

Автоматизация программирования в СССР: **Трансляторы (60-70-е годы)**

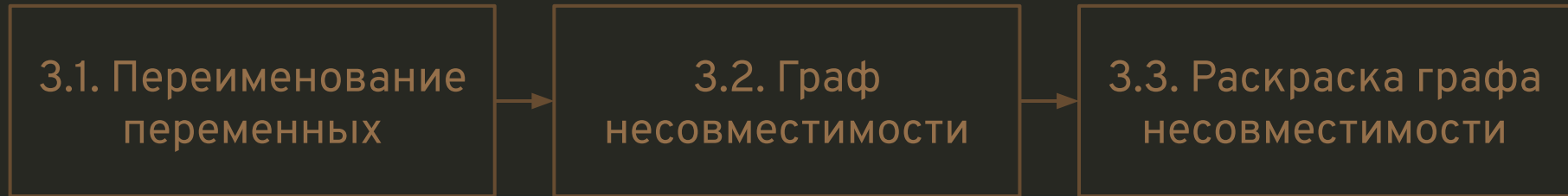
Петр Советов, РТУ МИРЭА

24 Мая 2023 г.

1. Введение

2. Пример экономии памяти

3. Фазы экономии памяти



4. Экономия памяти для массивов

5. Заключение

Транслятор	Дата завершения разработки	Срок разработки	Особенности
ТА-1	1962 г.	2 года	Быстрая трансляция
ТА-2	1963 г.	2 года	Полная поддержка языка
Альфа	1964 г.	4 года	Генерация качественного кода

Backus J. W. et al. *Revised report on the algorithm language ALGOL 60*
//Communications of the ACM. – **1963**. – Т. 6. – №. 1. – С. 1-17.

Альфа-транслятор

Размер кода транслятора: **251** Кбайт.

Язык реализации: машинный код ЭВМ **М-20**.

Количество проходов: **24**.

Руководитель: А.П. Ершов.

Главные разработчики: Г.И. Кожухин (frontend) и И.В. Поттосин (backend).

Сотрудник Вычислительного центра В. А. Катков с помощью транслятора решил в течение года около 30 задач — трехгодовую норму при ручном программировании.

“Одиннадцать авторов транслятора – все с университетским образованием – к концу 1962 г. имели следующий стаж работы в программировании: один – 8 лет, один – 6 лет, один – 5, пятеро – 3, двое – 2, один – меньше года”.

Г. И. Бабецкий, М. М. Бежанова, Ю. М. Волошин, А. П. Ершов, Б. А. Загацкий, Л. Л. Змиевская, Г. И. Кожухин, С. К. Кожухина, Р. Д. Мишкович, Ю. И. Михалевич, И. В. Поттосин, Л. К. Трохан, “Система автоматизации программирования АЛЬФА”, Ж. вычисл. матем. и матем. физ., 5:2 (1965), 317–325

- Трехадресная система команд память-память.
- Один индексный регистр.
- 4096 слов (23 Кбайт) ОЗУ.
- 12288 слов (69 Кбайт) магнитных барабанах.
- Более медленная внешняя память: магнитная лента и перфокарты.

Адрес	Команды и числа				Пояснения	№
1100	70				в МЗУ	49
1	15	0377	0041		$i = 0001$? \rightarrow на W	2
2	36		1122	7760	\downarrow ит. 0 \Rightarrow Фикс.	3
3	33	0377	0041	0005	эл. ОС \rightarrow τ_1 в A_2 и τ_2 в A_3	4
4	33	0005	0041	0004	эл. ОС \rightarrow τ_1 в A_2 и τ_2 в A_3	5
5	54	0064	0005	0005	эл. ОС в A_3 и τ_2	6
6	33	0006	0025	0006	ОС кон в A_1 и τ_2 и τ_1 в A_2	7
7	75	0005	0006	7760	Ком. 1. 00000000	8
1110	54	0064	0006	0006	ОС кон в A_2 и τ_2 и τ_1 в A_3	9
1	33	0006	0004	0006	ОС кон в A_2 и τ_2	10
2	13	7760	0006	7760	Гор. фиксатор / эл. ОС	11
3	54	0050	7760	0006	ОС кон в A_3 и τ_2	12
4	13	1117	0006	1117	Формир. ОС кон	50
5	00				Формир. ОС кон	2
6	72				Осн. РА	3
7	500	4327			Передача из НОС в ОС	4
1120	33	0004	0041	0004	$\tau_1 = 0$?	5
1	171		1117	7777	$\tau_2 < 0$? \rightarrow на МЗУ	6
2	00			7766	0 \Rightarrow Фикс. ТМ МЗУ	7
3	00			7765	0 \Rightarrow Фикс. ТМ МЗУ	8
4	00			7762	0 \Rightarrow Фикс. ТМ МЗУ	9
5	55	7747	0060	0012	$\tau_1 W = 0$? \rightarrow 31 on. $\tau_2 W$ в τ_2	10
6	36		1172	7761	\downarrow ит. 0 \Rightarrow Фикс. ТМ МЗУ	11
7	55	7755	0045	0006	ТМ МЗУ кон. \rightarrow τ_2	12

```
начало вещественный В, Х, Y – вектор n;  
вещественный А – матрица n * n;  
    X := Y + A ^ (-1) * B  
конец.
```

Главная особенность – работа на уровне **многомерных массивов**:

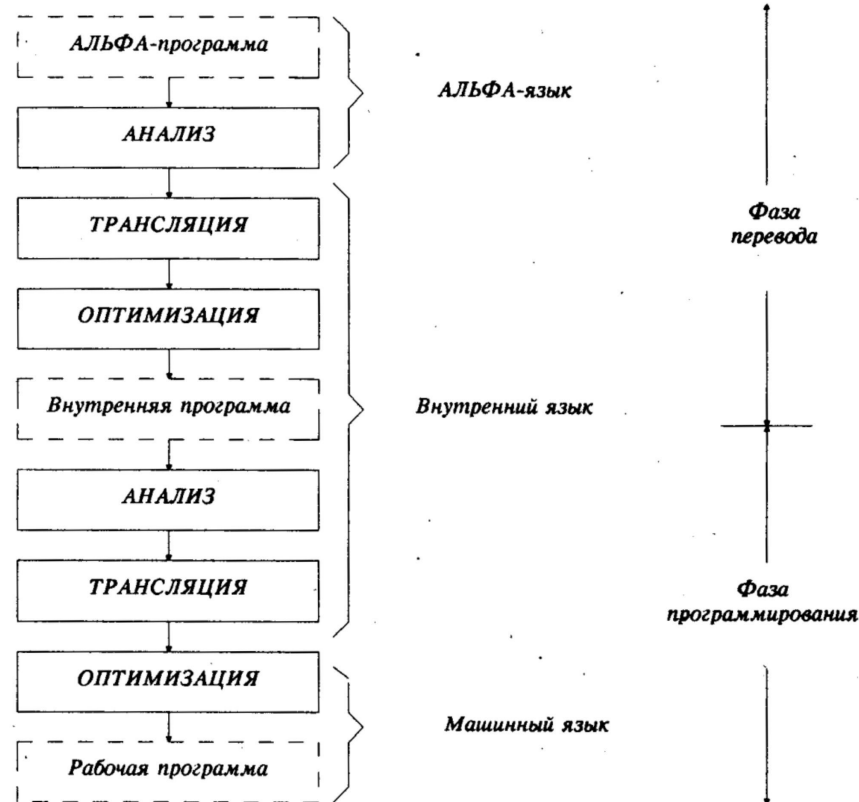
- создание массивов из переменных,
- компоновка массивов из других массивов,
- срезы,
- покомпонентные операции,
- ...

Структура Альфа-транслятора

Основные оптимизации:

- перевод операций над многомерными массивами в циклы,
- объединение циклов (loop fusion),
- чистка циклов (LICM),
- локальная экономия совпадающих выражений (local CSE),
- **межпроцедурная экономия памяти для скаляров и массивов.**


Рекурсия в программах не поддерживается.



`x := 10 * (a - b + c - d) / e`



`+ a -b c -d t1`

$x := 10 * (a - b + c - d) / e$


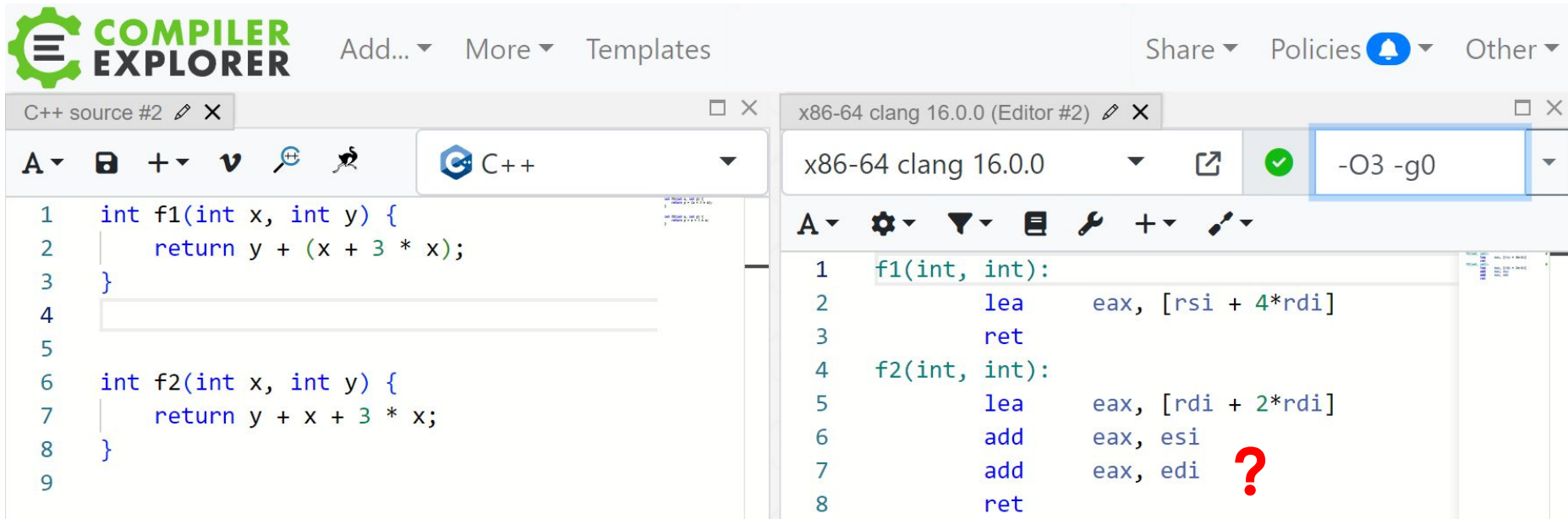
+	a	-b	c	-d	t1
*	10	t1	1/e		x

- SLP-векторизация [1].
- Алгебраические упрощения [2].

[1] Porpodas V. et al. Super-Node SLP: *Optimized vectorization for code sequences containing operators and their inverse elements* // **2019** IEEE/ACM International Symposium on Code Generation and Optimization (CGO). – IEEE, 2019. – С. 206-216.

[2] Norvig P. *Paradigms of artificial intelligence programming: case studies in Common LISP*. – Morgan Kaufmann, **1992**.

Алгебраические упрощения – разве не решенная задача? 11 из 82



The image shows the Compiler Explorer interface. On the left, the C++ source code is displayed in a file named 'C++ source #2'. It contains two functions: `f1` and `f2`. `f1` takes two integers `x` and `y` and returns `y + (x + 3 * x)`. `f2` takes two integers `x` and `y` and returns `y + x + 3 * x`. On the right, the assembly output for `x86-64 clang 16.0.0` is shown. The assembly for `f1` consists of a `leaq` instruction to load `[rsi + 4*rdi]` into `eax`, followed by a `ret` instruction. The assembly for `f2` consists of a `leaq` instruction to load `[rdi + 2*rdi]` into `eax`, followed by two `add` instructions: `add eax, esi` and `add eax, edi`, and finally a `ret` instruction. A red question mark is placed next to the second `add` instruction in the `f2` block, highlighting the redundancy of adding `esi` and `edi` separately when the result is already in `eax`.

Compiler Explorer Interface:

- Language:** C++
- Compiler:** x86-64 clang 16.0.0
- Optimization Level:** -O3 -g0

C++ Source Code (C++ source #2):

```
1 int f1(int x, int y) {  
2     return y + (x + 3 * x);  
3 }  
4  
5  
6 int f2(int x, int y) {  
7     return y + x + 3 * x;  
8 }  
9
```

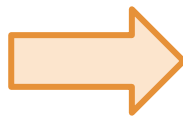
Assembly Output (x86-64 clang 16.0.0):

```
1 f1(int, int):  
2     leaq    eax, [rsi + 4*rdi]  
3     ret  
4 f2(int, int):  
5     leaq    eax, [rdi + 2*rdi]  
6     add     eax, esi  
7     add     eax, edi  
8     ret
```

Внутренний язык Альфы.

Сохранение высокоуровневой информации

```
массив a[1:10,1:10];
...
для i := b, ...
  для j := 0, ...
    для k := 1, ...
      a[k+i*(i+j-1), j+k+5]
      ...
      a[k+5, sin(j)] ...
```



```
...
+ i -1 t3
* t3 i t4
* t4 10 t5
  для j := 0, ... {
    sin j t6
    + t6 -1 t7
    для k := 1, ... { ...
      a[11 k t2 j t5 5]
      ...
      a[10 k t7 50] ...
```

LLVM

```
define i32 @main() {
    br label %1

1:
    %2 = phi i64 [ %8, %5 ], [ 0, %0 ]
    %3 = phi i32 [ %7, %5 ], [ 0, %0 ]
    %4 = icmp slt i64 %2, 11
    br i1 %4, label %5, label %9

5:
    %6 = trunc i64 %2 to i32
    %7 = add i32 %3, %6
    %8 = add i64 %2, 1
    br label %1

9:
    ret i32 %3
}
```

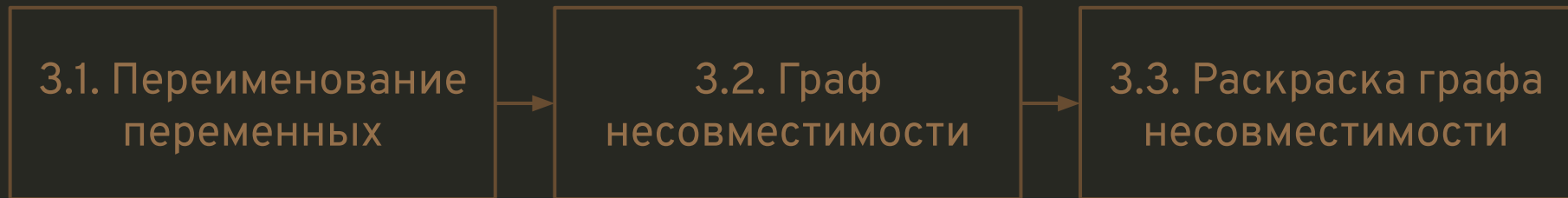
MLIR

```
func.func @main() → i32 {
    %sum_0 = arith.constant 0 : i32
    %sum = affine.for %i = 0 to 11 step 1
    iter_args(%sum_i = %sum_0) → (i32) {
        %t = arith.index_cast %i : index to i32
        %sum_next = arith.addi %sum_i, %t : i32
        affine.yield %sum_next : i32
    }
    return %sum : i32
}
```

1. Введение

2. Пример экономии памяти

3. Фазы экономии памяти



4. Экономия памяти для массивов

5. Заключение

```
x = input()  
y = x + 1  
x = 2  
print(x + y)
```

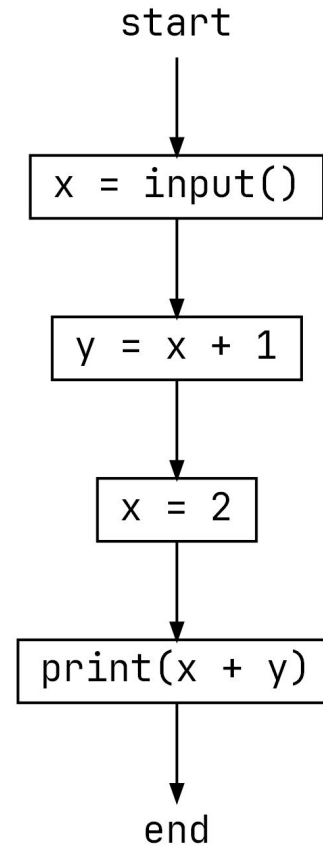
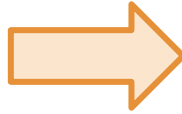


Схема Лаврова

16 из 82

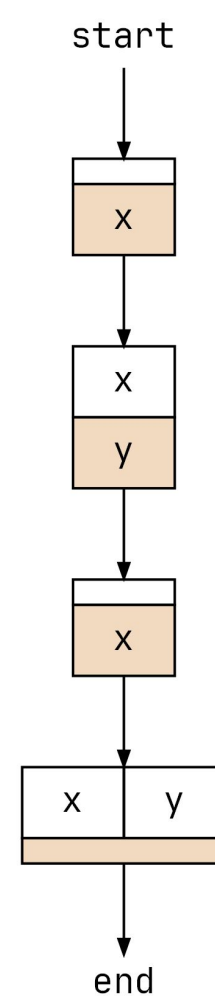
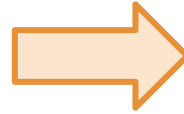
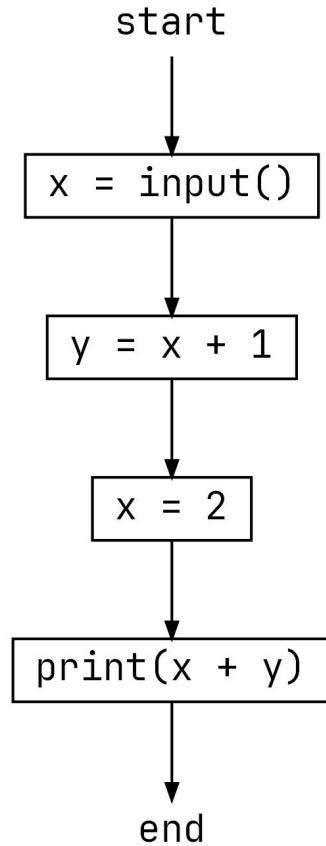
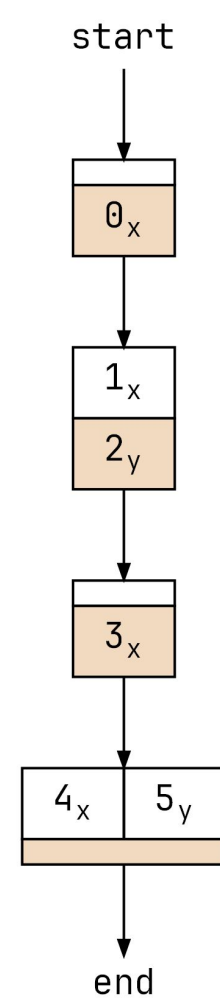
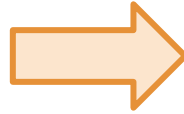
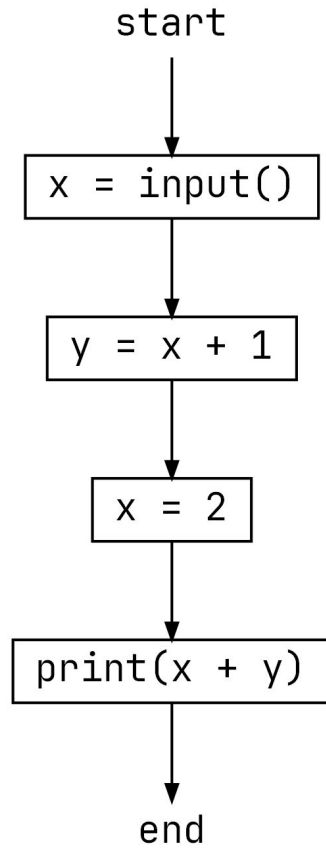


Схема Лаврова с полюсами

17 из 82



Информационный граф и Р-схема

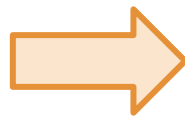
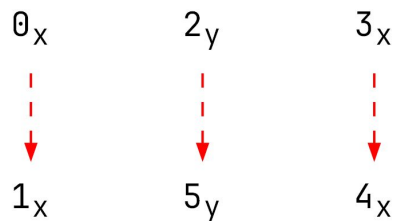
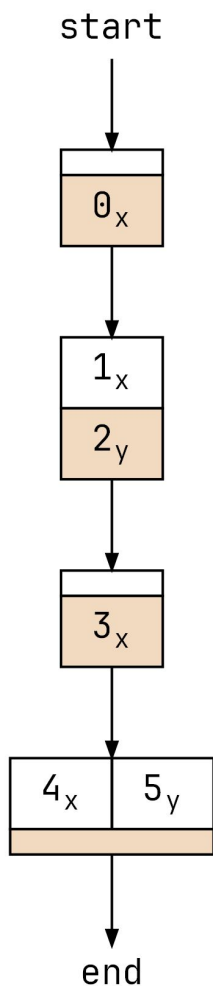
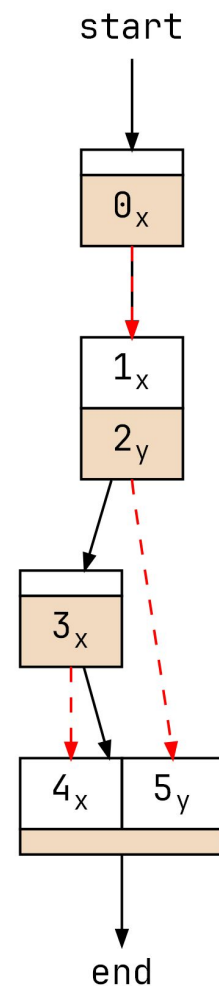
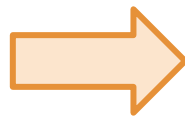
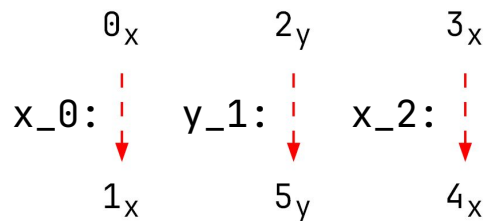
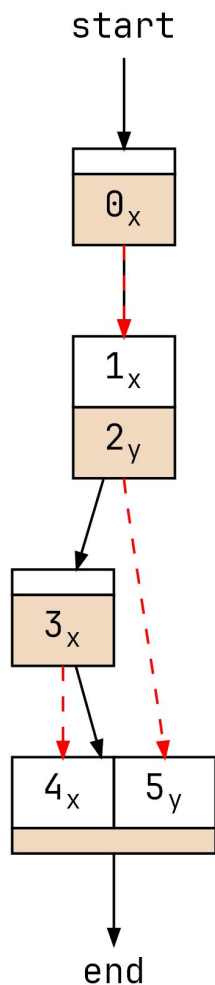


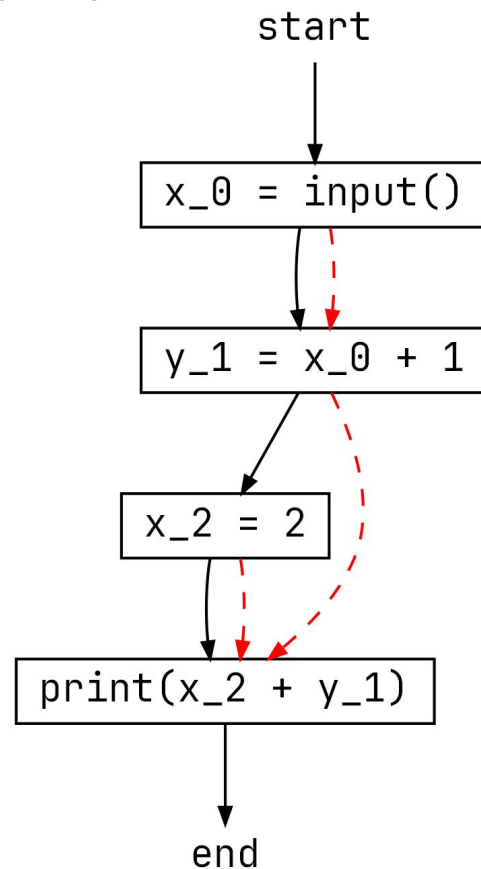
Схема с распределенной памятью (Р-схема) = управляющий граф + информационный граф



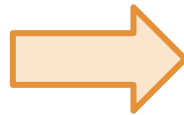
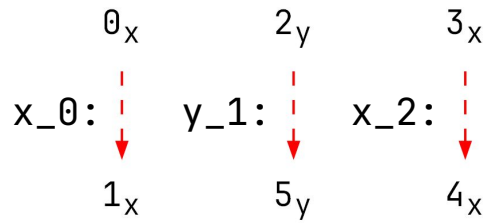
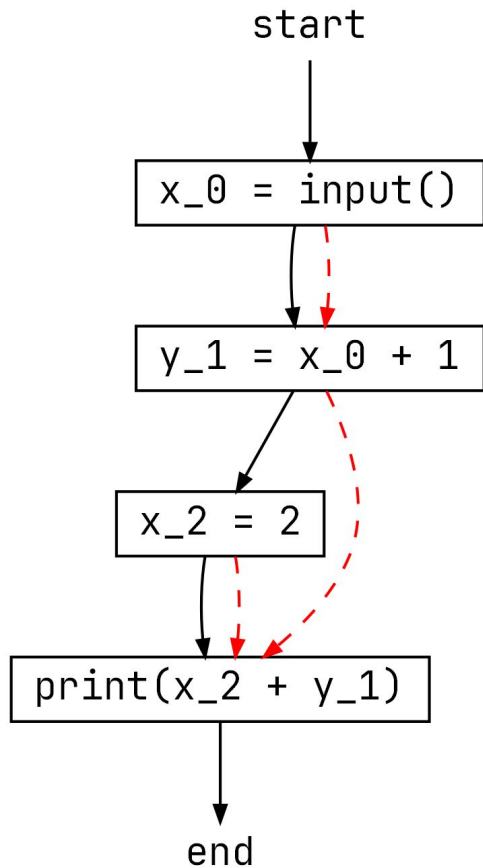
Каноническое распределение памяти



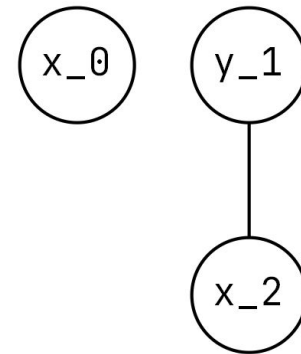
Введем новые имена для областей действия (как в SSA).

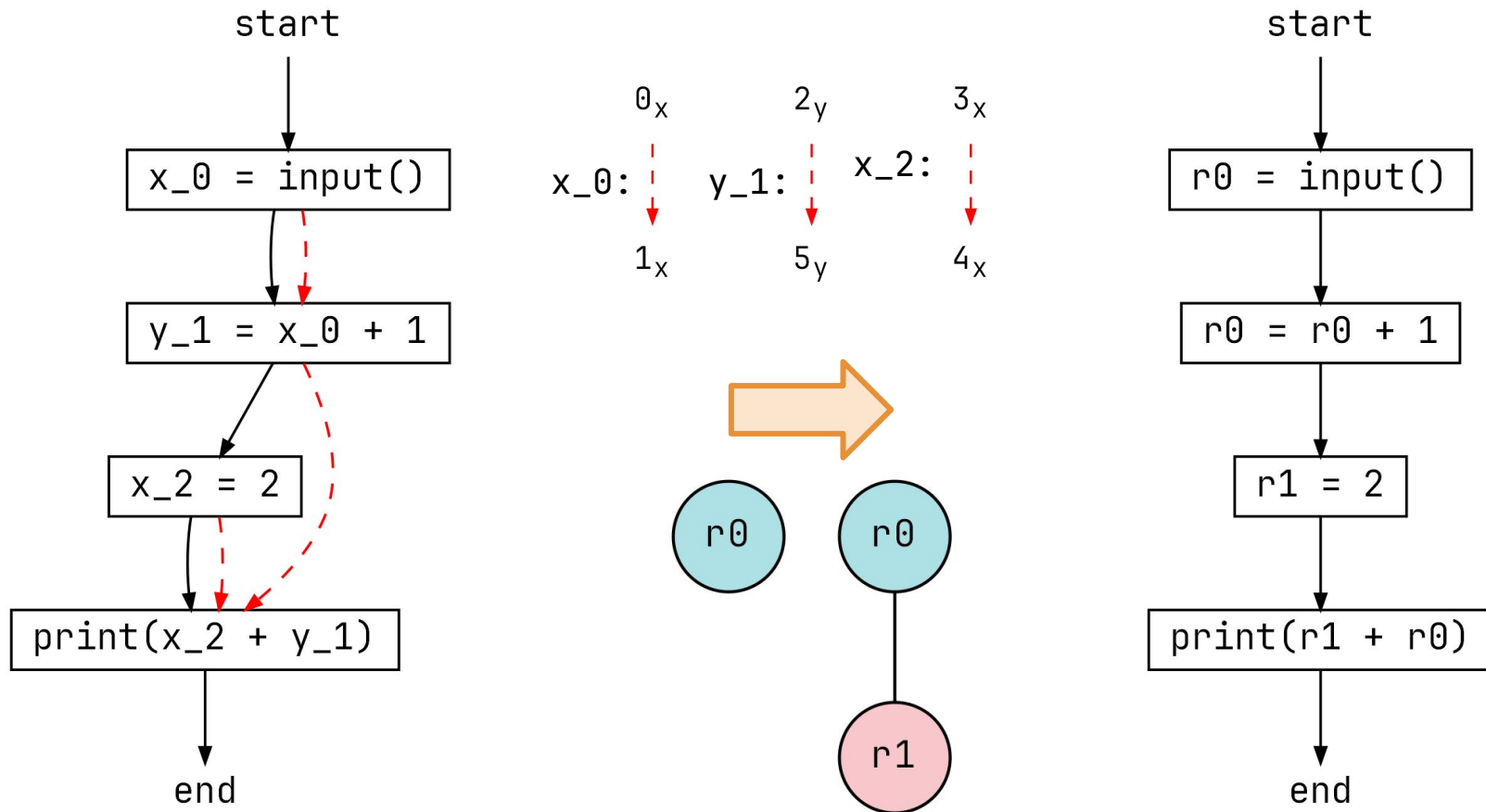


Граф несовместимости



Пересекаются
области действия
 $y_1 \neq x_2$





1. Введение

2. Пример экономии памяти

3. Фазы экономии памяти

**3.1. Переименование
переменных**



3.2. Граф
несовместимости

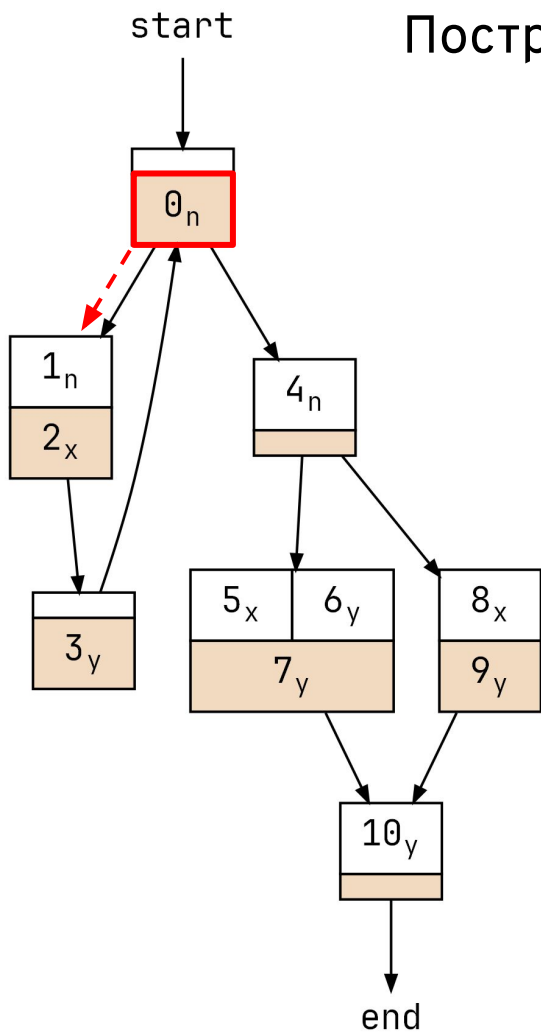


3.3. Раскраска графа
несовместимости

4. Экономия памяти для массивов

5. Заключение

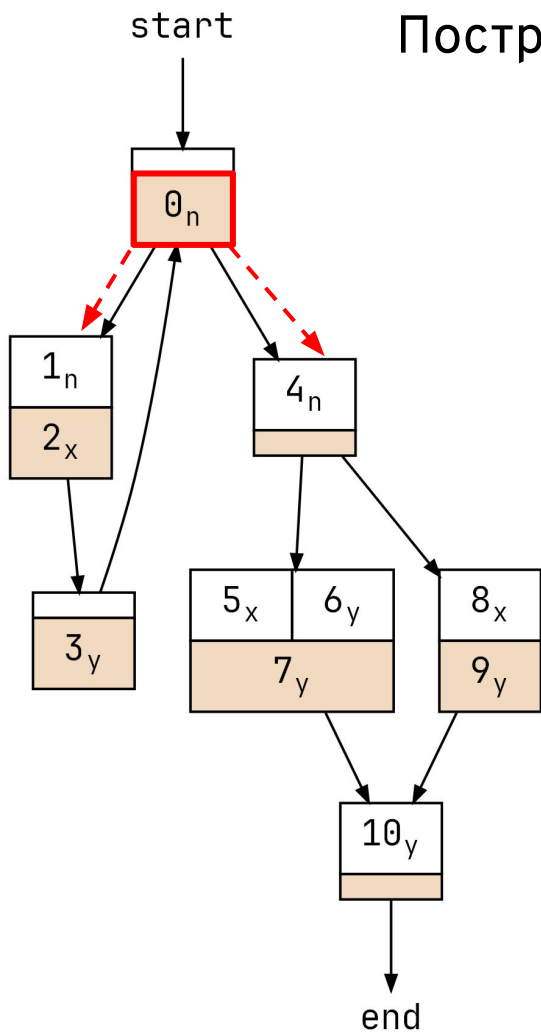
Построение информационного графа (1)



Обход схемы в глубину до операторов, вырабатывающих наш результат.

Рез.	0_n	2_x	3_y	7_y	9_y
Арг.	1_n				

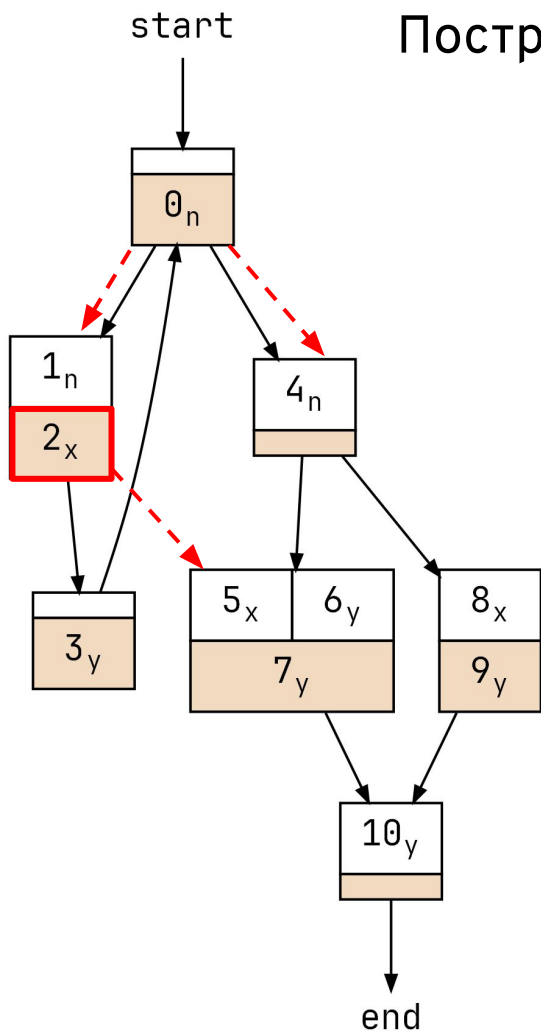
Построение информационного графа (2)



Обход схемы в глубину до операторов, вырабатывающих наш результат.

Рез.	0_n	2_x	3_y	7_y	9_y
Арг.	$1_n, 4_n$				

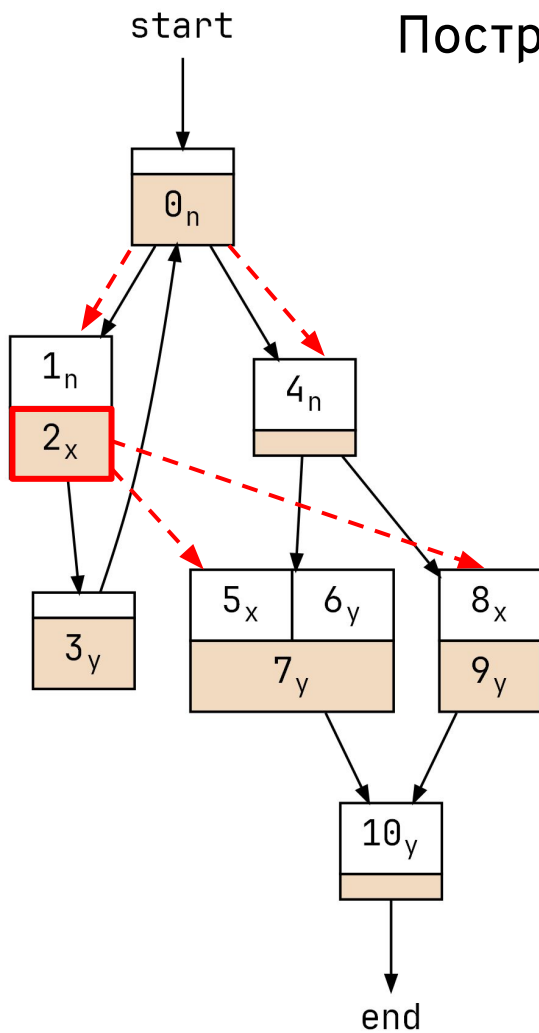
Построение информационного графа (3)



Обход схемы в глубину до операторов, вырабатывающих наш результат.

Рез.	0_n	2_x	3_y	7_y	9_y
Арг.	$1_n, 4_n$	5_x			

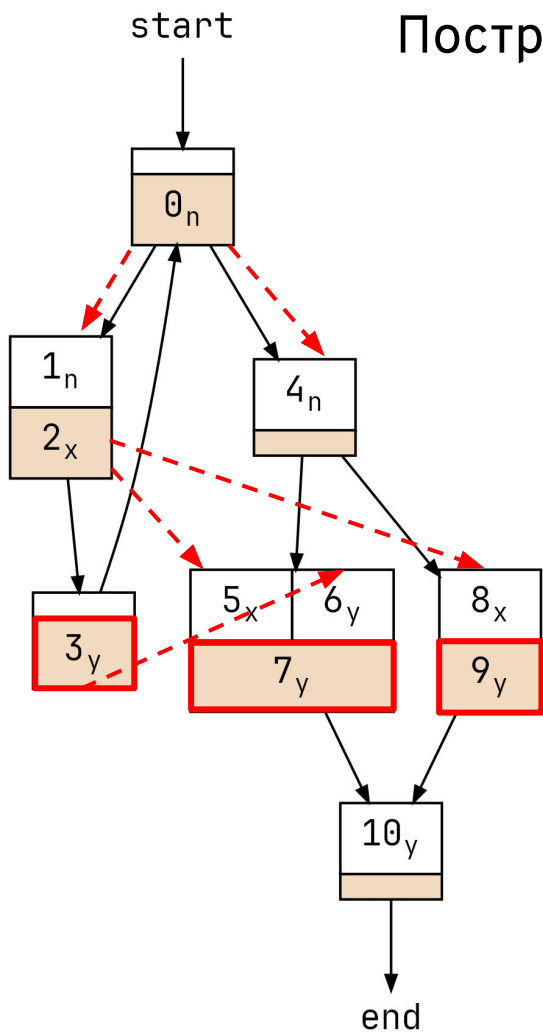
Построение информационного графа (4)



Обход схемы в глубину до операторов, вырабатывающих наш результат.

Рез.	0_n	2_x	3_y	7_y	9_y
Арг.	$1_n, 4_n$	$5_x, 8_x$			

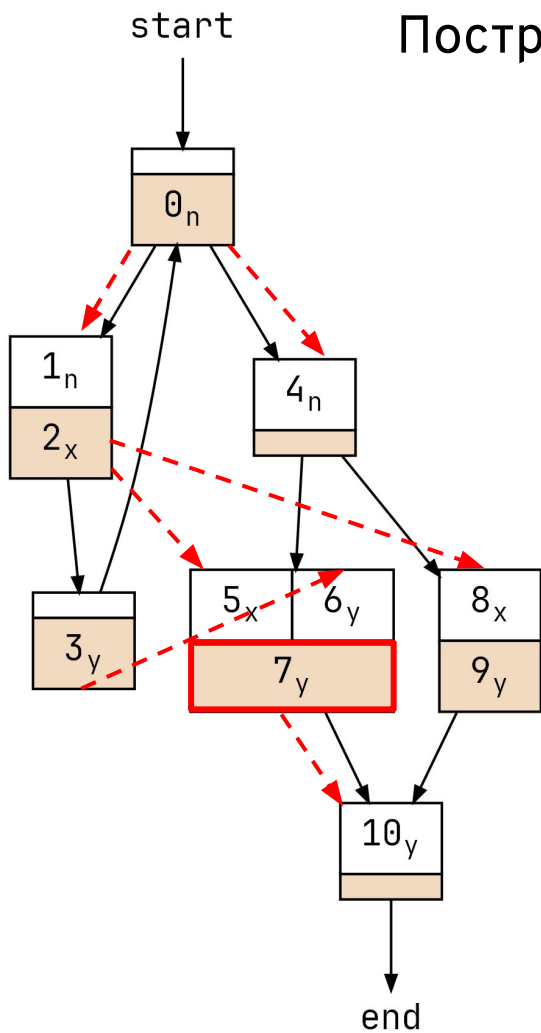
Построение информационного графа (5)



Обход схемы в глубину до операторов, вырабатывающих наш результат.

Рез.	0_n	2_x	3_y	7_y	9_y
Арг.	$1_n, 4_n$	$5_x, 8_x$	6_y		

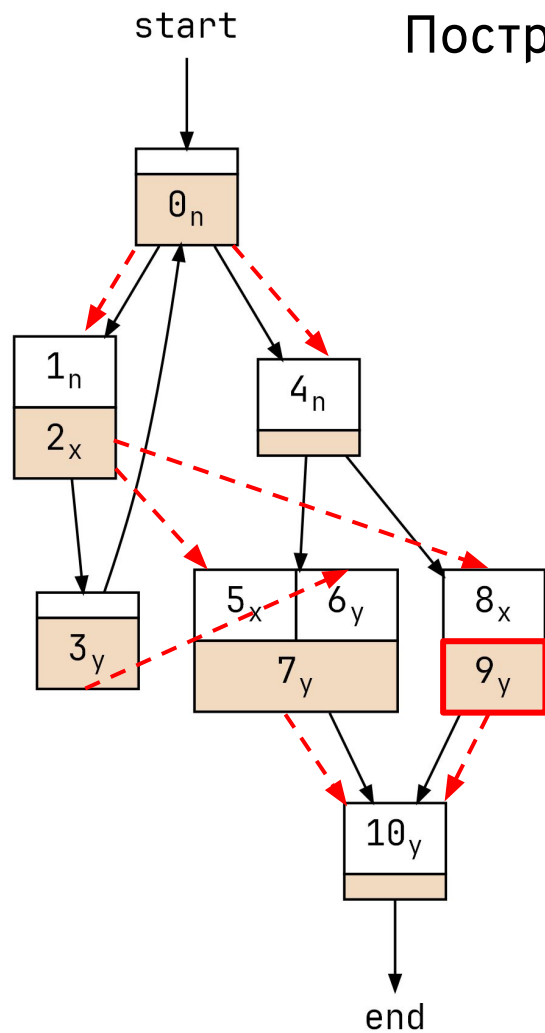
Построение информационного графа (6)



Обход схемы в глубину до операторов, вырабатывающих наш результат.

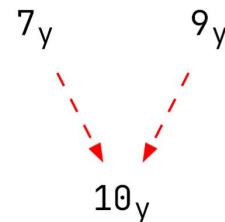
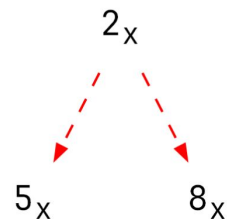
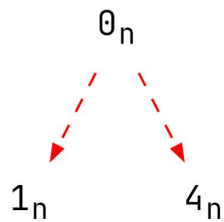
Рез.	0_n	2_x	3_y	7_y	9_y
Арг.	$1_n, 4_n$	$5_x, 8_x$	6_y	10_y	

Построение информационного графа (7)



Области действия – **компоненты связности** информационного графа.

Рез.	0_n	2_x	3_y	7_y	9_y
Арг.	$1_n, 4_n$	$5_x, 8_x$	6_y	10_y	10_y



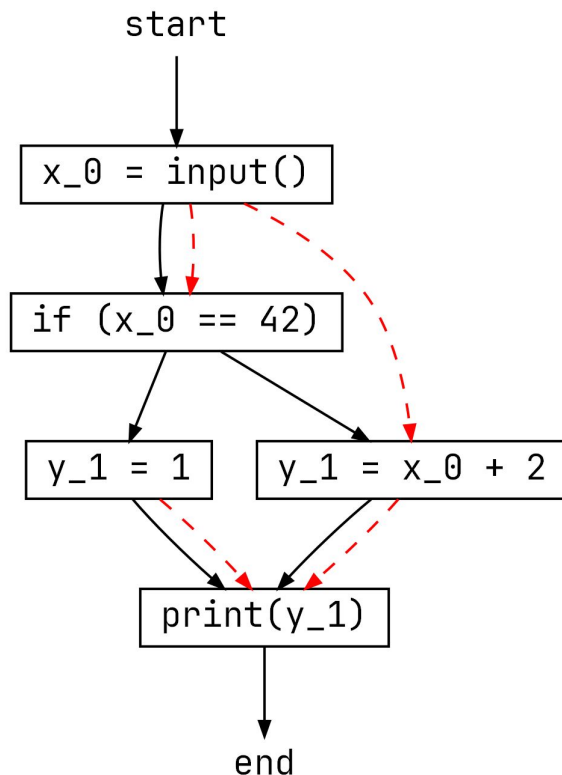
- В [1] информационная связь — **def-use chain**, а область действия — **def-use web**.
- В [2] область действия— **maximal-sized global live range**. Для получения областей действия строим **SSA**, затем выходим из **SSA**, объединяя имена, в том числе в точках ф-функций.
- В LLVM используются **live-intervals** [3] и нумерация значений для создания новых имен после выхода из **SSA**.

[1] Muchnick S. et al. *Advanced compiler design implementation*. – Morgan kaufmann, **1997**.

[2] Cooper K. D., Torczon L. *Engineering a compiler*. – Elsevier, **2022**.

[3] Matthias Braun. *Register Allocation: More than Coloring*. **2018**.

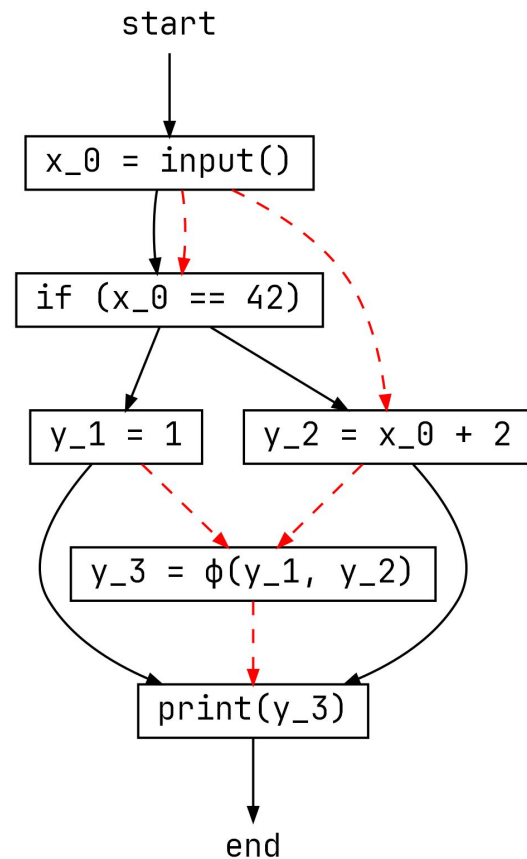
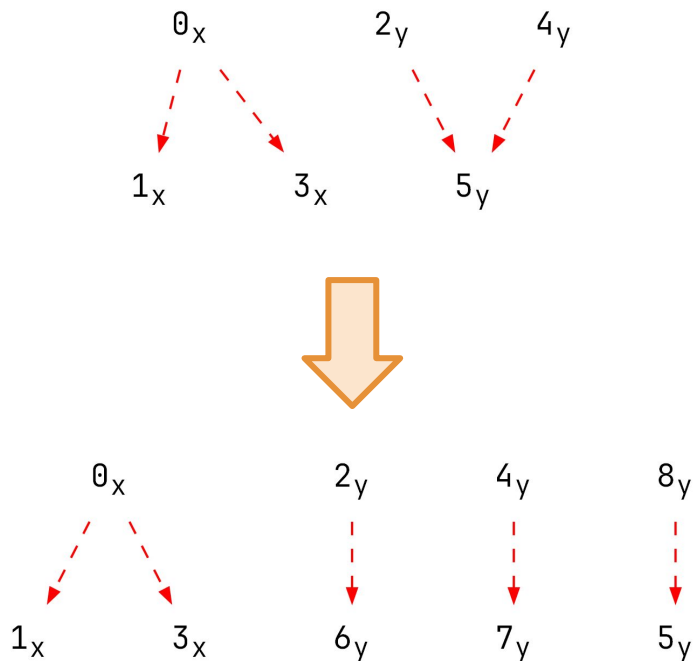
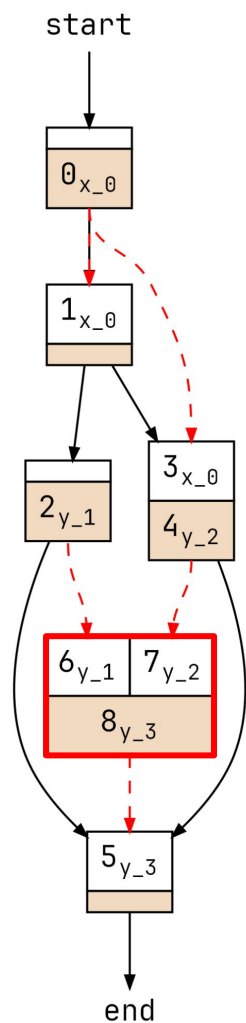
Р-схема не обладает свойством SSA (для экономии памяти это и не нужно)



Два присваивания для y_1.

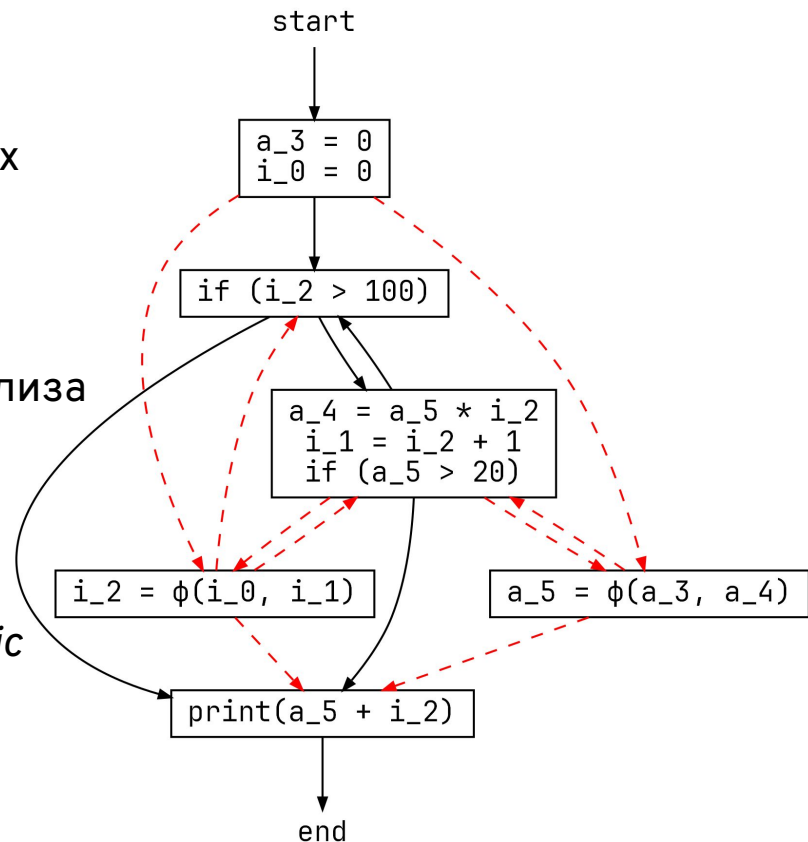
А если добавить ϕ -функции?

Соберем результаты для общего аргумента и направим их в **созданную ϕ -функцию** (с нумерацией значений).



Рф-схема: что получилось

- Получилось **графовое промежуточное представление** в духе PDG, SoN, VSDG, ...
- Алгоритм его построения **проще** самых простых алгоритмов построения формы SSA [1].
В Рф-схеме ϕ -функции **не привязаны** к управляющему графу.
- Рф-схема может использоваться в задачах анализа там, где сегодня используется **SSA graph** [2].



[1] Braun M. et al. *Simple and efficient construction of static single assignment form* //Lecture Notes in Computer Science. – **2013**. – T. 7791. – C. 102-122.

[2] Rastello F., Tichadou F. B. (ed.). *SSA-based Compiler Design*. – Springer, **2022**.

1. Введение

2. Пример экономии памяти

3. Фазы экономии памяти

3.1. Переименование
переменных

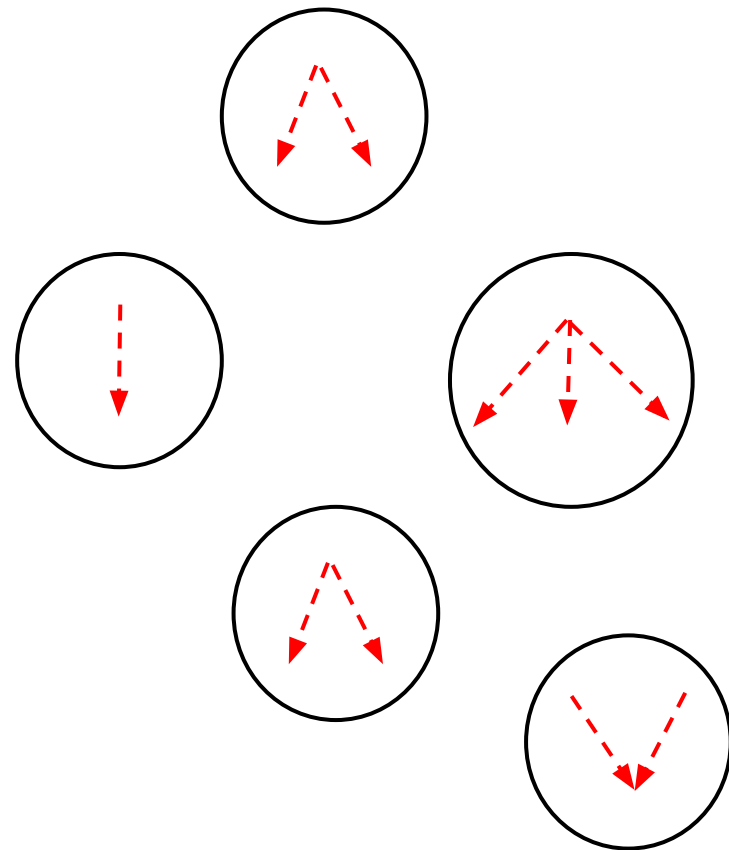
**3.2. Граф
несовместимости**

3.3. Раскраска графа
несовместимости

4. Экономия памяти для массивов

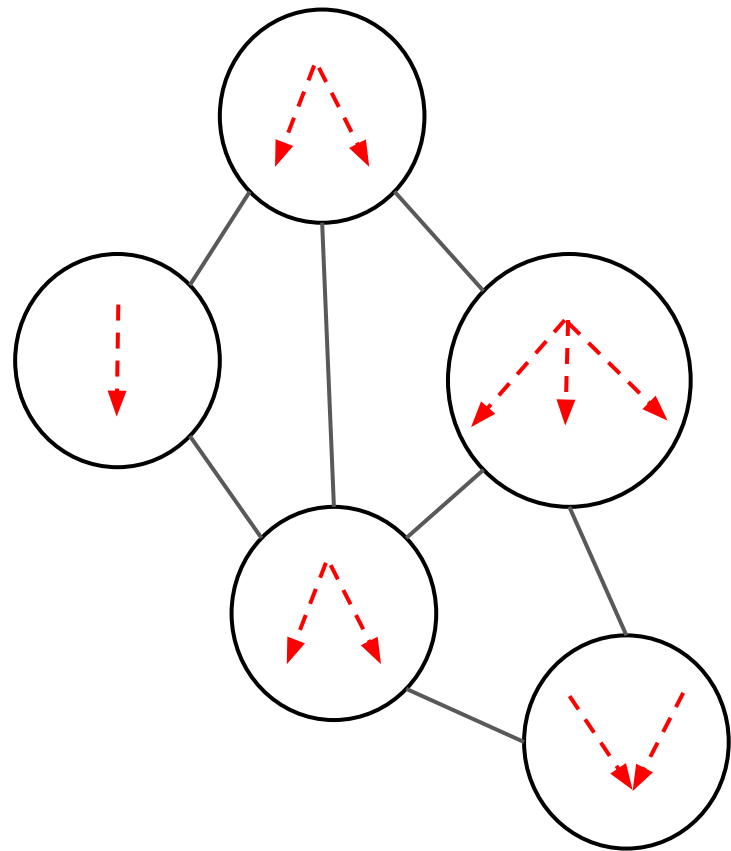
5. Заключение

Вершинами графа несовместимости являются области действия, то есть **компоненты связности** информационного графа.



Вершинами графа несовместимости являются **области действия**, то есть **компоненты связности** информационного графа.

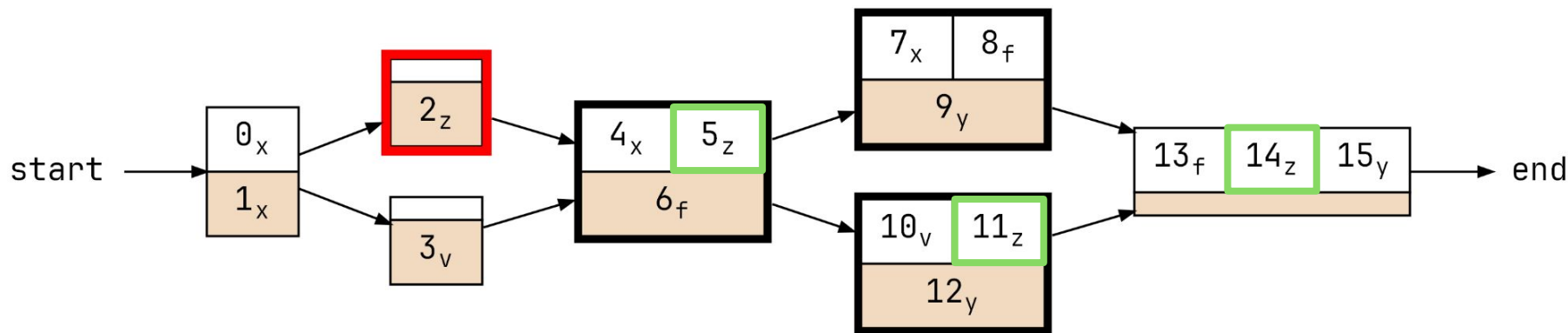
Если две вершины **несовместимы**, между ними создается **ребро**.

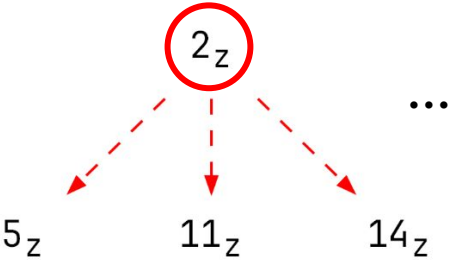


Две вершины несовместимы, если:

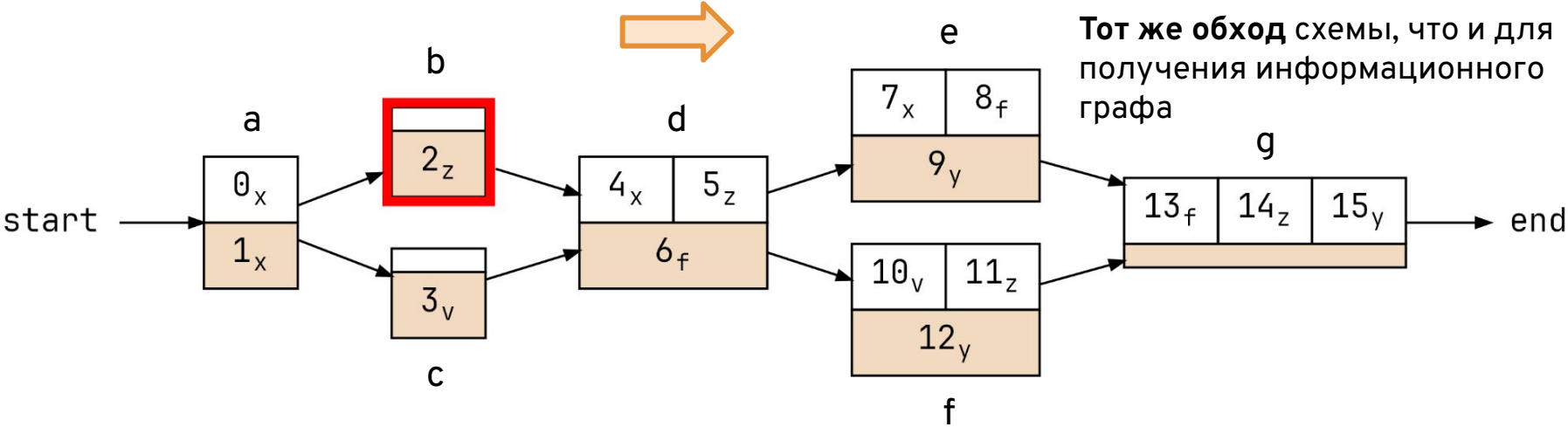
- Совпадают их операторы-результаты.
- Оператор-результат одной вершины входит в множество **транзитных операторов** другой вершины.

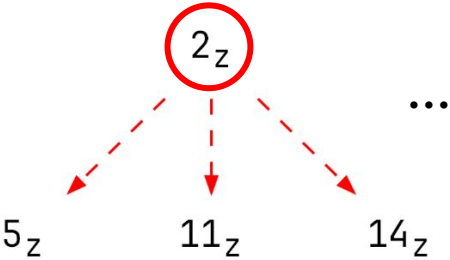
Транзитные операторы – операторы на пути от **оператора-результата** (красный) к **последнему оператору-аргументу**:



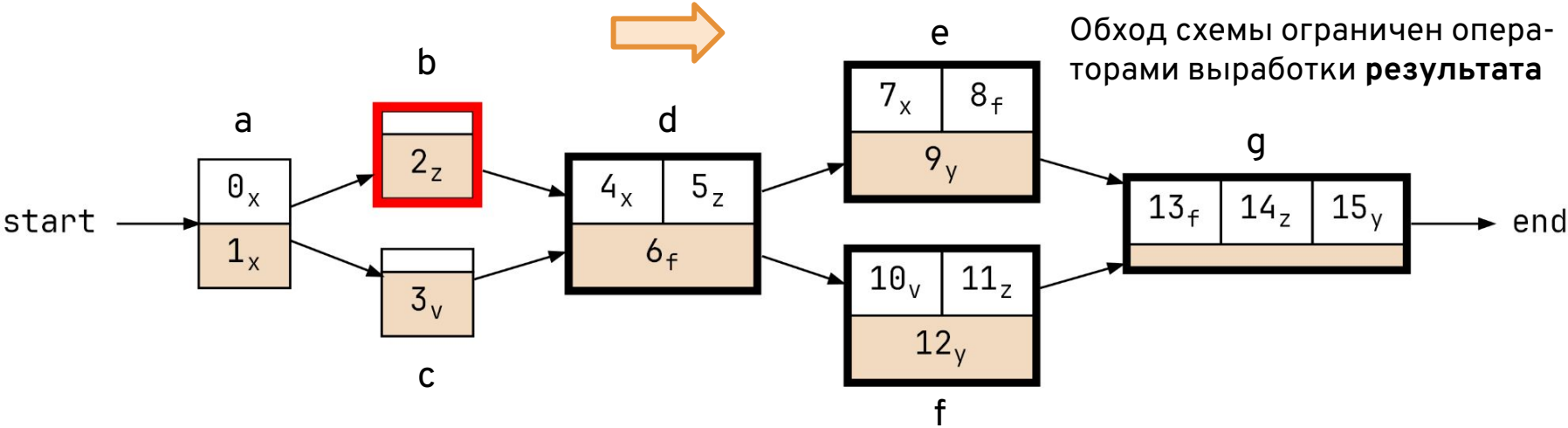


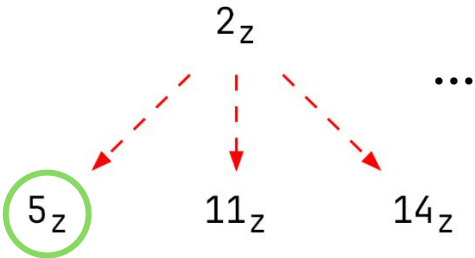
Прямой обход E	



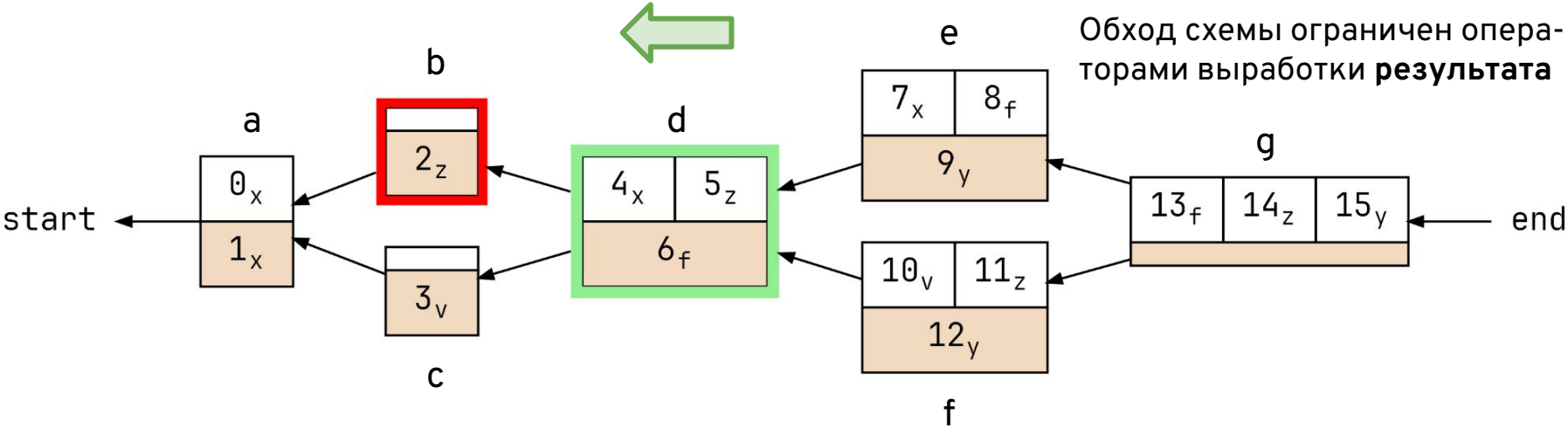


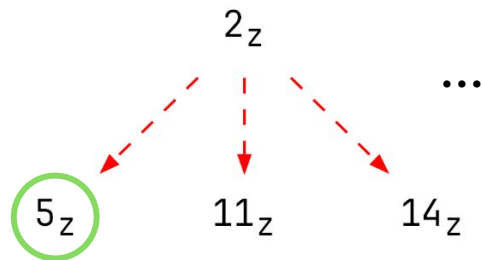
Прямой обход E	d, e, f, g



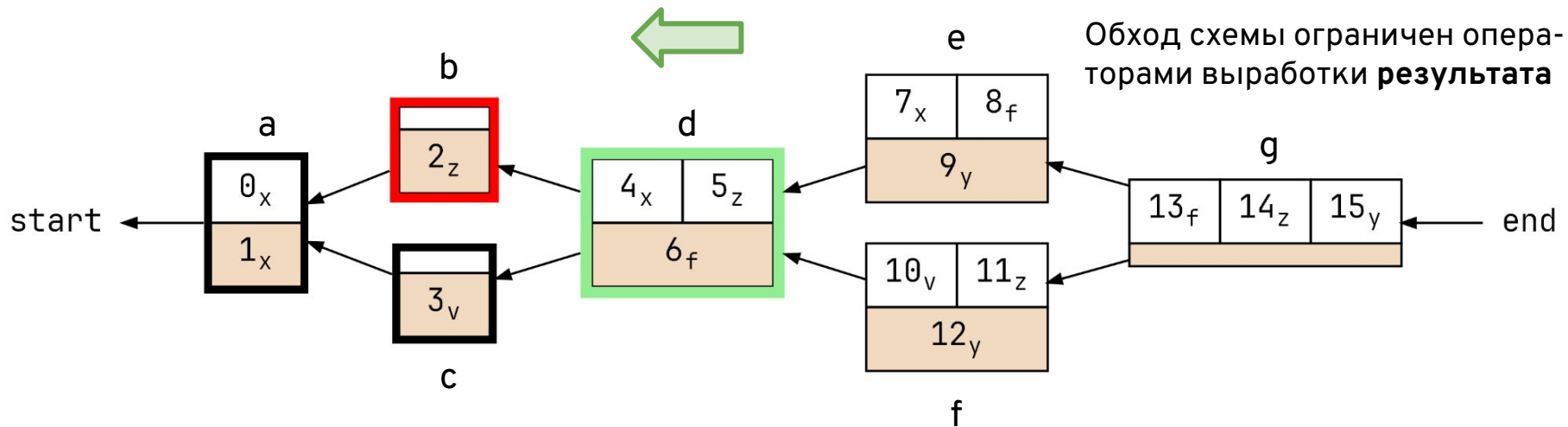


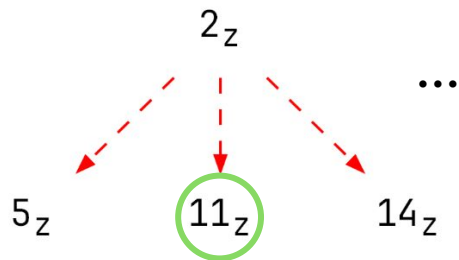
Прямой обход E	d, e, f, g
Обратный обход L	



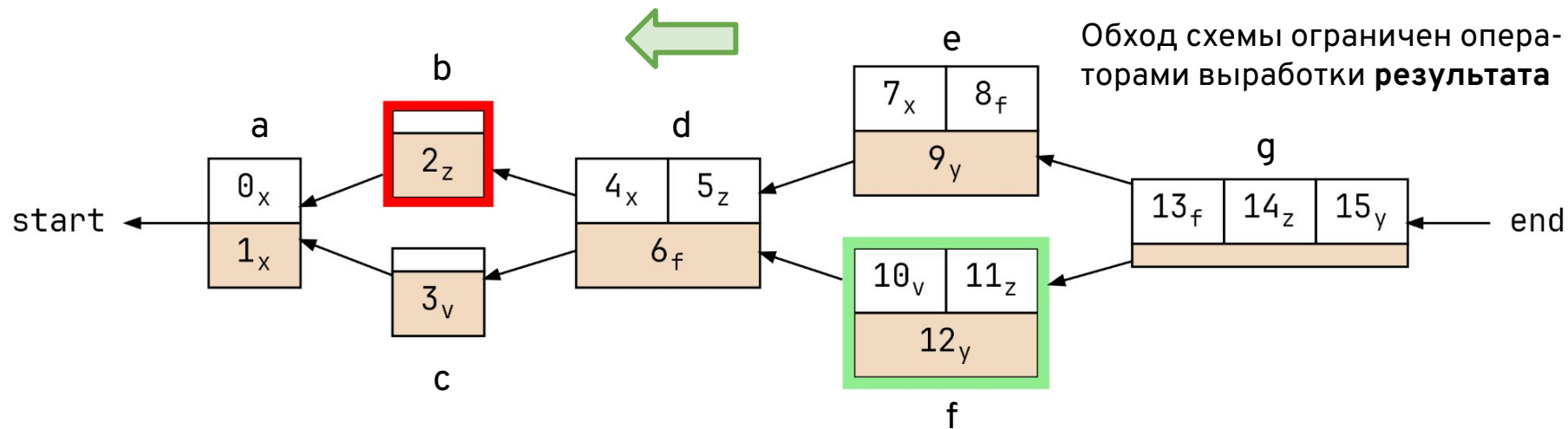


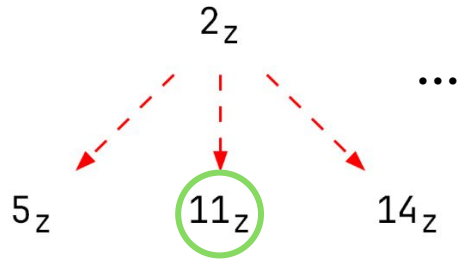
Прямой обход E	d, e, f, g
Обратный обход L	c, a



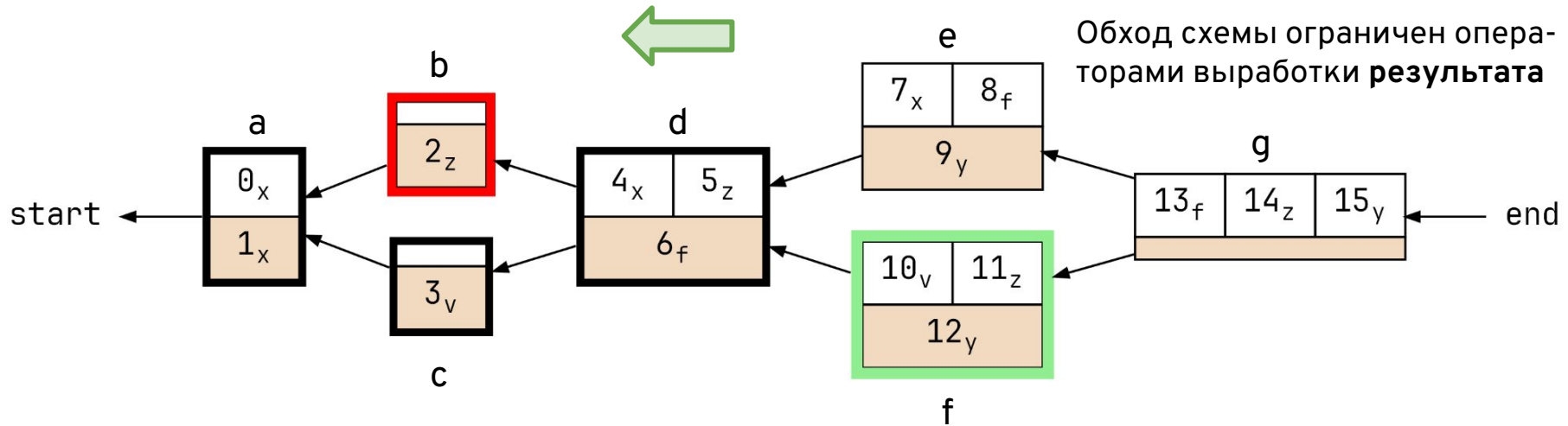


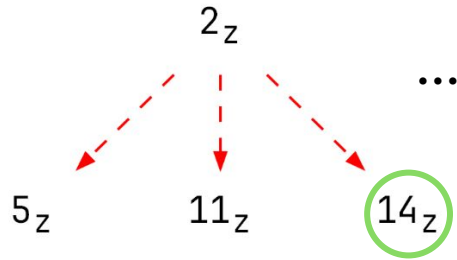
Прямой обход E	d, e, f, g
Обратный обход L	c, a



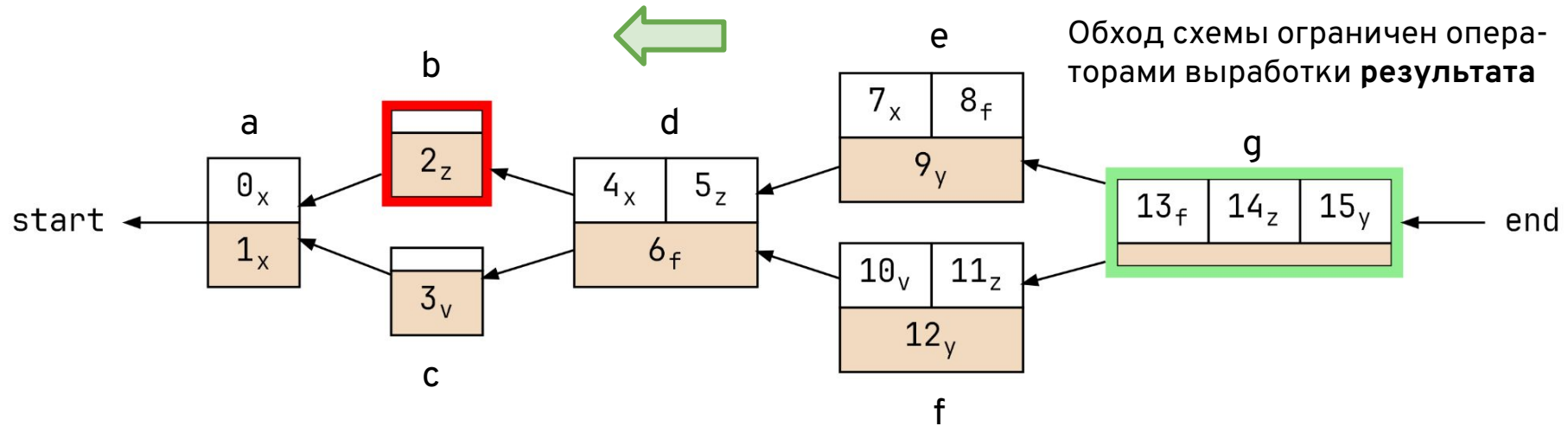


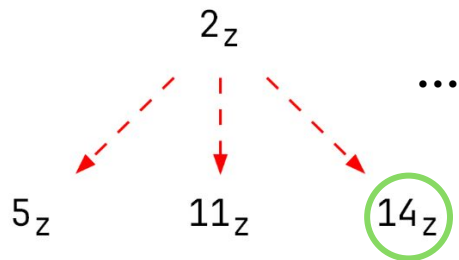
Прямой обход E	d, e, f, g
Обратный обход L	c, a, d



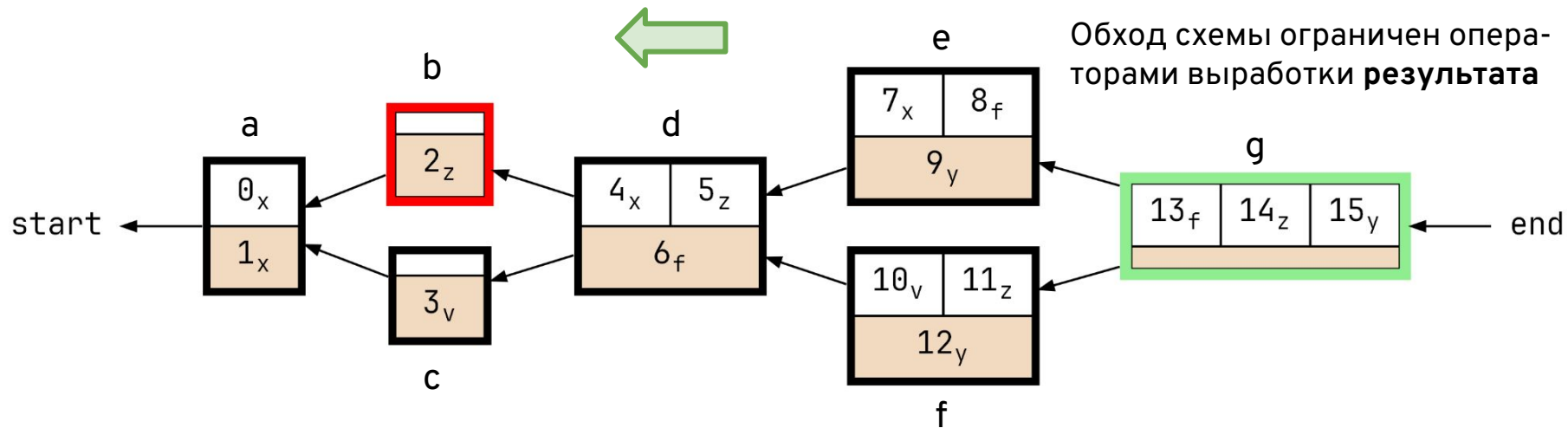


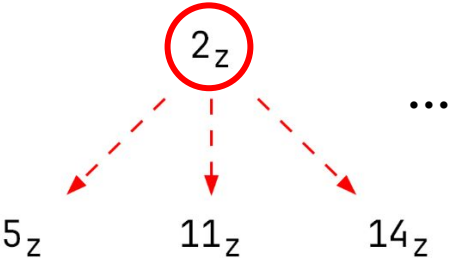
Прямой обход E	d, e, f, g
Обратный обход L	c, a, d



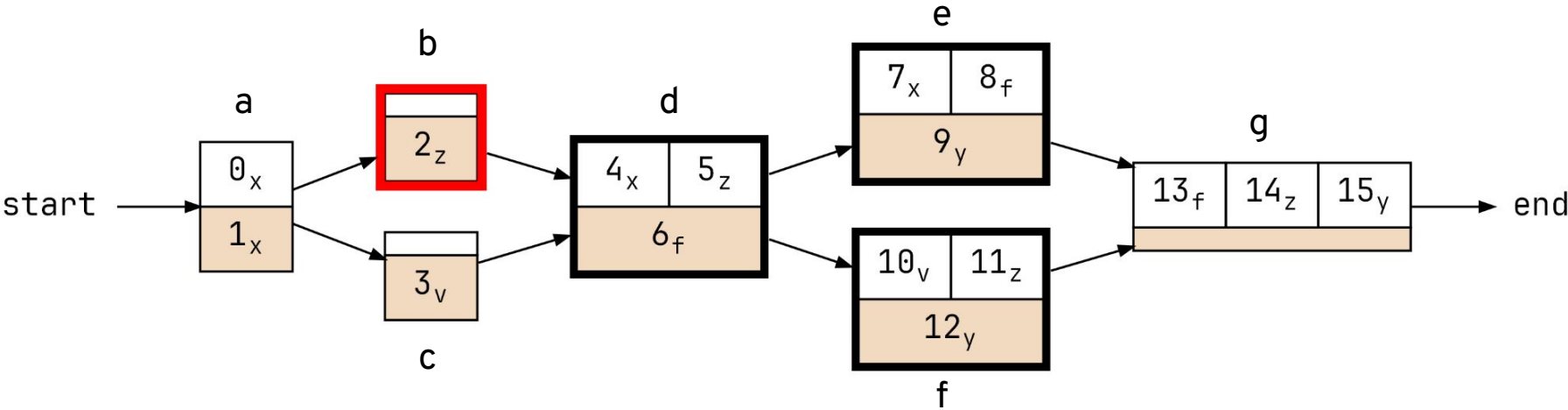


Прямой обход E	d, e, f, g
Обратный обход L	c, a, d, e, f





Прямой обход E	d, e, f, g
Обратный обход L	c, a, d, e, f
$E \cap L$	d, e, f



Зачем понадобился прямой обход?

Если на обратном пути по схеме **от аргумента** мы всегда **найдем** оператор, вырабатывающий **результат** для этого аргумента, то:

1. прямой обход не нужен,
2. такая схема называется **замкнутой** (по Лаврову [1]).

Сегодня программу с замкнутой схемой называют **strict program** [2].

[1] Лавров С. С. *Об экономии памяти в замкнутых операторных схемах* //Журнал вычислительной математики и математической физики. – **1961**. – Т. 1. – №. 4. – С. 687-701.

[2] Budimlic Z. et al. *Fast copy coalescing and live-range identification* //ACM SIGPLAN Notices. – **2002**. – Т. 37. – №. 5. – С. 25-32.

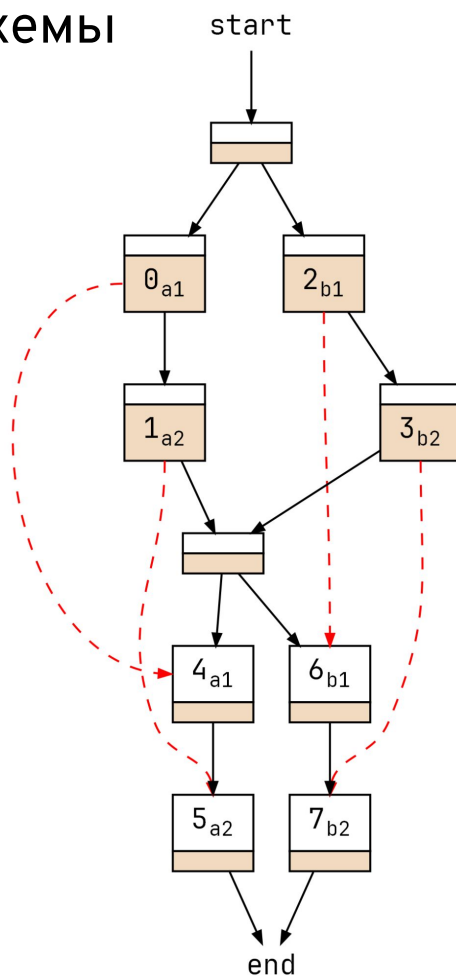
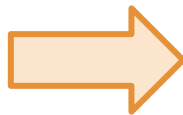
```
int a1, a2, b1, b2;
if (...) {
    a1 = ...;
    a2 = ...;
} else {
    b1 = ...;
    b2 = ...;
}
if (...) {
    print(a1);
    print(a2);
} else {
    print(b1);
    print(b2);
}
```

Пример незамкнутой схемы

```

int a1, a2, b1, b2;
if (...) {
    a1 = ...;
    a2 = ...;
} else {
    b1 = ...;
    b2 = ...;
}
if (...) {
    print(a1);
    print(a2);
} else {
    print(b1);
    print(b2);
}

```

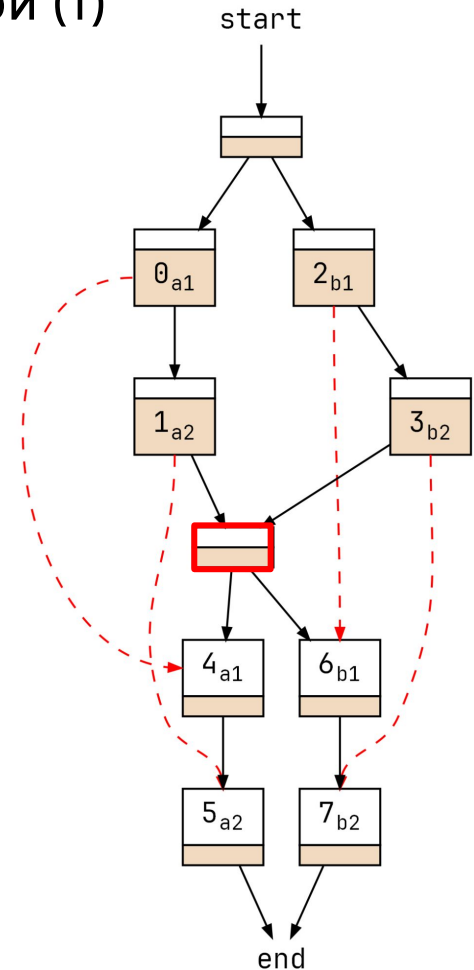
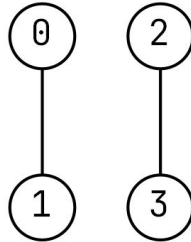


В Java, Swift и Rust
это НЕВОЗМОЖНО

Проблема с незамкнутой схемой (1)

Два определения несовместимости:

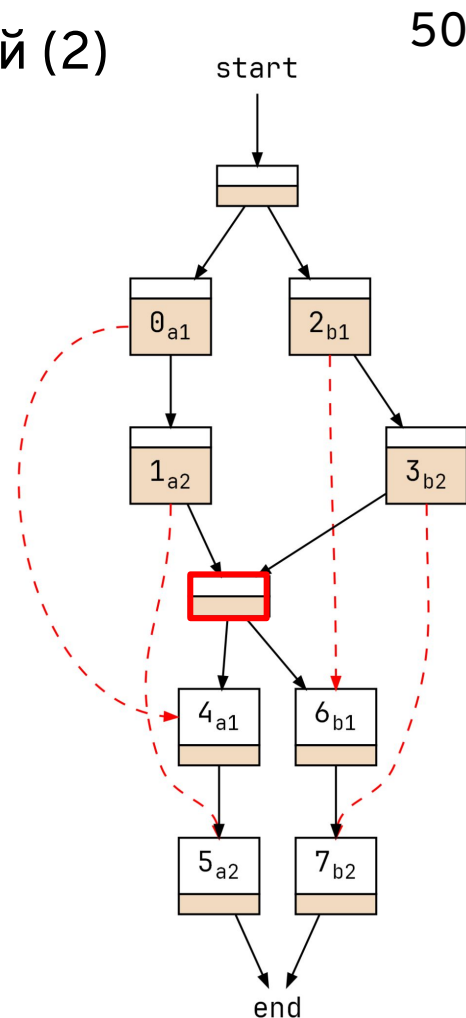
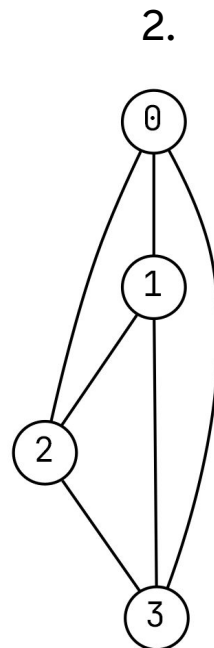
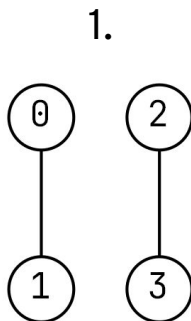
1. Две области действия несовместимы, если **оператор-результат** одной области находится **среди операторов** другой области действия.



Проблема с незамкнутой схемой (2)

Два определения несовместимости:

1. Две области действия несовместимы, если **оператор-результат** одной области находится **среди операторов** другой области действия.
2. Две области действия несовместимы, если у них **есть общие операторы**.



- Задача получения множества транзитных операторов – анализ жизни переменных (**liveness analysis**).
- Классический способ решения – с помощью потоковых уравнений (**data-flow equations**) [1].
- Сегодня все чаще используется подход **one variable at a time** [2] и легковесный **анализ форме в SSA** [3]. Это ближе к тому, что было у Ершова.

[1] Kildall G. A. *A unified approach to global program optimization* //Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages. – **1973**. – С. 194-206.

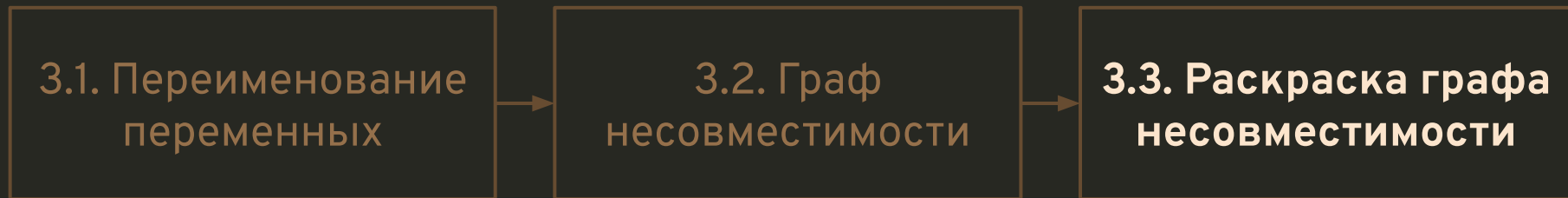
[2] Appel A. W. *Modern Compiler Implementation in ML*. Cambridge University Press, **1998**.

[3] Boissinot B. et al. *Fast liveness checking for SSA-form programs* //Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization. – **2008**. – С. 35-44.

1. Введение

2. Пример экономии памяти

3. Фазы экономии памяти



4. Экономия памяти для массивов

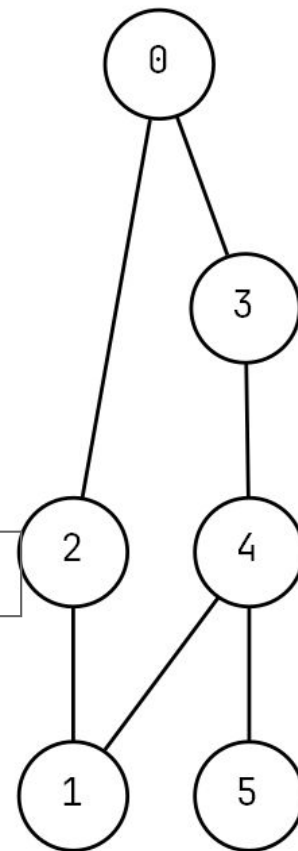
5. Заключение

Задача раскраски графа – **NP-полная**.

Для минимальной раскраски имеется порядок склеивания пар вершин **на взаимном расстоянии 2** в связном графе (по теореме Кожухина [1]).

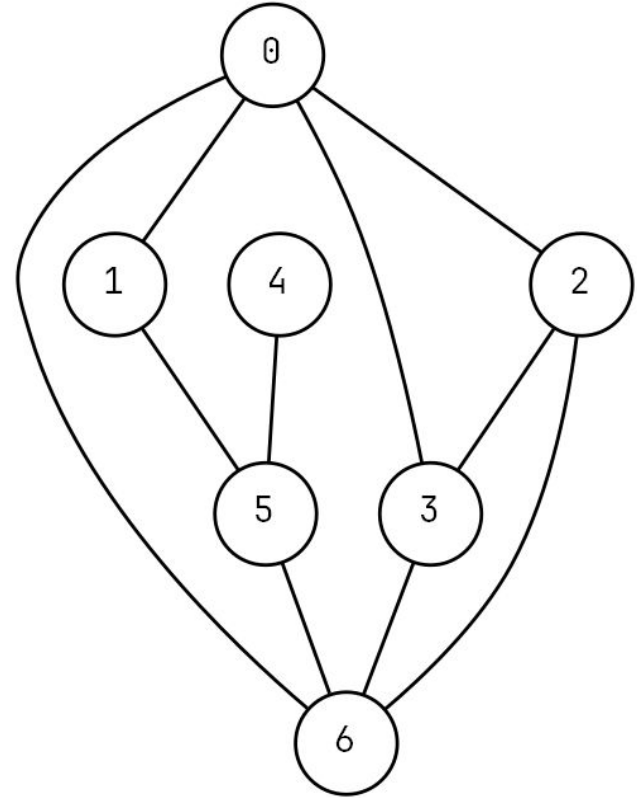
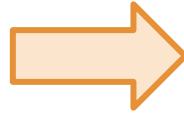
Этот порядок узнать сложно, поэтому используются эвристики.

Вершины 1 и 0
на расстоянии 2



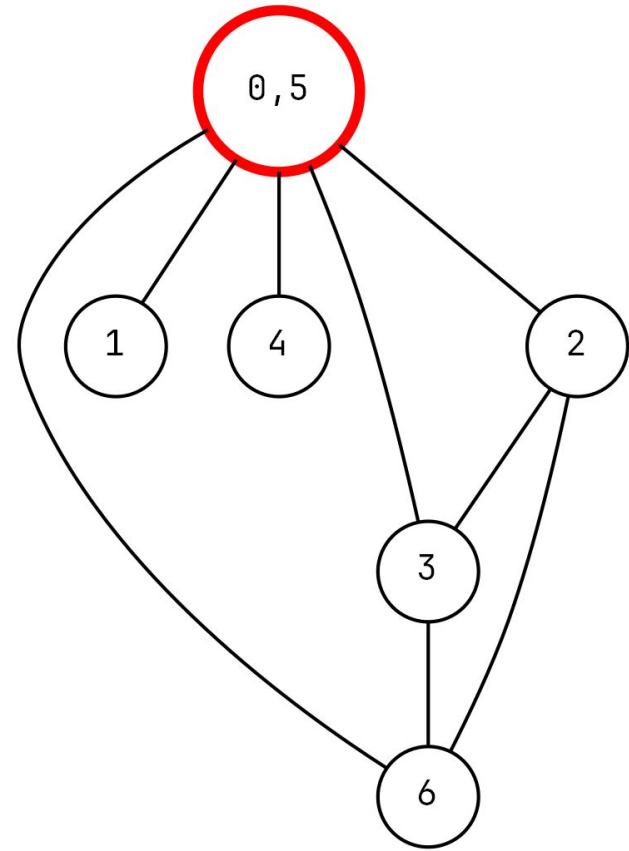
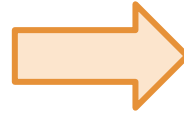
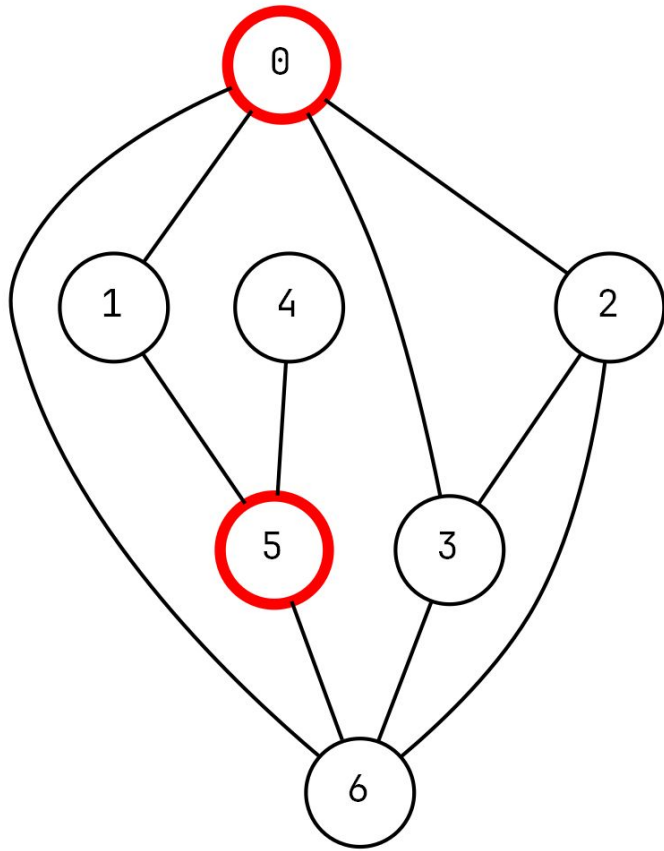
[1] Ершов А. П., Кожухин Г. И. *Об оценках хроматического числа связных графов* // Доклады Академии наук. – Российская академия наук, **1962**. – Т. 142. – №. 2. – С. 270-273.

$v_0 \neq v_6$
 $v_0 \neq v_2$
 $v_0 \neq v_3$
 $v_0 \neq v_1$
 $v_1 \neq v_5$
 $v_2 \neq v_3$
 $v_2 \neq v_6$
 $v_3 \neq v_6$
 $v_4 \neq v_5$
 $v_5 \neq v_6$



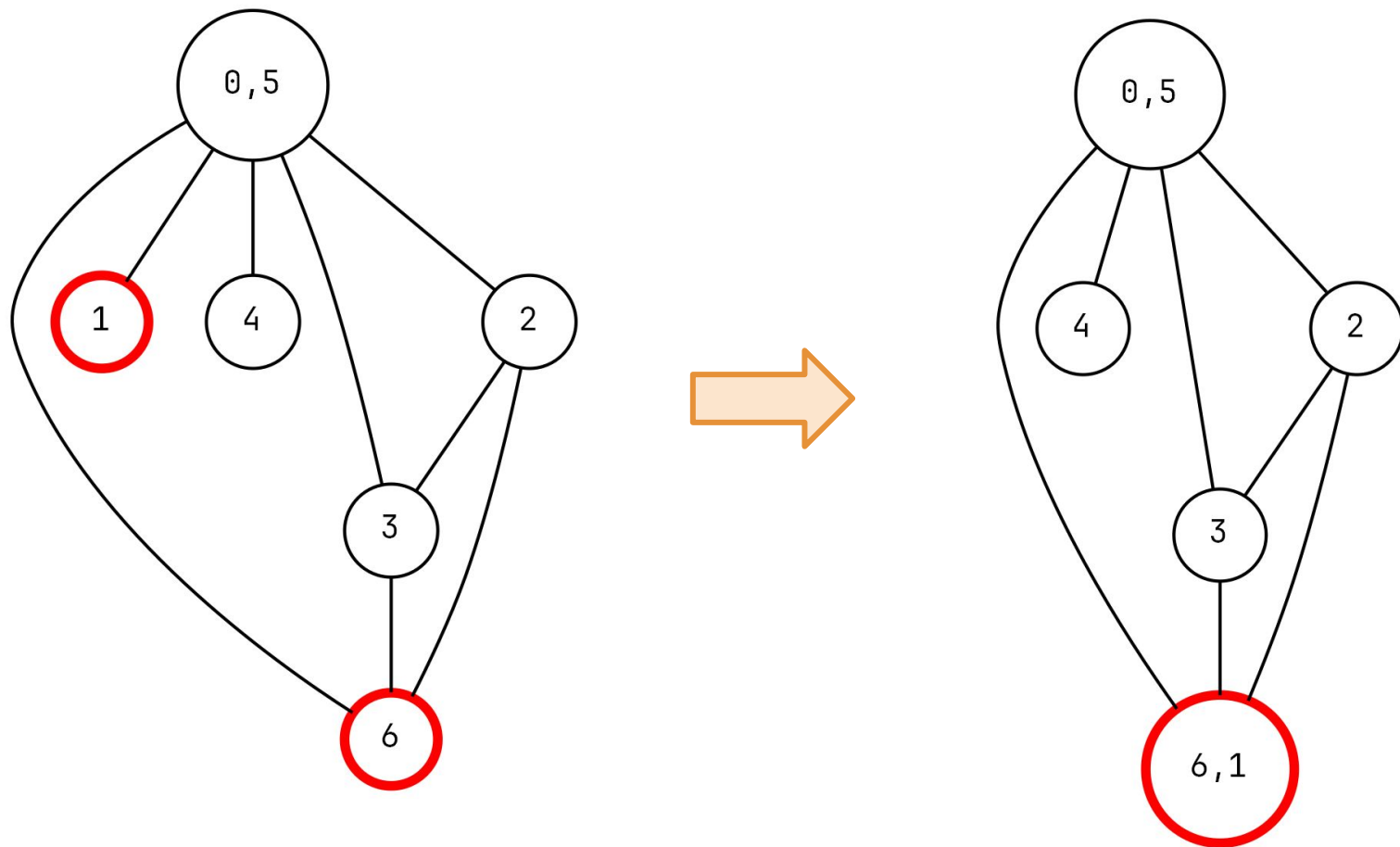
Склеивание вершин на расстоянии 2 (1)

55 из 82



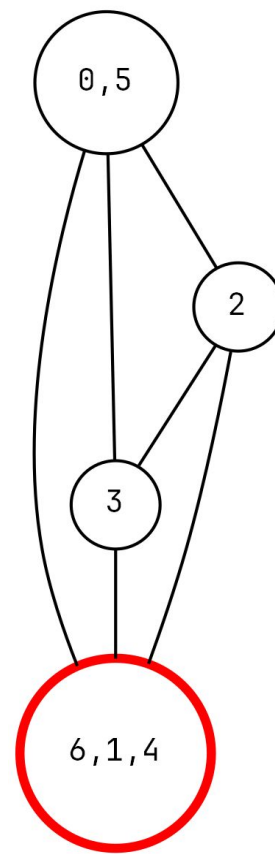
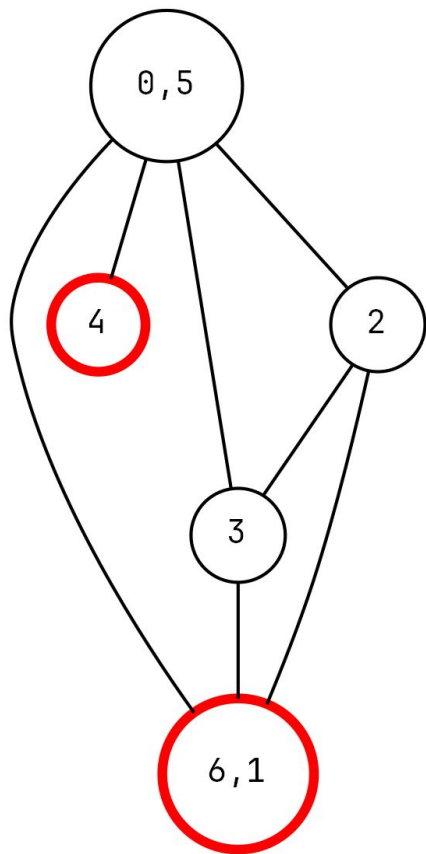
Склеивание вершин на расстоянии 2 (2)

56 из 82

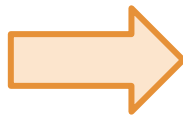
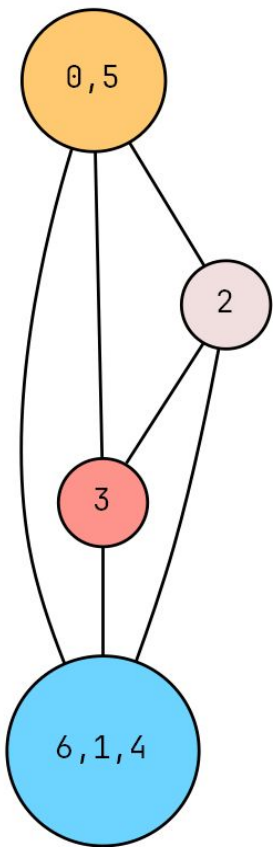


Склеивание вершин на расстоянии 2 (3)

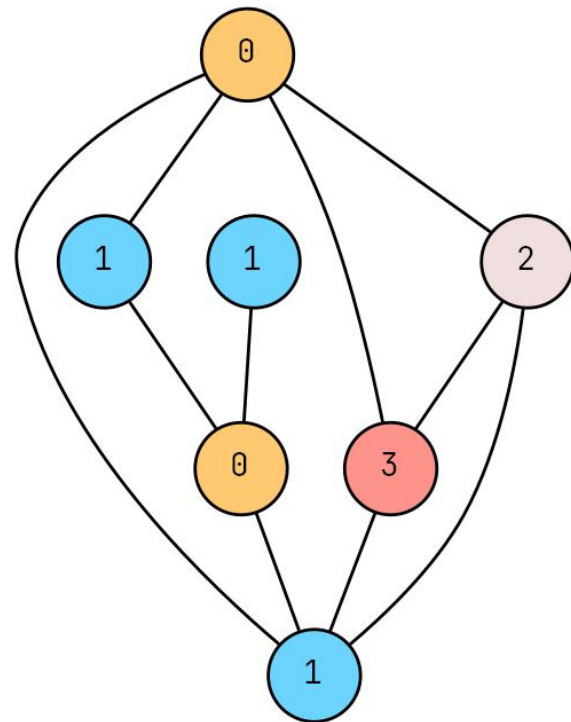
57 из 82



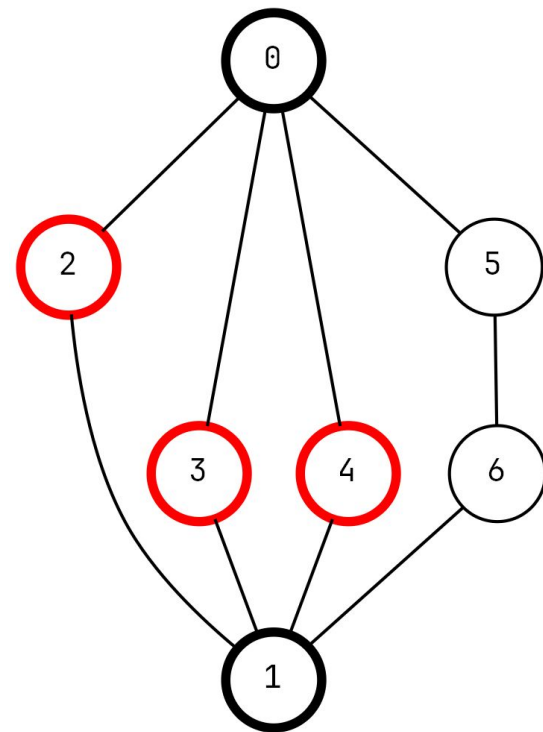
Полный граф!



Достаточно 4 ячеек вместо 7!



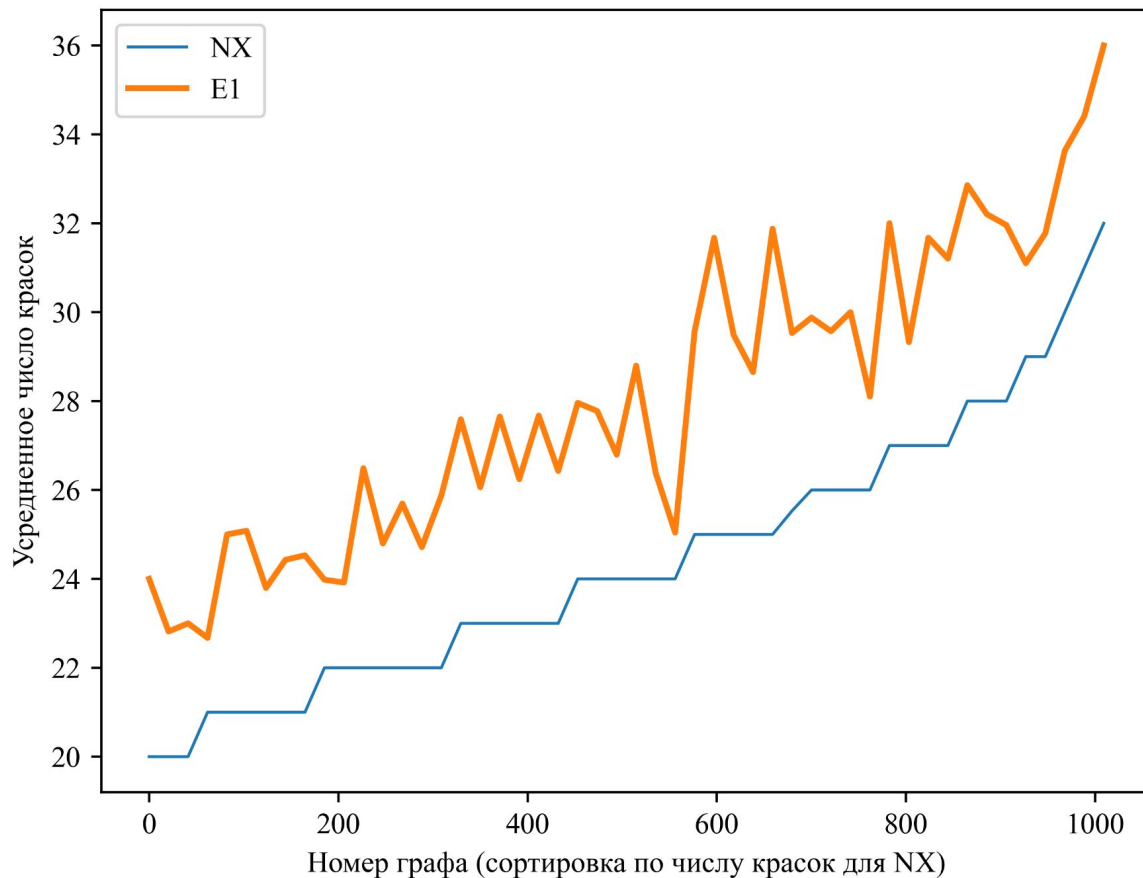
1. **E1** — пройти по всем вершинам, поочередно склеивая каждую вершину со всеми ее соседями на расстоянии 2.
2. **E2** — как E1, но начинаем склеивание с соседей, имеющих максимум **разделяющих** вершин с нашей вершиной.
3. **E3** — пройти по всем парам вершин на расстоянии 2, поочередно склеивая очередную пару, имеющую максимум **разделяющих** вершин. **Сложность $O(n^3)$!**



Вершины 0 и 1 разделяются вершинами **2, 3 и 4**.

Актуален ли алгоритм Ершова-Кожухина? E1 против NX 60 из 82

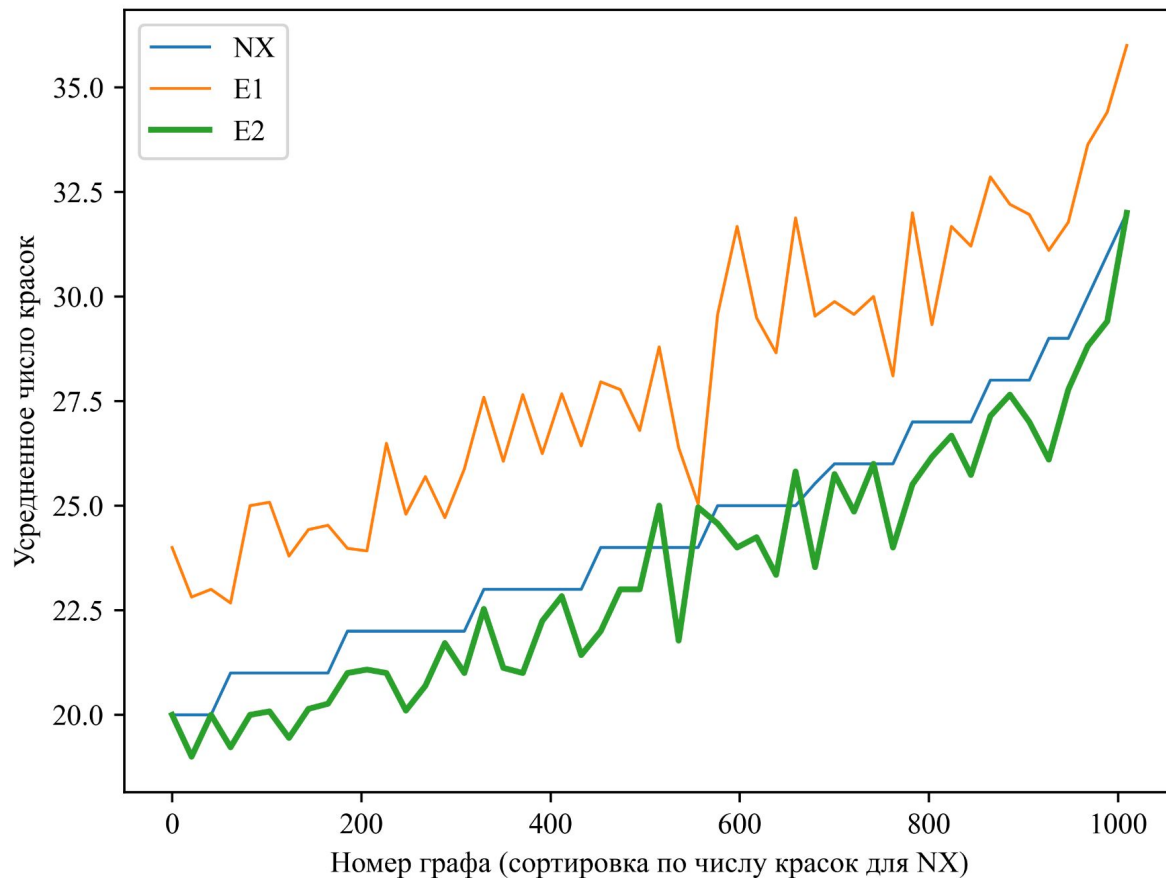
$NX = \min(\text{Welsh-Powell}, \text{Recursive-Largest-First}, \text{DSATUR})$



Е2 против NX

61 из 82

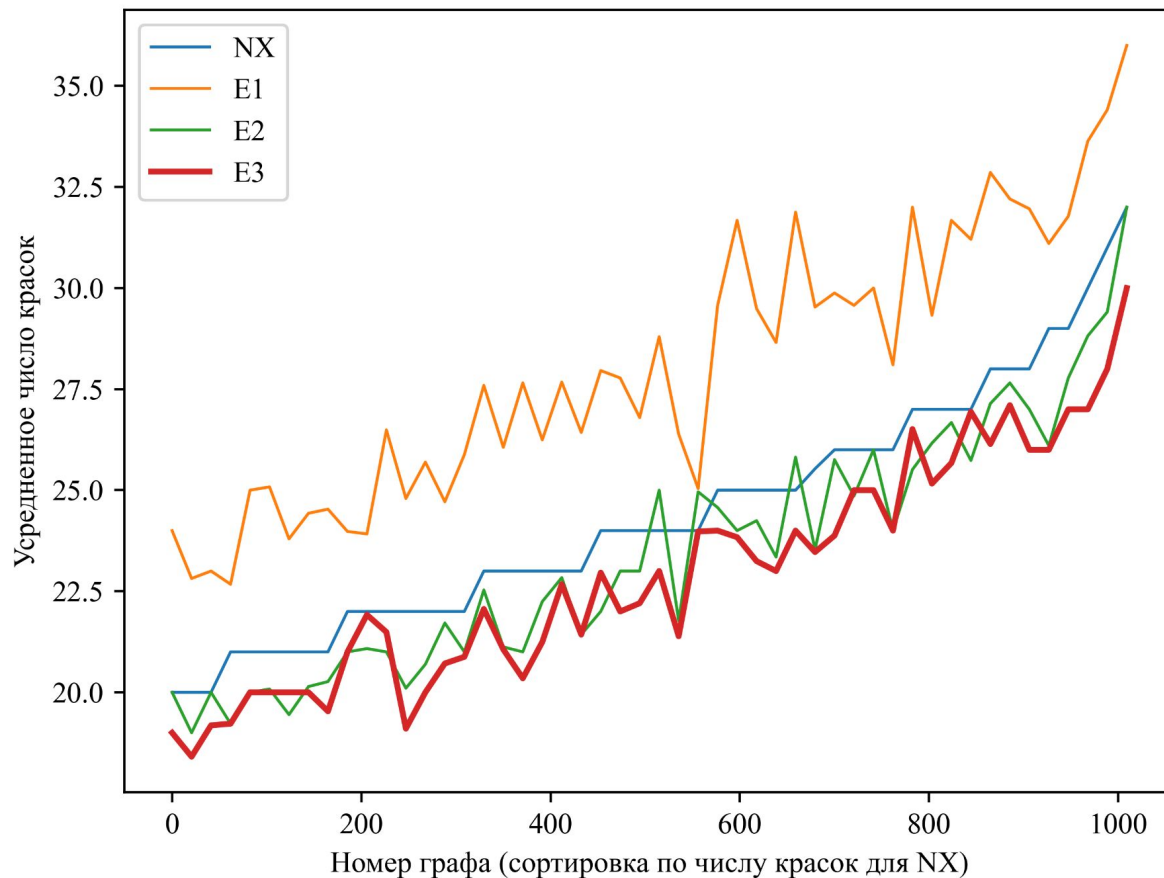
$NX = \min(\text{Welsh-Powell}, \text{Recursive-Largest-First}, \text{DSATUR})$



Е3 против NX

62 из 82

$NX = \min(\text{Welsh-Powell}, \text{Recursive-Largest-First}, \text{DSATUR})$



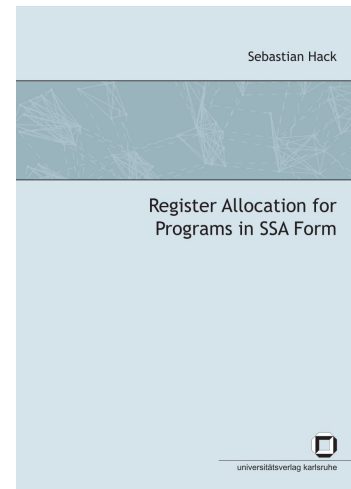
- В экспериментальном компиляторе PL/I [1] решалась **распределения ограниченного числа регистров** на основе раскраски графа и с учетом **выгрузки регистров в память** (spilling).
- Публикация об Альфа-трансляторе опередила эту работу на 17 лет [2], но речь шла не о регистрах, а о **неограниченной** памяти.

[1] Chaitin G. J. *Register allocation & spilling via graph coloring* //ACM Sigplan Notices. – **1982**. – Т. 17. – №. 6. – С. 98-101.

[2] Г. И. Бабецкий, М. М. Бежанова, Ю. М. Волошин, А. П. Ершов, Б. А. Загацкий, Л. Л. Змиевская, Г. И. Кожухин, С. К. Кожухина, Р. Д. Мишкович, Ю. И. Михалевич, И. В. Поттосин, Л. К. Трохан, “Система автоматизации программирования АЛЬФА”, Ж. вычисл. матем. и матем. физ., 5:2 (**1965**), 317–325

Иногда **отказываются** от построения графа несовместимости:

- В архитектурах с небольшим числом регистров ключевыми являются: 1) задача минимизации **выгрузки регистров в память**, 2) задача минимизации числа **пересылок между регистрами** [1].
- Если распределять регистры **в форме SSA**, то граф несовместимости хордален — его раскраска выполняется за $O(V^2)$ [2]. Многие связанные проблемы остаются NP-полными.
- Сложно учитывать **архитектурные особенности** на уровне графа несовместимости, поэтому используют программирование в ограничениях [3] и машинное обучение [4].



[1] Olesen J. *Register allocation in llvm 3.0* //LLVM Developers' Meeting. – **2011**.

[2] Hack S., Grund D., Goos G. *Register Allocation for Programs in SSA-Form*. **2006**.

[3] Lozano R. C. et al. *Constraint-based Register Allocation and Instruction Scheduling*. **2012**.

[4] VenkataKeerthy S. et al. *RL4ReAl: Reinforcement Learning for Register Allocation*. **2023**.

Задача экономии памяти на основе раскраски графа сегодня чаще применяется там, где возможно:

- Использовать изначально **неограниченное** число красок.
- Легко **добавить** нужные **ограничения** в граф несовместимости.

В частности, в LLVM используется раскраска графа для **экономии памяти на стеке** – см. проходы:

- StackSlotColoring. Экономия слотов стека для выгрузки регистров.
- StackColoring. **Экономия массивов.**

1. Введение

2. Пример экономии памяти

3. Фазы экономии памяти

3.1. Переименование
переменных



3.2. Граф
несовместимости



3.3. Раскраска графа
несовместимости

4. Экономия памяти для массивов

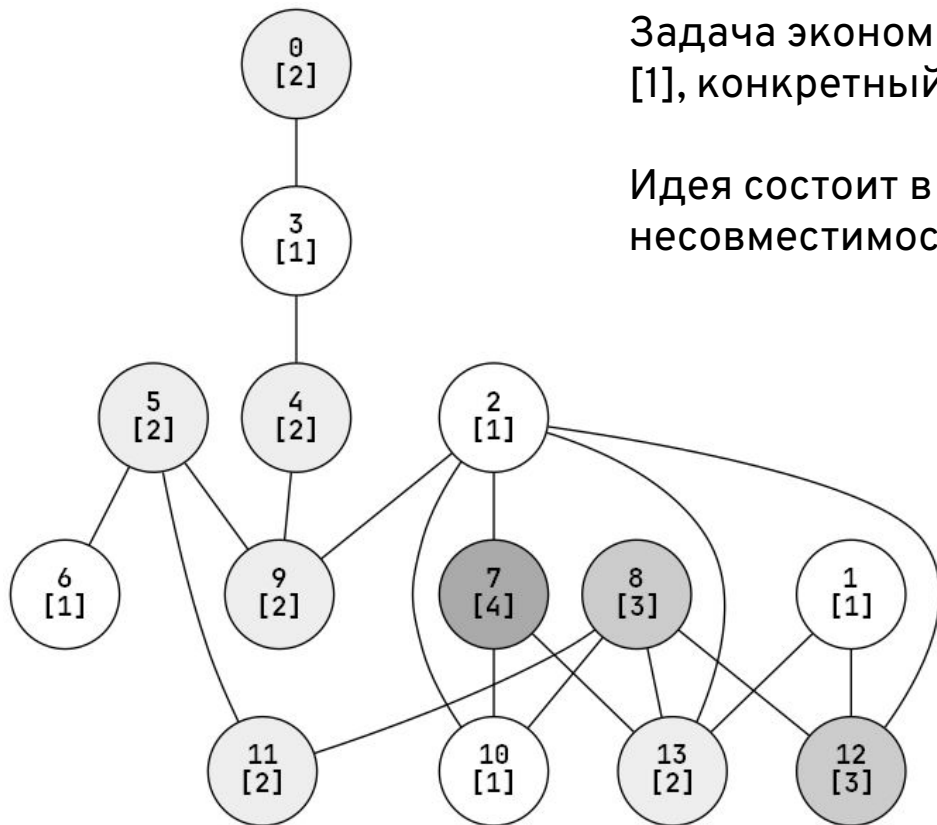
5. Заключение

- Экономия для массивов на стеке — проход StackColoring в LLVM.
- Экономия размещения в векторных регистрах и учета других архитектурных особенностей [1].
- Глобальная экономия массивов во встраиваемых системах без рекурсии.

[1] Smith M. D., Holloway G. *Graph-coloring register allocation for irregular architectures* //Submitted to PLDI. – **2000**. – Т. 1. – С. 1-8.

Задача экономии памяти для массивов сформулирована в [1], конкретный алгоритм описан в [2].

Идея состоит в раскраске **взвешенного** графа несовместимости.

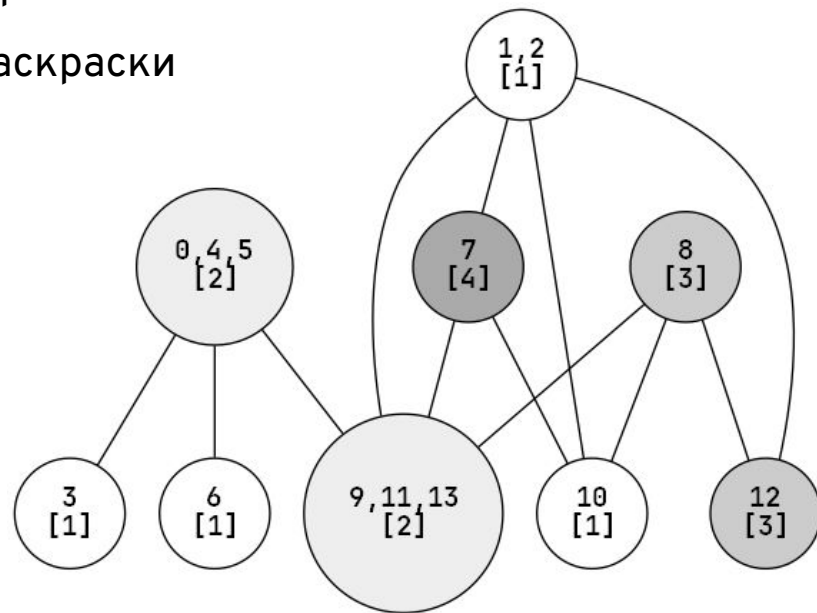
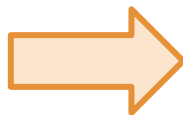
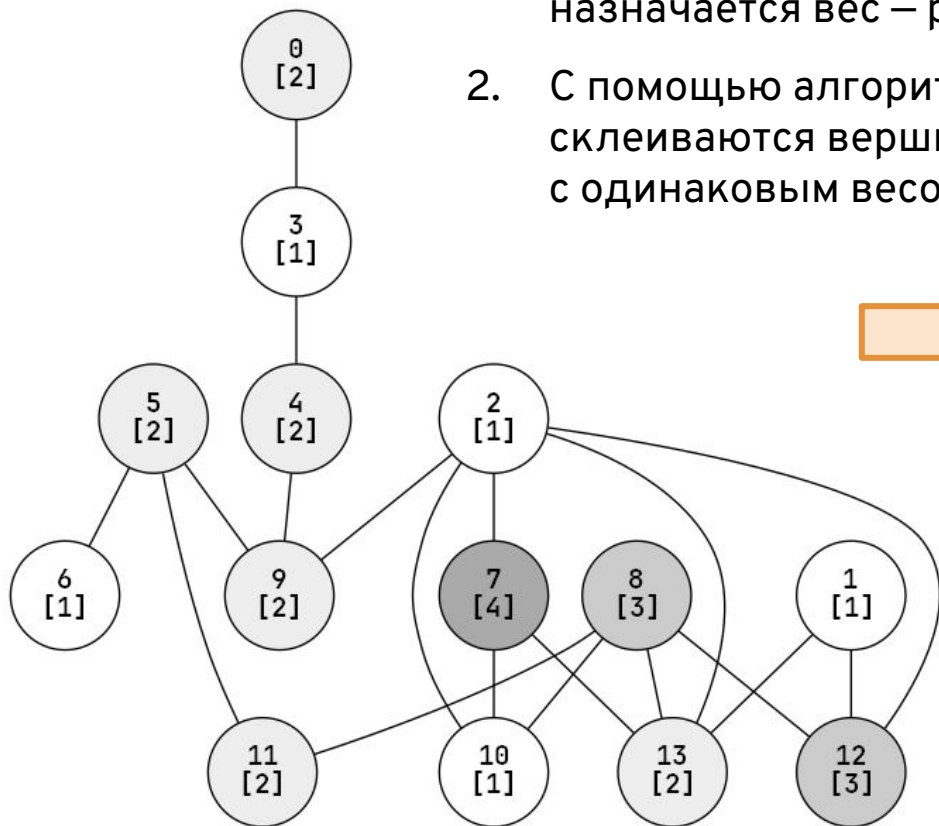


[1] Ершов А. П. *Сведение задачи распределения памяти при составлении программ к задаче раскраски вершин графов* // Доклады Академии наук. – Российская академия наук, **1962**. – Т. 142. – №. 4. – С. 785-787.

[2] Бабецкий Г. И. и др. *АЛЬФА-система автоматизации программирования*. – **1967**.

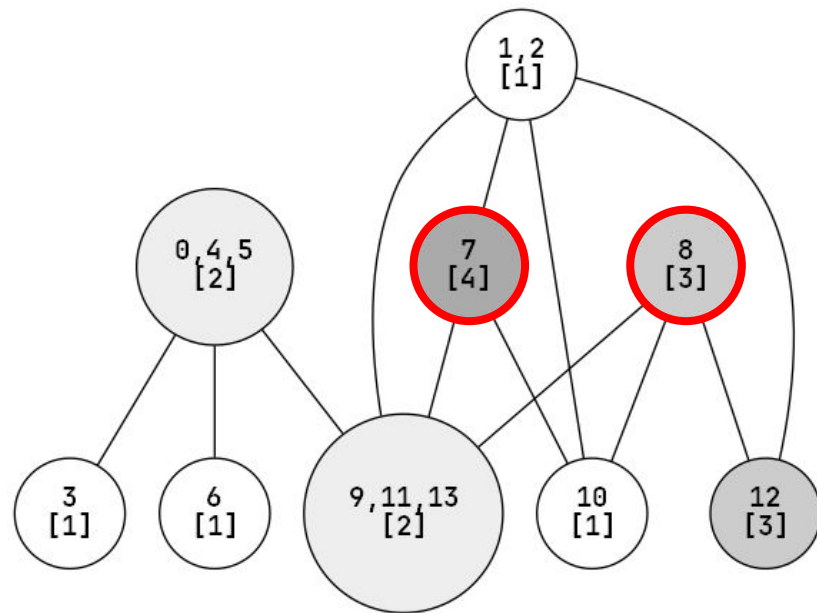
Экономия памяти для массивов на примере (1)

1. Каждой вершине в графе несовместимости назначается вес – размер массива.
2. С помощью алгоритма раскраски склеиваются вершины с одинаковым весом.



Вершины 3 и 6 можно было склеить!

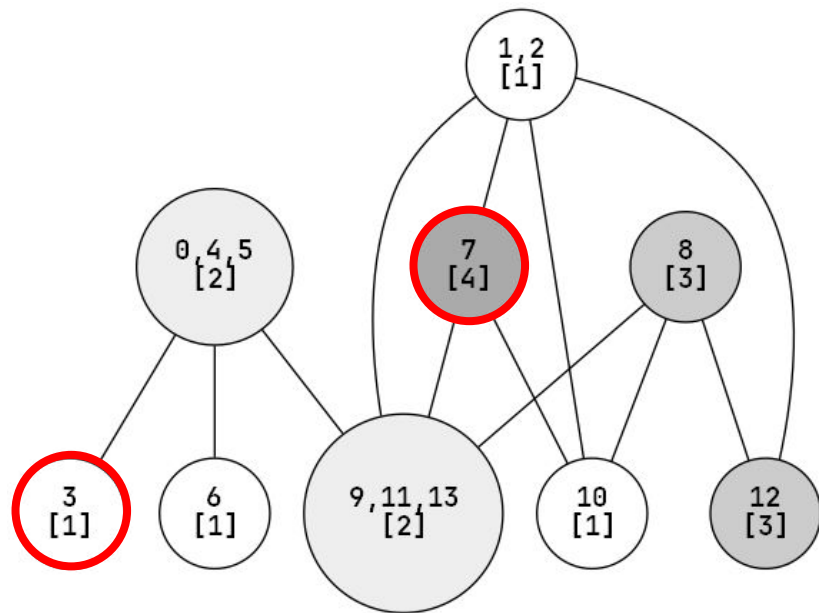
- Для каждой вершины n , начиная с наибольшего веса, ищем совместимые с ней вершины, тоже от наибольшей, и пытаемся их упаковать в массив n .



0	1	2	3
		7	
	8		

Наполняем массив вершины **7**. Добавляем туда **8**.

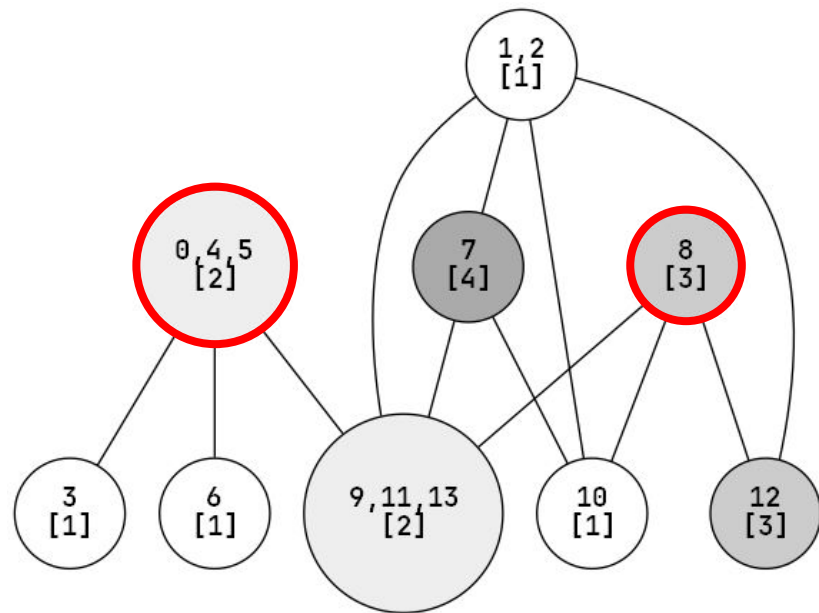
- Для каждой вершины n , начиная с наибольшего веса, ищем совместимые с ней вершины, тоже от наибольшей, и пытаемся их упаковать в массив n .



0	1	2	3
		7	
	8		3

Наполняем массив вершины 7. Добавляем туда 3.

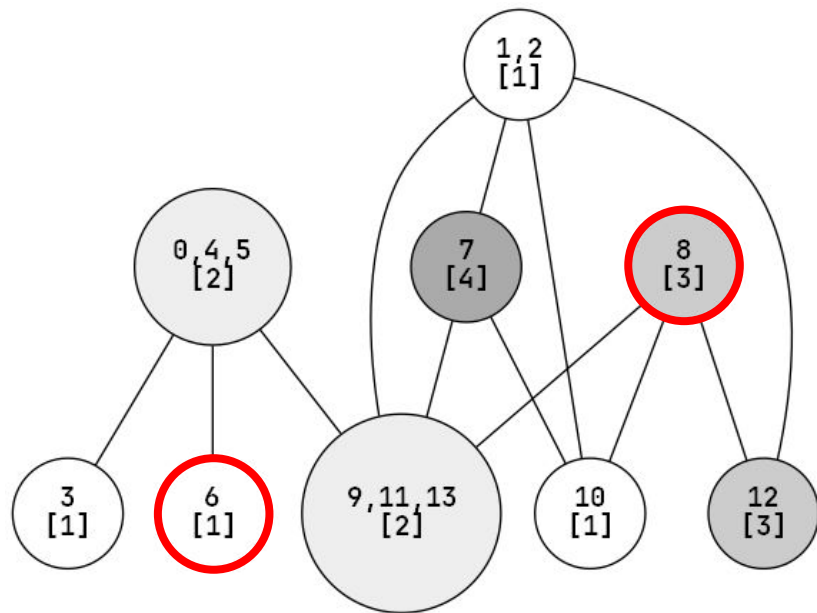
- Для каждой вершины n , начиная с наибольшего веса, ищем совместимые с ней вершины, тоже от наибольшей, и пытаемся их упаковать в массив n .



0	1	2	3
		7	
	8		3
0,4,5			

Наполняем массив вершины 8. Добавляем туда **0,4,5**.

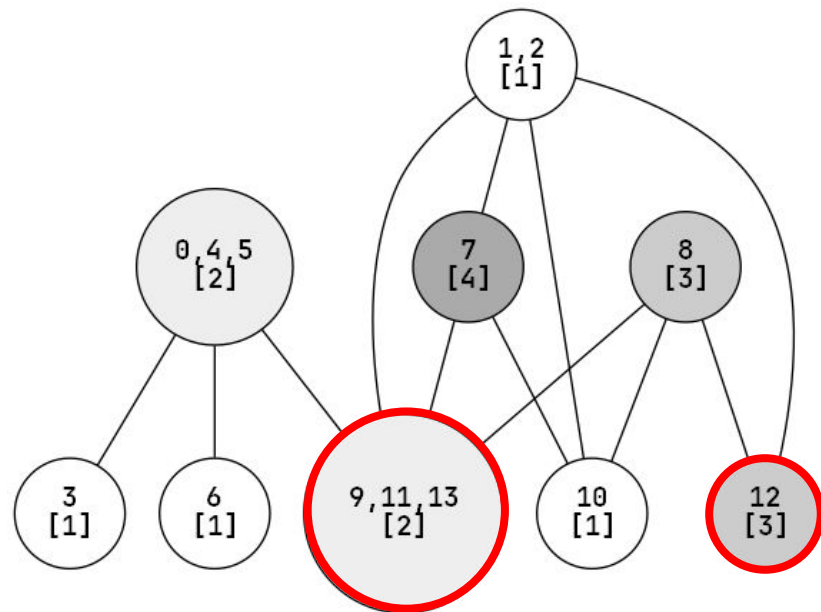
- Для каждой вершины n , начиная с наибольшего веса, ищем совместимые с ней вершины, тоже от наибольшей, и пытаемся их упаковать в массив n .



0	1	2	3
		7	
	8		3
0,4,5		6	

Наполняем массив вершины **8**. Добавляем туда **6**.

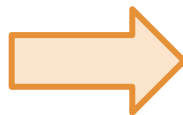
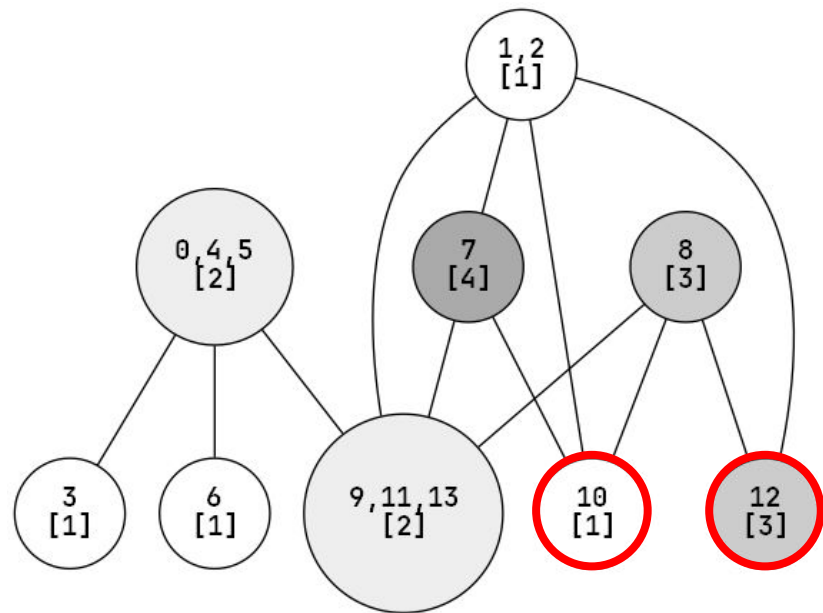
3. Для каждой вершины n , начиная с наибольшего веса, ищем совместимые с ней вершины, тоже от наибольшей, и пытаемся их упаковать в массив n .



0	1	2	3	4	5	6
		7			12	
	8		3	9, 11, 13		
0, 4, 5		6				

Наполняем массив вершины 12. Добавляем туда **9, 11, 13**.

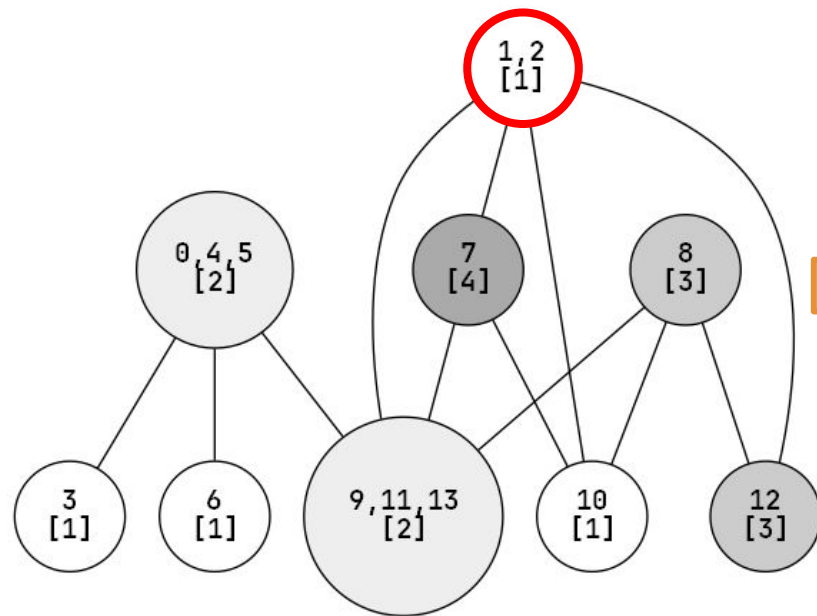
3. Для каждой вершины n , начиная с наибольшего веса, ищем совместимые с ней вершины, тоже от наибольшей, и пытаемся их упаковать в массив n .



0	1	2	3	4	5	6
	7			12		
	8		3	9,11,13		10
0,4,5		6				

Наполняем массив вершины **12**. Добавляем туда **10**.

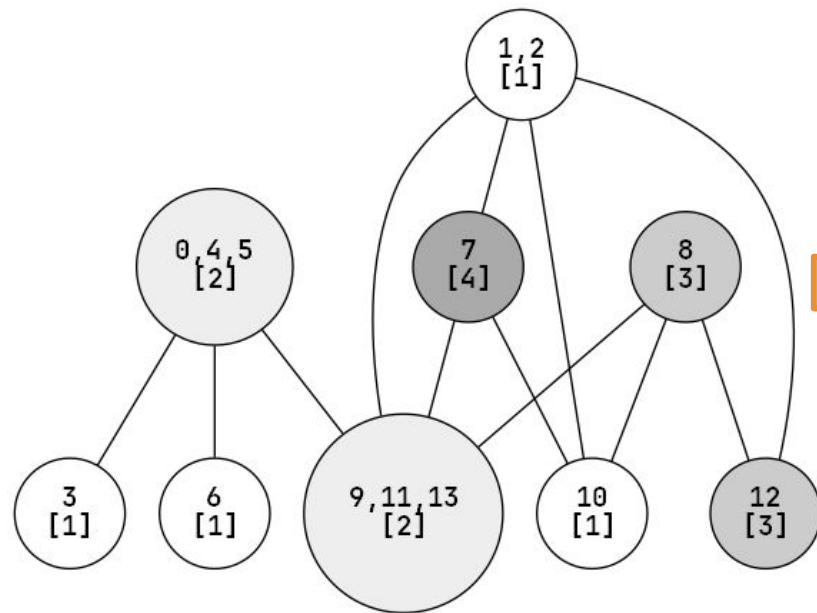
3. Для каждой вершины n , начиная с наибольшего веса, ищем совместимые с ней вершины, тоже от наибольшей, и пытаемся их упаковать в массив n .



0	1	2	3	4	5	6	7
		7			12		1,2
	8		3	9,11,13		10	
0,4,5		6					

Добавляем массив 1,2.

Еще можно заполнять “окна”
на стыках массивов, сдвигая
вложенные массивы.



0	1	2	3	4	5	6	7
		7			12		1,2
	8		3	9,11,13		10	
0,4,5		6					

8 ячеек вместо 27!

```
// TODO: In the future we plan to improve stack coloring in the following ways:  
// 1. Allow merging multiple small slots into a single larger slot at different  
// offsets.
```

The screenshot displays the Compiler Explorer web application. The left pane shows the C++ source code for a function `f` that calls `g` with two buffers. The right pane shows the corresponding x86-64 assembly generated by clang 16.0.0. A red question mark is placed over the first assembly instruction, `sub rsp, 1544`.

Compiler Explorer Interface:

- Top Bar:** COMPILER EXPLORER logo, navigation links (Add..., More, Templates), and utility links (Share, Policies, Other).
- Left Pane (C++ source #2):**

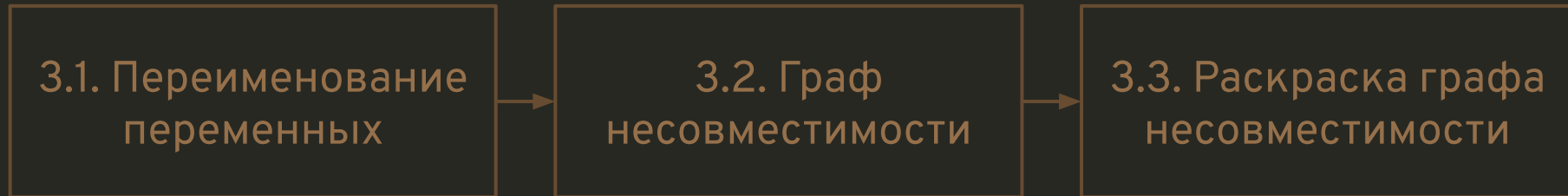
```
1 void g(char *, char *);  
2 void f(void) {  
3     {  
4         char buf1[1024];  
5         g(buf1, buf1 + 512);  
6     }  
7     {  
8         char buf1[512];  
9         char buf2[512];  
10        g(buf1, buf2);  
11    }  
12 }
```
- Right Pane (x86-64 clang 16.0.0 (Editor #2)):**
 - Compiler:** x86-64 clang 16.0.0
 - Options:** -O3 -g0
 - Assembly Output:**

```
2  sub    rsp, 1544  
3  lea    rsi, [rsp + 512]  
4  mov    rdi, rsp  
5  call   g(char*, char*)@PLT  
6  mov    rdi, rsp  
7  lea    rsi, [rsp + 1024]  
8  call   g(char*, char*)@PLT  
9  add    rsp, 1544  
10 ret
```
 - Output (0/0):** x86-64 clang 16.0.0 - 1714ms (711B) ~17 lines filtered
 - Compiler License:**

1. Введение

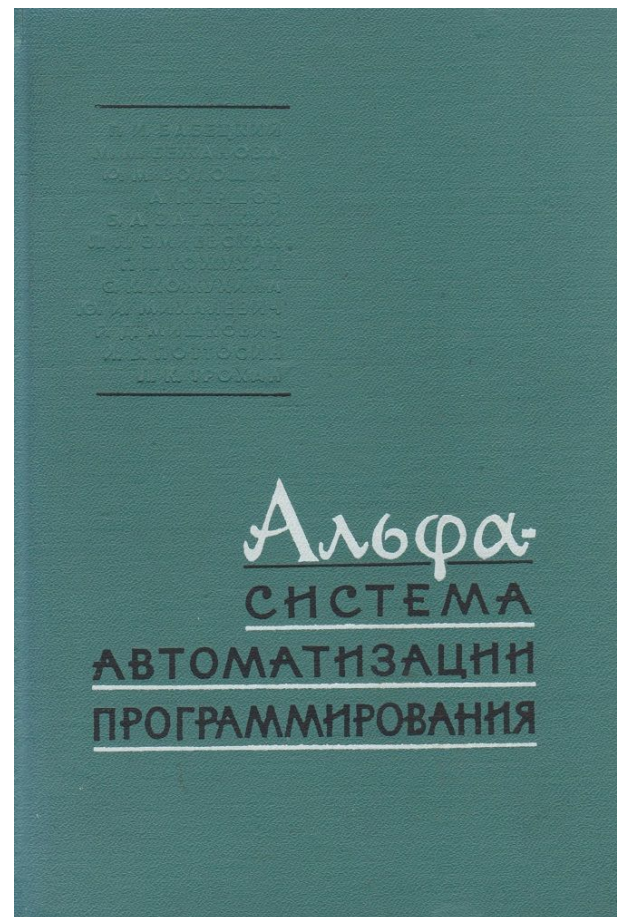
2. Пример экономии памяти

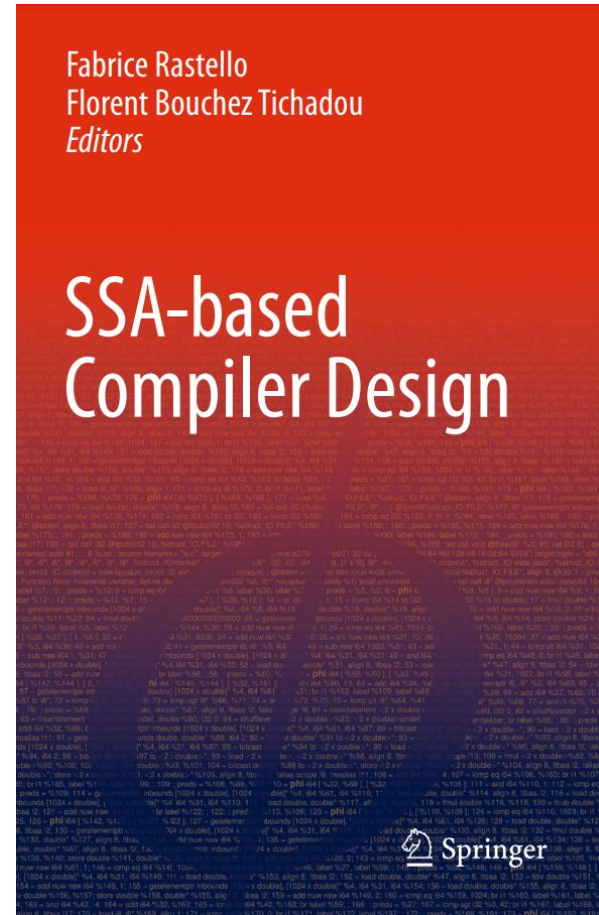
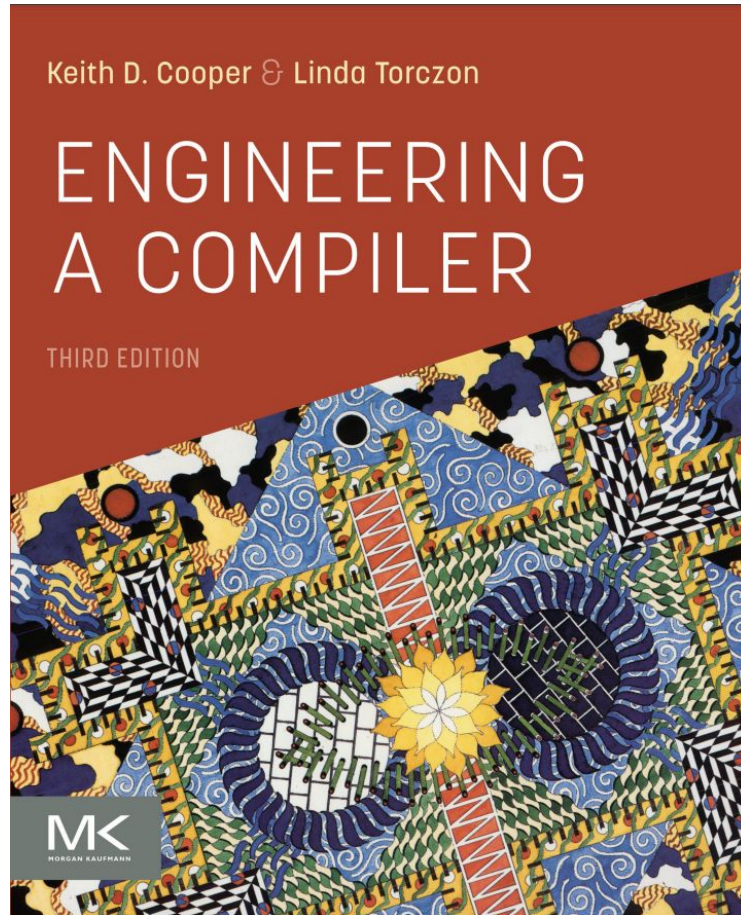
3. Фазы экономии памяти



4. Экономия памяти для массивов

5. Заключение





- Архиву академика А.П. Ершова (<http://ershov.iis.nsk.su/>) за бесценные отсканированные материалы.
- Букинистам (<https://www.alib.ru/>) за редкие книги по советской информатике.
- Павлу Павлову за то, что в свое время привлек мое внимание к алгоритму Ершова-Кожухина.