Курс «Компиляторные технологии»

Лекция 13

Генерация промежуточного кода (1)

Курносов Михаил Георгиевич

www.mkurnosov.net

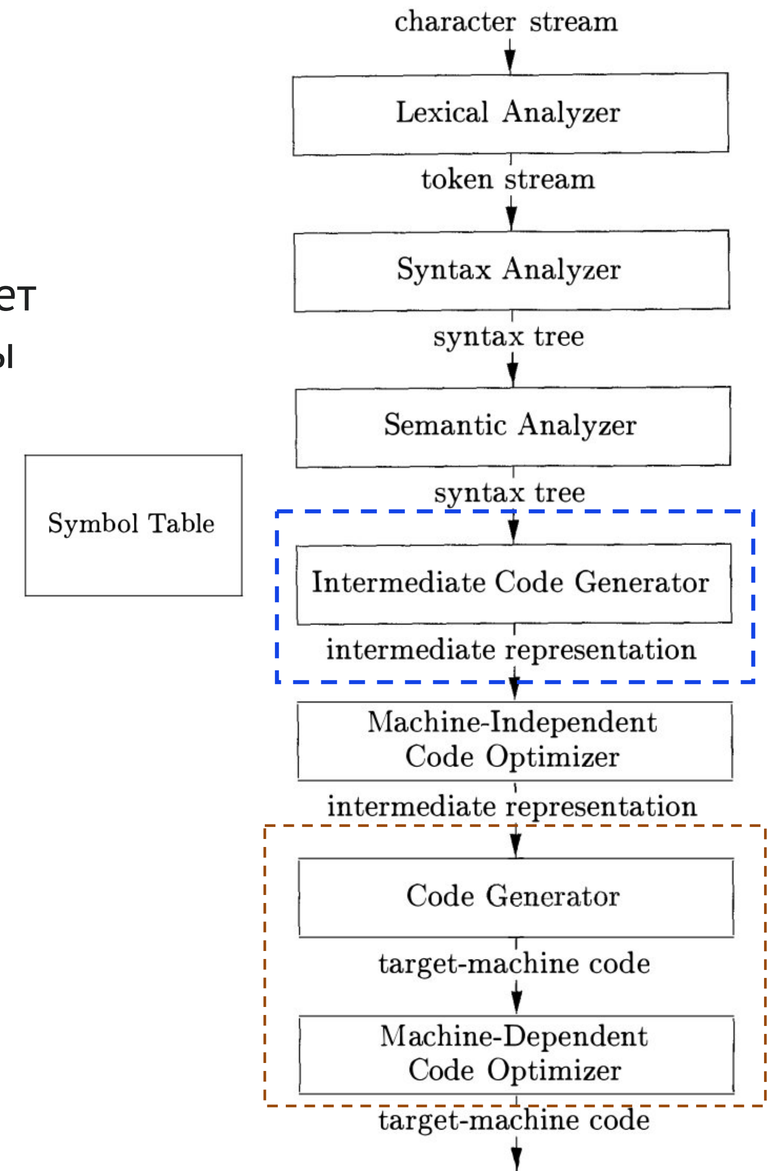
Сибирский государственный университет телекоммуникаций и информатики
Весенний семестр

Структура компилятора

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

- **Фаза синтеза** (backend, заключительная стадия) –
  транслирует программу на основе таблицы символов и промежуточного представления в код целевой архитектуры

- **Общий процесс компиляция 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

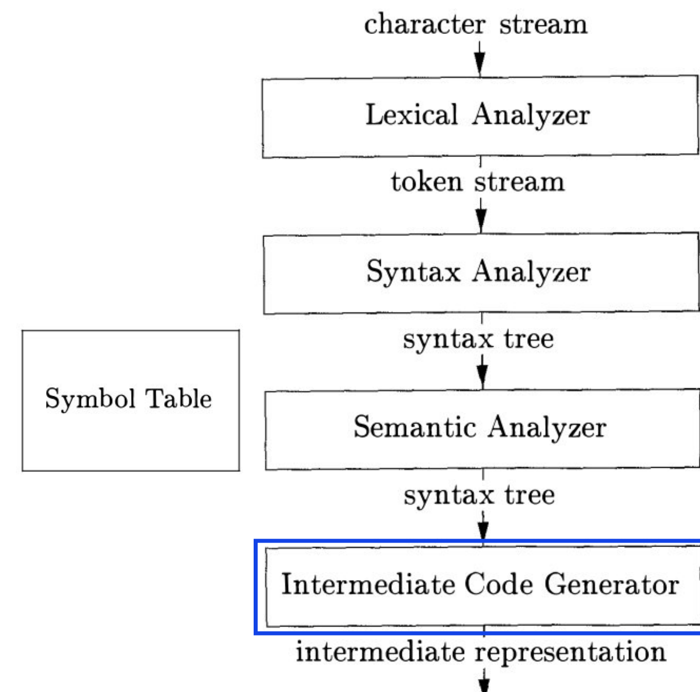


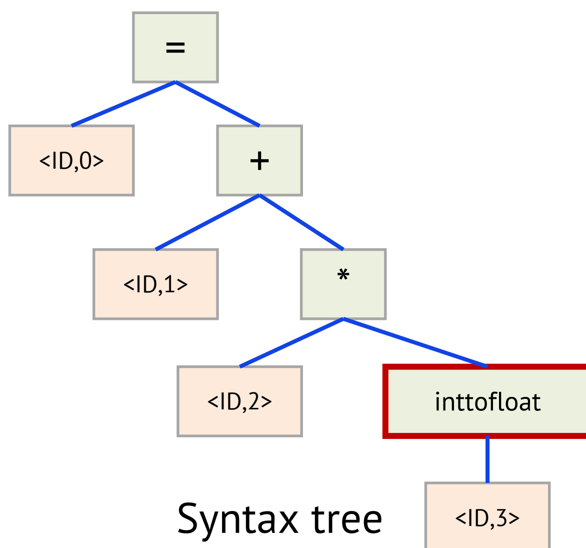
[*] Aho A., Lam M., Sethi R., Ullman J. **Compilers: principles, techniques, and tools**. – 2nd ed., 2007

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

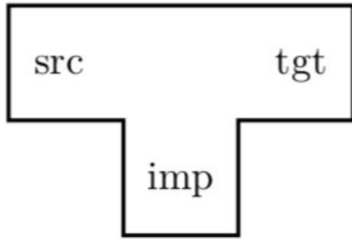




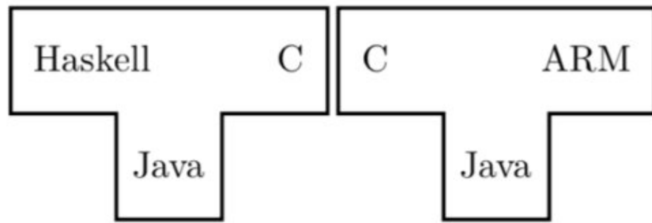
Трехадресный код (IR)

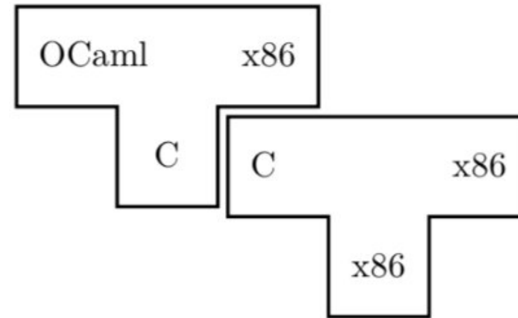
```
t1 = inttofloat(16)
t2 = id2 * t1
t3 = id1 + t2
id0 = t3
```

T-диаграммы (T-diagrams)



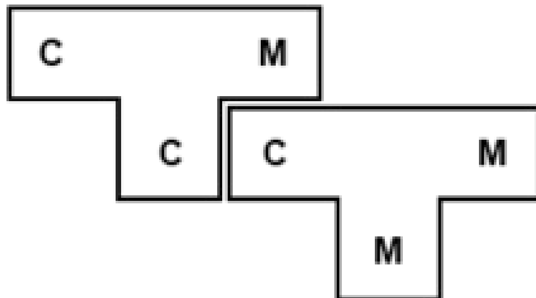
- Транслятор с языка src в язык tgt, реализованный на imp





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

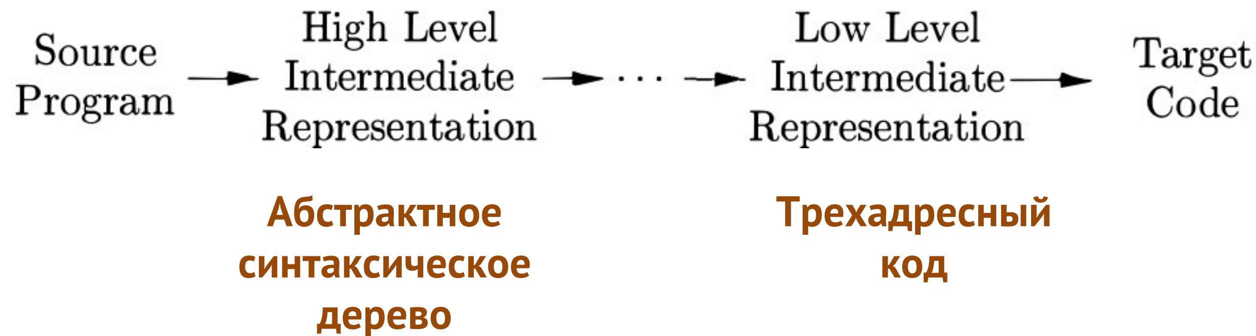
- Транслятор с OCaml в x86, реализованный на C, транслятор собирается для x86 компилятором на ассемблере x86



- **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

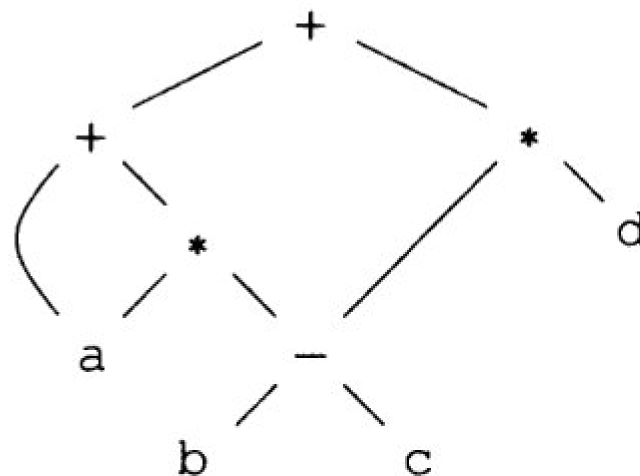
- Синтаксис и спецификация языка определяют действия начальной стадии компилятора (scanner, parser), а детали целевой машины (архитектуры) — заключительной стадией компилятора
- Эффективное промежуточное представление позволяет построить компилятор для языка *L* и целевой машины *M* путем комбинации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



Ориентированный ациклический граф

- **Ориентированный ациклический граф** (directed acyclic graph — DAG) для выражения — форма промежуточного представления для выражения
- Позволяет обнаруживать и группировать общие подвыражения (common subexpressions)
- **Внутренний узел** (internal node) — операция
- **Листовой узел** (leaf) — операнд
- Внутренний узел может иметь более одного родителя если соответствует общему подвыражению

```
a + a * (b – c) + (b – c) * d
```

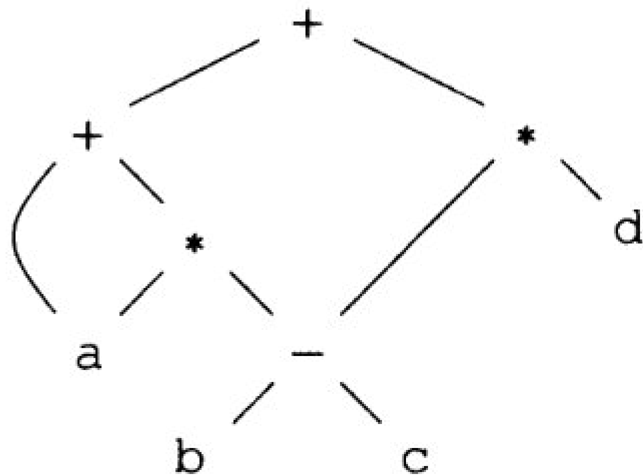


Построение ациклических графов для выражений

- Перед созданием нового узла функции *Node* и *Leaf* проверяют существование *идентичного узла*, если существует, то возвратить указатель на него
- Проверяем имеется ли узел с меткой *op* и дочерними узлами *left* и *right*

Продукция	Семантическое правило
1) $E \to E_1 + T$	$E.node = \textbf{new } Node\,('+', E_1.node, T.node)$
2) $E \to E_1 - T$	$E.node = \textbf{new } Node\,('-', E_1.node, T.node)$
3) $E \to T$	$E.node = T.node$
4) $T \to (E)$	$T.node = E.node$
5) $T \to \textbf{id}$	$T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$
6) $T \to \textbf{num}$	$T.node = \textbf{new } Leaf\,(\textbf{num}, \textbf{num}.val)$

a + a * (b – c) + (b – c) * d



$p_1 = Leaf(\textbf{id}, entry\text{-}a)$

$p_2 = Leaf(\textbf{id}, entry\text{-}a) = p_1$

$p_3 = Leaf(\textbf{id}, entry\text{-}b)$

$p_4 = Leaf(\textbf{id}, entry\text{-}c)$

$p_5 = Node\,('-', p_3, p_4)$

$p_6 = Node\,('*', p_1, p_5)$

$p_7 = Node\,('+', p_1, p_6)$

$p_8 = Leaf(\textbf{id}, entry\text{-}b) = p_3$

$p_9 = Leaf(\textbf{id}, entry\text{-}c) = p_4$

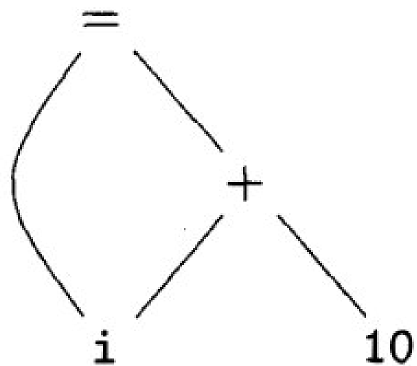
$p_{10} = Node\,('-', p_3, p_4) = p_5$

$p_{11} = Leaf(\textbf{id}, entry\text{-}d)$

$p_{12} = Node\,('*', p_5, p_{11})$

$p_{13} = Node\,('+', p_7, p_{12})$

Хранение графа в массиве



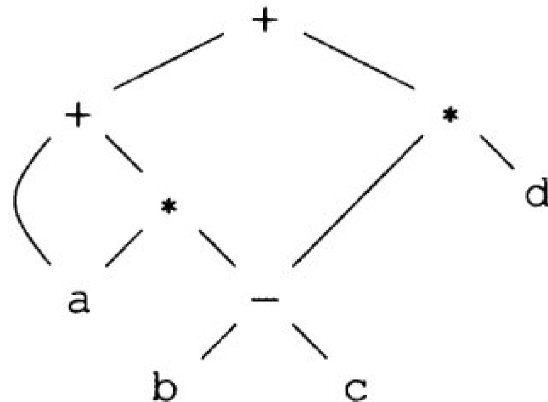
1	**id**	→ to entry for i	
2	**num**	10	
3	+	1	2
4	=	1	3
5	…		

Трехадресный код (three-address code)

```
dest = op1 op op2
```

- В трехадресном коде в правой части команды имеется не более одного оператора
- Трехадресный код – линеаризованное представление синтаксического дерева или ориентированного ациклического графа, явные имена соответствуют внутренним узлам графакст

a + a * (b – c) + (b – c) * d



$$t_1 = b - c$$
$$t_2 = a * t_1$$
$$t_3 = a + t_2$$
$$t_4 = t_1 * d$$
$$t_5 = t_3 + t_4$$

Трехадресный код (three-address code)

- **Команда присваивания** (assignment instructions): x = y **op** z,
  op — бинарная арифметическая или логическая операция, *x, y, z* — адреса
- **Команда присваивания** (assignment instructions): x = **op** y,
  op — унарная операция (минус, отрицание, сдвиг, конвертация типа), *x, y* — адреса
- **Команда копирования** (copy/move instructions): x = y, *x, y* — адреса
- **Безусловный переход** на метку L (unconditional jump/branch): goto L
- **Условные переходы**: if x relop y goto L,
  переход на метку L при истинности выражения, relop — оператор отношения (<, ==, >= и др.), *x, y* — адреса
- **Вызовы процедур, передачи параметров** и возврата:

```
param x1     # Передача первого параметра
...
param xn
call fun, n  # Вызов функции fun с n параметрами
return y     # Возврат значения из процедуры, y – необязательный адрес
```

- **Копирование с обращением по индексу** (indexed copy): x = y[i], x[i] = y
- **Присваивание адресов и указателей**:
  x = &y — устанавливает r-значение x равным l-значению y (адресу в памяти)
  x = *y — r-значение x становится равным содержимому ячейки с адресом y
  *x = y — r-значение, на которое указывает x, становится равным r-значению y

Назначения меток трехадресным командам

```
do
  i = i + 1;
while (a[i] < v);
```

```
L:  t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a [ t2 ]
    if t3 < v goto L
```

Вариант 1 — Символьные метки

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

Вариант 2 — Номера позиций (строк)

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)

Трёхадресный код	SSA
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

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)

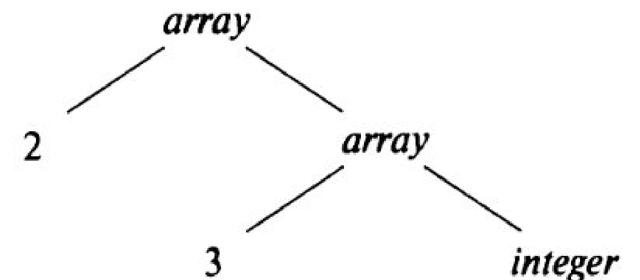
```
if (flag)
  x1 = -1; # Путь 1 потока управления
else
  x2 = 1;  # Путь 2 потока управления

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

Типы и объявления

- **Проверка типов** (type checking) – проверка соответствия (совместимости) типов операндов в конструкциях языка
- **Выведение типов** – информация о типах требуется чтобы определить размер ячейки, используется при адресации массивов
- Как правило, язык имеет фиксированное количество базовых типов (basic, buitlin, primitive)
- Составные типы (массивы, структуры) строятся на базе базовых
- Структуру составного типа данных можно описать ***выражением с конструкторами типов*** (type expressions)
- **Выражение типа** (type expression)
  - Фундаментальный тип является выражением типа (int, bool, char, void)
  - Имя типа является выражением типа
  - Выражение типа может быть образовано путем применения конструктора типа array к числу и выражению типа
  - Запись (record) представляет собой структуру данных с именованными полями

Выражение типа для int[2][3]



Эквивалентность типов

- Если выражения типов представлены графами, **два типа структурно эквивалентны** (structurally equivalent) тогда и только тогда, когда выполняется одно из следующих условий
  1. Они представляют один и тот же фундаментальный тип
  2. Образованы путем применения одного и того же конструктора к структурно эквивалентным типам
  3. Один тип представляет собой имя, обозначающее другой тип (синоним)

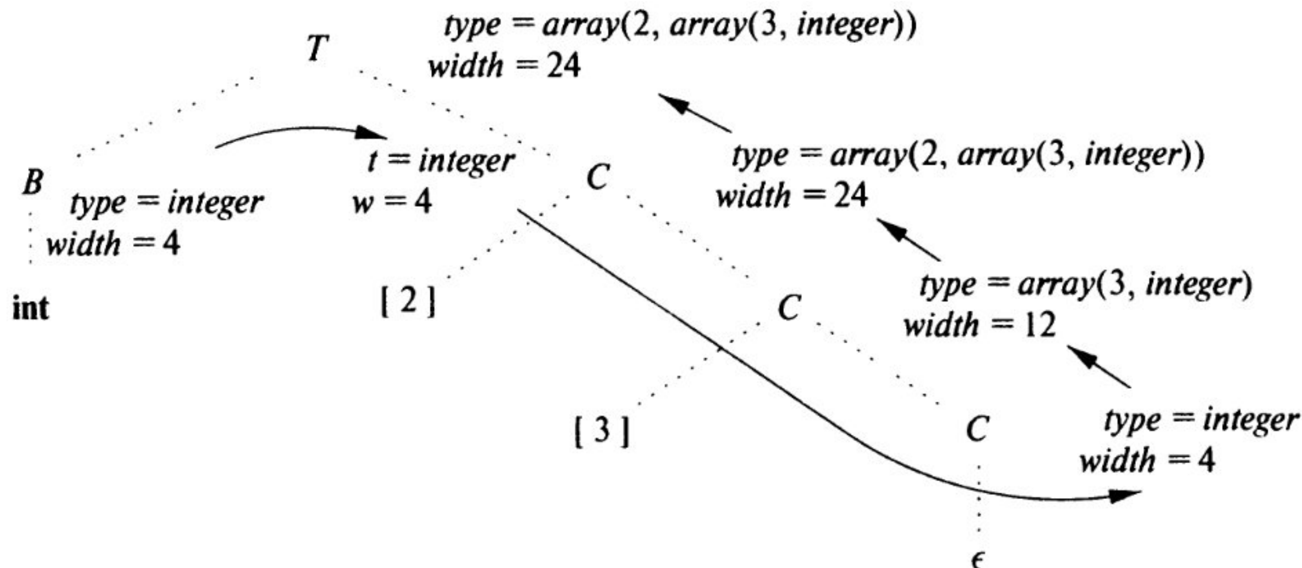
Объявления (declarations)

- **Грамматика объявления скаляров, массивов, записей**

```
int var;
record {
  float a;
  int b;
  float q;
} z;
int v[100];
```

$$
\begin{array}{lcl}
D & \rightarrow & T\ \textbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \textbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \textbf{int} \mid \textbf{float} \\
C & \rightarrow & \epsilon \mid [\ \textbf{num}\ ]\ C
\end{array}
$$

- **Вычисление размеров типов: int [2][3]**



$$
\begin{array}{lll}
T \rightarrow & B & \{\ t = B.type;\ w = B.width;\ \} \\
 & C & \\
B \rightarrow & \textbf{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B \rightarrow & \textbf{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{\ array(\textbf{num}.value,\ C_1.type); \\
 & & \quad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

Последовательности объявлений

- Отслеживание позиций размещения локальных переменных или полей в одной блоке
- Переменная вносится в таблицу символов со своим текущим смещением *offset*

$$
\begin{array}{lll}
P & \rightarrow & \{\ \mathit{offset} = 0;\ \} \\
 & & D \\
D & \rightarrow & T\ \mathbf{id}\ ; \quad \{\ \mathit{top.put}(\mathbf{id}.\mathit{lexeme},\ T.\mathit{type},\ \mathit{offset}); \\
 & & \qquad \mathit{offset} = \mathit{offset} + T.\mathit{width};\ \} \\
 & & D_1 \\
D & \rightarrow & \epsilon
\end{array}
$$

Поля в записях и классах

- Имена полей в классах, записях, структурах должны быть различны
- Смещение, или относительный адрес, имени поля отсчитывается относительно начала области данных, выделенной для записи

$$
\begin{array}{lll}
T \rightarrow \textbf{record}\ '\{' & \{\ Env.push(top);\ top = \textbf{new}\ Env(); & \text{сохраняет таблицу символов} \\
 & \quad Stack.push(offset);\ offset = 0;\ \} & \\
\quad D\ '\}' & \{\ T.type = record(top);\ T.width = offset; & \\
 & \quad top = Env.pop();\ offset = Stack.pop();\ \} & \text{восстанавливает таблицу символов}
\end{array}
$$

- После трансляции объявлений таблица символов *top* содержит типы и относительные адреса полей в этой записи

Трансляция выражений в трёхадресный код

```
a = b + −c;
```

```
t1 = minus c
t2 = b + t1
a = t2
```

Продукция	Семантические правила
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \;\|\|$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \;\|\|\; E_2.code \;\|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \;\|\|$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

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

Инкрементная трансляция

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

```
t1 = minus c
t2 = b + t1
a = t2
```

$$
\begin{array}{lll}
S \rightarrow \mathbf{id} = E\ ; & \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \} \\
E \rightarrow E_1 + E_2 & \{\ E.addr = \mathbf{new}\ Temp\,(); \\
 & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
\quad |\ \ - E_1 & \{\ E.addr = \mathbf{new}\ Temp\,(); \\
 & \quad gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr);\ \} \\
\quad |\ \ (\ E_1\ ) & \{\ E.addr = E_1.addr;\ \} \\
\quad |\ \ \mathbf{id} & \{\ E.addr = top.get(\mathbf{id}.lexeme);\ \}
\end{array}
$$

- *gen*() конструирует трехадресную команду и добавляет ее к последовательности уже сгенерированных команд

Адресация элементов массива

- Если размер каждого элемента одномерного массива *A* равен *w* байт, то *i*-й элемент массива начинается в ячейке *base* + *i* * *w*
- Двумерный массив A[i][j]: *base* + *i* * *w*1 + *j* * *w*2, *w*1 – размер строки, *w*2 – размер элемента

$$base + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_k \times w_k$$

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:

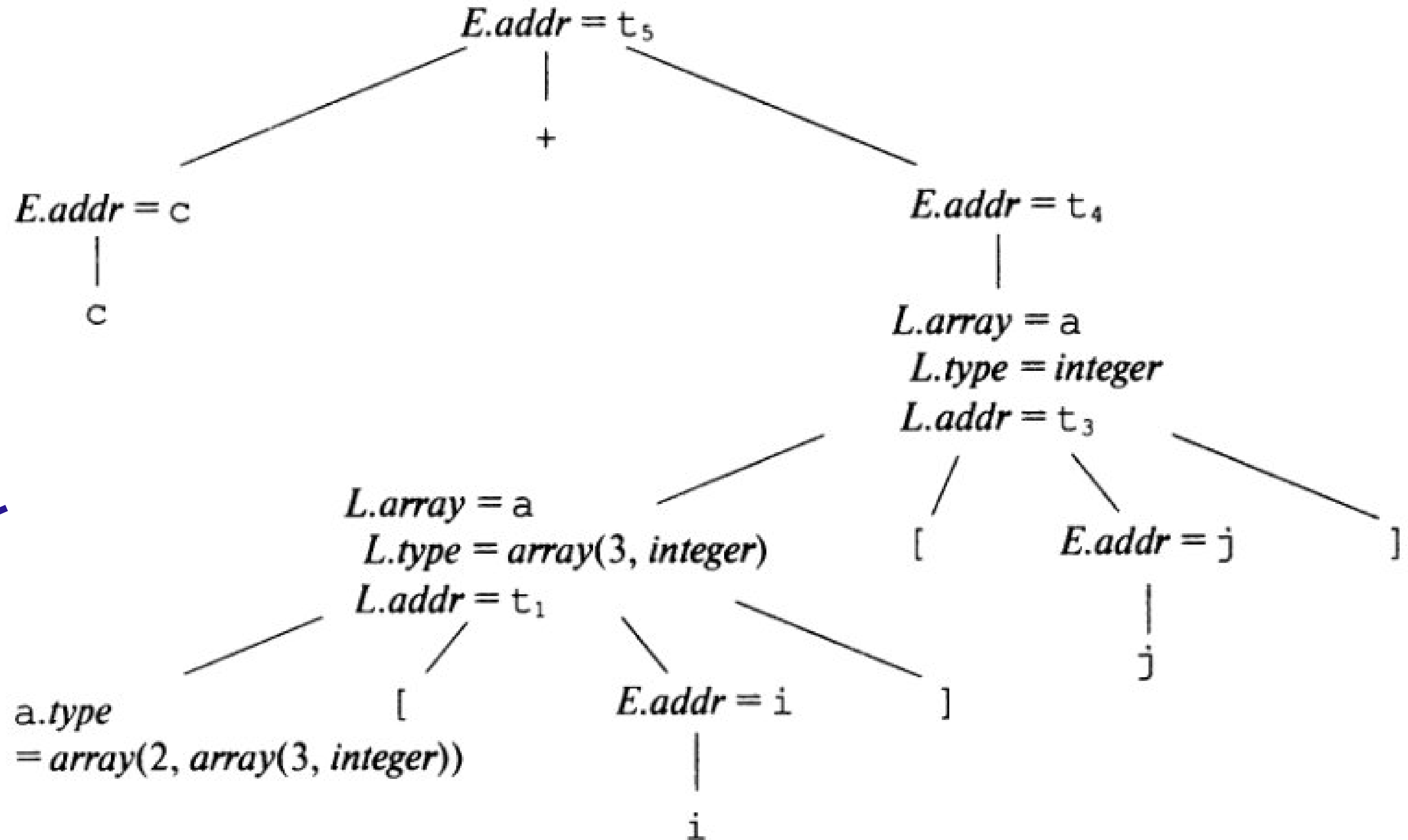
$$L \rightarrow L \; [E] \mid \textbf{id} \; [E]$$

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

$$
\begin{array}{lll}
S \rightarrow \textbf{id} = E \; ; & \{ \; gen(\; top.get(\textbf{id}.lexeme) \; '=' \; E.addr); \; \} \\
\quad \mid \; L = E \; ; & \{ \; gen(L.addr.base \; '[' \; L.addr \; ']' \; '=' \; E.addr); \; \} \\
E \rightarrow E_1 + E_2 & \{ \; E.addr = \textbf{new} \; Temp\,(); \\
 & \quad gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \; \} \\
\quad \mid \; \textbf{id} & \{ \; E.addr = top.get(\textbf{id}.lexeme); \; \} \\
\quad \mid \; L & \{ \; E.addr = \textbf{new} \; Temp\,(); \\
 & \quad gen(E.addr \; '=' \; L.array.base \; '[' \; L.addr \; ']'); \; \} \\
L \rightarrow \textbf{id} \; [\; E \;] & \{ \; L.array = top.get(\textbf{id}.lexeme); \\
 & \quad L.type = L.array.type.elem; \\
 & \quad L.addr = \textbf{new} \; Temp\,(); \\
 & \quad gen(L.addr \; '=' \; E.addr \; '*' \; L.type.width); \; \} \\
\quad \mid \; L_1 \; [\; E \;] & \{ \; L.array = L_1.array; \\
 & \quad L.type = L_1.type.elem; \\
 & \quad t = \textbf{new} \; Temp\,(); \\
 & \quad L.addr = \textbf{new} \; Temp\,(); \\
 & \quad gen(t \; '=' \; E.addr \; '*' \; L.type.width); \; \} \\
 & \quad gen(L.addr \; '=' \; L_1.addr \; '+' \; t); \; \}
\end{array}
$$

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




```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

Трехадресный код для выражения

Проверка типов (type checking)

- Для **проверки типов** (type checking) компилятор должен назначить каждому компоненту исходной программы выражение типа
- Компилятор должен определить, удовлетворяют ли эти выражения типов набору логических правил, которые называются *системой типов исходного языка* программирования
- **Динамическая проверка типов** – проверка типов может выполняться динамически, в ходе выполнения программы, если целевой код хранит не только значение элемента, но и его тип
- Реализация языка является **строго типизированной** (strongly typed), если компилятор гарантирует, что скомпилированная программа будет выполняться без ошибок, связанных с типами

- **Как определить тип выражения a + b?**
  - Первый подход: **синтез типа** (type synthesis) – тип выражения строится из типов операндов с соответствии со пецификацией языка (граф преобразований)
  - Второй подход: **выведение типа** (type inference) – определяет тип языковой конструкции из способа ее использования: listSize(x), функция принимает аргументом список, делаем вывод, что x список

Правила синтеза типа и выведение типа

- Тип выражение строится из типов подвыражений
- Тип E1 + E2 определяется типами E1 и E2
- **Вид правил синтеза**

if f имеет тип $s \to t$ **and** x имеет тип s,
then выражение $f(x)$ имеет тип t

- s —> t — функция от s и возвращающая t

- **Вид правил выведение типа**

if $f(x)$ является выражением,
then f имеет тип $\alpha \to \beta$ для некоторых α и β **and** x имеет тип α

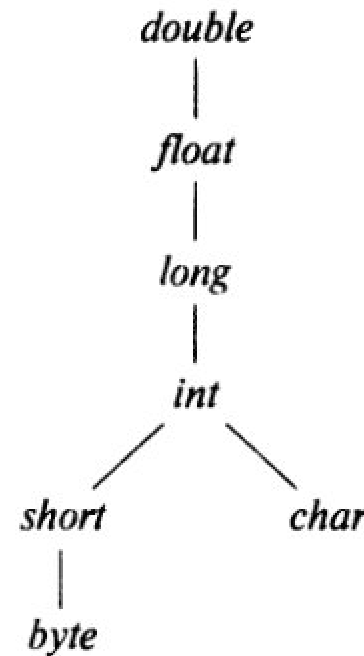
Преобразования типов (type conversion)

- Сложение целого числа и числа с плавающей точкой: i + t
- Спецификация языка определяет какие типы операндов совместимы и для каких можно без потери точности выполнять конвертации
- Целые числа при необходимости преобразуются в числа с плавающей точкой с использованием унарного оператора float

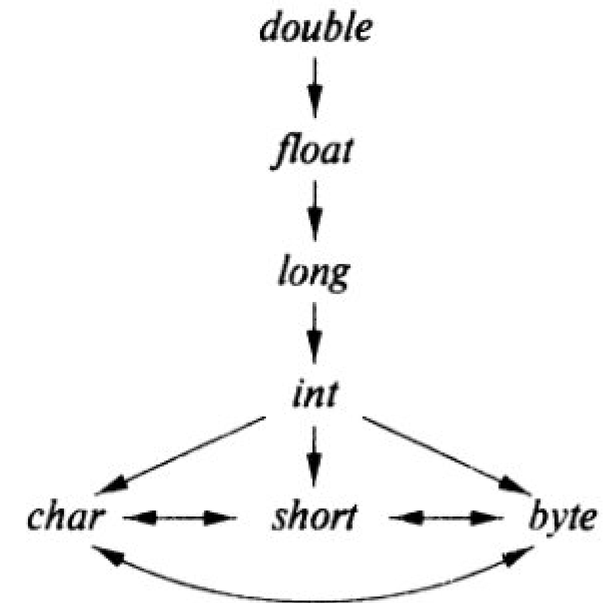
$$t_1 = (\texttt{float})\ 2$$
$$t_2 = t_1 * 3.14$$

- Синтез типов строится путем расширения схемы трансляции
- Вводится атрибут E.type — тип выражения
- Правило продукции E —> E1 + E2 строит псевдокод

if $(E_1.type = integer$ **and** $E_2.type = integer)\ E.type = integer;$
else if $(E_1.type = float$ **and** $E_2.type = integer)\ \cdots$
$\cdots$



Расширяющие преобразования типов Java (widenning conversions)
неявные, автоматические



Сужающие преобразования типов Java (Narrowing)

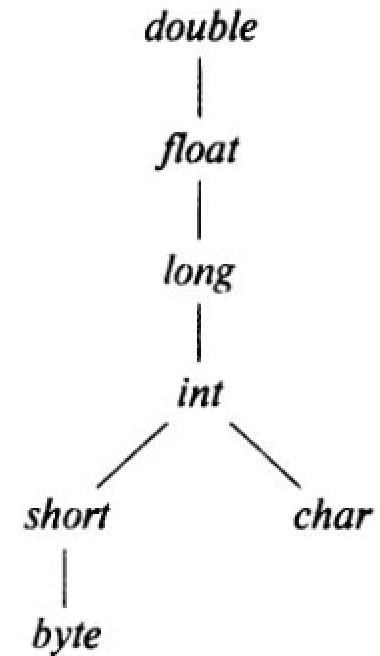
Автоматическое расширяющее преобразование типов

- Семантические действия для проверки E —> E1 + E2 используют функции *max*(t1, t2) и *widen*(a, t, w)

$$
\begin{array}{lll}
E \rightarrow E_1 + E_2 & \{ & E.type \;=\; max(E_1.type, E_2.type); \\
& & a_1 \;=\; widen(E_1.addr, E_1.type, E.type); \\
& & a_2 \;=\; widen(E_2.addr, E_2.type, E.type); \\
& & E.addr = \textbf{new}\ Temp\,(); \\
& & gen(E.addr\ '='\ a_1\ '+'\ a_2); \}
\end{array}
$$

- *max*(t1, t2) — возвращает максимальный из двух типов в иерархии расширения (или их наименьшую верхнюю границу), ошибка если типа нет в иерархии или он является указателем или массивом
- *max*(short, float) = float, *max*(short, char) = int

- *widen*(a, t, w) — генерирует расширяющее преобразование типа для значения по адресу a типа t в значение типа w

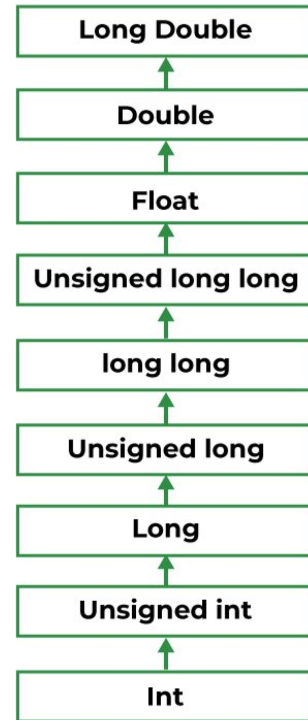


Расширяющие преобразования типов Java (widenning conversions)
неявные, автоматические

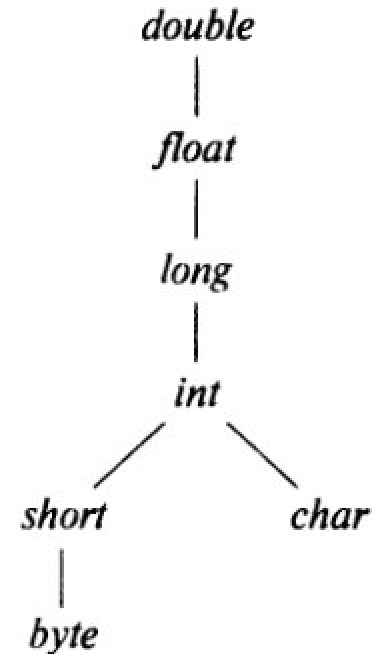
Автоматическое расширяющее преобразование типов

- *widen*(a, t, w) – генерирует расширяющее преобразование типа для значения по адресу a типа t в значение типа w

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```



C implicit type casting



Расширяющие преобразования типов Java (widenning conversions) *неявные, автоматические*

Поток управления (control flow)

- Трансляция инструкций изменения **потока управления** if-else и while связана с трансляцией булевых выражений

- **Булевы выражения**
  - **rel**: <, <=, =, !=, >, >=
  - старшинство операций: !, &&, ||

$$B \rightarrow B \| B \mid B \;\&\&\; B \mid !B \mid (B) \mid E \;\textbf{rel}\; E \mid \textbf{true} \mid \textbf{false}$$

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

- **При вычислениях по сокращенной схеме** (short-circuit code) булевы операторы &&, || и ! транслируются в уловные 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`

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

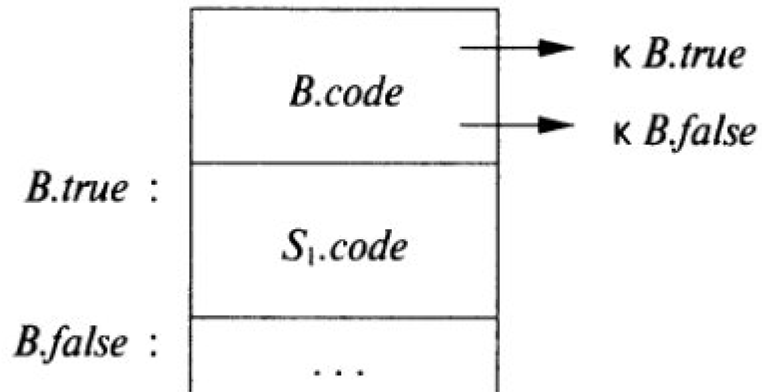
https://en.cppreference.com/w/cpp/language/operator_logical

C++
«Builtin operators **&&** *and* **||** *perform short-circuit evaluation (do not evaluate the second operand if the result is known after evaluating the first),* ***but overloaded operators behave like regular function calls and always evaluate both operands****».*

Трансляция if-then

$$S \rightarrow \textbf{if } (B)\ S_1$$

- Если B истинно, управление переходит к первой команде S1 .code, а если B ложно, управление переходит к команде, следующей непосредственно за S1.code




```
if B goto Btrue
ifFalse B goto Bfalse

Btrue:
# Ветвь true – S1

Bfalse:
```

- Работа с метками для переходов в B.code и S.code выполняется с использованием наследуемых атрибутов
- S.next – команда, следующая непосредственно за кодом S
- В некоторых случаях командой, непосредственно следующей за S.code, оказывается команда перехода к некоторой метке L
- Перехода к переходу к метке L из кода S.code можно избежать, используя S.next

Трансляция if-then

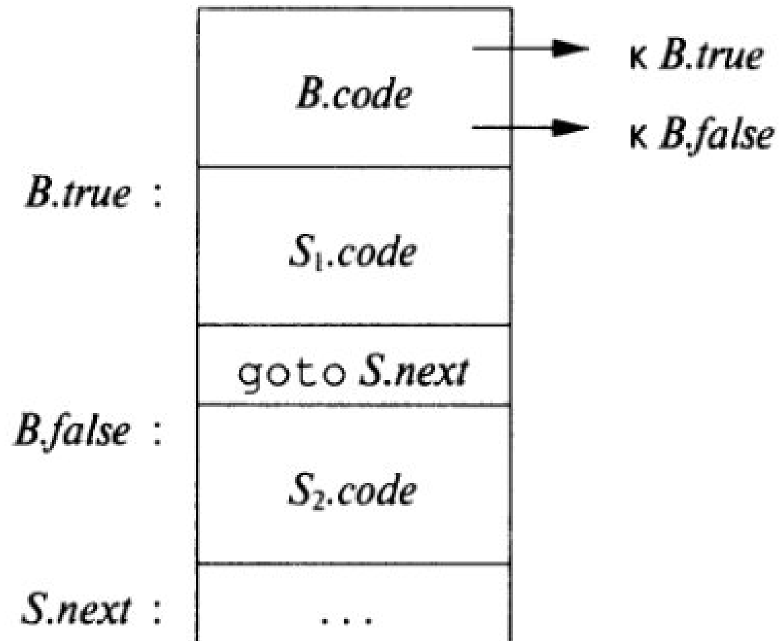
Продукция	Семантические правила
$P \to S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \to \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \to \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$

```
if B goto Btrue
ifFalse B goto Bfalse

Btrue:
# Ветвь true – S1

Bfalse:
```

- Вызов newlabel() создает новую метку
- label(L) назначает метку L очередной генерируемой трехадресной команде
- P.code состоит из S.code, за которым следует новая метка S.next
- Токен assign – «заполнитель» для инструкций присваивания (рассмотрели ранее)

Трансляция if-then-else

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




```
if B goto Btrue
ifFalse B goto Bfalse

Btrue:
# Ветвь true – S1
goto Snext

Bfalse:
# Ветвь false – S2

Snext:
```

Трансляция if-then-else

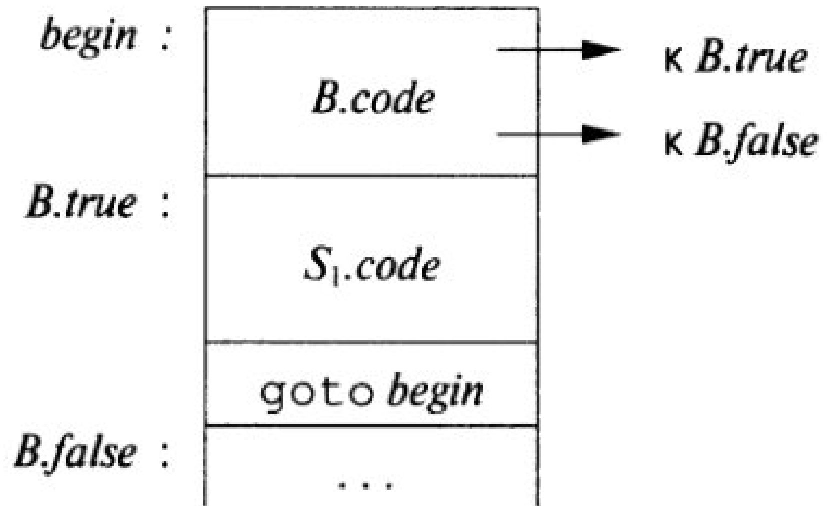
Продукция	Семантические правила
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen(\texttt{'goto'}\ S.next)$ $\parallel label(B.false) \parallel S_2.code$

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

```
if B goto Btrue
ifFalse B goto Bfalse

Btrue:
# Ветвь true – S1
goto Snext

Bfalse:
# Ветвь false – S2

Snext:
```

Трансляция цикла while

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




```
begin:

if B goto Btrue
ifFalse B goto Bfalse

Btrue:
# Тело цикла — S1
goto begin

Bfalse:
```

Трансляция цикла while

Продукция	Семантические правила
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{while} (B) S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$

```
begin:

if B goto Btrue
ifFalse B goto Bfalse

Btrue:
# Тело цикла — S1
goto begin

Bfalse:
```

Трансляция булевых выражений с помощью потока управления

- Булево (логическое) выражение B транслируется в трехадресные команды перехода к одной из двух меток: B.true или B.false

```
a < b
```

↓

```
if a < b goto B.true
goto B.false
```

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

↓

```
                    if x < 100 goto L2
                    goto L3
              L3:   if x > 200 goto L4
                    goto L1
              L4:   if x != y goto L2
                    goto L1
     B.true   L2:   x = 0
     B.false  L1
```

Трансляция булевых выражений с помощью потока управления

Продукция	Семантические правила
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$

$B \rightarrow !\, B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen($'if' $E_1.addr$ **rel**$.op$ $E_2.addr$ 'goto' $B.true)$ $\mid\mid gen($'goto' $B.false)$
$B \rightarrow$ **true**	$B.code = gen($'goto' $B.true)$
$B \rightarrow$ **false**	$B.code = gen($'goto' $B.false)$

Трансляция булевых выражений с помощью потока управления

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

```
                 if x < 100 goto L2
                 goto L3
            L3:  if x > 200 goto L4
                 goto L1
            L4:  if x != y goto L2
                 goto L1
     B.true L2:  x = 0
    B.false L1
```

$$S \rightarrow \textbf{if}\ (B)\ S_1$$

- Для if генерируется метка B.true = L2
- Оератор II имеет меньший приоритет, продукция B1 || B2, следовательно B1.true = L2, B1.false = newlabel() = L3 — первая инструкция B2

Трансляция булевых выражений с помощью потока управления

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

```
               if x < 100 goto L2
               goto L3
          L3:  if x > 200 goto L4
               goto L1
          L4:  if x != y goto L2
               goto L1
B.true    L2:  x = 0
B.false   L1
```

- **Код не оптимален** – содержит на три команды безусловного перехода больше, чем альтернативный вариант
- Команда goto L3 лишняя
- Две команды goto L1 могут быть устранены, если вместо команды if использовать команду if False

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

Устранение лишних команд перехода

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

```
             if x < 100 goto L2
             goto L3
        L3:  if x > 200 goto L4
             goto L1
        L4:  if x != y goto L2
             goto L1
 B.true L2:  x = 0
B.false L1
```

```
     ifFalse x > 200 goto L1
L4:  ...
```

- **Переход к ifFalse**
- Реализации концепции выполнения ветвлений «"falls through» – переход только при одном условии, по второй ветви просто проваливаемся на следующий блок
- Устраняется дополнительный goto

Устранение лишних команд перехода

```
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:
```

- **Переход к ifFalse**
- Реализации концепции выполнения ветвлений «"falls through» — переход только при одном условии, по второй ветви просто проваливаемся на следующий блок
- Устраняется дополнительный goto