# 14. Выбор команд и генерация машинного кода

# *1.1 Вводные замечания*

## 1.1.1 Постановка задачи

- ◊ Оптимизатор работает над промежуточным представлением программы, и когда процесс оптимизации закончится, получится оптимизированная программа в промежуточном представлении.
- ◊ На следующем этапе необходимо перевести программу из промежуточного представления в код целевого процессора.
- ◊ Отображение *инструкций* промежуточного представления в *команды* целевого процессора называется *выбором команд*.
- ◊ Существует несколько подходов к решению этой задачи. Рассмотрим подход, использующий сопоставление шаблонов на абстрактном синтаксическом дереве, построенном по оптимизированному промежуточному представлению.

# *1.1 Вводные замечания*

## 1.1. 2 Вход и выход генератора кода

- Входной поток генератора кода:
  - *промежуточное представление* исходной программы (последовательность трехадресных инструкций)
  - *таблица символов*, которая используется для определения адресов времени выполнения объектов данных, обозначаемых в промежуточном представлении именами.
- Выходной поток генератора кода: *объектный код* (последовательность команд *целевого процессора*)
- *Объектный модуль* – часть программы в объектном коде, соответствующая исходному модулю
- *Компоновщик* объединяет объектные модули и библиотеки в единую программу, которая и выполняется на целевом процессоре.

# *1.2 Объектный код*

## 1.2.1 Общая характеристика

Предположения о целевом процессоре (ядре)

- $CISC$-архитектура
- Только целочисленная арифметика.
- Байтовая адресация памяти
- $n$ регистров общего назначения $R_0$, $R_1$, ..., $R_{n-1}$
- Команда: **`op, dst, src1, src2`**
- Операции:
  - загрузки и сохранения регистров,
  - вычислительные,
  - условные и безусловные переходы.

# *1.2 Объектный код*

## 1.2.2 Система команд

| | |
|---|---|
| **LD, r, x** | *Загрузка* значения из ячейки памяти **x** на регистр **r** |
| **LD, r1, r2** | *Копирование* значения регистра **r2** на регистр **r1** |
| **ST, x, r** | С*охранение* значения регистра **r** в ячейке памяти **x** |
| **OP**[1] **dst, src1, src2** | Бинарная *операция*: **dst = src1 OP src2** |
| **OP**[2] **dst, src1** | Унарная *операция*: **dst = OP src1** |
| **BR L** | *Переход* на команду с меткой **L** (**goto L**). |
| **Bcond**[3] **r, L** | *Ветвление*: переход на команду с меткой **L**, если значение на регистре **r** удовлетворяет условию **cond**: **if(cond)goto L** |

[1] определены арифметические, булевы и поразрядные операции

[2] определены арифметические, булевы и поразрядные операции

[3] формирование условий выполняется с помощью операций отношения и арифметических операций

# *1.2 Объектный код*

## 1.2.3 Режимы адресации

◊ *Режим прямой адресации* – адрес задается непосредственно значением **`x`**

◊ *Режим индексированной адресации* **`a(r)`** , **`a`** – переменная, **`r`** – регистр. Адрес **`a(r)`** вычисляется путем прибавления к *l*-значению **`a`** значения из регистра **`r`**.

Например, команда **`LD R1, 100 (R2)`** выполняет присваивание

**`R1 = contents (100 + contents (R2))`**

Здесь **`contents(x)`** обозначает *содержимое аргумента* (регистра или ячейки памяти), но является разыменованием, т.е. **`(*x)`** , только в случае, если аргумент не регистр, а адрес. Этот режим адресации полезен для обращения к массивам: **`a`** – базовый адрес массива, **`r`** – смещение.

*Замечание.* **`ld r1, a(r2)`** соответствует **`ldr r1, [a + r2]`** в синтаксисе ассемблера ARM32.

# *1.2 Объектный код*

## 1.2.3 Режимы адресации

◊ *Режим косвенной адресации (1)*:

**`*r`** или **`*c(r)`** – ссылка на ячейку памяти, адрес которой находится по адресу **`c + contents(r)`**, т. е.
**`LD R1,*100(R2)`** выполняет присваивание
**`R1 = contents(contents(100+contents(R2))`**,
где **`c`** – константа, **`contents()`** – содержимое регистра или ячейки памяти)

*Замечание.* Команда **`LD R1,*100(R2)`** соответствует двум операциям в ARM32, т.е. имеет двойную косвенность:

```
ldr r3, [r2 + 100]
ldr r1, [r3]
```

В современных процессорах подобные команды с многоуровневым разыменованием практически не встречаются, но мы ее рассматриваем в рамках модельной архитектуры.

# *1.2 Объектный код*

## 1.2.3 Режимы адресации

◊ *Режим косвенной адресации (2)*:

**`*r2(c(r1))`** – ссылка на ячейку памяти, адрес которой находится по адресу **`c + contents(r)`**, т. е.

**`ST *R2(`**$C_a$**`(`**$R_{SP}$**`), R3`** выполняет присваивание **`contents(R2+contents(`**$C_a$**`+contents(`**$R_{SP}$**`)))`**,

где $C_a$ – константа, **`contents()`** – содержимое регистра или ячейки памяти), $R_{SP}$ – регистр, указывающий на вершину стека.

*Замечание.* Эта команда соответствует двум операциям в ARM32, т.е. имеет двойную косвенность:

```
ldr r4, [R_SP + C_a]
str r3, [r4 + r2]
```

В современных процессорах подобные команды с многоуровневым разыменованием практически не встречаются, но мы ее рассматриваем в рамках модельной архитектуры.

# *1.2 Объектный код*

## 1.2.3 Режимы адресации

◊ *Режим адресации с использованием констант*, для указания которых используется префикс **#**.

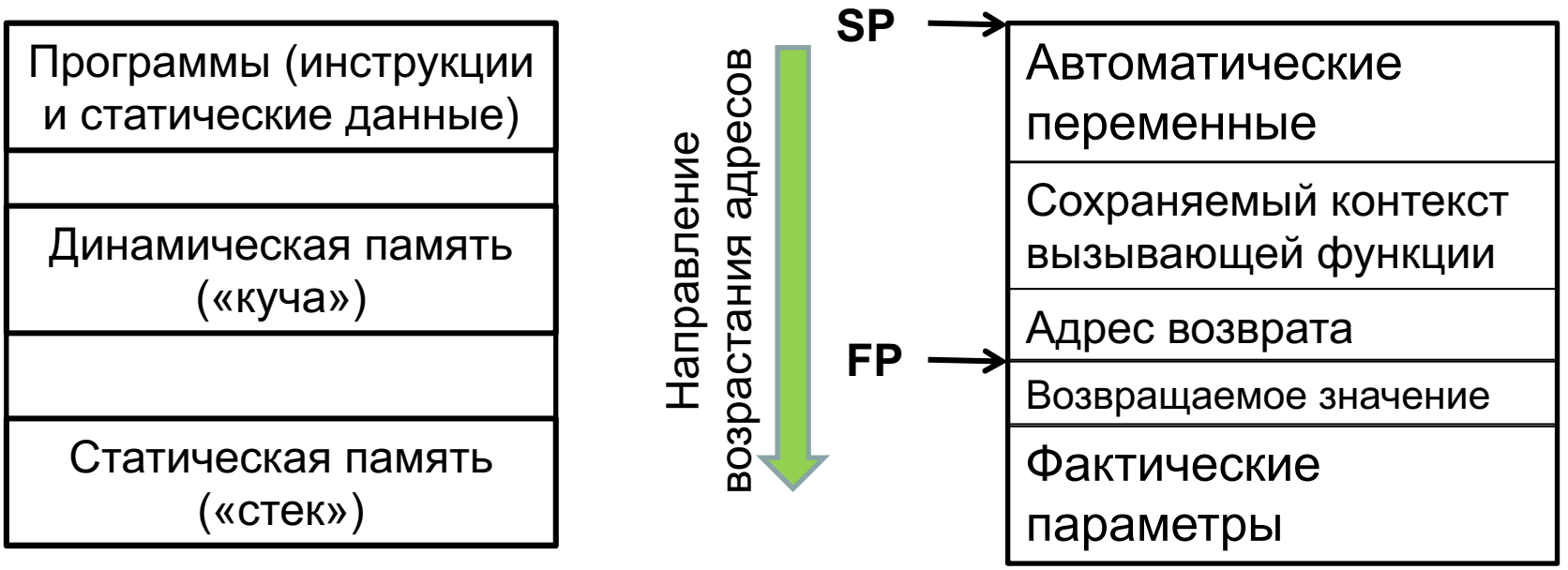
**Например**, команда **`LD r,#n`** загружает в регистр **`r`** целое число **`n`**

# *1.3 Основные фазы генерации кода*

- ◊ Распределение памяти
- ◊ Выбор команд
- ◊ Распределение регистров
- ◊ Выбор оптимального порядка команд (планирование кода)

# 1.3 Основные фазы генерации кода

## 1.3.1. Распределение памяти




Статическая память состоит из «фреймов», динамически выделяемых в стеке при вызове процедуры и удаляемых при выходе из нее. Структура фрейма показана на правом рисунке.

# *1.4 Выбор команд*

## 1.4.1. Постановка задачи

- Если не требовать эффективности целевой программы, выбор команд предельно прост:
  - для каждого типа трехадресных инструкций определяется шаблон целевого кода, генерируемого для таких инструкций;
  - каждая трехадресная инструкция заменяется соответствующим шаблоном
- Рассмотрим несколько простых примеров

# 1.4 Выбор команд

## 1.4.2. Примеры шаблонов

◊ **Пример 1**. При генерации кода для трехадресной инструкции **x ← +, y, z**, где память под **x**, **y** и **z** выделяется статически (**static**), можно использовать следующий шаблон:

```
LD    R0,  y          // загрузить y в R0
ADD   R0,  R0,  z     // сложить z и R0
ST    x,   R0         // сохранить R0 в x
```

Применим этот шаблон к последовательности из двух инструкций **a ← +, b, c**; **d ← +, a, e**;

```
LD    R0,  b          //R0 ← b
ADD   R0,  R0,  c     //R0 ← +, R0, c
ST    a,   R0         //a ← R0
LD    R0,  a          //R0 ← a
ADD   R0,  R0,  e     //R0 ← R0 + e
ST    d,   R0         //d ← R0
```

# *1.4 Выбор команд*

## 1.4.2. Примеры шаблонов

◊ **Пример 2**. Многие компьютеры имеют команду **`INC x`**:

◊ Ей соответствует шаблон

```
INC   x                    // x++
```

◊ Очевидно, что использование шаблона **`INC`** дает лучший код, чем использование шаблона из примера 1

```
LD    R0,   x              // загрузить x в R0
ADD   R0,   R0,   1        // сложить 1 и R0
ST    x,    R0             // сохранить R0 в x
```

# *1.4 Выбор команд*

## 1.4.2. Примеры шаблонов

◊ **Пример 3**. Шаблон для **+, x, y, z** (**x, y, z** – адреса ячеек памяти)

```
LD   R1, y
LD   R2, z
ADD  R1, R1, R2
ST   x, R1
```

◊ **Пример 4**. Шаблон для **ifTrue(y < z) gotoL**

```
LD     R1, y
LD     R2, z
SUB    R1, R1, R2
BLTZ   R1, L
```

# *1.4 Выбор команд*

## 1.4.2. Примеры шаблонов

◊ **Пример 5**. Шаблон для

(a) `b ← a[i]` (**a** – массив 8-байтных значений)

```
LD  R1, i
MUL R1, R1, #8
LD  R2, a(R1)
      // R2←contents(a+contents(i))
ST  b, R2
```

(b) `a[k] ← c`

```
LD  R1, c
LD  R2, k
MUL R2, R2, #8
ST  a(R2), R1
      // contents(a+contents(k))←R1
```

# *1.4 Выбор команд*

## 1.4.2. Примеры шаблонов

◊ **Пример 6**. Шаблоны для

(a) `x ← *p`

```
LD   R1, p
LD   R2, 0(R1)
ST   x,  R2
      //R2←contents(0+contents(R1))
```

(b) `*p ← y`

```
LD   R1,    p
LD   R2,    y
ST   0(R1), R2
      //contents(0+contents(R1))←R2
```

# *1.4 Выбор команд*

## 1.4.3 Стоимость команд

- Каждая команда целевого языка имеет связанную с ней стоимость.
- Если считать стоимость команды равной единице плюс стоимости, связанные с режимами адресации операндов, то стоимость будет пропорциональна длине команды в словах:
  - Стоимость операнда на регистре равна 0
  - Стоимость операнда из ячейки памяти равна 1
  - Стоимость операнда-константы равна 1
    Такой подход обосновывается тем, что адреса ячеек памяти и константы хранятся в словах, следующих за командой
  - Стоимость каждого разыменования равна 1

# *1.4 Выбор команд*

## 1.4.3 Стоимость команд

◊ **Примеры**.

1) Стоимость команды **`LD R0, R1`** равна 1.

2) Стоимость команды **`LD R0, M`** (**`M`** – адрес ячейки памяти) равна 2.

3) Стоимость команды **`LD R0, *100(R1)`**

($R_0 \leftarrow contents(contents(100 + contents(R_1)))$) равна 4.

◊ **Стоимость программы** (на целевом языке) равна сумме стоимостей ее команд. Чем меньше стоимость программы, тем быстрее она выполняется

◊ Алгоритм генерации кода должен минимизировать стоимость программы.

# *1.5 Метод переписывания дерева*

## 1.5.1 Схема трансляции деревьев

◊ Рассмотрим инструкцию присваивания

**`p[i] ← +, b, 1`** **(1)**

◊ Пусть

- ♦ массив располагается в динамической памяти (выделен с помощью `malloc` или `new`),

  указатель **`p`** на динамический массив хранится в стеке времени выполнения
  (адреса времени выполнения локальных переменных **`p`** и **`i`** заданы как смещения $C_p$ и $C_i$ относительно указателя на начало текущей *записи активации* **`SP`**, адрес **`SP`** хранится на специальном регистре $\mathbf{R_{SP}}$)
- ♦ переменная **`b`** хранится в глобальной ячейке памяти $M_b$.

◊ В целевом коде инструкции (1) будет соответствовать последовательность команд:

# 1.5 Метод переписывания дерева

## 1.5.1 Схема трансляции деревьев

◊ Пусть **p** – указатель на массив 64-битных чисел в динамической памяти, а сам указатель – автоматическая переменная (т.е. расположен на стеке), тогда в целевом коде инструкции **p[i] ← +, b, 1** будет соответствовать последовательность команд:

**LD** **R1,** $M_b$ **// R1 = b**

**ADD** **R1, R1, #1** **// R1 = b + 1**

**LD** **R2,** $C_i$ **(R$_{SP}$)** **// R2 = contents(Ci + contents(R$_{SP}$))**

**MUL** **R2, R2, 8** **// R2 = R2 * 8**

**ST** ***R2(**$C_p$ **(R$_{SP}$)), R1**

**//R1 = contents(contents(**$C_p$ **+ contents(R$_{SP}$)) + contents(R2))**

◊ Введем операцию индексирования **ind** (*index*):

**ind(**$C_i$ **+ R$_{SP}$) = contents(**$C_i$ **+ contents(R$_{SP}$)) = contents(R2) = i**

**ind(ind(**$C_p$ **+ R$_{SP}$) + ind(**$C_i$ **+ R$_{SP}$)*8) =**

**= contents(contents(**$C_p$ **+ contents(R$_{SP}$))+contents(R2)*8) =**

**= p[i]**

# *1.5 Метод переписывания дерева*

## 1.5.1 Схема трансляции деревьев

◊ Рассмотрим инструкцию присваивания

**a[i] ← +, b, 1** **(1)**

◊ Пусть

- ♦ массив **a** (сама адресуемая область массива) располагается в стеке времени выполнения (адрес начала массива **a** и локальная переменная **i** заданы как смещения $C_a$ и $C_i$ относительно указателя на начало текущей *записи активации* **SP**, адрес **SP** хранится на специальном регистре $\mathbf{R_{SP}}$)
- ♦ переменная **b** хранится в глобальной ячейке памяти $M_b$.

◊ В целевом коде инструкции (1) будет соответствовать последовательность команд:

# *1.5 Метод переписывания дерева*

## 1.5.1 Схема трансляции деревьев

◊ Пусть **a** – байтовый массив, расположенный в на стеке, адрес его начала вычисляется по смещению $C_a$ относительно $\mathbf{R_{SP}}$, тогда в целевом коде инструкции **a[i] ← +, b, 1**
будет соответствовать последовательность команд:

**LD** **R1,** $M_b$ **// R1 = b**

**ADD** **R1, R1, #1** **// R1 = b + 1**

**LD** **R2,** $C_i$**(**$\mathbf{R_{SP}}$**)** **// R2 = contents(Ci + contents(**$\mathbf{R_{SP}}$**))**

**ADD** **R2, RSP, R2**

**ST** $C_a$**(R2), R1**

**// R1 = contents(**$C_a$ **+ contents(**$\mathbf{R_{SP}}$**) + contents(R2))**

◊ Введем операцию индексирования **ind** (*index*):

**ind(**$C_i$ **+** $\mathbf{R_{SP}}$**) = contents(**$C_i$ **+ contents(**$\mathbf{R_{SP}}$**)) = contents(R2) = i**

**ind((**$C_a$ **+** $\mathbf{R_{SP}}$**) + ind(**$C_i$ **+** $\mathbf{R_{SP}}$**)*8) =**

**= contents(**$C_a$ **+ contents(**$\mathbf{R_{SP}}$**) + contents(R2)) =**

**= a[i]**

# 1.5 Метод переписывания дерева

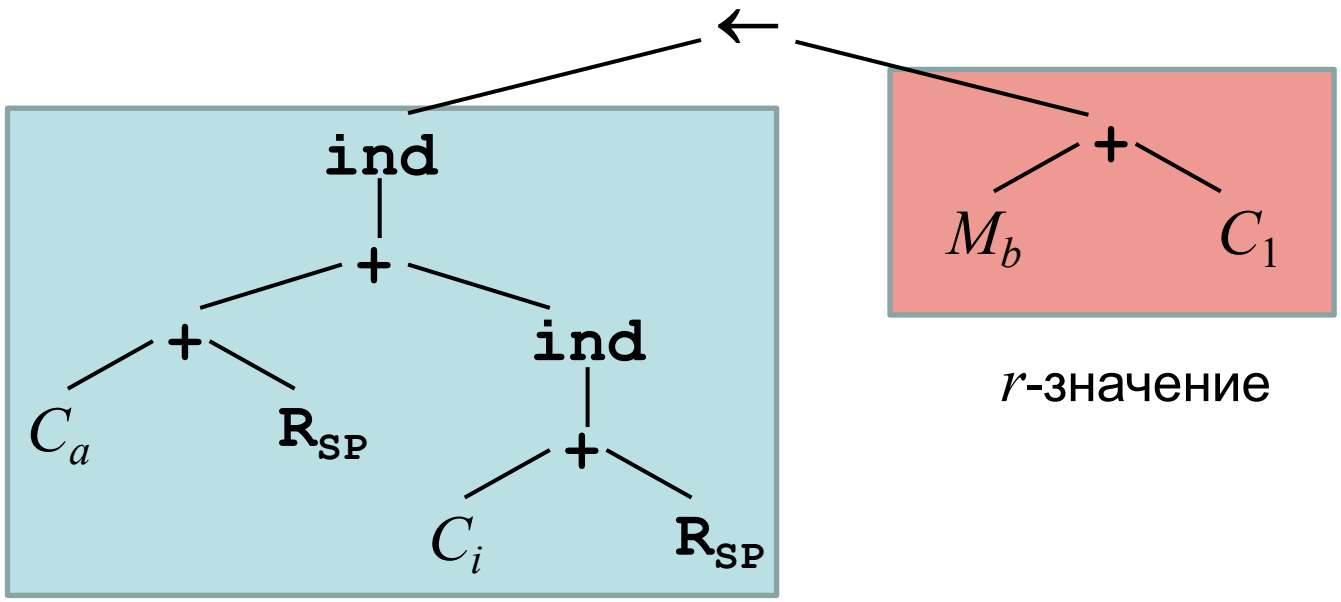
## 1.5.1 Схема трансляции деревьев

**a[i] ← +, b, 1  // байтовый массив на стеке**

**ind($C_i$ + $R_{SP}$) =**
**contents($C_i$ + contents($R_{SP}$)) = contents(R2) = i**

**ind(($C_a$ + $R_{SP}$) + ind ($C_i$ + $R_{SP}$)) =**
**contents(contents($C_a$ + contents($R_{SP}$))+contents(R2))**
**= a[i]**

**+($M_b$ $C_1$) = +, b, #1**



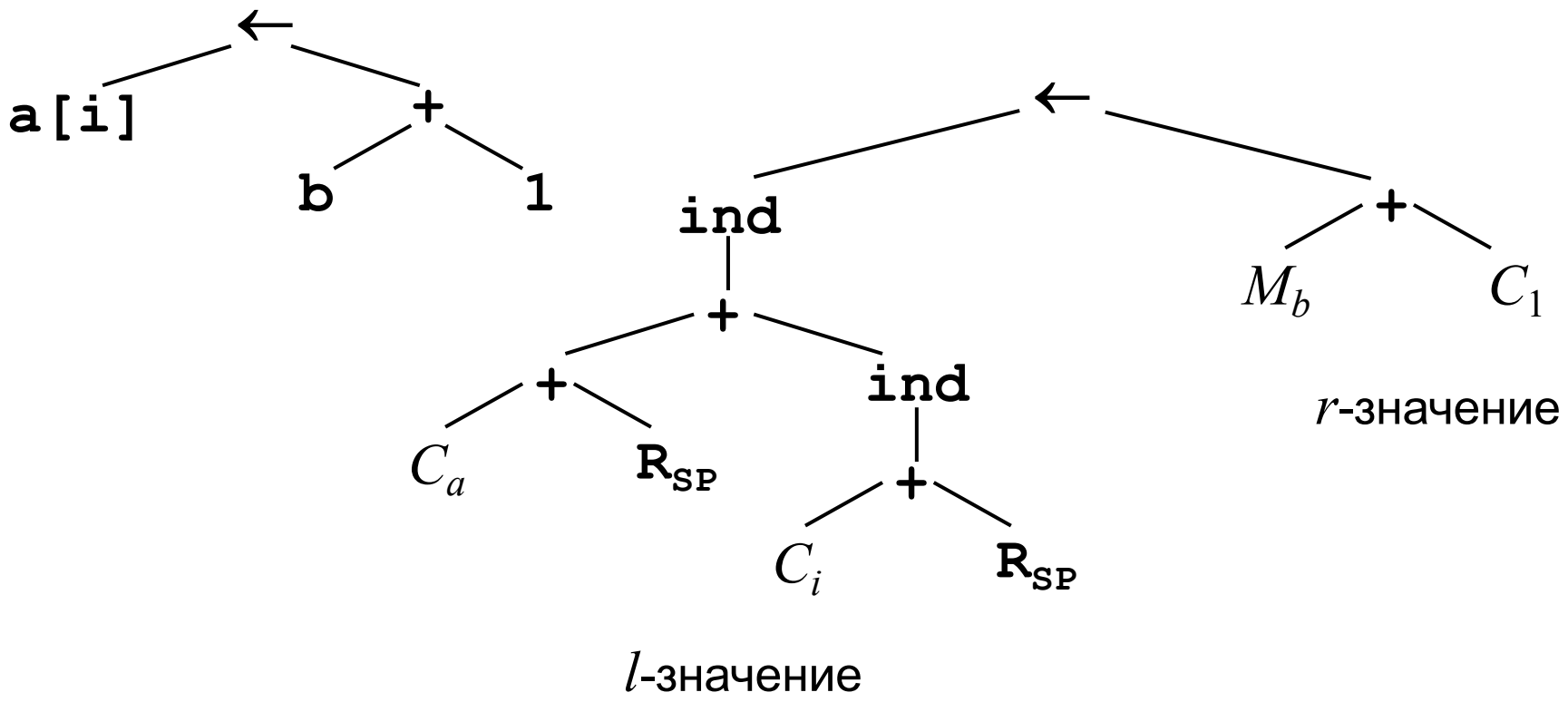

*r*-значение

*l*-значение

# *1.5 Метод переписывания дерева*

## 1.5.1 Схема трансляции деревьев

Таким образом, исходное дерево (выражение) на левом рисунке должно быть сначала преобразовано (переписано) в дерево на правом рисунке



# *1.5 Метод переписывания дерева*

## 1.5.2 Описание метода

- ◊ Целевой код генерируется в процессе *свертки входного дерева в единый узел* путем последовательного применения *правил преобразования дерева*.
- ◊ Каждое *правило преобразования дерева* представляет собой инструкцию вида

$$replacement \leftarrow template \{action\}$$

где *replacement* – отдельный узел, *template* – шаблон (поддерево), *action* – действие (фрагмент генерируемого кода, соответствующий шаблону).

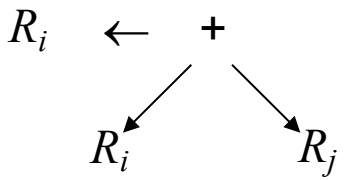
- ◊ Множество правил преобразования дерева именуется *схемой трансляции дерева*.
- ◊ Трансляция состоит из (возможно, пустой) последовательности машинных команд, которые генерируются действием (*action*), связанным с шаблоном (*template*).

# *1.5 Метод переписывания дерева*

## 1.5.3 Схема трансляции дерева

- Схема трансляции дерева – удобный способ описания схемы выбора команд в генераторе кода.
- Схема трансляции задается набором *правил свертки*
- Каждое правило содержит:
  - поддерево, соответствующее рассматриваемому шаблону
  - метку узла, на который заменяется поддерево при свертке
  - команды, которые помещаются в объектную программу при свертке

**Пример правила:** правило для команды сложения одного регистра с другим имеет вид:




**{ADD Ri, Ri, Rj}**

если входное дерево содержит поддерево, соответствующее данному шаблону, то это поддерево можно заменить одним узлом с меткой $R_i$ и сгенерировать команду

**ADD Ri, Ri, Rj.**

Такая замена называется *замещением поддерева*.

# *1.5 Метод переписывания дерева*

## 1.5.3 Схема трансляции дерева

- Листья как входного дерева, так и шаблона могут иметь атрибуты с индексами; иногда к значениям индексов применяется ряд ограничений – семантических предикатов, которым должен удовлетворять шаблон при установлении соответствия.

  **Пример предиката**: значение константы должно находиться в определенном диапазоне.

- Выбор команд ведется в следующих предположениях:
  - Любая инструкция промежуточного кода может быть реализована с помощью одной или нескольких машинных команд.
  - Имеется достаточно регистров для вычисления каждого узла.

# 1.5 Метод переписывания дерева

## 1.5.4 Пример

**Правила свертки**

На рисунке приведены правила свертки для некоторых команд целевой машины:

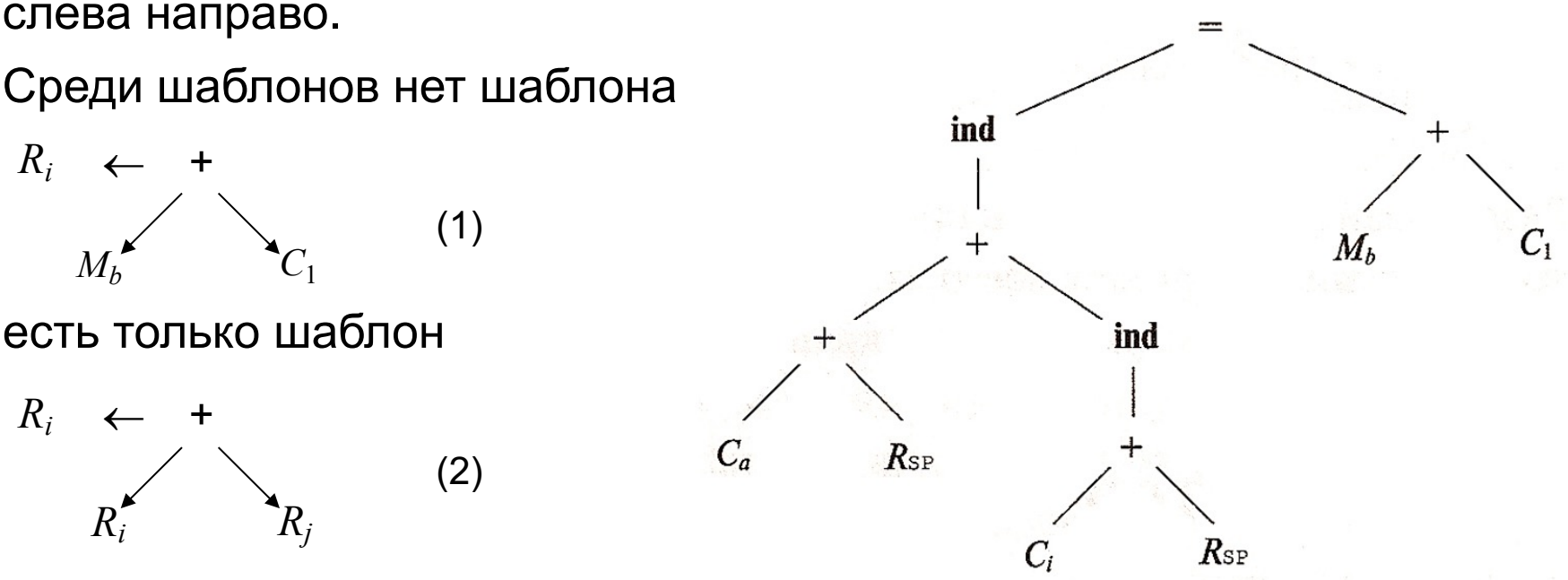
- правила 1) и 2) относятся к инструкциям загрузки (**LD**)
- правила 3) и 4) – инструкциям сохранения(**ST**)
- правила 5) и 6) – индексированным загрузкам
- правила 7) и 8) сложениям (**ADD**)

| | | |
|---|---|---|
| 1) | $R_i \leftarrow C_a$ | {LD Ri, #a} |
| 2) | $R_i \leftarrow M_x$ | {LD Ri, x} |
| 3) | $M_x \leftarrow =$ ($M_x$, $R_i$) | {ST x,Ri} |
| 4) | $M \leftarrow =$ (ind($R_i$), $R_j$) | {ST *Ri, Rj} |
| 5) | $R_i \leftarrow$ ind( $+$ ($C_a$, $R_j$) ) | {LD Ri, a(Rj)} |
| 6) | $R_i \leftarrow +$ ($R_i$, ind( $+$ ($C_a$, $R_j$) )) | {ADD Ri,Ri,a(Rj)} |
| 7) | $R_i \leftarrow +$ ($R_i$, $R_j$) | {ADD Ri,Ri,Rj} |
| 8) | $R_i \leftarrow +$ ($R_i$, $C_1$) | {INC Ri} |

# 1.5 Метод переписывания дерева

## 1.5.4 Пример

Начнем применять схему трансляции дерева к дереву из примера 1.5.1 слева направо.




Среди шаблонов нет шаблона




(1)

есть только шаблон




(2)

следовательно, чтобы свернуть шаблон (1) необходимо с помощью правил 1) и 2) привести его к виду (2)

правило 1) $R_i \leftarrow C_a$ **{LD Ri #a}** позволяет заменить $C_1$ на $R_0$

правило 2) $R_i \leftarrow M_x$ **{LD Ri x}** позволяет заменить $M_b$ на $R_1$

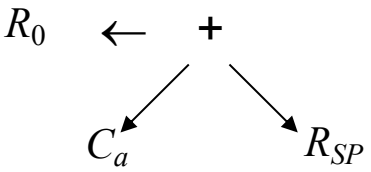
после этих замен шаблон принимает вид (2), что позволяет применить правило 7)

# 1.5 Метод переписывания дерева

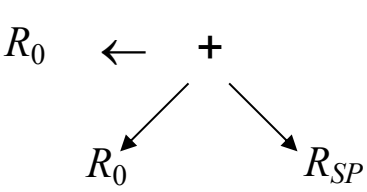
## 1.5.4 Пример

Применим приведенную схему трансляции дерева для генерации кода для примера 1.5.1.

Правило (1) позволяет привести шаблон




к виду




выдав команду

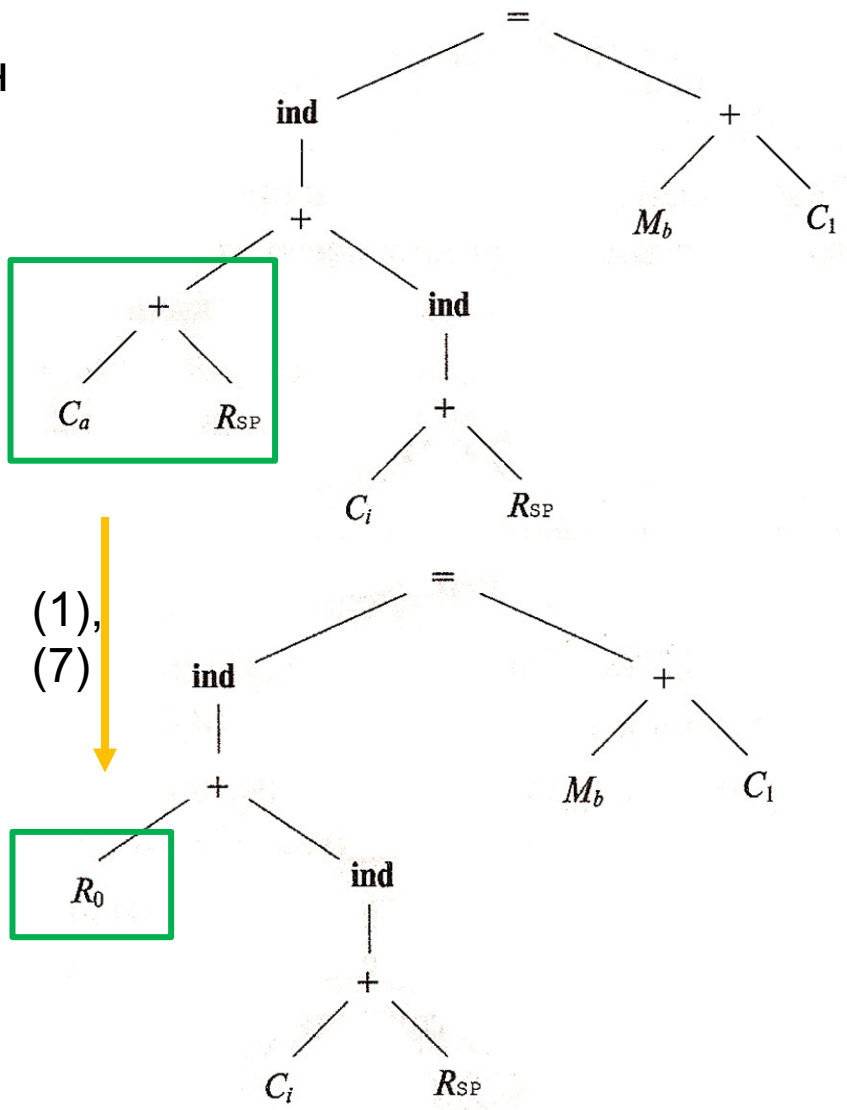
```
{LD R0 #a}
```

Теперь можно с помощью правила (7) привести исходное дерево к виду (нижний рисунок) и выдать команду

```
{ADD R0 R0 SP}
```

Итак, дерево приведено к виду (нижний рисунок) и выданы команды:

```
LD  R0 #a
ADD R0 R0 SP
```

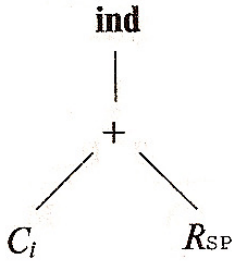


# *1.5 Метод переписывания дерева*

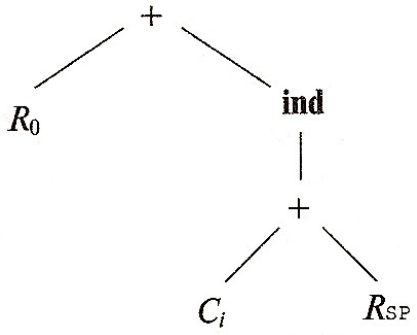
## 1.5.4 Пример

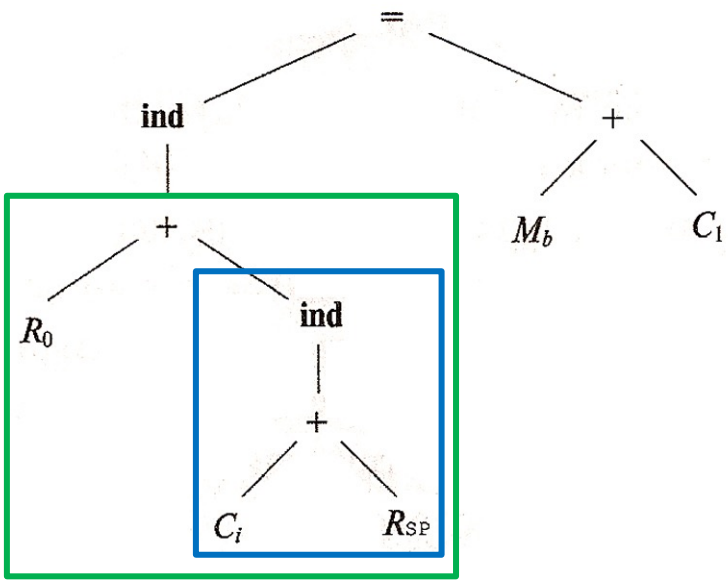
В левом поддереве дерева (верхний рисунок слева)

можно применить правило 5) к шаблону



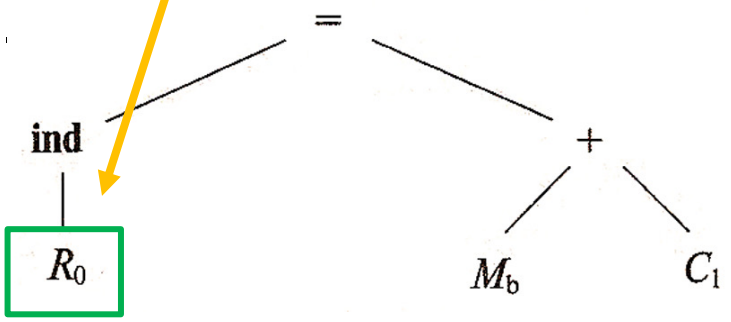

либо правило 6) к шаблону




Выгоднее применить правило 6), так как оно сворачивает большую часть дерева.

После применения правила 6) дерево примет вид (справа):



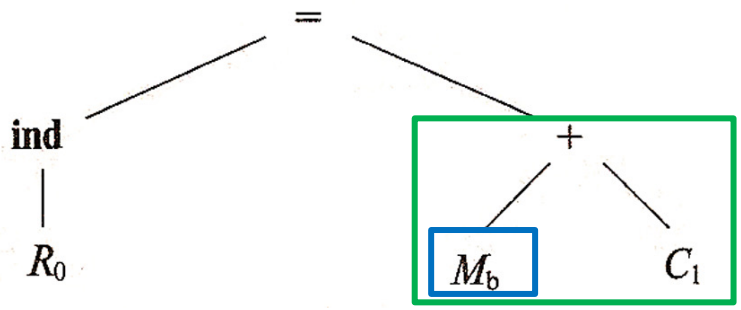

при этом будет выдана команда

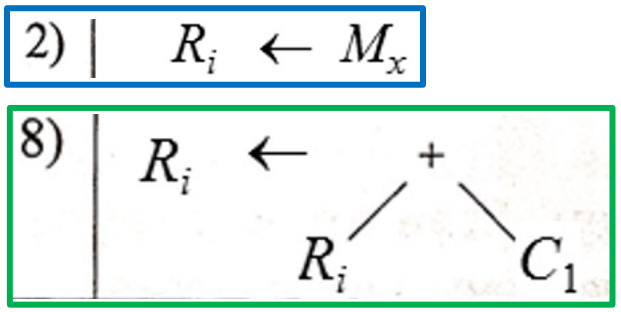
```
ADD R0 R0 i(SP)
```

# 1.5 Метод переписывания дерева

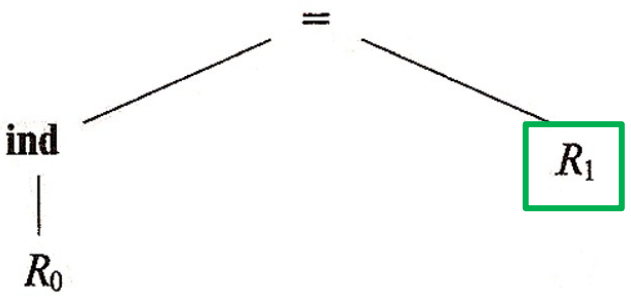
## 1.5.4 Пример

Применив к правому поддереву дерева




сначала правило 2), а потом правило 8), получим дерево



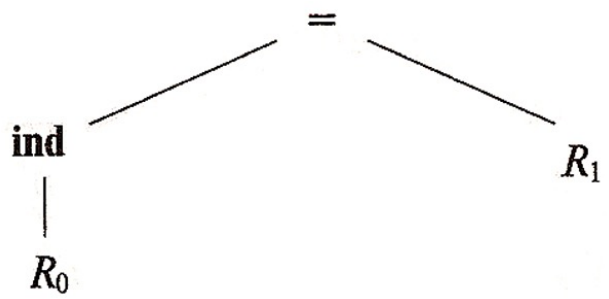

Будет выдано две команды:

```
LD   R1 b     (правило 2))
INC  R1       (правило 8))
```

# 1.5 Метод переписывания дерева

## 1.5.4 Пример

Последнее дерево




соответствует поддереву из правила 4); применение этого правила сворачивает дерево в единственный узел и выдает команду **`ST *R0 R1`**

Таким образом в процессе свертки исходного дерева была сгенерирована последовательность команд:

```
LD  R0  #a
ADD R0  R0 SP
ADD R0  R0 i(SP)
LD  R1  b
INC R1
ST  *R0 R1
```

# 1.5 Метод переписывания дерева

## 1.5.5 Проблемы

- Если на каком-нибудь шаге не будет найдено соответствия ни одному шаблону, процесс генерации кода блокируется.
- Для реализации процесса свертки, показанного на примере, необходимо решить два вопроса, связанных с поиском соответствия деревьев шаблону:
  - *Каким образом выполняется поиск соответствия?*

    Эффективность процесса генерации кода в процессе компиляции зависит от того, насколько эффективен применяющийся алгоритм поиска соответствий.
  - *Что делать, если оказалось, что можно применить более одного шаблона?*

    Эффективность сгенерированного кода может зависеть от порядка, в котором выявлялось соответствие шаблонам, так как различные последовательности соответствий будут приводить к разным последовательностям машинных команд, одни из которых более эффективны, чем другие.

# 1.5 Метод переписывания дерева

## 1.5.6 Поиск соответствий

◊ Правила преобразования дерева и входное дерево можно представить в виде строк, используя префиксную польскую запись: сначала записывается операция (корень поддерева), потом ее первый операнд (левое поддерево), потом второй операнд (правое поддерево).

◊ Для рассматриваемого примера правила преобразования дерева будут представлены в виде:

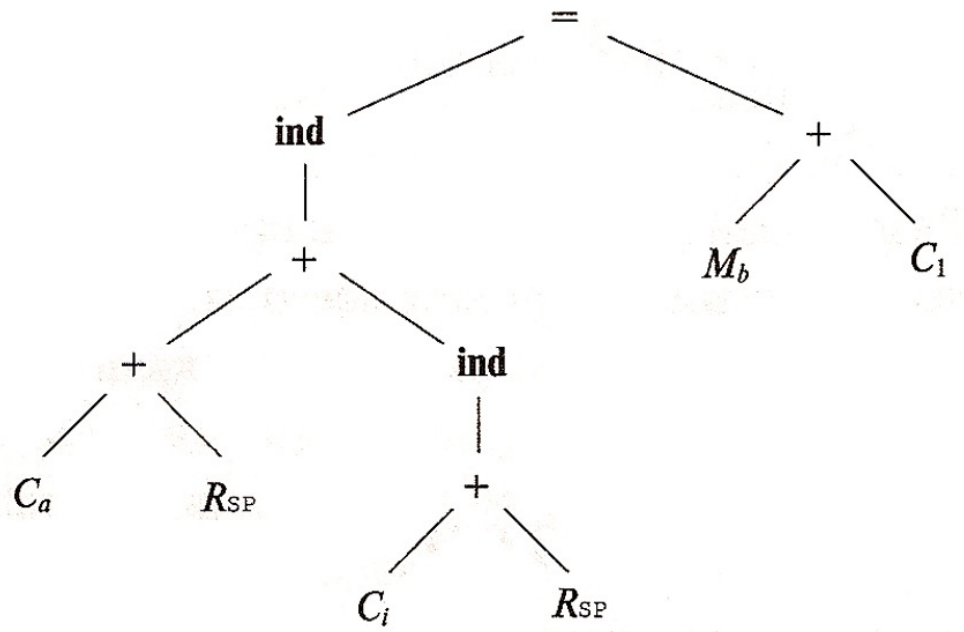
| | | |
|---|---|---|
| 1) | $R_i \rightarrow \mathbf{c}_a$ | `{ LD Ri #a }` |
| 2) | $R_i \rightarrow M_x$ | `{ LD Ri x }` |
| 3) | $M \rightarrow = M_x\ R_i$ | `{ ST x Ri }` |
| 4) | $M \rightarrow = \mathbf{ind}\ R_i\ R_j$ | `{ ST *Ri Rj}` |
| 5) | $R_i \rightarrow \mathbf{ind} + \mathbf{c}_a\ R_j$ | `{ LD Ri a(Rj)}` |
| 6) | $R_i \rightarrow + R_i\ \mathbf{ind} + \mathbf{c}_a\ R_j$ | `{ ADD Ri Ri a(Rj)}` |
| 7) | $R_i \rightarrow + R_i\ R_j$ | `{ ADD Ri Ri Rj }` |
| 8) | $R_i \rightarrow + R_i\ \mathbf{c}_1$ | `{ INC Ri }` |
| 9) | $R \rightarrow \mathbf{sp}$ | |
| 10) | $M \rightarrow \mathbf{m}$ | |

# *1.5 Метод переписывания дерева*

## 1.5.6 Поиск соответствий

◊ Дерево из примера 1.5.4 в префиксном представлении будет задаваться следующей строкой:

$$= \mathbf{ind} \; + \; + \; C_a \; R_{SP} \; \mathbf{ind} + C_i \; R_{SP} + M_b \; C_1$$



# 1.5 Метод переписывания дерева

## 1.5.7 Поиск соответствий с помощью синтаксического анализа

◊ Содержимое второго столбца таблицы можно рассматривать как продукции $LR$-грамматики. Поэтому соответствия можно искать с помощью $LR$-анализатора, используемого при синтаксическом анализе

| | | |
|---|---|---|
| 1) | $R_i \rightarrow \mathbf{c}_a$ | `{ LD Ri #a }` |
| 2) | $R_i \rightarrow M_x$ | `{ LD Ri x }` |
| 3) | $M \rightarrow = M_x\ R_i$ | `{ ST x Ri }` |
| 4) | $M \rightarrow = \mathbf{ind}\ R_i\ R_j$ | `{ ST *Ri Rj}` |
| 5) | $R_i \rightarrow \mathbf{ind} + \mathbf{c}_a\ R_j$ | `{ LD Ri a(Rj)}` |
| 6) | $R_i \rightarrow + R_i\ \mathbf{ind} + \mathbf{c}_a\ R_j$ | `{ ADD Ri Ri a(Rj)}` |
| 7) | $R_i \rightarrow + R_i\ R_j$ | `{ ADD Ri Ri Rj }` |
| 8) | $R_i \rightarrow + R_i\ \mathbf{c}_1$ | `{ INC Ri }` |
| 9) | $R \rightarrow \mathbf{sp}$ | |
| 10) | $M \rightarrow \mathbf{m}$ | |

# *1.5 Метод переписывания дерева*

## 1.5.7 Поиск соответствий с помощью синтаксического анализа

◊ На основе продукций схемы трансляции можно построить $LR$-анализатор: стековый автомат, который загоняет символы анализируемой строки в стек (*перенос*) пока в голове стека не получится строка, являющаяся правой частью одной из продукций. Когда это происходит выполняется *свертка*: правая часть найденной продукции удаляется из стека и вместо нее в стек помещается левая часть этой продукции. При каждой свертке генерируется машинная команда из соответствующей строки третьего столбца таблицы.

◊ Обычно грамматика генерации кода неоднозначна и поэтому должны быть предприняты меры по разрешению конфликтов. При отсутствии информации о стоимости общее правило состоит в предпочтении больших сверток меньшим.
Это означает, что в случае конфликта «свертка – свертка» выбирается более длинная свертка, а при конфликте «перенос – свертка» – перенос. Такой подход обеспечивает выполнение большего числа операций с помощью одной машинной команды.

# *1.5 Метод переписывания дерева*

## 1.5.7 Поиск соответствий с помощью синтаксического анализа

- Преимущества использования $LR$-анализа для генерации кода:
  - Методы синтаксического анализа эффективны и хорошо изучены, использование алгоритмов $LR$-анализа обеспечивает надежность и эффективность генератора кода.
  - Облегчается перенастройка генератора кода для другой целевой машины, так как создание генератора кода для новой машины сводится к разработке грамматики, описывающей ее команды.
  - Качество генерируемого кода может быть сделано весьма высоким за счет добавления продукций для особых случаев, чтобы использовать преимущества машинных идиом.

# *1.5 Метод переписывания дерева*

## 1.5.8 Проверка атрибутов

◊ В схеме трансляции для генерации кода встречаются ограничения на значения атрибутов. Такие ограничения могут возникнуть в связи с машинными идиомами.

◊ Машинная команда может требовать, чтобы значения атрибута находились в определенном диапазоне или чтобы два значения атрибутов были связаны некоторым соотношением.

Такие ограничения на значения атрибутов могут быть указаны как предикаты, проверяемые перед выполнением свертки. Использование предикатов может обеспечить большую гибкость и простоту описания по сравнению с чисто грамматической спецификацией генератора кода.

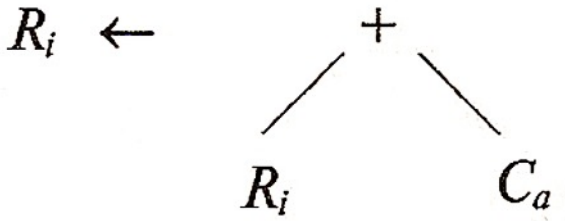
◊ Некоторые аспекты архитектуры целевой машины, например режимы адресации, могут быть выражены с помощью ограничений на атрибуты. Это позволяет сократить описание целевой машины.

# *1.5 Метод переписывания дерева*

## 1.5.7 Проверка атрибутов

◊ В случае использования *обобщенных шаблонов* для выбора команд в частных случаях могут использоваться семантические проверки

◊ **Пример**.
С помощью одного обобщенного шаблона могут быть представлены два варианта команд сложения: **`ADD`** и **`INC`**.
Для выбора команды используется *семантическая проверка*.



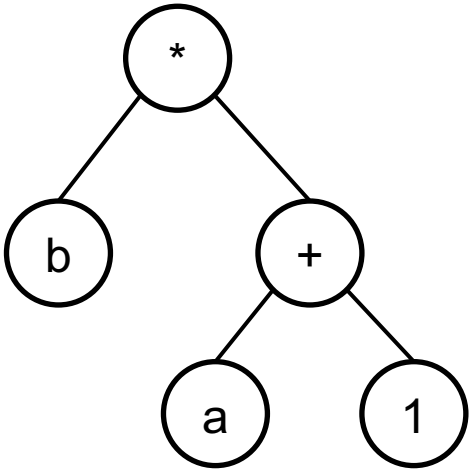

Обобщенный шаблон

```
if (a = 1)
    INC  Ri
else
    ADD  Ri, Ri, #a
```

Семантическая проверка

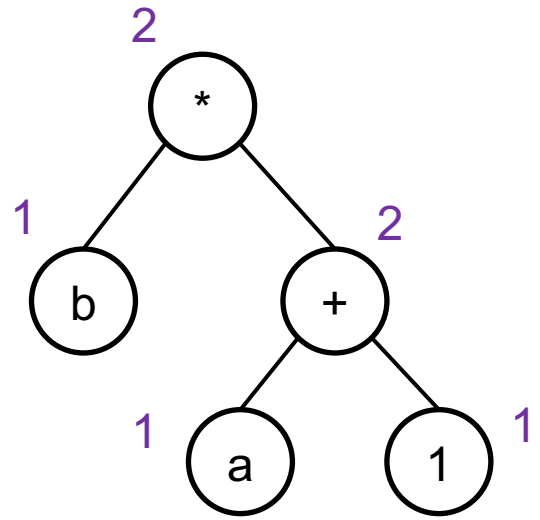
# *1.6 Генерация оптимального кода для выражений*

## 1.6.1 Разметка деревьев выражений. Числа Ершова



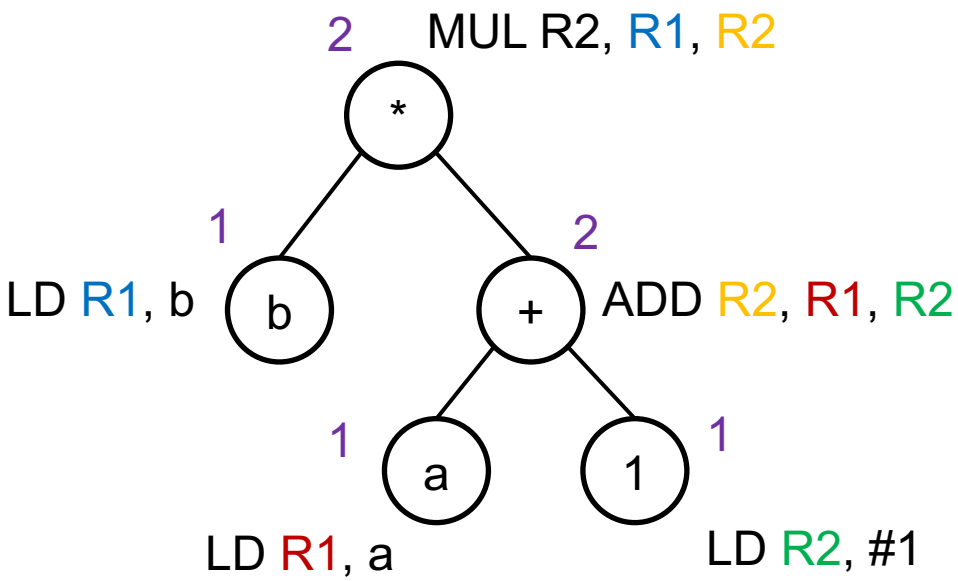
# *1.6 Генерация оптимального кода для выражений*

## 1.6.1 Разметка деревьев выражений. Числа Ершова



# *1.6 Генерация оптимального кода для выражений*

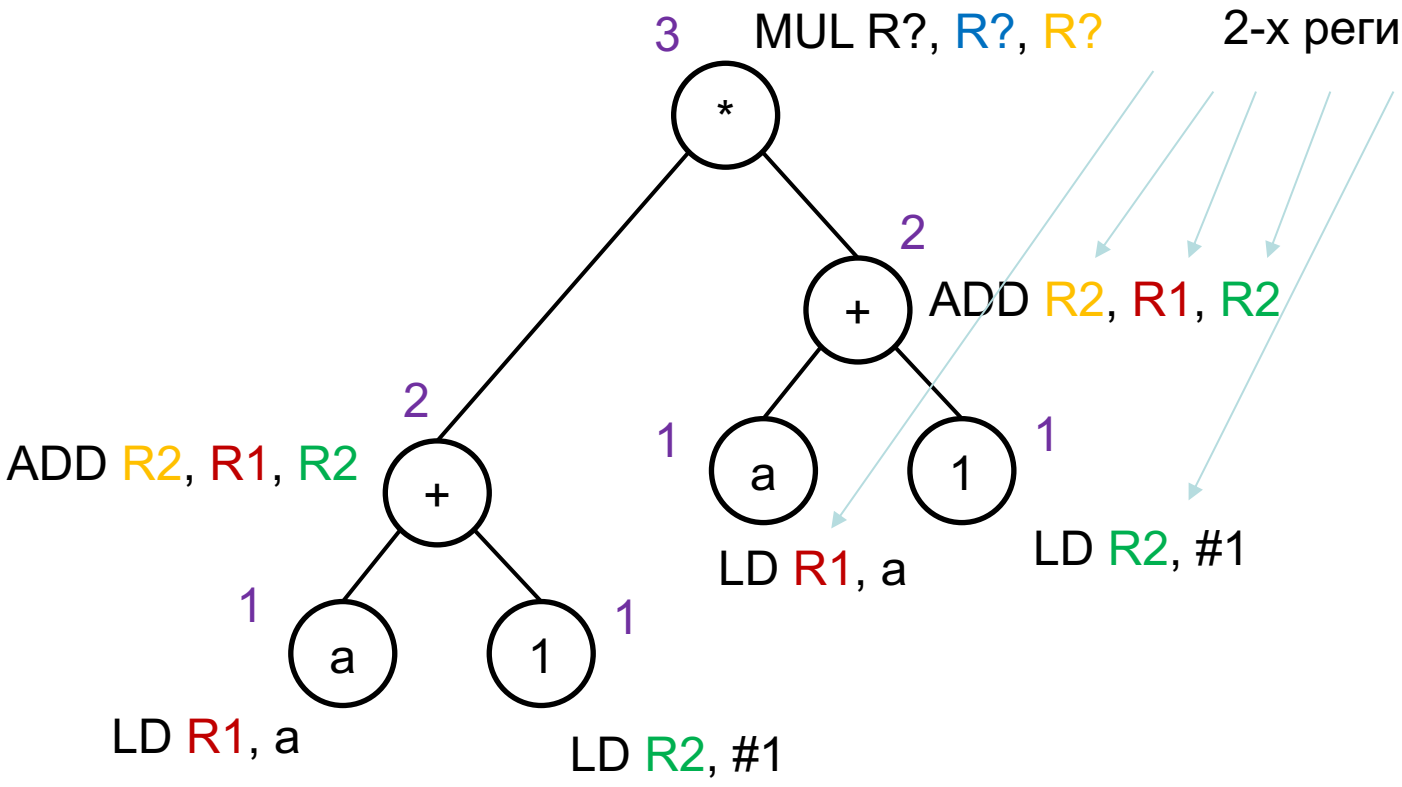
## 1.6.1 Разметка деревьев выражений. Числа Ершова



# *1.6 Генерация оптимального кода для выражений*

## 1.6.1 Разметка деревьев выражений. Числа Ершова

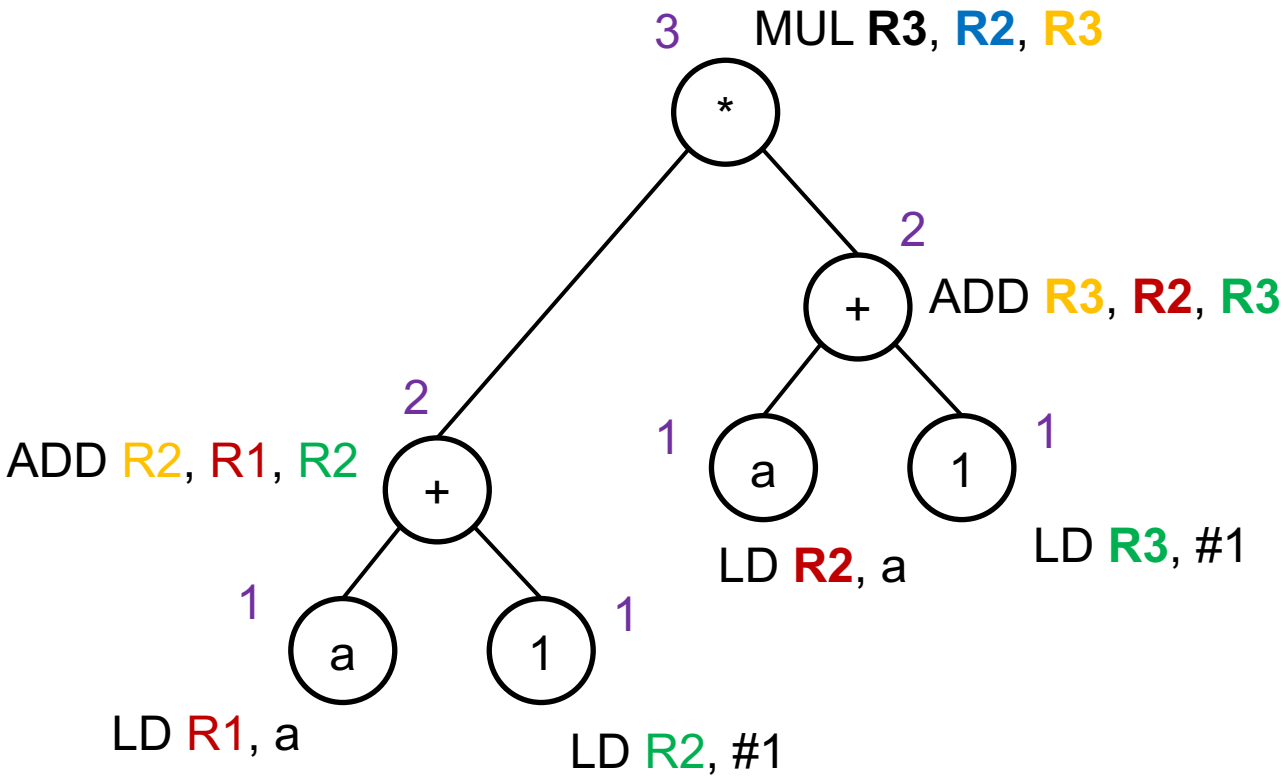
Как изменятся номера регистров, если слева будет такое же поддерево, требующее 2-х регистров?



# *1.6 Генерация оптимального кода для выражений*

## 1.6.1 Разметка деревьев выражений. Числа Ершова

Как изменятся номера регистров, если слева будет такое же поддерево, требующее 2-х регистров?



# *1.6 Генерация оптимального кода для выражений*

## 1.6.1 Разметка деревьев выражений. Числа Ершова

◊ *Числа Ершова* назначаются каждому узлу дерева выражения и указывают, **сколько регистров требуется для вычисления этого узла без сохранения временных переменных.**

◊ Числа Ершова назначаются по следующим правилам:

1. Все листья имеют метку 1 (кроме $R_{SP}$, т.к. в этом случае никаких дополнительных регистров этому узлу выделять не нужно).
2. Метка внутреннего узла с одним дочерним узлом равна метке дочернего узла.
3. Метка внутреннего узла с двумя дочерними равна:
   (а) если метки дочерних узлов различны – наибольшей из меток дочерних узлов;
   (б) если метки дочерних узлов совпадают – метке дочернего узла, увеличенной на 1.

# *1.6 Генерация оптимального кода для выражений*

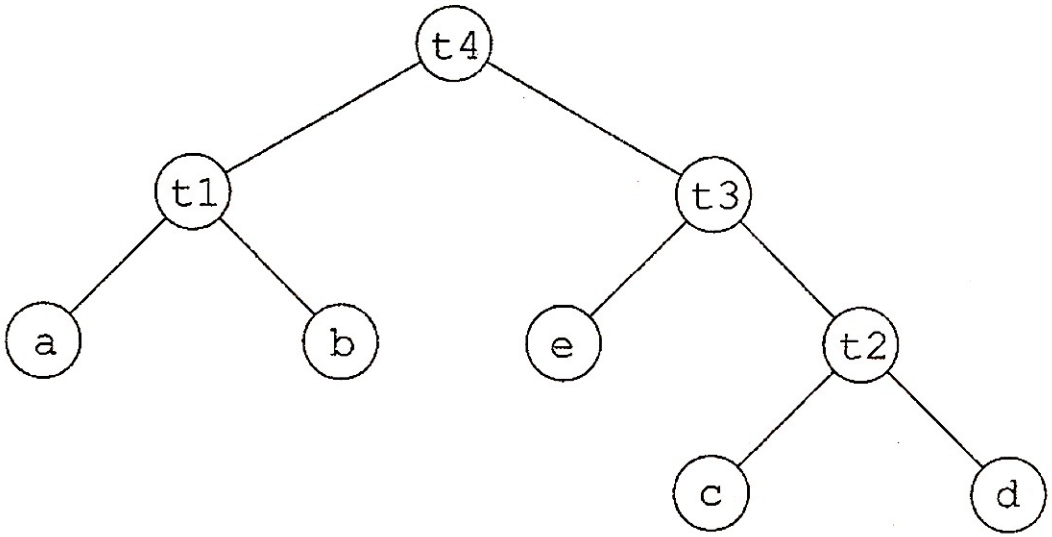
## 1.6.1 Разметка деревьев выражений. Числа Ершова

◊ **Пример**. Трехадресный код, соответствующий выражению

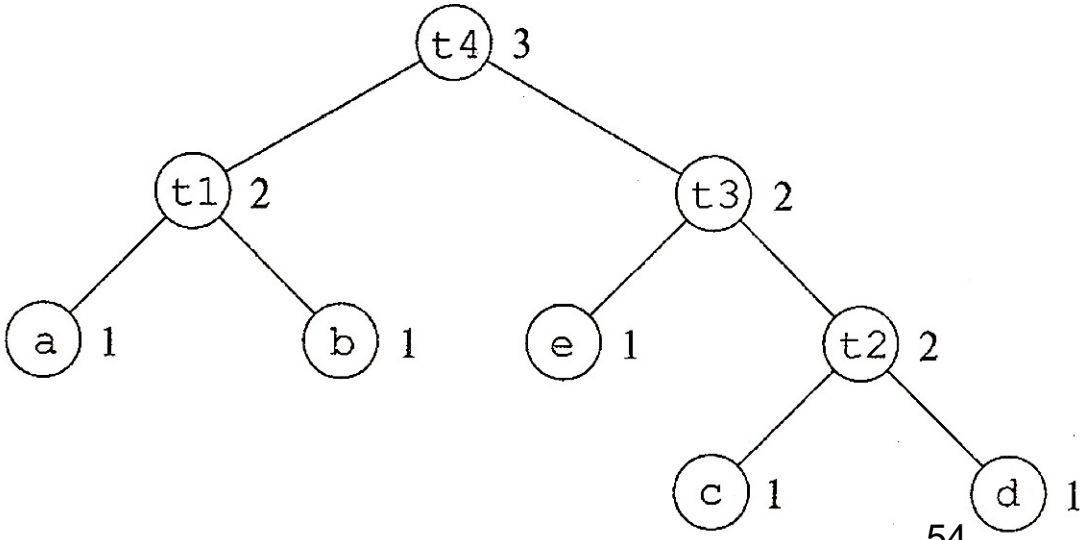
$(a-b)+e\times(c+d)$:

```
t1 = a  - b
t2 = c  + d
t3 = e  * t2
t4 = t1 + t3
```

можно представить в виде дерева на верхнем рисунке




◊ Разметив узлы этого дерева в соответствии со сформулированными правилами, получим дерево на нижнем рисунке



# *1.6 Генерация оптимального кода для выражений*

## 1.6.2 Алгоритм генерации кода для размеченных деревьев выражений

◊ **Вход**: размеченное дерево, в котором каждый операнд появляется по одному разу (т.е. отсутствуют общие подвыражения).

◊ **Выход**: оптимальная последовательность машинных команд для вычисления значения корня в регистре.

◊ **Метод**: *рекурсивный алгоритм* генерации машинного кода (на следующем слайде).

◊ В алгоритме имеется "база" $b \geq 1$ для используемых регистров, т.е. фактически используемыми регистрами будут $R_b$, $R_{b+1}$, ..., $R_{b+k-1}$

◊ Т.е. если алгоритм применяется к узлу с меткой $k$, то будут использованы ровно $k$ регистров: $R_b$, $R_{b+1}$, ..., $R_{b+k-1}$.
Результат всегда будет помещаться в регистр $R_{b+k-1}$.

# *1.6 Генерация оптимального кода для выражений*

## 1.6.2 Алгоритм генерации кода для размеченных деревьев выражений

1. Генерация машинного кода для внутреннего узла с меткой $k$ и двумя дочерними узлами с **равными** метками (в этом случае метки обоих дочерних узлов равны $k-1$):

   (a) Рекурсивная генерация кода для правого дочернего узла с базой *b + 1*, используя регистры $R_{b+1}$, …, $R_{b+k-1}$. Результат правого дочернего узла помещается в регистр $R_{b+k-1}$.

   (b) Рекурсивная генерация кода с базой *b* для левого дочернего узла, используя регистры $R_b$, …, $R_{b+k-2}$. Результат левого дочернего узла помещается в регистр $R_{b+k-2}$.

   (c) Генерация команды

   **OP** $R_{b+k-1}$, $R_{b+k-2}$, $R_{b+k-1}$,

   где **OP** – операция в узле с меткой $k$.

# *1.6 Генерация оптимального кода для выражений*

## 1.6.2 Алгоритм генерации кода для размеченных деревьев выражений

2. Генерация кода для внутреннего узла с меткой $k$ и двумя дочерними узлами с **разными** метками (в этом случае один из дочерних узлов («*большой*») имеет метку $k$, а второй («*маленький*») – метку $m < k$):

   (a) Рекурсивная генерация кода для ***большого*** узла с базой *b*, используя $k$ регистров $R_b$, ..., $R_{b+k-1}$; результат получается в регистре $R_{b+k-1}$.

   (b) Рекурсивная генерация кода для ***маленького*** узла, с базой *b*, используя $m$ регистров $R_b$, ..., $R_{b+m-1}$; результат находится в регистре $R_{b+m-1}$.
   $m < k$, следовательно при вычислениях не используются ни регистр $R_{b+m}$, ни какой-либо иной регистр с большим номером.

   (c) Генерируем команду
   **OP** $R_{b+k-1}$, $R_{b+m-1}$, $R_{b+k-1}$ или **OP** $R_{b+k-1}$, $R_{b+k-1}$, $R_{b+m-1}$
   в зависимости от того, является большой узел правым или левым.

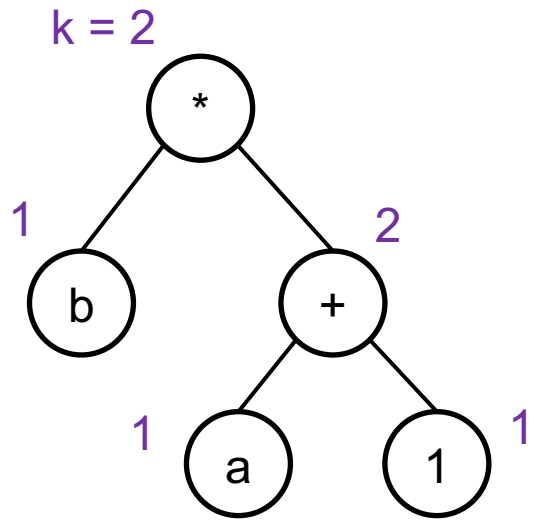
### 1.6.2 Алгоритм генерации кода для размеченных деревьев выражений

3. Для листа, представляющего операнд $x$, при использовании базы $b$ генерируем команду **LD** $R_b$, $x$.

# *1.6 Генерация оптимального кода для выражений*

## 1.6.1 Разметка деревьев выражений. Числа Ершова

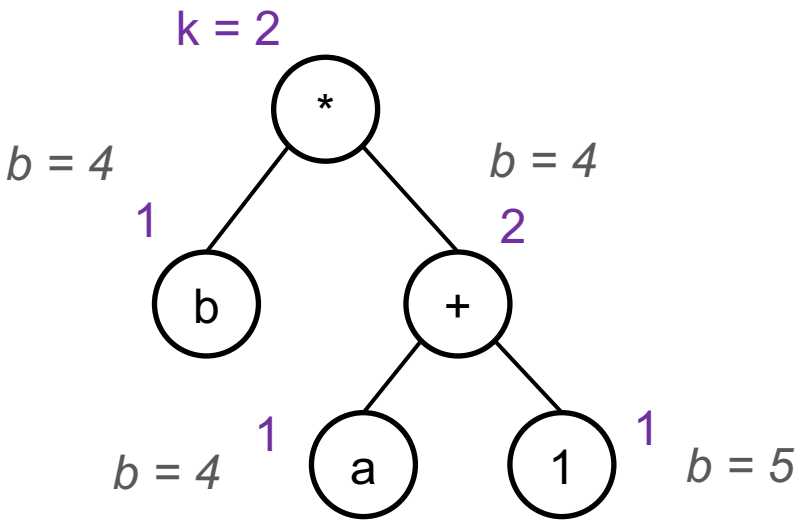
**Рассмотрим *поддерево* некоторого выражения, и пусть в данный узел мы пришли со значением *b = 4*:**



# *1.6 Генерация оптимального кода для выражений*

## 1.6.1 Разметка деревьев выражений. Числа Ершова

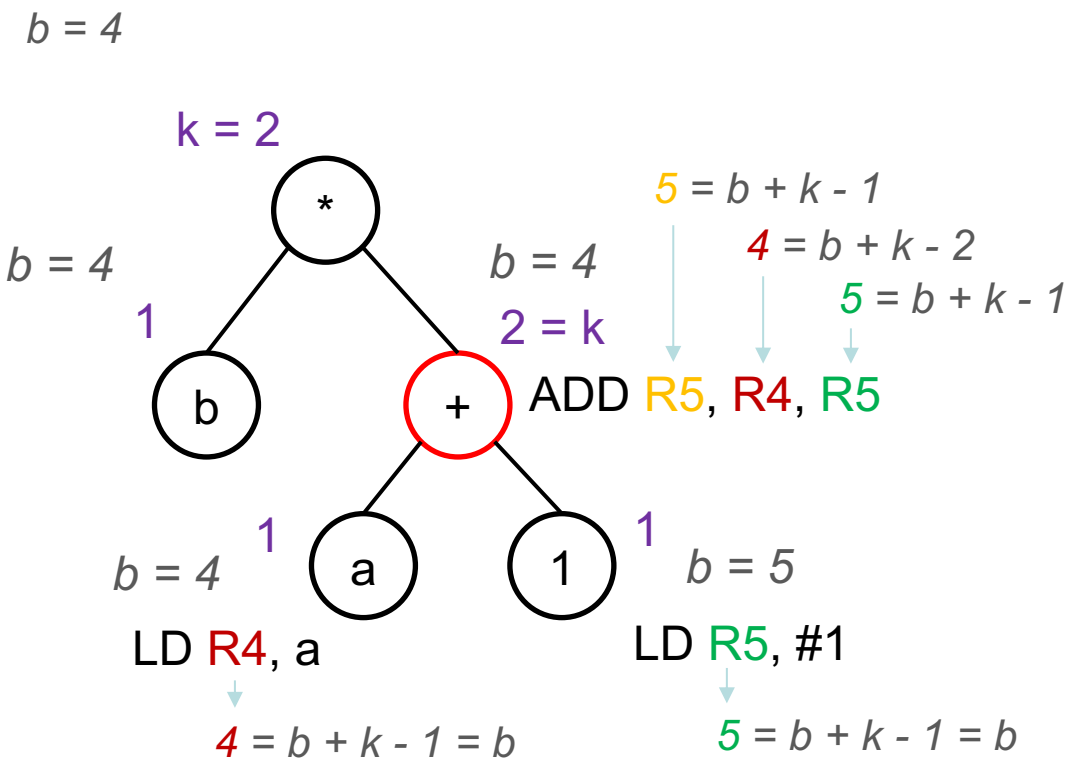
(пусть изначально пришли в узел (*) с *b = 4*)




сначала вычислим базу *b* для каждого узла:
*b* увеличивается на узлах с равными метками для правой ветви

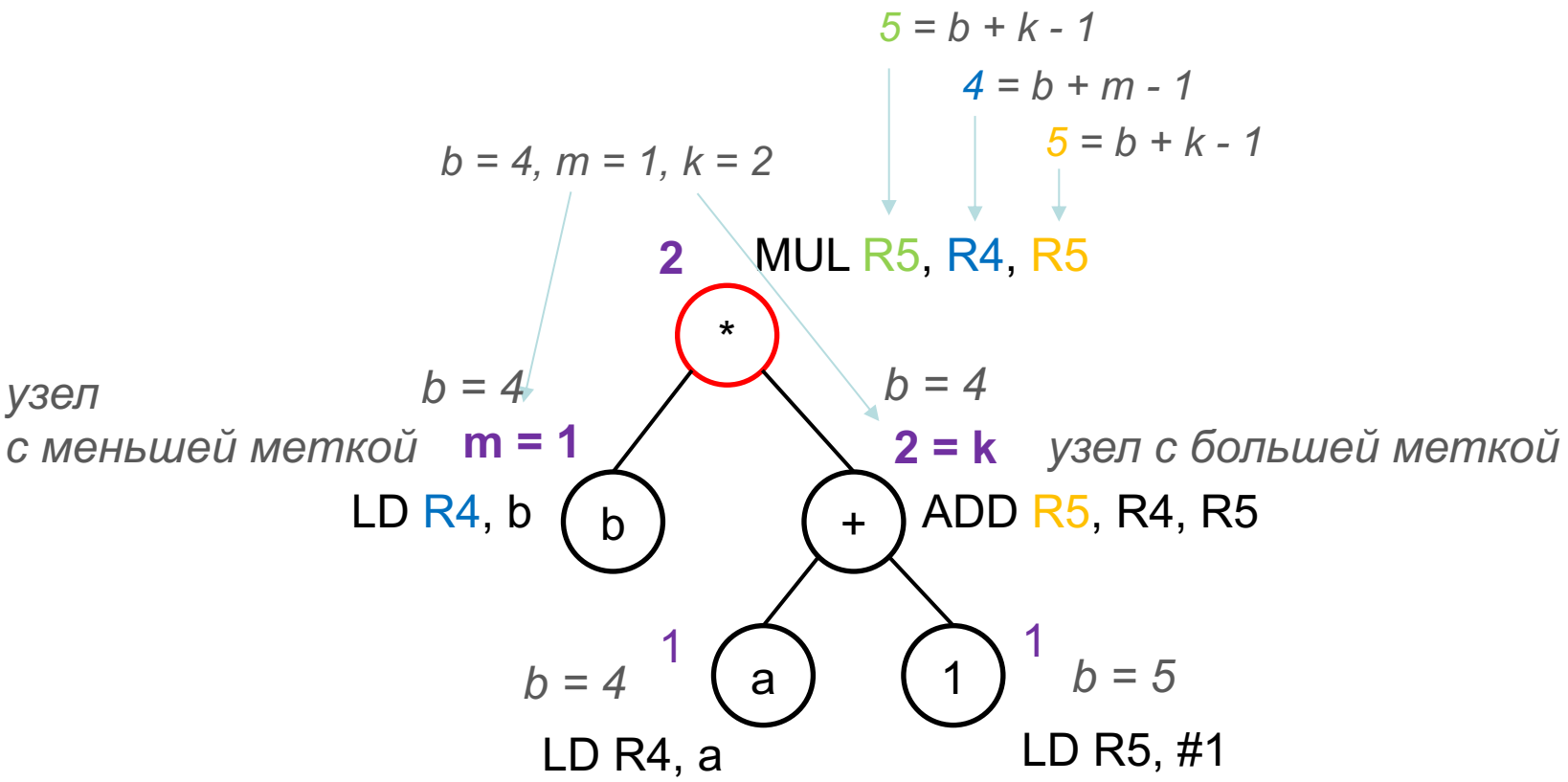
# *1.6 Генерация оптимального кода для выражений*

## 1.6.1 Разметка деревьев выражений. Числа Ершова



# *1.6 Генерация оптимального кода для выражений*

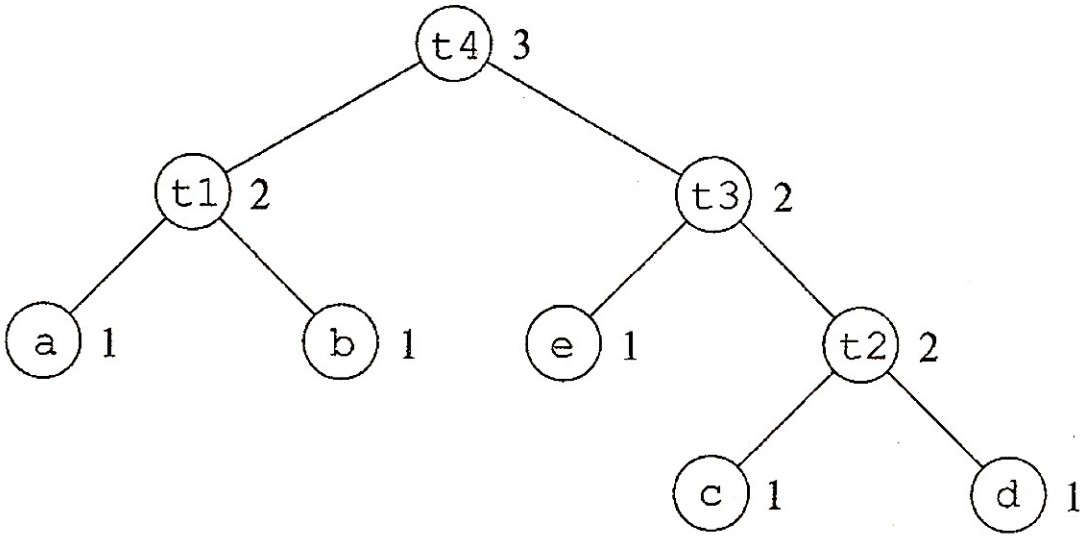
## 1.6.1 Разметка деревьев выражений. Числа Ершова



# *1.6 Генерация оптимального кода для выражений*

## 1.6.3 Пример применения алгоритма генерации кода для размеченных деревьев выражений

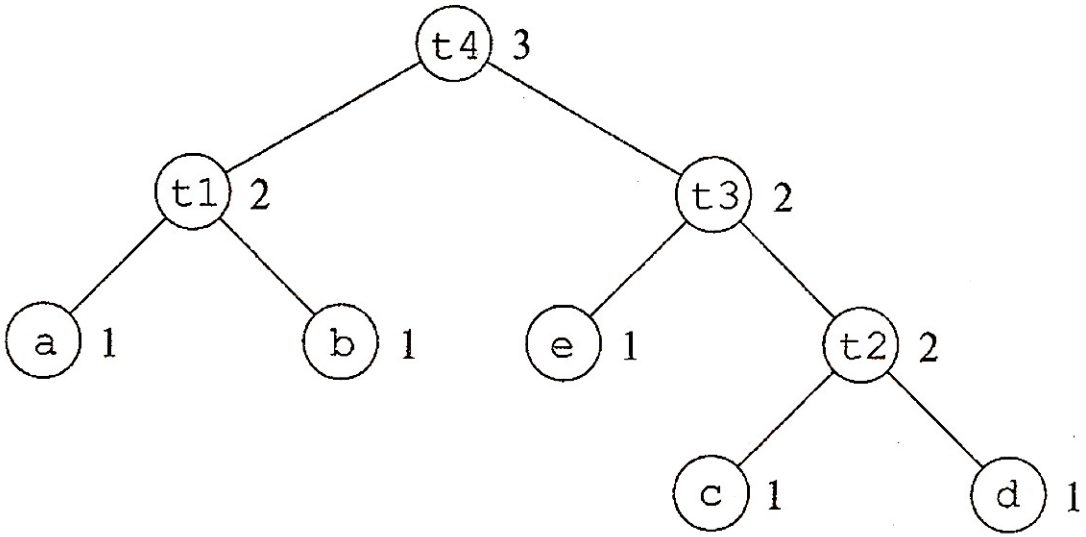
◊ Применим описанный алгоритм к дереву на рисунке. Поскольку метка корня равна 3, результат получится в регистре **R3**, а использоваться при вычислениях будут только регистры **R1**, **R2** и **R3**.




◊ 1) Рассматривается корень **t4**. У него два дочерних узла **t1** и **t3** с одинаковыми метками, следовательно должен работать п.1 алгоритма.

# *1.6 Генерация оптимального кода для выражений*

## 1.6.3 Пример применения алгоритма генерации кода для размеченных деревьев выражений



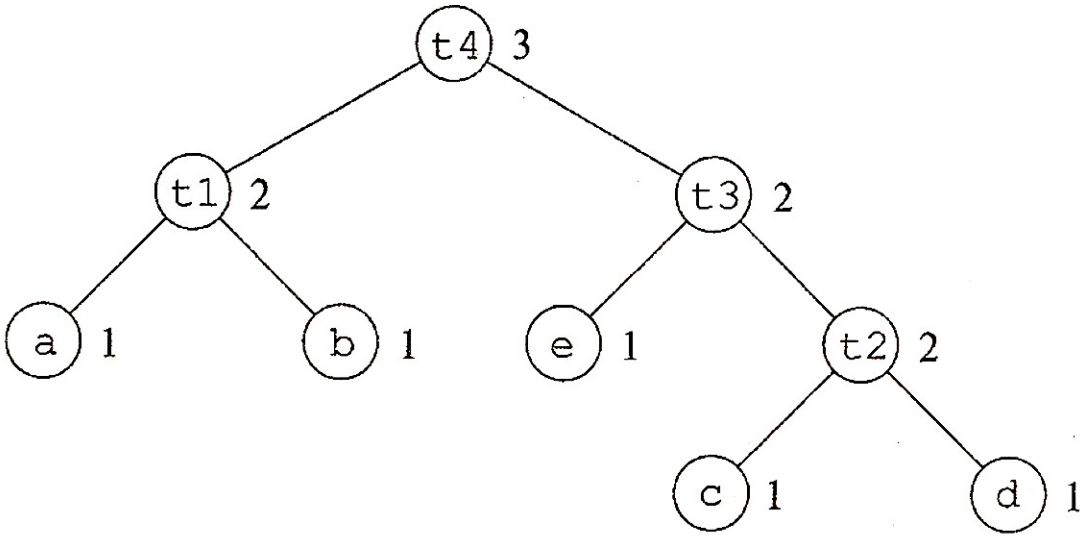

◊ 2) Согласно п.1 (а) сначала нужно сгенерировать код для правого поддерева (с корнем **t3**).

У него правый узел большой, а левый – маленький, следовательно должен работать п.2 алгоритма.

3) Согласно п.2 (а) сначала генерируется код для большого поддерева (с корнем **t2**).

# *1.6 Генерация оптимального кода для выражений*

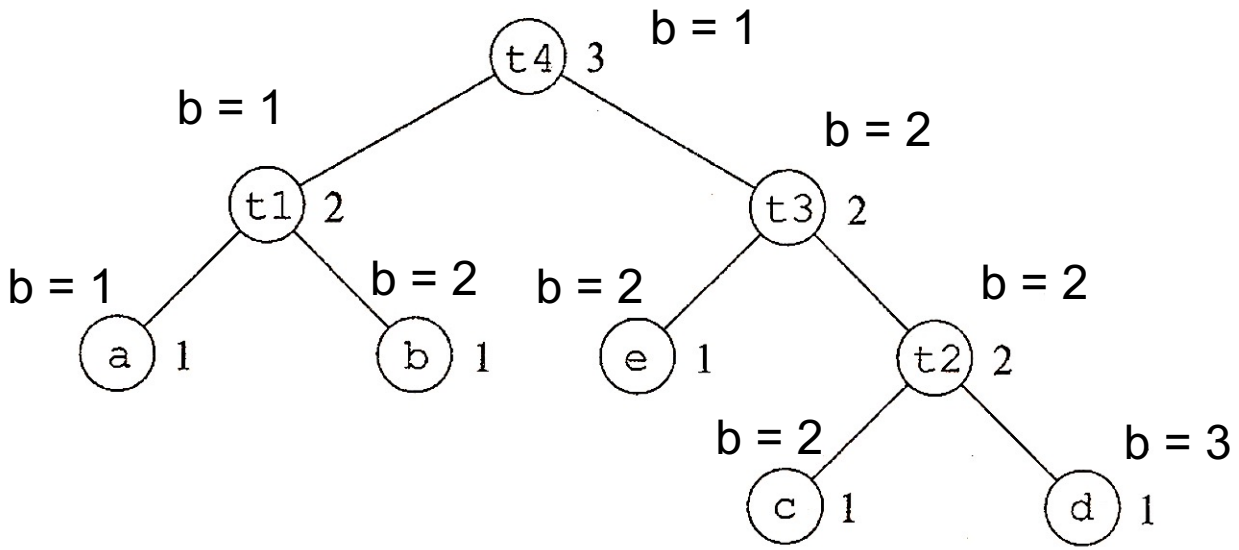
## 1.6.3 Пример применения алгоритма генерации кода для размеченных деревьев выражений




◊ 4) Рассматривается корень **t2**. У него два дочерних листовых узла с одинаковыми метками, следовательно должен работать п.1 алгоритма.

# *1.6 Генерация оптимального кода для выражений*

## 1.6.3 Пример применения алгоритма генерации кода для размеченных деревьев выражений

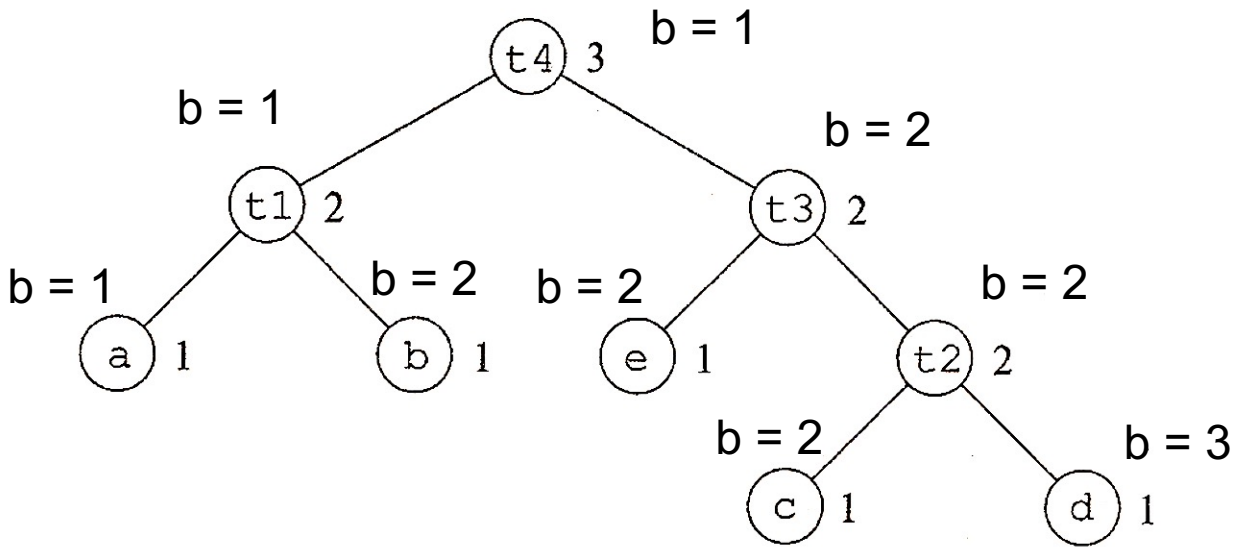



◊ 5) Сначала согласно п.1 (a) нужно сгенерировать код для **правого** листа, потом согласно п.1 (b) нужно сгенерировать код для **левого** листа, потом согласно п.1 (c) нужно сгенерировать **команду**. В результате получатся следующие три команды:

```
LD   R3, d
LD   R2, c
ADD  R3, R2, R3
```

# *1.6 Генерация оптимального кода для выражений*

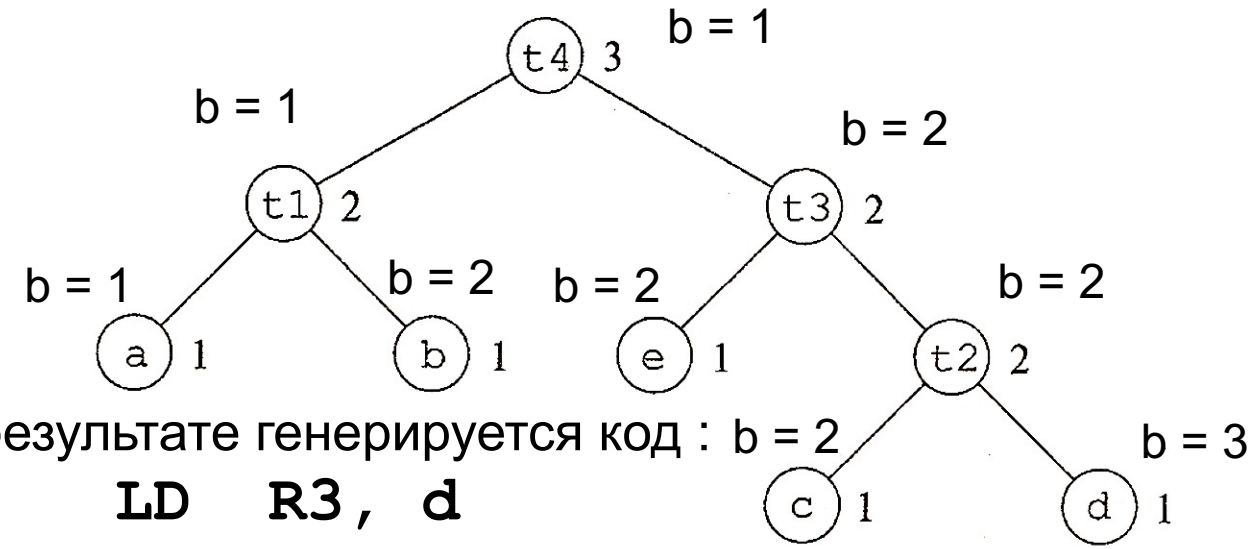
## 1.6.3 Пример применения алгоритма генерации кода для размеченных деревьев выражений




◊ 6) Далее генерируются команды для:
левого поддерева дерева с корнем **t3**,
для корня **t3**,
для левого поддерева дерева с корнем **t4**
и для всего дерева с корнем **t4**

# *1.6 Генерация оптимального кода для выражений*

## 1.6.3 Пример применения алгоритма генерации кода для размеченных деревьев выражений

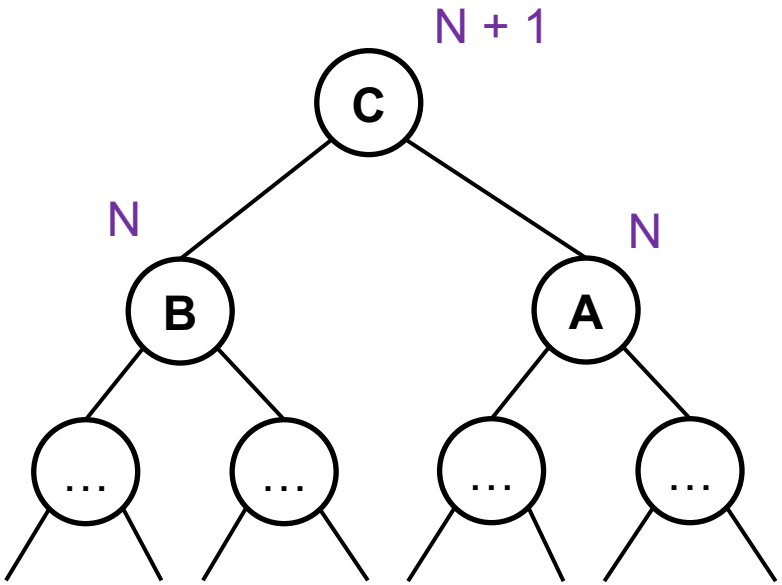



◊ В результате генерируется код :

```
LD  R3, d
LD  R2, c
ADD R3, R2, R3
LD  R2, e
MUL R3, R2, R3
LD  R2, b
LD  R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

# 1.6 Генерация оптимального кода для выражений

## 1.6.4 Вычисление выражений при недостаточном количестве регистров




1) Вычислить A в регистр $R_N$
2) Сохранить $R_n$ в ячейку памяти $t_A$
3) Вычислить B в регистр $R_N$
4) Загрузить из $t_A$ в регистр $R_{N-1}$
5) Вычислить C, используя $R_{N-1}$ и $R_N$

Рассмотрим случай, когда доступно **N** аппаратных регистров, метка корневого узла выражения равна **N + 1**, а метки его потомков – **N**.

Т.е. оба поддерева по отдельности влезают в **N** регистров, а всё дерево **C** с меткой **N + 1** в корне – уже нет.

В этой конфигурации требуется **N** регистров для вычисления правого поддерева **A**, и столько же для левого **B**, но в момент вычисления **B** значение уже вычисленного правого **A** нужно где-то хранить, и при отсутствии свободных регистров остается его сохранить в память.

Затем, после вычисления левого поддерева **B**, его результат будет занимать всего один регистр, и у нас появится свободный регистр, чтобы загрузить сохраненный ранее результат **A** из памяти.

# *1.6 Генерация оптимального кода для выражений*

## 1.6.4 Вычисление выражений при недостаточном количестве регистров

◊ **Алгоритм генерации кода для размеченного дерева выражения (с учетом конечного числа доступных регистров)**

**Вход**: (1) размеченное дерево, в котором каждый операнд встречается по одному разу (т.е. отсутствуют общие подвыражения)

(2) количество доступных регистров $r \geq 2$

**Выход**: последовательность машинных команд для вычисления значения корня дерева на регистре с использованием не более, чем $r$ регистров

(регистры $R_1, R_2, \ldots, R_r$)

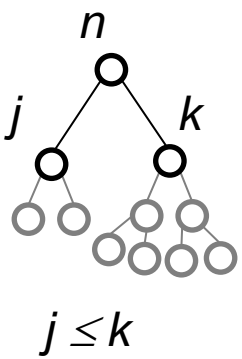
**Метод**: рекурсивный алгоритм генерации машинного кода

# *1.6 Генерация оптимального кода для выражений*

## 1.6.4 Вычисление выражений при недостаточном количестве регистров

◊ **Алгоритм генерации кода для размеченного дерева выражения (с учетом конечного числа доступных регистров)**

**Метод**: рекурсивный алгоритм генерации машинного кода




$j \le k$

Пусть $n$ – метка рассматриваемого узла $N$ (корень рассматриваемого поддерева).

Если $n \le r$, алгоритм совпадает с алгоритмом 1.6.2

Если $n > r$, каждое поддерево обрабатывается отдельно, и результат большого поддерева (оно обрабатывается первым) может быть сохранен в памяти (если $j \ge r$).

Тогда непосредственно перед вычислением $N$ этот результат будет загружен из памяти на регистр $R_{r-1}$ и использован вместе с результатом маленького поддерева (полученного на регистре $R_r$) для вычисления результата $N$ (на регистре $R_r$).

# *1.6 Генерация оптимального кода для выражений*

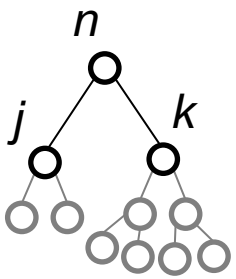
## 1.6.4 Вычисление выражений при недостаточном количестве регистров

◊ Метод

(1) **Выбор большого узла**: Узел $N$ имеет как минимум один дочерний узел с меткой $k \geq r$. Дочерний узел с большей меткой $k$ выбирается как «большой», а узел с меньшей меткой $j$ – как «маленький».

Если метки обоих узлов одинаковы ($k = j$), то в качестве «большого» берется правый дочерний узел.

(2) Рекурсивная генерация кода для большого дочернего узла **с использованием базы *b = 1*** и регистров $R_1, R_2, \ldots, R_r$. Результат вычислений получается на регистре $R_r$.



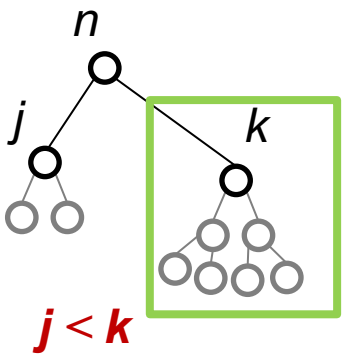

$j \leq k$

# 1.6 Генерация оптимального кода для выражений

## 1.6.4 Вычисление выражений при недостаточном количестве регистров

Пусть $j$ – метка «маленького» узла. Тогда рассмотрим два случая:

(3.1) Если $j < r$, то для маленького дочернего узла используется рассмотренный ранее алгоритм 1.6.2 (генерация кода при наличии достаточного количества регистров), причем используется база $b = r - j$ для генерации кода с использованием регистров $R_{r-j}, \ldots, R_{r-1}$, и результат вычислений «маленького» узла получается на регистре $R_{r-1}$, а результат «большого» остался в $R_r$.




результат правого поддерева с меткой $k \geq r$ будет так или иначе рекурсивно получен в $R_r$
«маленькое» поддерево укладывается в $r - 1$ регистров

# *1.6 Генерация оптимального кода для выражений*

## 1.6.4 Вычисление выражений при недостаточном количестве регистров

Пусть $j$ – метка «маленького» узла. Тогда рассмотрим два случая:

(3.2) Если $j \geq r$, то выполняются следующие шаги:

(3.2.1) Генерация команды **ST $t_n$ $R_r$**, для сохранения результата вычисления «большого» поддерева. Здесь **$t_n$** – временная переменная, где $n$ – метка «верхнего» узла $N$, а переменная **$t_n$** будет переиспользоваться для сохранения промежуточных результатов всех узлов, имеющих метку $n$. После сохранения результата вычисления «большого» поддерева из $R_r$ в память, нам снова доступны $r$ регистров.

(3.2.2) Рекурсивная генерация кода для «маленького» дочернего узла с использованием регистров с базой *b = 1* в регистрах $R_1, \ldots, R_r$. Результат вычислений получается на регистре $R_r$.

(3.2.3) Генерация команды **LD $R_{r-1}$ $t_n$** (загрузка результата «большого» поддерева на $R_{r-1}$).

# *1.6 Генерация оптимального кода для выражений*

## 1.6.4 Вычисление выражений при недостаточном количестве регистров

И, наконец, нужно сгенерировать команду, вычисляющую значение операции для самого узла $N$:

(4) Генерация команды **OP $R_r$ $R_r$ $R_{r-1}$**

(либо в обратном порядке: **OP $R_r$ $R_{r-1}$ $R_r$**

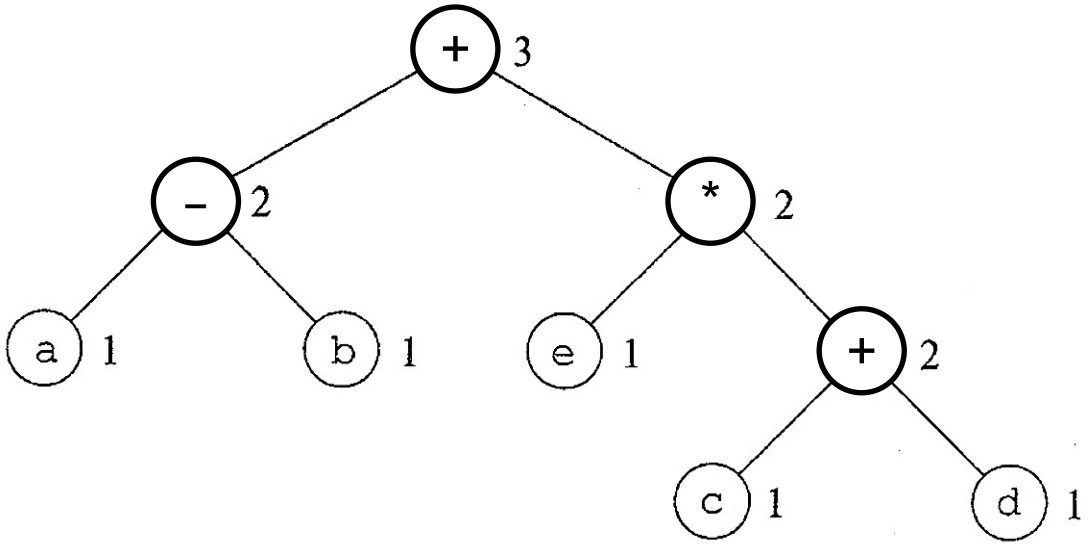
– в зависимости от того, является ли большой узел правым или левым дочерним узлом корня $N$, и потребовалось ли сохранение значения в память – в последнем случае регистры меняются местами).

В любом случае, первый из регистров-аргументов операции должен содержать результат вычисления левого поддерева, а второй – правого.

# *1.6 Генерация оптимального кода для выражений*

## 1.6.5 Пример

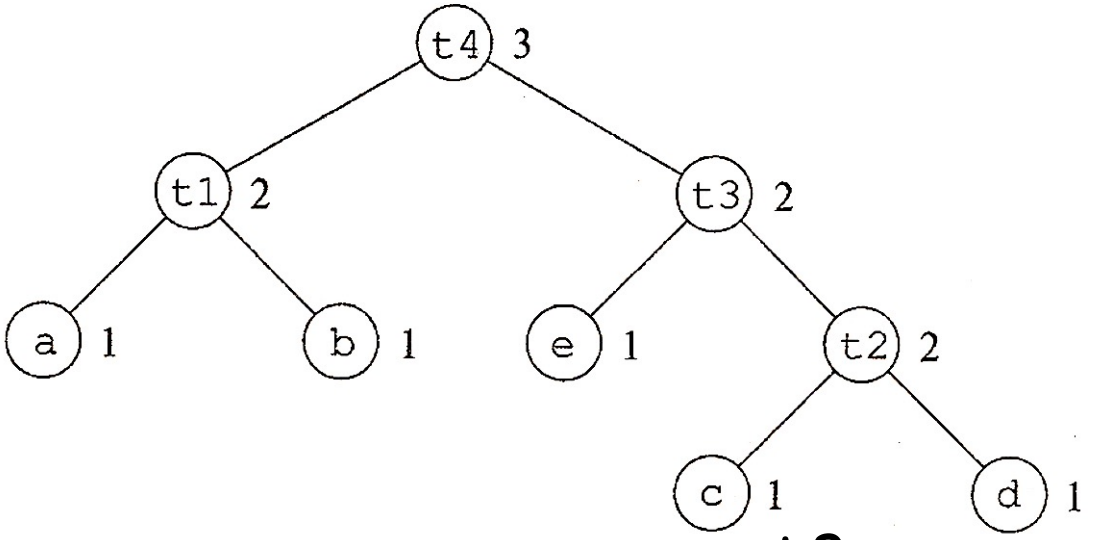
◊ Применим алгоритм 1.6.4 к дереву на рисунке, в предположении, что количество доступных регистров равно 2 ($r$ = 2), т.е. для хранения временных значений при вычислении выражения можно использовать только два регистра – **R1** и **R2**.




◊ Корень дерева имеет метку 3, большую, чем $r$ = 2.
Согласно п. (1) алгоритма 1.6.4 выбирается большой узел.
Так как у обоих дочерних узлов метки одинаковы, выбирается правый узел.

# *1.6 Генерация оптимального кода для выражений*

## 1.6.5 Пример

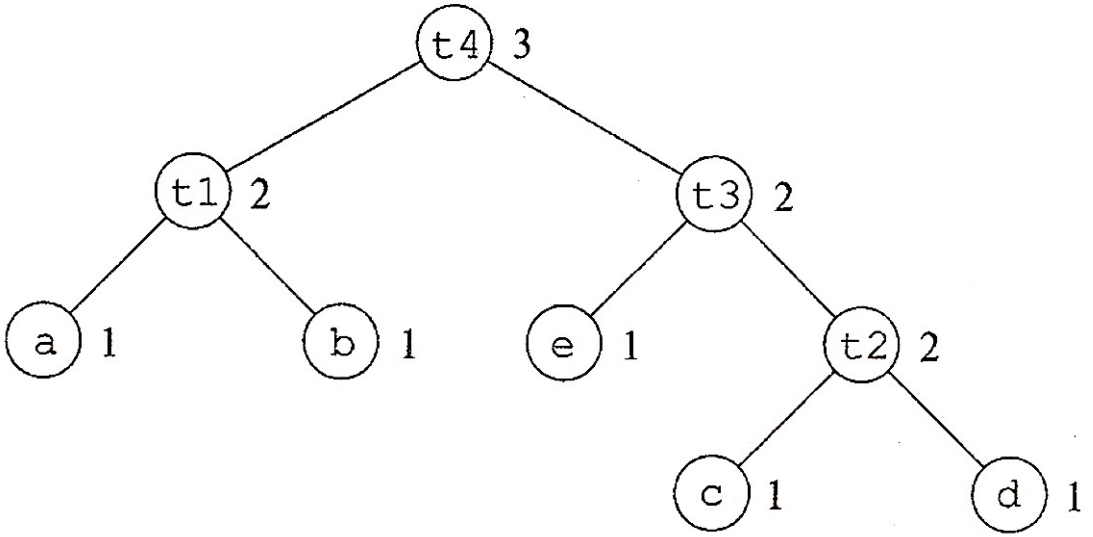



◊ Рекурсивная генерация кода для поддерева с корнем **t3** выполняется алгоритмом 1.6.2, так как метка **t3** равна 2, т.е. регистров для его вычисления достаточно.
Результат похож на результат примера 1.6.3, но вместо регистров **R2** и **R3** используются регистры **R1** и **R2** :

```
LD   R2 d
LD   R1 c
ADD  R2 R1 R2
LD   R1 e
MUL  R2 R1 R2
```

# *1.6 Генерация оптимального кода для выражений*

## 1.6.5 Пример




◊ Поскольку для вычисления левого дочернего узла корня нужны оба регистра, генерируется команда

**ST $t_3$ R2**

◊ Для левого дочернего узла регистров хватает для вычислений, так что алгоритмом 1.6.2 генерируется код

```
LD  R2 b
LD  R1 a
SUB R2 R1 R2
```

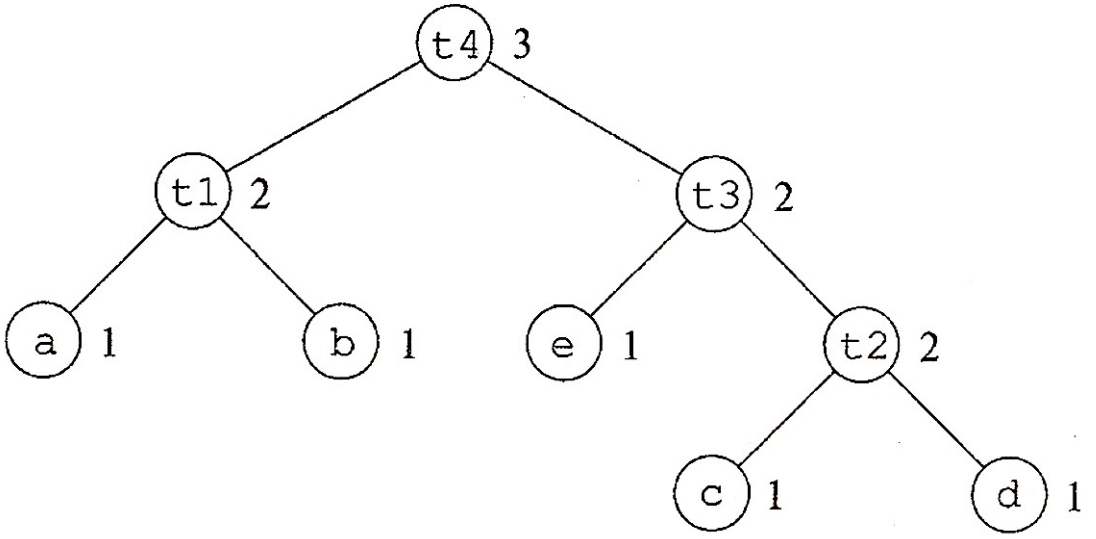
◊ На регистре **R1** восстанавливается значение правого поддерева командой **LD R1 $t_3$**

◊ Выполняется операция в корне дерева командой

```
ADD R2 R2 R1
```

# *1.6 Генерация оптимального кода для выражений*

## 1.6.5 Пример




◊ Поскольку для вычисления левого дочернего узла корня нужны оба регистра, генерируется команда

**ST $t_3$ R2**

◊ Здесь $t_3$ – ячейка памяти, выделенная для хранения промежуточных результатов в узлах с меткой (числом Ершова) =3, т.е. для сохранений и загрузок, возникающих в данном примере при обработке узла `t4` (а не `t3`), имеющего метку 3.
Использование одной и той же буквы t и для узла `t4` в данном примере, и для ячейки $t_3$ в алгоритме – совпадение.

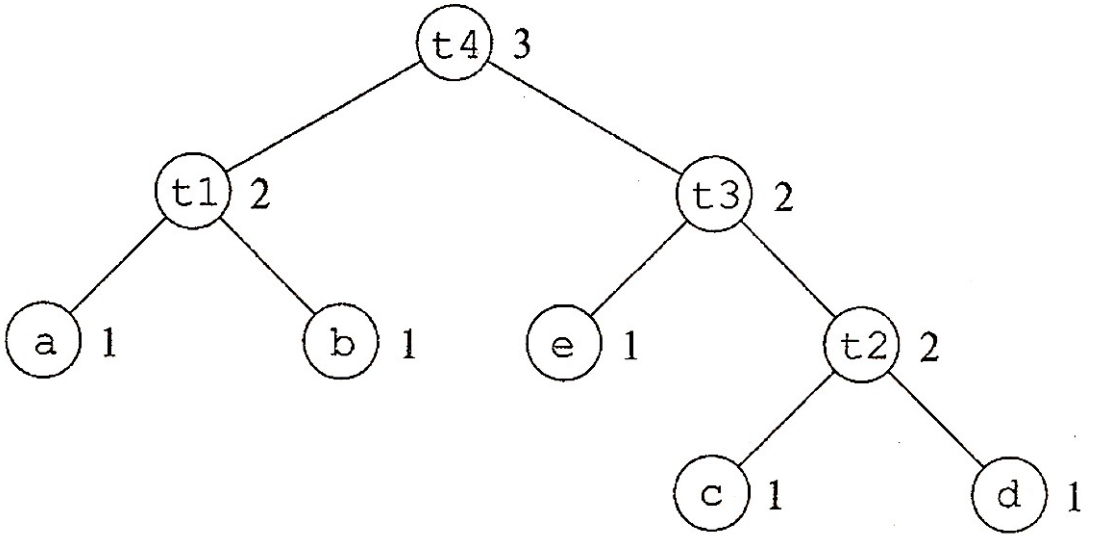
◊ На регистре **R1** восстанавливается значение правого поддерева командой **LD R1 $t_3$**

◊ Выполняется операция в корне дерева командой

**ADD R2 R2 R1**

# 1.6 Генерация оптимального кода для выражений

## 1.6.5 Пример




В результате для дерева на рисунке генерируется код:

```
LD   R2 d
LD   R1 c
ADD  R2 R1 R2
LD   R1 e
MUL  R2 R1 R2
ST   t3 R2
LD   R2 b
LD   R1 a
SUB  R2 R1 R2
LD   R1 t3
ADD  R2 R2 R1
```

Этот код вычисляет дерево, используя только два регистра

# 1.7 Генерация кода с использованием алгоритма динамического программирования

## 1.7.1 Постановка задачи

◊ Алгоритм 1.6.4 генерирует оптимальный код по дереву выражения за время, линейно зависящее от размера дерева. Этот алгоритм работает для машин, у которых все вычисления выполняются в регистрах, а команды состоят из операторов, применяемых к двум регистрам или к регистру и ячейке памяти.

◊ Для расширения класса машин, для которых возможно построение оптимального кода на основе деревьев выражений за линейно зависящее от размера дерева время, можно воспользоваться алгоритмом на основе динамического программирования.

◊ Алгоритм динамического программирования может использоваться для генерации кода для любой машины с $r$ взаимозаменяемыми регистрами $R_0, R_1, \ldots, R_{(r-1)}$ и командами загрузки, сохранения и операций.

Для простоты сделано предположение, что все команды имеют одинаковую стоимость, равную единице, хотя алгоритм динамического программирования можно легко модифицировать для случая, когда стоимости команд различны.

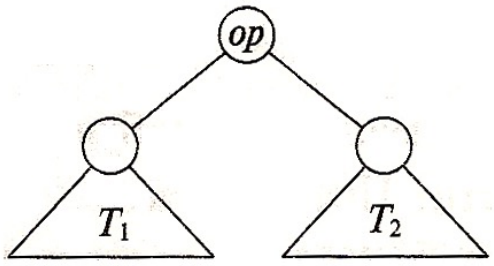
# 1.7 Генерация кода с использованием алгоритма динамического программирования

## 1.7.1 Постановка задачи

◊ Алгоритм динамического программирования – это эвристика, которая состоит в разбиении сложной задачи на подзадачи аналогичной структуры и поиске оптимального решения для каждой подзадачи: предполагается, что каждая подзадача проще основной задачи, и ее оптимальное решение найти проще.

◊ Генерация оптимального кода для выражения сводится к генерации оптимального кода для подвыражений этого выражения и последующей генерации кода для выражения в целом.

◊ Рассмотрим выражение $E$ вида $E_1\ op\ E_2$.




Алгоритм динамического программирования составляет оптимальную программу для $E$, выбирая оптимальный порядок выполнения программ для вычисления $E_1$ и $E_2$, и помещая вслед за ними код для выполнения операции $op$.

Подзадачи генерации оптимального кода для вычисления подвыражений $E_1$ и $E_2$ решаются аналогично.

# Динамическое программирование: задача о рюкзаке

| Номер, $i$ | Масса, $m_i$ | Объем, $v_i$ |
|---|---|---|
| 1 | 6 | 11 |
| 2 | 3 | 5 |
| 3 | 5 | 10 |

Дано: набор из $N$ предметов (каждый имеется в одном экземпляре) с массой $m_i$ и объемом $v_i$ максимальная вместимость рюкзака $V_{max}$

Требуется найти множество предметов с максимально возможной массой, так, чтобы их суммарный объем не превосходил заданного $V_{max}$:

$$\sum_{i=1}^{N} m_i \to max\,, \qquad \sum_{i=1}^{N} v_i \le V_{max}$$

В таблицу последовательно записывается максимальная масса упакованного рюкзака, Используя объем V и только K первых предметов:

**M[K, V] = max( M[K – 1, V], $v_i$ + M[K – 1, V – $v_i$ ] )**

Используя объем V →

Используя K первых предметов →

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | … |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **6** | 6 | 6 | 6 | 6 | |
| 2 | 0 | 0 | 0 | 0 | 0 | **3** | 3 | 3 | 3 | 3 | 3 | 6 | 6 | 6 | 6 | 6 | |
| 3 | 0 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | **5** | 6 | 6 | 6 | 6 | **8** | |
| 4 | … | → | | | | | | | | | | | | | | | |
| | | ↓ | | | | | | | | | | | | | | | |

Максимальный вес **8** для $V=10$ получается при выборе предмета №3 ($m_3=5$, $v_3=10$) и оптимальной упаковке оставшегося объема 5 предметом №2 ($m_2=3$, $v_2=5$)

В каждой ячейке M[3, V] строки 3 сравниваем, что лучше:

- положить предмет №3 ($m_3=5$, $v_3=10$) в рюкзак $V$, а оптимальную упаковку для оставшегося объема с использованием только первых 2-х предметов взять в предыдущей строке в колонке $V - v_3$,
- или не брать предмет №3, а оптимальную упаковку на весь объем $V$ скопировать из предыдущей строки.

# 1.7 Генерация кода с использованием алгоритма динамического программирования

## 1.7.2 Последовательные вычисления

◊ По определению программа $P$ вычисляет дерево $T$ *последовательно*, если она вначале вычисляет поддеревья $T_1$ и $T_2$ дерева $T$ ($T$ либо в порядке $T_1$, $T_2$ и затем корень, либо в порядке $T_2$, $T_1$ ) и **запоминает их значения в памяти**. Затем $P$ вычисляет корень.
При необходимости **используются предварительно вычисленные значения, хранящиеся в памяти.**

◊ **Утверждение**. В случае регистровой машины для любой программы $P$, вычисляющей дерево выражения $T$, можно найти эквивалентную программу $P'$, такую, что

- ◈ стоимость $P'$ не больше стоимости $P$;
- ◈ $P'$ использует не больше регистров, чем $P$;
- ◈ $P'$ вычисляет дерево последовательно.

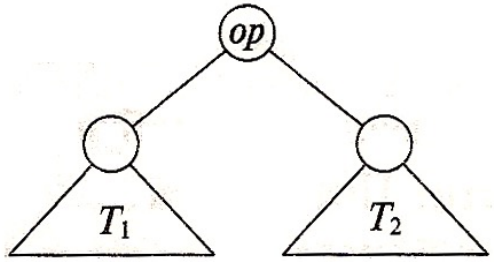
◊ **Следствие**. Каждое дерево выражения можно вычислить оптимальным образом с помощью последовательной программы.

# 1.7 Генерация кода с использованием алгоритма динамического программирования

## 1.7.2 Последовательные вычисления

◊ Оптимальная программа, порожденная алгоритмом динамического программирования, имеет важное свойство, заключающееся в том, что она вычисляет выражение $E = E_1\ op\ E_2$ «последовательно».

Синтаксическое дерево $T$ для выражения $E$.




$T_1$ и $T_2$ – синтаксические деревья для $E_1$ и $E_2$ соответственно.

# 1.7 Генерация кода с использованием алгоритма динамического программирования

## 1.7.3 Алгоритм динамического программирования

◊ Алгоритм динамического программирования состоит из трех фаз.

(1) В восходящем порядке для каждого узла $n$ дерева выражения $T$ вычисляется массив стоимости $C$, $i$-й элемент которого $C[i]$ представляет собой оптимальную стоимость вычисления поддерева $S$ с корнем $n$ с при условии, что для вычисления имеется $i$ ($1 \leq i \leq r$) доступных регистров
($r$ – число регистров целевой машины)

(2) Обход дерева выражения $T$ с использованием векторов стоимости для определения, какие из поддеревьев $T$ должны быть вычислены в память.

(3) Обход каждого дерева с использованием векторов стоимости и связанных с ними команд для генерации конечного целевого кода. Первым генерируется код для поддеревьев, вычисляемых в память.

◊ Каждая из фаз может быть выполнена за время, линейно пропорциональное размеру дерева выражения.

# 1.7 Генерация кода с использованием алгоритма динамического программирования

## 1.7.3 Алгоритм динамического программирования

◊ Стоимость вычисления узла $n$ включает загрузки и сохранения, необходимые для вычисления $S$ с данным количеством регистров и стоимость выполнения операции в корне $S$. Нулевой компонент вектора стоимости – оптимальная стоимость вычисления поддерева $S$ с сохранением результата в памяти.

Свойство последовательного вычисления гарантирует, что оптимальная программа для вычисления поддерева $S$ может быть сгенерирована путем рассмотрения комбинаций только оптимальных программ для поддеревьев дерева с корнем $S$. Сформулированное ограничение сокращает количество случаев, которые необходимо рассмотреть.

# 1.7 Генерация кода с использованием алгоритма динамического программирования

## 1.7.3 Алгоритм динамического программирования

◊ Для вычисления стоимости $C[i]$ в узле $n$ будем рассматривать команды как правила преобразования дерева.
Для каждого шаблона $E$, соответствующего входному дереву в узле $n$:

- ⧫ Изучая векторы стоимости соответствующих наследников $n$, определить стоимости вычисления операндов в листьях $E$.
- ⧫ Для операндов $E$, являющихся регистрами, рассмотреть все возможные порядки вычисления поддеревьев $T$ в регистры.
  Для каждого из рассматриваемых порядков вычисления первое поддерево, соответствующее регистровому операнду, можно вычислить с использованием $i$ доступных регистров, второе – с помощью $i - 1$ доступных регистров и т.д.
- ⧫ Добавить стоимость команды, связанной с корнем $E$.
  Значение $C[i]$ представляет собой минимальную стоимость среди всех возможных порядков вычисления.

# *1.7 Генерация кода с использованием алгоритма динамического программирования*

## 1.7.3 Алгоритм динамического программирования

◊ Векторы стоимости для всего дерева $T$ могут быть вычислены в восходящем порядке (снизу вверх) за линейное время. Команды, соответствующие наилучшей стоимости $C[i]$ для каждого значения $i$, удобно хранить в узлах дерева; при этом наименьшая стоимость корневого узла $T$ соответствует минимальной стоимости вычисления дерева $T$.

# 1.7 Генерация кода с использованием алгоритма динамического программирования

## 1.7.4 Пример

◊ Пусть машина имеет **два** регистра **R0** и **R1** и следующие команды, стоимость каждой из которых равна единице:

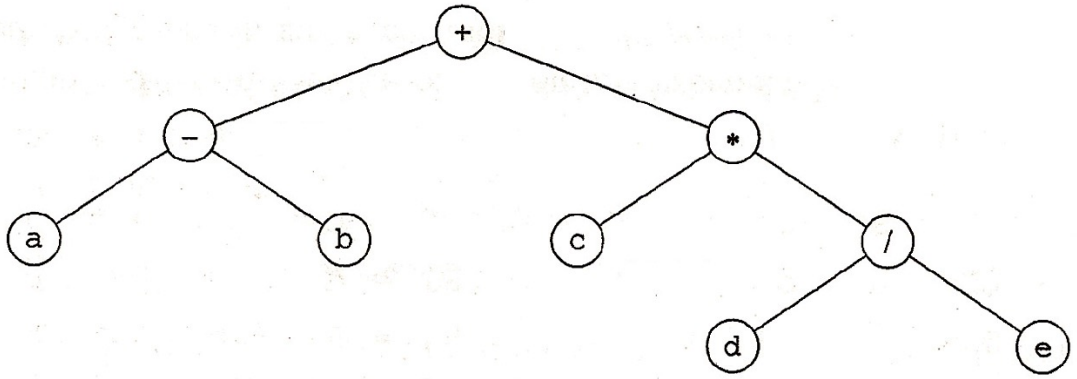
| *Команда* | *Семантика команды* |
|---|---|
| **LD Ri, Mj** | $R_i = M_j$ |
| **op Ri, Ri, Rj** | $R_i = R_i \; op \; R_j$ |
| **op Ri, Ri, Mj** | $R_i = R_i \; op \; M_j$ |
| **LD Ri, Rj** | $R_i = R_j$ |
| **ST Mi, Rj** | $M_i = R_j$ |

◊ $R_i$ – либо $R_0$, либо $R_1$, так как регистров всего **два**
$M_j$ – адрес в памяти.
$op$ – знак арифметической операции.

# 1.7 Генерация кода с использованием алгоритма динамического программирования

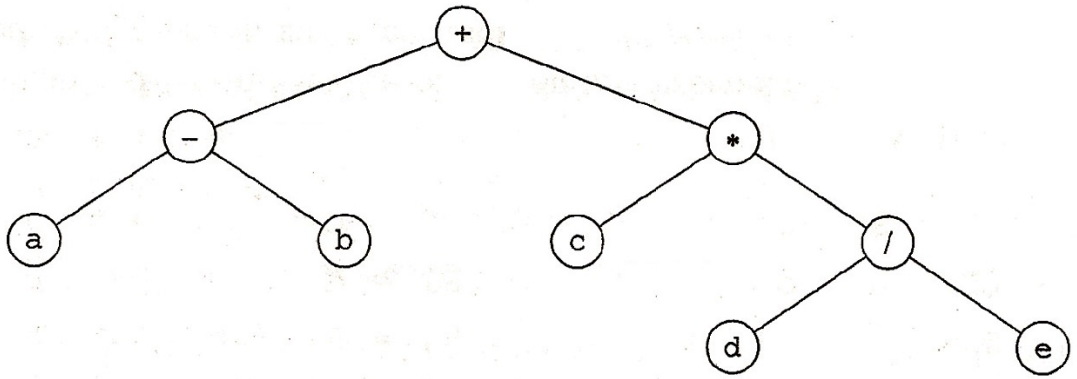
## 1.7.4 Пример (продолжение)

◊ Применим алгоритм динамического программирования для генерации оптимального кода вычисления выражения, представленного синтаксическими деревом (см. рисунок).



# 1.7 Генерация кода с использованием алгоритма динамического программирования

## 1.7.4 Пример (продолжение)

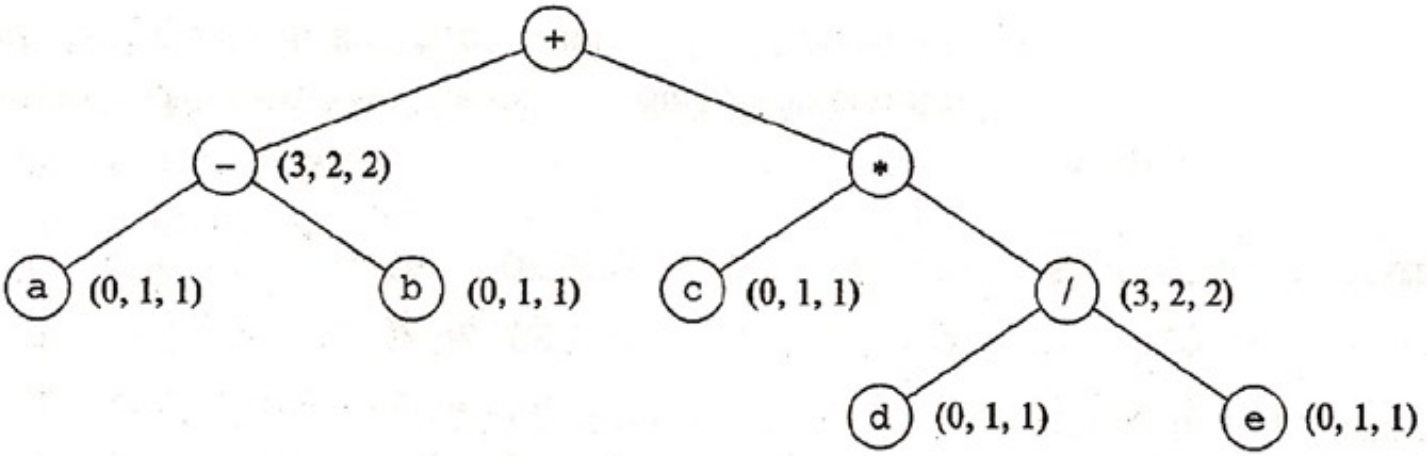



(1) Вычисление векторов стоимости.

- Вычисление векторов стоимости для листьев **a**, **b**, **c**, **d** и **e**:
  - Пусть каждый лист (переменная) хранится в памяти, т.е. стоимость листа вычисления связана с необходимостью его загрузки на регистр; пусть $C[i]$ – стоимость загрузки листа в случае, когда доступно $i$ регистров; в рассматриваемом случае (у компьютера всего два регистра) $i$ может принимать значения 0, 1 и 2.
  - $C[0] = 0$, так если нет доступных регистров, загрузки не происходит;
  - $C[1] = C[2] = 1$, так как в обоих случаях значение листа можно загрузить на регистр одной командой: **`LD R0, a`**.
  - Следовательно, вектор стоимости каждого листа равен $(0, 1, 1)$.

# 1.7 Генерация кода с использованием алгоритма динамического программирования

## 1.7.4 Пример

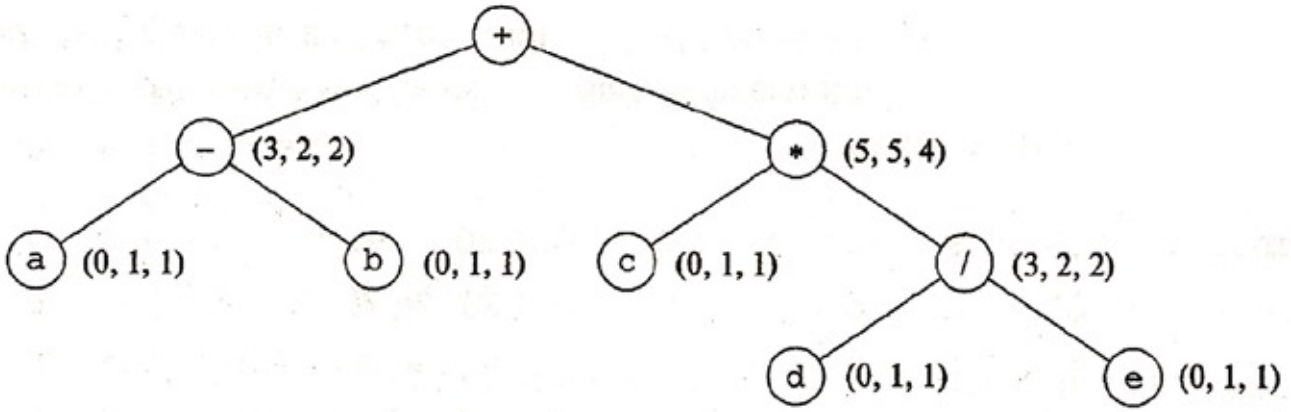



(1) Вычисление векторов стоимости.

- Вычисление векторов стоимости для узлов «-» и «/»:
  - $C[0] = 3$, так как оба операнда нужно загрузить на регистры (1+1=2) и выполнить соответствующую арифметическую операцию (1)
  - $C[1] = C[2] = 2$, так как на регистр загружается только один операнд: нужно загрузить этот операнд (1) и выполнить операцию (1)

# 1.7 Генерация кода с использованием алгоритма динамического программирования

### 1.7.4 Пример

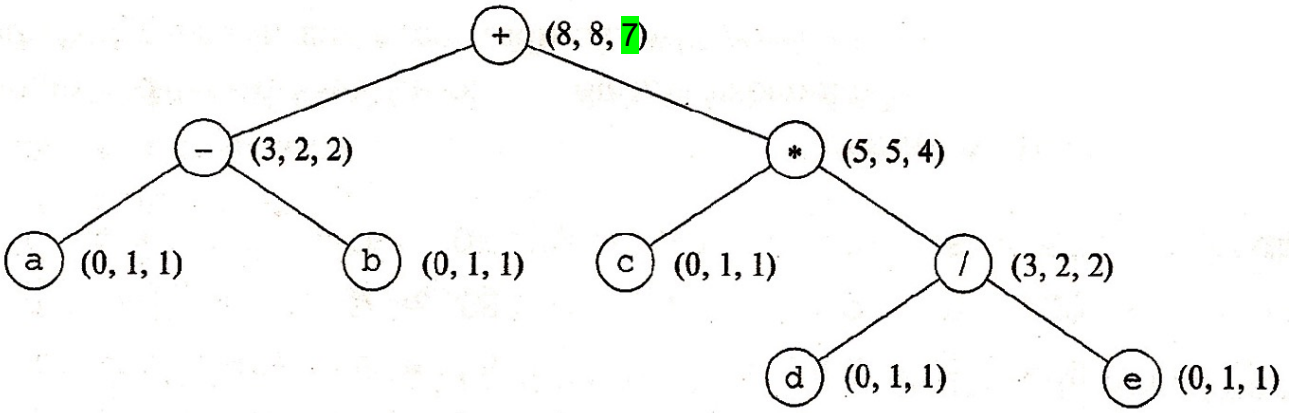



(1) Вычисление векторов стоимости.

- Вычисление вектора стоимости для узла «*»: нужно учесть стоимость вычисления правого поддерева (узла «/»)
  - $C[0] = 5$, так как мы знаем, что 0 на самом деле 1 (загрузка регистра), вычисление второго операнда стоит 3, операция «*****» стоит еще 1; итого: **`1+3+1=5`**
  - $C[1] = 5$: мы не можем взять слева 1, так как единственный доступный регистр занят результатом операции «/»; манипуляции с памятью добавляют в стоимость лишнюю 1 и 2 превращается в 3
  - $C[2] = 2$, так как оба операнда на регистрах: **`1+2+1=4`**

# 1.7 Генерация кода с использованием алгоритма динамического программирования

## 1.7.4 Пример

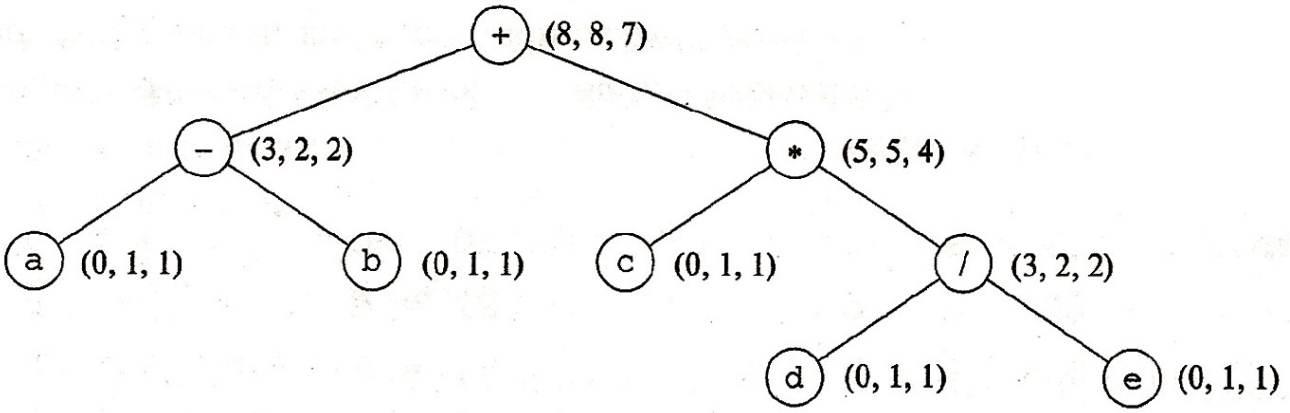



(1) Вычисление векторов стоимости.

- Вычисление вектора стоимости для корня дерева. Различные варианты вычисления корня (для случая 2-х свободных регистров, 3-я позиция вектора)
  - (1) (2 свободных регистра) левое поддерево ⇒ **R0** (стоимость 2)
    (2) (1 свободный регистр остается, т.к. еще в 1 регистре сохранен результат левого поддерева) правое поддерево ⇒ **R1** (стоимость 5),
    (3) команда **ADD R0,R0,R1** (стоимость 1). **Итого 2 + 5 + 1 = 8**
  - (1) правое поддерево ⇒ **M** (стоимость 5)
    (2) (2 свободных регистра) левое поддерево ⇒ **R0** (стоимость 2),
    (3) команда **ADD R0,R0,M** (стоимость 1). **Итого 5 + 2 + 1 = 8**
  - (1) (2 свободных регистра) правое поддерево ⇒ **R1** (стоимость 4)
    (2) (1 свободный регистр) левое поддерево ⇒ **R0** (стоимость 2),
    (3) команда **ADD R0,R0,R1** (стоимость 1). **Итого 4 + 2 + 1 = 7**

# 1.7 Генерация кода с использованием алгоритма динамического программирования

## 1.7.4 Пример




(2) Обход дерева и генерация кода

Имея векторы стоимости можно построить код путем обхода дерева. Для рассматриваемого дерева в предположении доступности двух регистров оптимальный код имеет следующий вид:

```
LD      R0,     c               //R0 = c
LD      R1,     d               //R1 = d
DIV     R1,     R1,     e       //R1 = R1 / e
MUL     R0,     R0,     R1      //R0 = R0 * R1
LD      R1,     a               //R1 = a
SUB     R1,     R1,     b       //R1 = R1 - b
ADD     R1,     R1,     R0      //R1 = R1 + R0
```