Методы трансляции

Михайлов Максим

3 ноября 2021 г.

Оглавление

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:

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

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:

- $E \to T$

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

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

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

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

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

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

$$\text{FIRST}(\alpha) \coloneqq \{c \mid \alpha \Rightarrow^* c\beta\} \cup \{\varepsilon \mid \alpha \Rightarrow^* \varepsilon\}$$
$$\text{FOLLOW}(A) \coloneqq \{c \mid S \Rightarrow^* \alpha Ac\beta\} \cup \{\$ \mid S \Rightarrow^* \alpha A\}$$

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

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

Лекция 2

9 сентября

2.1 Вычисление FIRST

Определим массив *(или map)* FIRST[] : $N \to 2^{\Sigma \cup \{\varepsilon\}}$, 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 FIRST[] следующим алгоритмом:

```
while (change):
    for A → α ∈ Γ:
        FIRST[A] ∪= FIRST(α)
```

Докажем, что итоговый массив 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. $k \to \min$. Можем расписать $A \Rightarrow^k c\xi$ как $A \Rightarrow \alpha \Rightarrow^{k-1} c\xi$ для некоторого α.

Рассмотрим структуру α. Это некоторая строка $x_1 x_2 \ldots x_k$, при этом все символы с $x_1 \ldots x_{i-1}$[1]

[1] Эта последовательность может быть пустой.

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

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

2.2 Вычисление FOLLOW

```
FOLLOW[S] = {$}
while (change):
    for A → α ∈ Γ:
        for B : α = βBγ:
            FOLLOW(B) ∪= FIRST(γ)\ε ∪ (FOLLOW(A) if ε ∈  FIRST(γ))
```

Доказательство. Аналогично FIRST. □

Пример. Вспомним грамматику арифметических выражений с прошлой лекции:

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

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

Для правил $E \to E + T$, $E \to T$ 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 характеризации $LL(1)$

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

1. ξ не породил ε, η не породил ε

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

2. ξ породил ε, η породил ε

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

3. ξ породил ε, η не породил ε

   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. $\square$

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: $A \to A\alpha$. Пусть ещё есть правило $A \to \beta$.[2] 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 β, то $\Gamma \in LL(1)$. $\square$

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

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

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). $A \to \alpha\beta, A \to \alpha\gamma$

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

Преобразование, удаляющее правое ветвление:

- $A \to \alpha A'$

[2] Мы не теряем общности, т.к. иначе A — бесполезный нетерминал. Мы не рассматриваем грамматики с такими нетерминалами, т.к. их можно убрать.

- $A' \to \beta$
- $A' \to \gamma$

2.4.3 Пример

Преобразуем грамматику арифметических выражений:

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

Посчитаем FIRST и FOLLOW:

	FIRST	FOLLOW
E	n (	\$)
E'	ε +	\$)
T	n (	+ \$)
T'	ε $*$	+ \$)
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 $A \to \alpha_1, A \to \alpha_2 \dots A \to \alpha_k$.

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

```
A(): Tree
    r = Tree(A)
    switch (token)
        case FIRST1(α₁)
            fα₁
        case FIRST1(α₂)
            fα₂
        ⋮
    return r
```

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

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

```
fαᵢ:
    ⋮
    // Xᵢ = c --- terminal
    assert Xᵢ == token
    nextToken()
    r.addChild(Xᵢ)

    // Xⱼ = A --- nonterminal
    r.addChild(Xⱼ)
```

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. $\lessdot A \to A\alpha, A \to \beta_1|\beta_2\dots$, после устранения левой рекурсии мы получим $A \to \beta A', A' \to \alpha A', A' \to \varepsilon$. Запишем рекурсивный спуск для A без спуска для A':

```
A()
    res = Node(A)
    switch (token)
        case FIRST1(β₁)
            ...
        case FIRST1(β₂)
            ...
        cur = Node(Aprime)
        res.add(cur)
        while token ∈ FIRST1(αᵢ)
            switch (token)
                case FIRST(α₁)
                    cur.add(...)
            next = Node(Aprime)
            cur.add(next)
            cur = next
        // assert token ∈ 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.

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

Обозначение.

- $\to$ — `nextToken()`
- $\perp$ — дно стека.

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

		Σ	$\$$
N	A	$\dots i \dots i \dots i \dots i \dots$	
Σ		$\begin{matrix} \to & & \text{ошибка} \\ & \to & \\ & & \ddots \\ \text{ошибка} & & \to \end{matrix}$	ошибка
$\perp$			ок

Как разборщик пользуется таблицей? На каждой итерации он смотрит на вершину стека[1] 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 токену.
   (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 токену.
   (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).

[1] 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