}$ , или  $y = \varepsilon$ , или  $z = \varepsilon$ , тогда  $\xi = \eta$ .

В частности,  $LL(0)$  — линейные программы.

$LL(1)$  грамматики есть класс всех грамматик, которые можно разобрать рекурсивным спуском.

Определение  $LL(1)$  грамматик не конструктивно, т.к. проверка определения может длиться бесконечно (*по количеству всех выводов*). Определим конструктивный критерий принадлежности  $LL(1)$ , для этого мы рассмотрим две вспомогательные функции:

- FIRST:  $(N \cup \Sigma)^* \rightarrow 2^{\Sigma \cup \{\varepsilon\}}$
- FOLLOW:  $N \rightarrow 2^{\Sigma \cup \{\$ \}}$

$$\text{FIRST}(\alpha) := \{c \mid \alpha \Rightarrow^* c\beta\} \cup \{\varepsilon \mid \alpha \Rightarrow^* \varepsilon\}$$

$$\text{FOLLOW}(A) := \{c \mid S \Rightarrow^* \alpha A c \beta\} \cup \{\$ \mid S \Rightarrow^* \alpha A\}$$

*Примечание.* Мы считаем, что в грамматике нет нетерминалов, из которых нельзя вывести строку из терминалов. Это допущение не теряет общности, т.к. существует алгоритм удаления “бесполезных” нетерминалов, см. курс дискретной математики.

**Теорема 1.**  $\Gamma \in LL(1) \Leftrightarrow \forall A \rightarrow \alpha, A \rightarrow \beta$ :

1.  $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
2.  $\varepsilon \in \text{FIRST}(\alpha) \Rightarrow \text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \emptyset$

# Лекция 2

## 9 сентября

### 2.1 Вычисление FIRST

Определим массив (или *map*)  $\text{FIRST}[] : N \rightarrow 2^{\Sigma \cup \{\varepsilon\}}$ , который будет возвращать FIRST от нетерминалов.

**Лемма 1.**

- $\alpha = c\beta \Rightarrow \text{FIRST}(\alpha) = \{c\}$
- $\alpha = \varepsilon \Rightarrow \text{FIRST}(\alpha) = \{\varepsilon\}$
- $\alpha = A\beta \Rightarrow \text{FIRST}(\alpha) = \text{FIRST}[A] \setminus \varepsilon \cup (\text{FIRST}(\beta) \text{ if } \varepsilon \in \text{FIRST}[A])$

*Доказательство.* Очевидно. □

По лемме мы можем найти  $\text{FIRST}[]$  следующим алгоритмом:

```
while (change):
    for  $A \rightarrow \alpha \in \Gamma$ :
         $\text{FIRST}[A] \cup= \text{FIRST}(\alpha)$ 
```

Докажем, что итоговый массив  $\text{FIRST}[]$  как функция от  $N$  равен функции FIRST.

*Доказательство.* Очевидно, что  $\text{FIRST}[A] \subset \text{FIRST}(A)$ , т.к. мы не добавляем лишнего (*по лемме*).

Докажем, что  $\text{FIRST}[A] \supset \text{FIRST}(A)$  от противного — пусть  $\exists c : c \in \text{FIRST}(A), c \notin \text{FIRST}[A]$ . Среди всех таких  $c$  найдем такое, что вывод  $A \Rightarrow^k c\xi$  имеет минимальную длину, т.е.  $k \rightarrow \min$ . Можем расписать  $A \Rightarrow^k c\xi$  как  $A \Rightarrow \alpha \Rightarrow^{k-1} c\xi$  для некоторого  $\alpha$ .

Рассмотрим структуру  $\alpha$ . Это некоторая строка  $x_1x_2 \dots x_k$ , при этом все символы с  $x_1 \dots x_{i-1}$ <sup>1</sup>

<sup>1</sup> Эта последовательность может быть пустой.

породили пустые строки, и  $x_i$  породил строку, начинающуюся с  $c$ . Т.к.  $k \rightarrow \min$ , то  $c \in \text{FIRST}[x_i]$ , т.к.  $c \in \text{FIRST}(x_i)$ . Но тогда на последней итерации алгоритма, когда рассматривается правило  $A \rightarrow \alpha$ , в  $\text{FIRST}[A]$  должно было добавиться  $\text{FIRST}(A)$ , в котором лежит  $c$ . Противоречие.  $\square$

По массиву  $\text{FIRST}[]$  можно восстановить  $\text{FIRST}(\alpha)$  для любого  $\alpha$  по лемме.

## 2.2 Вычисление FOLLOW

```

FOLLOW[S] = {$}
while (change):
    for  $A \rightarrow \alpha \in \Gamma$ :
        for  $B : \alpha = \beta B \gamma$ :
            FOLLOW(B)  $\cup=$   $\text{FIRST}(\gamma) \setminus \varepsilon \cup (\text{FOLLOW}(A) \text{ if } \varepsilon \in \text{FIRST}(\gamma))$ 

```

*Доказательство.* Аналогично FIRST.  $\square$

*Пример.* Вспомним грамматику арифметических выражений с прошлой лекции:

- $E \rightarrow E + T$
- $E \rightarrow T$
- $T \rightarrow T * F$
- $T \rightarrow F$
- $F \rightarrow (E)$
- $F \rightarrow n$

	FIRST	FOLLOW
$E$	$(n$	$\$ + )$
$T$	$(n$	$\$ + * )$
$F$	$(n$	$\$ + * )$

Для правил  $E \rightarrow E + T$ ,  $E \rightarrow T$  множества FIRST от правых частей пересекаются, следовательно эта грамматика не  $LL(1)$ . Это вызвано простой проблемой — эти два правила образуют левую рекурсию и очевидно условие 1 теоремы не выполнено.

## 2.3 Доказательство теоремы о характеристизации $LL(1)$

*Доказательство.* Предположим, что  $\Gamma \notin LL(1)$ .

1.  $\xi$  не породил  $\varepsilon$ ,  $\eta$  не породил  $\varepsilon$

Тогда  $c \in \text{FIRST}(\xi)$  и  $c \in \text{FIRST}(\eta)$

2.  $\xi$  породил  $\varepsilon$ ,  $\eta$  породил  $\varepsilon$

Тогда  $\varepsilon \in \text{FIRST}(\xi)$  и  $\varepsilon \in \text{FIRST}(\eta)$

3.  $\xi$  породил  $\varepsilon$ ,  $\eta$  не породил  $\varepsilon$

Тогда  $\varepsilon \in \text{FIRST}(\xi)$ ,  $c \in \text{FOLLOW}(A)$  и  $c \in \text{FIRST}(\eta)$

Таким образом, если  $\Gamma \notin LL(1)$ , то условие теоремы не выполнено.

В обратную сторону доказательство не написано. □

## 2.4 Проблемы грамматик

В этой части мы обсудим типичные причины, по которым грамматика может быть  $\notin LL(1)$ .

### 2.4.1 Левая рекурсия

**Определение.**  $A \Rightarrow^+ A\alpha$  — левая рекурсия

*Утверждение.* Левая рекурсия  $\notin LL(1)$ . (почти всегда)

*Доказательство.* Рассмотрим непосредственную левую рекурсию:  $A \rightarrow A\alpha$ . Пусть ещё есть правило  $A \rightarrow \beta$ .<sup>2</sup> Рассмотрим  $c \in \text{FIRST}(\beta)$ , тогда ещё  $c \in \text{FIRST}(A\alpha)$ , следовательно  $\Gamma \notin LL(1)$ . Если же  $\nexists c \in \text{FIRST}(\beta)$  для любого  $\beta$ , то  $\Gamma \in LL(1)$ . □

От левой рекурсии можно избавиться следующим преобразованием для всех  $A \rightarrow A\alpha$ ,  $A \rightarrow \beta$ :

- $A \rightarrow \beta A'$
- $A' \rightarrow \alpha A'$
- $A' \rightarrow \varepsilon$

Никто не гарантирует, что после такого преобразования  $\Gamma' \in LL(1)$ , т.к. у грамматики могут быть другие проблемы.

### 2.4.2 Правое ветвление

**Определение** (правое ветвление).  $A \rightarrow \alpha\beta$ ,  $A \rightarrow \alpha\gamma$

*Утверждение.* Правое ветвление  $\notin LL(1)$ . (почти всегда)

Преобразование, удаляющее правое ветвление:

- $A \rightarrow \alpha A'$

---

<sup>2</sup> Мы не теряем общности, т.к. иначе  $A$  — бесполезный нетерминал. Мы не рассматриваем грамматики с такими нетерминалами, т.к. их можно убрать.



- $A' \rightarrow \beta$
- $A' \rightarrow \gamma$

### 2.4.3 Пример

Преобразуем грамматику арифметических выражений:

- $E \rightarrow TE'$
- $E' \rightarrow +TE'$
- $E' \rightarrow \varepsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT'$
- $T' \rightarrow \varepsilon$
- $F \rightarrow (E)$
- $F \rightarrow n$

Посчитаем FIRST и FOLLOW:

	FIRST	FOLLOW
$E$	$n ($	$\$ )$
$E'$	$\varepsilon +$	$\$ )$
$T$	$n ($	$+ \$ )$
$T'$	$\varepsilon *$	$+ \$ )$
$F$	$n ($	$* + \$ )$

Ура, эта грамматика  $\in LL(1)$ !

## 3 Построение парсеров

Есть два метода построения деревьев разбора — сверху и снизу. Для  $LL$  грамматик используется сверху, для  $LR$  (определим позже) — снизу.

Для каждого нетерминала определим функцию, которая возвращает дерево разбора с корнем в этом нетерминале.

Также у нас есть контекст, в котором есть `token` — рассматриваемый токен и функция `nextToken()` — достает новый токен. Для каждой функции инвариант — при вызове функции  $f$  `token` есть нетерминал  $f$ .

*Пример.* Пусть есть правила  $A \rightarrow \alpha_1, A \rightarrow \alpha_2 \dots A \rightarrow \alpha_k$ .

$$\text{FIRST1}(\alpha) := \text{FIRST}(\alpha) \setminus \varepsilon \cup (\text{FOLLOW}(A) \text{ if } \varepsilon \in \text{FIRST}(\alpha))$$

A(): Tree

```
r = Tree(A)
switch (token)
  case FIRST1( $\alpha_1$ )
    f $\alpha_1$ 
  case FIRST1( $\alpha_2$ )
    f $\alpha_2$ 
  :
return r
```

, где  $f\alpha_i$  — блок обработки  $\alpha_i$ .

Пусть  $\alpha_i = X_1 X_2 \dots X_t$ . Тогда  $f\alpha_i$ :

```
f $\alpha_i$ :
:
//  $X_i = c$  --- terminal
assert  $X_i == \text{token}$ 
nextToken()
r.addChild( $X_i$ )

//  $X_j = A$  --- nonterminal
r.addChild( $X_j$ )
```

# Лекция 3

## 16 сентября

*Пример.* Напишем парсер для арифметических выражений.

```
E()
    res = Node(E)
    switch (token)
        case 'n':
        case 'c':
            res.add(T())
            res.add(Eprime())
    return res

Eprime()
    res = Node(Eprime)
    switch (token)
        case '+':
            assert(token == '+')
            res.add(Node(+))
            nextToken()
            res.add(T())
            res.add(Eprime())
        case '$':
        case ')':
            pass
    return res

T()
    res = Node(T)
    switch (token)
        case 'n':
```

```

    assert(token == 'n')
    res.add(Node(n))
    nextToken()
  case '(':
    assert(token == '(')
    res.add(Node('('))
    nextToken()
    res.add(E())
    assert(token == ')')
    res.add(Node(''))
    nextToken()
  return res

```

Вспомним код, который мы сдавали Георгию Александровичу:

```

int E():
  res = T()
  while token == '+':
    nextToken()
    res += T()
  return res

```

Он мощнее, потому что написан для другой грамматики (без  $E'$ ), которая не  $LL(1)$ .

*Пример.*  $\langle A \rightarrow A\alpha, A \rightarrow \beta_1\beta_2\dots, \text{после устранения левой рекурсии мы получим } A \rightarrow \beta A', A' \rightarrow \alpha A', A' \rightarrow \varepsilon. \text{ Запишем рекурсивный спуск для } A \text{ без спуска для } A':$

```

A()
  res = Node(A)
  switch (token)
    case FIRST1( $\beta_1$ )
      ...
    case FIRST1( $\beta_2$ )
      ...
  cur = Node(Aprime)
  res.add(cur)
  while token  $\in$  FIRST1( $\alpha_i$ )
    switch (token)
      case FIRST( $\alpha_1$ )
        cur.add(...)
    next = Node(Aprime)
    cur.add(next)
    cur = next
  // assert token  $\in$  FOLLOW(A)

```

Таким образом, мы можем не устранять левую рекурсию в грамматике, а делать это при

написании парсера.

Писать парсеры так, как мы сейчас это делали, неудобно. Будем писать по-другому.

$\langle \Gamma \in LL(1)$ . Запишем информацию в таблицу, называемую **управляющей таблицей**:

*Обозначение.*

- $\rightarrow$  — nextToken()
- $\perp$  — дно стека.

Для каждого  $A \in N$  пусть есть правила  $A \rightarrow \alpha_i$ . Тогда для каждого  $t \in \text{FIRST1}(\alpha_i)$  впишем в ячейку  $(A, t)$  число  $i$ :

		$\Sigma$	$\$$
$N$	$A$	$\dots i \dots i \dots i \dots i \dots$	
	$\Sigma$	<div> <div> <math>\rightarrow</math> </div> <div>ошибка</div> </div> <div> <math>\rightarrow</math> </div> <div> <math>\ddots</math> </div> <div>ошибка</div> <div> <math>\rightarrow</math> </div>	ошибка
	$\perp$		ок

Как разборщик пользуется таблицей? На каждой итерации он смотрит на вершину стека<sup>1</sup> и на управляющую таблицу.

1. Если на вершине стека терминал:
  - (a) Если терминал совпал с токеном, то с вершины стека снимается терминал и происходит переход к следующему токеному.
  - (b) Если терминал не совпал с токеном, то ошибка.
2. Если на вершине стека нетерминал, то в таблице ищется номер правила грамматики, соответствующий данному нетерминалу и токеному.
  - (a) Если ячейка не пуста, то из вершины стека удаляется элемент и в стек кладется правая часть соответствующего правила.
  - (b) Если ячейка пуста, то ошибка.

*Пример* (арифметические выражения).

<sup>1</sup> В котором лежат разбираемые нетерминалы/терминалы.

Ввод Стек						
	$n$	+	*	(	)	\$
$E$	1			1		
$E'$		2			3	3
$T$	4			4		
$T'$		6	5		6	6
$F$	7			8		
$n$	$\rightarrow$					
+		$\rightarrow$				
*			$\rightarrow$			
(				$\rightarrow$		
)					$\rightarrow$	
$\perp$						OK