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
>
> $\Sigma = \{n, +, \times, \ (, \ )\}$
>
> $(2 \ + \ 2) \times 2 \to (n + n) \times n$
>
> $n : (0|1|\ldots|9)(0|1|\ldots|9)^*$

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

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

   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

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

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 $\to \alpha$
           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

Γ является LL(1) $\Leftrightarrow \forall A \to \alpha, A \to \beta$:

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

> Доказательство:
>
> $\Rightarrow$) от противного:
>
>] не (1)
>
> 1. $\exists A \to \alpha, A \to \beta, c \in FIRST(\alpha) \cap FIRST(\beta)$
>
> $S \Rightarrow^* xA\sigma \Rightarrow x\alpha\sigma \Rightarrow^* xc\xi\sigma$
>
> $S \Rightarrow^* xA\sigma \Rightarrow x\beta\sigma \Rightarrow^* xc\eta\sigma$
>
> 2. $\epsilon \in FIRST(\alpha) \cap FIRST(\beta)$
>
> $S \Rightarrow^* xA\sigma \Rightarrow x\alpha\sigma \Rightarrow^* x\sigma \Rightarrow xc\tau$

> $$S \Rightarrow^* xA\sigma \Rightarrow x\beta\sigma \Rightarrow^* x\sigma \Rightarrow xc\tau$$
>
>] не (2)
>
> ...

Рекурсивный спуск

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

Node:

s: $N \cup \Sigma$

ch: array(Node)

token: $\Sigma \cup \{\$\}$

next()

$FIRST'(A \to \alpha) = (FIRST(\alpha) \setminus \epsilon) \cup (FOLLOW(A) \; if \; \epsilon \in FIRST(\alpha))$

```
Node A() {
    Node res = Node(A)
    switch (token)
        FIRST'(A -> a1):
            // a1 = X1X2...Xl
            // X1 in N
            Node x1 = 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
E	`n, (`
T	`n, (`
F	`n, (`

`FIRST (E + T) = {n, (}`
`FIRST(T) = {n, (}`

def Γ называется *леворекурсивной*, если в $\Gamma: A \Rightarrow^{+} A\alpha$

comment Γ - леворекурсивная $\Rightarrow\ \Gamma \notin LL(1)$

$$\begin{gathered}A \Rightarrow \beta \Rightarrow^{*} A\alpha \\ A \Rightarrow^{*} B\xi \Rightarrow \gamma\xi \Rightarrow^{*} A\alpha \\ A \Rightarrow^{*} B\xi \Rightarrow \delta\xi \Rightarrow^{*} x = cy \\ c \in (FIRST(\delta)) \setminus \epsilon \cup (FIRST(\xi)\ if\ \epsilon \in FIRST(\delta)) \\ c \in FIRST(\gamma) \setminus \epsilon \cup (FIRST(\xi)\ if\ \epsilon\ in FIRST(\gamma))\end{gathered}$$

$A \to A\alpha$ - непосредственная левая рекурсия
$A \to \beta$
$\beta\alpha^{*}$

Устранение левой рекурсии:

$A \to \beta A'$
$A' \to \epsilon$
$A' \to \alpha A'$

$E \to E \overset{\alpha}{+T}$
$E \to \overset{\beta}{T}$

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
```

$$\begin{gathered} A \to A\alpha \\ A \to \beta \\ ---\\ A \to \beta A' \\ A' \to \alpha A' \\ A' \to \epsilon \end{gathered}$$

$\beta\alpha^*$

A

switch

FIRST'($A \to \beta_1$)

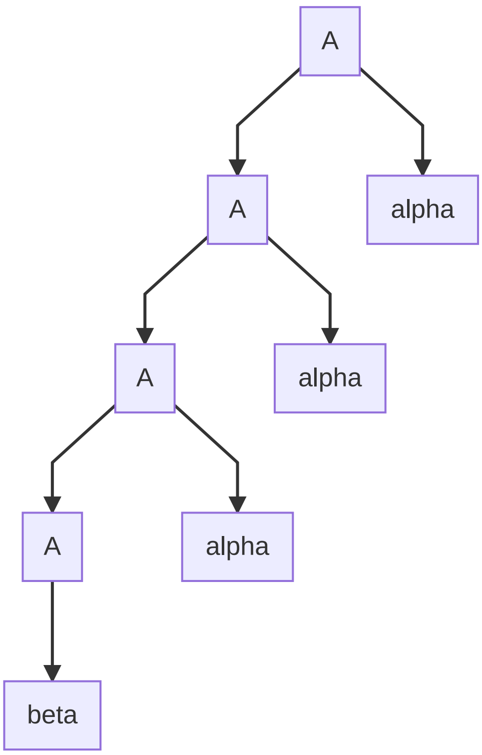
β_1

FIRST'($A \to \beta_2$)

β_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 нерекурснвных нисходящих разборов

Стек, управлящая таблица

			Σ	**c**		**$**
N	A					
					ERROR	
Σ			SKIP			ERROR
		ERROR				
$\perp$						

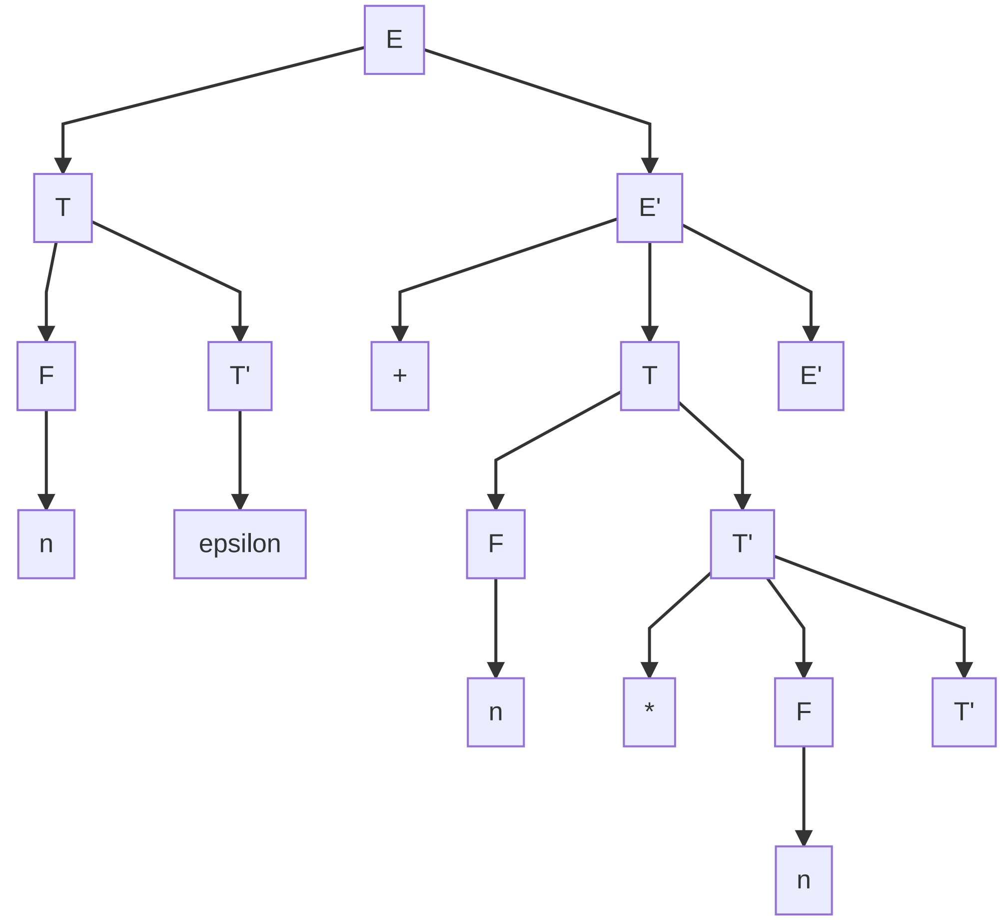
$$
\begin{gathered}
E \to TE' \\
E' \to \epsilon \\
E' \to +TE' \\
T \to FT' \\
T' \to \epsilon \\
T' \to \times FT' \\
F \to n \\
F \to (E)
\end{gathered}
$$

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

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

> 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**

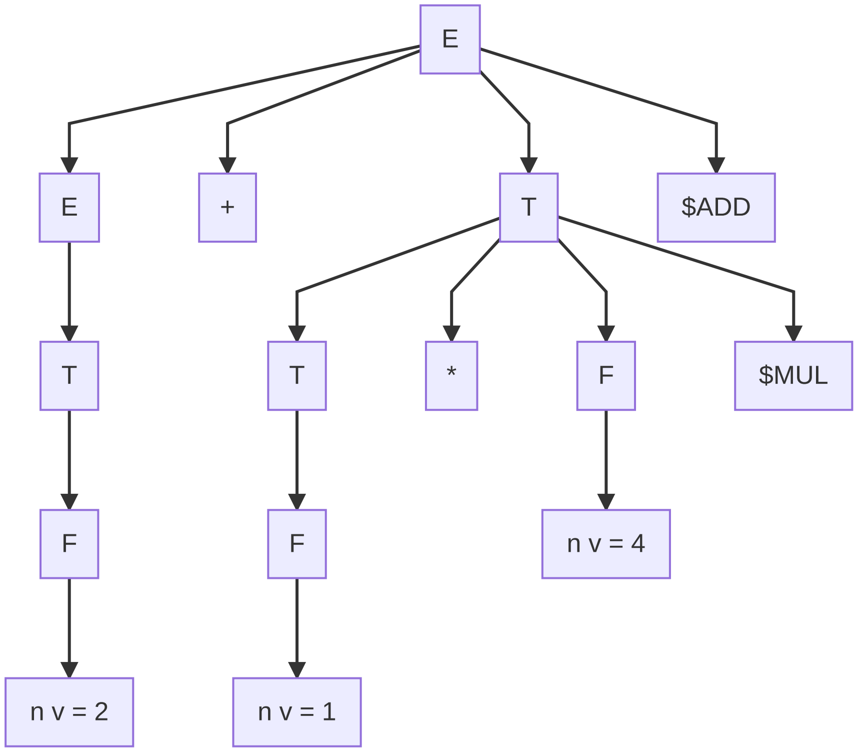
$E \to E + T$	**\$MUL op1 = T_1.v** **\$MUL op2 = F.v** **T_0.v = \$MUL res**
$E \to T$	`E.v = T.v`
$T_0 \to T_1 \ \times_2 \ F_3$	\$MUL op1 = T_1.v \$MUL op2 = F.v T_0.v = \$MUL res
$T \to F$	`T.v = F.v`
$F \to n$	`F.v = n.v`
$F \to (E)$	`F.v = E.v`

```
$MUL {
res = op1 * op2
}
```

$$\$MUL \begin{cases} op1 \text{ наследуемый} \\ op2 \text{ наследуемый} \\ res \text{ синтезируемый} \end{cases}$$

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

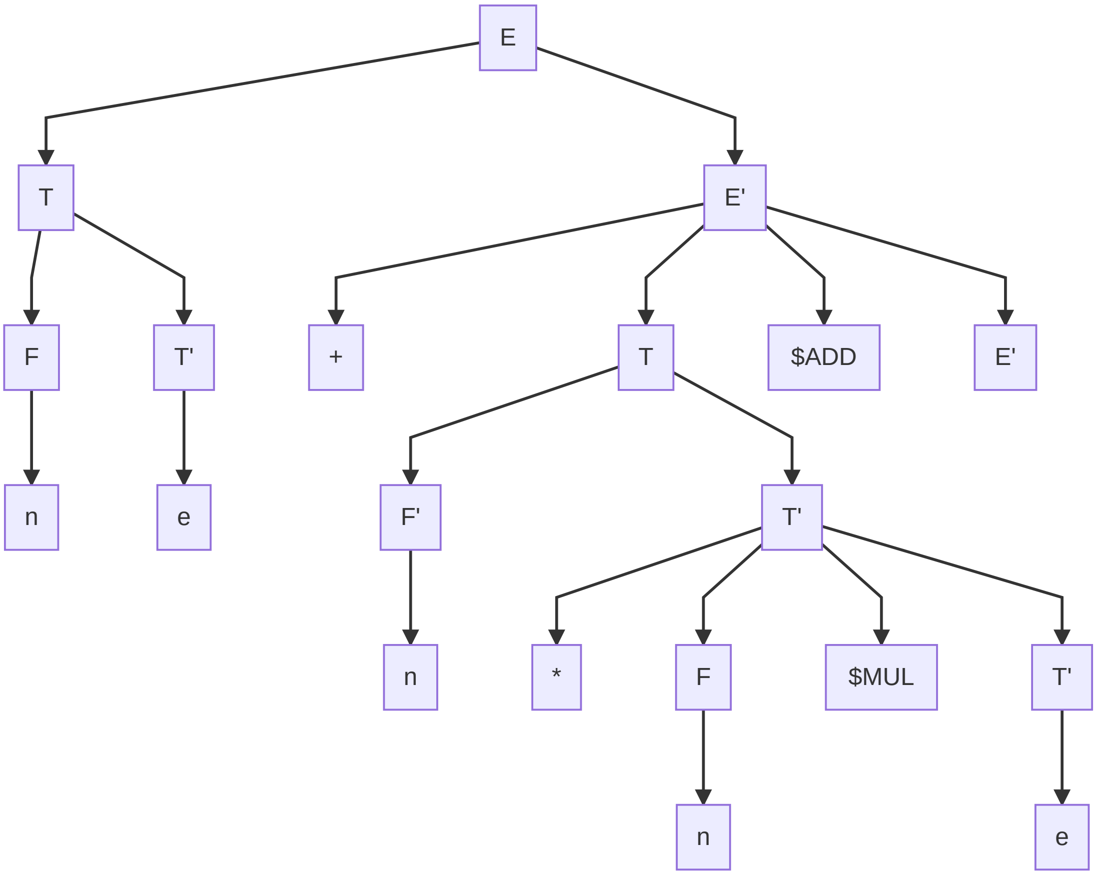
E	v	
T	v	
F		синтезируемый
n		синтезируемый



$E \rightarrow TE'$		**E'.a = T.v** **E.v = E'.v**
$E' \rightarrow +TE'$	$ADD E'	$ADD op1 = E'_0a $ADD op2 = T.v E'_4.a = $ADD.res
$E' \rightarrow \epsilon$		E'.v = E'.a
$T \rightarrow FT'$		T'.a = E'.a T.v = T'.v
$T' \rightarrow \times FT'$	$MUL T'	$MUL op1 = T'_0.a $MUL op2 = F.v T'_4.a = $MUL res
$T' \rightarrow \epsilon$		T'.v = T'.a
$F \rightarrow n$		F.v = n.v
$F \rightarrow (E)$		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
```

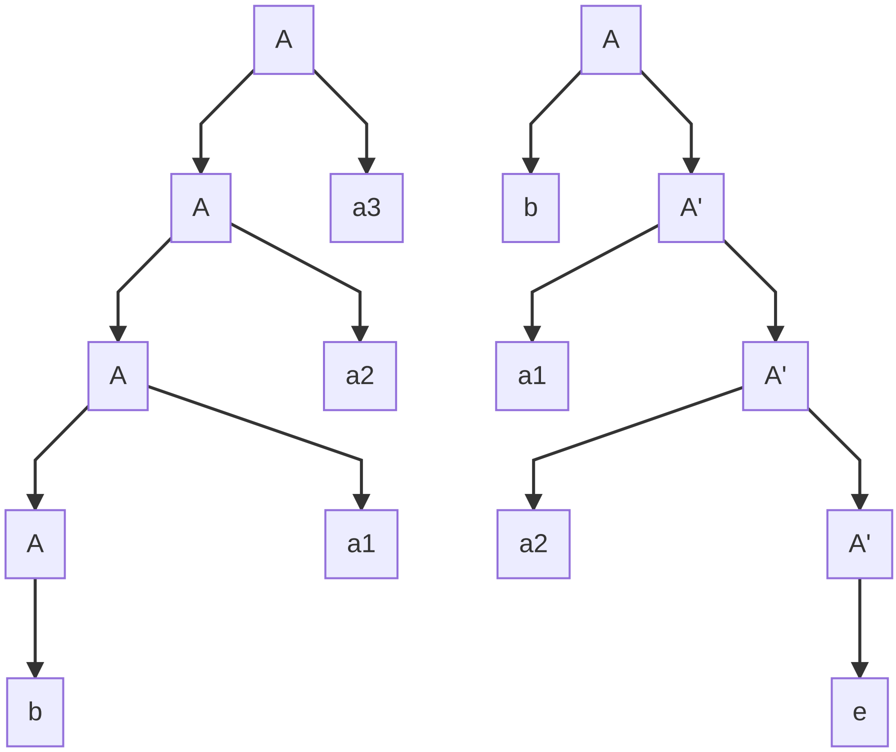
```
E(): int
   switch
       case
           T.v = T()
           E'v. = E'(T.v)
           return E'.v

$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
$A \to A\alpha$
$A \to \beta$
x - синтезируемый атрибут A

$A \to \beta A'$
$A' \to \epsilon$
$A' \to \alpha A'$



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

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

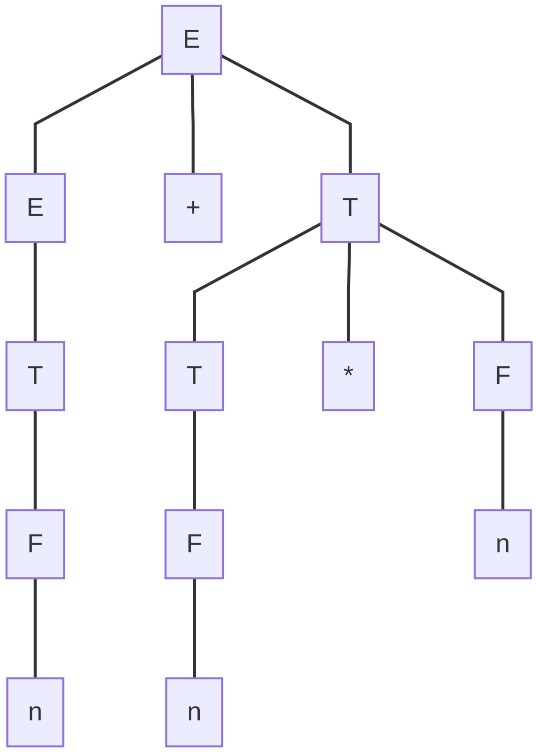
A s - синтезируемый атрибут
a - наследуемый атрибут

```
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`



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

$$\eta B u \Rightarrow \eta \beta u = \xi A t \Rightarrow \xi \alpha t = \omega$$

$\gamma = \xi t \quad S \Rightarrow^* \gamma \quad \xi \in (\Sigma \cup N)^*, t \in \Sigma^*$
α - подстрока γ
$\gamma = \xi' \alpha t' \quad \xi'$ - подстрока ξ, t' - суффикс t

$$S \Rightarrow^* \xi' A t' \Rightarrow \xi' \alpha t' = \xi t$$

Ситуации (items)
LR(1) - ситуация $(A \to \alpha, K \in \{0, \dots, |\alpha|\})$

Теорема

Ели строка альфа допускается автоматом, построенным по этим правилам, то:

1. Существует правио $B \to \gamma$, γ - суффикс α

def Грамматика называется *LR0 грамматикой*, если детерминированная версия автомата по поиску основы каждое состояние содержит либо одно состояние недетерминированного автомата и ничего больше, либо содержит только нетерминальные состояния недетермнированного автомата.

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

Когда не работает SLR:

LR1

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

$[A \to \alpha \bullet \beta, c]$
$[A \to \alpha \bullet d\, \beta, c] \xrightarrow{d} [A \to \alpha a \bullet \beta, c]$

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]