Translation Methods

> Лабораторные:
>
> 1. Perl
> 2. Ручное построение трансляторов
> 3. Использование автоматических генераторов трансляторов
>   **e.g.** ANTLR (java), Bison + Yacc (c++), Happy (haskell)
> 4. Написание автоматического генератора транслятора

$\Sigma, \Sigma^*, L \subset \Sigma^*$ - формальный язык

Базовый класс формальных языков - регулярные (= автоматные). Для порождения - регулярные выражения, для распознавания - конечные автоматы

Контекстно-свободные языки: КС-грамматики / МП-автоматы (магазинная память)

Токены (лексемы) - единые неделимые элементы языка ($\in \Sigma$)

Лексический анализ

Первый этап любого разбора - *лексический анализ*

Последовательность символов -> последовательность токенов ($\in \Sigma^*$)

> **e.g.** арифмитические выражения
>
> $\Sigma = \{n, +, \times, \ (, \ )\}$
>
> $(2 \ + \ 2) \times 2 \rightarrow (n + n) \times n$
>
> $n : (0|1|...|9)(0|1|...|9)^*$

жадный лексический анализ на базе регулярных выражений: пропускаем пробельные символы, смотрим первый непробельный, находим максимальный префикс какого-то возможного токена

---

1. Проверить, что строка выводится в грамматике Γ // алгоритм КЯК(???) $\mathrm{O}(n^3)$
2. Построить дерево разбора
3. Синтаксически управляемая трансляция

   > $E \rightarrow T$
   > $E \rightarrow E \ + \ T$
   > $T \rightarrow F$
   > $T \rightarrow T \ \times \ F$
   > $F \rightarrow n$
   > $F \rightarrow (E)$

   Аттрибутно-транслирующие грамматики - контекстно-свободные языки с добавлением двух элементов: аттрибуты и транслирующие символы

   Транслирующие символы - фрагменты кода, которые вставляем в грамматику, которые могут взаимодействовать с аттрибутами

   > $E \rightarrow E \ + \ T \ \{E_0.v = E_1.v \ + \ T.v\}$
   >
   > $T \rightarrow T \ \times \ F \ \{T_0.v = T_1.v \ + \ F.v\}$

Нужно быстрее, чем за куб $\Rightarrow$ накладываем ограничения на грамматики

Однозначность - если у любого слова не более одного дерева разбора в этой грамматике // Модификация алгоритма Эрли - $\mathrm{O}(n^2)$

LL, LR - грамматики, на которые наложены дополнительные ограничения, чтобы разбор работал за линейное время. **LL(R)** - **L**: left to right parse - обходим слово слева направо; **L(R)**: leftmost derivation (right most derivation) - левосторонний (правосторонний) вывод.

$\Gamma,\ w$ на вход

Можем строить дерево разбора сверзу вниз - **нисходящая трансляция**(used **LL**). Шаг называется *раскрытие нетерминала*. Нисходящий парсер :

> 1. Находим нетерминал,у которого неизвестно поддерево
> 2. Раскрываем его
>   В основном это самый левый нетерминал

Снизу вверх - **восходящий разбор** (used **LR**). Шаг - *свёртка*

> 1. Находим правую часть какого-то терминала
> 2. Сворачиваем ее
>   Получается правосторонний вывод слова (поэтому R)

Метод нисходящих трансляций для LL грамматик

LL(k) - грамматика

Def: Грамматика $\Gamma = \langle \Sigma, N, S, P \rangle$, где Σ - множество терминалов (terms), N - множество нетерминалов (nonterms), S - стартовый символ ($S \in N$), P - множество правил вывода (productions) $\alpha \to \beta$. Пусть Γ - контекстно-свободная (в левой части только одиночные нетерминалы)

Def: LL(k)-грамматика - если достаточно посмотреть на первые k символов γ, чтобы понять, какое правило применить для нетерминала A:

S - стартовый нетерминал, **w** - слово, префикс которого разобран. Рассмотрим два произвольных левосторонних вывода слова **w** .

$$s \Rightarrow^* xA\alpha \Rightarrow x\gamma\alpha \Rightarrow^* xy\zeta$$
$$s \Rightarrow^* xA\beta \Rightarrow x\xi\beta \Rightarrow^* xy\mu$$

где x и γ - цепочки из терминалов - разобранная часть слова **w**, A - нетерминал грамматики, в которой есть правила $A \to \gamma, A \to \xi$, причем $\alpha, \beta, \xi, \gamma, \mu, \zeta$ - последовательности из терминалов и нетерминалов. Если из выполнения условий, что ($|y| = k$) или ($|y| < k, \mu = \zeta = \epsilon$) , следует равенство $\gamma = \xi$, то Γ называется **LL(k)-грамматикой**

Грамматика Γ называется **LL(1) грамматикой** (посмотрев на первый символ можно понять какое следующее правило нужно применить), если $s \Rightarrow^* xA\alpha \Rightarrow x\gamma\alpha \Rightarrow^* xc\zeta$
$s \Rightarrow^* xA\beta \Rightarrow x\xi\beta \Rightarrow^* xc\mu$

Неформально это означает, что, посмотрев на очередной символ после уже выведенной части слова, можно однозначно определить, какое правило из грамматики выбрать.

(Смотрим на символ c в строке и сразу понимаем, что $\gamma = \xi$, что значит, что мы используем одно и то же правило для A)

LL(0) грамматика - для каждого нетерминала есть только одно правило. По-другому называются "линейные программы". Такие грамматики лежат в основе теории архивации (если грамматика короче, то слово сжато, тк каждый нетерминал будет задавать только одно слово и вы можете его заменить на соответсвующий нетерминал)

Example 1: Рассмотрим грамматику и покажем, что она **LL(1)**.

$S \to aA|bB$
$A \to aB|cB$
$B \to bC|a$
$C \to bD$
$D \to d$

w = aaabd

$S \Rightarrow aA \Rightarrow aaB \Rightarrow aaaC \Rightarrow aaabD \Rightarrow aaabd$
$S \Rightarrow^* aaabd$

Каждый раз когда мы смотрели на очередной символ мы сразу определяли правило для дальнейшего вывода.

Example 2: Рассмотрим грамматику, которая по первому символу не позволяет определить правило для дальнейшего вывода.

$S \to abB|aaA$
$B \to d$
$A \to c|d$

w = abd

Смотрим на первый символ **w**, он подходит под несколько правил стартового нетерминала, только со второго символа понятно какое правило выбирать $\Rightarrow$ не **LL(1)-грамматика**.

Example 3:

$E \to T$
$E \to E\ +\ T$
$T \to F$
$T \to T\ \times\ F$
$F \to n$
$F \to (E)$

Данная грамматика не является **LL(k)**. Контр-пример:

> $2 * 2 * 2 * 2 * .... + 2$, где k символов до $+$

Мы не можем понять по первым k символам понять по какому нетерминалу нам применять правило.

$FIRST$ и $FOLLOW$

---

def *FIRST*: $(N \cup \Sigma)^* \to 2^{\Sigma \cup \{\epsilon\}}$. По строчке из терминалов и нетерминалов возвращается множество, которое состоит из символов и ϵ
$c \in FIRST(\alpha) \Leftrightarrow \alpha \Rightarrow^* cx$. Множество символов, с которых может начинаться α
$e \in FIRST(\alpha) \Leftrightarrow \alpha \Rightarrow^* \epsilon$

> *Example* $S \to SS$
> $S \to (S)$
> $S \to \epsilon$
>
> $FIRST(S) = \{c, \epsilon\}$
>
> $FIRST('S)') = \{(,)\}$
>
> $FIRST(\epsilon) = \{\epsilon\}$
>
> $FIRST('))((') = \{')'\}$

def *FOLLOW*: $N \to 2^{\Sigma \cup \{\$\}}$
$c \in FOLLOW(A) \Leftrightarrow S \Rightarrow^* \alpha A c \beta$. Множество символов, которые могут быть после
$\$ \in FOLLOW(A) \Leftrightarrow S \Rightarrow^* \alpha A$
нетерминала

> *Example* $E \to T$
> $E \to E\ +\ T$
> $T \to F$
> $T \to T\ \times\ F$
> $F \to n$
> $F \to (E)$
>
> $FOLLOW(F) = \{), \$, +, \times\}$
>
> $FOLLOW(E) = \{), \$, +\}$

Лемма о рекурсивном вычислении $FIRST$

$\alpha = c\beta$
$FIRST(\alpha) = \{c\}$

$\alpha = A\beta$
$FIRST(\alpha) = (FIRST(A)) \setminus \epsilon) \cup (FIRST(\beta)\ if\ \epsilon \in FIRST(A))$

$FIRST(\epsilon) = \{\epsilon\}$

Алгоритм построения $FIRST$

$\forall A\ FIRST[A] = \emptyset$
$while\ (FIRST\ changes)\{$
$for\ A \to \alpha :$
$FIRST[A] \cup = FIRST[\alpha]$
$\}$

Алгоритм построения $FOLLOW$

$FOLLOW : map < N, set < \Sigma \cup \$ >>$
$FOLLOW(S) = \$$
$do\{$
$for\ A \to\ \alpha$
$for\ B\ in\ \alpha$
$let\ \alpha = \xi B \eta$
$FOLLOW(B) = FIRST(\eta) \setminus \epsilon$
$if\ \epsilon \in FIRST(\eta)$
$FOLLOW(B) \cup = FOLLOW(A)$ // почему FOLLOW(A)
$\}\ while\ FOLLOW\ changes$

Алгоритм TODO()

1. Удалить непорождающие символы
2. Удалить недостижимые

   Менять шаги алгоритма нельзя

   ex: Grammar:

Удаление непорождающих символов TODO()

1. Множество непорождающих символов $Gen = \emptyset$

   do {
   for A $\rightarrow$ α
   if $\alpha \in (\Sigma \cup Gen)^*$:
   Gen $\cup =$ A
   } while Gen change

   NonGen = N $\setminus$ Gen

A - порождающий, но Алгоритм 1 выбрал как порождающий

$A \Rightarrow \alpha \Rightarrow^{k-1} x$

Алгоритм TODO()

FIRST: map<N, set<$\Sigma \cup \epsilon$>>

function getFIRST(α)

if $\alpha = \epsilon$ return $\{\epsilon\}$

if $\alpha[i] \in \Sigma$ return $\{\alpha[i]\}$

// $\alpha[0] \in N$

return $(FIRST[\alpha[0]] \setminus \epsilon) \cup (getFIRST(\alpha[1:]), if\ \epsilon \in FIRST[\alpha[0]])$

Теорема 1

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

1. $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
2. $\epsilon \in FIRST(\alpha) \Rightarrow FIRST(\beta) \cap FOLLOW(A) = \emptyset$

> Доказательство:
> Определение **LL(1)**.
>
> 1. $S \Rightarrow^* xA\xi \Rightarrow^* x\gamma\xi \Rightarrow^* xc\tau$
> 2. $S \Rightarrow^* xA\eta \Rightarrow^* x\delta\eta \Rightarrow^* xc\sigma$
>   Тогда $\gamma = \delta$
>
> $\Rightarrow$) (Необходимость). Пусть $\Gamma \in$ **LL(1)**
>
> 1. $\exists A \rightarrow \alpha, A \rightarrow \beta, c \in FIRST(\alpha) \cap FIRST(\beta)$
> $S \Rightarrow^* xA\sigma \Rightarrow x\alpha\sigma \Rightarrow^* xc\xi\sigma$
> $S \Rightarrow^* xA\sigma \Rightarrow x\beta\sigma \Rightarrow^* xc\eta\sigma$. По символу c мы не можем понять какое из правил выбирать следующим $\Rightarrow \Gamma \notin$ **LL(1)**
> 2. $\epsilon \in FIRST(\alpha) \cap FIRST(\beta)$
> $S \Rightarrow^* xA\sigma \Rightarrow x\alpha\sigma \Rightarrow^* x\sigma \Rightarrow xc\tau$

$S \Rightarrow^* xA\sigma \Rightarrow x\beta\sigma \Rightarrow^* x\sigma \Rightarrow xc\tau$. По символу c мы не можем понять какое из правил выбирать следующим $\Rightarrow \Gamma \notin$ **LL(1)**

3. $\epsilon \in FIRST(\alpha)$ и $c \in FIRST(\beta) \cap FOLLOW(A)$
   $S \Rightarrow^* xA\xi \Rightarrow^* x\alpha\xi \Rightarrow^* x\xi \Rightarrow xc\eta$
   $S \Rightarrow^* xA\xi \Rightarrow^* x\beta\xi \Rightarrow^* xc\beta'\xi$. Аналогино первым двум пунктам

$\Leftarrow$) (Достаточность). Пусть $\gamma \neq \delta$. При этом выполнены условия 1 и 2:

1. Если из γ выводится c и из δ выводится c, то $c \in FIRST(\gamma)$ и $c \in FIRST(\delta)$, что противоречит условию 1 теоремы.
2. Если из γ выводится c и из δ выводится ϵ, при этом c лежит в η, тогда $c \in FIRST(\gamma)$, $c \in FOLLOW(A)$ и $\epsilon \in FIRST(\delta)$, что противоречит условию 2 теоремы. (аналогично для $\gamma \Rightarrow^* \epsilon\zeta$ и $\delta \Rightarrow^* c\sigma$).
3. Если из γ выводится ϵ и из δ выводится ϵ, тогда, соответственно, $\epsilon \in FIRST(\gamma)$ и $\epsilon \in FIRST(\delta)$, что противоречит 1 пункту теоремы.

Рекурсивный спуск

Алгоритм

$A \to \alpha_1|\alpha_2|\ldots|\alpha_k$

1. Построить множества $FIRST$ и $FOLLOW$
2. Определим структуру $Node$

   $Node$:
   s : $N \cup \Sigma$
   ch : array(Node)

   $token$: $\Sigma \cup \{\$\}$ // текущий терминал
   $next()$ // функция взятия следующего токена
3. Определим мета-фукцию $FIRST'$ (которая, на самом деле, является множеством)

   $$FIRST'(A \to \alpha) = (FIRST(\alpha) \setminus \epsilon) \cup (FOLLOW(A)\ if\ \epsilon \in FIRST(\alpha))$$
4. Для каждого нетерминала построим функции (строим дерево разбора).

```
Node A() {
    Node res = Node(A)
    switch (token)
        FIRST`(A -> a1):
            // a1 = X1X2...Xl
            // X1 in N
            Node x1 = X1() // вызывается рекурсивно X1
            res.addChild(x1)
            // X1 in N
            Node x2 = X2()
            res.addChild(x2)
            // X3 in Sigma
            assert x3 = token or Error()
            res.addChild(token)
            next()

            ...
            // Xl ...
            ...
```

```
            return res
        FIRST`(A -> a2)
            ...

        default:
            Error()
}
```

ETF (expression, therm, factor)

Grammar:

$$
\begin{gathered}
E \to E + T \\
E \to T \\
T \to T \times F \\
T \to F \\
F \to n \\
F \to (E)
\end{gathered}
$$

	FIRST	FOLLOW
E	`n, (`	`$, +, )`
T	`n, (`	`$, +, *, )`
F	`n, (`	`$, +, *, )`

$FIRST`(E+T) = n, ($
$FIRST`(T) = n, ($ замечаем, что наша грамматика не **LL(1)** (она леворекурсивная)

Левая рекурсия и правое ветвление

Определение. Γ называется *леворекурсивной*, если в $\Gamma: A \Rightarrow^+ A\alpha$
Определение. Говорят, что в грамматике есть правое ветвление, если $A \to \alpha\beta, A \to \alpha\gamma$ и $\beta \neq \gamma$

Теорема 2.

Γ - леворекурсивная или в Γ есть правое ветвление $\Rightarrow \ \Gamma \notin LL(1)$

Доказательство

1. Левая рекурсия
   $A \Rightarrow^* x, \ x \in \Sigma^*$
   $A \Rightarrow^+ A\alpha$
   $A \Rightarrow^* B\xi \Rightarrow \gamma\xi \Rightarrow^* A\alpha \Rightarrow^* x\alpha = cy\alpha$
   $A \Rightarrow^* B\xi \Rightarrow \delta\xi \Rightarrow^* x = cy$
   $c \in (FIRST(\delta)) \setminus \epsilon \cup (FIRST(\xi) \ if \ \epsilon \in FIRST(\delta))$
   $c \in FIRST(\gamma) \setminus \epsilon \cup (FIRST(\xi) \ if \ \epsilon \ inFIRST(\gamma))$
2. Правое ветвление

   Очевидно из того, по $A \to \alpha\beta, A \to \alpha\gamma$ и $\beta \neq \gamma$ не выполняется пункт 1 теоремы 1 ($FIRST(\alpha\beta) \cap FIRST(\alpha\gamma) \neq \emptyset$)

Непосредственная левая рекурсия
$A \to A\alpha$
$A \to \beta$
$\beta\alpha^*$

Устранение левой рекурсии и правого ветвления

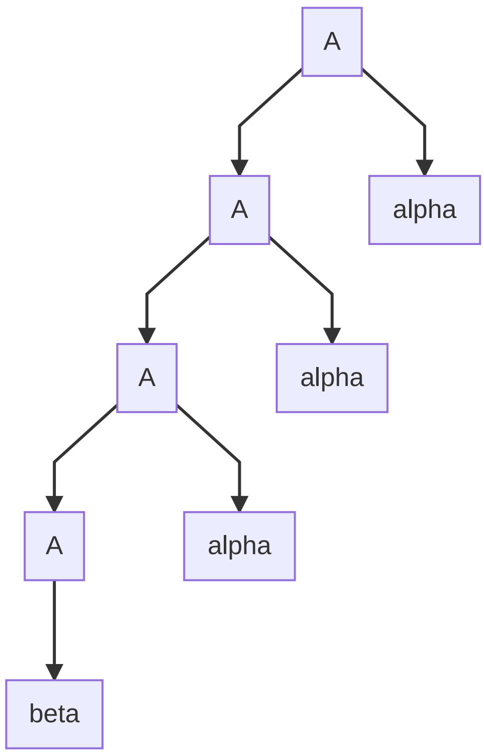
Правое ветвление
Проблема:

$A \to \alpha\beta$
$A \to \alpha\gamma$

Решение:

$A \to \alpha A`$
$A` \to \beta$
$A` \to \gamma$

Непосредственная левая рекурсия
Проблема:



$A \to A\alpha$
$A \to \beta$

Решение:

$A \to \beta A`$
$A` \to \epsilon$
$A` \to \alpha A`$

Косвенная левая рекурсия (без ϵ правил)
Добьемся такого:

$A_1, \dots, A_n$
Если $A_i \Rightarrow^+ A_j\alpha$, то $j > i$

$for\ i = 1..n$
$\quad removeDescentRecursion(A_i \to A_i\alpha)$
$\quad for\ j = i + 1..n$
$\qquad if(exist(A_j \to A_i\beta))\{$
$\qquad\quad forall\ A_i \to \gamma$
$\qquad\qquad insert(A_j \to \gamma\beta)$
$\qquad\quad remove(A_j \to A_i\beta)$
$\qquad\quad \}$

Инвариант:
Если $k < i, A_k \to \beta, \beta[1] = A_l$, тогда $l > k$
Если $k \geq i, A_k \to \beta, \beta[1] = A_l$, тогда $l \geq i$

Грамматика с устранённой непосредственной левой рекурсией

$$
\begin{gathered}
E \to TE`\\
E` \to \epsilon\\
E` \to +TE`\\
T \to FT`\\
T` \to \epsilon\\
T` \to \times FT`\\
F \to n\\
F \to (E)
\end{gathered}
$$

	FIRST	FOLLOW
E	`( n`	`$ )`
E`	`+ e`	`$ )`
T	`( n`	`+ $ )`
T`	`* e`	`+ $ )`
F	`( n`	`* + $ )`

```
Node E()
    Node res = Node(E)
    switch (token)
        case n, (:
            // E -> TE'
            Node t = T()
            res.addChild(t)
            Node e' = E'()
            res.addChild(e')
            return res

        default:
            Error()

Node E'()
    Node res = Node(E')
    switch (token)
       case $, ):
           // E' -> e
           return res
       case +, e:
           // E' -> +TE'
```

```
            assert token == +
            res.addChild(Node(t))
            next()
            Node t = T()
            res.addChild(t)
            Node e' = E'()
            res.addChild(e')
            return res

        default:
            Error()

    // T and T' are similar with above

Node F()
    Node res = Node(F)
    switch (token)
        case n:
            assert token == n
            res.addChild(n)
            next()
            return res
        case (:
            assert token == (
            res.addChild(\()
            next()
            Node e = E()
            res.addChild(e)
            assert token == )
            res.addChild(Node(\)))
            next()
            return res
```

Устранение левой рекурсии полный алгоритм (с удалением ϵ правил)

$\beta\alpha^*$
$A()$
$\quad switch$
$\qquad FIRST`(A \to \beta_1)$
$\qquad\quad \beta_1$
$\qquad FIRST`(A \to \beta_2)$
$\qquad\quad \beta_2$
...
while (token $\in$ FIRST'($A \to A\alpha$))

$A \Rightarrow^+ A\alpha$
$A \to X\alpha,\ X \in \Sigma$ или $\#X > \#A$
$A_1, A_2, \ldots, A_n,\ \#A_i = i$

$A_1 \to A_1\alpha$
$A_1 \to \beta$
$A_1 \to \beta A_1'$
$A_1' \to \alpha A_1'$
$A_1' \to \epsilon$

1. Избавиться от ϵ-правила

$A_1 \to \beta A_1'$
$A_1 \to \beta$
$A_1' \to \alpha A_1'$
$A_1' \to \alpha$

2. $A_2 \to A_1\alpha \rightsquigarrow A_2 \to \xi\alpha \text{ для всех } A_1 \to \xi \ (A_2 \to A_2\beta, A_2 \to \gamma)$
   $A_2 \to A_2\beta$
   $A_2 \to \gamma$

```
for i = 1..n
    for j = 1..i - 1
       A_i -> A_j alpha
       for A_j -> xi alpha
          add A_i -> xi alpha
      remove A_i -> A_j alpha
```

$A \to \alpha\beta$
$A \to \alpha\gamma$
$L(\alpha) \neq \{\epsilon\}$, то LL(1)

$A \to \alpha A'$
$A' \to \beta$
$A' \to \gamma$

Построение нерекурсивных нисходящих разборов

Стек, управлящая таблица. В стеке будем хранить неразобранную часть дерева. (алгоритм ака dfs со стеком)

Напишем грамматику и занумеруем правила, построим для нее множества $FIRST$ и $FOLLOW$.

$$E \to TE' \ (1)$$
$$E' \to +TE' \ (2)$$
$$E' \to \epsilon \ (3)$$
$$T \to FT' \ (4)$$
$$T' \to \times FT' \ (5)$$
$$T' \to \epsilon \ (6)$$
$$F \to n \ (7)$$
$$F \to (E) \ (8)$$

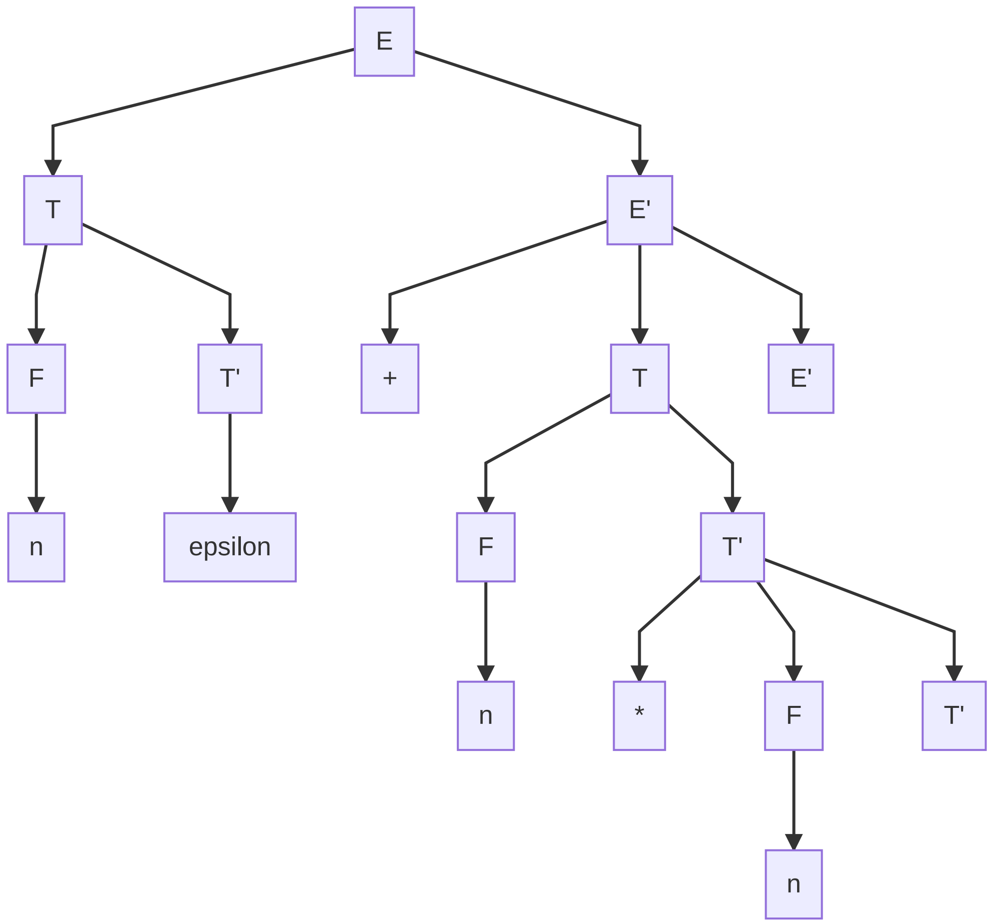
	FIRST	FOLLOW
E	`( n`	`$ )`
E'	`+ e`	`$ )`
T	`( n`	`+ $ )`
T'	`* e`	`+ $ )`
F	`( n`	`* + $ )`

Управляющая таблица нерекурсивного парсера

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

> пустые ячейки соответствуют ошибке

> e.g. to parse: `2 + 2 * 2`
> tree:



Атрибутно-транслирующие грамматики (АТГ)

$N, S \in N; \Sigma; P$ - правила
Расширим определние грамматики
N & Σ определяется в Z
атрибуты
Σ, N

0. имя
1. тип
2. значение (может быть не определено)
3. правило вычисления
   S-атрибуты - только присваивание атрибута

Атрибуты бывают:

1. **Синтезируемые атрибуты**
   Если его значение зависит только от поддерева, в том числе, когда этот атрибут - атрибут терминала и его значение на этапе лексического анализа
2. **Наследуемый атрибут**
   Значение зависит от родителей или братьев
   L-атрибутная

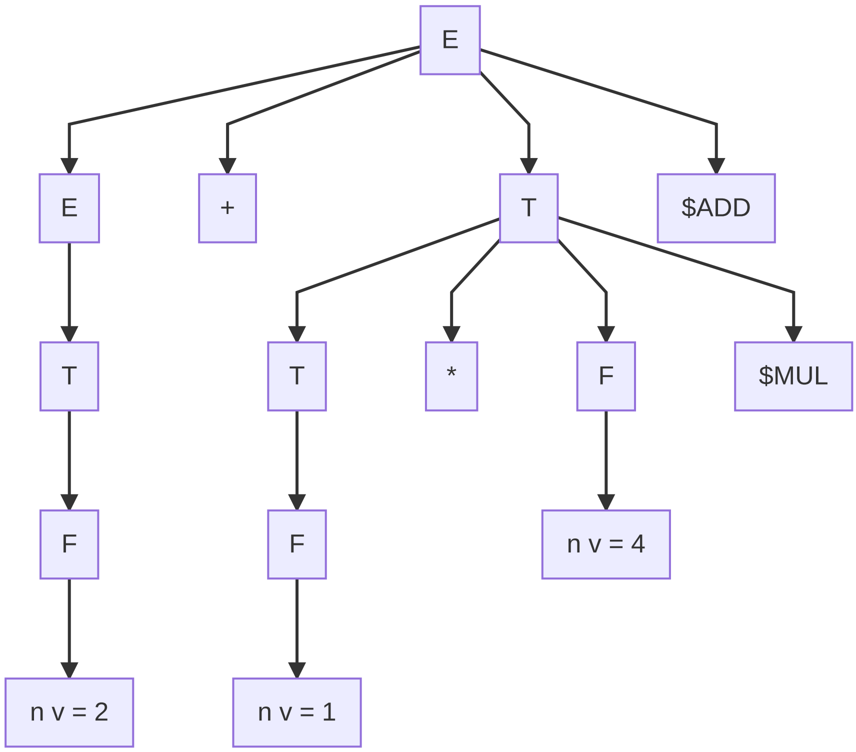
Транслирующий символ - специальный нетерминал, у которого единственное правило раскрыть его в ϵ и которого есть связанный с ним код, внутри которого мы можем работать с атрибутами

Могут быть **именными** и **анонимными**

> | | |
> |---|---|
> | $E \to E + T$ | **$MUL op1 = T_1.v**
$MUL op2 = F.v
T_0.v = $MUL res |
> | $E \to T$ | `E.v = T.v` |
> | $T_0 \to T_1 \;\; \times_2 \;\; F_3$ | $MUL op1 = T_1.v
$MUL op2 = F.v
T_0.v = $MUL res |
> | $T \to F$ | `T.v = F.v` |
> | $F \to n$ | `F.v = n.v` |
> | $F \to (E)$ | `F.v = E.v` |
>
> ```
> $MUL {
> res = op1 * op2
> }
> ```
>
> $$\$MUL \begin{cases} op1 \text{ наследуемый} \\ op2 \text{ наследуемый} \\ res \text{ синтезируемый} \end{cases}$$

```
$ADD {
add = op1 + op2
}
```

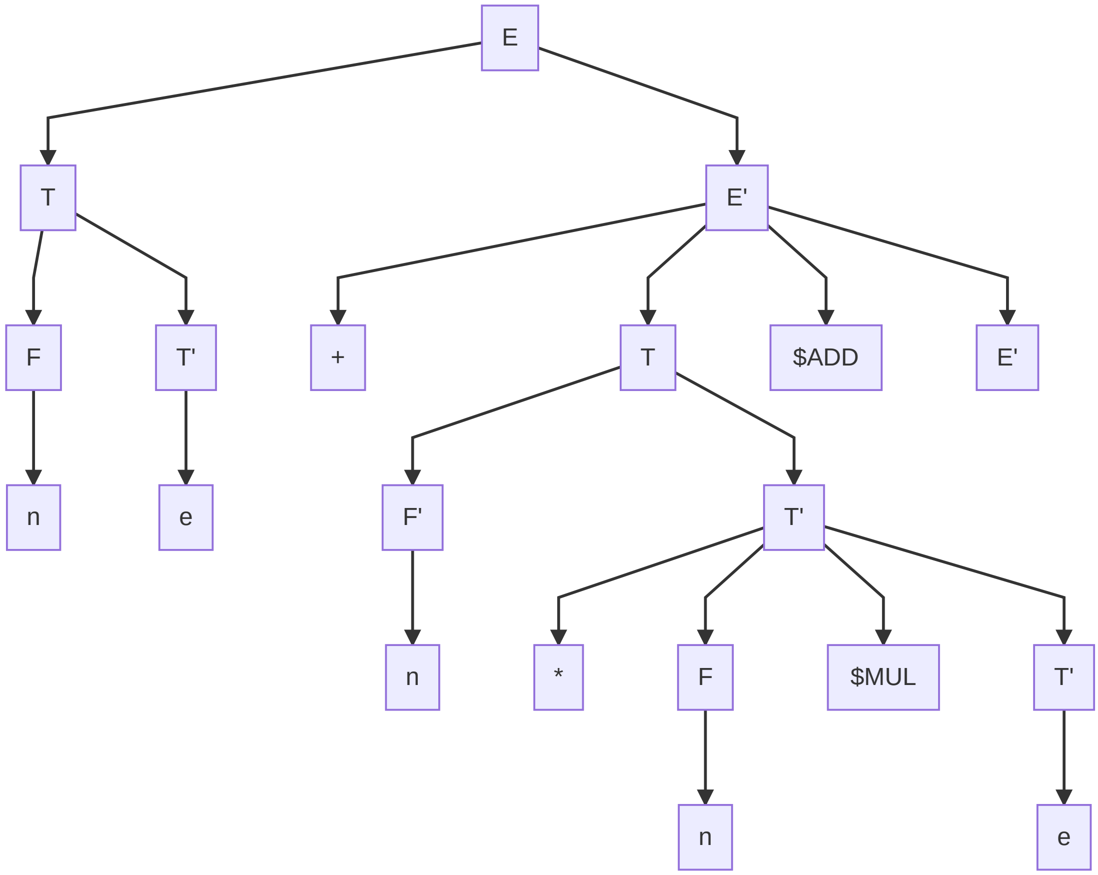
E	v	
T	v	
F		синтезируемый
n		синтезируемый



$E \to TE'$		**E'.a = T.v** **E.v = E'.v**
$E' \to +TE'$	$ADD E'	$ADD op1 = E'$_0$a $ADD op2 = T.v E'$_4$.a = $ADD.res
$E' \to \epsilon$		E'.v = E'.a
$T \to FT'$		T'.a = E'.a T.v = T'.v
$T' \to \times FT'$	$MUL T'	$MUL op1 = T'$_0$.a $MUL op2 = F.v T'$_4$.a = $MUL res
$T' \to \epsilon$		T'.v = T'.a
$F \to n$		F.v = n.v
$F \to (E)$		F.v = E.v

E	v
T	v
F	v синтезируемый
n	v синтезируемый
E'	a наследуемый v синтезируемый
T'	a наследуемый v синтезируемый

`2 + 3 * 4`




```
E'(a: int): int
    switch
        case // -> e
            return a
        case // +T $ADD E'
            skip +
            T.v = T()
            $ADD.res = $ADD(a, T.v)
            E'.v = E'($ADD.res)
            return E'.v

E(): int
   switch
       case
           T.v = T()
           E'v. = E'(T.v)
           return E'.v

$ADD(op1, op2: int): int
   return op1 + op2

// alternative:
Node E'(a)
    Node res = Node(E, atr = {a.a})

    switch
        -> e
           res v = res.a
           return res
```

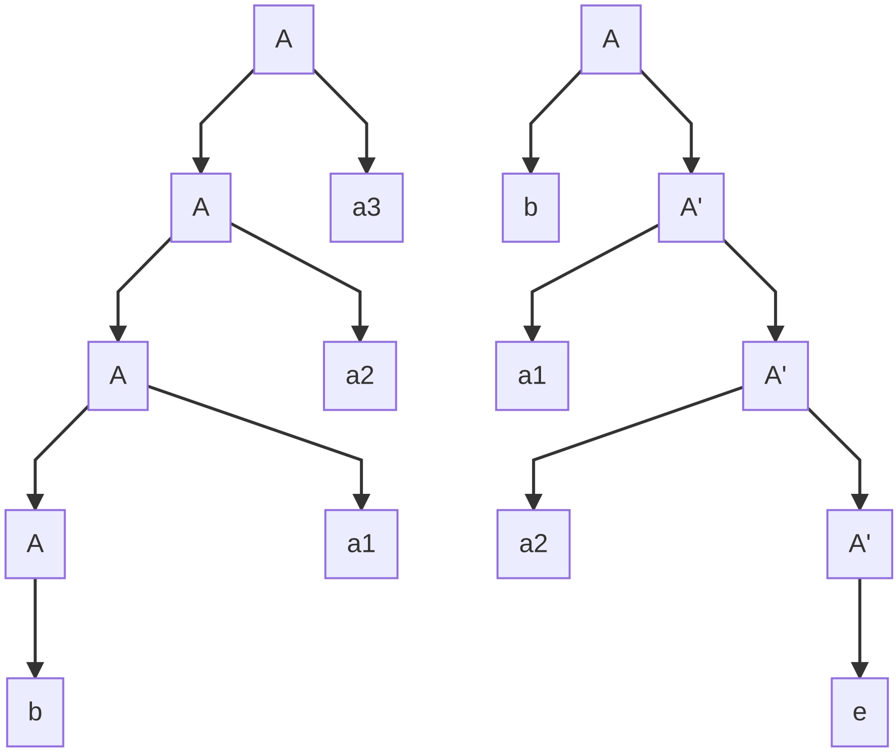
```
        -> +TE'
           skip +
           T = T()
           E'4.a = res.a + T.v
           E' = E'(E'4.a)
           res.v = E'v
           return res
```

Регистровые машины и **Стековые** машины
операции регистровых машин: load загрузить значение и store выгрузить в память
преимущество перед регистровыми, в регистровых конечное количество регистров, здесь есть стек и операции push, pop

---

Непосредственная левая рекурсия
$A \rightarrow A\alpha$
$A \rightarrow \beta$
x - синтезируемый атрибут A

$A \rightarrow \beta A'$
$A' \rightarrow \epsilon$
$A' \rightarrow \alpha A'$



A' x - соответствует Ax - синтезируемый
a - аккумулятор - наследуемый

$A \rightarrow \beta A' \qquad A'a = f(\beta)$
$A' \rightarrow \epsilon$
$A' \rightarrow \alpha A'$

A s - синтезируемый атрибут
a - наследуемый атрибут

```
A(a) -> s
    switch ()
        ...
        // A -> a
        s = f(alpha)
        // alpha_k = B
        B(<->)
```

Но вообще генерируются парсеры со стеком

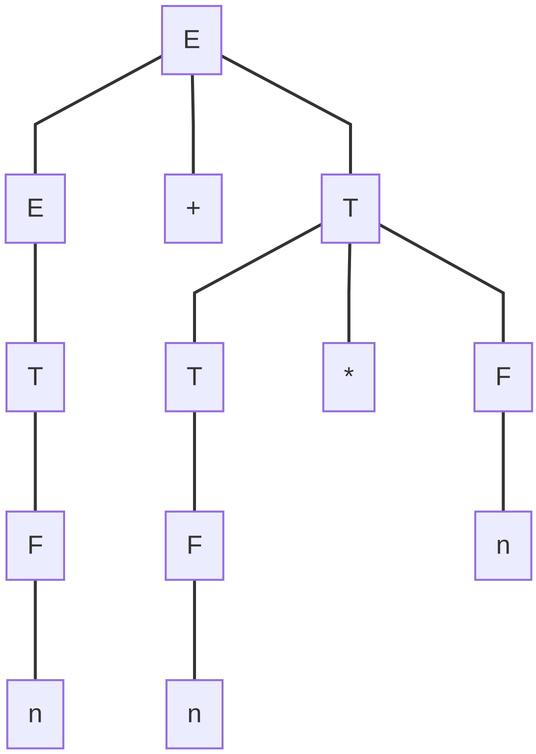
Восходящий разбор

перенос - свёртка
`shift - reduce`

Рабочий стек (WS) и стек предпросмотра (PS) (изначально вся последовательность токенов лежит в стеке предпросмотра).
Есть две операции:

1) Перенос. Берем символ из PS и переносим в WS.
2) Свёртка. Берем суффикс из WS, находим правило, для которого это правая часть. Если правил несколько, то все зависит от метода разбора.

Пример. Следующее дерево построено снизу-вверх.



Сразу понятно, что выбор операции (свертка или перенос) на текущем суффиксе WS неочевиден, есть различные способы как это можно сделать. Первый из них - это по приоритетам операторов.

LR - анализ

Неформальный смысл **LR(k)**. Если мы знаем строчку целиком и следующие **k** символов, то мы можем выбрать правило, по которому будем сворачивать

Отличие от **LL**. В **LL** грамматике мы знаем только первый символ того, что мы собираемся разбирать, а в **LR** грамматике мы знаем все символы того, что мы собираемся сворачивать и еще следующий.

Будем в дальнейшем рассматривать грамматику правильных скобочных последовательностей:
$S' \to S$
$S \to (S)S$
$S \to \epsilon$

LR(0)-ситуация

LR(0)-ситуация называется пара из правила и позиции в его правой части. Где количество позиций = длина правила + 1

$A \to \alpha \bullet \beta$, где $\bullet$ - позиция

Рассмотрим стек, давайте построим конечный автомат по поиску основ: посмотрим в строку α в WS, будем называет ее суффикс основой, если существует такое содержимое в PS, что, учитывая содержимое строки α можно их свернуть в левую часть некоторого правила, чтобы у нас получилась свертка до стартового терминала. (ака пытаемся по какому-то кол-ву символов угадать дерево разбора)

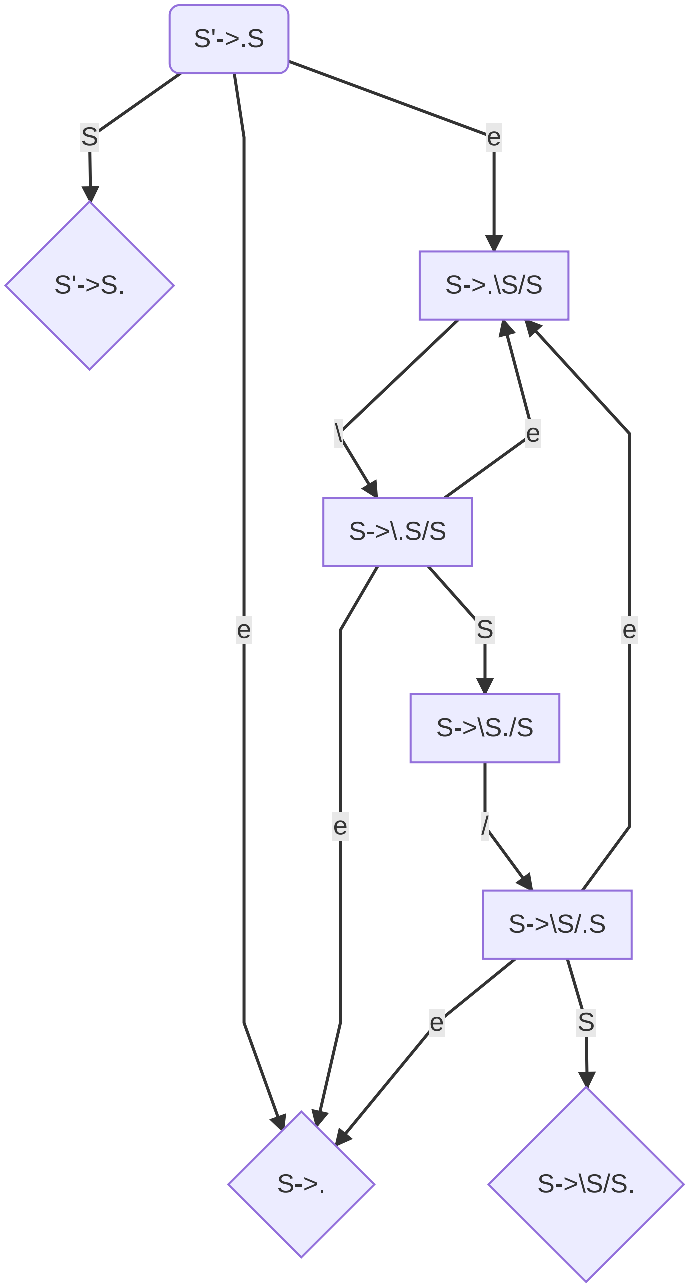
Добавим формализма:
$\alpha = \beta\gamma$, γ - основа, если $\exists t$, $A \to \gamma$, то $S \Rightarrow^* \beta A t \Rightarrow \beta\gamma t$, где S - стартовый нетерминал, $A \in N$, $t \in \Sigma$, $\alpha, \gamma, \beta \in (\Sigma \cup N)^*$

Конфликт свёртки/свёртки: ноль нетерминальных и больше одного терминала
Конфликт переноса/свёртки: больше нуля нетерминальных и больше нуля терминальных

def Грамматика называется **LR(0)-грамматикой**, если детерминированная версия автомата по поиску основы выглядит: каждый шаг содержит либо одно состояние недетерминированного автомата и ничего больше, либо содержит только нетерминальные состояния недетермнированного автомата.

Автомат

LR(0)-ситуации - это состояния автомата. (\ - левая скобка, / - правая скобка, ибо mermaid душит). Алфавит автомата $\Sigma_A = \Sigma \cup N$.
1) Если точка стоит перед каким-то символом, то по этому же символу в этом же правиле переходим в следующую ситуацию.
2) По ϵ: если точка стоит перед нетерминалом, то по ϵ переходим в начало всех правил, которые начинаются с этого нетерминала.
3) Стартовое $S' \to .S$. Ромбовидные - терминальные состояния.



Рассмотрим наш недетерминированный автомат:
1) Если мы находимся только в терминальном состоянии, то должны сделать свертку
2) Если мы находимся только в нетерминальных состояниях мы должны сделать перенос
3) Если мы находимся в терминальных и нетерминальных, то у нас конфликт перенос-свертка
4) Если мы находимся в нескольких терминальных, то у нас конфликт свертка-свертка

Заметим, что наша грамматика не **LR(0)**

LR(0) редко используется, есть LR(1)
LR -> SLR (Simple LR) -> LALR -> LR(1)

SLR - анализ

Разновидность **LR(0)**, в котором конфликты перенос-свертка решаются с помощью $FOLLOW$.

Алгоритм для нашей грамматики с **SLR** анализом:
1) передаем в наш автомат строку с WS;
2) если оказались только в терминальном - это свертка;
3) только в нетерминальных - перенос;
4) если хоть одно терминальное и не верно, что только оно одно - смотрим $FOLLOW$ всех

терминальных состояний, в которых мы оказались и если следующий символ в PS лежит ровно в одном $FOLLOW$, то сворачиваем по этому правилу, если не лежит ни в одном, то делаем перенос. Если лежит в нескольких $FOLLOW$, то у нас неустранимый конфликт свертка-свертка.

Построим $FOLLOW$:

	$FOLLOW$
S'	$
S	), $

На самом деле, текущий алгоритм на практике применим только к детерминированному автомату. Распишем его полностью заново

Алгоритм для SLR анализа

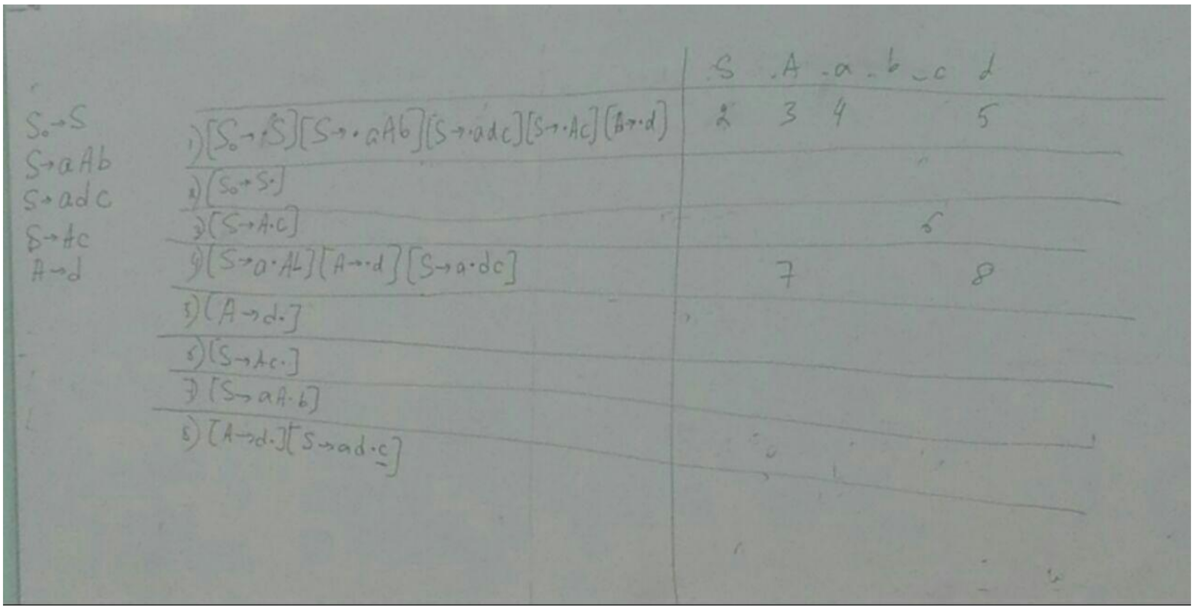
1. Строим **LR(0)** автомат
2. Детерминизируем
3. Строим управляющую таблицу **SLR** анализа
    1. $[A \to \alpha\bullet]$ - терминальное, свертка $A \to \alpha$
    2. $[A \to \alpha\bullet, B \to \beta\bullet, \ldots, C \to \xi \bullet X\eta \ldots], FOLLOW(A)$ - свертка $[A \to \alpha]$, $FOLLOW(B)$ - свертка $[B \to \beta]$, если любой нетерминал, то перенос.
    3. $[A \to \alpha \bullet X\beta, B \to \gamma \bullet Y\delta, \ldots]$ - перенос

Построим нашу управляющую таблицу. (shift,3 - перенеси и перейди в 3 номер; reduce(3), сверни по 3 правилу грамматики и смотри предыдущий номер состояния для следующего перехода)

Номер	**Состояния**	**S**	**(**	**)**	**$**
1	$S' \to \bullet, S \to \bullet(S)S, S \to \bullet$	2	shift,3	reduce(3)	reduce(3)
2	$S' \to S\bullet$	error	error	error	reduce(1)
3	$S \to (\bullet S)S, S \to \bullet(S)S, S \to \bullet$	4	shift,3	reduce(3)	reduce(3)
4	$S \to (S\bullet)S$	error	error	shift,5	error
5	$S \to (S) \bullet S, S \to \bullet(S)S, S \to \bullet$	6	shift,3	reduce(3)	reduce(3)
6	$S \to (S)S\bullet$	error	error	reduce(2)	reduce(2)

Проходом по таблице храним все прошедшие позиции по номерам состояний, сворачиваем по правилам и переносим перемещаясь на требуемое состояние.

Когда не работает SLR:



Мы не можем понять по $FOLLOW$, как дальше сворачивать, потому что $FOLLOW$ не учитывает контекст. Однако **LR(1)** разбор нам поможет

LR(1)-ситуация

def *LR1-ситуация* **-** это тройка из правила, числа от 0 до длины правой части и символа, который называется *символом предпросмотра (look ahead)

$[A \to \alpha \bullet \beta, c]$
$[A \to \alpha \bullet d\,\beta, c] \xrightarrow{d} [A \to \alpha d \bullet \beta, c]$
Если $d \in N$

lr1 **-** грамматике **-** если в детерминированном автомате по поиску lr1 основ

одно из них терминальное,а другое нетерминальное, то их символ предпросмотра отличается от символа перед которым находится позиция в правой части нетерминального

если они оба терминальные, то их символ предпросмотра не совпадает

[lr1 приколюхи]

LR(k)

Определение. LR(k)-рамматика

$S \Rightarrow^* \alpha Axy \Rightarrow^* \alpha\gamma xy \Rightarrow^* sxy, |x| \leq k, |x| < k \Rightarrow y = \epsilon, z = \epsilon$
$S \Rightarrow^* \beta Bt \Rightarrow^* \beta\delta xz \Rightarrow^* sxz$, t-суффикс xz
Тогда $A = B, \gamma = \delta$

Говоря неформально, мы делаем правостороннюю свёртку нашей строки в стартовый нетерминал. Если по не более чем kk символам неразобранной части строки мы можем однозначно определить, во что сворачивается хвост выведенного правила, то грамматика будет LR(k).

Утверждение $\Gamma - LR(K)$, т и тт когда $LR(k)$ автомат имеет управляющу таблицу без конфликтов

LALR-разбор

LR(0) ядро LR(1)-состояние

Пусть есть 2 состояния в детерминированном **LR(1)** автомате, которые различаются только символами предпросмотра, тогда это **LR(0)** ядро **LR(1)** состояния.

$[A_1 \to \alpha_1\beta_1, c_1][A_2 \to \alpha_2\beta_2, d_1]$
$[A_1 \to \alpha_1\beta_1, d_1][A_2 \to \alpha_2\beta_2, d_2]$ если совпадают LR(0) ядра, то мерджим в

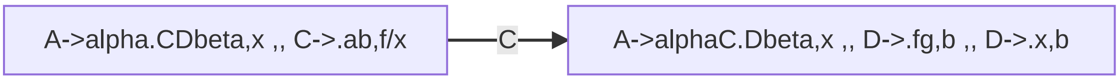
$[A_1 \to \alpha_1\beta_1, c_1/d_1][A_2\alpha_2\beta_2, c_2/d_2]$

Введем отношение эквивалетности по **LR(0)** ядрам (предыдущая операция слияния). Теперь посмотрим, что происходит с нашим автоматом и какие конфликты появляются.

$\to^x A_i \to \alpha_i B_i \bullet \beta'_i, c_i$, если $\beta'_i = C_i\beta''_i$ ($C_i \in N$), то добавляем $c_i \to \bullet\gamma_{i.j}, FIRST(\beta''_i c_i)$
Тогда можно заметить, что переходы будут те же самые

Пример

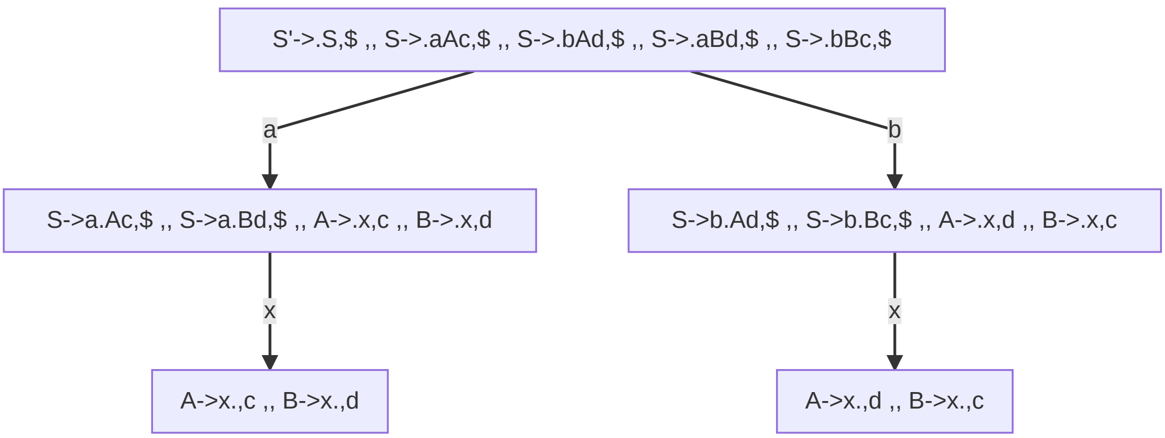
$C \to ab$
$D \to fg$
$D \to x$
Рассмотрим состояния детерминированного автомата.



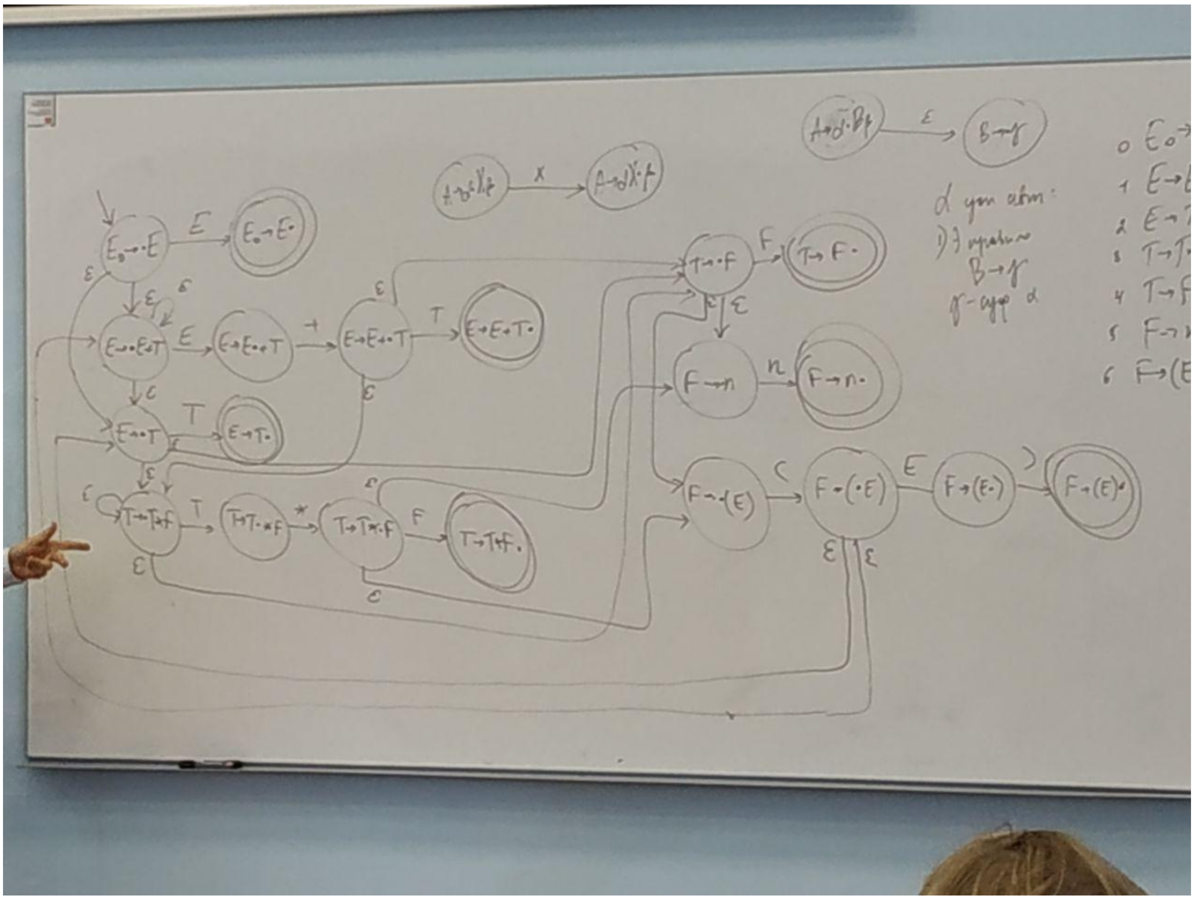
Грамматика с конфиктом свертка свертка

$S \to aAc$
$S \to bAd$
$S \to aBd$
$S \to bBc$
$A \to x$
$B \to x$

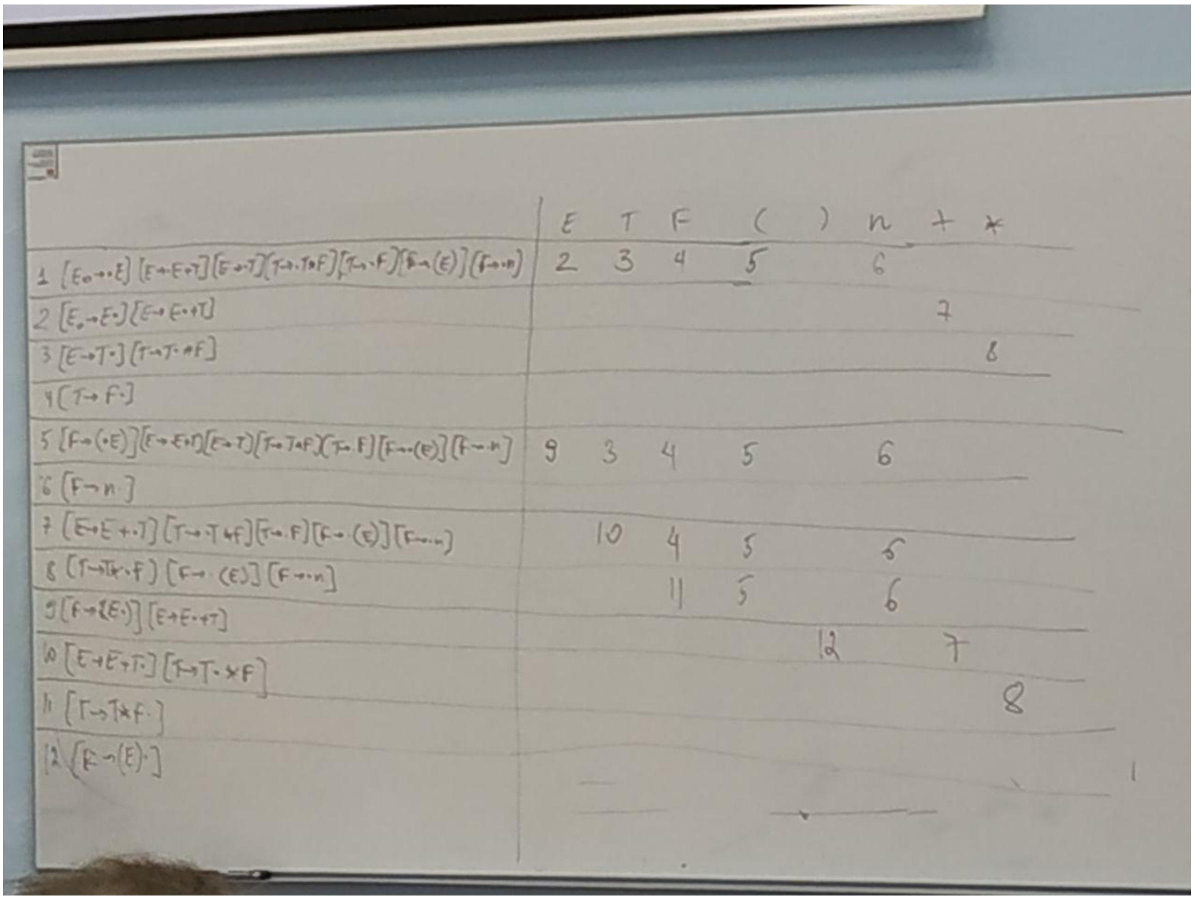
Автомат



Грамматика простых арифметических выражений **LR(1)**

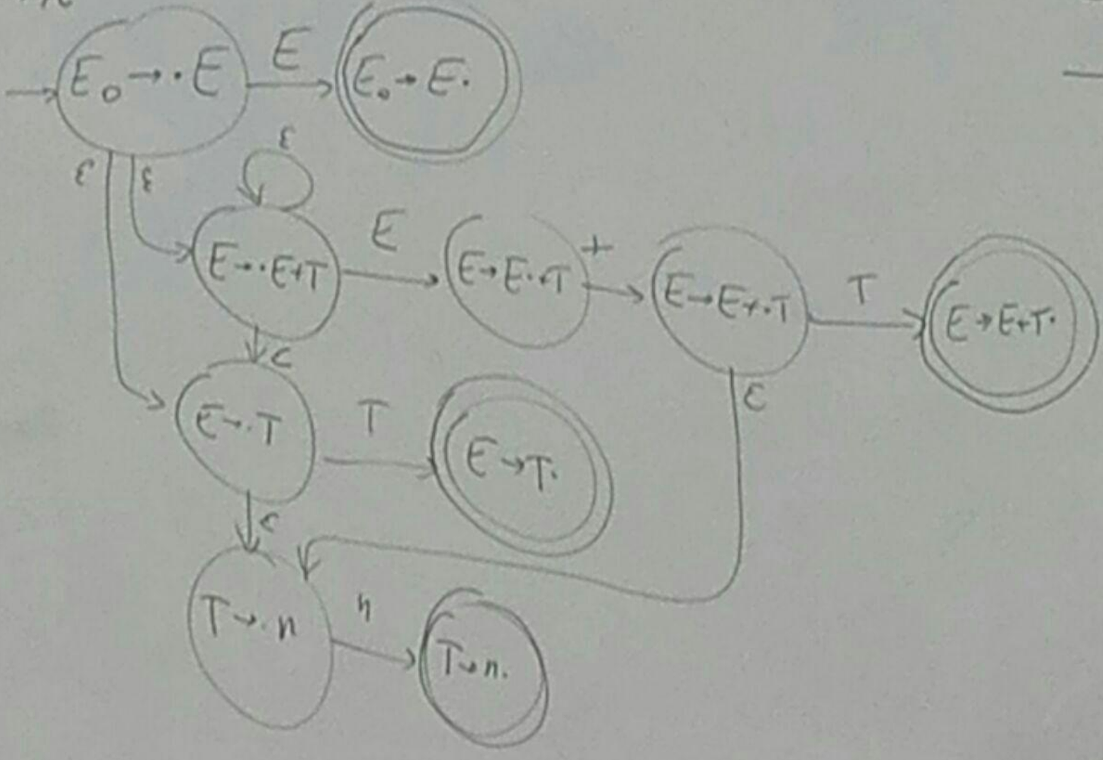


E₀→·E
E₀→E·
E→·E+T
E→E·+T
E→E+·T
E→E+T·
E→·T
E→T·
T→·T*F
T→T·*F
T→T*·F
T→T*F·
T→·F
T→F·
F→·n
F→n·
F→·(E)
F→(·E)
F→(E·)
F→(E)·



E T F ( ) n + *
1 [E₀→·E] [E→·E+T] [E→·T] [T→·T*F] [T→·F] [F→·(E)] [F→·n] 2 3 4 5 6
2 [E₀→E·] [E→E·+T] 7
3 [E→T·] [T→T·*F] 8
4 [T→F·]
5 [F→(·E)] [E→·E+T] [E→·T] [T→·T*F] [T→·F] [F→·(E)] [F→·n] 9 3 4 5 6
6 [F→n·]
7 [E→E+·T] [T→·T*F] [T→·F] [F→·(E)] [F→·n] 10 4 5 6
8 [T→T*·F] [F→·(E)] [F→·n] 11 5 6
9 [F→(E·)] [E→E·+T] 12 7
10 [E→E+T·] [T→T·*F] 8
11 [T→T*F·]
12 [F→(E)·]

$E_0 \to E$
$E \to E+T$
$E \to T$
$T \to n$



	E	T	n	+
старт → 1) $[E_0 \to \cdot E][E \to \cdot E+T][E \to \cdot T][T \to \cdot n]$	2	3	4	
2) $[E_0 \to E\cdot][E \to E\cdot +T]$				5
3) $[E \to T\cdot]$				
4) $[T \to n\cdot]$				
5) $[E \to E+\cdot T][T \to \cdot n]$		6	4	
6) $[E \to E+T\cdot]$				