Курс «Компиляторные технологии»

Лекция 4

Синтаксически управляемая трансляция

Курносов Михаил Георгиевич

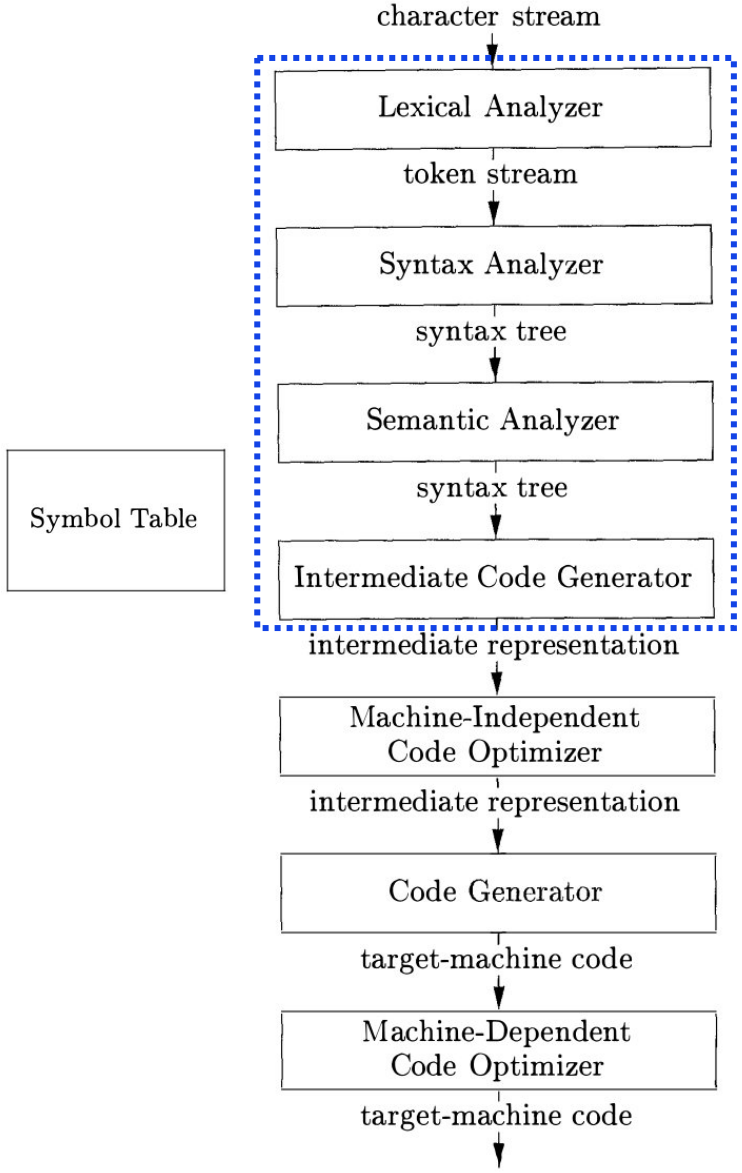
www.mkurnosov.net

Сибирский государственный университет телекоммуникаций и информатики
Весенний семестр

Структура начальной стадии компилятора (frontend)

- **Фаза анализа** (frontend, начальная стадия) — разбивает программу на последовательность минимально значимых единиц языка, накладывает на них грамматическую структуру языка, обнаруживает синтаксические и семантические ошибки, формирует таблицу символов, генерирует промежуточное представление программы

- **Стадии (фазы) начальной стадии компилятора:**
  - **лексический** анализ —> выделение токенов + таблица символов
  - **синтаксический** анализ —> синтаксическое дерево (tree-based IR)
  - **семантический** анализ —> синтаксическое дерево
  - **генерация** (синтез) промежуточного представления —> линеаризованное представление синтаксического дерева (linear IR)



[*] Aho A., Lam M., Sethi R., Ullman J. **Compilers: principles, techniques, and tools**. — 2nd ed., 2007

Входной язык программирования

- **Конструкции языка программирования yacl** (yet another C-like language, DragonBook, Appendix A)**:**
  - циклы, ветвления
  - арифметические выражения: инфиксная форма, бинарные операции, унарные (приоритеты, ассоциативность)
  - переменные: локальные, (автоматические), скаляры и массивы, статическая типизация
  - область видимости: структурные блоки

```
{
    int i; int j; float[100] a; float v; float x;

    while (true) {
        do i = i + 1; while (a[i] < v);
        do j = j - 1; while (a[j] > v);

        if (i >= j)
            break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
}
```

Структура начальной стадии компилятора (frontend)

1. Разбор основных понятий на примере трансляции инфиксных выражений в постфиксные
2. Трансляция программы в промежуточное представление — трехадресный код

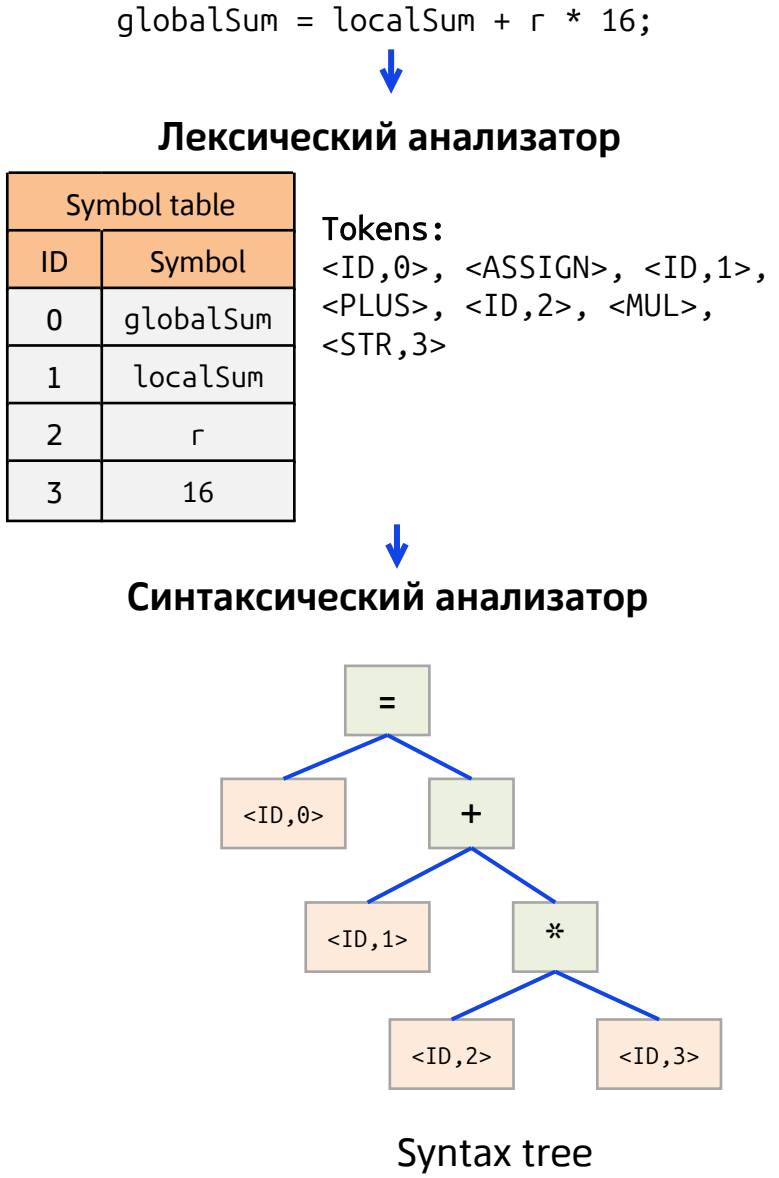

```
{
    int i; int j; float[100] a; float v; float x;

    while (true) {
        do i = i + 1; while (a[i] < v);
        do j = j - 1; while (a[j] > v);

        if (i >= j)
            break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
}
```

```
1 : i = i + 1
2 : t1 = a[i]
3 : if t1 < v goto 1
4 : j = j - 1
5 : t2 = a[j]
6 : if t2 > v goto 4
7 : ifFalse i >= j goto 9
8 : goto 14
9 : x = a[i]
10: t3 = a[j]
11: a[i] = t3
12: a[j] = x
13: goto 1
14:
```

- **Трехадресный код** (three-address code, TAC, 3AC) линеаризованное представление синтаксического дерева
- Бинарные операции: `t1 := t2 op t3`
- Переходы: `ifFalse, goto`

Синтаксический анализ

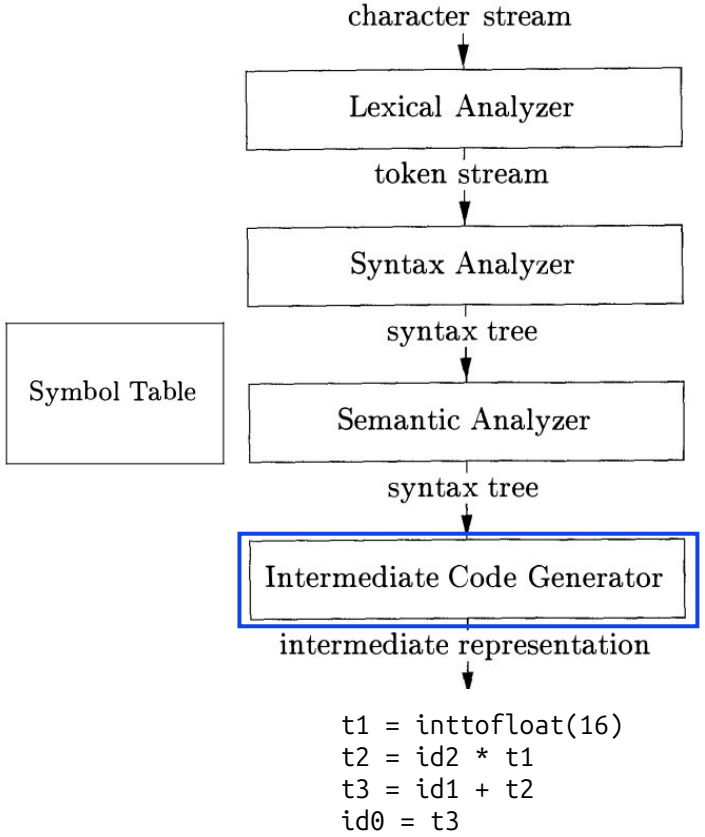
- **Синтаксический анализ** (разбор, parsing) — процесс проверки соответствия входного потока токенов (текста программы) синтаксису входного языка и построения синтаксического дерева
- Синтаксический анализатор строится на основе синтаксиса входного языка, который описывается *формальной грамматикой языка*
- **Синтаксическое дерево** (syntax tree) — каждый внутренний узел дерева представляет операцию языка, дочерние узлы — аргументы операции
- Синтаксическое дерево — это форма промежуточного представления (tree-based IR)

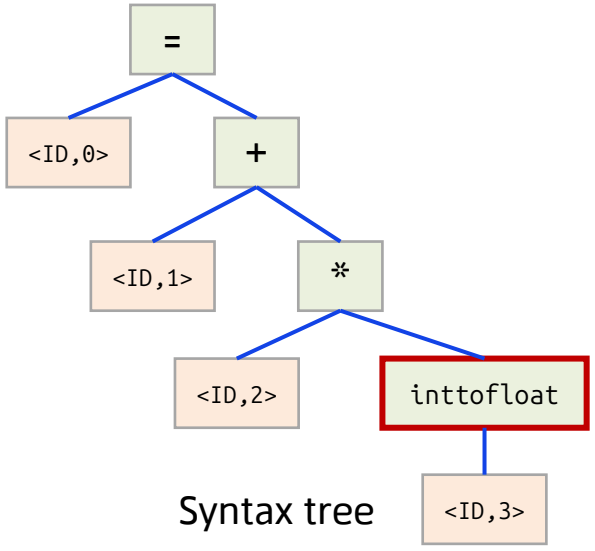


Syntax tree

Генерация кода в промежуточное представление

- **Трансформация** из синтаксического дерева (tree-based IR) в линеаризованное промежуточное представление (linear IR)
- **Промежуточное представление** (intermediate representation, IR) — архитектура набора команд (ISA) абстрактной вычислительной машины, в который легко транслировать синтаксическое дерево, над которым легко выполнять оптимизации и трансформации и генерировать машинный код для целевой архитектуры
- Трехадресный код — в каждой команде 3 операнда
- Стековые и регистровые машины





Syntax tree

Трехадресный код (IR)

```
t1 = inttofloat(16)
t2 = id2 * t1
t3 = id1 + t2
id0 = t3
```

Язык записи арифметических выражений

- **Пример**: 12 + 1 - 4 + 64
- **Бинарные операции**: +, —
- **Операнды**: целые числа в десятичной систем исчисления
- **Форма записи**: инфиксная «a + b», альтернативы: префиксная «+ a b», постфиксная «a b —»

- **Задача** — описать синтаксис указанного языка записи арифметических выражений

- Алфавит языка?
- Тип языка по иерархии Хомского — какой формализм описания минимально необходим?
- Регулярный язык (тип 3)?
- Контекстно-свободный (тип 2)?
- Контекстно-зависимый (тип 1)?
- Рекурсивно перечислимый язык (распознаваемый по Тьюрингу, тип 0)?

Язык записи арифметических выражений

- **Пример**: 12 + 1 - 4 + 64
- **Бинарные операции**: +, —
- **Операнды**: целые числа в десятичной систем исчисления
- **Форма записи**: инфиксная «a + b», альтернативы: префиксная «+ a b», постфиксная «a b —»

- **Задача** — описать синтаксис указанного языка записи арифметических выражений

- Алфавит языка? **A = {0, 1, …, 9, +, —, \32, \9}**
- Тип языка по иерархии Хомского — какой формализм описания минимально необходим?
- Регулярный язык (тип 3)?
- Контекстно-свободный (тип 2)?
- Контекстно-зависимый (тип 1)?
- Рекурсивно перечислимый язык (распознаваемый по Тьюрингу, тип 0)?

Иерархия Хомского

- **Иерархия Хомского** — классификация формальных языков и формальных грамматик на 4 типа по их условной сложности [Ноам Хомский, 1956, https://chomsky.info/wp-content/uploads/195609-.pdf]
- Для отнесения грамматики к определенному типу необходимо соответствие всех её продукций некоторым схемам

Тип	Грамматика	Вид продукций	Применение
Тип 0	**Неограниченные грамматики** (рекурсивно перечислимые, recursively enumerable)	$a \to b$ ▪ a $\in V^+$ — непустая цепочка, содержащая хотя бы один нетерминал ▪ $b \in V^*$ — любая цепочка символов из $V = V_T \cup V_N$ (эквивалентны машинам Тьюринга)	Практического применения в силу своей сложности (общности) такие грамматики не имеют
Тип 1	**Контекстно-зависимые** (context-sensitive)	$aAb \to acb$ ▪ a, $b \in V^*$ — любая цепочка символов из V ▪ $c \in V^+$ — непустая цепочка из V ▪ $A \in V_N$ (эквивалентны линейно ограниченным автоматам)	Анализ текстов на естественных языках, при построении компиляторов практически не используются
Тип 2	**Контекстно-свободные** (context-free)	$A \to b$ ▪ $A \in V_N$ ▪ $b \in V^*$ — любая цепочка символов из V (эквивалентны магазинным автоматам)	Описание синтаксиса компьютерных языков
Тип 3	**Регулярные** (regular)	$A \to Bc$ или $A \to c$ — леволинейные грамматики $A \to cB$ или $A \to c$ — праволинейные грамматики ▪ $A, B \in V_N$ ▪ $c \in V_T^*$ (эквивалентны конечным автоматам)	Описание простейших конструкций: идентификаторов, строк, констант, языков ассемблера, командных процессоров

Язык записи арифметических выражений

- **Пример**: 12 + 1 - 4 + 64
- **Бинарные операции**: +, −
- **Операнды**: целые числа в десятичной системе исчисления
- **Форма записи**: инфиксная «a + b» (префиксная «+ a b», постфиксная «a b −»)

- **Возможно ли описать синтаксис языка записи арифметических выражений регулярными выражениями?**
- Если да, то язык относится к регулярным

Описание регулярными выражениями синтаксиса языка записи арифметических выражений

- **Пример**: 12+ 1 - 4 + 64
- **Бинарные операции**: +, —
- **Операнды**: целые числа в десятичной системе исчисления
- **Форма записи**: инфиксная «a + b»

```
ws          [ \t\f\r\v]*
num         [1-9][0-9]*

%%

{num}({ws}[+-]{ws}{num})+      { return TOKEN_EXPR; }
{num}                          { return TOKEN_EXPR; }

{ws}          { /* skip spaces */ }
\n            { lineno++; }
.             { Error("unrecognized character"); }

%%

void Lexer::Error(const char* msg) const {
    std::cerr << "Error (" << lineno + 1 << "): " << msg << ": '" << YYText() << "'\n";
    std::exit(YY_EXIT_FAILURE);
}
```

Распознавание арифметических выражений на Flex

Заданный язык записи арифметических выражений (+, —) является регулярным (тип 3)

Язык записи вызова функции

- **Примеры**: terminate(), exit(error), sum(a, b), max(a, b, c)
- **Количество аргументов** (фактических параметров): >= 0

- Грамматика языка (продукции):

```
call      -> id ( optparams )
optparams -> params | eps
params    -> params , param | param
```

```
ws              [ \t\f\r\v]*
ident           [_a-zA-Z][_a-zA-Z0-9]*
param           {ident}
params          {param}{ws}({ws}","{ws}{param})*
optparams       {ws}({params})?{ws}

%%
{ident}{ws}"("{optparams}")"    { return TOKEN_FUN_CALL; }

{ws}          { /* skip spaces */ }
\n            { lineno++; }
.             { Error("unrecognized character"); }
%%
void Lexer::Error(const char* msg) const {
    std::cerr << "Lexer error (" << lineno + 1 << "): " << msg << ":'" << YYText() << "'\n";
    std::exit(YY_EXIT_FAILURE);
}
```

Распознавание языка записи вызова функции на Flex

Заданный язык записи вызова функции является регулярным (тип 3)

Описание грамматикой (продукциями) синтаксиса языка записи арифметических выражений

- Грамматика $G = (V_T, V_N, P, S)$
- V_T — алфавит, символы которого называют терминальными символами (терминалами, terminal);
- V_N — алфавит с нетерминальными символами (нетерминалами, nonterminal);
- P — множество продукций (правил), каждый элемент которого состоит из пары (a, b), где a — левая часть продукции, b — правая часть продукции, а продукция записывается: $a \to b$;
- S — начальный символ грамматики (start symbol)

$$V = V_T \cup V_N, \quad V_T \cap V_N = \varnothing$$

- **Грамматика языка записи арифметических выражения $G = (V_T, V_N, P, list)$**
- P = {

```
  list  –> list + digit
  list  –> list - digit
  list  –> digit
  digit –> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

  }

```
# Альтернативная запись
list –> list + digit | list - digit | digit
```

- V_T = {0, 1, ..., 9, +, –}
- V_N = {list, digit}

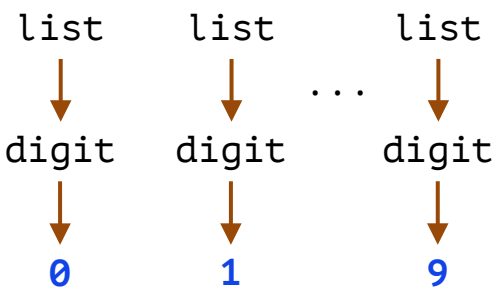
Порождение грамматикой строк (выведение)

- **Грамматика** языка списка цифр, разделенных знаками "плюс" и "минус"
- **Примеры**: «3», «9+1», «2-4+3», «9-5+2», «1+2+3+4+5+6»

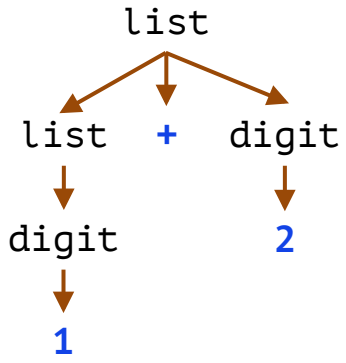
```
list  -> list + digit | list - digit | digit
digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- Грамматика *выводит*, или *порождает* (derive), все возможные строки, начиная со стартового символа и неоднократно замещая нетерминалы телами продукций этих нетерминалов
- Бесконечное множество строк токенов, порождаемые из стартового символа, образуют язык, определяемый грамматикой

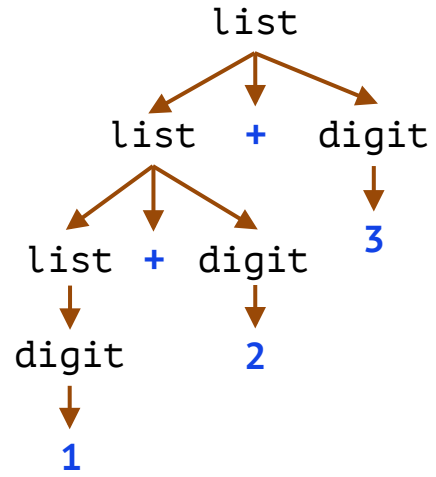
Порождения строк из одной цифры
(list —> digit —> 0)



Порождение строки «1+2»
(list —> list + digit —> digit + digit -> 1+2)



Порождение строки «1+2+3»



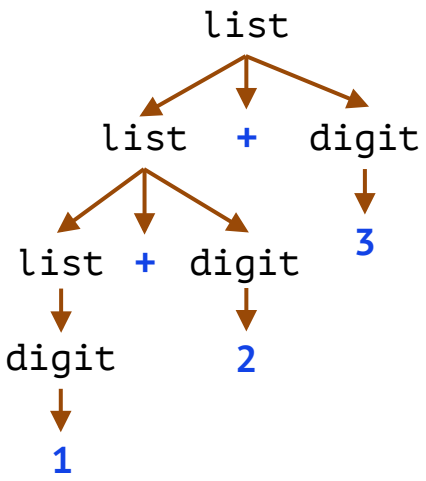
Синтаксический анализ

- **Грамматика** языка списка цифр, разделенных знаками "плюс" и "минус"

```
list  –> list + digit | list - digit | digit
digit –> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

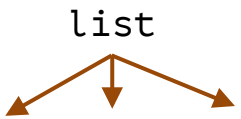
- **Синтаксический анализ** (разбор, parsing) — установление для строки терминалов (программы) способа её вывода из стартового символа грамматики (поиск дерева разбора)
- Если строка не может быть выведена из стартового символа, синтаксический анализатор сообщает о *синтаксической ошибке* в строке (syntax error)

Синтаксический анализ строки «1+2+3»



Синтаксический анализ строки «1+»

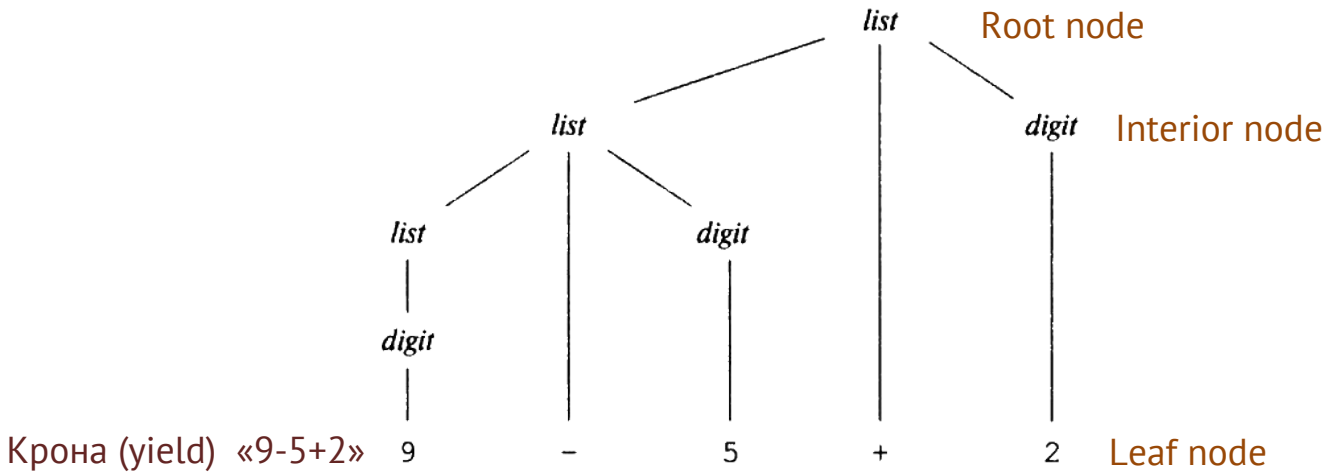
Синтаксическая ошибка — порождения не существует



Деревья разбора

- **Дерево разбора** (parse tree) — древовидное представление порождения строки языка из стартового символа грамматики
- Структура дерева разбора:
  1. Корень дерева (root) — стартовый символ грамматики
  2. Листовой узел (leaf node) — терминал или пустая строка ε
  3. Внутренний узел (interior, internal) — нетерминал
  4. Если внутренний узел *A* имеет дочерние узлы X_1, X_2, ..., X_N, то должна существовать продукция $A \rightarrow X_1X_2...X_N$, где X_i — терминальный или нетерминальный символ
- Листья дерева разбора образуют *крону* (yield) — строку, выведенную (derived), или порожденную (generated), из стартового символа в корне

Дерево разбора для строки «9-5+2»



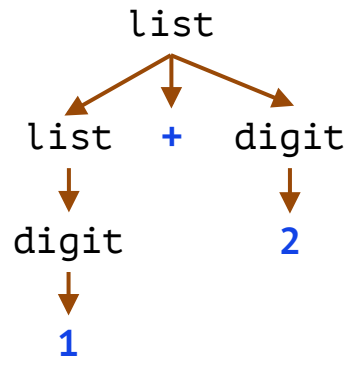
Порождение грамматикой строк (выведение)

- **Грамматика** языка списка цифр, разделенных знаками "плюс" и "минус"
- Порядок нетерминалов в правой части продукций определяет направление роста дерева разбора
- Обе грамматики порождают один и тот же язык

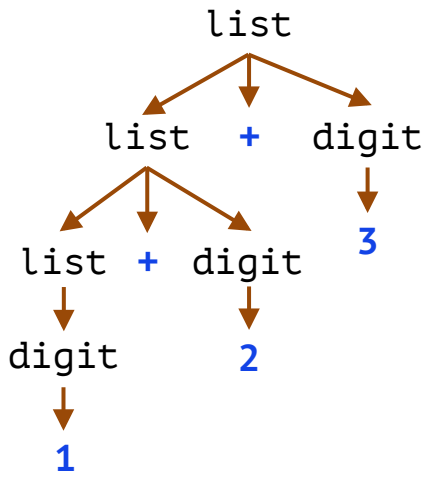
Грамматика 1:

```
list  -> list + digit | list - digit | digit
digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Порождение «1+2»



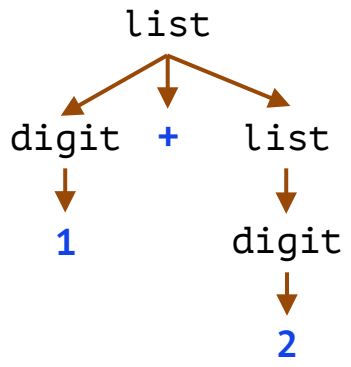
Порождение «1+2+3»



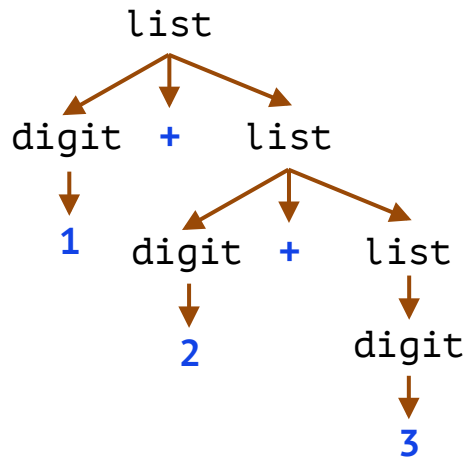
Грамматика 2:

```
list  -> digit + list | digit - list| digit
digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Порождение «1+2»



Порождение «1+2+3»



Неоднозначные грамматики

- **Неоднозначная грамматика** (ambiguous grammar) — грамматика, в которой существует более одного дерева разбора для строки терминалов
- Показать неоднозначность грамматики — найти строку терминалов, которая имеет более одного дерева разбора
- Следует разрабатывать однозначные (непротиворечивые) грамматики либо неоднозначные грамматики с дополнительными правилами для разрешения неоднозначностей в процессе синтаксического анализа

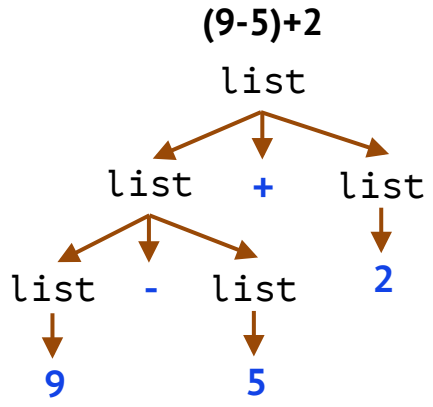
Грамматика 1:

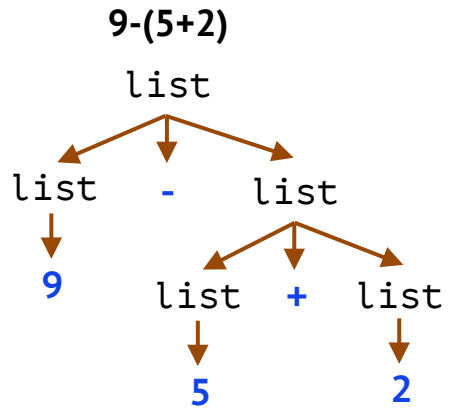
```
list  –> list + digit | list - digit | digit
digit –> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Грамматика 2 (включили digit в list):

```
list –> list + list | list - list | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- Выражение 9-5+2 в Грамматике 2 имеет больше одного дерева разбора
- Эти два дерева разбора соответствуют двум вариантам расстановки скобок в выражении:
(9-5)+2 и 9-(5+2)





Ассоциативность операторов

- По соглашению 9 + 5 + 2 эквивалентно (9 + 5) + 2, а 9 - 5 - 2 эквивалентно (9 - 5) - 2

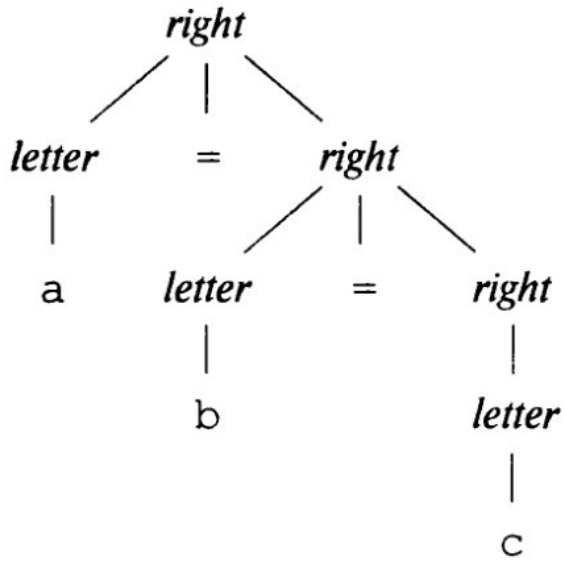
- Если операнд имеет слева и справа от себя операторы, то к какому из них он относится?
- Пример:
  - 9 - **5** + 2
  - К какому оператору относится операнд 5 («-» или «+»)?

- **Ассоциативность** (associativity) — свойство операций, устанавливающее последовательность их выполнения при отсутствии явных указаний на очерёдность при равном приоритете
- **Левая ассоциативность** — вычисление выражения происходит слева направо
  - поразрядный сдвиг (Python): `x << y << z == (x << y) << z`
- **Правая ассоциативность** — вычисление выражения происходит справа налево
  - возведение в степень (Python): `x ** y ** z == x ** (y ** z)`
- Арифметические операторы сложение, вычитание, умножение и деление — левоассоциативны
- Оператор присваивания в C/C++ правоассоциативен: `a=b=c == a=(b=c)`

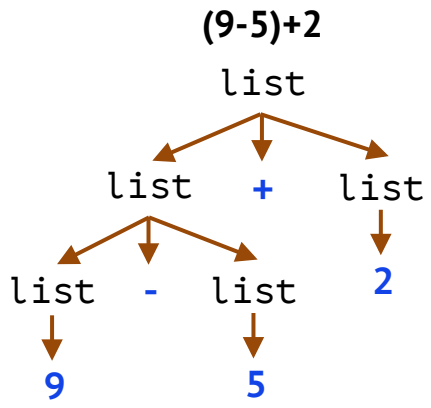
Ассоциативность операторов

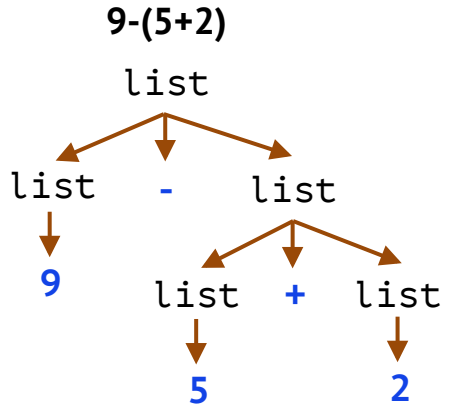
- Структура продукций грамматики позволяет задать ассоциативность операторов
- Строки типа «a=b=c» с правоассоциативным оператором «=» порождаются грамматикой:

```
right  –> letter = right | letter
letter –> a | b | ... | z
```



Дерево разбора для
правоассоциативной грамматики
(растет вниз вправо)





Приоритет операторов (precedence of operators)

- Если в языке имеется более одного типа операторов, необходимы правила, определяющие их относительный приоритет

- Как интерпретировать выражение «9 + 5 * 2» ?
  - (9 + 5) * 2
  - 9 + (5 * 2)

- Ассоциативные правила (структура продукций) позволяют установить порядок операций только для операторов одинакового приоритета
- Оператор * имеет более *высокий приоритет* (higher precedence), чем +, если * получает свои операнды раньше +
- Грамматика арифметических выражений может быть построена на основе таблицы ассоциативность и приоритет операторов

Приоритет	Ассоциативность	Операторы
1 (низший)	Левоассоциативные	+ –
2 (высший)	Левоассоциативные	* /

[*] https://en.cppreference.com/w/cpp/language/operator_precedence

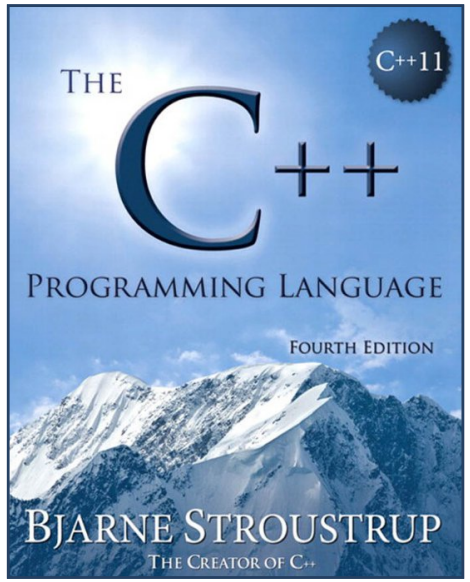
Грамматика арифметических выражений

- Алфавит {0-9, +, —, *, /, (,)}
- Нетерминал expr — для операторов + и —
- Нетерминал term— для операторов * и /
- Нетерминал factor — для генерации базовых составляющих в выражениях (цифр и выражений в скобках)

```
expr   –> expr + term | expr – term | term
term   –> term * factor | term / factor | factor
factor –> digit | ( expr )
```

- Выражение (expr) — список элементов (term), разделенных знаками + и -
- Каждый элемент списка представляет собой список множителей (factor), разделенных знаками * и /
- Любое выражение в скобках является множителем => с помощью скобок можно создать выражение с любым уровнем вложенности (дерево разбора произвольной высоты)

Грамматика арифметических выражений



10.2 A Desk Calculator

10.2.1 The Parser

Here is a grammar for the language accepted by the calculator:

```
program:
        end                             // end is end-of-input
        expr_list end

expr_list:
        expression print                // print is newline or semicolon
        expression print expr_list

expression:
        expression + term
        expression – term
        term

term:
        term / primary
        term * primary
        primary

primary:
        number                          // number is a floating-point literal
        name                            // name is an identifier
        name = expression
        – primary
        ( expression )
```

Описание грамматики

```
double expr(bool get)           // add and subtract
{
        double left = term(get);

        for (;;) {                              // "forever"
                switch (ts.current().kind) {
                case Kind::plus:
                        left += term(true);
                        break;
                case Kind::minus:
                        left –= term(true);
                        break;
                default:
                        return left;
                }
        }
}
```

Реализация синтаксического анализатора методом рекурсивного спуска (recursive descent parser)

Грамматика для подмножества инструкций Java

stmt	→	**id** = *expression* ;
	|	**if** (*expression*) *stmt*
	|	**if** (*expression*) *stmt* **else** *stmt*
	|	**while** (*expression*) *stmt*
	|	**do** *stmt* **while** (*expression*) ;
	|	{ *stmts* }
stmts	→	*stmts stmt*
	|	ϵ