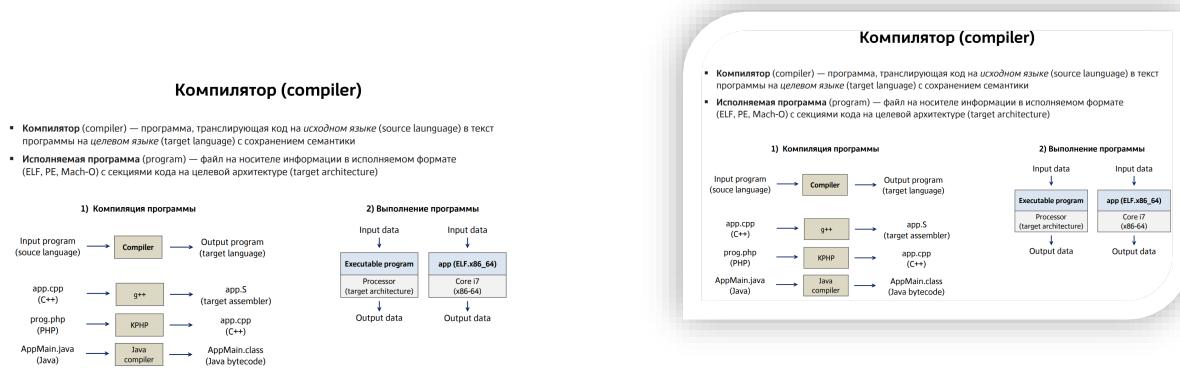


# FULL VERSION

## Правила чтения:

- 1) Не доверять написанному**
- 2) Проверять все самому**
- 3) Правило получения минимальных знаний:**



Это обязательно ^

а это нет ^

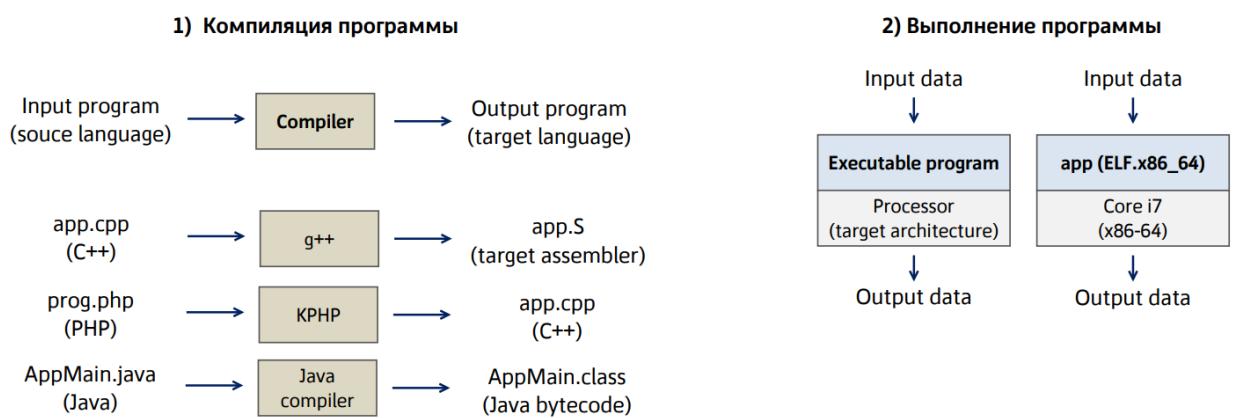
## Оглавление

<b>1. Компилятор. Основные фазы. Статические и динамические компиляторы.</b>	
Промежуточные представления.....	3
<b>2. Понятие формального языка. Синтаксис языка. Семантика. Регулярные выражения для описания языков.....</b>	8
<b>3. Формальные грамматики. Эквивалентные грамматики. Иерархия Хомского. Форма Бэкуса-Наура.....</b>	10
<b>4. Синтаксически управляемая трансляция. Деревья разбора. Семантические правила.</b>	
.....	15
<b>5. Нисходящий анализ. Анализ методом рекурсивного спуска. Левая рекурсия в продукциях. Абстрактное синтаксическое дерево.....</b>	17
<b>6. Таблица символов. Область видимости. Цепочки таблиц символов. Лексический анализ. Регулярные выражения. Архитектура лексического анализатора на основе диаграммы переходов.....</b>	22
<b>7. Переход от регулярных выражений к конечным автоматам. Построение НКА. Конвертация НКА в ДКА. Моделирование НКА.</b>	30
<b>8. Синтаксический анализ. Методы разбора. Устранение неоднозначности грамматики. Левая факторизация.....</b>	34
<b>9. Нисходящий синтаксический анализ. LL(1)-грамматики. Предиктивный синтаксический анализ, управляемый таблицей.</b>	37
<b>10. Синтаксический анализ. Восходящий синтаксический анализ. Свертки. Обрезка основ. Конфликты в процессе ПС-анализа. Простой LR-анализатор. Алгоритм LR-анализа.</b>	43
<b>11. Генерация промежуточного кода. Трехадресный код. SSA. Трансляция выражений в трёхадресный код.</b>	51
<b>12. Генерация промежуточного кода. Трехадресный код. Трансляция обращений к массиву.</b>	55
<b>13. Генерация промежуточного кода. Трехадресный код. Вычисления булевых выражений по сокращенной схеме.</b>	59
<b>14. Генерация промежуточного кода. Трехадресный код. Трансляция if-then-else.</b>	62
<b>15. Генерация промежуточного кода. Трехадресный код. Трансляция цикла while</b>	65

# 1. Компилятор. Основные фазы. Статические и динамические компиляторы. Промежуточные представления.

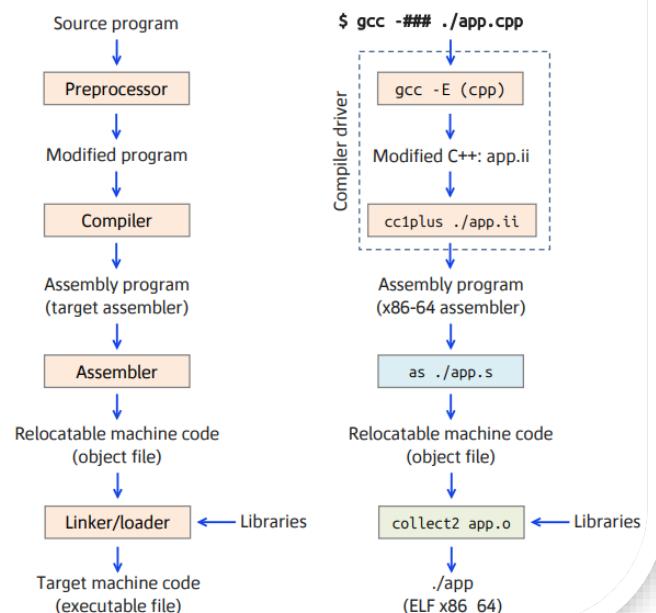
## Компилятор (compiler)

- **Компилятор** (compiler) — программа, транслирующая код на *исходном языке* (source language) в текст программы на *целевом языке* (target language) с сохранением семантики
- **Исполняемая программа** (program) — файл на носителе информации в исполняемом формате (ELF, PE, Mach-O) с секциями кода на целевой архитектуре (target architecture)



## Процесс компиляции (повторение)

- **Препроцессор** (preprocessor) — обрабатывает директивы (#include, #define, #ifdef)
- **Компилятор** (compiler) — программа, транслирующая код на *исходном языке* (source language) в текст программы на *целевом языке* (target language) с сохранением семантики
- **Ассемблер** (assembler) — транслятор с ассемблера в машинный код
- **Компоновщик** (linker) — программа объединяющая объектные файлы в исполняемый файл
- **Исполняемая программа** (program) — файл на носителе информации в исполняемом формате (ELF, PE, Mach-O) с секциями кода для целевой архитектуры (target architecture)
- **Загрузчик** (loader) — загружает секции исполняемого файла в память, загружает требуемые библиотеки динамической компоновки, передает управление на точку старта



## Статические и динамические компиляторы

- **AOT-компилятор** (ahead-of-time) — выполняет полную трансляцию программы до её выполнения (статическая компиляция)
- **JIT-компилятор** (just-in-time) — выполняет динамическую трансляцию промежуточного кода (байт-кода) в исполняемый код целевой архитектуры
  - Трудоемкие операции выполнены статическим транслятором: синтаксический анализ, оптимизация кода
  - Плюсы: позволяет задействовать аппаратные возможности (векторизация, счетчики производительности, оптимизации доступа к кеш-памяти), встраивание кода (inlining), оптимизация для заданных входных данных, байт код обеспечивает переносимость между платформами
  - Минусы: затраты на динамическую компиляцию (warm-up time, startup delay)
- **Гибридный компилятор** — выполняет статическую трансляцию в промежуточный код (bytecode, p-code) и динамическую JIT-компиляцию отдельных частей программы из промежуточного кода в код целевой архитектуры
  - Java Bytecode (Java VM) — стековая виртуальная машина
  - Google Dalvik — регистровая Java VM
  - Microsoft .NET — стековая VM (CIL, MSIL)
  - JavaScript V8 bytecode — регистровая VM
  - CPython bytecode — стековая VM
- **Общий процесс компиляции включает фазы (phases):**
  - лексический анализ
  - синтаксический анализ
  - семантический анализ
  - генерация (синтез) промежуточного представления
  - машинно-независимая оптимизация промежуточного представления
  - генерация машинного кода
  - машинно-зависимые оптимизации кода

## Лексический анализ

- Лексический анализатор (lexical analyzer, lexer, scanner) — разбивает входную программу на последовательность лексем (lexeme), минимально значимых единиц входного языка
- Тип допустимых лексем определяется описанием языка
- Игнорирует пробельные символы, комментарии, отслеживает номер текущей строки для корректного информирования о положении возможных ошибок

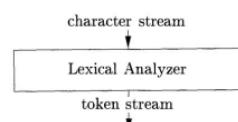
```
// Увеличить сумму
globalSum = localSum + г * 16;
```

Лексемы: «globalSum», «=», «localSum», «+», «г», «\*», «16», «;»

- Для каждой найденной лексемы анализатор формирует токен (token) — пара <имя-токена, значение-атрибута>, имя-токена — тип/класс лексемы, значение-атрибута — непосредственно лексема или ссылка на запись в таблице символов

Tokens: <ID,0>, <ASSIGN>, <ID,1>, <PLUS>, <ID,2>, <MUL>, <STR,3>, <SEMICOLON>

Token-names: ID, ASSIGN, PLUS, MUL, STR, SEMICOLON



<ID,0>, <ASSIGN>, <ID,1>, <PLUS>, <ID,2>, <MUL>, <STR,3>, <SEMICOLON>

Symbol table	
ID	Symbol
0	globalSum
1	localSum
2	г
3	16

## 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

globalSum = localSum + г \* 16;

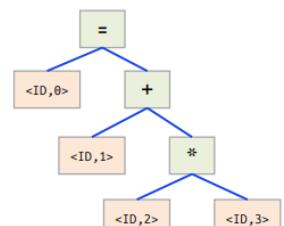
### Лексический анализатор

Symbol table	
ID	Symbol
0	globalSum
1	localSum
2	г
3	16

### Tokens:

<ID,0>, <ASSIGN>, <ID,1>, <PLUS>, <ID,2>, <MUL>, <STR,3>

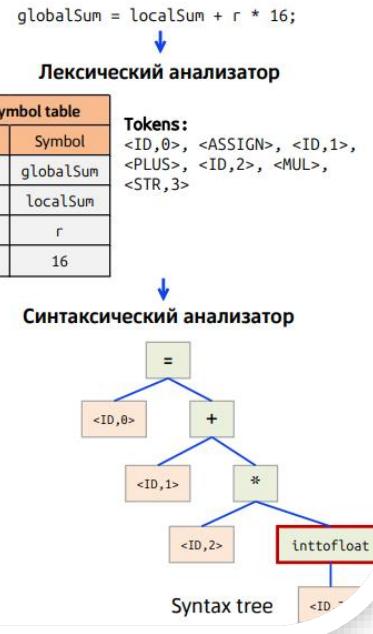
### Синтаксический анализатор



Syntax tree

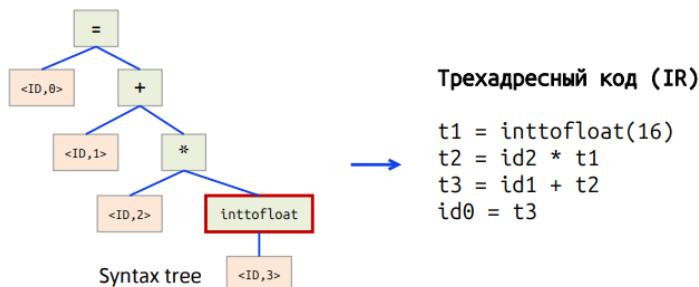
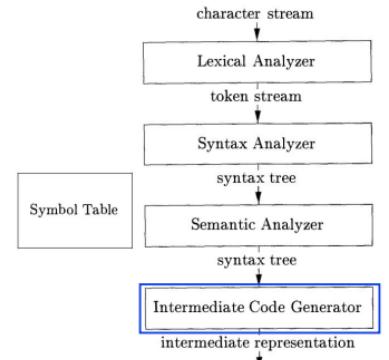
## Семантический анализ

- Семантический анализатор проверяет исходную программу (синтаксическое дерево) на семантическую согласованность с определением языка
- Проверка типов данных (операнды, аргументы), отсутствие циклов в графе наследования классов (и др.), проверка существования имен объектов в области видимости
- Дополняет синтаксическое дерево и таблицу символов информацией о типах данных, добавляет преобразования типов
- Синтаксическое дерево — форма промежуточного представления фазы анализа



## Генерация кода в промежуточное представление

- 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



## Оптимизации на уровне промежуточного представления

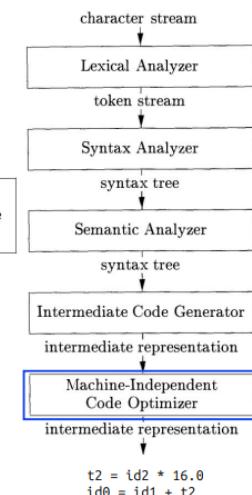
- Оптимизации на уровне промежуточного представления — совокупность применяемых алгоритмов машинно-независимых оптимизаций (проходы, passes)
- Цели оптимизации: минимизация времени, минимизация размера кода, минимизация использования ресурсов
- Оптимизирующие преобразования — длительная фаза компиляции
- Область оптимизации: базовый блок, функция, файл (единица трансляции)

### Трехадресный код (IR)

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

### Трехадресный код (IR)

```
t2 = id2 * 16.0
id0 = id1 + t2
```



## Генерация кода

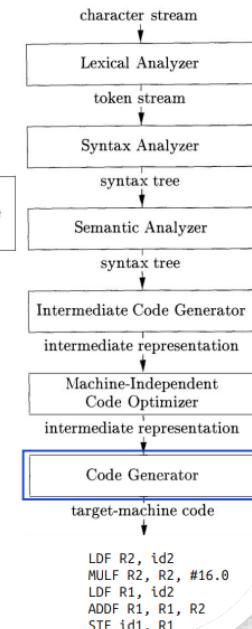
- Генерация кода (code generation) — трансляция промежуточного представления в язык целевой системы (ассемблер)
- Решаются задачи распределения ресурсов целевой архитектуры: распределение регистров, управление стеком, соблюдение ABI
- Компилятор должен иметь описание целевой системы: описание набора команд, их временные характеристики, число регистров, соглашение ABI и др.

### Трехадресный код (IR)

```
t2 = id2 * 16.0
id0 = id1 + t2
```

### Код целевой системы (ассемблер)

```
LDF R2, id2      # Load float id2 to reg R2
MULF R2, R2, #16.0 # R2 = R2 * 16.0
LDF R1, id2      # Load id2 to reg R1
ADDF R1, R1, R2   # R1 = R1 + R2
STF id1, R1       # Store R1 to id1
```



## 2. Понятие формального языка. Синтаксис языка. Семантика. Регулярные выражения для описания языков.

### Понятие формального языка

- **Формальный язык** (formal language) — множество  $L$  конечных слов (строк, цепочек символов) над конечным алфавитом  $A$
- **Синтаксис** (syntax) формального языка — набор правил, описывающих корректный вид его программ (цепочек символов)
- **Семантика** (semantics) формального языка — набор правил, описывающих «смысл» программ (правила области видимости переменных, совместимость типов данных, правила наследования классов, передача параметров в функции)
- Если язык состоит из конечного числа слов (корректных программ), то его можно задать перечислением множества  $L$

$$L = \{a, b, \text{hello}\}, \quad A = \{a, b, h, e, l, o\}$$

### Формальный язык с бесконечным числом строк

- Формальный язык  $L_1$ , состоящий из строк, содержащих произвольное целое число символов  $x$

Примеры цепочек (программ):  $x$ ,  $xxx$ ,  $xx$ ,  $\langle\rangle$ ,  $xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx$

$$L_1 = \{x^n \mid n > 0\}, \text{ где } x^n \text{ — конкатенация символа } x \text{ с самим собой } n \text{ раз}$$

- Язык  $L_2 = \{x^n y^n \mid n > 0\}$  — строки вида: один и более  $x$ , за которым следует такое же число символов  $y$ 
  - $xy$
  - $xxxxuuuu$
  - $xxxx$  — не принадлежит  $L_2$
- Язык  $L_3 = \{x^m y^n \mid m, n > 0\}$  — строки вида: один и более  $x$ , за которым следует хотя бы один  $y$ 
  - $xy$
  - $xxxuy$
- Язык  $L_4 = \{x^m y^n \mid m, n \geq 0\}$  — нуль и более  $x$ , за которым следует нуль и более  $y$ 
  - $x$
  - $yy$
  - $\epsilon$  — пустая строка принадлежит  $L_4$

## Регулярные выражения для описания языков

- Языки  $L_1$  —  $L_4$  имеют простую структуру (простые виды допустимых цепочек)
- Все допустимые строки языков  $L_1, L_3, L_4$  можно описать при помощи *регулярных выражений*
- Все строки языка  $L_4 = \{x^m y^n \mid m, n \geq 0\}$  можно задать регулярным выражением  $\{x^* y^*\}$
- регулярное выражение:  $a^*$  — символ  $a$  повторяется 0 или более раз (звезда Клини, Kleene star)
- Язык  $L_3 = \{x^m y^n \mid m, n > 0\}$ , определение строк языка регулярным выражением:  $\{xx^* yy^*\} = \{x^+ y^+\}$
- регулярное выражение:  $a^+$  — символ  $a$  повторяется 1 или более раз (эквивалентно  $aa^*$ )
- Язык записи регулярных выражений (regular expression)
  - $ab$  — строки из двух символов  $a$  и  $b$  (конкатенация)
  - $a^*$  — строки из нуль или более повторений  $a$  (звезда Клини)
  - $a \mid b$  — строки из символа  $a$  или  $b$
  - $(a \mid b)^*$  — строки из нуля или большего числа элементов  $a$  или  $b$   
(строка из нуля или большего числа знаков, каждый из которых может быть  $a$  или  $b$ )
- Операция  $|$  имеет приоритет над  $*$ :
  - $a \mid b^*$
  - $(aab \mid ab)^*$

## Регулярные выражения для описания языков

- Область применения регулярных выражений — определение языковых лексем на этапе лексического анализа
- Идентификатор в языке программирования:  
$$(\underline{|}a|b|c|\dots|z|A|B|\dots|Z)(\underline{|}a|b|c|\dots|z|A|B|\dots|Z|0|1|2|\dots|9|)^*$$
- Целочисленный литерал в десятичной системе исчисления:  
$$(1|2|\dots|9)(0|1|2|\dots|9)^*$$
- Ключевые слова:

if|for|while|do

### 3. Формальные грамматики. Эквивалентные грамматики. Иерархия Хомского. Форма Бэкуса-Наура.

## Формальные грамматики

- **Формальная грамматика языка** (formal grammar) — способ описания формального языка, выделения подмножества  $L$  из множества всех слов некоторого конечного алфавита  $A$
- **Порождающие грамматики** — задают правила, с помощью которых можно построить любое слово языка
- **Распознающие** (аналитические) грамматики — позволяют по данному слову определить, принадлежит оно языку или нет (синтаксически корректная программа или нет)
- **Формальная грамматика  $G$**  — это описание формального языка (его синтаксиса) четверкой

$$G = (V_T, V_N, P, S),$$

где

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

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

## Формальные грамматики

- **Грамматика** используется для генерации последовательностей символов, составляющих строки языка, начиная со стартового символа  $S$  и последовательно заменяя его или нетерминалы, которые появятся позднее, с помощью одного из порождений грамматики
- На каждом этапе к нетерминалу из левой части применяется продукция, заменяющая этот нетерминал последовательностью символов своей правой части
- Процесс прекращается после получения строки, состоящей только из терминальных символов (не содержащей нетерминалов)
- Языку принадлежат те, и только те строки символов, которые можно получить с помощью заданной грамматики (породить, вывести)
- **Пример:** грамматика языка  $L_2 = \{x^n y^n \mid n > 0\}$

$$G = (V_T, V_N, P, S),$$

- $V_T = \{x, y\}$
- $V_N = \{S\}$
- $P = \{$
- $S \rightarrow xSy,$
- $S \rightarrow xy$
- }

## Формальные грамматики

- Язык  $L4 = \{x^m y^n \mid m, n \geq 0\}$  — нуль и более  $x$ , за которым следует нуль и более  $y$ 
  - $x$
  - $yy$
- $\epsilon$  — пустая строка принадлежит  $L4$
- Грамматика для языка  $L4$ :

$$G4 = (V_T, V_N, P, S),$$

- $V_T = \{x, y\}$
- $V_N = \{S, B\}$
- $P = \{$ 
  - $S \rightarrow xS,$
  - $S \rightarrow yB,$
  - $S \rightarrow x,$
  - $S \rightarrow y,$
  - $B \rightarrow yB,$
  - $B \rightarrow y,$
  - $S \rightarrow \epsilon$ $\}$

Пример вывода строки « $xxxyy$ » из грамматики:

$$S \Rightarrow xS \Rightarrow xxS \Rightarrow xxyB \Rightarrow xxyyB \Rightarrow xxxyy$$

1.  $S \Rightarrow xS$  — порождение из первой продукции
2.  $S \Rightarrow xS \Rightarrow xxS$  — порождение из первой продукции)
3.  $S \Rightarrow xS \Rightarrow xxS \Rightarrow xxyB$  — порождение из второй продукции
4.  $S \Rightarrow xS \Rightarrow xxS \Rightarrow xxyB \Rightarrow xxyyB$  — порождение из пятой продукции
5.  $S \Rightarrow xS \Rightarrow xxS \Rightarrow xxyB \Rightarrow xxyyB \Rightarrow xxyy$  — порождение из шестой продукции

## Эквивалентные грамматики

- Для генерации языка обычно не существует единственной грамматики
- Язык  $L4 = \{x^m y^n \mid m, n \geq 0\}$  — нуль и более  $x$ , за которым следует нуль и более  $y$

Грамматика  $G1 = (V_T, V_N, P, S)$ ,

- $V_T = \{x, y\}$
- $V_N = \{S, B\}$
- $P = \{$ 
  - $S \rightarrow xS,$
  - $S \rightarrow yB,$
  - $S \rightarrow x,$
  - $S \rightarrow y,$
  - $B \rightarrow yB,$
  - $B \rightarrow y,$
  - $S \rightarrow \epsilon$ $\}$

Грамматика  $G2 = (V_T, V_N, P, S)$ ,

- $V_T = \{x, y\}$
- $V_N = \{X, Y\}$
- $P = \{$ 
  - $S \rightarrow XY,$
  - $X \rightarrow xX,$
  - $X \rightarrow \epsilon,$
  - $Y \rightarrow yY,$
  - $Y \rightarrow \epsilon$ $\}$

Пример вывода строки « $xxxyy$ » из  $G2$ :

$$S \Rightarrow XY \Rightarrow xXY \Rightarrow xxXY \Rightarrow xxY \Rightarrow xxyY \Rightarrow xxyyY \Rightarrow xxxyy$$

- Грамматики  $G1$  и  $G2$  генерируют язык  $L4$
- Две грамматики, генерирующие один и тот же язык, называются **эквивалентными**

## Формальные грамматики общего вида

- Формальная грамматика  $G$  — это описание формального языка (его синтаксиса) четверкой

$$G = (V_T, V_N, P, S),$$

где  $P$  — множество продукции, продукции:  $a \rightarrow b$

- В общем случае левые части продукции могут содержать более одного символа

- Пример грамматики  $G = (\{a\}, \{S, N, Q, R\}, P, S)$ :

$P = \{$

$S \rightarrow QNQ,$

$QN \rightarrow QR, \quad // N \text{ можно заменить на } R, \text{ только если } N \text{ следует после } Q$

$RN \rightarrow NNR,$

$RQ \rightarrow NNQ, \quad // R \text{ можно заменить на } NN, \text{ только если после } R \text{ следует } Q$

$N \rightarrow a,$

$Q \rightarrow \epsilon$

$\}$

- Продукции 2, 4 являются **контекстно-зависимыми** (context-sensitive production)

## Формальные грамматики общего вида

- Грамматика  $G = (\{a\}, \{S, N, Q, R\}, P, S)$ :

$P = \{$

$S \rightarrow QNQ,$

$QN \rightarrow QR, \quad // N \text{ можно заменить на } R, \text{ только если } N \text{ следует после } Q$

$RN \rightarrow NNR,$

$RQ \rightarrow NNQ, \quad // R \text{ можно заменить на } NN, \text{ только если после } R \text{ следует } Q$

$N \rightarrow a,$

$Q \rightarrow \epsilon$

$\}$

- Порождает язык  $\{a^m \mid m = 2^k, k = 1, 2, \dots\} = \{aa, aaaa, aaaaaaaaa, \dots\}$ ,  $m$  — положительная степень двойки

- $S \Rightarrow QNQ \Rightarrow QRQ \Rightarrow QNNQ \Rightarrow \dots \Rightarrow aa$

- $S \Rightarrow QNQ \Rightarrow QRQ \Rightarrow QNNQ \Rightarrow QRNQ \Rightarrow QNNRQ \Rightarrow QNNNNQ \Rightarrow \dots \Rightarrow aaaa$

## Формальные грамматики общего вида

- Грамматика  $G = (\{a\}, \{S, N, Q, R\}, P, S)$ :

```

 $P = \{$ 
   $S \rightarrow QNQ,$ 
   $QN \rightarrow QR,$  //  $N$  можно заменить на  $R$ , только если  $N$  следует после  $Q$ 
   $RN \rightarrow NNR,$ 
   $RQ \rightarrow NNQ,$  //  $R$  можно заменить на  $NN$ , только если после  $R$  следует  $Q$ 
   $N \rightarrow a,$ 
   $Q \rightarrow \epsilon$ 
 $\}$ 

```

- Порождает язык  $\{a^m \mid m = 2^k, k = 1, 2, \dots\} = \{aa, aaaa, aaaaaaaaa, \dots\}$ ,  $m$  — положительная степень двойки
- $S \Rightarrow QNQ \Rightarrow QRQ \Rightarrow QNNQ \Rightarrow \dots \Rightarrow aa$
- $S \Rightarrow QNQ \Rightarrow QRQ \Rightarrow QNNQ \Rightarrow QRNQ \Rightarrow QNNRQ \Rightarrow QNNNNQ \Rightarrow \dots \Rightarrow aaaa$

## 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 продукции некоторым схемам

Тип	Грамматика	Вид продукции	Применение
Тип 0	<b>Неограниченные грамматики</b> (рекурсивно перечислимые, recursively enumerable)	$a \rightarrow b$ <ul style="list-style-type: none"> <li><math>a \in V^*</math> — непустая цепочка, содержащая хотя бы один нетерминал</li> <li><math>b \in V^*</math> — любая цепочка символов из <math>V = V_T \cup V_N</math> (эквивалентны машинам Тьюринга)</li> </ul>	Практического применения в силу своей сложности (общности) такие грамматики не имеют
Тип 1	<b>Контекстно-зависимые</b> (context-sensitive)	$aAb \rightarrow acb$ <ul style="list-style-type: none"> <li><math>a, b \in V^*</math> — любая цепочка символов из <math>V</math></li> <li><math>c \in V^*</math> — непустая цепочка из <math>V</math></li> <li><math>A \in V_N</math> (эквивалентны линейно ограниченным автоматам)</li> </ul>	Анализ текстов на естественных языках, при построении компиляторов практически не используются
Тип 2	<b>Контекстно-свободные</b> (context-free)	$A \rightarrow b$ <ul style="list-style-type: none"> <li><math>A \in V_N</math></li> <li><math>b \in V^*</math> — любая цепочка символов из <math>V</math> (эквивалентны магазинным автоматам)</li> </ul>	Описание синтаксиса компьютерных языков
Тип 3	<b>Регулярные</b> (regular)	$A \rightarrow Bc$ или $A \rightarrow c$ — леволинейные грамматики $A \rightarrow cB$ или $A \rightarrow c$ — праволинейные грамматики <ul style="list-style-type: none"> <li><math>A, B \in V_N</math></li> <li><math>c \in V_T^*</math> (эквивалентны конечным автоматам)</li> </ul>	Описание простейших конструкций: идентификаторов, строк, констант, языков ассемблера, командных процессоров

## Иерархия Хомского

- Иерархия Хомского — классификация формальных языков и формальных грамматик на 4 типа по их условной сложности [Ноам Хомский, 1956]
- Для отнесения грамматики к определенному типу необходимо соответствие всех её продукции некоторым схемам

Тип	Грамматика	Вид продукции	Применение
Тип 0	Неограниченные грамматики (рекурсивно перечислимые, recursively enumerable)	$a \rightarrow b$ <ul style="list-style-type: none"><li>▪ <math>a \in V^*</math> — непустая цепочка, содержащая хотя бы один нетерминал</li><li>▪ <math>b \in V^*</math> — любая цепочка символов из</li></ul>	Практического применения в силу своей сложности (общности) такие грамматики не имеют
Тип 1		<ul style="list-style-type: none"><li>▪ Один и тот же язык может быть задан разными грамматиками, относящимися к разным типам</li><li>▪ Язык относится к наиболее простому из типов грамматик, которыми может быть описан</li></ul>	при че
Тип 2		<ul style="list-style-type: none"><li>▪ Например, формальный язык, описанный грамматикой с фразовой структурой, контекстно-зависимой и контекстно-свободной грамматиками, будет контекстно-свободным</li></ul>	ков
Тип 3		<ul style="list-style-type: none"><li>▪ Наиболее сложные — языки с фразовой структурой (сюда можно отнести естественные языки), далее — К3-языки, КС-языки и самые простые — регулярные языки</li></ul>	ов
		(эквивалентны конечным автоматам)	

## Компьютерное описание контекстно-свободных грамматик

- Форма Бэкуса-Наура (Backus–Naur Form — BNF) // [Algol-58-60](#)

```
<expression> ::= <term> | <term> "+" <expression>
<term>    ::= <factor> | <factor> "*" <term>
<factor>   ::= <constant> | <variable> | "(" <expression> ")"
<variable> ::= "x" | "y" | "z"
<constant> ::= <digit> | <digit> <constant>
<digit>    ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

### Поддержка БНФ

- ANTLR
- Coco/R
- JavaCC
- Bison
- Yacc

- Расширенная форма Бэкуса-Наура (Extended Backus–Naur Form — EBNF) // [N. Wirth, 1977](#)  
ISO/IEC 14977 (1996). Синтаксический метаязык – Расширенная Форма Бэкуса-Наура

```
expression = term , [ "+" , expression ];
term      = factor , [ "*" , term ];
factor    = constant | variable | "(" , expression , ")";
variable  = "x" | "y" | "z";
constant  = digit , { digit };
digit     = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

### Форма Бэкуса-Наура (ФБН)

Условные обозначения:

<Имя> – нетерминальный символ – конструкция;

Имя – терминальный символ – символ алфавита;

::= – «можно заменить на»;

| – «или»

**Пример:**

<Целое> ::= <Знак><Целое без знака>|<Целое без знака>

<Целое без знака> ::= <Цифра><Целое без знака>|<Цифра>

<Цифра> ::= 0|1|2|3|4|5|6|7|8|9

<Знак> ::= +| -

## 4. Синтаксически управляемая трансляция. Деревья разбора. Семантические правила.

### Синтаксически управляемая трансляция

- Синтаксически управляемая трансляция (syntax-directed translation) выполняется путем присоединения правил (программных фрагментов) к продукциям грамматики

```
expr -> expr1 + term      {  
    Трансляция expr1  
    Трансляция term  
    Обработка +  
}
```

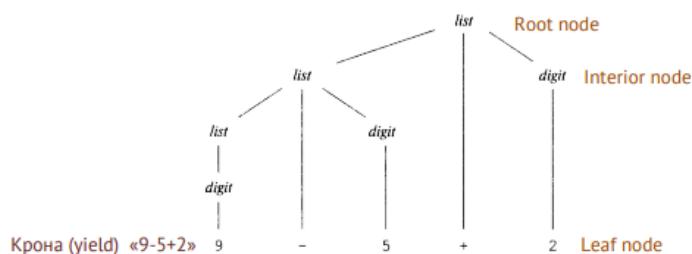
- Фрагменты выполняются при использовании продукции в процессе синтаксического анализа
- Объединенный результат выполнения всех фрагментов грамматики в порядке, определяемом синтаксическим анализом, и есть трансляция заданной программы, к которой применяется этот процесс

(5 стр. 5 лекции)

### 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  $\epsilon$
  3. Внутренний узел (interior, internal) – нетерминал
  4. Если внутренний узел  $A$  имеет дочерние узлы  $X_1, X_2, \dots, X_N$ , то должна существовать продукция  $A \rightarrow X_1X_2\dots X_N$ , где  $X_i$  – терминальный или нетерминальный символ
- Листья дерева разбора образуют корону (yield) – строку, выведенную (derived), или порожденную (generated), из стартового символа в корне

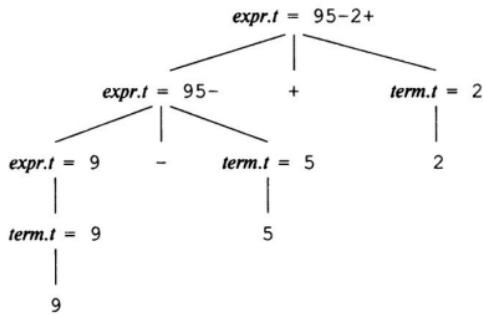
Дерево разбора для строки «9-5+2»



(16 стр. 4 лекции)

## Семантические правила (semantic rules)

- Назначаем терминалам и нетерминалам атрибуты (исходя из целей трансляции)
- Назначаем каждой продукции **семантическое правило** (semantic rule) – правило вычисления значений атрибутов, связанных с символами продукции



Аннотированное дерево разбора 95 – 2+

(стр.7 лекция 5)

ПРОДУКЦИЯ	СЕМАНТИЧЕСКОЕ ПРАВИЛО
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t    term.t    '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t    term.t    '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Синтаксически управляемые определения для трансляции инфиксных выражений в постфиксные  
( $a || b$  – конкатенация строк a и b)

## Простые синтаксически управляемые определения

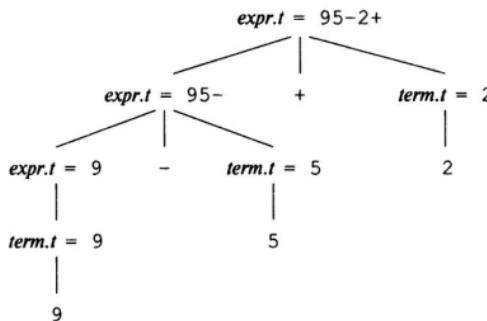
- Синтаксически управляемое определение называемое простым (simple),** если строка, представляющая трансляцию нетерминала в заголовке каждой продукции, является **конкатенацией трансляций нетерминалов в теле продукции в том же порядке, в котором они встречаются в продукции,** с необязательными дополнительными строками
- Простое синтаксически управляемое определение может быть реализовано путем печати (выдачи) только дополнительных строк в порядке их появления в определении

ПРОДУКЦИЯ	СЕМАНТИЧЕСКОЕ ПРАВИЛО
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t    term.t    '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t    term.t    '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Синтаксически управляемые определения для трансляции инфиксных выражений в постфиксные  
( $a || b$  – конкатенация строк a и b)

## Семантические правила (semantic rules)

- Назначаем терминалам и нетерминалам атрибуты (исходя из целей трансляции)
- Назначаем каждой продукции **семантическое правило** (semantic rule) – правило вычисления значений атрибутов, связанных с символами продукции



Аннотированное дерево разбора 95 – 2 +

ПРОДУКЦИЯ	СЕМАНТИЧЕСКОЕ ПРАВИЛО
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t    term.t    '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t    term.t    '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
$\dots$	$\dots$
$term \rightarrow 9$	$term.t = '9'$

Синтаксически управляемые определения для трансляции инфиксных выражений в постфиксные  
( $a || b$  – конкатенация строк  $a$  и  $b$ )

Семантические правила – операции со строками!  
Можно обобщить на программные фрагменты

## 5. Нисходящий анализ. Анализ методом рекурсивного спуска. Левая рекурсия в продукциях. Абстрактное синтаксическое дерево.

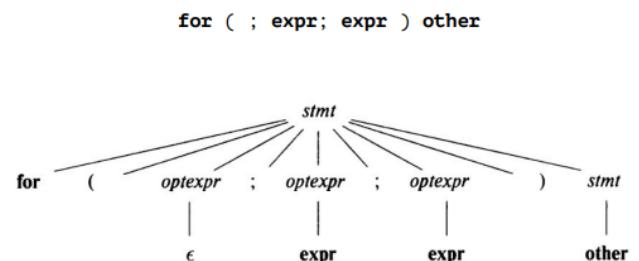
### Нисходящий анализ (top-down parsing)

- Построение дерева разбора методом сверху вниз (top-down) начинается с корня, стартового нетерминала, и осуществляется многократным выполнением двух шагов:
  - В узле  $N$ , помеченному нетерминалом  $A$  выбираем одну из производств для  $A$  и строим дочерние узлы  $N$  для символов из правой части производств
  - Находим следующий узел, в котором должно быть построено поддерево (обычно это крайний слева неразвернутый нетерминал дерева)

Грамматика подмножества языка C

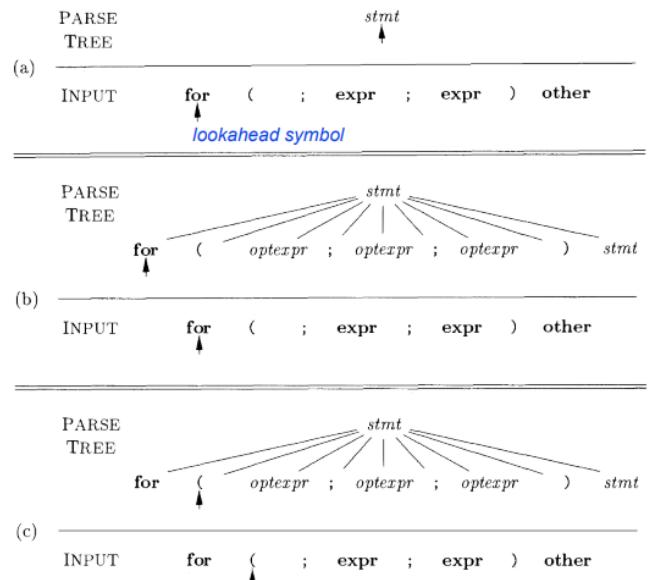
$$\begin{aligned}
 stmt &\rightarrow expr ; \\
 &| if ( expr ) stmt \\
 &| for ( optexpr ; optexpr ; optexpr ) stmt \\
 &| other \\
 \\ 
 optexpr &\rightarrow \epsilon \\
 &| expr
 \end{aligned}$$

Дерево разбора методом сверху вниз  
для строки



## Нисходящий анализ (top-down parsing)

- Для некоторых грамматик построение дерева разбора может быть реализовано за один проход слева направо по входной строке
- Текущий сканируемый терминал входной строки называют *сканируемым символом, символом предпросмотра* или "предсимволом" (lookahead symbol)
- Как только у узла дерева разбора создаются дочерние узлы, следует рассмотреть крайний слева узел



## Анализ методом рекурсивного спуска (recursive-descent parsing)

- Анализ методом рекурсивного спуска** (recursive-descent parsing) – тип нисходящего синтаксического анализа, при котором для обработки входной строки используется множество рекурсивных процедур, для каждого нетерминала грамматики
- Предиктивный анализ методом рекурсивного спуска** (предсказывающий, predictive parsing) – сканируемый символ однозначно определяет поток управления в теле рекурсивной процедуры для каждого нетерминала
- В предиктивном анализе последовательность вызовов процедур при обработке входной строки неявно определяет его дерево разбора и при необходимости может использоваться для явного построения дерева

## Предиктивный анализ методом рекурсивного спуска (predictive recursive-descent parsing)

### Грамматика подмножества языка C

```
stmt → expr ;
| if ( expr ) stmt
| for ( optexpr ; optexpr ; optexpr ) stmt
| other

optexpr → ε
| expr
```

- Анализ начинается с вызова процедуры для стартового нетерминала stmt()

```
void stmt() {
    switch ( lookahead ) {
        case expr:
            match(expr); match(''); break;
        case if:
            match(if); match('('); match(expr); match(')'); stmt();
            break;
        case for:
            match(for); match('(');
            optexpr(); match(''); optexpr(); match(''); optexpr();
            match(')'); stmt(); break;
        case other:
            match(other); break;
        default:
            report("syntax error");
    }
}

void optexpr() {
    if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
    if ( lookahead == t ) lookahead = nextTerminal;
    else report("syntax error");
}
```

## Предиктивный анализ методом рекурсивного спуска (predictive recursive-descent parsing)

### Грамматика подмножества языка C

```
stmt → expr ;
| if ( expr ) stmt
| for ( optexpr ; optexpr ; optexpr ) stmt
| other

optexpr → ε
| expr
```

```
void stmt() {
    switch ( lookahead ) {
        case expr:
            match(expr); match(''); break;
        case if:
            match(if); match('('); match(expr); match(')'); stmt();
            break;
        case for:
            match(for); match('(');
            optexpr(); match(''); optexpr(); match(''); optexpr();
            match(')'); stmt(); break;
        case other:
            match(other); break;
        default:
            report("syntax error");
    }
}
```

- Предиктивный анализ основан на информации о первых символах, которые могут быть сгенерированы телом продукции
- FIRST( $\alpha$ ) – множество терминалов, которые могут появиться в качестве первого символа одной или нескольких строк, сгенерированных из  $\alpha$
- $\alpha$  начинается либо с терминала, который, является единственным символом в FIRST( $\alpha$ ), либо с нетерминала
- FIRST(stmt) = {expr, if, for, other};    FIRST(expr;) = {expr}

# Предиктивный анализ методом рекурсивного спуска (predictive recursive-descent parsing)

## Грамматика подмножества языка C

```
stmt → expr ;
      | if ( expr ) stmt
      | for ( optexpr ; optexpr ; optexpr ) stmt
      | other

optexpr → ε
      | expr
```

```
void stmt() {
    switch ( lookahead ) {
        case expr:
            match(expr); match(''); break;
        case if:
            match(if); match('('); match(expr); match(')'); stmt();
            break;
        case for:
            match(for); match('(');
            optexpr(); match(''); optexpr(); match(''); optexpr();
            match(')'); stmt(); break;
        case other:
            match(other); break;
        ...
    }
}
```

- Если в грамматике присутствуют две продукции:
  - $A \rightarrow \alpha$
  - $A \rightarrow \beta$
- Предиктивный анализатор требует, чтобы множества FIRST( $\alpha$ ) и FIRST( $\beta$ ) были непересекающимися.
- Это обеспечивает возможность использования текущего сканируемого символа (lookahead) для принятия решения, какую из продуктов следует применить
- Если сканируемый символ (lookahead) принадлежит множеству FIRST( $\alpha$ ), используется продукция для  $\alpha$ ; в противном случае, если сканируемый символ принадлежит множеству FIRST( $\beta$ ), применяется продукция

## Левая рекурсия в продукциях

- Анализатор на основе рекурсивного спуска зациклиться при "леворекурсивных" продукциях типа:

```
expr → expr + term
```

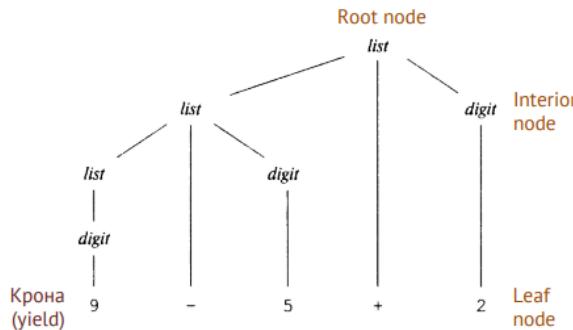
```
void expr() {
    expr();
    match('+')
    term();
}
```

- Сканируемый символ lookahead изменяется только тогда, когда он соответствует терминалу в теле продукции, между рекурсивными вызовами expr() не происходит никаких изменений
- Как устранить левую рекурсию в продукциях?

# Абстрактное синтаксическое дерево

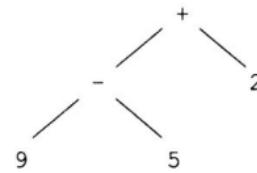
**Дерево разбора (parse tree)** –  
древовидное представление порождения  
строки языка из стартового символа  
грамматики

Дерево разбора для строки «9-5+2»



**Абстрактное синтаксическое дерево**  
(abstract syntax tree) – древовидное  
представление порождения строки языка,  
в котором узлами являются программные  
конструкции

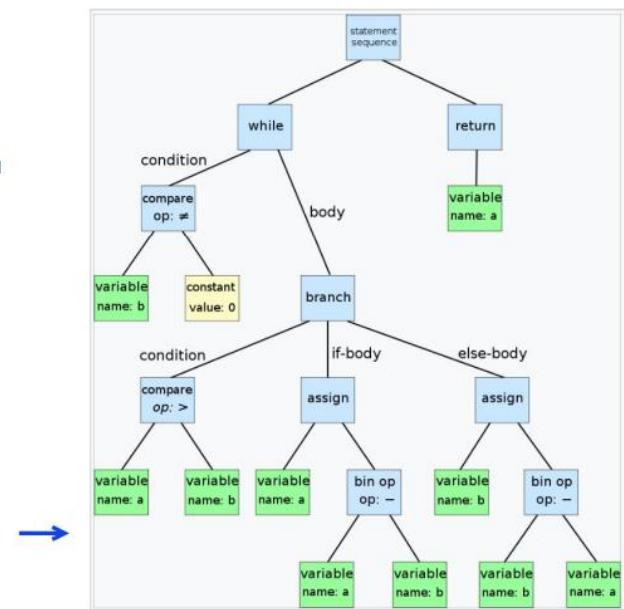
Абстрактное синтаксическое дерево разбора  
для строки «9-5+2»



## Абстрактное синтаксическое дерево

- Абстрактное синтаксическое дерево**  
(abstract syntax tree, AST) – конечное помеченное  
ориентированное дерево, в котором внутренние  
вершины сопоставлены с операторами языка  
программирования, а листья – с соответствующими  
операндами
- Листья являются пустыми операторами  
и представляют только переменные и константы

```
while b ≠ 0:  
    if a > b:  
        a := a - b  
    else:  
        b := b - a  
return a
```



(22 стр. 5 лекция)

## 6. Таблица символов. Область видимости. Цепочки таблиц символов. Лексический анализ. Регулярные выражения. Архитектура лексического анализатора на основе диаграммы переходов.

### Таблица символов

- **Таблица символов** (symbol table) – структура данных, которая хранит информацию о конструкциях исходной программы
- Таблица заполняется и модифицируется инкрементно на начальной стадии работы компилятора
- Записи в таблице символов:
  - информацию об идентификаторах
  - местоположение в памяти
- Поддержка множественных объявлений одного и того же идентификатора (отдельная таблица символов для каждой области видимости)
- Операции:
  - Добавление символа с указанием типа токена
  - Поиск по символу

(стр. 10 лекция 6)

### Область видимости (scope)

```
{  
    int x; char y;      // Объявление x, y  
  
    {  
        bool y;  
        x; y;          // Объявление y, использование x:int, y:bool  
    }  
  
    x; y;              // Использование x:int, y:char  
}
```

- **Правило последнего вложения** (most-closely nested) для блоков – идентификатор *x* находится в области видимости последнего по вложенности объявления *x*
- Определение идентификатора следует искать путем перебора блоков изнутри наружу (снизу вверх), начиная с блока, в котором находится интересующий идентификатор
- **Реализация таблиц символов для вложенных блоков:**
  - **Стек локальных таблиц символов:** на вершине стека находится таблица для текущего блока, ниже в стеке располагаются таблицы охватывающих блоков
  - **Единая хеш-таблица:** при выходе из блока компилятор должен отменить все изменения, внесенные в хеш-таблицу объявлениями в блоке (вспомогательный стек для отслеживания изменений в хеш-таблице при обработке блока)

## Области видимости

```
1) { int x1; int y1;  
2)   { int w2; bool y2; int z2;  
3)     ... w2 ...; ... x1 ...; ... y2 ...; ... z2 ...;  
4)   }  
5)     ... w0 ...; ... x1 ...; ... y1 ...;  
6) }
```

Два вложенных блока:

- Блок 1: x<sub>1</sub>, y<sub>1</sub>,
- Блок 2: w<sub>2</sub>, y<sub>2</sub>, z<sub>2</sub>
- Блок 0 (внешний): w<sub>0</sub>

- Реализация правила ближайшего вложенного блока на базе цепочек таблиц символов – таблица для вложенного блока указывает на таблицу для охватывающего блока (outer)

$B_0$ :	<table border="1"><tr><td>w</td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td>...</td></tr></table>	w								...
w										
		...								
$B_1$ :	<table border="1"><tr><td>x</td><td>int</td><td></td></tr><tr><td>y</td><td>int</td><td></td></tr><tr><td>z</td><td>int</td><td></td></tr></table>	x	int		y	int		z	int	
x	int									
y	int									
z	int									
$B_2$ :	<table border="1"><tr><td>w</td><td>int</td><td></td></tr><tr><td>y</td><td>bool</td><td></td></tr><tr><td>z</td><td>int</td><td></td></tr></table>	w	int		y	bool		z	int	
w	int									
y	bool									
z	int									

## Области видимости

```
1) { int x1; int y1; Создание B1  
2)   { int w2; bool y2; int z2;  
3)     ... w2 ...; ... x1 ...; ... y2 ...; ... z2 ...;  
4)   }  
5)     ... w0 ...; ... x1 ...; ... y1 ...;  
6) }
```

Два вложенных блока:

- Блок 1: x<sub>1</sub>, y<sub>1</sub>,
- Блок 2: w<sub>2</sub>, y<sub>2</sub>, z<sub>2</sub>
- Блок 0 (внешний): w<sub>0</sub>

- Реализация правила ближайшего вложенного блока на базе цепочек таблиц символов – таблица для вложенного блока указывает на таблицу для охватывающего блока (outer)

$B_0$ :	<table border="1"><tr><td>w</td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td>...</td></tr></table>	w								...
w										
		...								
$B_1$ :	<table border="1"><tr><td>x</td><td>int</td><td></td></tr><tr><td>y</td><td>int</td><td></td></tr><tr><td>z</td><td>int</td><td></td></tr></table>	x	int		y	int		z	int	
x	int									
y	int									
z	int									
$B_2$ :	<table border="1"><tr><td>w</td><td>int</td><td></td></tr><tr><td>y</td><td>bool</td><td></td></tr><tr><td>z</td><td>int</td><td></td></tr></table>	w	int		y	bool		z	int	
w	int									
y	bool									
z	int									

## Области видимости

```
1) { int x1; int y1;
2)   { int w2; bool y2; int z2; Создание B2
3)     ... w2 ...; ... x1 ...; ... y2 ...; ... z2 ...
4)   }
5)   ... w0 ...; ... x1 ...; ... y1 ...
6) }
```

Два вложенных блока:

- Блок 1: x<sub>1</sub>, y<sub>1</sub>,
- Блок 2: w<sub>2</sub>, y<sub>2</sub>, z<sub>2</sub>
- Блок 0 (внешний): w<sub>0</sub>

- Реализация правила ближайшего вложенного блока на базе цепочек таблиц символов – таблица для вложенного блока указывает на таблицу для охватывающего блока (outer)

$B_0$ :	w		
	...		

$B_1$ :	x	int	
	y	bool	
	z	int	

$B_2$ :	w	int	
	y	bool	
	z	int	

## Области видимости

```
1) { int x1; int y1;
2)   { int w2; bool y2; int z2;
3)     ... w2 ...; ... x1 ...; ... y2 ...; ... z2 ...
4)   }
5)   ... w0 ...; ... x1 ...; ... y1 ...; Переключение на B1
6) }
```

- Два вложенных блока:
- Блок 1: x<sub>1</sub>, y<sub>1</sub>,
  - Блок 2: w<sub>2</sub>, y<sub>2</sub>, z<sub>2</sub>
  - Блок 0 (внешний): w<sub>0</sub>

- Реализация правила ближайшего вложенного блока на базе цепочек таблиц символов – таблица для вложенного блока указывает на таблицу для охватывающего блока (outer)

$B_0$ :	w		
	...		

$B_1$ :	x	int	
	y	int	

$B_2$ :	w	int	
	y	bool	
	z	int	



Цепочки таблиц символов (symbol table chains) - это метод организации таблиц символов в компиляторах, который позволяет обрабатывать области видимости переменных.

Каждая таблица символов, как правило, соответствует отдельной области видимости в программе (например, глобальной области, области функции, блока кода и т.д.). В цепочке таблиц символов каждая таблица символов связана с предыдущей таблицей символов, образуя последовательность или цепочку таблиц.

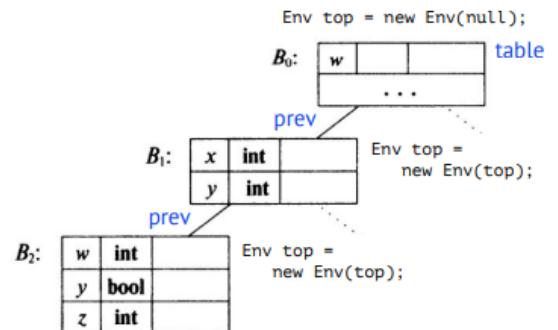
Когда компилятор встречает новую область видимости (например, новую функцию или блок кода), он создает новую таблицу символов и добавляет ее в цепочку. При этом новая таблица символов становится текущей для данной области видимости. Когда область видимости заканчивается, таблица символов удаляется, и компилятор переключается на предыдущую таблицу символов в цепочке.

Цепочки таблиц символов позволяют эффективно управлять областями видимости переменных, обеспечивая быстрый доступ к символам в текущей области видимости и их корректное разрешение при обнаружении и использовании в коде.



## Реализация цепочки таблиц символов

```
package symbols;
import java.util.*;
public class Env {
    private Hashtable table;
    protected Env prev;
    public Env(Env p) {
        table = new Hashtable(); prev = p;
    }
    public void put(String s, Symbol sym) {
        table.put(s, sym);
    }
    public Symbol get(String s) {
        for( Env e = this; e != null; e = e.prev ) {
            Symbol found = (Symbol)(e.table.get(s));
            if( found != null ) return found;
        }
        return null;
    }
}
```

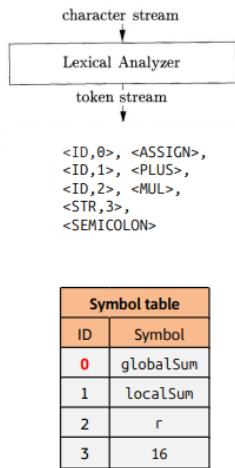


# Лексический анализ

- Лексический анализатор (lexical analyzer, lexer, scanner) — разбивает входную программу на последовательность лексем (lexeme), минимально значимых единиц входного языка
- Тип допустимых лексем определяется описанием языка
- Игнорирует пробельные символы, комментарии, отслеживает номер текущей строки для корректного информирования о положении возможных ошибок

```
// Увеличить сумму  
globalSum = localSum + r * 16;
```

Лексемы: «globalSum», «=», «localSum», «+», «r», «\*», «16», «;»



- Для каждой найденной лексемы анализатор формирует *токен* (token) — пара *имя-токена, значение-атрибута*, имя-токена — тип/класс лексемы, значение-атрибута — непосредственно лексема или ссылка на запись в таблице символов

Tokens: <ID,0>, <ASSIGN>, <ID,1>, <PLUS>, <ID,2>, <MUL>, <STR,3>, <SEMICOLON>

Token-names: ID, ASSIGN, PLUS, MUL, STR, SEMICOLON

## Регулярные выражения для описания языков

- Языки  $L_1$  —  $L_4$  имеют простую структуру (простые виды допустимых цепочек)
- Все допустимые строки языков  $L_1, L_3, L_4$  можно описать при помощи регулярных выражений
- Все строки языка  $L_4 = \{x^m y^n \mid m, n \geq 0\}$  можно задать регулярным выражением  $\{x^*y^*\}$
- регулярное выражение:  $a^*$  — символ  $a$  повторяется 0 или более раз (звезда Клини, Kleene star)
- Язык  $L_3 = \{x^m y^n \mid m, n > 0\}$ , определение строк языка регулярным выражением:  $\{xx^*yy^*\} = \{x^*y^*\}$
- регулярное выражение:  $a^+$  — символ  $a$  повторяется 1 или более раз (эквивалентно  $aa^*$ )
- Язык записи регулярных выражений (regular expression)
  - $ab$  — строки из двух символов  $a$  и  $b$  (конкатенация)
  - $a^*$  — строки из нуль или более повторений  $a$  (звезда Клини)
  - $a \mid b$  — строки из символа  $a$  или  $b$
  - $(a \mid b)^*$  — строки из нуля или большего числа элементов  $a$  или  $b$   
(строка из нуля или большего числа знаков, каждый из которых может быть  $a$  или  $b$ )
- Операция  $|$  имеет приоритет над  $^*$ :
  - $a \mid b^*$
  - $(aab \mid ab)^*$

# Регулярные выражения для описания языков

- Область применения регулярных выражений — определение языковых лексем на этапе лексического анализа
- Идентификатор в языке программирования:

$(\_|a|b|c|\dots|z|A|B|\dots|Z)(\_|a|b|c|\dots|z|A|B|\dots|Z|0|1|2|\dots|9|)^*$

- Целочисленный литерал в десятичной системе исчисления:

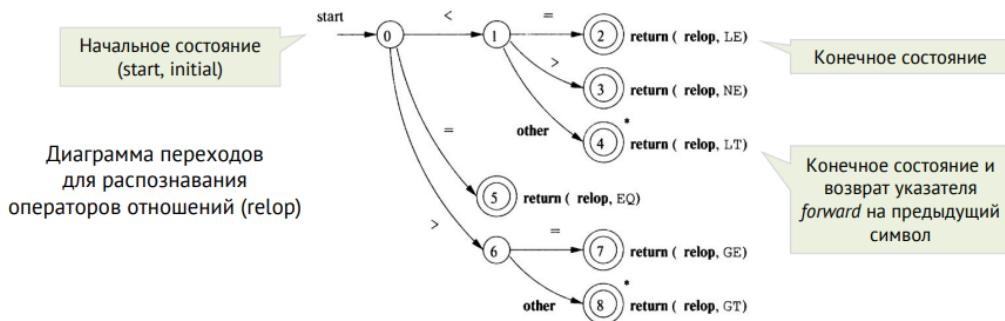
$(1|2|\dots|9)(0|1|2|\dots|9)^*$

- Ключевые слова:

if|for|while|do

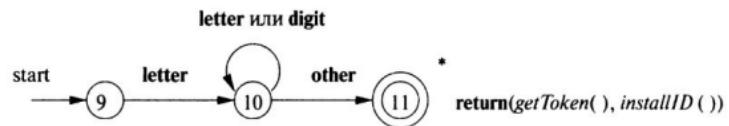
## Диаграммы переходов

- **Диаграмма переходов** (transition diagram, state transition diagram) — ориентированный граф, задающий возможные состояния лексического анализатора и переходы между ними в ходе распознавания токенов
- **Состояния** (state) — узлы графа, представляет ситуацию, которая может возникнуть в процессе сканирования входного потока в поисках лексемы, соответствующей одному из нескольких шаблонов
- **Допускающее состояние** (конечное, принимающее, accepting, final) — искомая лексема найдена
- **Ребро** (дуга, edge) — показывает **переход** (transition) при чтении символа, которым помечена дуга
- **Детерминированная диаграмма переходов** (deterministic) — имеется не более одной дуги, выходящей из данного состояния с данным символом среди ее меток

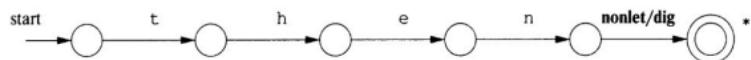


# Распознавание зарезервированных слов и идентификаторов

- **Вариант 1** – внести зарезервированные слова в таблицу символов

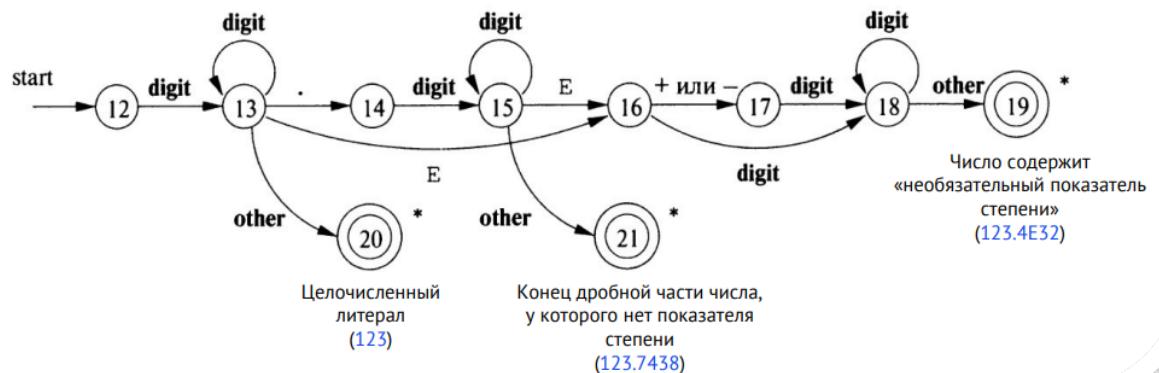


- **Вариант 2** – создать отдельные диаграммы переходов для каждого ключевого слова



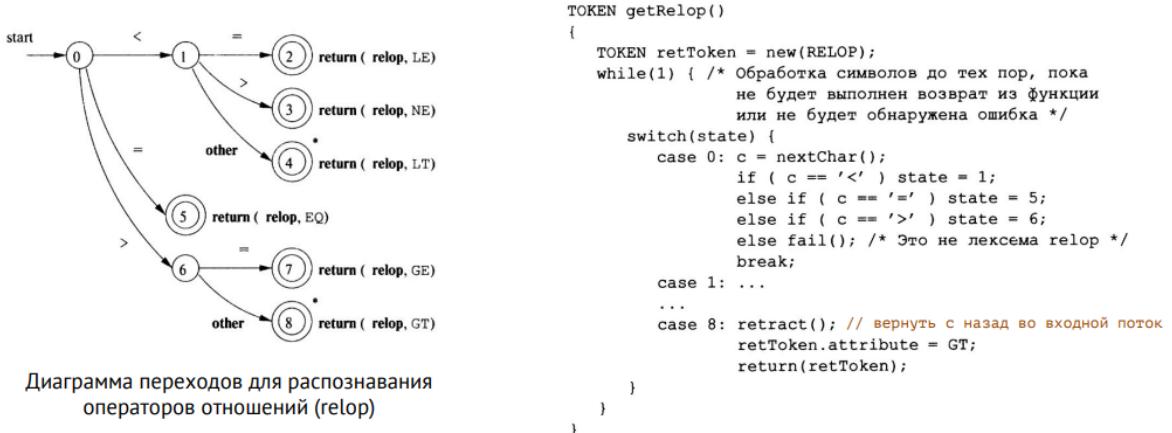
## Диаграмма переходов для беззнаковых чисел

- Если первый встреченный символ – цифра, переходим в состояние 13
- В состоянии 13 можем считать любое количество дополнительных символов – если попадется символ, отличный от цифры, точки или Е, значит, имеем дело с целым числом (состояние 20)



## Архитектура лексического анализатора на основе диаграммы переходов

- Каждому состоянию (state) соответствует фрагмент кода, обрабатывающий его



## Архитектура лексического анализатора на основе диаграммы переходов

- Можно последовательно испытывать **диаграммы переходов для каждого токена**
  - функция `fail()` сбрасывает значение указателя `forward` для обработки новой диаграммы переходов
- Можно работать с **разными диаграммами переходов «параллельно»**, передавая очередной считанный символ им всем и выполняя соответствующий переход в каждой из диаграмм переходов
- Объединение всех диаграмм переходов в одну** – диаграмма переходов считывает символы до тех пор, пока возможные следующие состояния не оказываются исчерпаны, после этого выбирается наибольшая лексема, соответствующая некоторому шаблону

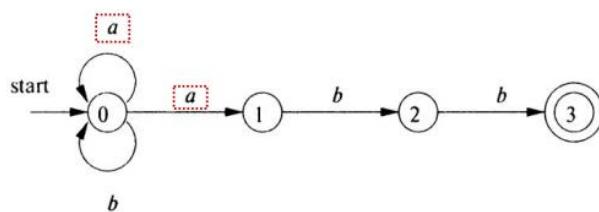
## 7. Переход от регулярных выражений к конечным автоматам. Построение НКА. Конвертация НКА в ДКА. Моделирование НКА.

### Конечные автоматы (finite automata)

- **Конечный автомат** – граф, подобный диаграмме переходов, включает:
  - входной алфавит  $\Sigma$  (конечное множество входных символов)
  - множество внутренних состояний  $S$
  - начальное состояние  $s_0$
  - множество конечных состояний  $F$
  - функцию переходов  $t(state, a) \rightarrow state$
- Конечные автоматы являются **распознавателями** (recognizer), отвечают "да" или "нет" для каждой возможной входной строки
- **Недетерминированные конечные автоматы** (НКА, nondeterministic finiteautomata – NFA) не имеют ограничений на свои дуги
  - символ может быть меткой нескольких дуг, исходящих из одного и того же состояния
  - одна из возможных меток – пустая строка  $\epsilon$
- **Детерминированные конечные автоматы** (ДКА, deterministic finiteautomata – DFA) – для каждого состояния и каждого символа входного алфавита имеют ровно одну дугу с указанным символом, покидающим это состояние
- Как детерминированные, так и недетерминированные конечные автоматы способны распознавать одни и те же языки – регулярные языки (regular language), которые могут быть описаны регулярными выражениями

### Недетерминированные конечные автоматы (НКА, NFA)

- **Недетерминированный конечный автомат (НКА):**
  1. Множество состояний  $S$
  2. Множество входных символов  $\Sigma$  (входной алфавит), не включает пустую строку  $\epsilon$
  3. Функция переходов – для каждого состояния и каждого символа из  $\Sigma \cup \{\epsilon\}$  дает множество следующих состояний (next state)
  4. Стартовое состояние  $s_0$  из  $S$
  5. Множество допускающих (конечных) состояний  $F$ , являющееся подмножеством  $S$

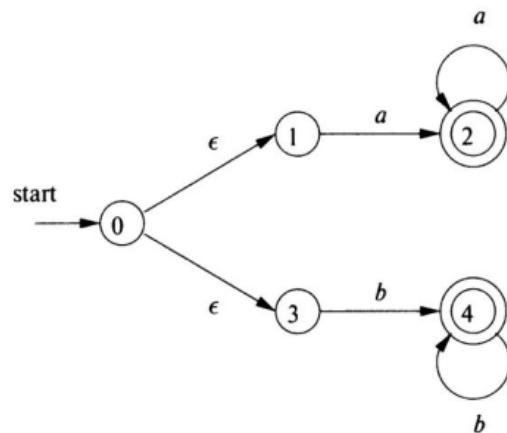


Состояние	a	b	$\epsilon$
0	{0, 1}	{0}	$\emptyset$
1	$\emptyset$	{2}	$\emptyset$
2	$\emptyset$	{3}	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Таблица переходов

Недетерминированный конечный автомат  
распознающий язык регулярного выражения  $(a \mid b)^* abb$  – строки из а и б, заканчивающиеся подстрокой abb

## Недетерминированные конечные автоматы (НКА, NFA)



НКА, принимающий  $aa^* | bb^*$

## Детерминированные конечные автоматы (ДКА, DFA)

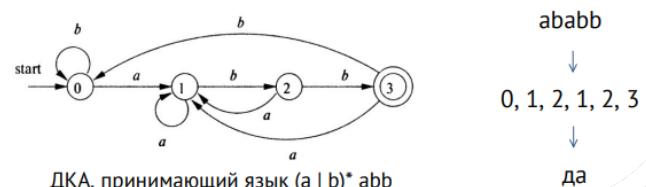
- Детерминированный конечный автомат (ДКА) – частный случай НКА:
  1. Отсутствуют переходы для входа  $\epsilon$
  2. Для каждого состояния  $s$  и входного символа  $a$  имеется ровно одна дуга, выходящая из  $s$  и помеченная  $a$
- ДКА является конкретным алгоритмом распознавания строк
- Любое регулярное выражение и каждый НКА могут быть преобразованы в ДКА, принимающий тот же язык
- При построении лексического анализатора реализуется (симулируется, моделируется) детерминированный конечный автомат

### Алгоритм моделирования ДКА (simulating a DFA)

- Вход: входная строка  $x$ , завершенная символом конца файла eof; детерминированный конечный автомат  $D$  с начальным состоянием  $s_0$ , принимающими состояниями  $F$  и функцией переходов  $move$
- Выход: ответ «да», если  $D$  принимает (распознает)  $x$ , и «нет» в противном случае

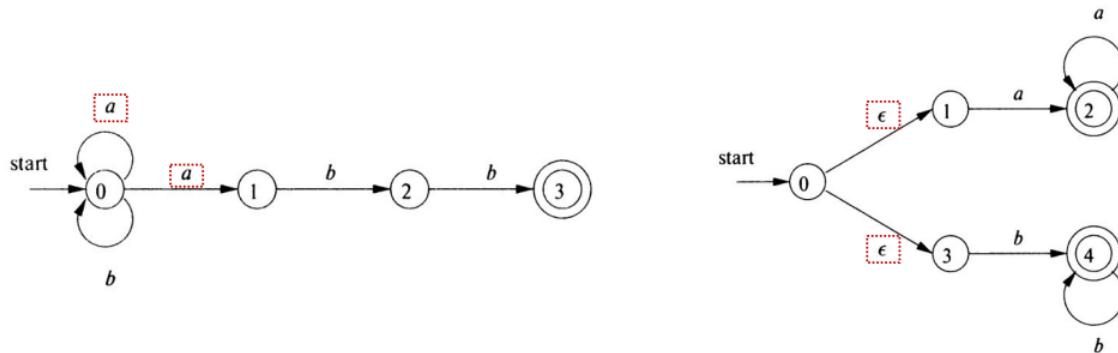
```

s = s0
c = nextChar()
while (c != eof) {
  s = move(s, c);
  c = nextChar();
}
if (s in F) return "да"
else return "нет"
  
```



## Переход от регулярных выражений к конечным автоматам

- Регулярное выражение представляет собой способ описания лексических анализаторов
- Реализация разбора регулярного выражения требует моделирования ДКА или, возможно, моделирования НКА
- При работе с НКА может требоваться делать выбор перехода для входного символа или для  $\epsilon$ , моделирование НКА существенно сложнее, чем моделирование ДКА
- Важной является задача конвертации НКА в ДКА, который принимает тот же язык



## Переход от регулярных выражений к конечным автоматам

- Генераторы лексических анализаторов и системы обработки строк часто начинают работу с регулярного выражения
- Возможные варианты реализации – преобразовывать регулярные выражения в ДКА или в НКА

АВТОМАТ	НАЧАЛЬНОЕ ПОСТРОЕНИЕ	РАБОТА НАД СТРОКОЙ
НКА	$O( r )$	$O( r  \times  x )$
ДКА: типичный случай	$O( r ^3)$	$O( x )$
ДКА: наихудший случай	$O( r ^2 2^{ r })$	$O( x )$

Вычислительная сложность начального построения и обработки одной строки  
х различными методами распознавания языка регулярных выражений  
( $|r|$  – число состояний,  $|x|$  – длина входной строки)

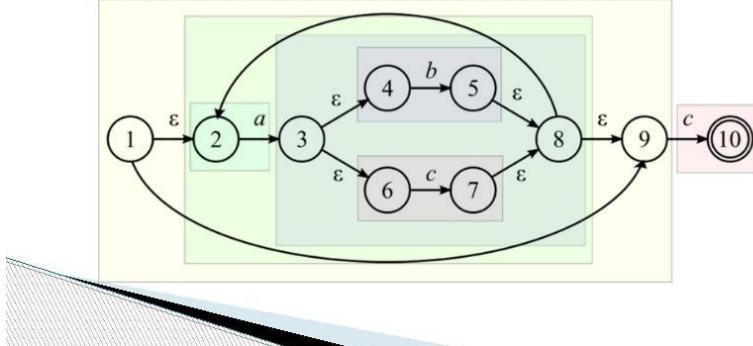
- Если доминирует время обработки одной строки, как в случае построения лексического анализатора, очевидно, что следует предпочесть ДКА
- В программах наподобие gperf, в которых автомат работает только с одной строкой, обычно предпочтительнее использовать НКА
- Пока  $|x|$  не становится порядка  $|r|^3$ , нет смысла переходить к ДКА

## Пример

- Рассмотрим регулярное выражение:

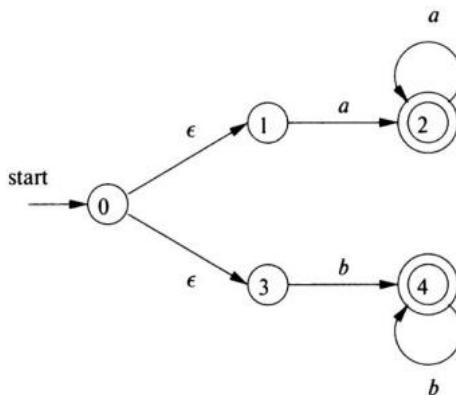
$$(a(b|c))^*c$$

- Построим соответствующий НКА:



20

## Недетерминированные конечные автоматы (НКА, NFA)



НКА, принимающий  $aa^* | bb^*$

Смотри для 7 вопроса все необязательные картинки!!!

Четкого ответа на «Построение НКА. Конвертация НКА в ДКА. Моделирование НКА.» не найдешь. Поэтому смотри на все картинки 7 вопроса и включай лампочку в тыкве.

Для особо одаренных:

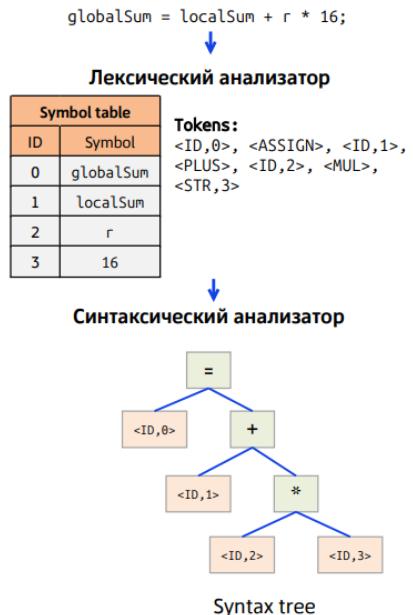
[https://kovriguineda.ucoz.ru/TK/konechnye\\_avtomaty.pdf](https://kovriguineda.ucoz.ru/TK/konechnye_avtomaty.pdf)

[https://neerc.ifmo.ru/wiki/index.php?title=%D0%9F%D0%BE%D1%81%D1%82%D1%80%D0%BE%D0%B5%D0%BD%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5%D0%BD%D0%BD%D0%BE%D0%B3%D0%BE %D0%94%D0%9A%D0%90, %D0%BD%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC %D0%A2%D0%BE%D0%BC%D0%BF%D1%81%D0%BE%D0%BD%D0%BD%D0%BD](https://neerc.ifmo.ru/wiki/index.php?title=%D0%9F%D0%BE%D1%81%D1%82%D1%80%D0%BE%D0%B5%D0%BD%D0%B8%D0%B5 %D0%BF%D0%BE %D0%9D%D0%9A%D0%90 %D1%8D%D0%BA%D0%B2%D0%B8%D0%B2%D0%BD%D0%BB%D0%B5%D0%BD%D1%82%D0%BD%D0%BE%D0%B3%D0%BE %D0%94%D0%9A%D0%90, %D0%BD%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC %D0%A2%D0%BE%D0%BC%D0%BF%D1%81%D0%BE%D0%BD%D0%BD)

## 8. Синтаксический анализ. Методы разбора. Устранение неоднозначности грамматики. Левая факторизация.

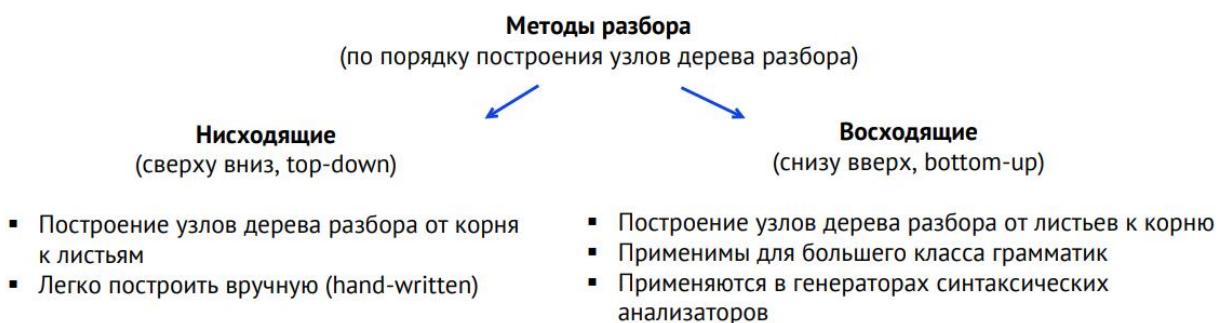
### Синтаксический анализ

- **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



### Разбор (parsing)

- Для любой контекстно-свободной грамматики существует анализатор, который требует для разбора строки из  $n$  терминалов время, не превышающее  $O(n^3)$
- Для разбора почти всех встречающихся на практике языков программирования можно построить алгоритм с линейным временем разбора  $O(n)$
- **Основные типы синтаксических анализаторов:**
  - **универсальные**: алгоритм Кока-Янгера-Касами (Cocke-Younger-Kasami), алгоритм Эрли (Earley), редко используются на практике из-за низкой эффективности
  - **восходящие** (bottom-up)
  - **нисходящие** (top-down)



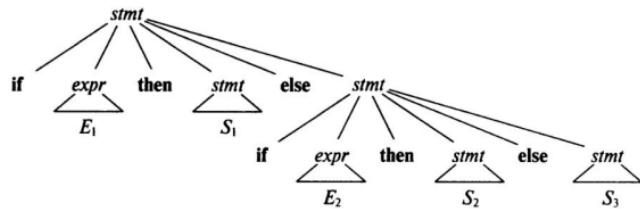
## Устранение неоднозначности грамматики

- Устраним неоднозначность из следующей грамматики с "висящим else"

$$\begin{aligned}stmt &\rightarrow \text{if } expr \text{ then } stmt \\&\quad | \quad \text{if } expr \text{ then } stmt \text{ else } stmt \\&\quad | \quad \text{other}\end{aligned}$$

- Строка:** if E1 then S1 else if E2 then S2 else S3

- Дерево разбора:



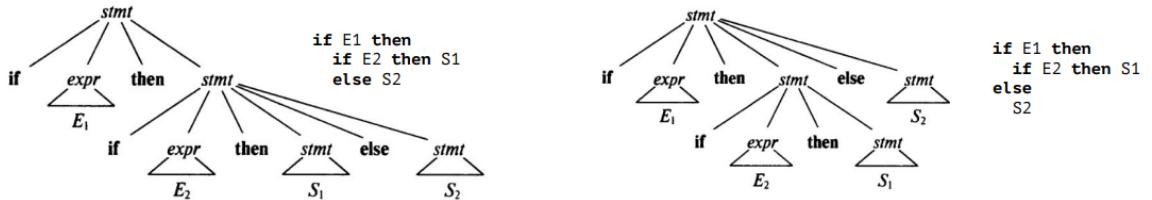
## Устранение неоднозначности грамматики

- Устраним неоднозначность из следующей грамматики с "висящим else"

$$\begin{aligned}stmt &\rightarrow \text{if } expr \text{ then } stmt \\&\quad | \quad \text{if } expr \text{ then } stmt \text{ else } stmt \\&\quad | \quad \text{other}\end{aligned}$$

- Строка:** if E1 then if E2 then S1 else S2

- Два дерева разбора – неоднозначность:



- В языках программирования с if-then-else такого вида предпочтительно первое дерево разбора
- Общее правило: «сопоставить каждое else ближайшему незанятыму then»

В помощь:

[https://translated.turbopages.org/proxy\\_u/en-ru.ru.8d6214fd-6645ee8f-52bdf68c-74722d776562/https/www.geeksforgeeks.org/ambiguous-grammar/](https://translated.turbopages.org/proxy_u/en-ru.ru.8d6214fd-6645ee8f-52bdf68c-74722d776562/https/www.geeksforgeeks.org/ambiguous-grammar/)

## Левая факторизация (left factoring)

- В построенной грамматике может быть не ясно, какая из двух альтернативных продукции должна использоваться для нетерминала
- Пример:** встретив во входном потоке if, мы не можем выбрать ни одну из продукции, нужны следующие символы потока

$$\begin{array}{lcl} \text{stmt} & \rightarrow & \text{if } \text{expr} \text{ then stmt else stmt} \\ & | & \text{if } \text{expr} \text{ then stmt} \end{array}$$

- Левая факторизация (left factoring)** – преобразование грамматики в пригодную для предиктивного, или нисходящего, синтаксического анализа

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \quad \longrightarrow \quad \begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \end{array}$$

## Левая факторизация (left factoring)

- Алгоритм 4.10. Левая факторизация грамматики**

- Вход: грамматика G.

- Выход: эквивалентная левофакторизованная грамматика.

- Для каждого нетерминала A находим самый длинный префикс  $\alpha$ , общий для двух или большего числа альтернатив
- Если  $\alpha \neq \epsilon$ , имеется нетривиальный общий префикс, заменим все продукции

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma,$$

где  $\gamma$  представляет все альтернативы, не начинающиеся с  $\alpha$ , продукциями

$$A \rightarrow \alpha A' \mid \gamma$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

- Выполняем это преобразование до тех пор, пока никакие две альтернативы нетерминала не будут иметь общий префикс

## Левая факторизация (left factoring)

- Алгоритм 4.10. Левая факторизация грамматики

- Вход: грамматика G.

- Выход: эквивалентная левофакторизованная грамматика.

1. Для каждого нетерминала  $A$  находим самый длинный префикс  $\alpha$ , общий для двух или большего числа альтернатив
2. Если  $\alpha \neq \epsilon$ , имеется нетривиальный общий префикс, заменим все продукции

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n | \gamma,$$

где  $\gamma$  представляет все альтернативы, не начинающиеся с  $\alpha$ , продукциями

$$A \rightarrow \alpha A' | \gamma$$

$$A' \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

3. Выполняем это преобразование до тех пор, пока никакие две альтернативы нетерминала не будут иметь общий префикс

$\begin{array}{l} \text{if } E \text{ then} \\ \quad S \\ \text{else} \end{array}$	$S \rightarrow i EtS   i EtSeS   a$ $E \rightarrow b$	—————>	$S \rightarrow i EtSS'   a$ $S' \rightarrow eS   \epsilon$ $E \rightarrow b$
--	--	--------	--

## 9. Нисходящий синтаксический анализ. LL(1)-грамматики.

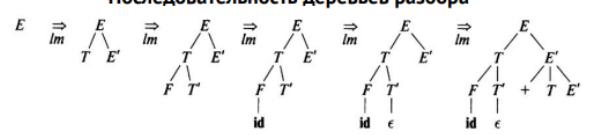
### Предиктивный синтаксический анализ, управляемый таблицей.

### Нисходящий синтаксический анализ (top-down parsing)

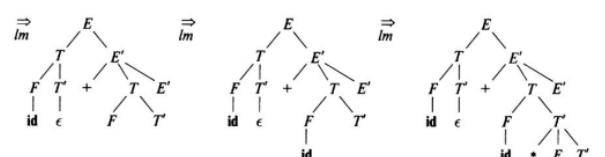
- Нисходящий синтаксический анализ (top-down parsing) – построение дерева разбора для входной строки, начиная с корня и создавая узлы дерева разбора в прямом порядке обхода (обход в глубину: корень, левый потомок, правый потомок, pre-order traversal)

$E \rightarrow TE'$ $E' \rightarrow +T E'   \epsilon$ $T \rightarrow FT'$ $T' \rightarrow *FT'   \epsilon$ $F \rightarrow (E)   id$	—————>	
---	--------	--

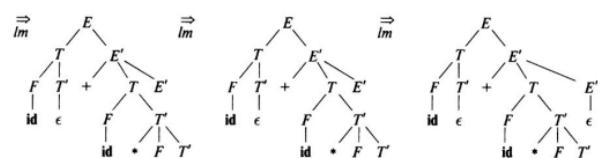
Последовательность деревьев разбора



id + id \* id



id + id \* id



id + id \* id



## LL(1)-грамматики

- Предиктивные синтаксические анализаторы (на базе рекурсивного спуска без возврата) могут быть построены для класса LL(1)-грамматик
  - LL(1): первое L – сканирование входного потока слева направо
  - LL(1): второе L – получение левого порождения (leftmost derivation)
  - LL(1) – использование на каждом шаге предпросмотра (lookahead) одного символа для принятия решения о действиях синтаксического анализатора
- В LL(1)-грамматике не может быть ни левой рекурсии, ни неоднозначности
- Определение. Грамматика  $G$  принадлежит классу LL(1) тогда и только тогда, когда для любых двух различных продукции  $A \rightarrow \alpha \mid \beta$  выполняются следующие условия:
  1. Не существует такого терминала  $a$ , для которого  $\alpha$  и  $\beta$  порождают строку, начинающуюся с  $a$
  2. Пустую строку может порождать не более чем одна из продукции  $\alpha$  или  $\beta$
  3. Если  $\beta \Rightarrow \dots \Rightarrow \epsilon$ , то  $\alpha$  не порождает ни одну строку, начинающуюся с терминала из  $\text{FOLLOW}(A)$
  4. Если  $\alpha \Rightarrow \dots \Rightarrow \epsilon$ , то  $\beta$  не порождает ни одну строку, начинающуюся с терминала из  $\text{FOLLOW}(A)$

FIRST( $\alpha$ ) и FIRST( $\beta$ ) –  
непересекающиеся  
множества

## LL(1)-грамматики

- Для LL(1)-грамматики может быть построен предиктивный синтаксический анализатор – корректная продукция для применения к нетерминалу может быть выбрана путем просмотра только текущего входного символа
- Языковые конструкции управления потоком (control flow) с их определяющими ключевыми словами обычно удовлетворяют ограничениям LL(1)

```
stmt → if (expr) stmt else stmt
      | while (expr) stmt
      | {stmt_list}
```

- Ключевые слова **if**, **while** и символ { однозначно определяют, какая из альтернатив должна быть выбрана

## Пример не LL(1)-грамматики

- В LL(1)-грамматике не может быть ни левой рекурсии, ни неоднозначности
- **Определение.** Грамматика  $G$  принадлежит классу LL(1) тогда и только тогда, когда для любых двух различных продукции  $A \rightarrow \alpha | \beta$  выполняются следующие условия:
  1. Не существует такого терминала  $a$ , для которого  $\alpha$  и  $\beta$  порождают строку, начинающуюся с  $a$
  2. Пустую строку может порождать не более чем одна из продукции  $\alpha$  или  $\beta$
  3. Если  $\beta \Rightarrow \dots \Rightarrow \epsilon$ , то  $\alpha$  не порождает ни одну строку, начинающуюся с терминала из FOLLOW( $A$ )
  4. Если  $\alpha \Rightarrow \dots \Rightarrow \epsilon$ , то  $\beta$  не порождает ни одну строку, начинающуюся с терминала из FOLLOW( $A$ )

**if E then  
S  
else  
S**

$$\begin{array}{l} S \rightarrow i E t S' | a \\ S' \rightarrow e S | \epsilon \\ E \rightarrow b \end{array}$$

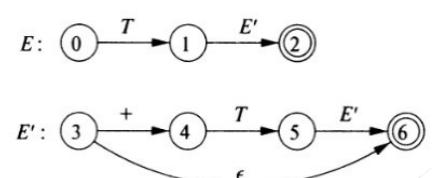
- **Грамматика неоднозначна**
- Неоднозначность проявляется в выборе продукции при встрече в потоке  $e$  (**else**)

## Диаграмма переходов на основе грамматики

- **Диаграммы переходов** (transition diagram) полезны для визуализации предиктивных синтаксических анализаторов
- **Построение диаграммы переходов на основе грамматики:**
  1. Удалить левую рекурсию
  2. Выполнить левую факторизацию грамматики
  3. Для каждого нетерминала  $A$ :
    - A. Создать начальное и конечное состояния
    - B. Для каждой продукции  $A \rightarrow X_1 X_2 \dots X_k$  создать путь из начального в конечное состояние с дугами, помеченными  $X_1, X_2, \dots, X_k$  (или  $\epsilon$ , если  $A \rightarrow \epsilon$ )
- По диаграмме для каждого нетерминала
- Метки ребер – терминалы (токены) или нетерминалами
- Переход для терминала – переход выполняется, если этот токен будет очередным входным символом
- Переход для нетерминала  $A$  – вызов процедуры для  $A$

$$\begin{array}{l} E \rightarrow T E' \\ E' \rightarrow + T E' | \epsilon \\ T \rightarrow F T' \\ T' \rightarrow * F T' | \epsilon \\ F \rightarrow (E) | \text{id} \end{array}$$

Диаграммы  
переходов для  
нетерминалов  $E$  и  $E'$



## Таблица предиктивного синтаксического анализа (predictive parsing table)

- Таблица предиктивного синтаксического анализа  $M[A, a]$  – продукция, при помощи которой выполняется разворачивание нетерминала  $A$  ( $A$  – нетерминал,  $a$  – терминал или символ  $\$$ )
- Алгоритм 4.17 может быть применен для получения таблицы  $M$  к любой грамматике  $G$

**Алгоритм 4.17.** Построение таблицы предиктивного синтаксического анализа

Вход: грамматика  $G$ .

Выход: таблица синтаксического анализа  $M$ .

Метод: для каждой продукции грамматики  $A \rightarrow \alpha$  выполняем следующие действия.

1. Для каждого терминала  $a$  из  $\text{FIRST}(\alpha)$  добавляем  $A \rightarrow \alpha$  в ячейку  $M[A, a]$ .
2. Если  $\epsilon \in \text{FIRST}(\alpha)$ , то для каждого терминала  $b$  из  $\text{FOLLOW}(A)$  добавляем  $A \rightarrow \alpha$  в  $M[A, b]$ . Если  $\epsilon \in \text{FIRST}(\alpha)$  и  $\$ \in \text{FOLLOW}(A)$ , то добавляем  $A \rightarrow \alpha$  также и в  $M[A, \$]$ .

- Если после выполнения алгоритма ячейка  $M[A, a]$  осталась без продукции (пустая запись таблицы), устанавливаем ее значение равным **error**

## Таблица предиктивного синтаксического анализа (predictive parsing table)

**Алгоритм 4.17.** Построение таблицы предиктивного синтаксического анализа

Вход: грамматика  $G$ .

Выход: таблица синтаксического анализа  $M$ .

Метод: для каждой продукции грамматики  $A \rightarrow \alpha$  выполняем следующие действия.

1. Для каждого терминала  $a$  из  $\text{FIRST}(\alpha)$  добавляем  $A \rightarrow \alpha$  в ячейку  $M[A, a]$ .
2. Если  $\epsilon \in \text{FIRST}(\alpha)$ , то для каждого терминала  $b$  из  $\text{FOLLOW}(A)$  добавляем  $A \rightarrow \alpha$  в  $M[A, b]$ . Если  $\epsilon \in \text{FIRST}(\alpha)$  и  $\$ \in \text{FOLLOW}(A)$ , то добавляем  $A \rightarrow \alpha$  также и в  $M[A, \$]$ .

$$E \rightarrow T E'$$

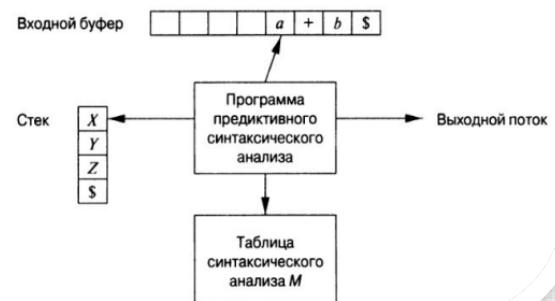
$$\text{FIRST}(T E') = \text{FIRST}(T) = \{ , \text{id} \}$$

$$\begin{array}{l} \uparrow \\ \begin{array}{l} E \rightarrow T E' \\ E' \rightarrow + T E' | \epsilon \\ T \rightarrow F T' \\ T' \rightarrow * F T' | \epsilon \\ F \rightarrow (E) | \text{id} \end{array} \end{array}$$

НЕТЕР-МИНАЛ	Входной символ					
	<b>id</b>	+	*	(	)	\$
<i>E</i>	<i>E</i> → <i>T E'</i>				<i>E</i> → <i>T E'</i>	
<i>E'</i>		<i>E'</i> → + <i>T E'</i>   $\epsilon$				
<i>T</i>			<i>E'</i> → + <i>T E'</i>			
<i>T'</i>				<i>T</i> → <i>F T'</i>		
<i>F</i>					<i>T</i> → <i>F T'</i>	
	<i>F</i> → <b>id</b>		<i>T'</i> → $\epsilon$	<i>T'</i> → * <i>F T'</i>		<i>F</i> → ( <i>E</i> )
					<i>E'</i> → $\epsilon$	<i>E'</i> → $\epsilon$
					<i>T'</i> → $\epsilon$	<i>T'</i> → $\epsilon$

## Нерекурсивный предиктивный синтаксический анализ

- Нерекурсивный предиктивный синтаксический анализатор: явное использование стека (структура данных) и таблицы синтаксического анализа
- Синтаксический анализатор имитирует левое порождение
- Синтаксический анализатор рассматривает символ на вершине стека  $X$  и текущий входной символ  $a$
- Если  $X$  является нетерминалом, синтаксический анализатор выбирает  $X$ -продукцию в соответствии с записью  $M[X, a]$  таблицы синтаксического анализа (может выполняться дополнительный код – построения узла дерева разбора)
- Если  $X$  является терминалом  $B$  – проверяется соответствие между терминалом  $X$  и текущим входным символом  $a$
- Поведение синтаксического анализатора может быть описано в терминах его **конфигураций** (configuration), которые дают содержимое стека и оставшийся входной поток



## Предиктивный синтаксический анализ, управляемый таблицей (table-driven predictive parsing)

Вход: строка  $w$  и таблица синтаксического анализа  $M$  для грамматики  $G$ .

Выход: если  $w \in L(G)$  – левое порождение  $w$ ; в противном случае – сообщение об ошибке.

Устанавливаем указатель входного потока  $ip$  так, чтобы он указывал на первый символ строки  $w$ ;  
Устанавливаем  $X$  равным символу на вершине стека;  
**while** ( $X \neq \$$ ) { /\* Стек не пуст \*/

    Устанавливаем  $a$  равным символу, на который в настоящий момент указывает  $ip$   
    **if** ( $X$  равен  $a$ ) {  
        Снимаем символ со стека и перемещаем  $ip$  к следующему символу строки;  
    }  
    **else if** ( $X$  – терминал) **error**();  
    **else if** ( $M[X, a]$  – запись об ошибке) **error**();  
    **else if** ( $M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$ ) {  
        Выводим продукцию  $X \rightarrow Y_1 Y_2 \dots Y_k$ ;  
        Снимаем символ со стека;  
        Помещаем в стек  $Y_k, Y_{k-1}, \dots, Y_1; Y_1$  помещается на вершину стека;  
    }  
    Устанавливаем  $X$  равным символу на вершине стека;  
}

### Начальное состояние

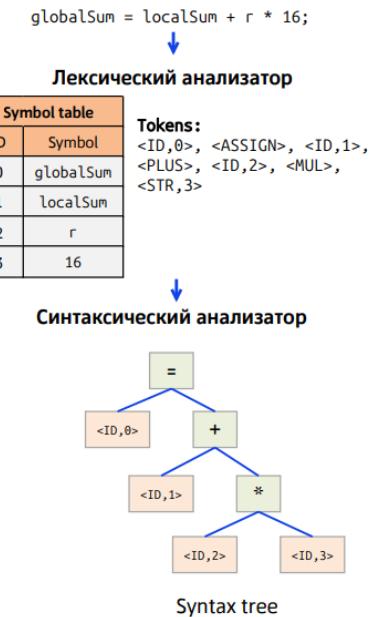
- входная строка:  $w\$$
- стек: стартовый символом  $S$  грамматики



## 10. Синтаксический анализ. Восходящий синтаксический анализ. Свертки. Обрезка основ. Конфликты в процессе ПС-анализа. Простой LR-анализатор. Алгоритм LR-анализа.

### 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



### Восходящий синтаксический анализ (bottom-up)

- Восходящий синтаксический анализ соответствует построению дерева разбора для входной строки, начиная с листьев (снизу) и идя по направлению к корню (вверх)

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$



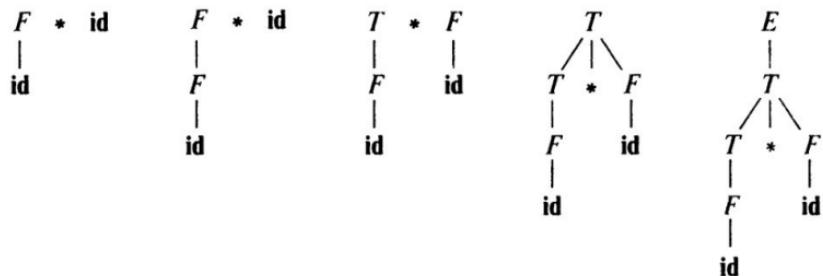
$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \text{id} \Rightarrow F * \text{id} \Rightarrow \text{id} * \text{id}$$

## Свертки (reductions)

- Восходящий синтаксический анализ – процесс «**свертки**» (reducing) строки  $w$  к стартовому символу грамматики
- На каждом шаге **свертки** (reduction) определенная подстрока, соответствующая телу продукции, заменяется нетерминалом из заголовка этой продукции
- Ключевой вопрос в процессе восходящего синтаксического анализа – когда выполнять свертку и какую продукцию применять

**id \* id**

Свертки: **id \* id, F \* id, T \* id, T \* F, T, E**



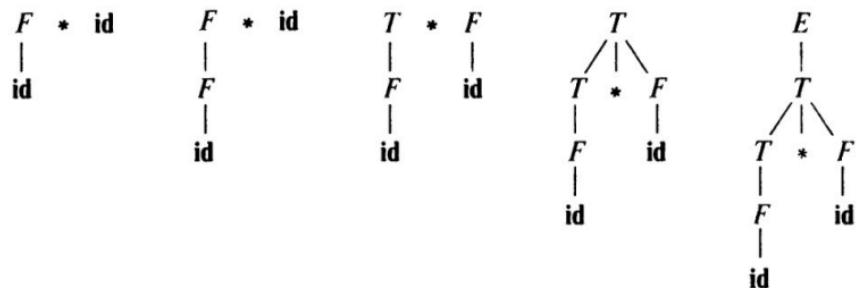
## Свертки (reductions)

- **Свертка** – шаг, обратный порождению (выводу из корня дерева разбора)
- Цель восходящего синтаксического анализа – построение порождения в обратном порядке

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{id} \end{array}$$

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \text{id} \Rightarrow F * \text{id} \Rightarrow \text{id} * \text{id}$$

Свертки: **id \* id, F \* id, T \* id, T \* F, T, E**



## Обрезка основ (handle pruning)

- **Восходящий синтаксический анализ** в процессе сканирования входного потока слева направо строит правое порождение в обратном порядке
- **Правосторонний вывод** (rightmost derivation) – вывод слова, в котором каждая последующая строка получена из предыдущей путем замены по одному из правил (продукций) самого правого встречающегося в строке нетерминала
- **Основа** (дескриптор, handle) – подстрока, которая соответствует телу продукции и свертка которой представляет собой один шаг правого порождения в обратном порядке

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{id} \end{array} \quad \text{Разбор строки: } \text{id} * \text{id}$$

ПРАВАЯ СЕНТЕНЦИАЛЬНАЯ ФОРМА	ОСНОВА	СВОРАЧИВАЮЩАЯ ПРОДУКЦИЯ
$\text{id}_1 * \text{id}_2$	$\text{id}_1$	$F \rightarrow \text{id}$
$F * \text{id}_2$	$F$	$T \rightarrow F$
$T * \text{id}_2$	$\text{id}_2$	$F \rightarrow \text{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

## Обрезка основ (handle pruning)

- **Обращенное правое порождение** (rightmost derivation, RM) может быть получено посредством «обрезки основ»
- Мы начинаем процесс со строки терминалов, которую хотим проанализировать
- Если  $w$  – строка грамматики, то пусть  $w = \gamma_n$ , где  $\gamma_n$  –  $n$ -я правосентенциальная форма еще неизвестного правого порождения

$$S = \gamma_0 \xrightarrow{rm} \gamma_1 \xrightarrow{rm} \gamma_2 \xrightarrow{rm} \cdots \xrightarrow{rm} \gamma_{n-1} \xrightarrow{rm} \gamma_n = w.$$

- Для воссоздания порождения в обратном порядке мы находим основу  $\gamma_n$  в  $\beta_n$  и заменяем ее левой частью продукции  $A_n \rightarrow \beta_n$  для получения предыдущей правосентенциальной формы  $\gamma_{n-1}$
- Затем мы повторяем процесс: находим в  $\gamma_{n-1}$  основу  $\beta_{n-1}$  и свертываем ее для получения правосентенциальной формы  $\gamma_{n-2}$ , ...
- Если после очередного шага правосентенциальная форма содержит только стартовый символ  $S$  мы прекращаем процесс и сообщаем об успешном завершении анализа
- Каким образом искать основы?

## Конфликты в процессе ПС-анализа (shift/reduce, reduce/reduce conflicts)

- Существуют контекстно-свободные грамматики, для которых восходящий ПС-анализ неприменим
- При работе с такой грамматикой ПС-анализатор может достичь конфигурации, в которой не может принять решение о том, следует ли выполнить **перенос или свертку** (конфликт «перенос-свертка», shift/reduce conflict) либо какое именно из нескольких приведений должно быть выполнено (конфликт «свертка-свертка», reduce/reduce conflict)
- Неоднозначная грамматика с «висящим else»

```
stmt → if expr then stmt
      | if expr then stmt else stmt
      | other
```

- ПС-анализатор находится в конфигурации

СТЕК       $\downarrow^{top}$   
... if expr then stmt      Вход  
                else ... \$

- Анализатор не может определить является ли **if expr then stmt** основой, без учета того, что находится ниже в стеке
- Конфликт «перенос/свертка» (shift/reduce conflict)

## Конфликты в процессе ПС-анализа (shift/reduce, reduce/reduce conflicts)

- Имеется язык, в котором вызываются процедуры по именам с параметрами, заключенными в скобки, и что тот же синтаксис используется и для работы с массивами

(1)	stmt	→	id ( parameter_list )
(2)	stmt	→	expr := expr
(3)	parameter_list	→	parameter_list , parameter
(4)	parameter_list	→	parameter
(5)	parameter	→	id
(6)	expr	→	id ( expr_list )
(7)	expr	→	id
(8)	expr_list	→	expr_list , expr
(9)	expr_list	→	expr

- Входная строка: p(i, j)
- Поток токенов: id (id, id)
- После переноса первых трех токенов в стек ПС-анализатор окажется в конфигурации:

СТЕК	Вход
... id ( id	, id ) ...
- В какую продукцию свернуть токен id на вершине стека?
- Правильный выбор:
  - ✓ продукцией (5), если p – процедура
  - ✓ продукцией (7), если p – массив

## Конфликты в процессе ПС-анализа (shift/reduce, reduce/reduce conflicts)

- Имеется язык, в котором вызываются процедуры по именам с параметрами, заключенными в скобки, и что тот же синтаксис используется и для работы с массивами

(1)	$stmt \rightarrow id ( parameter\_list )$	▪ Входная строка: $p(i, j)$
(2)	$stmt \rightarrow expr := expr$	▪ Поток токенов: $id (id, id)$
(3)	$parameter\_list \rightarrow parameter\_list , parameter$	▪ Один из вариантов разрешения конфликта – замена токена <b>id</b> в продукции (1) на <b>procid</b>
(4)	$parameter\_list \rightarrow parameter$	
(5)	$parameter \rightarrow id$	СТЕК    ВХОД
(6)	$expr \rightarrow id ( expr\_list )$	… procid ( id , id ) …
(7)	$expr \rightarrow id$	
(8)	$expr\_list \rightarrow expr\_list , expr$	▪ Выбор определяется третьим от вершины символом в стеке, который не участвует в свертке
(9)	$expr\_list \rightarrow expr$	▪ Для управления разбором ПС-анализ
		▪ может использовать информацию «из глубин» стека

## Введение в LR-анализ: простой LR (simple LR)

- LR-анализаторы – наиболее распространенный тип восходящих синтаксических анализаторов
- LR( $k$ ) parser:**
  - L – сканирование входного потока слева направо
  - R – построение правого порождения в обратном порядке (rightmost derivation in reverse)
  - $k$  – количество предпросматриваемых символов входного потока, необходимое для принятия решения
- Практический интерес:**
  - LR(0) – решение о действиях принимается только на основании содержимого стека, символы входной строки не учитываются
  - LR(1) – решение о действиях принимается на основании содержимого стека и одного символа предпросмотра входной строки
- LR-анализаторы управляются таблицами наподобие нерекурсивных LL-анализаторов
- Чтобы грамматика была LR-грамматикой, достаточно, чтобы синтаксический анализатор, работающий слева направо методом переноса/свертки, был способен распознавать основы правосентенциальных форм при их появлении на вершине стека

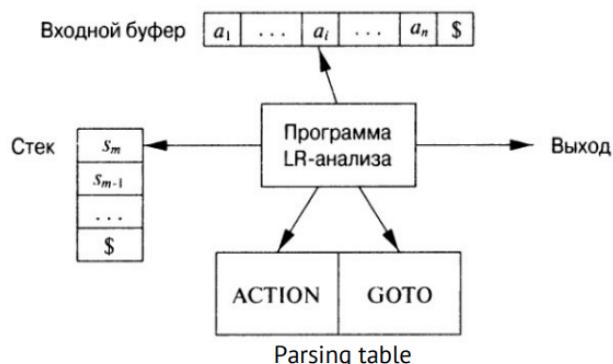
## Введение в LR-анализ: простой LR (simple LR)

- LR-анализаторы могут быть созданы для распознавания практически всех конструкций языков программирования, для которых может быть написана контекстно-свободная грамматика
- Контекстно-свободные грамматики, не являющиеся LR-грамматиками, существуют, для типичных конструкций языков программирования их можно избежать
- Метод LR-анализа – наиболее общий метод ПС-анализа без возврата, который, кроме того, не уступает в эффективности другим, более примитивным ПС-методам
- LR-анализатор может обнаруживать синтаксические ошибки сразу же, как только это становится возможным при сканировании входного потока
- Класс грамматик, которые могут быть проанализированы LR-методами – надмножество класса грамматик, которые могут быть проанализированы с использованием предиктивных или LL-методов
- В случае грамматик, принадлежащих классу LR( $k$ ), мы должны быть способны распознать правую часть продукции в порожденной ею правосентенциальной форме с дополнительным предпросмотром к входных символов – это требование существенно мягче требования для LL( $k$ )-грамматик, в которых мы должны быть способны распознать продукцию по первым  $k$  символам порождения ее тела
- LR-грамматики могут описать существенно больше языков, чем LL-грамматики
- Основной недостаток LR-метода – построение LR-анализатора для грамматики типичного языка программирования вручную требует очень большого объема работы
- Для решения этой задачи нужен специализированный инструмент – генератор LR-анализатора (Bison/Yacc, )

[https://translated.turbopages.org/proxy\\_u/en-ru.ru.d308aa95-6645f470-1574d42b-74722d776562/https://en.wikipedia.org/wiki/Simple\\_LR\\_parser](https://translated.turbopages.org/proxy_u/en-ru.ru.d308aa95-6645f470-1574d42b-74722d776562/https://en.wikipedia.org/wiki/Simple_LR_parser)

<https://wiki2.org/LR-%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%82%D0%BE%D1%80>

## Алгоритм LR-анализа (LR-Parsing Algorithm)



- **Входной буфер:** анализатор читает по одному символу предпросмотра
- **Стек:** В случае методов SLR, LR, LALR стек хранит состояния LR(0)-автомата
- **Таблица синтаксического анализа (parsing table):** функции действий синтаксического анализа ACTION и функции переходов GOTO

## Структура таблицы LR-анализа (LR parsing table)

- **Функция ACTION[i, a]:**
  - $i$  – состояние и терминал  $a$  (или маркер \$ конца входной строки)
- **Значения ACTION[i, a]:**
  - **Перенос**  $j$  (shift): перено входного символа  $a$  в стек, но для представления  $a$  используется состояние  $j$
  - **Свертка**  $A \rightarrow \beta$  (reduce): свёртка  $\beta$  на вершине стека в заголовок  $A$
  - **Принятие** (accept): анализатор принимает входную строку и завершает анализ
  - **Ошибка** (error): синтаксический анализатор обнаруживает ошибку во входной строке и предпринимает корректирующее действие
- **Расширенная функция GOTO**, определенная на множествах пунктов, распространяется на состояния: если  $GOTO[I_i, A] = I_j$ , то GOTO отображает также состояние  $i$  и нетерминал  $A$  на состояние  $j$

## Алгоритм LR-анализа

- **Вход:** входная строка  $w$  и таблица LR-анализа с функциями ACTION и GOTO для грамматики  $G$
- **Выход:** если  $w \in L(G)$ , шаги сверток восходящего синтаксического анализа  $w$ , в противном случае – сообщение об ошибке
- **Начальное состояние:** в стеке начальное состояние  $s_0$ , а во входном буфере  $w\$$

Пусть  $a$  – первый символ  $w\$$ .

```
while(1) { /* Бесконечный цикл */
    Пусть  $s$  – состояние на вершине стека.
    if ( ACTION [s, a] = перенос t ) {
        Внести  $t$  в стек.
        Присвоить  $a$  очередной входной символ.
    } else if ( ACTION [s, a] = свертка  $A \rightarrow \beta$  ) {
        Снять  $|\beta|$  символов со стека.
        Пусть теперь на вершине стека находится состояние  $t$ .
        Внести в стек GOTO [t, A].
        Вывести продукцию  $A \rightarrow \beta$ .
    } else if ( ACTION [s, a] = принятие ) {
        break; /* Синтаксический анализ завершен */
    } else Вызов подпрограммы восстановления после ошибки.
}
```

## Пример таблицы для LR-анализа

Грамматика для выражений:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

Продукции:

$$\begin{array}{ll} (1) \quad E \rightarrow E + T & (4) \quad T \rightarrow F \\ (2) \quad E \rightarrow T & (5) \quad F \rightarrow (E) \\ (3) \quad T \rightarrow T * F & (6) \quad F \rightarrow \text{id} \end{array}$$

- si – перенос и размещение в стеке состояния i
- rj – свертка в соответствии с продукцией с номером j
- acc – принятие
- пустое поле – ошибка

Таблица синтаксического анализа (функции ACTION и GOTO)

Состояние	ACTION					GOTO			
	<b>id</b>	+	*	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4			r4	r4		
4	s5		s4				8	2	3
5		r6	r6		r6	r6			
6	s5		s4					9	3
7	s5		s4						10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

## Пример действий в ходе LR-анализа

Действия LR-анализатора для строки **id \* id + id**

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		<b>id * id + id \$</b>	shift
(2)	0 5	<b>id</b>	* id + id \$	reduce by $F \rightarrow \text{id}$
(3)	0 3	<b>F</b>	* id + id \$	reduce by $T \rightarrow F$
(4)	0 2	<b>T</b>	* id + id \$	shift
(5)	0 2 7	<b><math>T *</math></b>	<b>id + id \$</b>	shift
(6)	0 2 7 5	<b><math>T * id</math></b>	+ id \$	reduce by $F \rightarrow \text{id}$
(7)	0 2 7 10	<b><math>T * F</math></b>	+ id \$	reduce by $T \rightarrow T * F$
(8)	0 2	<b>T</b>	+ id \$	reduce by $E \rightarrow T$
(9)	0 1	<b>E</b>	+ id \$	shift
(10)	0 1 6	<b><math>E +</math></b>	<b>id \$</b>	shift
(11)	0 1 6 5	<b><math>E + id</math></b>	\$	reduce by $F \rightarrow \text{id}$
(12)	0 1 6 3	<b><math>E + F</math></b>	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	<b><math>E + T</math></b>	\$	reduce by $E \rightarrow E + T$
(14)	0 1	<b>E</b>	\$	accept

[https://translated.turbopages.org/proxy\\_u/en-ru.ru.b05c535b-6645f585-e6ae1c70-74722d776562/https/www.geeksforgeeks.org/lr-parser/](https://translated.turbopages.org/proxy_u/en-ru.ru.b05c535b-6645f585-e6ae1c70-74722d776562/https/www.geeksforgeeks.org/lr-parser/)

мб:

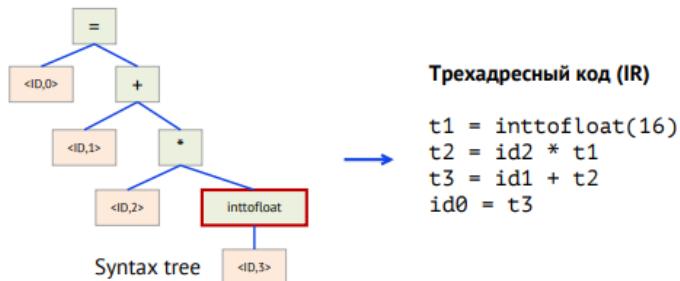
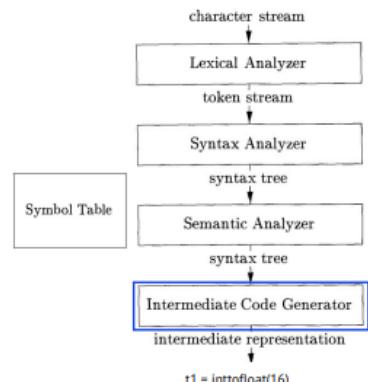
[https://neerc.ifmo.ru/wiki/index.php?title=LR\(0\)-%D1%80%D0%B0%D0%B7%D0%B1%D0%BE%D1%80](https://neerc.ifmo.ru/wiki/index.php?title=LR(0)-%D1%80%D0%B0%D0%B7%D0%B1%D0%BE%D1%80)

## 11. Генерация промежуточного кода. Трёхадресный код. SSA.

Трансляция выражений в трёхадресный код.

### Генерация кода в промежуточное представление

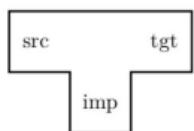
- Промежуточное представление (intermediate representation, IR) – архитектура набора команд (ISA) абстрактной вычислительной машины, в который легко транслировать синтаксическое дерево, над которым легко выполнять оптимизации и трансформации и генерировать машинный код для целевой архитектуры
- Трёхадресный код – в каждой команде 3 операнда
- Стековые и регистровые машины



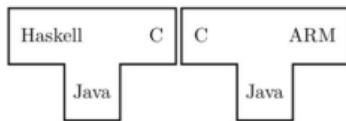
#### Трёхадресный код (IR)

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

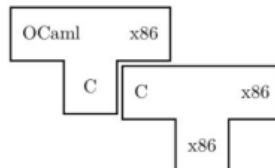
## Т-диаграммы (T-diagrams)



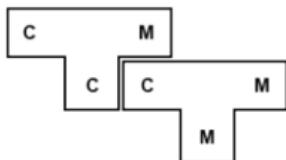
- Транслятор с языка src в язык tgt, реализованный на imp



- Транслятор с Haskell в C, реализованный на Java, и затем в ARM транслятором на Java



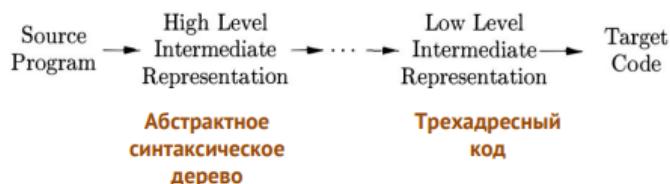
- Транслятор с OCaml в x86, реализованный на C, транслятор собирается для x86 компилятором на ассемблере x86



- **Bootstrapping (раскрутка)** – процесс создания компилятора языка  $L$ , способного скомпилировать свой код  $L$  (self-compiling compiler)
- **Bootstrap compiler** – начальная версия компилятора, создается на другом языке, доступном на целевой системе

## Генерация кода в промежуточное представление

- Синтаксис и спецификация языка определяют действия начальной стадии компилятора (scanner, parser), а детали целевой машины (архитектуры) – заключительной стадией компилятора
- Эффективное промежуточное представление позволяет построить компилятор для языка  $L$  и целевой машины  $M$  путем комбинации начальной стадии для  $L$  и заключительной – для машины
- $L \times M$  компиляторов могут быть созданы путем создания  $L$  начальных стадий  $M$  заключительных



## Генерация промежуточного кода

Любой компилятор может непосредственно генерировать машинный код из исходного. Так зачем же тогда нужна фаза генерации промежуточного кода?

Существуют различные типы машин. Таким образом, машинный код зависит от системы, а высокоуровневый исходный код — нет. Если компилятор непосредственно генерирует машинный код из исходного кода, то каждая машина нуждается в полной компиляции от фронта к бэку. Но когда компилятор генерирует промежуточный код (**промежуточное представление**), он уже может генерировать машинный код для каждой машины с его помощью, без повторения лексического анализа и парсинга для каждой машины.

Существует два основных типа промежуточных представлений:

- Высокоуровневый — более близкий к высокоуровневому языку.
- Низкоуровневый — более близкий к машинному коду.

Существует также несколько способов представления промежуточного представления.

- AST — абстрактное синтаксическое дерево (графическое).
- Постфиксная нотация.
- Трехадресный код.
- Двухадресный код.

[https://translated.turbopages.org/proxy\\_u/en-ru.ru.886321a3-6645f73d-c23d5bc3-74722d776562/https/www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/](https://translated.turbopages.org/proxy_u/en-ru.ru.886321a3-6645f73d-c23d5bc3-74722d776562/https/www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/)

## Промежуточное представление в виде статических единственных присваиваний (SSA)

- Промежуточное представление в виде статических единственных присваиваний (static single-assignment form) упрощает некоторые формы оптимизаций
- 1. Все присваивания в SSA выполняются для переменных с различными именами** (единственное присваивание)

$$\begin{array}{ll} p = a + b & p_1 = a + b \\ q = p - c & q_1 = p_1 - c \\ p = q * d & p_2 = q_1 * d \\ p = e - r & p_3 = e - p_2 \\ q = p + q & q_2 = p_3 + q_1 \end{array}$$

Трёхадресный код

SSA

## Промежуточное представление в виде статических единственных присваиваний (SSA)

- Промежуточное представление в виде статических единственных присваиваний (static single-assignment form) упрощает некоторые формы оптимизаций
- Все присваивания в SSA выполняются для переменных с различными именами (единственное присваивание)
- Одна переменная может быть определена в двух разных путях потока управления

```
if (flag)
    x = -1; # Путь 1 потока управления
else
    x = 1; # Путь 2 потока управления

y = x * a;
```

- Если применить SSA, то x в двух потоках управления будут иметь разные имена x1, x2
- Какую переменную использовать для вычисления y?

## Промежуточное представление в виде статических единственных присваиваний (SSA)

- Промежуточное представление в виде статических единственных присваиваний (static single-assignment form) упрощает некоторые формы оптимизаций
- Все присваивания в SSA выполняются для переменных с различными именами (единственное присваивание)
- 2. SSA использует для комбинации двух определений x специальную функцию – φ-функцию (phi function)

```
if (flag)
    x1 = -1; # Путь 1 потока управления
else
    x2 = 1; # Путь 2 потока управления

x3 = φ(x1, x2) # phi-function
y = x3 * a
```

- Функция φ(x1, x2) принимает значение x1, если поток управления проходит по истинной части конструкции if, и x2 – если по ложной

## Трансляция выражений в трёхадресный код

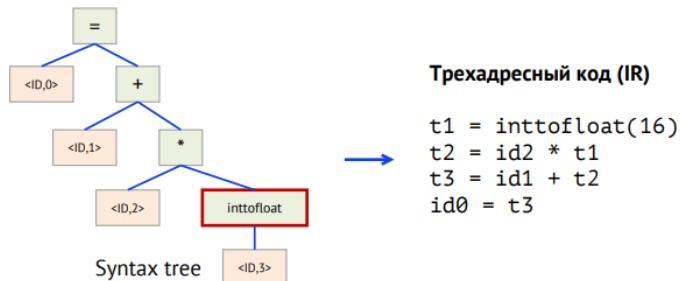
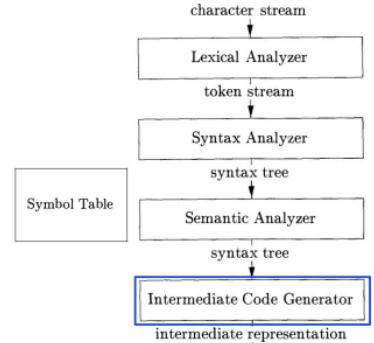
ПРОДУКЦИЯ	СЕМАНТИЧЕСКИЕ ПРАВИЛА
$S \rightarrow \mathbf{id} = E ;$	$S.code = E.code   $ $\underline{\underline{gen(top.get(\mathbf{id}.lexeme) !=' E.addr)}}$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new Temp}()$ $E.code = E_1.code    E_2.code   $ $\underline{\underline{gen(E.addr !=' E_1.addr +' E_2.addr)}}$
$- E_1$	$E.addr = \mathbf{new Temp}()$ $E.code = E_1.code   $ $\underline{\underline{gen(E.addr !=' 'minus' E_1.addr)}}$
$( E_1 )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

■ Атрибуты-коды (\*.code) могут быть очень длинными строками

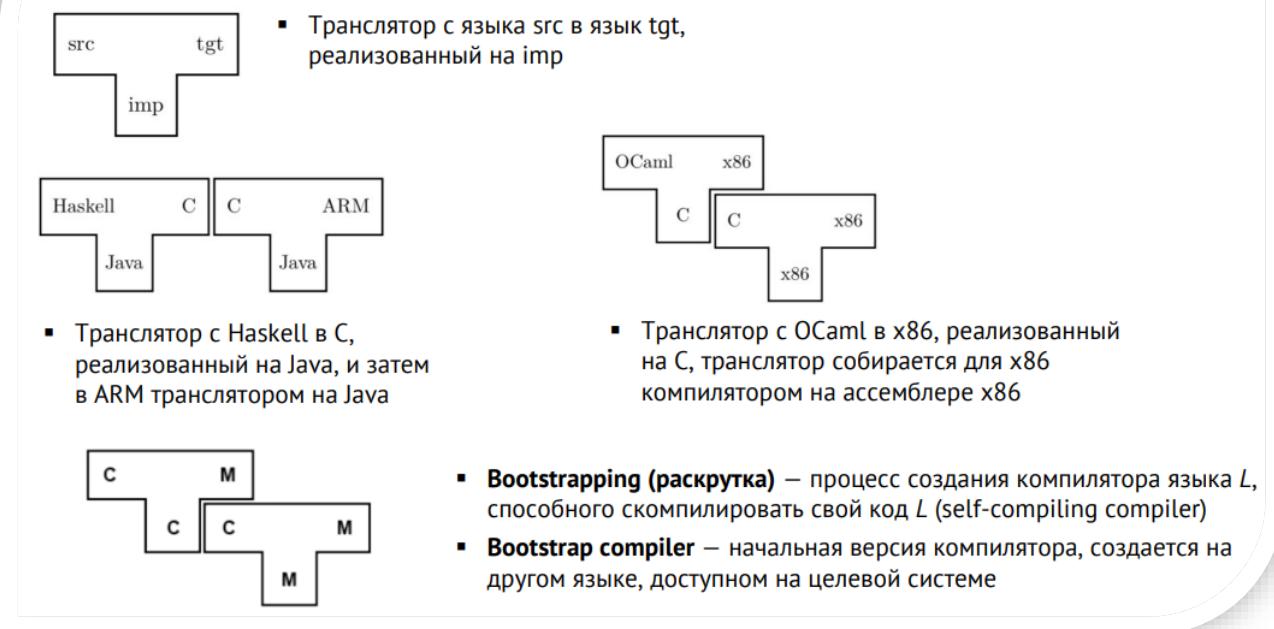
## 12. Генерация промежуточного кода. Трёхадресный код. Трансляция обращений к массиву.

### Генерация кода в промежуточное представление

- 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
- Трёхадресный код – в каждой команде 3 операнда
- Стековые и регистровые машины

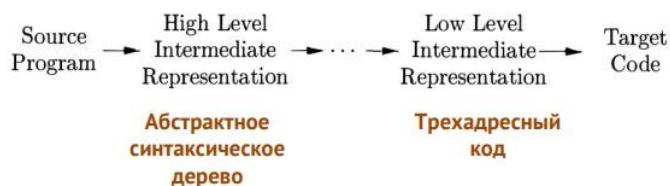


## Т-диаграммы (T-diagrams)



## Генерация кода в промежуточное представление

- Синтаксис и спецификация языка определяют действия начальной стадии компилятора (scanner, parser), а детали целевой машины (архитектуры) – заключительной стадией компилятора
- Эффективное промежуточное представление позволяет построить компилятор для языка  $L$  и целевой машины  $M$  путем комбинации начальной стадии для  $L$  и заключительной – для машины
- $L \times M$  компиляторов могут быть созданы путем создания  $L$  начальных стадий  $M$  заключительных



## Генерация промежуточного кода

Любой компилятор может непосредственно генерировать машинный код из исходного. Так зачем же тогда нужна фаза генерации промежуточного кода?

Существуют различные типы машин. Таким образом, машинный код зависит от системы, а высокоуровневый исходный код — нет. Если компилятор непосредственно генерирует машинный код из исходного кода, то каждая машина нуждается в полной компиляции от фронта к бэку. Но когда компилятор генерирует промежуточный код (**промежуточное представление**), он уже может генерировать машинный код для каждой машины с его помощью, без повторения лексического анализа и парсинга для каждой машины.

Существует два основных типа промежуточных представлений:

- Высокоуровневый — более близкий к высокоуровневому языку.
- Низкоуровневый — более близкий к машинному коду.

Существует также несколько способов представления промежуточного представления.

- AST — абстрактное синтаксическое дерево (графическое).
- Постфиксная нотация.
- Трехадресный код.
- Двухадресный код.

[https://translated.turbopages.org/proxy\\_u/en-ru.ru.886321a3-6645f73d-c23d5bc3-74722d776562/https/www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/](https://translated.turbopages.org/proxy_u/en-ru.ru.886321a3-6645f73d-c23d5bc3-74722d776562/https/www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/)

## Трансляция обращений к массиву

- Основная задача — связывание вычислений адресов из раздела элементов с грамматикой для обращения к массивам

- Пусть нетерминал  $L$  генерирует имя массива с последовательностью индексных выражений:

$$L \rightarrow L [E] | \mathbf{id} [E]$$

- $L.addr$  — смещение
- $L.array$  — имя массива
- $L.type$  — тип подмассива

```
S → id = E ; { gen( top.get(id.lexeme) != E.addr); }
| L = E ; { gen(L.addr.base '[' L.addr ')' != E.addr); }

E → E1 + E2 { E.addr = new Temp();
                gen(E.addr != E1.addr +'+' E2.addr); }

| id { E.addr = top.get(id.lexeme); }

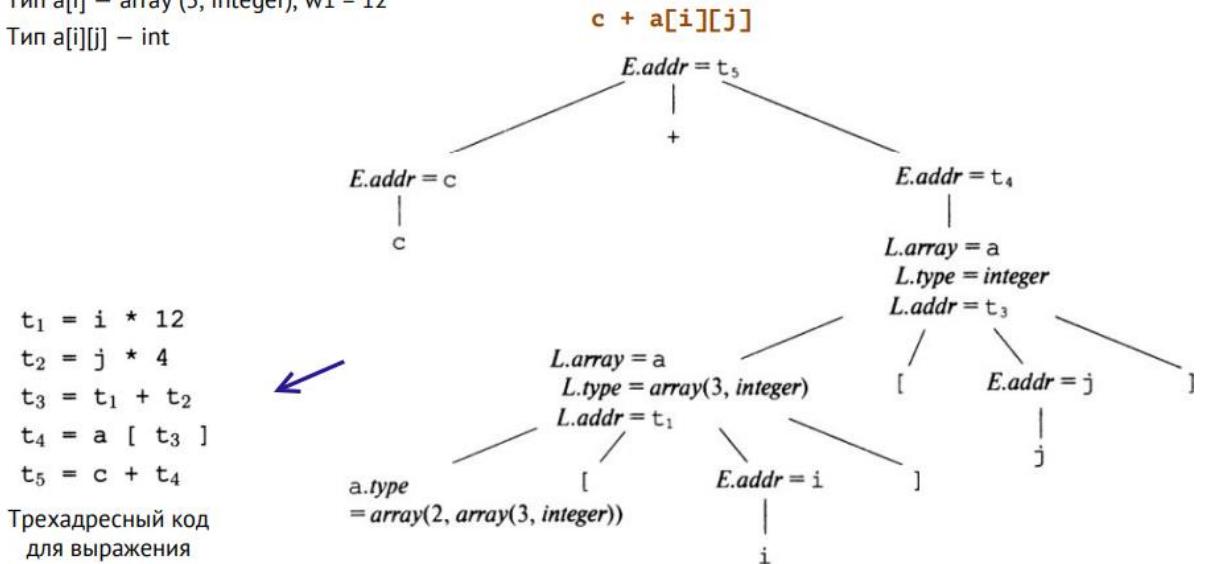
| L { E.addr = new Temp();
      gen(E.addr != L.array.base '[' L.addr ')'); }

L → id [ E ] { L.array = top.get(id.lexeme);
                  L.type = L.array.type.elem;
                  L.addr = new Temp();
                  gen(L.addr != E.addr '*' L.type.width); }

| L1 [ E ] { L.array = L1.array;
              L.type = L1.type.elem;
              t = new Temp();
              L.addr = new Temp();
              gen(t != E.addr '*' L.type.width); }
              gen(L.addr != L1.addr +'+' t); }
```

## Трансляция обращений к массиву

- Тип a – array (2, array (3, integer)), w = 24
- Тип a[i] – array (3, integer), w1 = 12
- Тип a[i][j] – int



Трансляция обращений к массиву (array indexing) - это процесс преобразования инструкций или выражений, которые обращаются к элементам массива, из высокоуровневого исходного кода в низкоуровневый машинный код или другой язык программирования.

Вот основные шаги, которые обычно включаются в процесс трансляции обращений к массиву:

1. **Определение массива:** Сначала нужно определить массив в исходном коде. Это включает в себя указание типа элементов массива, его размера и, возможно, начальное значение элементов.
2. **Разрешение индексов:** При обращении к массиву необходимо вычислить индексы элементов, к которым происходит обращение. Если индексы являются выражениями, они могут содержать переменные, константы или другие выражения.
3. **Проверка границ массива:** Перед доступом к элементу массива необходимо убедиться, что индексы находятся в допустимом диапазоне. Если индексы выходят за границы массива, может возникнуть ошибка времени выполнения.

4. Вычисление адреса элемента: После разрешения индексов необходимо вычислить адрес элемента массива в памяти. Это может включать в себя умножение индексов на размер элемента и добавление смещения относительно базового адреса массива.
5. Доступ к элементу в памяти: После вычисления адреса элемента можно получить доступ к самому элементу в памяти. Это может включать чтение значения из этого адреса или запись значения в него.
6. Генерация машинного кода или кода на целевом языке: Наконец, на основе обращения к массиву создается машинный код (если речь идет о компиляции) или код на целевом языке программирования (если это интерпретация или компиляция в высокоровневый язык).

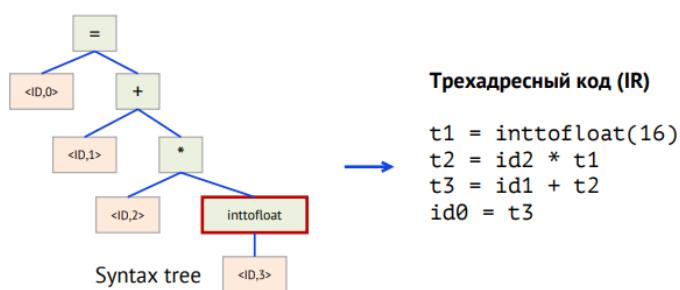
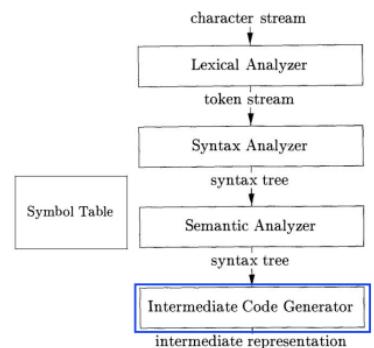
Трансляция обращений к массиву является важной частью процесса компиляции или интерпретации программы и обеспечивает корректное и эффективное выполнение операций с массивами в программе.



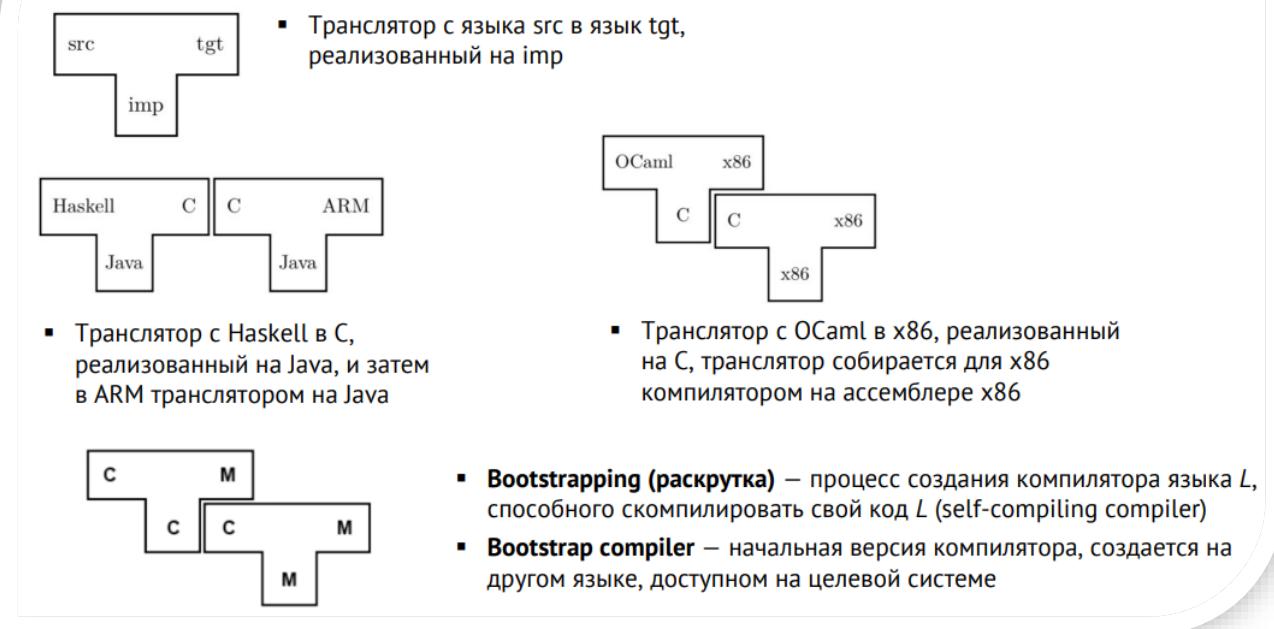
## 13. Генерация промежуточного кода. Трехадресный код. Вычисления булевых выражений по сокращенной схеме.

### Генерация кода в промежуточное представление

- 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

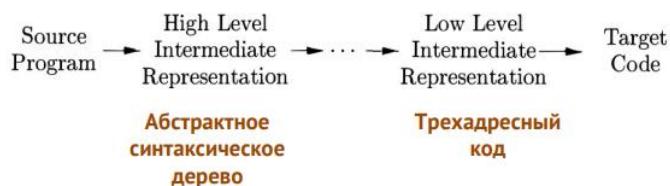


## Т-диаграммы (T-diagrams)



## Генерация кода в промежуточное представление

- Синтаксис и спецификация языка определяют действия начальной стадии компилятора (scanner, parser), а детали целевой машины (архитектуры) – заключительной стадией компилятора
- Эффективное промежуточное представление позволяет построить компилятор для языка  $L$  и целевой машины  $M$  путем комбинации начальной стадии для  $L$  и заключительной – для машины
- $L \times M$  компиляторов могут быть созданы путем создания  $L$  начальных стадий  $M$  заключительных



## Генерация промежуточного кода

Любой компилятор может непосредственно генерировать машинный код из исходного. Так зачем же тогда нужна фаза генерации промежуточного кода?

Существуют различные типы машин. Таким образом, машинный код зависит от системы, а высокоуровневый исходный код — нет. Если компилятор непосредственно генерирует машинный код из исходного кода, то каждая машина нуждается в полной компиляции от фронта к бэку. Но когда компилятор генерирует промежуточный код (**промежуточное представление**), он уже может генерировать машинный код для каждой машины с его помощью, без повторения лексического анализа и парсинга для каждой машины.

Существует два основных типа промежуточных представлений:

- Высокоуровневый — более близкий к высокоуровневому языку.
- Низкоуровневый — более близкий к машинному коду.

Существует также несколько способов представления промежуточного представления.

- AST — абстрактное синтаксическое дерево (графическое).
- Постфиксная нотация.
- Трехадресный код.
- Двухадресный код.

[https://translated.turbopages.org/proxy\\_u/en-ru.ru.886321a3-6645f73d-c23d5bc3-74722d776562/https/www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/](https://translated.turbopages.org/proxy_u/en-ru.ru.886321a3-6645f73d-c23d5bc3-74722d776562/https/www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/)

## Вычисления булевых выражений по сокращенной схеме (short-circuit code)

- При вычислениях по сокращенной схеме (short-circuit code) булевые операторы `&&`, `||` и `!` транслируются в условные переходы
- Операторы отношений (`<`, `>`, `=` и др.) в коде отсутствуют, в значение булева выражения представлено в виде позиции в последовательности команд

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```



```
if x < 100 goto L2
iffalse x > 200 goto L1
iffalse x != y goto L1
L2: x = 0
L1:
```

- Вычисление булева выражения заканчивается как только становится известен его результат, даже если не все операнды вычислены

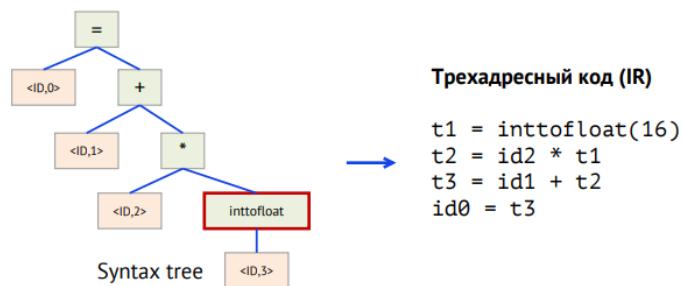
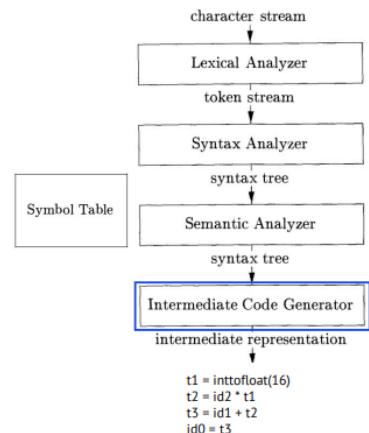
- `true || b => true`  
`b` не вычисляется
- `false && b => false`  
`b` не вычисляется

[https://en.cppreference.com/w/cpp/language/operator\\_logical](https://en.cppreference.com/w/cpp/language/operator_logical)

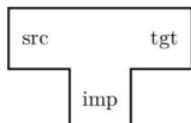
## 14. Генерация промежуточного кода. Трехадресный код. Трансляция if-then-else.

### Генерация кода в промежуточное представление

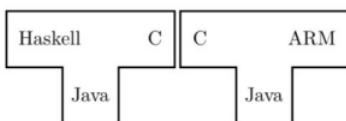
- Промежуточное представление (intermediate representation, IR) – архитектура набора команд (ISA) абстрактной вычислительной машины, в который легко транслировать синтаксическое дерево, над которым легко выполнять оптимизации и трансформации и генерировать машинный код для целевой архитектуры
- Трехадресный код – в каждой команде 3 операнда
- Стековые и регистровые машины



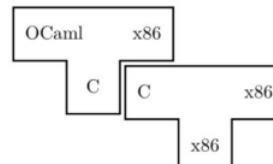
### T-диаграммы (T-diagrams)



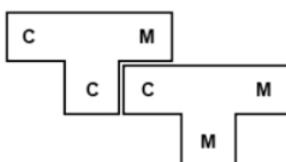
- Транслятор с языка src в язык tgt, реализованный на imp



- Транслятор с Haskell в C, реализованный на Java, и затем в ARM транслятором на Java



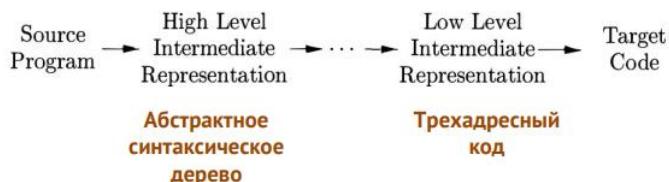
- Транслятор с OCaml в x86, реализованный на C, транслятор собирается для x86 компилятором на ассемблере x86



- **Bootstrapping (раскрутка)** – процесс создания компилятора языка  $L$ , способного скомпилировать свой код  $L$  (self-compiling compiler)
- **Bootstrap compiler** – начальная версия компилятора, создается на другом языке, доступном на целевой системе

## Генерация кода в промежуточное представление

- Синтаксис и спецификация языка определяют действия начальной стадии компилятора (scanner, parser), а детали целевой машины (архитектуры) – заключительной стадией компилятора
- Эффективное промежуточное представление позволяет построить компилятор для языка  $L$  и целевой машины  $M$  путем комбинации начальной стадии для  $L$  и заключительной – для машины
- $L \times M$  компиляторов могут быть созданы путем создания  $L$  начальных стадий  $M$  заключительных



### Генерация промежуточного кода

Любой компилятор может непосредственно генерировать машинный код из исходного. Так зачем же тогда нужна фаза генерации промежуточного кода?

Существуют различные типы машин. Таким образом, машинный код зависит от системы, а высокоуровневый исходный код – нет. Если компилятор непосредственно генерирует машинный код из исходного кода, то каждая машина нуждается в полной компиляции от фронта к бэку. Но когда компилятор генерирует промежуточный код (**промежуточное представление**), он уже может генерировать машинный код для каждой машины с его помощью, без повторения лексического анализа и парсинга для каждой машины.

Существует два основных типа промежуточных представлений:

- Высокоуровневый – более близкий к высокоуровневому языку.
- Низкоуровневый – более близкий к машинному коду.

Существует также несколько способов представления промежуточного представления.

- AST – абстрактное синтаксическое дерево (графическое).
- Постфиксная нотация.
- Трехадресный код.
- Двухадресный код.

[https://translated.turbopages.org/proxy\\_u/en-ru.ru.886321a3-6645f73d-c23d5bc3-74722d776562/https/www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/](https://translated.turbopages.org/proxy_u/en-ru.ru.886321a3-6645f73d-c23d5bc3-74722d776562/https/www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/)

## Трансляция if-then-else

$S \rightarrow \text{if } (B) S_1 \text{ else } S_2$



## Трансляция if-then-else

ПРОДУКЦИЯ	СЕМАНТИЧЕСКИЕ ПРАВИЛА
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \text{assign}$	$S.code = \text{assign}.code$
$S \rightarrow \text{if } (B) S_1 \text{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen('goto' S.next)$ $\parallel label(B.false) \parallel S_2.code$

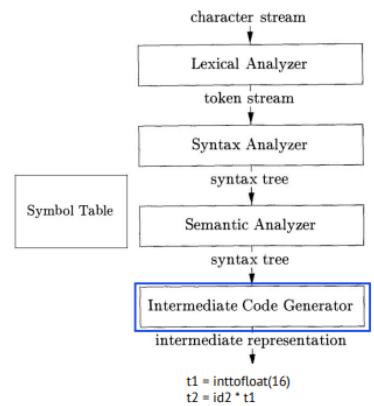
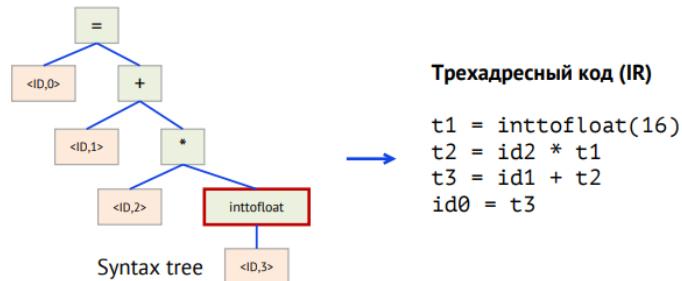
Синтаксически управляемое определение для инструкций потока управления if-then-else

<https://wasm.in/blogs/identifikacija-if-then-else.105/>

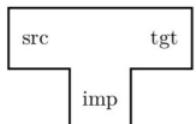
## 15. Генерация промежуточного кода. Трехадресный код. Трансляция цикла `while`

## Генерация кода в промежуточное представление

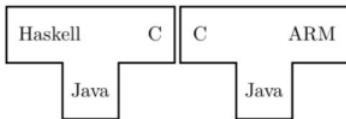
- **Промежуточное представление** (intermediate representation, IR) – архитектура набора команд (ISA) абстрактной вычислительной машины, в который легко транслировать синтаксическое дерево, над которым легко выполнять оптимизации и трансформации и генерировать машинный код для целевой архитектуры
  - Трехадресный код – в каждой команде 3 операнда
  - Стековые и регистровые машины



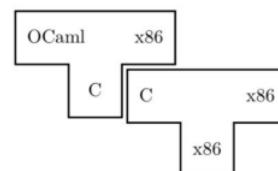
## Т-диаграммы (T-diagrams)



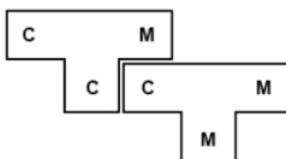
- Транслятор с языка src в язык tgt, реализованный на imp



- Транслятор с Haskell в C, реализованный на Java, и затем в ARM транслятором на Java



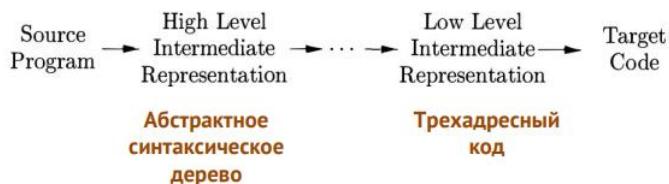
- Транслятор с OCaml в x86, реализованный на C, транслятор собирается для x86 компилятором на ассемблере x86



- **Bootstrapping (раскрутка)** – процесс создания компилятора языка  $L$ , способного скомпилировать свой код  $L$  (*self-compiling compiler*)
  - **Bootstrap compiler** – начальная версия компилятора, создается на другом языке, доступном на целевой системе

## Генерация кода в промежуточное представление

- Синтаксис и спецификация языка определяют действия начальной стадии компилятора (scanner, parser), а детали целевой машины (архитектуры) — заключительной стадией компилятора
- Эффективное промежуточное представление позволяет построить компилятор для языка  $L$  и целевой машины  $M$  путем комбинации начальной стадии для  $L$  и заключительной — для машины
- $L \times M$  компиляторов могут быть созданы путем создания  $L$  начальных стадий  $M$  заключительных



### Генерация промежуточного кода

Любой компилятор может непосредственно генерировать машинный код из исходного. Так зачем же тогда нужна фаза генерации промежуточного кода?

Существуют различные типы машин. Таким образом, машинный код зависит от системы, а высокоуровневый исходный код — нет. Если компилятор непосредственно генерирует машинный код из исходного кода, то каждая машина нуждается в полной компиляции от фронта к бэку. Но когда компилятор генерирует промежуточный код (**промежуточное представление**), он уже может генерировать машинный код для каждой машины с его помощью, без повторения лексического анализа и парсинга для каждой машины.

Существует два основных типа промежуточных представлений:

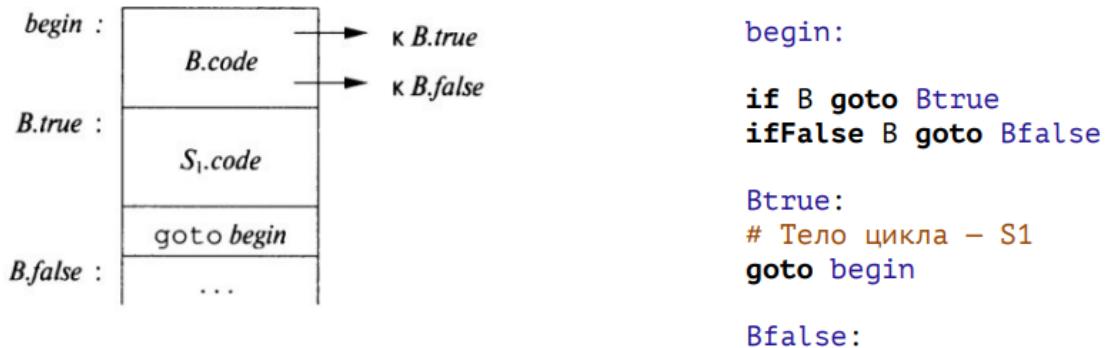
- Высокоуровневый — более близкий к высокоуровневому языку.
- Низкоуровневый — более близкий к машинному коду.

Существует также несколько способов представления промежуточного представления.

- AST — абстрактное синтаксическое дерево (графическое).
- Постфиксная нотация.
- Трехадресный код.
- Двухадресный код.

## Трансляция цикла while

$S \rightarrow \text{while } (B) S_1$



## Трансляция цикла while

ПРОДУКЦИЯ	СЕМАНТИЧЕСКИЕ ПРАВИЛА
$P \rightarrow S$	$S.next = \text{newlabel}()$ $P.code = S.code \parallel \text{label}(S.next)$  <b>begin:</b>
$S \rightarrow \text{assign}$	$S.code = \text{assign}.code$  <b>if B goto Btrue</b> <b>ifFalse B goto Bfalse</b>
$S \rightarrow \text{while } (B) S_1$	$begin = \text{newlabel}()$ $B.true = \text{newlabel}()$ $B.false = S.next$ $S_1.next = begin$ $S.code = \text{label}(begin) \parallel B.code$ $\quad \parallel \text{label}(B.true) \parallel S_1.code$ $\quad \parallel \text{gen('goto' begin)}$  <b>Btrue:</b> <i># Тело цикла – S1</i> <b>goto begin</b>  <b>Bfalse:</b>
$S \rightarrow S_1 S_2$	$S_1.next = \text{newlabel}()$ $S_2.next = S.next$ $S.code = S_1.code \parallel \text{label}(S_1.next) \parallel S_2.code$