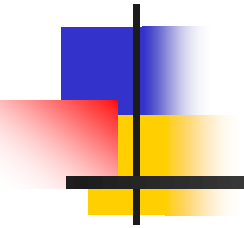


Автоматы и формальные языки

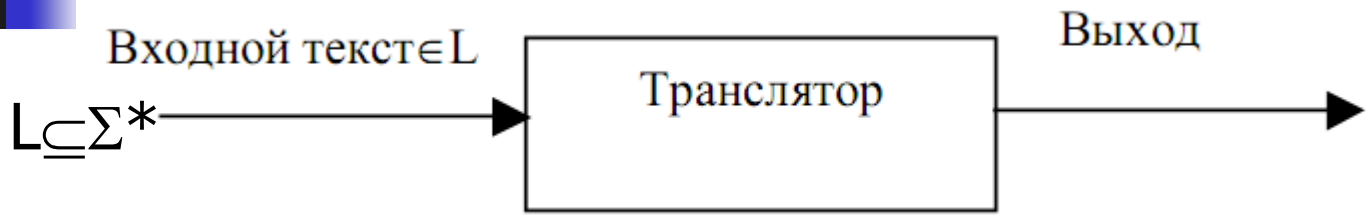
Карпов Юрий Глебович
профессор, д.т.н., зав.кафедрой
“Распределенные вычисления и компьютерные сети”
Санкт-Петербургского Политехнического университета

karpov@dcn.infos.ru

Структура курса

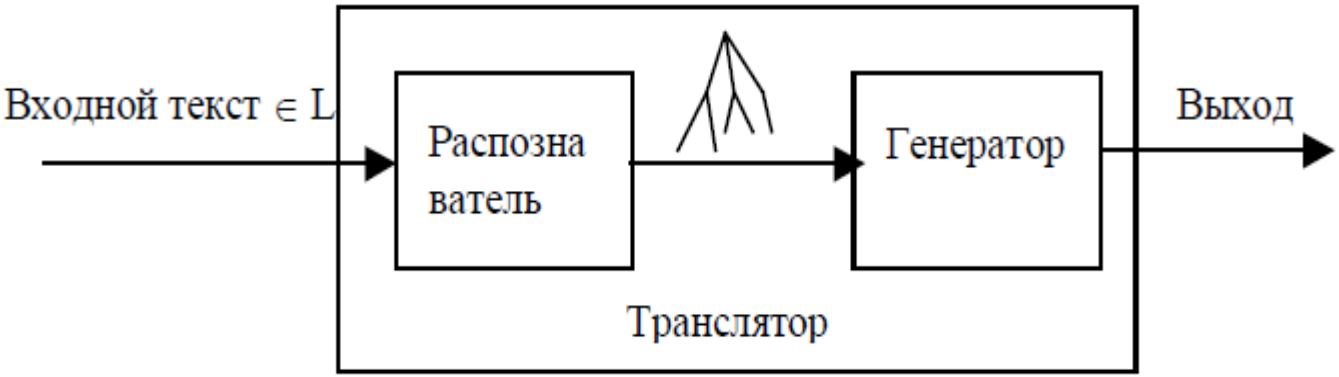
- Конечные автоматы-распознаватели – 4 л
- Порождающие грамматики Хомского – 3 л
- Атрибутные трансляции и двусмысленные КС-грамматики – 2 л
  - Лекция 8. Атрибутная семантика Кнута и атрибутные грамматики
  - Лекция 9. Примеры атрибутных трансляций. Трансляция арифметических выражений
- Распознаватели КС-языков и трансляция – 6 л
- Дополнительные лекции - 2 л

Как строить транслятор??

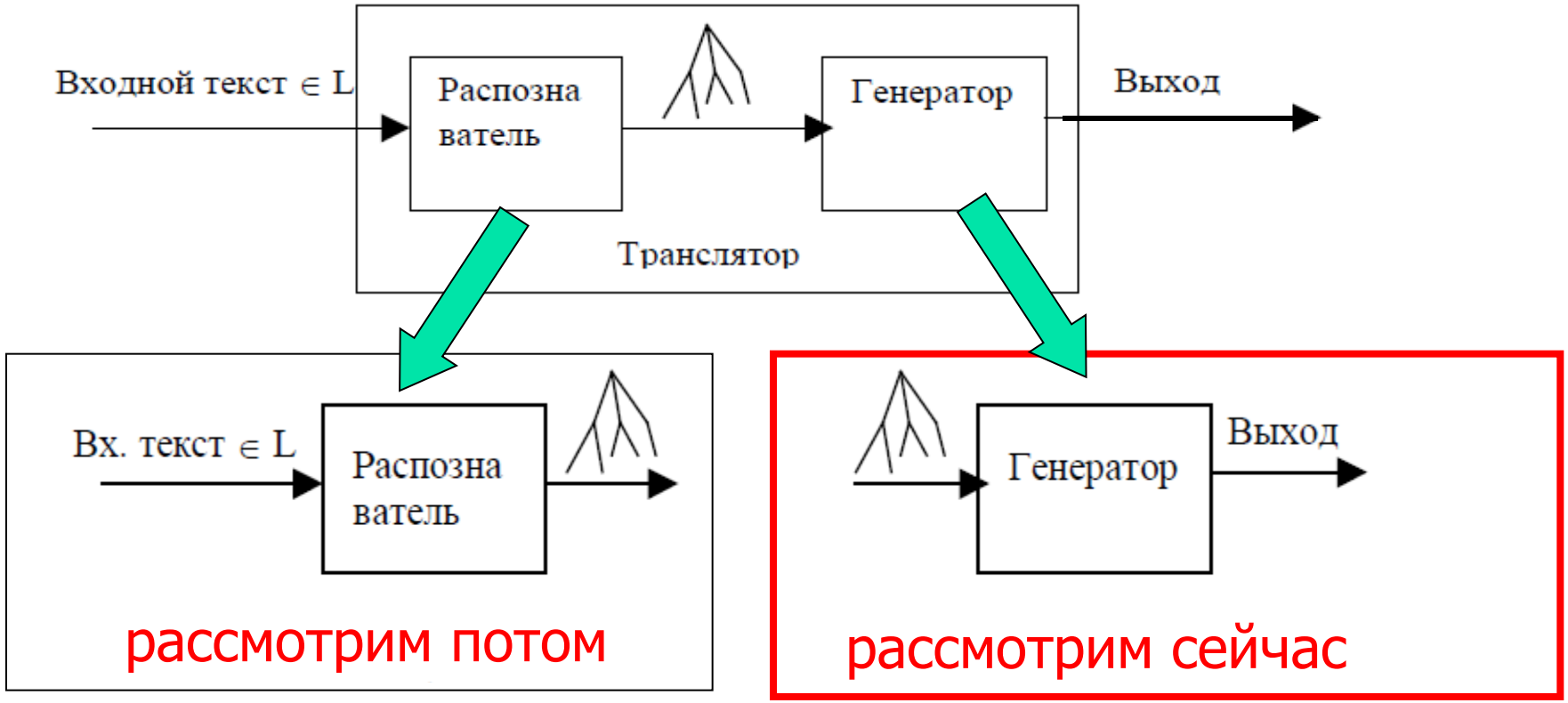


Идея Хомского: структура предложения помогает пониманию

На основе этой идеи, транслятор ЯВУ строится из ДВУХ блоков: Распознавателя и Генератора



Зачем нужна структура входной цепочки?



Будем считать, что дерево вывода (структура цепочки) уже построено

Как структура входной цепочки языка помогает восстановить тот смысл, который несет эта цепочка?

Грамматика языка Милан

Программа вычисления НОД

```
begin
   m:=read;
   n:=read;
   while m≠n do
        if m>n then m:=m-n
        else n:=n-m;
   write(m)
end
```

G^1::

Prog → begin **L** end

L → **S** | **S** ; **L**

S → *i* := **E**

| if **B** then **L**

| if **B** then **L** else **L**

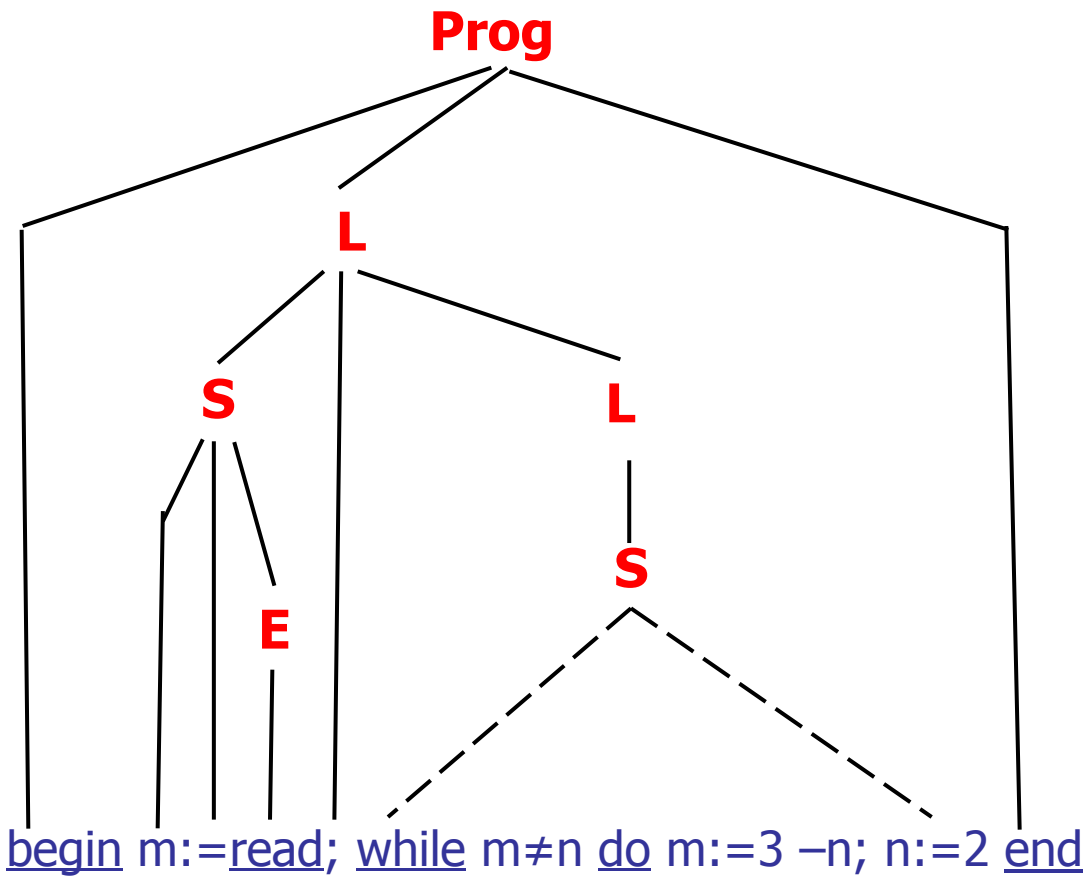
| while **B** do **L**

| write (**E**)

B → **E** rel **E**

E → (**E**) | **E** op **E**

| *i* | *c* | read



Является ли грамматика Милана однозначной???

Грамматика языка Милан

Программа вычисления НОД

```
begin
   m:=read;
   n:=read;
   while m≠n do
        if m>n then m:=m-n
        else n:=n-m;
   write(m)
end
```

G^1::

Prog → begin **L** end

L → **S** | **S** ; **L**

S → *i* := **E**

| if **B** then **L**

| if **B** then **L** else **L**

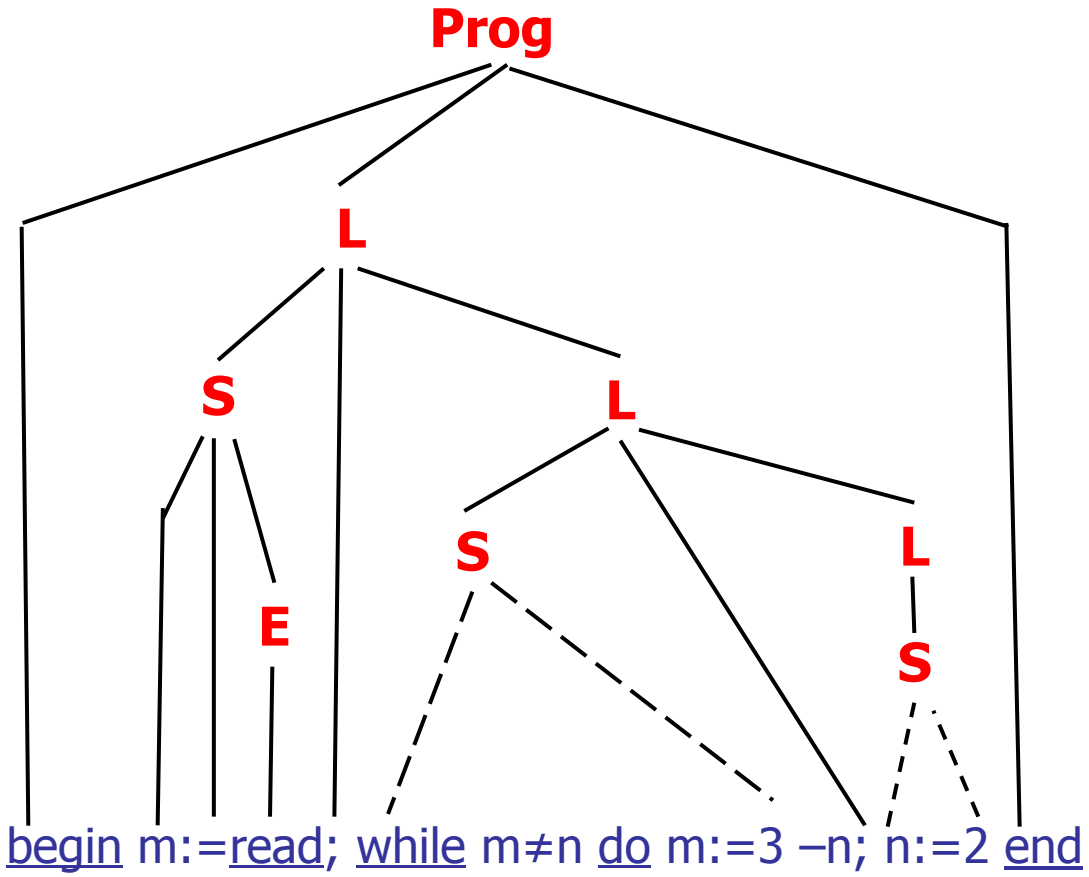
| while **B** do **L**

| write (**E**)

B → **E** rel **E**

E → (**E**) | **E** op **E**

| *i* | *c* | read



Грамматика Милана двусмысленна!!

Поэтому результат трансляции неоднозначен

Устранение двусмысленности грамматики Милана: изменим язык

Возможны различные решения, позволяющие сделать грамматику условных операторов и циклов однозначной. Наиболее простое и очевидное – потребовать "закрывающих скобок" в конце оператора. Таким служебным словом, может быть либо end, либо fi – симметрично "открывающей скобке" if. Такая грамматика, в которой окончание условного оператора будет фиксировано, будет недвусмысленной :

по аналогии с

(	**)**
[	**]**
{	**}**
do	**od**
case	**esac**
if	**fi**

S→ **if** B then S else L **fi**
| **if** B then S **fi**
| while B **do** L **od**
| s

Измененная грамматика языка Милан

Изменяем язык! Добавляем закрывающие скобки в while-do и if-then

```
begin
   m:=read;
   n:=read;
   while m≠n do
        if m>n then m:=m-n
               else n:=n-m
        fi
   od;
   write(m)
end
```

G^1: **Prog** → begin **L** end

L → **S** | **S** ; **L**

S → *i* := **E**

| if **B** then **L** **fi**

| if **B** then **L** else **L** **fi**

| while **B** do **L** **od**

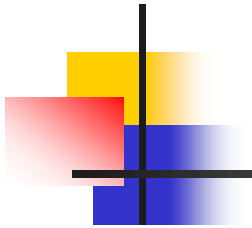
| write (**E**)

B → **E** rel **E**

E → (**E**) | **E** op **E** | *i* | *c* | read

- В грамматике все терминалы – это лексемы, выход первого блока транслятора
  - begin – служебное слово begin, end – служебное слово end, do, od, while, write, ...
  - i_k – имя переменной (какая конкретно переменная – указано в индексе k)
  - rel_k – символ отношения (какое конкретно отношение – указано в индексе k)
  - c_k - константа (какая конкретно константа – указано в индексе k)
  - op – арифметическая операция (+, -, /, *)
  - (,), :=, ; … лексемы

Стала ли грамматика Милана однозначной???



Арифметические выражения

Грамматика арифметических выражений

$E \to E + E$
$E \to E - E$
$E \to E * E$
$E \to E / E$
$E \to i$

- Введем смысловые параметры (семантические атрибуты) для каждого терминального символа (символа языка) и каждого нетерминального символа (конструкции языка)

Свяжем с каждой нетерминальной конструкцией E, имеющей смысл "выражение", семантический параметр (атрибут) E.v – (value) - значение того выражения, которое выводится из этой конструкции E

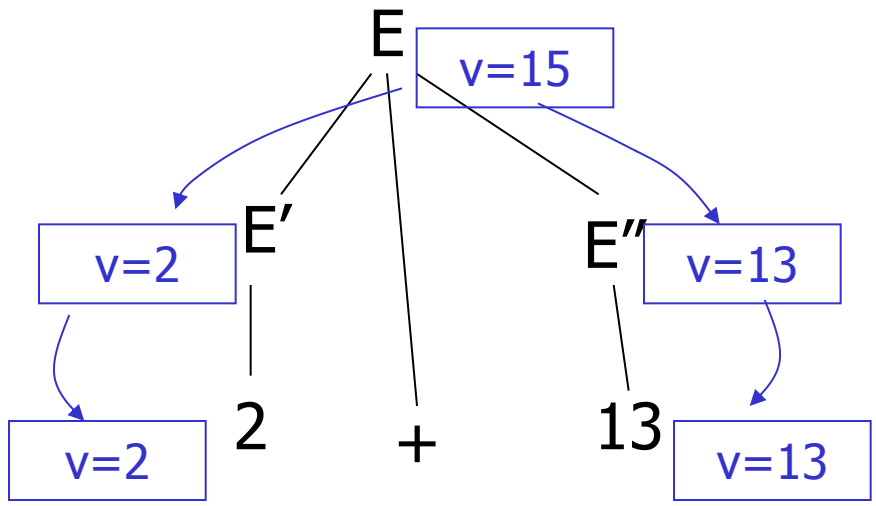
Атрибутная грамматика

N	Синтаксис	Семантика
1	$E \to E' + E''$	E.v := E′.v + E″.v
2	$E \to i$	E.v := i.v

символ в цепочке языка

операция над значениями при трансляции

Значения семантических атрибутов вычисляются по дереву вывода

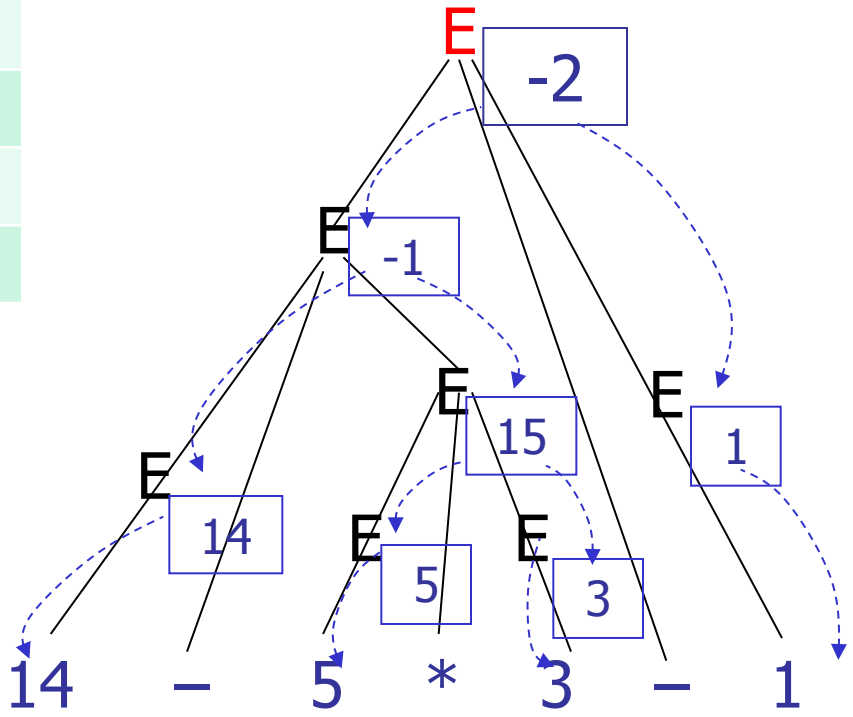


зависимости атрибутов

Атрибутная грамматика арифметических выражений

$E \to E + E$
$E \to E - E$
$E \to E * E$
$E \to E / E$
$E \to i$

N	Синтаксис	Семантика
1	$E \to E' + E''$	E.v := E′.v + E″.v
2	$E \to E' + E''$	E.v := E′.v - E″.v
3	$E \to E' + E''$	E.v := E′.v * E″.v
4	$E \to E' + E''$	E.v := E′.v / E″.v
5	$E \to i$	E.v := i.v



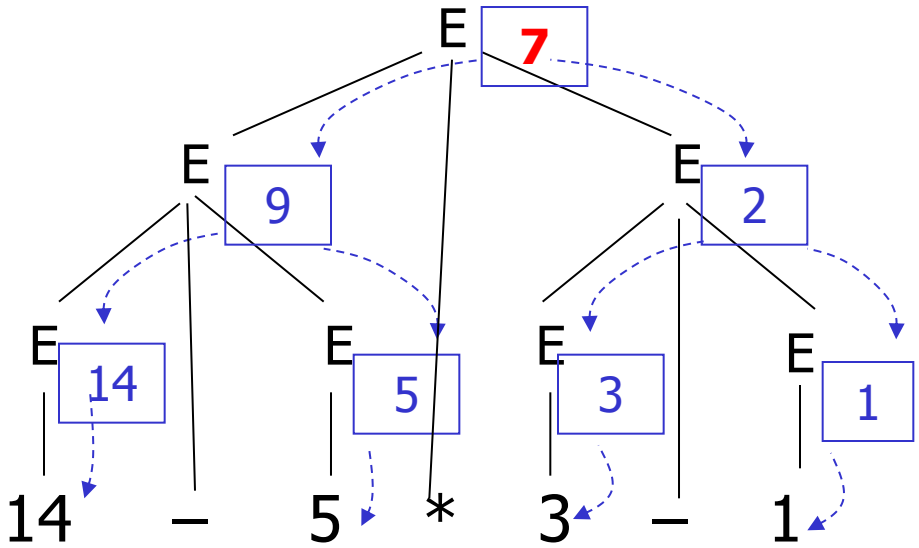
Дерево вывода цепочки, по которому вычисляется “правильное” значение выражения

Простейшая грамматика арифметических выражений ДВУСМЫСЛЕННА

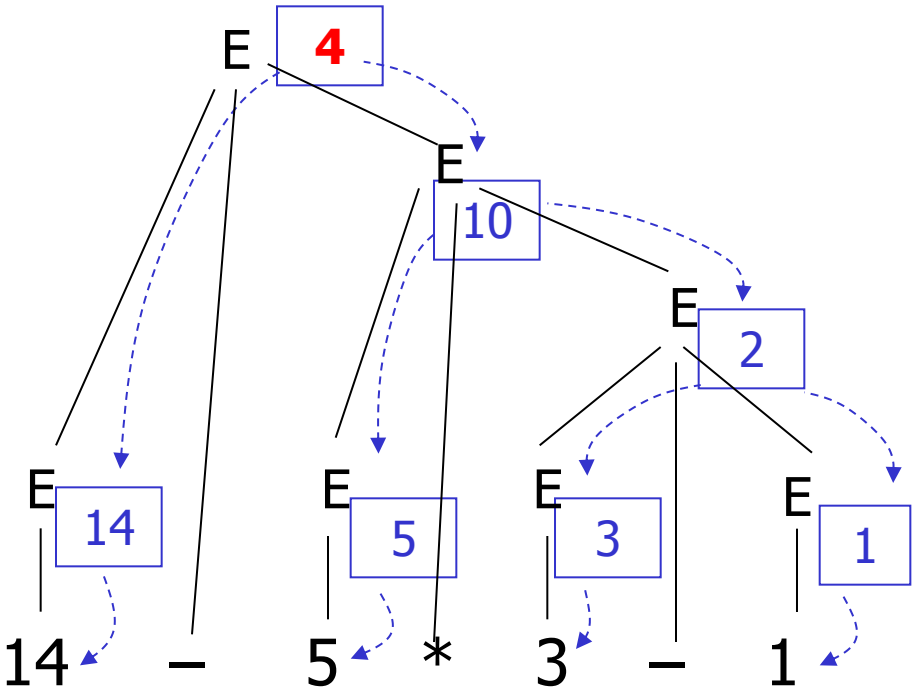
$E \to E' \otimes E''$	E.v := v.E′.v ⊗ E″.v
$E \to i$	E.v := i.v

⊗ - любая бинарная операция

Одно из возможных деревьев вывода:



В языке APL – счет справа налево
без приоритетов



В обычной алгебре =-2!!

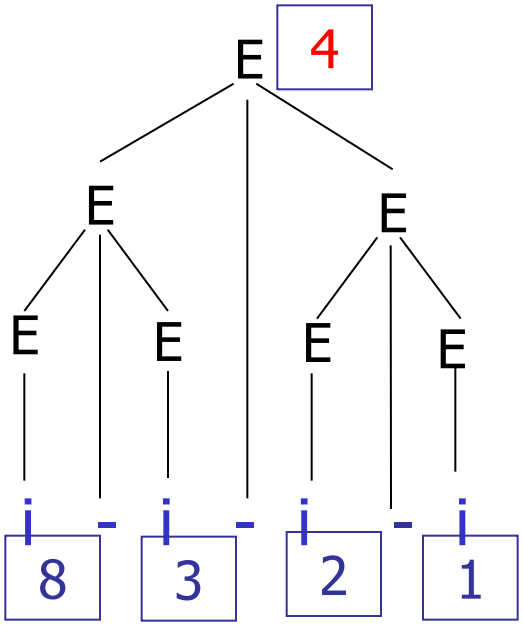
Что такое “правильное” значение?

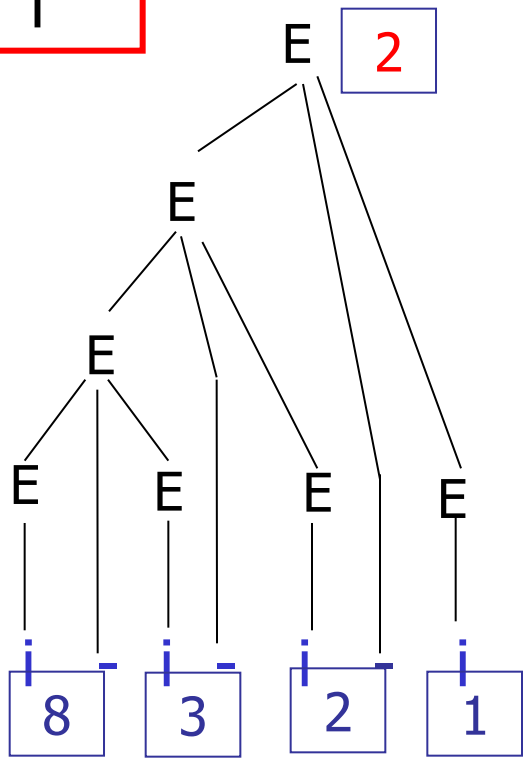
Простейшая грамматика арифметических выражений ДВУСМЫСЛЕННА

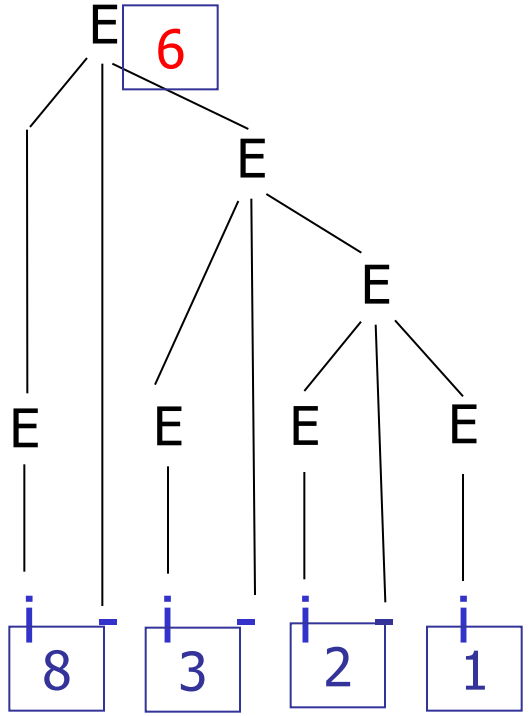
$E \to E' - E''$	E.v := v.E′.v - E″.v
$E \to i$	E.v := i.v

$$E \Rightarrow E - E \Rightarrow i - E - E \Rightarrow$$
$$i - i - E \Rightarrow i - i - E - E \Rightarrow$$
$$i - i - i - E \Rightarrow i - i - i - i$$

$$E \Rightarrow^* i - i - i - i$$





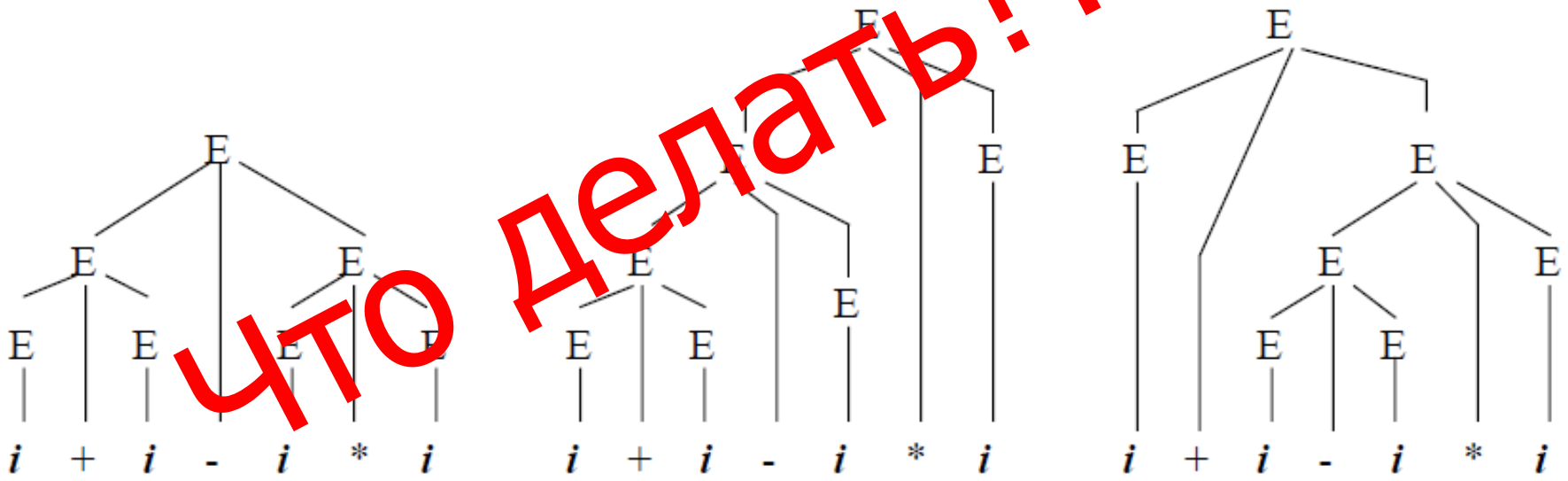


Грамматики арифметических выражений двусмысленна!

G_0^E:: E → *i* | (E) | E+E| E-E| E×E| E/E

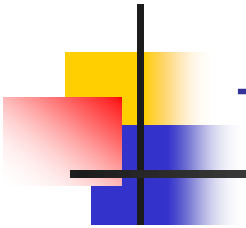
Например, цепочка $i + i - i \times i$ имеет в этой грамматике 6 различных деревьев вывода. Если семантика для этой грамматики определяется обычным образом, то можно получить несколько различных значений одного и того же арифметического выражения

Что делать???



Грамматика G_0^E выполняет главную функцию, которая требуется от грамматики – порождает все правильные арифметические выражения. НО она двусмысленна.

G_0^E часто приводят как правила построения выражений, но ее нельзя использовать для трансляции



Три возможных выхода

- 1. Изменить ЯЗЫК арифметических выражений. Разрешить использовать в языке выражения ТОЛЬКО с одним либо двумя аргументами
- 2. Изменить ЯЗЫК арифметических выражений. Разрешить использовать в языке ТОЛЬКО скобочные выражения
- 3. Построить такую грамматику арифметических выражений, которая была бы недвусмысленна и давала бы “нужное” дерево вывода любому арифметическому выражению

Арифметические выражения только с одним либо двумя операндами

Грамматика

$E \to i + i$
$E \to i - i$
$E \to i * i$
$E \to i / i$
$E \to i$

Грамматика однозначна!

Для вычисления
y:= a – b * c + d
пишем программу:

```
x1:= b * c;
x2:= a - x1;
x3:= x2 + d;
y := x3;
```

Неудобно!

Автокод ЭВМ “Наири”

```
i=3 j=4 a
i=2 x
1 допустим i=0
2 допустим j=0
3 введем aij
4 вставим j=j+1
5 если j-3≤0 идти к 3
6 вставим i=i+1
7 если i-2≤0 идти к 2
8 программа су (а 3 х)
9 печатаем с 5 знаками х0 х1
х2 10 кончаем
```

- Первые трансляторы требовали именно такого представления арифметических выражений. В частности, автокод ЭВМ “Наири”

Скобочные арифметические выражения

Для вычисления

y:= a – b * c + d

Нужно записать:

y := ((a – (b * c)) +d) ;

Грамматика

$E \to (E + E)$

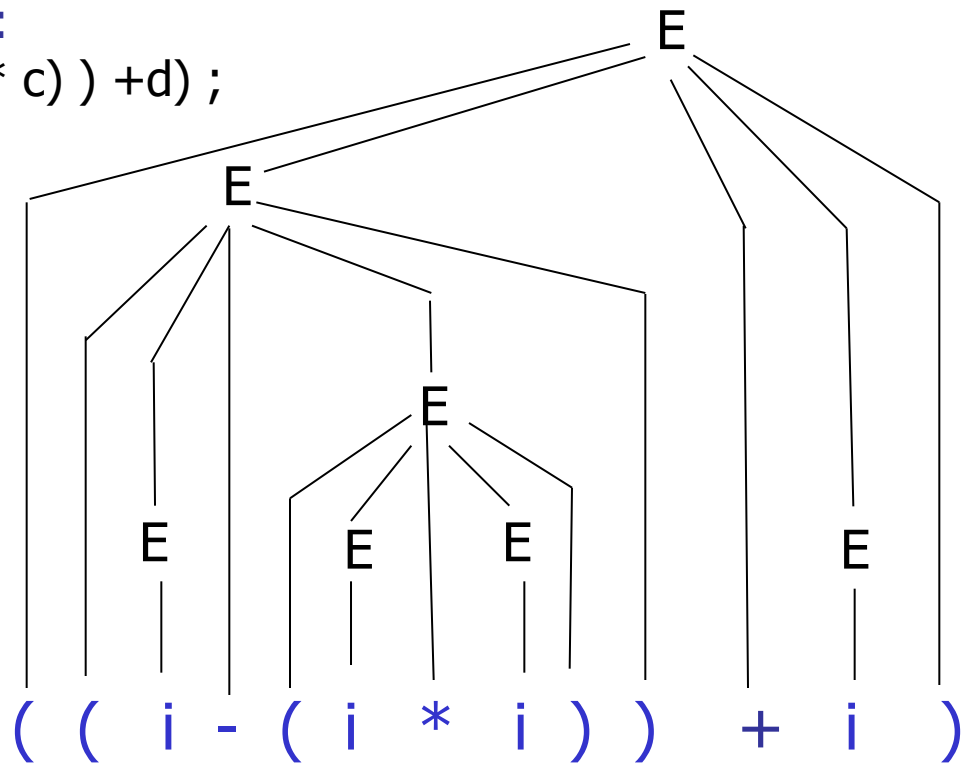
$E \to (E - E)$

$E \to (E * E)$

$E \to (E / E)$

$E \to i$

Грамматика однозначна!



- Первые трансляторы с языка Фортран требовали именно такого представления арифметических выражений

Можно ли изменить грамматику?

- Для любого языка можно построить бесконечное множество грамматик Хомского, порождающих этот язык
- Можно ли построить грамматику Хомского языка арифметических выражений получше, чем G_0^E, НЕдвусмысленную грамматику, но без обязательных скобок, определяющих порядок выполнения операций??

МОЖНО!!!

Изменение грамматики арифметических выражений

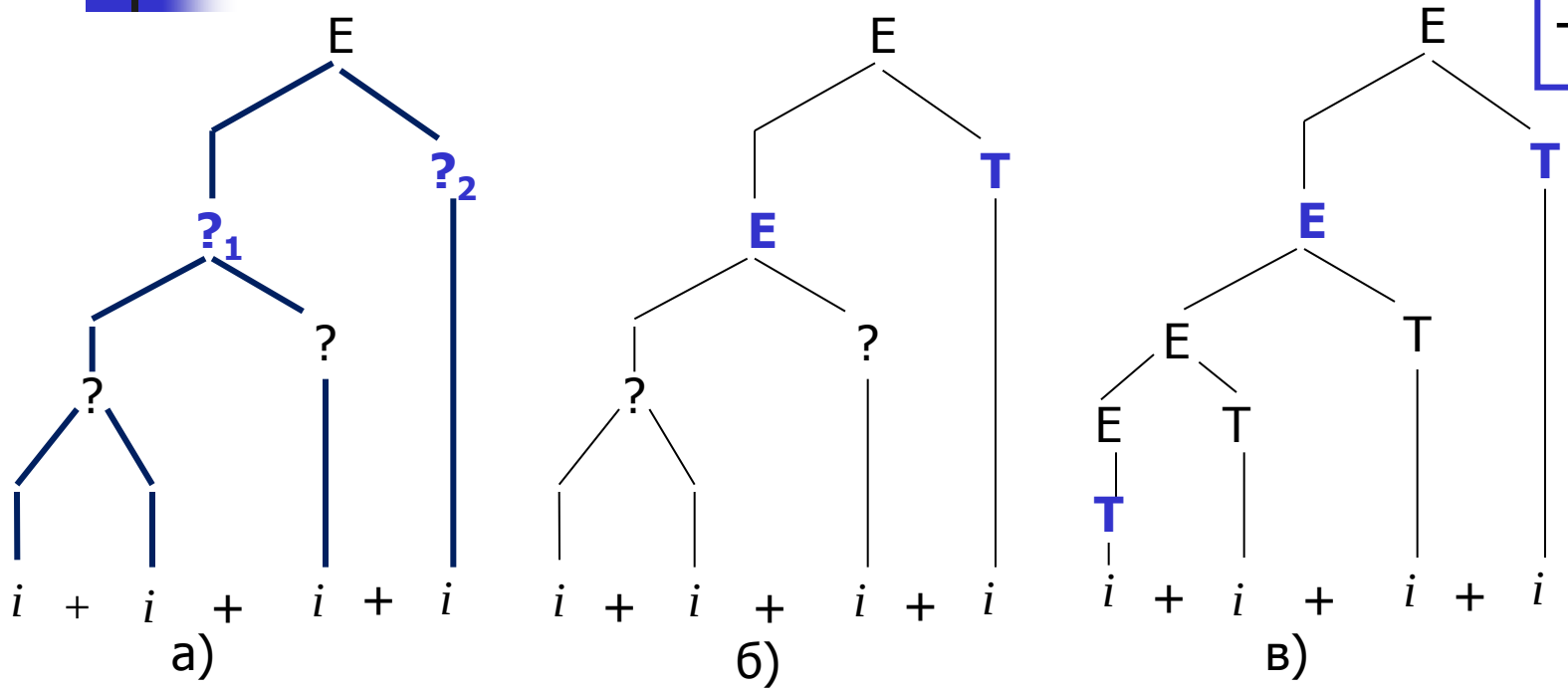
Попробуем построить новую грамматику, эквивалентную грамматике G^0_E, но отличающуюся от нее удобством семантических вычислений атрибутной грамматикой: результат сложного арифметического выражения вычисляется как функция атрибутов подвыражений, его составляющих

Избавимся сначала от двусмысленности грамматики. Рассмотрим простейший вид арифметических выражений - арифметические выражения с одной бинарной операцией, которую обозначим '$\otimes$'.

Для любой цепочки операндов, связанных одной операцией, в новой грамматике мы должны иметь только одно дерево вывода, отражающее общепринятый порядок вычисления значения выражения: слева направо для одной операции

Грамматика арифметических выражений с операцией сложения, выполняющейся слева направо

$E \to E + T$
$E \to T$
$T \to i$



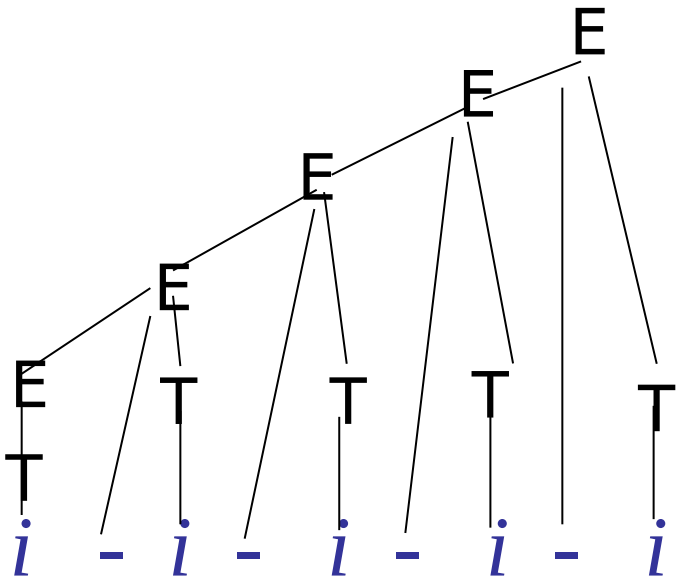
- а) отражает желаемую структуру. Вся цепочка должна выводиться из начального символа, поэтому корень дерева помечен E
- Цепочка '*i* + *i* + *i*', выводящаяся из $?_1$ (рис. а), - имеет ту же структуру "арифметическое выражение", что и вся цепочка, в отличие от оставшейся части выражения, выводящейся из $?_2$

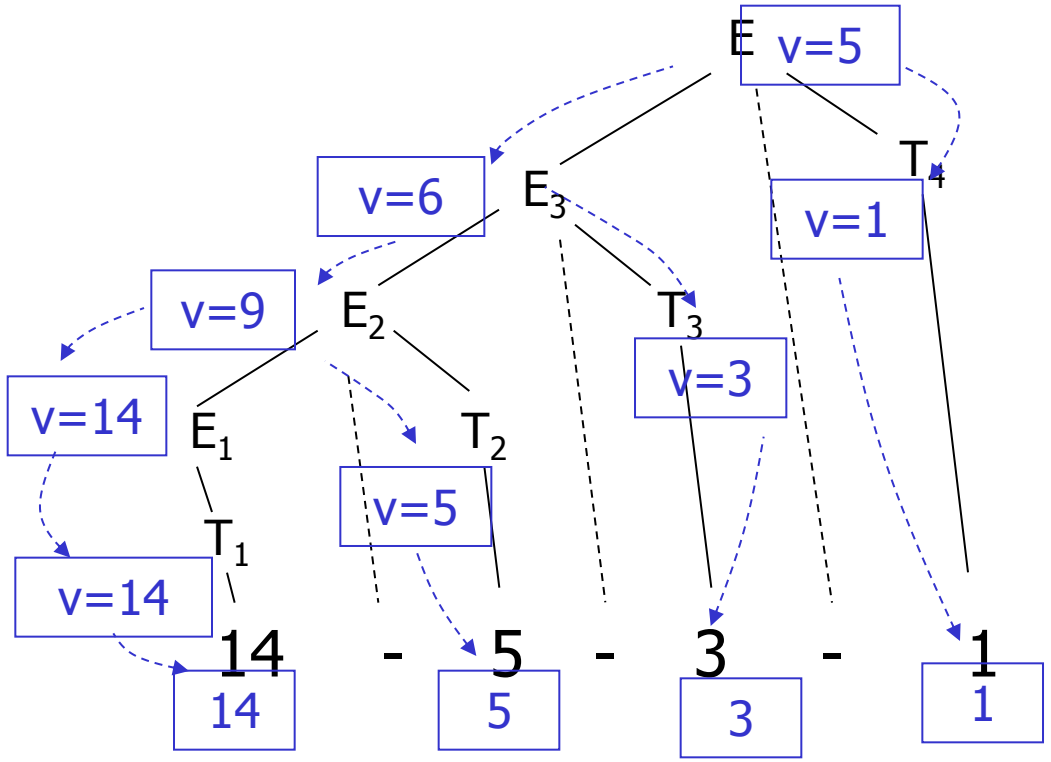
Поэтому в узле дерева вместо $?_1$ должен стоять нетерминал E, а в узле $?_2$ – другой нетерминал, который назовем T (рис. б)

Атрибутная грамматика для одной бинарной операции, выполняющейся слева направо

G^1_E:: $E \to E \otimes T$
$E \to T$
$T \to i$

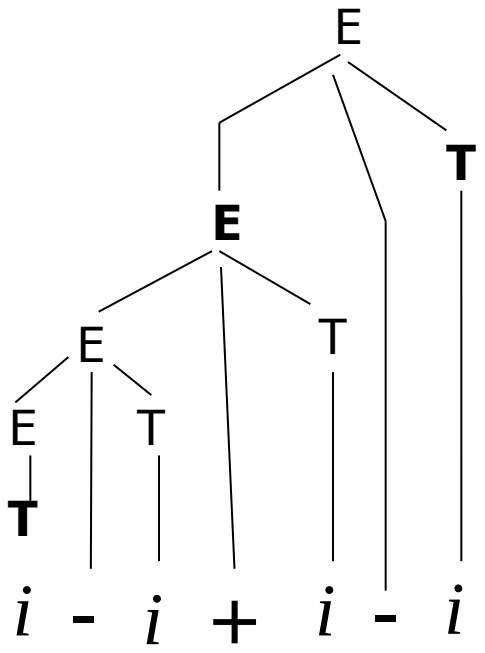
Для любых арифметических выражений с одной операцией G^1_E позволяет построить единственное дерево вывода, отражающее общепринятую семантику выполнения повторяющейся бинарной операции слева направо:





Грамматика Недвусмысленна и дает нужную структуру

Две бинарные операции одного приоритета: + и -



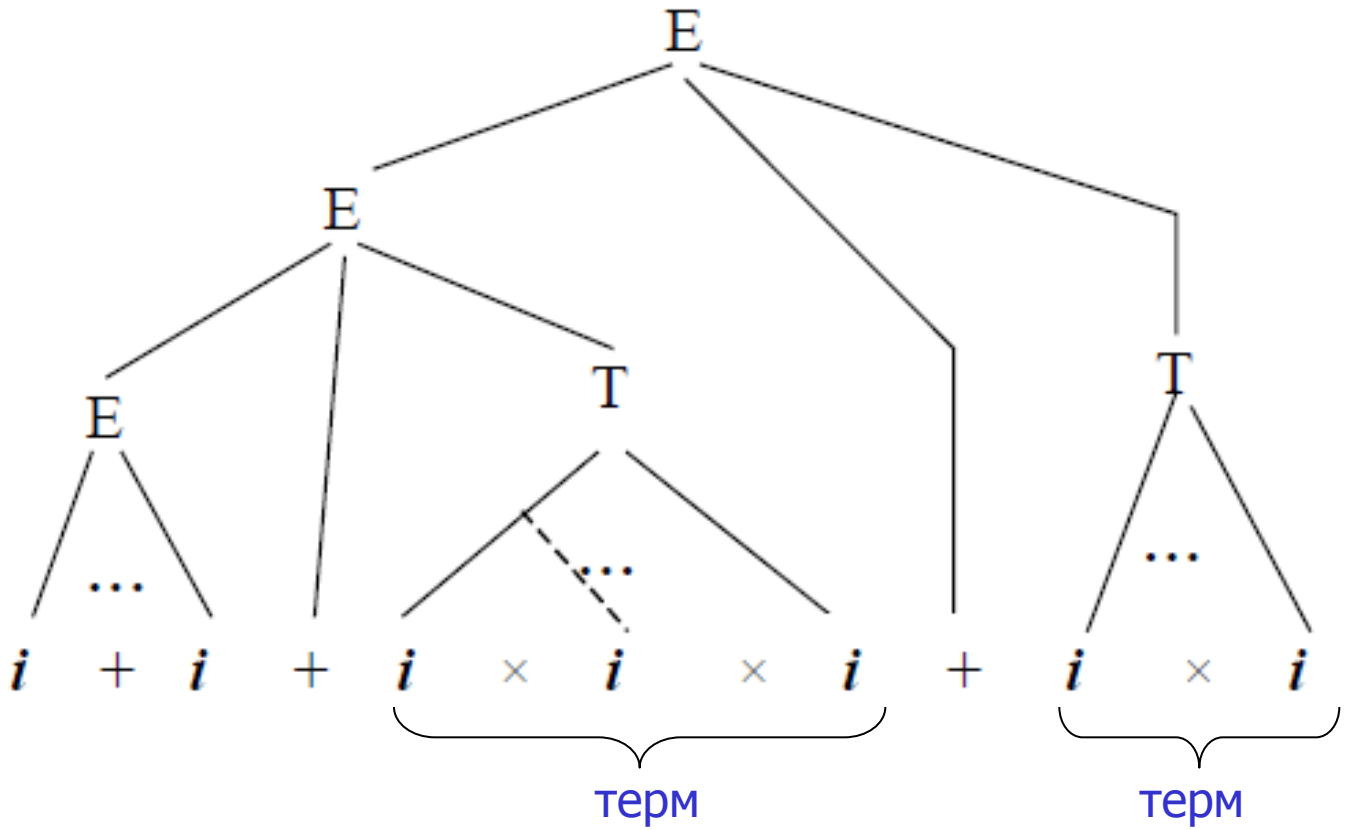
$E \to E + T$
$E \to E - T$
$E \to T$
$T \to i$

$E \to E$ addop T
$E \to T$
$T \to i$

- С точки зрения СИНТАКСИСА (структуры), операции + и – полностью эквивалентны. Поэтому обычно на этапе лексического анализа строят лексему addop - ‘операция типа сложения’

Грамматика арифметических выражений с учетом приоритетов

Построим грамматику арифметических выражений с двумя операциями, ‘+’ и ‘×’.
Операция '×', имеет больший приоритет, чем '+'
Рассмотрим "желаемую" структуру дерева вывода цепочки:



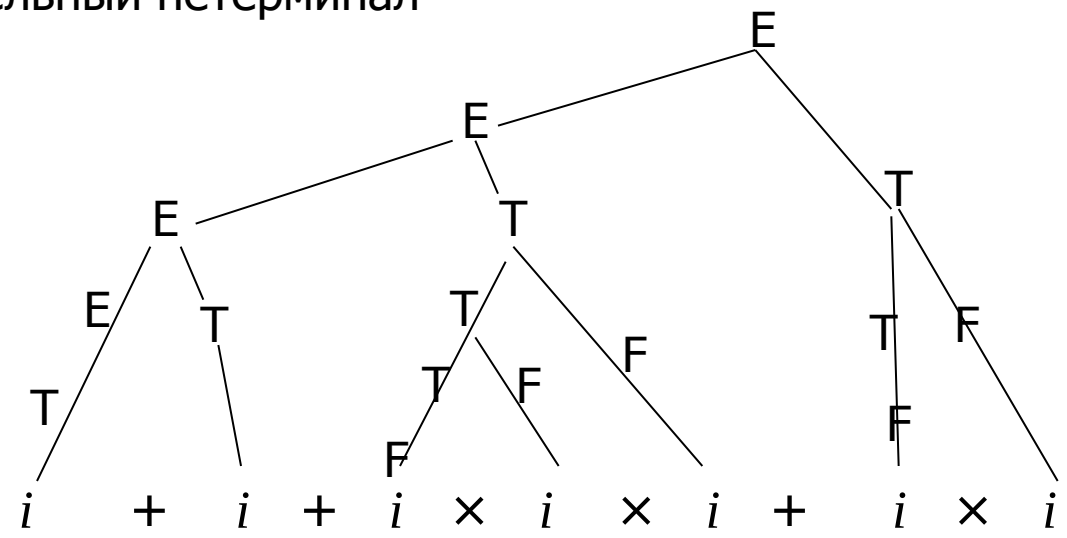
Грамматика арифметических выражений с учетом приоритетов

Более приоритетная операция '×' должна выполняться до того, как ее результат будет использоваться в вычислениях. Следовательно, в искомой грамматике группы операндов, связанные операцией '×', должны порождаться из нетерминала T так же, как в грамматике G^1_E порождается сам операнд

T должен играть роль начального символа грамматики, порождающей цепочки операндов, связанные операцией '×'

Так же, как в G^1_E из E порождались цепочки операндов, связанных операцией '+', с выполнением операций слева направо, из T должны порождаться цепочки операндов, связанных операцией '×' – и тоже с выполнением слева направо. Необходимо ввести дополнительный нетерминал

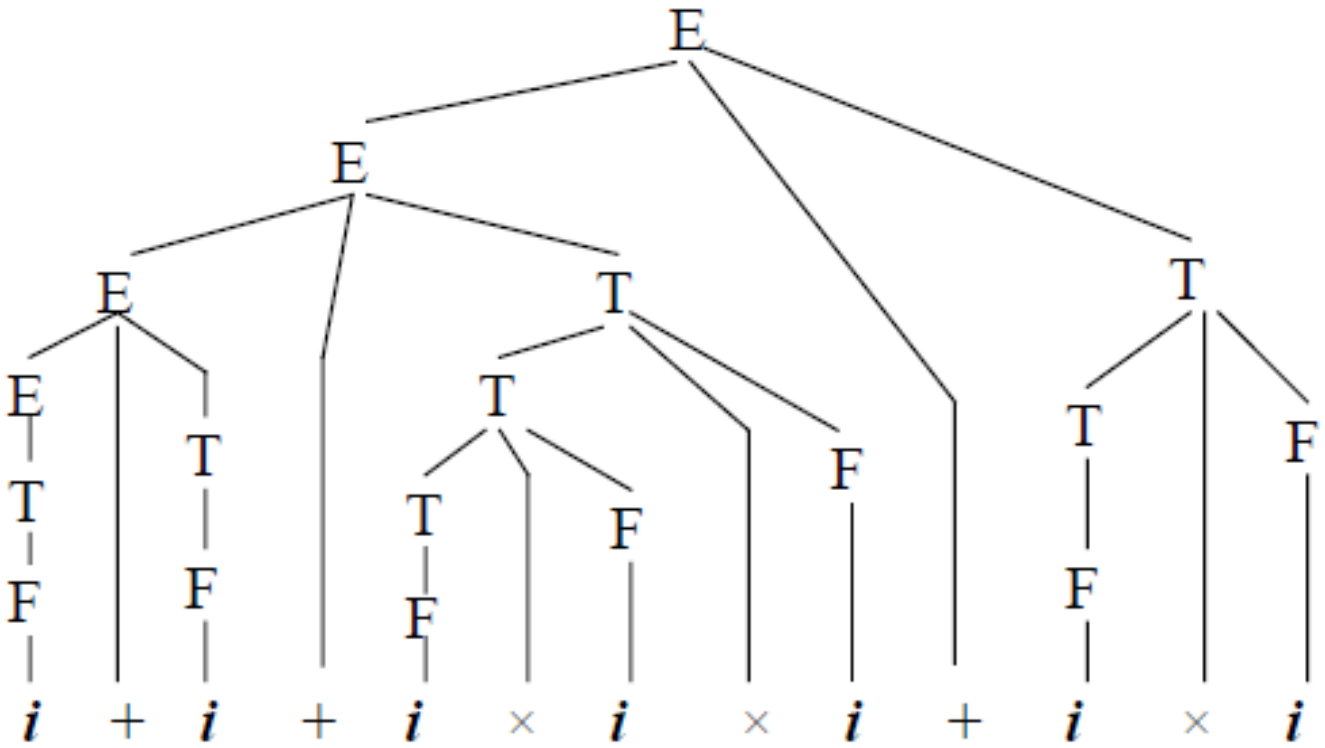
G^3_E::
$E \to E + T$
$E \to T$
$T \to T \times F$
$T \to F$
$F \to i$



Дерево вывода в грамматике G^3_E

Чем хороша эта грамматика? Она однозначна, и в ней простая семантика с использованием синтезируемых атрибутов дает правильные вычисления арифметических выражений

G^3_E:: $E \to E + T$
$E \to T$
$T \to T \times F$
$T \to F$
$F \to i$



Дерево вывода сложное, но оно отражает правильную последовательность выполнения операций без изменения языка!!

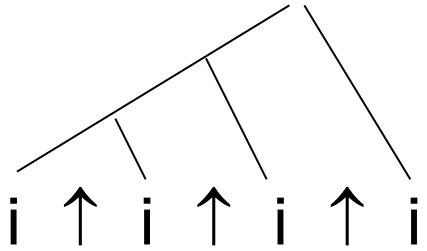
Добавление операции возведения в степень

- Нетерминал F должен быть начальным символом грамматики, порождающей фрагменты арифметического выражения, состоящие только из операндов и знаков возведения в степень. Пусть этот знак '↑'
  Грамматика

  $F \rightarrow F \uparrow P \mid P$

  $P \rightarrow i$

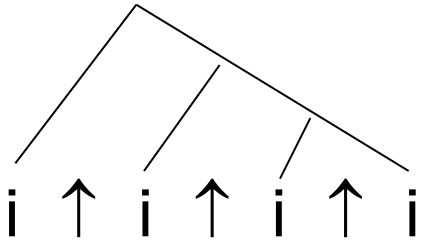
  порождает цепочки вида $i \uparrow i \uparrow i \uparrow i$.



- Вычисление значения выражения при естественном определении семантики будет здесь выполняться слева направо, т.е. *2↑3↑4* будет пониматься как (2↑3) ↑ 4 , т.е. 2↑(3×4)
- В некоторых языках программирования (Фортран) вычисление цепочки 2 ↑ 3 ↑ 4 понимается (и вычисляется!), как 2↑(3↑4).
- Иными словами, выполнение операций ↑ должно проводиться справа налево. Чтобы достичь этого при естественном определении семантики, нужно изменить грамматику в этой части:

  $F \rightarrow P \uparrow F \mid P$

  $P \rightarrow i$



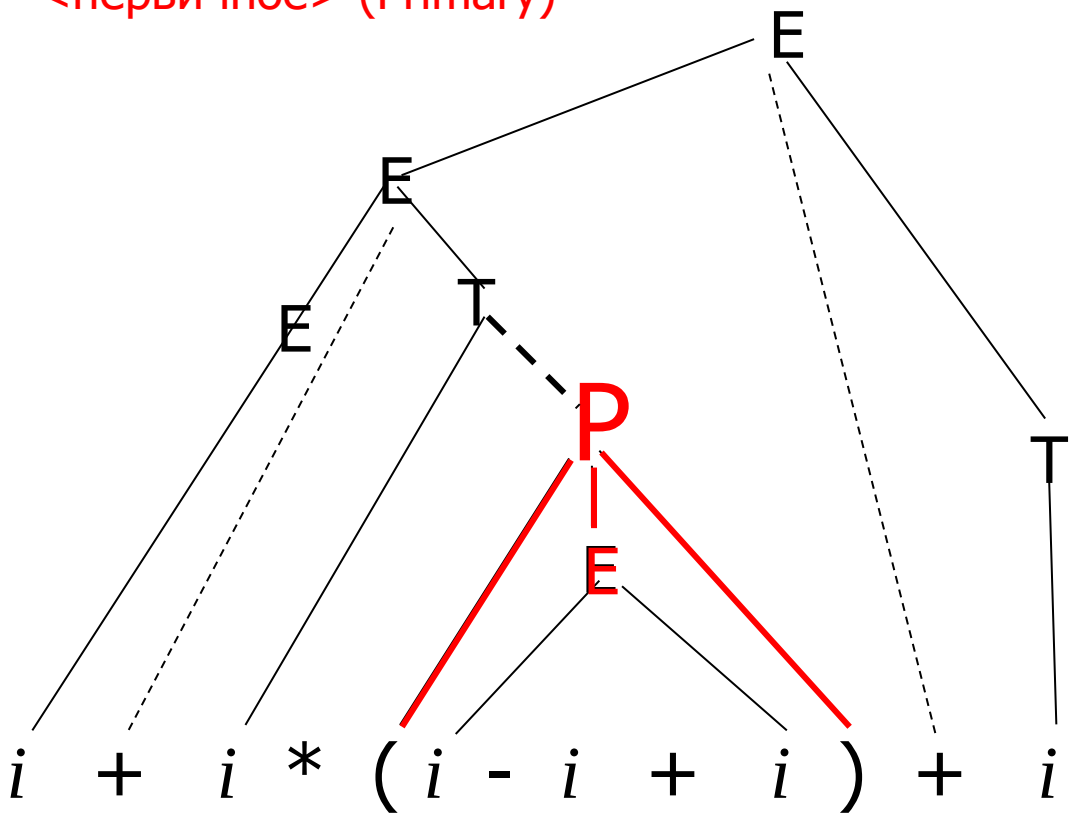
Использование скобок в арифметических выражениях

- Любое арифметическое выражение, взятое в скобки, рассматриваем как операнд на следующих этапах вычислений вверх по дереву вывода
- Вводим новый нетерминал: P - <первичное> (Primary)

G^5_E ::
E → E + T | E – T | T
T → T * F | T / F | F
F → P ↑ F | P
P→ *i* | (E)

Новые нетерминалы:
T-<терм>
F-<фактор>
P-<первичное>

Старая грамматика:
G^0_E:: E → E op E | (E) | i



Полная грамматика арифметических выражений

Старая грамматика: G^0_E:: E → E op E | (E) | i

Грамматика арифметических выражений:

G^5_E :: E → E + T | E – T | T
T → T * F | T / F | F
F → P ↑ F | P
P→ *i* | (E)

G^5_E :: E → E addop T | T
T → T mulop F | F
F → P ↑ F | P
P→ *i* | (E)

Новые нетерминалы:
T-<терм>,
F-<фактор>,
P-<первичное>, или <primary>

Часто используют

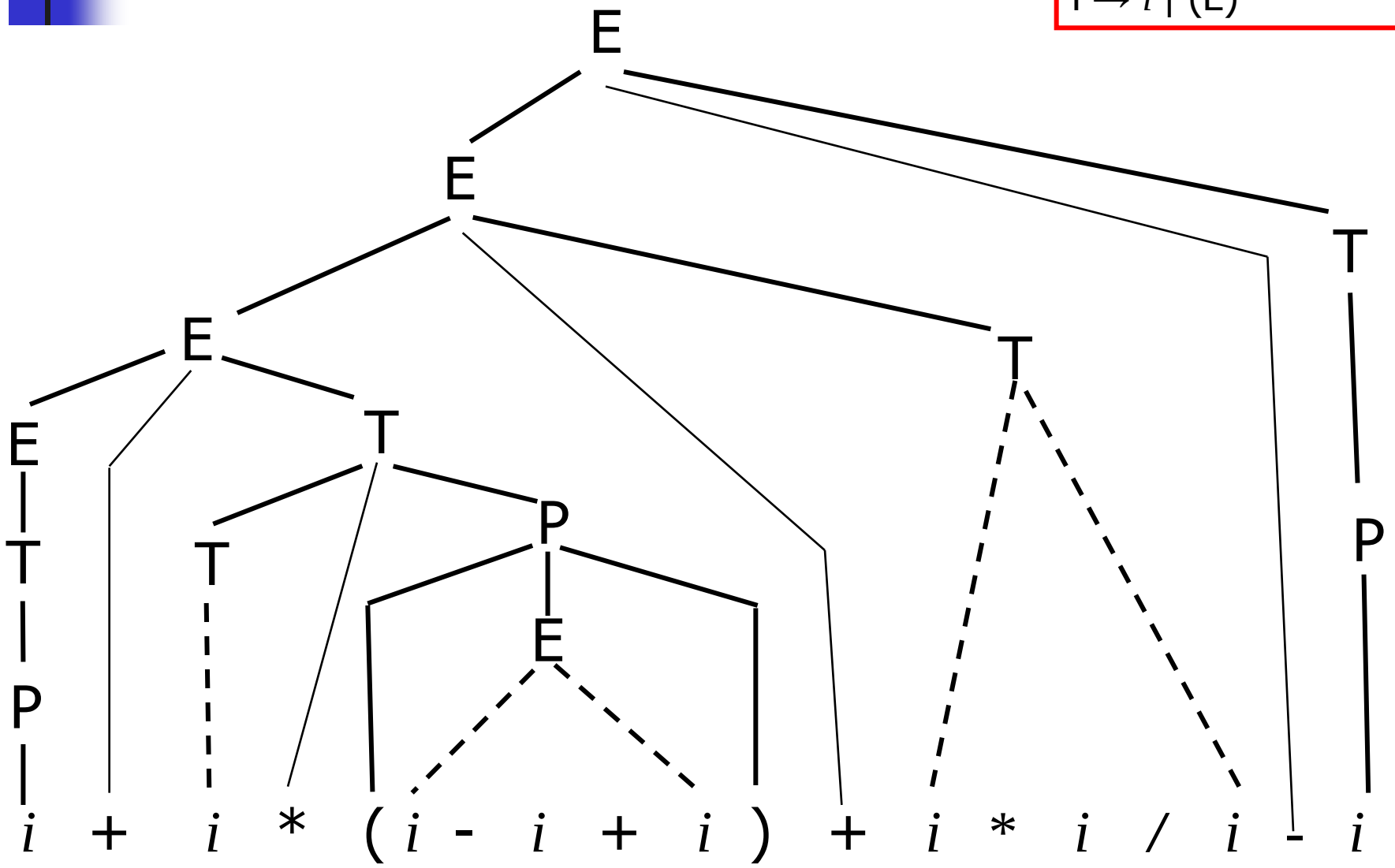
G_E :: E → E addop T | T
T → T mulop P | P
P→ *i* | (E)

G_E :: E → E + T | T
T → T * P | P
P→ *i* | (E)

1) G^5_E эквивалентна G^0_E, но однозначна
2) в G^5_E простая семантика с синтезируемыми семантическими атрибутами определяет простой порядок вычисления значения с учетом приоритетов операций и выполнение одноприоритетных операций слева направо (а возведение в степень – справа налево)

Пример построения дерева вывода

E → E addop T | T
T → T mulop P | P
P→ *i* | (E)



Пример 1: Интерпретация арифметических выражений

Атрибутная грамматика для интерпретации арифметических выражений

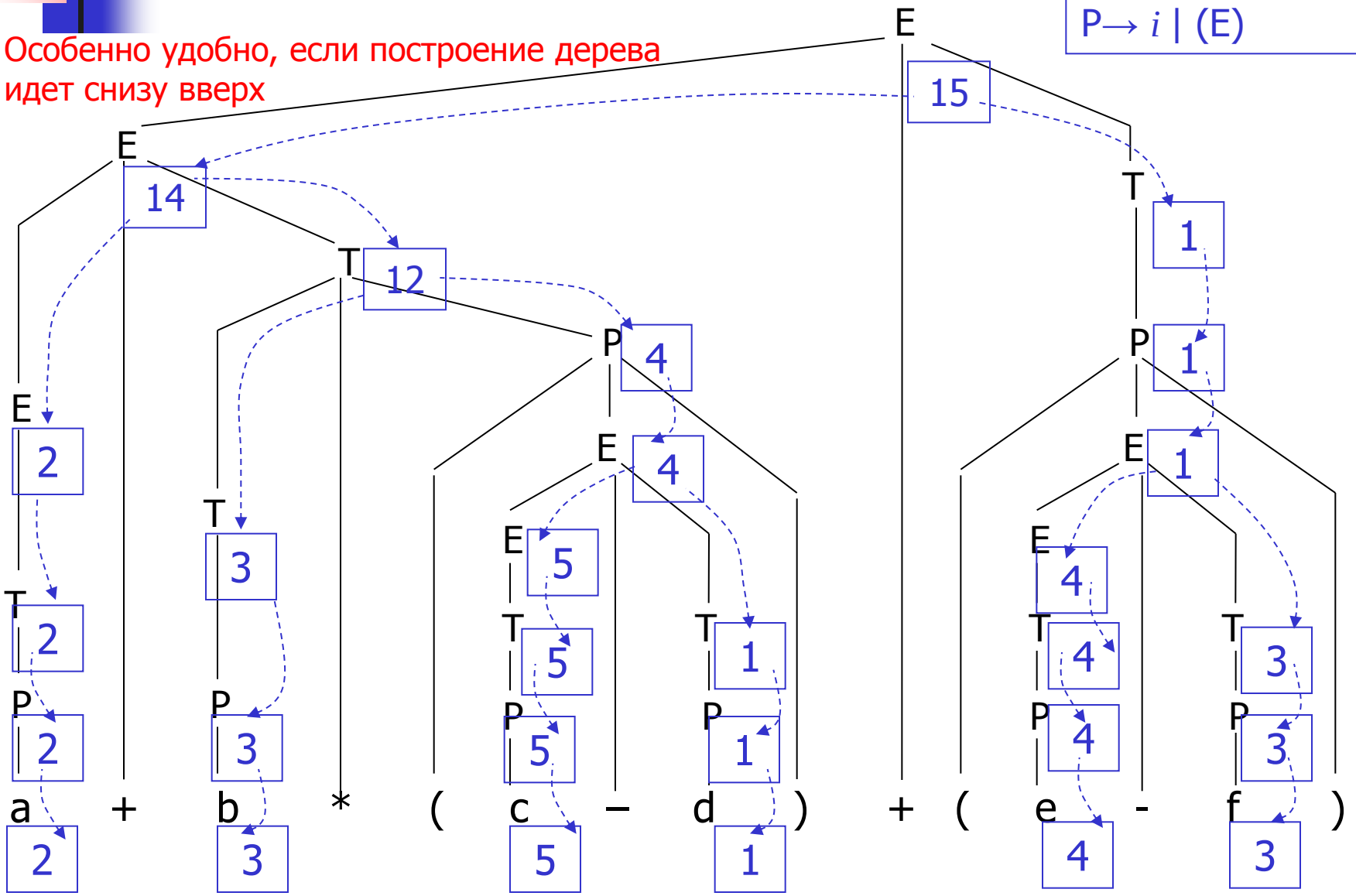
n	Синтаксис	Семантика
1	$E \rightarrow E'\ \underline{addop}_k\ T$	$\mu(E) := (k=0)?\ \mu(E') + \mu(T)\ :\ \mu(E') - \mu(T)$
2	$E \rightarrow T$	$\mu(E) := \mu(T)$
3	$T \rightarrow T'\ \underline{mulop}_k\ P$	$\mu(T) := (k=0)?\ \mu(T') * \mu(P)\ :\ \mu(T')\ /\ \mu(P)$
4	$T \rightarrow P$	$\mu(T) := \mu(P)$
5	$P \rightarrow i$	$\mu(P) := \mu(i)$
6	$P \rightarrow (E)$	$\mu(P) := \mu(E)$

μ - семантический атрибут, приписанным нетерминалам E, T и P во всех узлах дерева вывода исходного арифметического выражения.
μ(A) - значение выражения A. Задачей интерпретатора является вычисление значения $\mu(E_0)$, где E_0 – пометка корня дерева вывода

Построение атрибутированного дерева

$E \rightarrow E$ addop $T \mid T$
$T \rightarrow T$ mulop $P \mid P$
$P \rightarrow i \mid (E)$

Особенно удобно, если построение дерева идет снизу вверх



Пример 2: Представление арифметических выражений во внутренней форме (синтаксическое дерево)

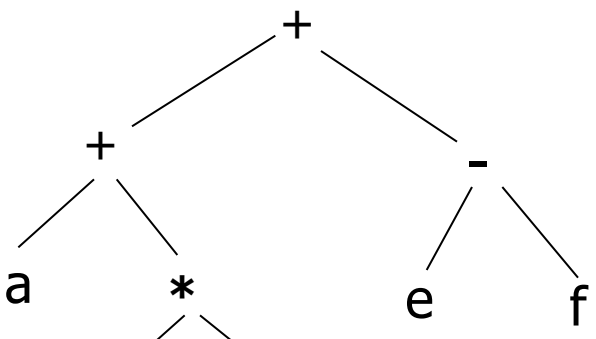
Дерево вывода и синтаксическое дерево

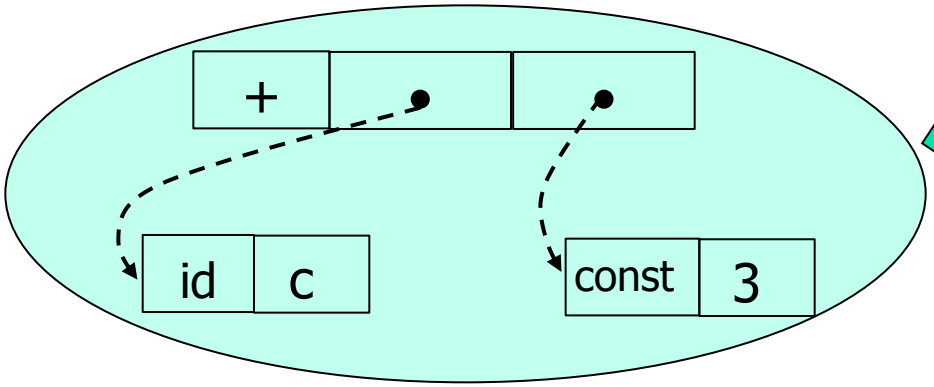
- Синтаксическим деревом арифметического выражения называется дерево, листьями которого являются операнды, а в узлах стоят операции, выполняемые над соответствующими подвыражениями

Как преобразовать??

??

a + b * (c – 3) + (e - f)





- Синтаксическое дерево очень компактно, оно используется для внутреннего представления арифметического выражения

Построение синтаксического дерева с помощью атрибутной семантики

Вводим семантические функции:

NewLeaf(a, entry) – функция строит лист a с указателем entry на таблицу имен

NewNode(op, L, R) – функция строит вершину с операцией op и указателями L и R

Вводим семантические атрибуты:

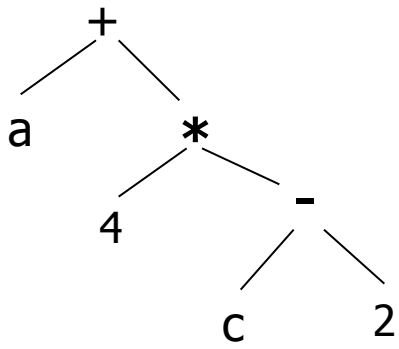
Для каждого нетерминала N вводим атрибут N.ptr – указатель на вершину, соответствующую подвыражению, выводимому из N

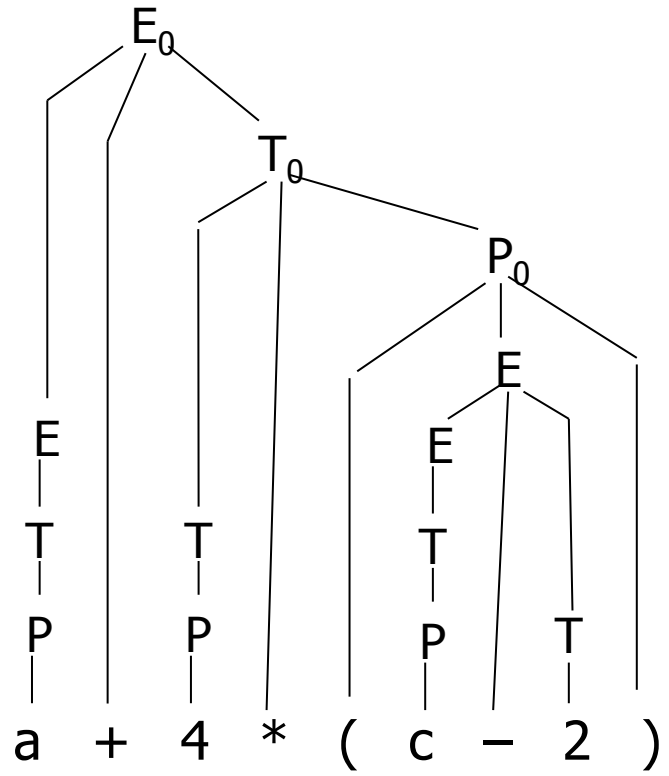
N	Синтаксис	Семантика
1	E → E′ + T	E.ptr := NewNode(op+, E′.ptr, T.ptr)
2	E → T	E.ptr := T.ptr
3	T → T′ * P	E.ptr := NewNode(op*, T′.ptr, P.ptr)
4	T → P	T.ptr := P.ptr
5	P → i_k	P.ptr := NewLeaf(id, k)
6	P → c_k	P.ptr := NewLeaf(const, k)
7	P → (E)	P.ptr := E.ptr

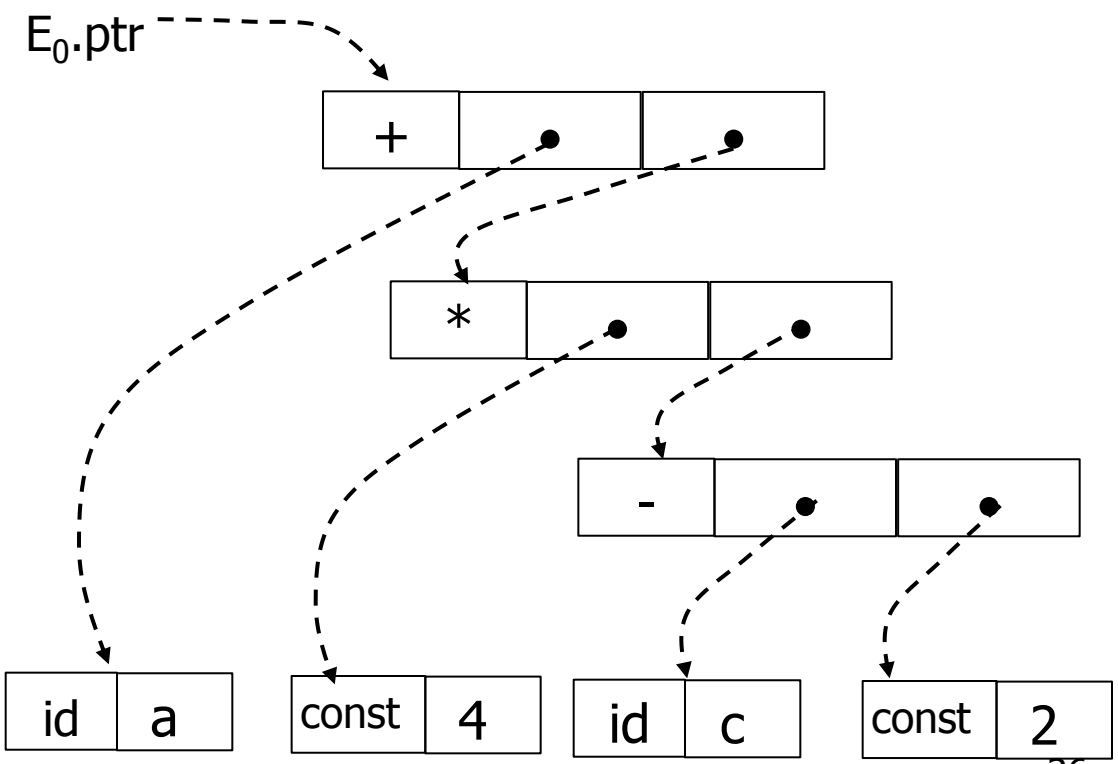
Синтаксическое дерево как результат трансляции

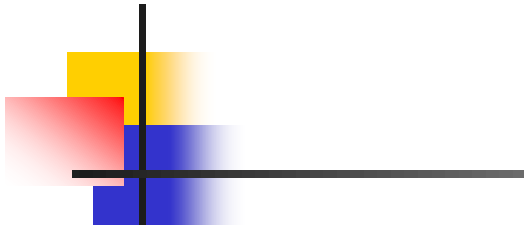
a + 4 * (c – 2)

Это – задача трансляции!









Пример 3: Польская инверсная запись

Перевод арифметических выражений в ПОЛИЗ

- ПОЛИЗ, или "*Польская инверсная запись*" – это специальная бесскобочная форма записи арифметических выражений, удобная для автоматического вычисления. Другое название этой формы – постфиксная запись, отражающая структуру выражения, при которой знак операции непосредственно следует за своими операндами
- Три формы записи выражения:
  - инфиксная a+b A := B + C * D
  - префиксная +a b := A + B * C D ('`:=`' понимается как операция)
  - постфиксная a b+ A B C D *+ :=
- Префиксную форму можно понимать как обычную функциональную запись $f_+(a,b)$, в которой опущены скобки и знак функции '+' стоит непосредственно перед операндами
- Постфиксная форма – функциональная запись $(a,b)f_+$. В ней опущены скобки, и знак операции стоит после своих операндов
- Удобство как префиксной, так и постфиксной форм состоит в том, что они, в отличие от обычной инфиксной формы записи, не требуют скобок. Операции (функции), используемые здесь, не обязательно бинарные и не обязательно арифметические

Арифметические выражения в разных формах

Примеры форм записи приведены в таблице (трехместная функция F(a,b,c), выдающая в качестве результата *b,* если а истинно, а в противном случае выдающая с, обозначена как '**?**')

N	Инфиксная форма	Префиксная форма	Постфиксная форма (ПОЛИЗ)
1.	*a+b*	*+ab*	*ab+*
2.	*a+b×c*	*+a×bc*	*abc×+*
3.	*a+b×c–d*	*–+a×bcd*	*abc×+d–*
4.	*a+b×(c–d) +(f–e)*	*++a×b–cd–fe*	*abcd–×+fe–+*
5	*g:=**if** a>c* **OR** *d **then** e–f **else** q*	*:=g ?* **OR***>acd–efq*	*gac>d***OR***ef–q ? :=*

Префиксная форма была введена в 1925 г. польским логиком Яном Лукасевичем, поэтому ее вариант – постфиксная запись – традиционно называется Польской инверсной записью, или ПОЛИЗ

ПОЛИЗ - удобная форма: простой алгоритм с использованием стека позволяет вычислить значение выражения.

Пример: вычисление арифметического выражения

ПОЛИЗ(a+b×(c–d)–f+e) = abcd–×+fe – +

N	Стек	Очередной входной символ	Остаток входной цепочки	Выполняемая операция	Примечание
1	$	*a*	*bcd*–×+*fe*– +$	*a* – в стек	Стек пуст. Первый символ входной цепочки - операнд
2	$ *a*	*b, c, d*	–×+*fe*– +$	*b, c, d* – в стек	Три однотипных шага
3	$ *abcd*	–	×+*fe*– +$	x_1:=*c*-*d*; x_1 – в стек	Выполнение операции '–'
4	$ *ab* x_1	×	+*fe*– +$	x_2:= *b*×x_1; x_2– в стек	Выполнение операции '×'
5	$ *a*x_2	+	*fe*– +$	x_3:= *a*+x_2; x_3– в стек	Выполнение операции '+'
6	$ x_3	*f, e*	– +$	*f, e* – в стек	Два однотипных шага
7	$ x_3 *f e*	–	+$	x_4:= *f*–*e*; x_4 – в стек	Выполнение операции '–'
8	$ x_3 x_4	+	$	x_5:= x_3+x_4; x_5 – в стек	Выполнение операции '+'
9	$ x_5	$		Вычисления закончены	Результат – в верхушке стека

Пример 2: вычисление входной цепочки gac>dORef–q ? :=

g:= if a > c OR d then e - f else q

? – операция с тремя операндами

N	Стек	Очередной входной символ	Остаток входной цепочки	Выполняемая операция	Примечание
1	$	*g, a, c*	*>d***OR***ef–q* **?**:= $	*g, a, c* – в стек	Три однотипных шага
2	$ *g a c*	>	*d***OR***ef–q***?**:= $	x_1:= *a>c*; x_1 – в стек	Выполнение сравнения; $x_1 \in$ {**TRUE**, **FALSE**}
3	$ *g* x_1	*d*	**OR***ef–q***?**:= $	*d* – в стек	
4	$ *g* x_1 *d*	**OR**	*ef–q***?**:= $	x_2:= x_1**OR** *d*; x_2 – в стек	Выполнение операции; $x_2 \in$ {**TRUE**, **FALSE**}
5	$ *g* x_2	*e, f*	*–q***?**:= $	*e, f* – в стек	Два однотипных шага
6	$ *g* x_2 *e f*	–	*q***?**:= $	x_3:=e–*f*; x_3 – в стек	Выполнение операции;
7	$ *g* x_2 x_3	*q*	**?**:= $	*q* – в стек	Запись в стек
8	$ *g* x_2 x_3 *q*	**?**	:= $	x_4:=**?**(x_2,x_3,*q*); x_4 – в стек	**?** =если x_2 истинно, то x_3, в противном случае *q*
9	$ *g* x_4	:=	$	*g* ←x_4	по адресу *g* записывается значение x_4
10	$	$		Вычисления закончены	Стек пуст; результат записан по адресу *g*

Преобразование выражения в ПОЛИЗ

Итак, по конструкции программы, приведенной в форму ПОЛИЗ, выполнение вычислений с помощью стека очень просто
ПОЛИЗ часто используется, как промежуточная структура данных при компиляции. Как перевести арифметическое выражение в ПОЛИЗ?
Существуют различные алгоритмы такого преобразования

Рассмотрим алгоритм, основанный на синтаксически-ориентированной трансляции.
Исходное множество объектов для такого алгоритма – множество цепочек, результаты – тоже цепочки символов
Поэтому можно считать искомый алгоритм транслятором, который осуществляет преобразование выражения в ПОЛИЗ на основе структуры входных цепочек

Преобразование арифметического выражения в ПОЛИЗ

Построим атрибутную грамматику арифметических выражений. Синтаксис:

G_E : E → E $addop_k$ T | T
T → T $mulop_k$ P | P
P→ i_k | (E)

Нашей целью является формирование постфиксной формы любого входного арифметического выражения. Обозначим μ(A) постфиксную форму выражения A, т.е. μ - семантический атрибут, приписанный нетерминалам E,T и P

Атрибутная грамматика для трансляции арифметических выражений в ПОЛИЗ

Если структура некоторого фрагмента выражения E_0 есть $E_1 op\ T$, то его постфиксная форма $\mu(E_0)$ может быть построена так:

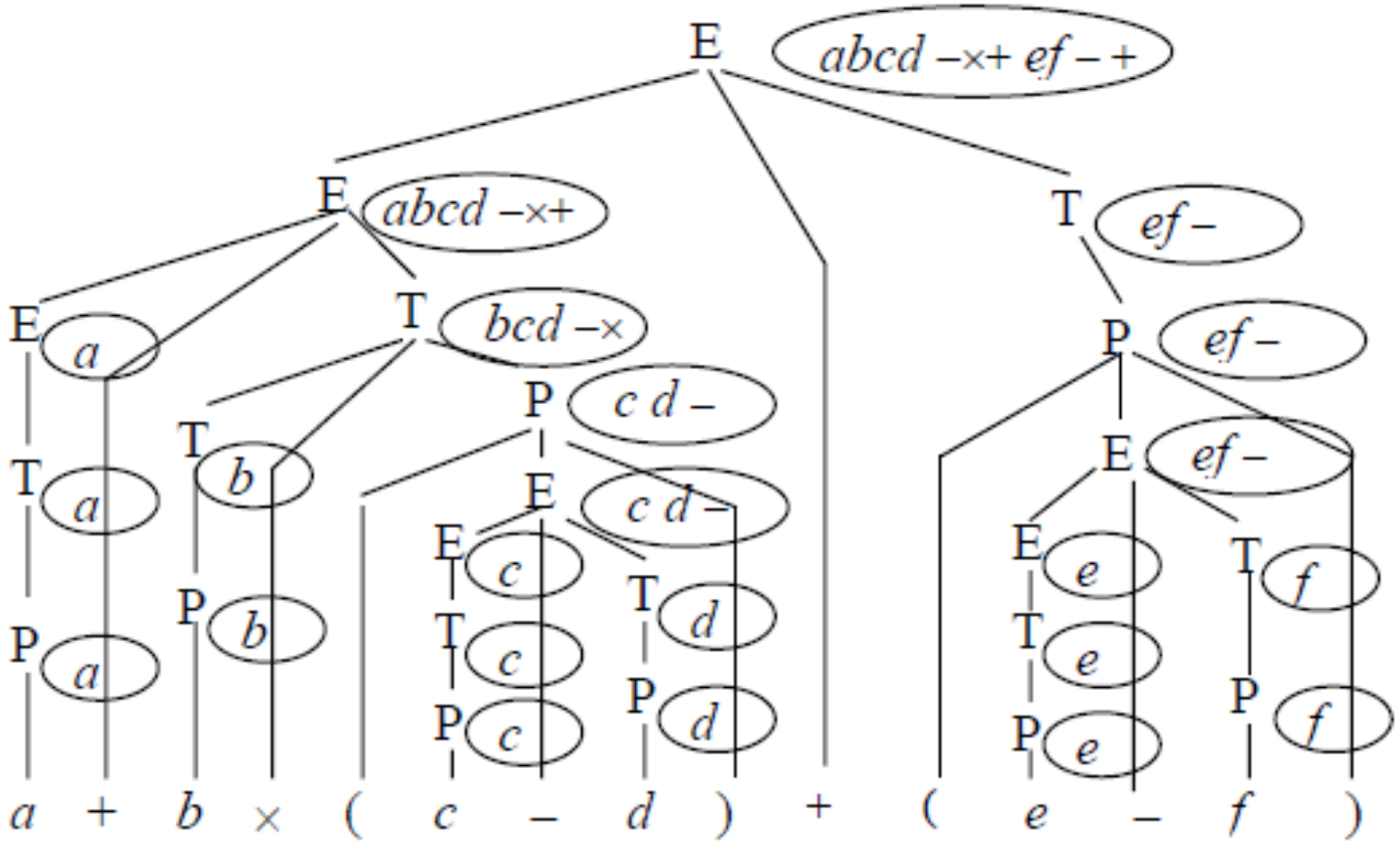
$$\mu(E_0) = \mu(E_1)\mu(T)op$$

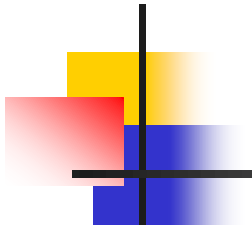
Это приводит к следующей атрибутной грамматике арифметических выражений:

N	Продукции	Семантические правила
1.	$E_0 \rightarrow E_1 omc\ T$	$\mu(E_0) = \mu(E_1)\ \mu(T)omc$
2.	$E \rightarrow T$	$\mu(E) = \mu(T)$
3.	$T_0 \rightarrow T_1 omy\ P$	$\mu(T_0) = \mu(T_1)\ \mu(P)omy$
4.	$T \rightarrow P$	$\mu(T) = \mu(P)$
5.	$P \rightarrow (\ E\ )$	$\mu(P) = \mu(E)$
6.	$P \rightarrow \boldsymbol{i}$	$\mu(P) = \boldsymbol{i}$

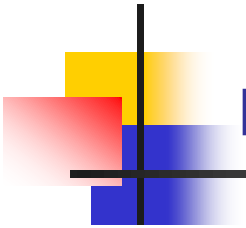
Построение ПОЛИЗ арифметического выражения по дереву вывода

Рассмотрим преобразование в ПОЛИЗ выражения *a+b×(c–d) +(e–f).* Будем считать, что дерево вывода этой цепочки построено синтаксическим анализатором, и стоит задача вычисления ПОЛИЗ этого выражения по дереву его вывода в атрибутной грамматике





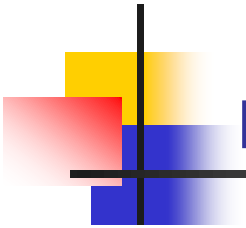
Пример 4: компиляция арифметических выражений



Компиляция арифметических выражений

На основе дерева вывода, построенного при синтаксическом анализе, транслятор языков программирования высокого уровня обычно генерирует некоторое промежуточное представление исходной программы

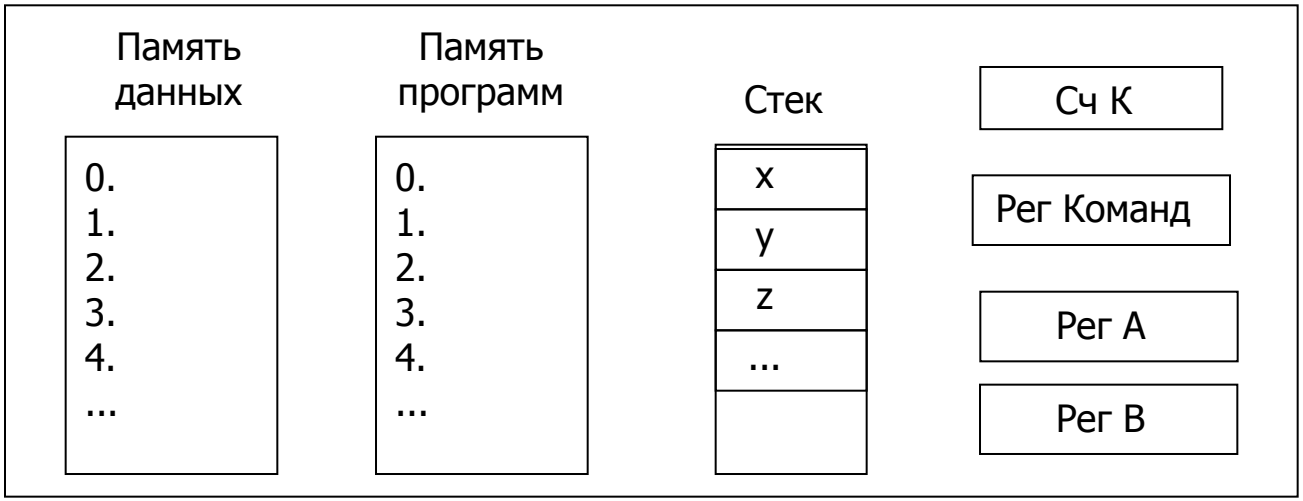
Такое представление может иметь различные формы; часто для этого используют постфиксную нотацию, трехадресный код либо так называемый *p*-код. Два последних представления можно понимать как коды ассемблера для некоторых виртуальных машин, предоставленных в распоряжение программиста. С этих кодов может легко быть проведена трансляция в ассемблер любой машины



p-код

- Вместо непосредственной трансляции в машинный код для конкретной платформы, с 70-х годов принята концепция двухуровневой трансляции: сначала со входного языка происходит компиляция в байт-код (или p-код), некоторой виртуальной машины – стековой машины, а потом этот код интерпретируется. Эта концепция аппаратно-независимой трансляции
- Вместо того чтобы транслировать код, написанный на высокоуровневом языке, в машинный код конкретной модели процессора, привязывая таким образом программу к аппаратной платформе, можно транслировать высокоуровневый код в специальный, независимый от модели процессора байткод, который во время исполнения будет интерпретироваться-выполняться в целевом процессоре
- Для того чтобы перенести существующее ПО на новую аппаратную платформу, достаточно создать лишь среду исполнения для p-кода, которая обеспечит уровень совместимости со средами исполнения, реализованными на другой аппаратной платформе

Стековая машина



- Пересылки
  LOAD a
  STORE a
  PUSH 25
  POP
- Арифметика:
  ADD
  SUB
  MULT
  DIV

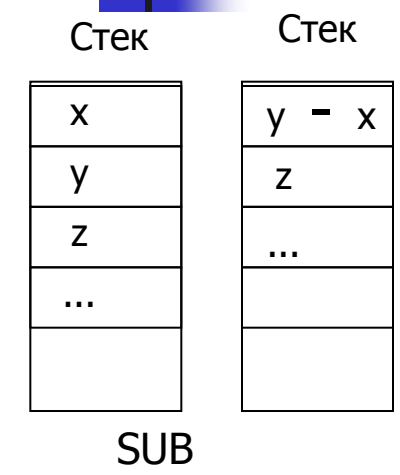
LOAD <адрес>	загрузка из адреса памяти в стек
STORE <адрес>	запоминание значения верхушки стека в памяти по адресу
PUSH <число>	число записывается в верхушку стека
POP	верхушка стека выталкивается
ADD SUB MULT	сложение, вычитание, ... (безадресные операции)
COMPARE <код>	загрузка из адреса памяти в верхушку стека

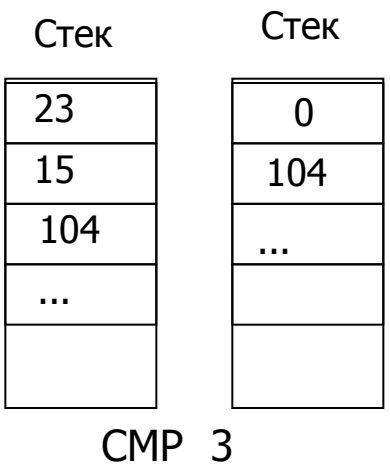
- Сравнение:
  COMPARE k

k=0? '='
k=1? '!='
k=2? '<'
k=3? '>'
k=4? '≤'
k=5? '≥'

Пример выполнения стековых команд

$a - b \le c + d \times f$



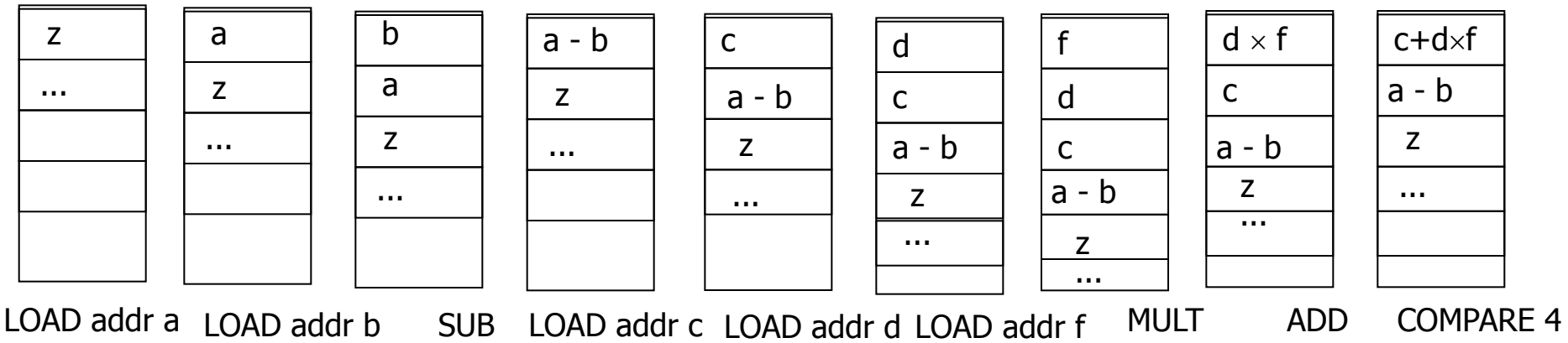


15 > 23 ?

q_k = k=0? '='
k=1? '≠'
k=2? '<'
k=3? '>'
k=4? '≤'
k=5? '≥'

```
LOAD    addr a
LOAD    addr b
SUB
LOAD    addr c
LOAD    addr d
LOAD    addr f
MULT
ADD
COMPARE    4
```

$a - b \le c + d \times f$



Атрибутная грамматика для трансляции условных выражений в *p*-код

Условные выражения Милана:

$a + b \times (c - d) > e - f$

$a - b \le (c + d) \times f$

$B \to E\ \underline{rel}_k\ E$
$E \to E\ \underline{addop}_k\ T \mid T$
$T \to T\ \underline{mulop}_k\ P \mid P$
$P \to i_k \mid (E)$

k=0? '='
k=1? '≠'
k=2? '<'
k=3? '>'
k=4? '≤'
k=5? '≥'

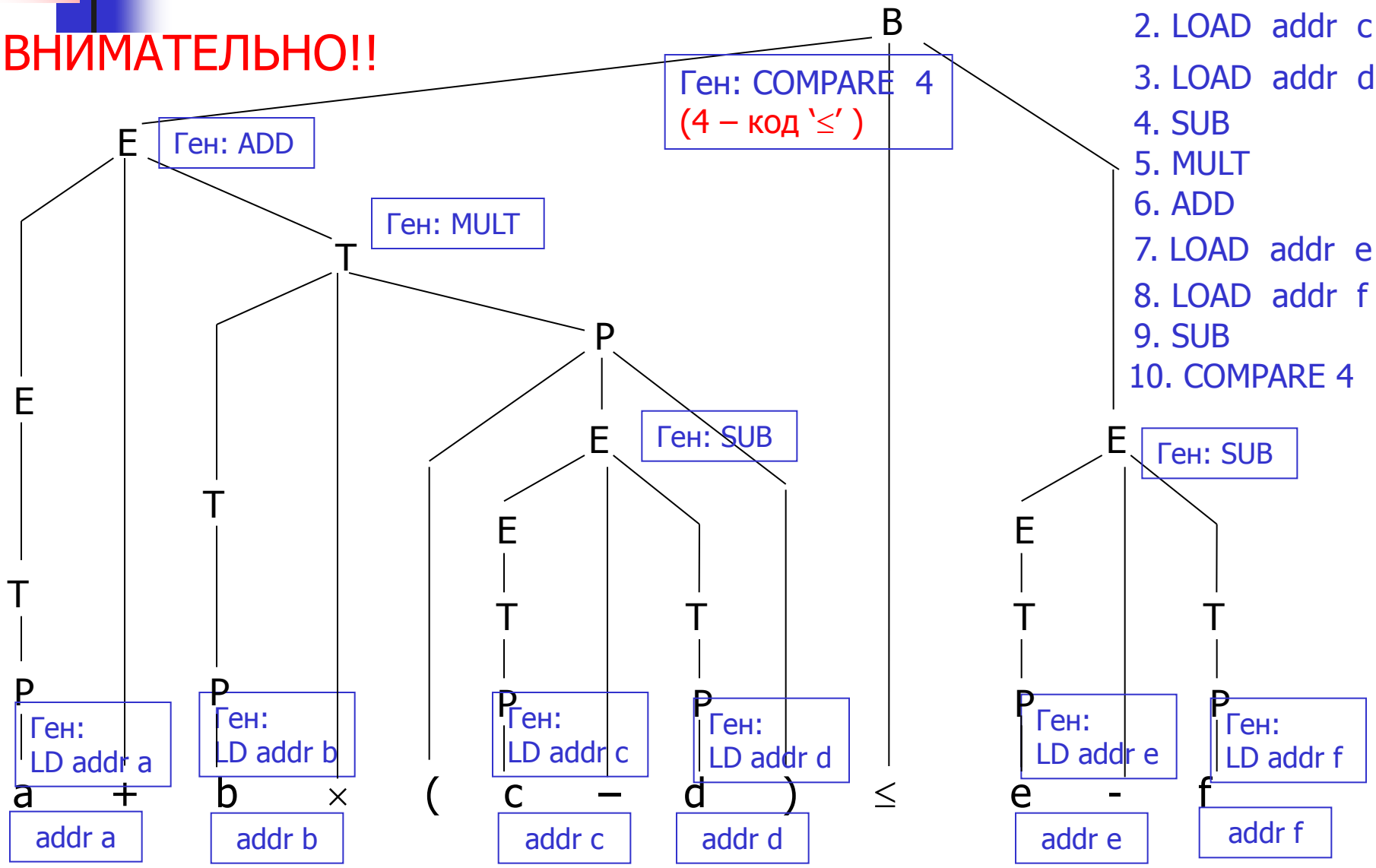
n	Синтаксическое правило	Семантическое правило
1	$B \to E'\ q_k\ E$	Ген: CMP k
2	$E \to E'\ \underline{addop}_k\ T$	Ген: (k=0)? ADD : SUB
3	$E \to T$	
4	$T \to T'\ \underline{mulop}_k\ P$	Ген: (k=0)? MULT : DIV
5	$T \to P$	
6	$P \to i_k$	Ген: LOAD addr i_k
7	$P \to (E)$	

Работает, если разбор идет снизу вверх (т.е. дерево строится лево, право, вершина)

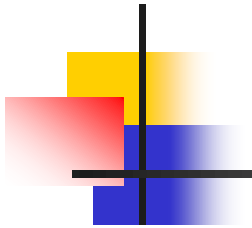
Если нет, то нужно подсчитывать адрес команды. Как?

Компиляция одновременно с построением дерева вывода снизу вверх для условного выражения

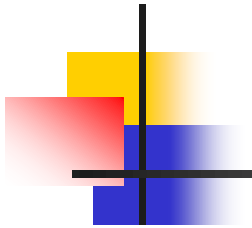
ВНИМАТЕЛЬНО!!



0. LOAD addr a
1. LOAD addr b
2. LOAD addr c
3. LOAD addr d
4. SUB
5. MULT
6. ADD
7. LOAD addr e
8. LOAD addr f
9. SUB
10. COMPARE 4

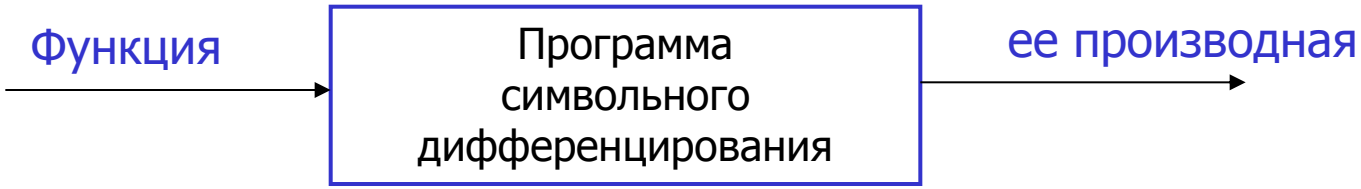


При построении дерева вывода снизу вверх последовательное выполнение семантических правил, связанных с каждым синтаксическим правилом в дереве вывода, изменяет глобальный атрибут – генерируемую программу – так, что в результате трансляции на выходе будет сгенерирована искомая программа



Пример 5. Символьное дифференцирование

Символьное дифференцирование



- Поскольку построение производной есть задача преобразования цепочек, то можно попробовать решить ее на основе синтаксически-ориентированной трансляции
- Этот подход, при котором в обрабатываемой цепочке восстанавливается ее структура, и правила преобразования цепочки строятся для ее структурных единиц, оказывается естественным и удобным при решении этой трудной проблемы математического анализа
- Записи функций, для которых строится производная – это обычные арифметические выражения, дополненные функциональными записями. При этом явно разделяются переменные и константы
- G_E: $E \rightarrow E+T \mid T$
  $T \rightarrow T \times P \mid P$
  $P \rightarrow x \mid c \mid (E) \mid Sin(E) \mid Cos(E)$

Атрибутная грамматика для символьного дифференцирования

Построение производной функции производится на основе правил, которые определены в матанализе:

производная суммы есть сумма производных,
производная константы есть ноль, и т.д.

Это есть, фактически, семантические правила атрибутной грамматики, порождающей аналитические записи функций

Построим эту атрибутную грамматику. В качестве атрибутов нетерминалов E, T, P выберем два.

параметр ψ(N) нетерминала N означает цепочку терминальных символов (подвыражение) выведенное из этого нетерминала N

параметр φ (N) нетерминала N означает цепочку терминальных символов - производную подвыражения, выводимого из N

Атрибутная грамматика для символьного дифференцирования

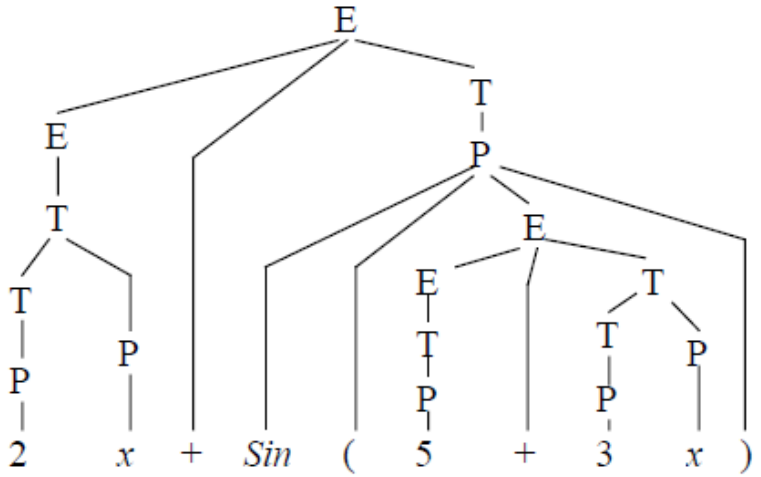
N	Продукции	Семантические правила	
1.	$E_0 \rightarrow E_1+T$	$\varphi(E_0) = \varphi(E_1) + \varphi(T);$	$\psi(E_0) = \psi(E_1) + \psi(T)$
2.	$E \rightarrow T$	$\varphi(E) = \varphi(T);$	$\psi(E) = \psi(T)$
3.	$T_0 \rightarrow T_1 P$	$\varphi(T_0) = \varphi(T_1)\,\psi(P) + \psi(T_1)\,\varphi(P);$	$\psi(T_0) = \psi(T_1)\,\psi(P)$
4.	$T \rightarrow P$	$\varphi(T) = \varphi(P);$	$\psi(T) = \psi(P)$
5.	$P \rightarrow Sin(E)$	$\varphi(P) = Cos(\psi(E))\,(\varphi(E));$	$\psi(P) = Sin(\psi(E))$
6.	$P \rightarrow Cos(E)$	$\varphi(P) = -Sin(\psi(E))\,(\varphi(E)),$	$\psi(P) = Cos(\psi(E))$
7.	$P \rightarrow \boldsymbol{x}$	$\varphi(P) = 1;$	$\psi(P) = \boldsymbol{x}$
8.	$P \rightarrow \boldsymbol{C}$	$\varphi(P) = 0;$	$\psi(P) = \boldsymbol{C}$
9.	$P \rightarrow (E)$	$\varphi(P) = \varphi(E);$	$\psi(P) = (\psi(E))$

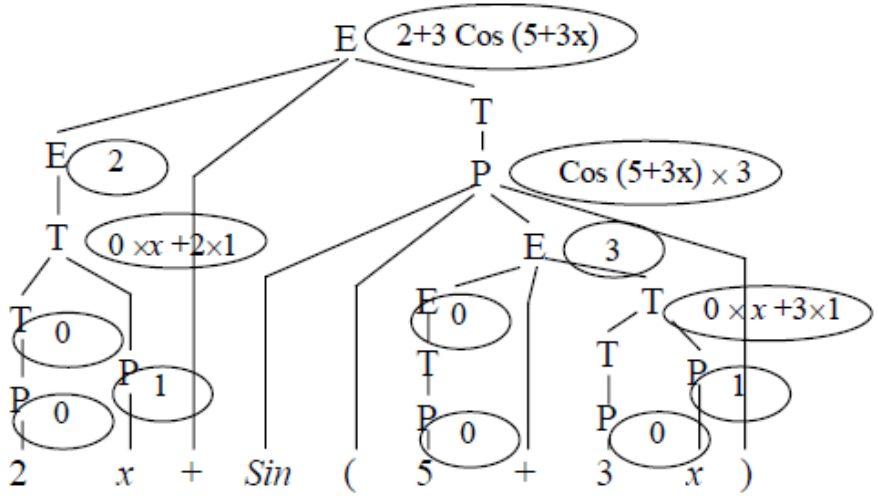
ψ(N) - цепочка терминальных символов выведенная из нетерминала N

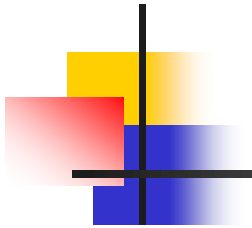
φ(N) - производная выражения, выведенного из N

Символьное дифференцирование выражения 2x + Sin(5+3x)

На дереве указаны только значения атрибута φ нетерминалов, т.е. производная выражения, выводимого из N





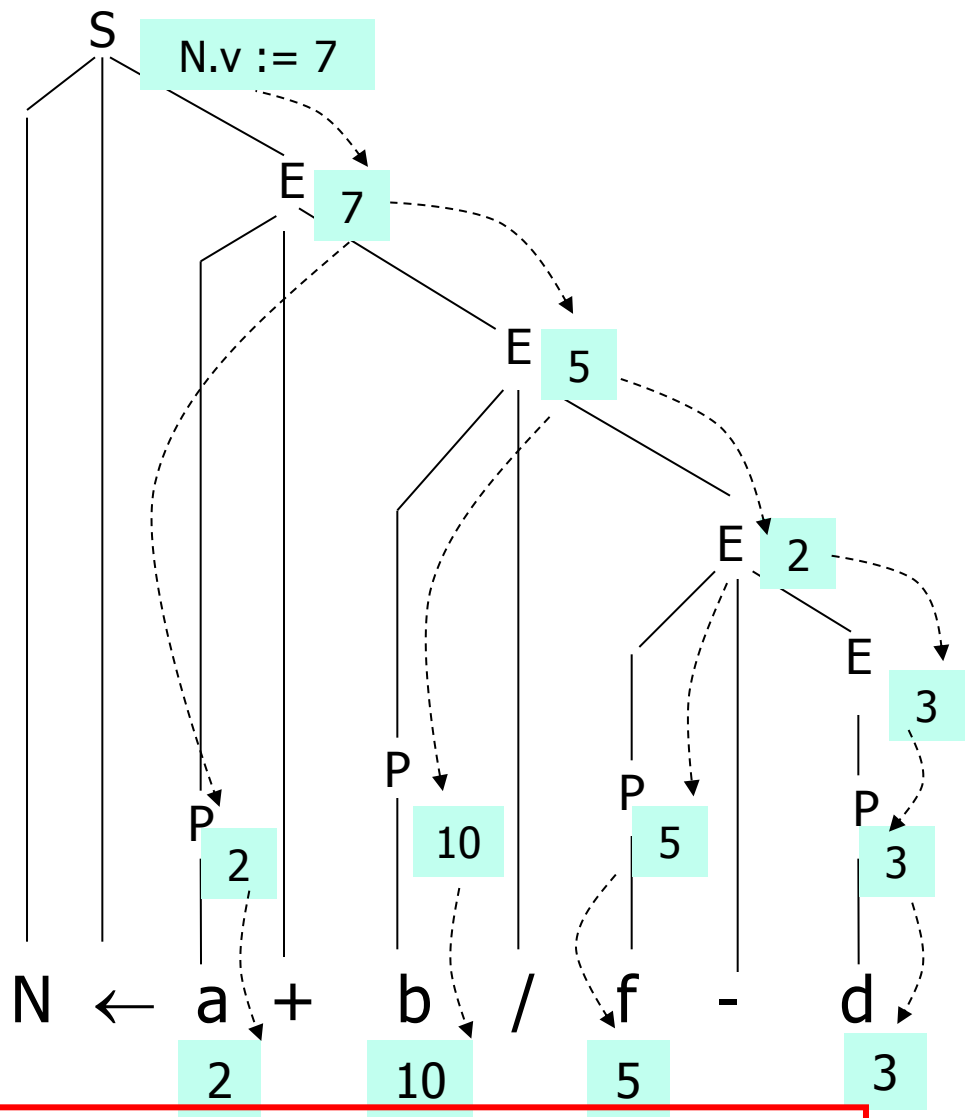


Пример 6. Язык APL

Язык APL: что такое “правильное” значение арифметического выражения?

- Язык APL (A Programming Language) имеет специфический синтаксис и своеобразную семантику, часть программистов использует его для построения эффективного кода
- Вычисление значения арифметич. выражения - справа налево без учета приоритетов. Скобки, как обычно, выделяют подвыражения
- Грамматика арифм. выражения:

$E \rightarrow P + E$
$E \rightarrow P - E$
$E \rightarrow P * E$
$E \rightarrow P / E$
$E \rightarrow P$
$P \rightarrow i$
$P \rightarrow (E)$ скобки и в Африке скобки



Семантические вычисления одновременно с построением дерева

НЕдвусмысленная грамматика языка Милан

В операторы условный и цикла введены доп служебные слова и изменена грамматика ар. выражений

Prog→begin L end
L →S | L ; S
S → i_k := E
| while B do L od
| if B then L fi
| if B then L else L fi
| write (E)
B→ E rel_k E
E→ E $addop_k$ T | T
T→ T $mulop_k$ P | P
P→ i_k | c_k | (E) | read

```
begin
   a :=5;
   x:=read;
   a :=x+a*3;
   while x>0 do a:=a*2;
x:=x-1
end
```

$addop_k$ = (k==0) ? '+' : '-'

$mulop_k$ = (k==0) ? '*' : '/'

i_k : k - номер переменной

c_k : k - номер константы

rel_k

k=0? '='
k=1? '≠'
k=2? '<'
...

Является ли эта грамматика однозначной? Далее мы рассмотрим такие подклассы КС-грамматик, которые точно однозначны, и покажем, что грамматика Милана принадлежит некоторым из этих классов

Пример формального описания синтаксиса формул линейной темпоральной логики LTL

Формулы LTL "полуформально" задаются так:

Пусть AP множество атомарных высказываний. Тогда:

1. Если $a \in AP$, то a – формула LTL
2. Если f и g – формулы LTL, то формулами LTL будут:
   $\neg f$, $f \vee g$, $f \wedge g$, Xf, Ff, Gf, fUg (избыточно, $f \vee g$ либо $f \wedge g$ можно убрать!!)

Формально: Грамматика формул LTL:

$$\phi ::= a \mid \neg \phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid X \phi \mid F \phi \mid G \phi \mid \phi \, U \, \phi$$

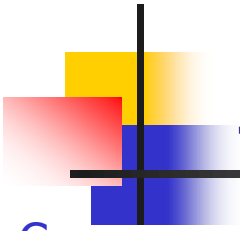
Пример синтаксически правильной формулы LTL:

$$a \vee G\ b \wedge \neg F\ a\ U\ G\ b \vee X\ d$$

Что означает эта формула?

Сколько деревьев вывода (смыслов) у этой формулы? >10

Подобные грамматики могут использоваться для задания СИНТАКСИСА формул, но для вычисления семантики их использовать НЕЛЬЗЯ!



Заключение (1)

Существо синтаксически-ориентированного подхода к трансляции: вычисление "смысла", "значения" цепочки языка на основе структуры этой цепочки, выраженной деревом вывода этой цепочки в конкретной грамматике

Семантика является дополнительной к синтаксису, она определяется тем смыслом, который мы хотим вложить в фразы языка, а этот смысл может быть различным даже для одного и той же строки языка

Строго говоря, ни множество смыслов, которые мы хотим сопоставить цепочкам языка, ни семантические атрибуты, ни семантические вычисления не являются частью языка или задающей язык грамматики

Если один и тот же язык с одним и тем же смыслом задается разными грамматиками, то семантические функции будут различными, потому что разные грамматики связывают с конкретной цепочкой разные структуры

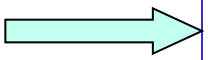
Заключение (2)

- Любой язык может быть порожден бесконечным множеством грамматик, и при построении грамматики языка следует выбрать наиболее удобную грамматику. Требования к грамматике:
  - грамматика должна порождать именно тот язык, который нам хочется
  - грамматика должна быть недвусмысленной
  - грамматика должна обеспечить для цепочек языка удобную структуру, позволяющую выполнять простые семантические вычисления
- Грамматики для порождения (описания языка) и для трансляции этого же языка могут быть разными
- Грамматика арифметических выражений, построенная нами для трансляции, обеспечивает простые и эффективные семантические вычисления. Ее нужно помнить и полностью понимать

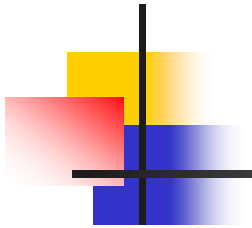
$$E \rightarrow E\ \underline{addop}\ T \mid T$$
$$T \rightarrow T\ \underline{mulop}\ F \mid F$$
$$F \rightarrow P \uparrow F \mid F$$
$$P \rightarrow i \mid (\ E\ )$$

Требования к знаниям студента по разделам курса

- Конечные автоматы
- Грамматики Хомского
- **Атрибутная семантика**
- Синтаксический анализ
- Трансляция ЯВУ



- понимать идею трансляции автоматных языков как частный случай синтаксически-ориентированной трансляции
- понимать идеи атрибутной семантики, использование синтезируемых и наследуемых атрибутов
- уметь проверять корректность атрибутной семантики
- уметь использовать атрибутную семантику в трансляции языков
- уметь строить семантические атрибуты в различных задачах трансляции арифметических выражений
- понимать организацию виртуальной стековой машины и уметь строить программу стековой машины по программе на языке Милан
- уметь строить КС-грамматику, семантические атрибуты и выполнять трансляцию арифметических выражений для языка APL
- понимать другие типы семантических вычислений: (например, операционная семантика)
- досконально понимать все идеи, заложенные в грамматике арифметических выражений



Спасибо за внимание