Семантический анализ

Семантический анализ

Семантический анализ

Число
Символ
Логическое значение
Строка
Составное значение

Семантический анализ

Семантический анализ

Семантический анализ

Семантический анализ

Семантический анализ

Размер

Количество байт на целевой архитектуре

Количество аргументо
Значения аргументо
Возвращаемо
значени

Число
имвол
огическое значение
трока
оставное значение

Компилятор должен уметь исследовать смысл (семантику) программы

Лексический и синтаксический анализ оценивают только форму программы

Время жизни

Локальное значение
Глобальное значение

Выделе-ние памяти

Неявное выделение памяти
Явное выделение памяти

Этапы семантического анализа

- Проверка соблюдения семантических соглашений входного языка
- Дополнение внутреннего представления программы операторами и действиями, неявно предусмотренными семантикой входного языка
- Проверка смысловых норм языков программирования, напрямую не связанных с входным языком

Проверка соблюдения семантических соглашений

Сопоставление входных цепочек исходной программы с требованиями семантики входного языка программирования

Примеры семантических соглашений

- ✓ Идентификаторы должны быть уникальны в пределах одного блока
- ✓ Операнды в выражениях и операциях должны иметь типы, допустимые для данных выражений и операций
- ✓ Согласование типов операндов в выражениях
- ✓ Количество и типы фактических параметров функций должны совпадать с количеством и типами формальных параметров

Дополнение внутреннего представления программы

```
double a;
int b = 2;
float c = 3.5;
                    a = b + c;
```

Дополнение внутреннего представления программы

```
double a;
int b = 2;
float c = 3.5;
                    a = b + c;

          a = (double)(b + (int)c);
```

Дополнение внутреннего представления программы

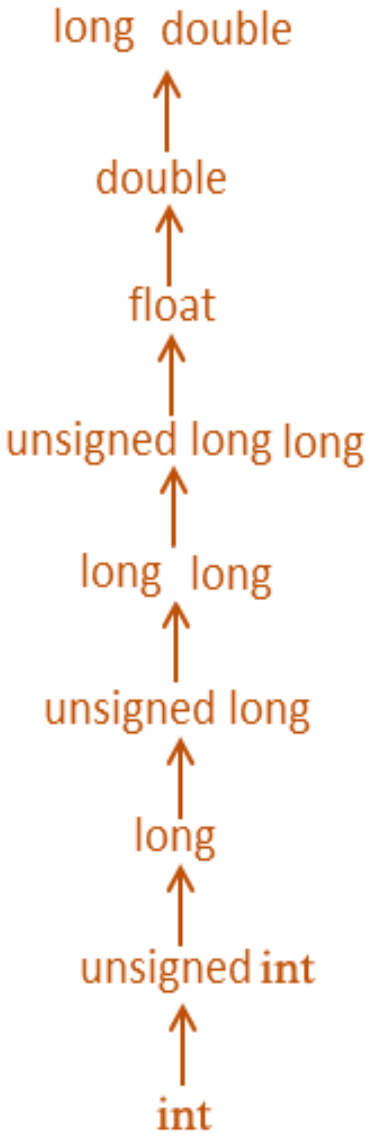
```
double a;
int b = 2;
float c = 3.5;
                    a = b + c;

          a = (double)(b + (int)c);
         a = (double)((float)b + c);
```

Дополнение внутреннего представления программы

```
double a;
int b = 2;
float c = 3.5;
```

```
a = b + c;
```

```
a = (double)(b + (int)c);
a = (double)((float)b + c);
```



Система типов

Множество типов языка программирования вместе с правилами их использования для спецификации поведения программы называется **системой типов**

Система типов

Множество типов языка программирования вместе с правилами их использования для спецификации поведения программы называется **системой типов**

Компилятор

+ обеспечивает безопасность (приведение типов)

+ улучшает выразительность (перегрузка (overloading) операторов)

+ обеспечивает эффективность (во время выполнения программы не тратится на проверку типов)

Система типов

- **Сильно типизированный язык** – язык, в котором каждому выражению может быть присвоен однозначный тип
- **Статически типизированный язык** – язык, в котором тип каждого выражения может быть определен во время компиляции
- **Динамически типизированный язык** – язык, в котором есть выражения, чей тип можно определить только во время выполнения программы

Нетипизированный язык = ассемблер

Компоненты системы типов

- Множество базовых (встроенных) типов
  - Числовые
    - целые
    - вещественные с плавающей запятой
  - Символьные
  - Булевы

Компоненты системы типов

- Множество базовых (встроенных) типов

- Правила построения новых типов на основе существующих
  - Массивы
  - Строки
  - Перечисляемый тип (enum)
  - Структуры
  - Объединения
  - Указатели

Компоненты системы типов

- Множество базовых (встроенных) типов
- Правила построения новых типов на основе существующих
- Метод определения эквивалентности или совместимости двух типов

```
struct Tree {
    struct Tree *left;
    struct Tree *right;
    int value;
}
```

$\cong ?$

```
struct STree {
    struct STree *left;
    struct STree *right;
    int value;
}
```

- Эквивалентность по имени
- Структурная эквивалентность

Компоненты системы типов

- Множество базовых (встроенных) типов
- Правила построения новых типов на основе существующих
- Метод определения эквивалентности или совместимости двух типов
- Правила определения типа для каждого выражения языка
  - Типы переменных задаются при их декларации
  - Типы констант определяются
    - их внешним видом
      2 = int 2.0 = float
    - их использованием в программе
      sin(2) int x = 2
  - Прототипы функций

```
unsigned int strlen(const char *s);
```

Возвращаемый тип

Типы формальных параметров

Компоненты системы типов

- Множество базовых (встроенных) типов
- Правила построения новых типов на основе существующих
- Метод определения эквивалентности или совместимости двух типов
- Правила определения типа для каждого выражения языка
  - Типы переменных задаются при их декларации
  - Типы констант определяются
    - их внешним видом
      2 = int 2.0 = float
    - их использованием в программе
      sin(2) int x = 2
  - Прототипы функций

```
unsigned int strlen(const char *s);
```

Сигнатура функции

```
strlen : const char * → unsigned int
```

Компоненты системы типов

- Множество базовых (встроенных) типов
- Правила построения новых типов на основе существующих
- Метод определения эквивалентности или совместимости двух типов
- Правила определения типа для каждого выражения языка
- Правила неявного приведения типов

int → unsigned int → long → unsigned long → long long → unsigned long long → float → double → long double

Атрибутные грамматики

- **Атрибут** – значение, связанное с одним или несколькими узлами синтаксического дерева разбора

- **Атрибутная грамматика** – контекстно-свободная грамматика, расширенная правилами, определяющими вычисления

- Каждое правило определяет одно значение, *атрибут*, через значения других атрибутов

Атрибутные грамматики (пример)

$$P = \left\{ \begin{array}{c} Number \ \rightarrow Sign\ List \\ Sign \ \rightarrow \ + \mid - \\ List \ \rightarrow List\ Bit \mid Bit \\ Bit \ \rightarrow 0 \mid 1 \end{array} \right\}$$

T = {+, -, 0, 1}
NT = {Number, Sign, List, Bit}
S = {Number}

-101 +11 -01 +11111001100

10

Атрибутные грамматики (пример)

$$P = \left\{ \begin{array}{c} Number \rightarrow Sign\ List \\ Sign \rightarrow + \mid - \\ List \rightarrow List\ Bit \mid Bit \\ Bit \rightarrow 0 \mid 1 \end{array} \right\}$$

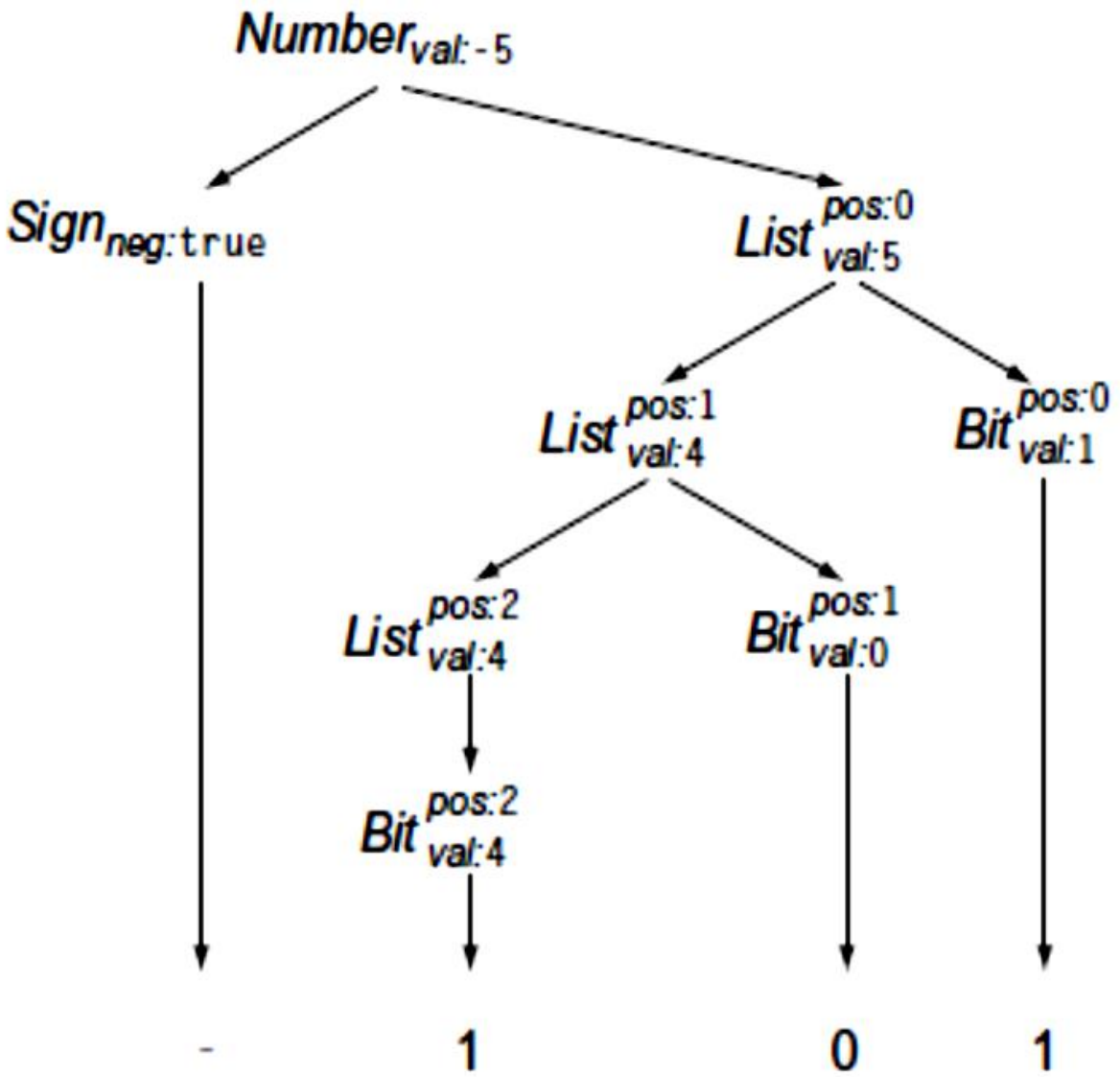
T = {+, –, 0, 1}
NT = {Number, Sign, List, Bit}
S = {Number}

Символ	Атрибуты
Number	value
Sign	negative
List	position, value
Bit	position, value

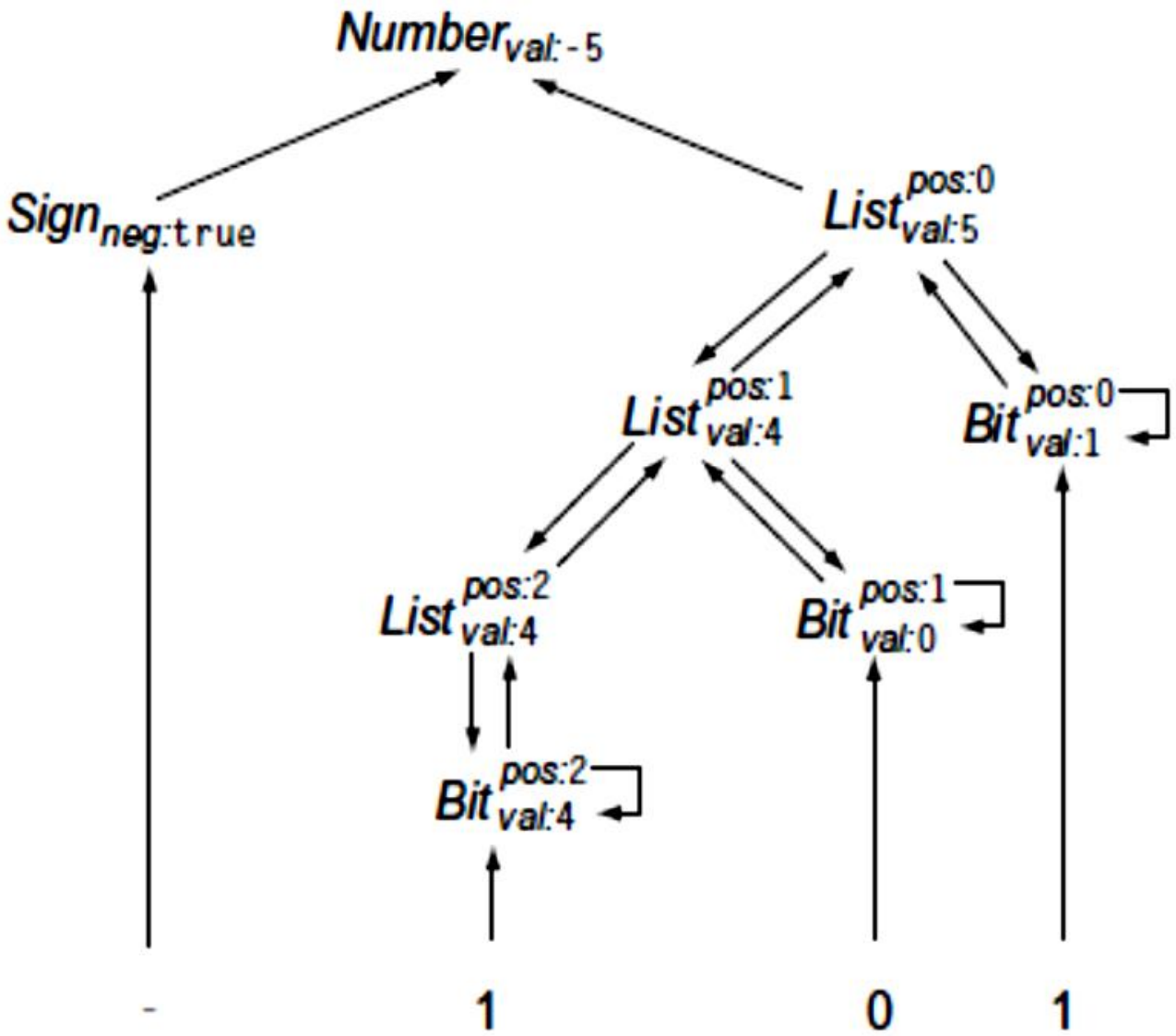
Атрибутные грамматики (пример)

Правило грамматики	Атрибутное правило
Number → Sign List	`List.position = 0` **if** `Sign.negative` **then** `Number.value = – List.value` **else** `Number.value = List.value`
Sign → +	`Sign.negative =` **false**
Sign → –	`Sign.negative =` **true**
List → Bit	`Bit.position = List.position` `List.value = Bit.value`
$List_0 \rightarrow List_1$ Bit	$List_1$`.position =` $List_0$`.position + 1` `Bit.position =` $List_0$`.position` $List_0$`.value =` $List_1$`.value + bit.value`
Bit → 0	`Bit.value = 0`
Bit → 1	`Bit.value =` $2^{\texttt{Bit.position}}$

Атрибутные грамматики (пример)



(a) Parse Tree for -101



(b) Dependence Graph for -101

Типы атрибутов

- **Синтезированный атрибут** – атрибут, полностью определенный через атрибуты узла, его потомков и константы

- **Унаследованный атрибут** – атрибут, полностью определенный через собственные атрибуты узла и атрибуты его братьев или родителя в синтаксическом дереве разбора (плюс константы)

Синтаксически управляемые схемы трансляции

- **Схема трансляции** — это запись присоединенных к правилам грамматики программных фрагментов
- Эти фрагменты выполняются при использовании правила в процессе синтаксического анализа
- Объединенный результат выполнения всех этих фрагментов в порядке, определяемом синтаксическим анализом, и есть трансляция программы
- Программные фрагменты, вставленные в тела правил, называются **семантическими действиями**
- Позиция выполняемого действия указывается фигурными скобками в теле правила

$$E \rightarrow E_1 + T \{\text{print '+'}\}$$

Постфиксные схемы трансляции

Восходящий синтаксический анализ

Каждое действие размещается в конце правила и выполняется вместе со сверткой тела правила

$$
\begin{array}{ll}
L \rightarrow E\ \mathbf{n} & \{print\,(E.val)\,;\} \\
E \rightarrow E_1 + T & \{E.val = E_1.val + T.val;\} \\
E \rightarrow T & \{E.val = T.val;\} \\
T \rightarrow T_1 * F & \{T.val = T_1.val \times F.val;\} \\
T \rightarrow F & \{T.val = F.val;\} \\
F \rightarrow (E) & \{F.val = E.val;\} \\
F \rightarrow \mathbf{digit} & \{F.val = \mathbf{digit}.lexval;\}
\end{array}
$$

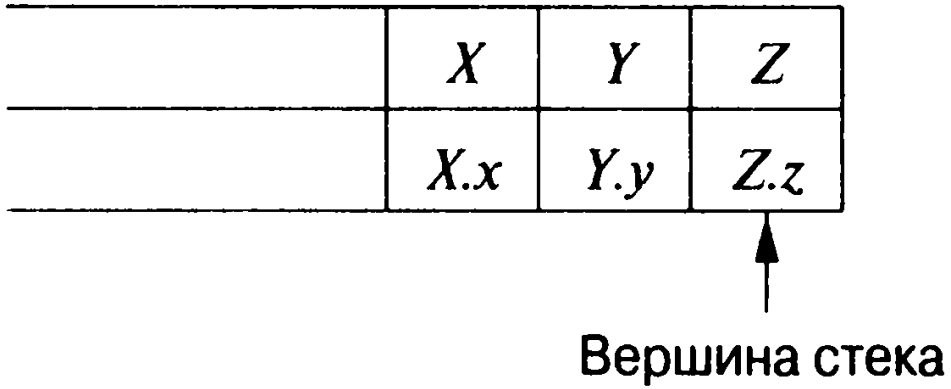
Постфиксные схемы трансляции

$$
\begin{array}{ll}
L \rightarrow E\ \mathbf{n} & \{print\,(E.val)\,;\} \\
E \rightarrow E_1 + T & \{E.val = E_1.val + T.val;\} \\
E \rightarrow T & \{E.val = T.val;\} \\
T \rightarrow T_1 * F & \{T.val = T_1.val \times F.val;\} \\
T \rightarrow F & \{T.val = F.val;\} \\
F \rightarrow (E) & \{F.val = E.val;\} \\
F \rightarrow \mathbf{digit} & \{F.val = \mathbf{digit}.lexval;\}
\end{array}
$$

```
calclist: exp EOL { printf("= %d\n", $2); }
exp: term
     | exp ADD term { $$ = $1 + $3; }
term: factor
     | term MUL factor { $$ = $1 * $3; }
term: NUMBER
     | (exp)
```

Постфиксные схемы трансляции

$$A \rightarrow XYZ$$



Состояние/Грамматический символ

Синтезируемые атрибуты

Постфиксные схемы трансляции

$$
\begin{array}{ll}
L \to E\ \mathbf{n} & \{\ \mathrm{print}(stack\,[top-1].val);\\
 & \quad top = top - 1;\ \}\\
E \to E_1 + T & \{\ stack\,[top-2].val = stack\,[top-2].val + stack\,[top].val;\\
 & \quad top = top - 2;\ \}\\
E \to T & \\
T \to T_1 * F & \{\ stack\,[top-2].val = stack\,[top-2].val \times stack\,[top].val;\\
 & \quad top = top - 2;\ \}\\
T \to F & \\
F \to (\ E\ ) & \{\ stack\,[top-2].val = stack\,[top-1].val;\\
 & \quad top = top - 2;\ \}\\
F \to \mathbf{digit} &
\end{array}
$$

Схемы трансляции с действиями внутри правил

- Действия могут находиться в любой позиции в теле правила
- Действие выполняется сразу же после того, как обработаны все символы слева от него

$$B \rightarrow X \{a\} Y$$

действие a выполняется после того, как распознан X (если X – терминал) или все терминалы, порожденные из X (если X – нетерминал)

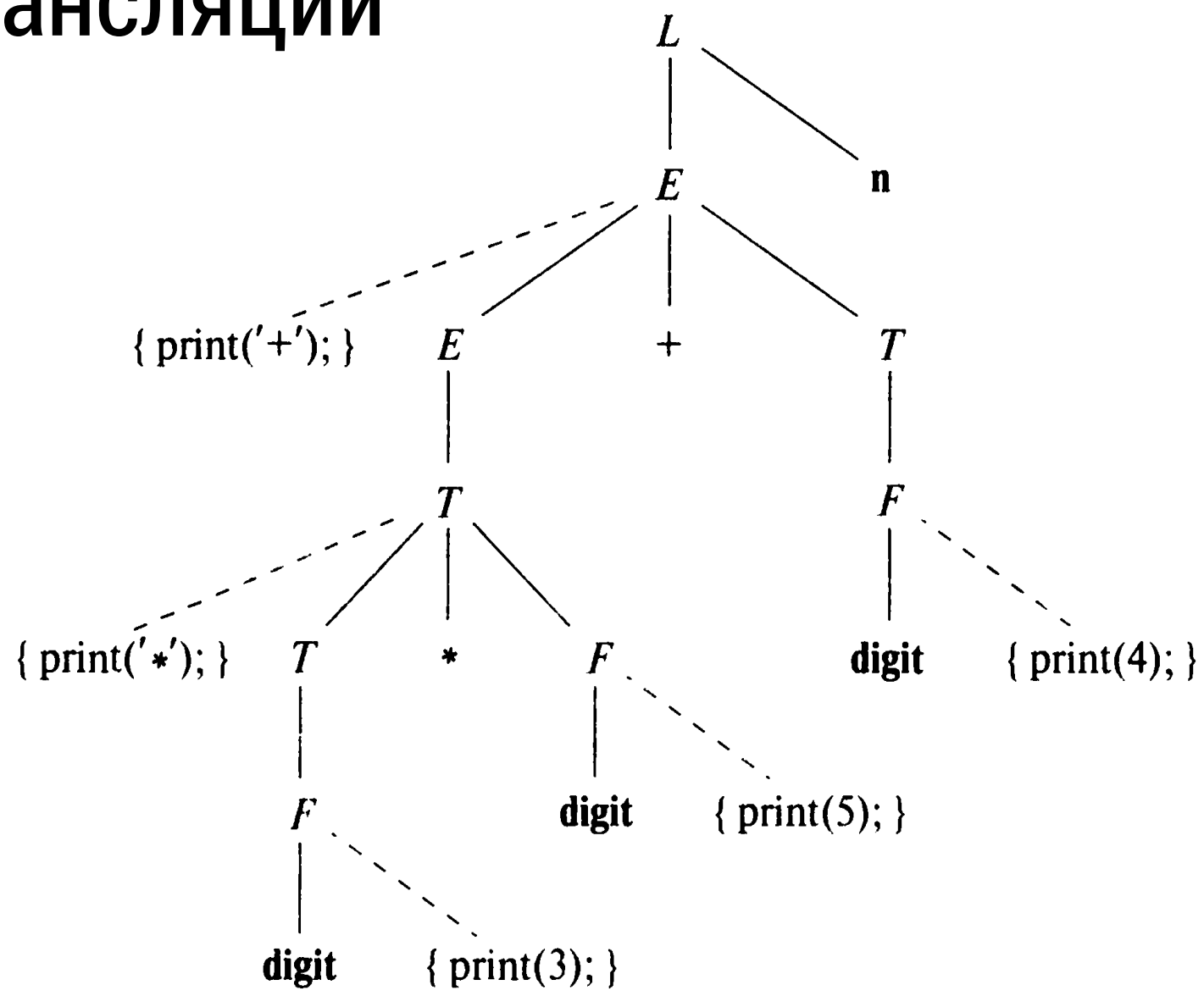
- При восходящем синтаксическом анализе действие a выполняется, как только данный грамматический символ X оказывается на вершине стека
- При нисходящем синтаксическом анализе действие a выполняется непосредственно перед тем, как выполняется раскрытие данного Y (если Y – нетерминал) или проверяется его наличие во входной строке (если Y – терминал)

Проблемы в схемах трансляции

1) $L \rightarrow E\ \mathbf{n}$
2) $K \rightarrow \{print\,('+')\,;\}\ E_1 + T$
3) $E \rightarrow T$
4) $T \rightarrow \{print\,('*')\,;\}\ T_1 * F$
5) $T \rightarrow F$
6) $F \rightarrow (E)$
7) $F \rightarrow \mathbf{digit}\ \{print\,(\mathbf{digit}.lexval)\,;\}$

$$3*5+4$$

$$+\ *\ 3\ 5\ 4$$



Пример

$$S \rightarrow \textbf{while } (C)\ S_1$$

$L1 = new();$

$L2 = new();$

$S_1.next = L1;$

$C.false = S.next;$

$C.true = L2;$

$S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code$

$$\begin{array}{lll} S \rightarrow \textbf{while} \ (& \{\ L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;\ \} \\ \quad C\ ) & \{\ S_1.next = L1;\ \} \\ \quad S_1 & \{\ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code;\ \} \end{array}$$

Пример

$$S \rightarrow \textbf{while } (C)\ S_1$$

```
string S(label next) {
    string Scode, Ccode; /* Локальные переменные с фрагментами кода */
    label L1, L2; /* Локальные метки */
    if ( Текущий входной символ == токен while ) {
        Перемещение по входному потоку;
        Проверить наличие '(' во входной строке и перейти к новой
            позиции;
        L1 = new();
        L2 = new();
        Ccode = C(next, L2);
        Проверить наличие ')' во входной строке и перейти к новой
            позиции;
        Scode = S(L1);
        return("label"||L1||Ccode||"label"||L2||Scode);
    }
    else /* Инструкции других видов */
}
```