САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

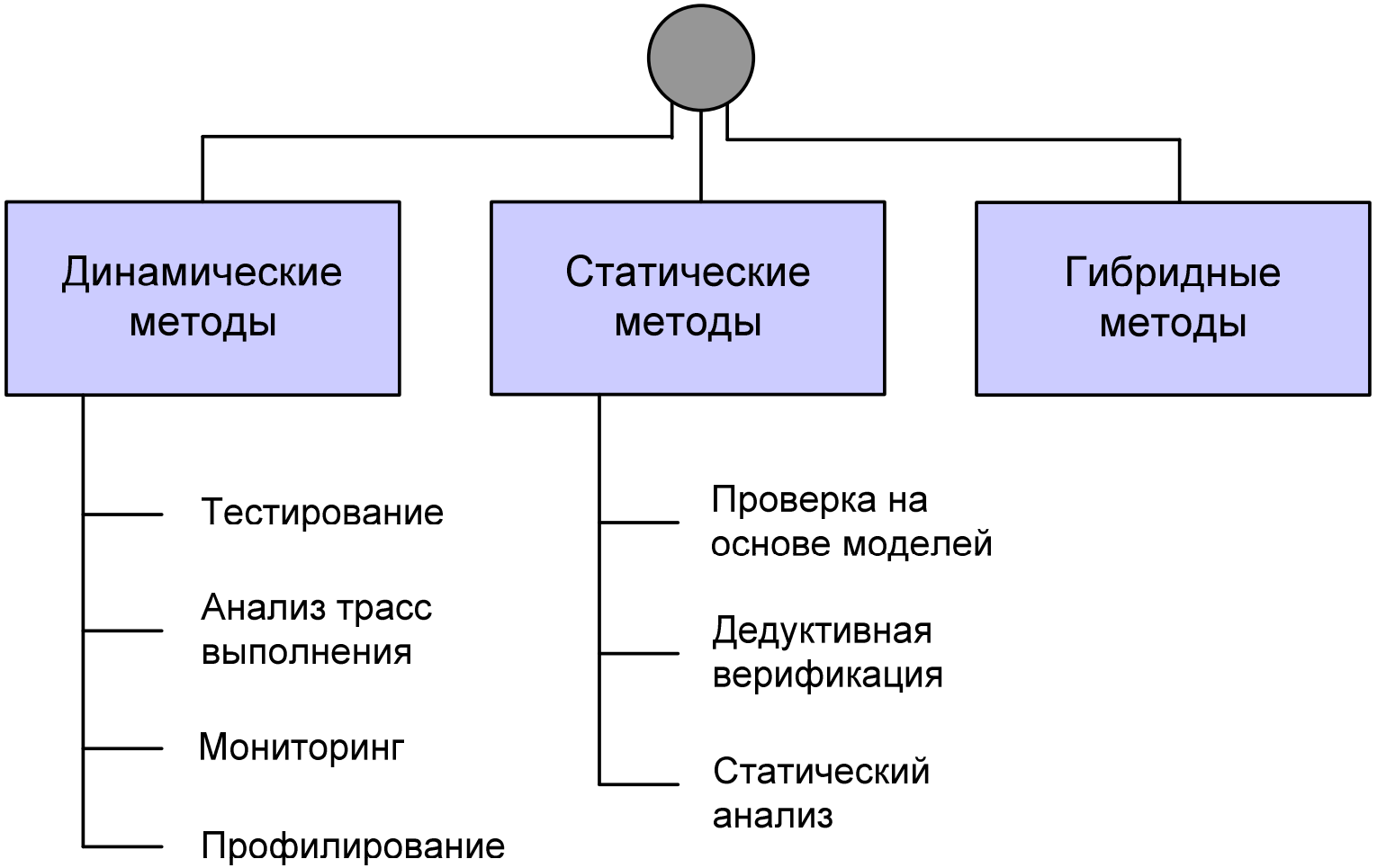
Методы анализа и обеспечения качества ПО

Михаил Моисеев

Введение в статический анализ

Санкт-Петербург
2011

Методы анализа ПО



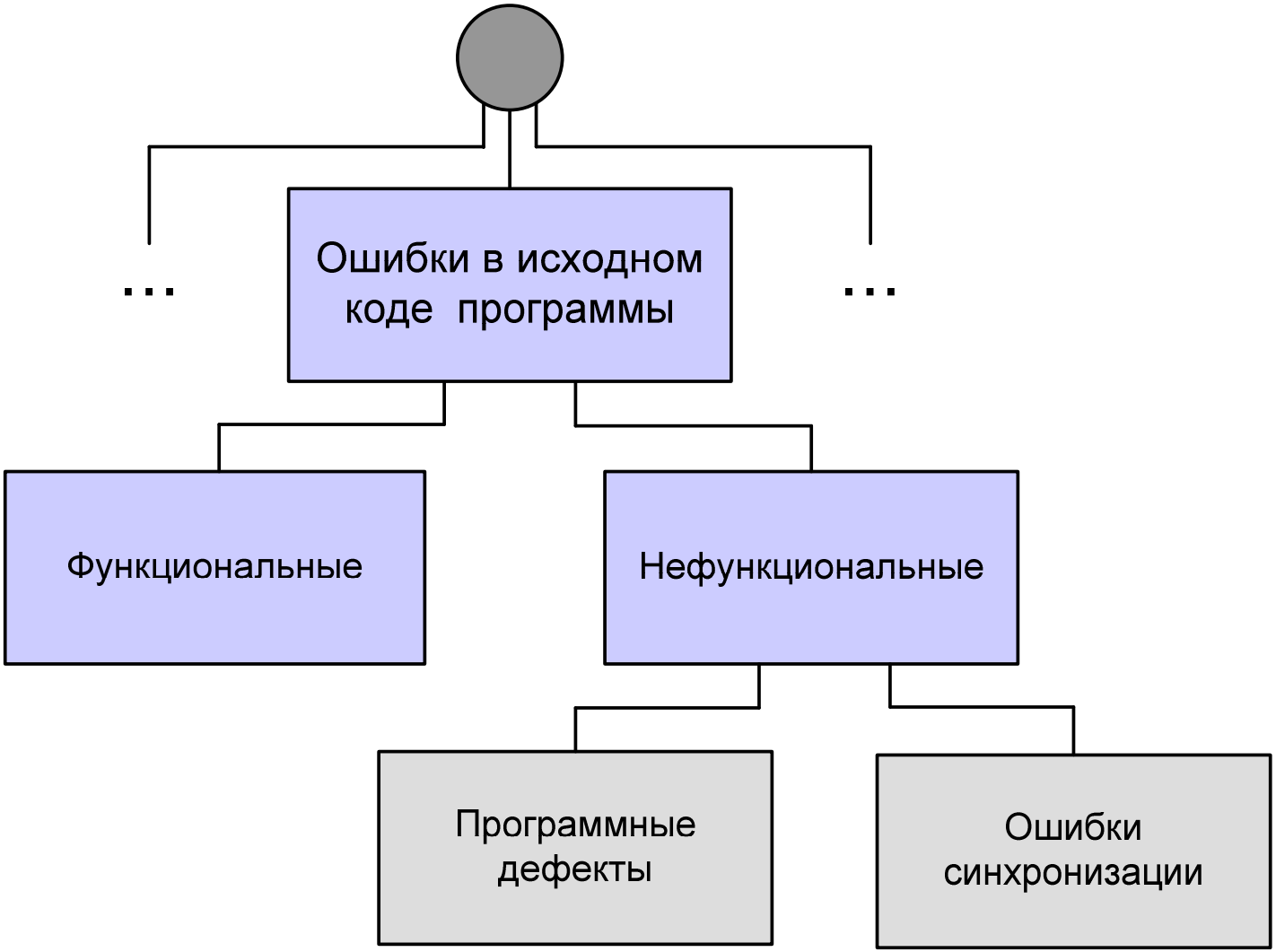
Статический анализ

- **Статический анализ** – группа методов, использующих исходный код программы для извлечения необходимой информации

Задачи статического анализа

- Оптимизация программ
  - вычисление константных выражений
  - обнаружение мертвого кода
  - распараллеливание программы
- Преобразование программ
  - перевод на другие платформы, языки и библиотеки
- Обфускация / деобфускация
- **Обнаружение ошибок**

Программные ошибки



Показатели эффективности

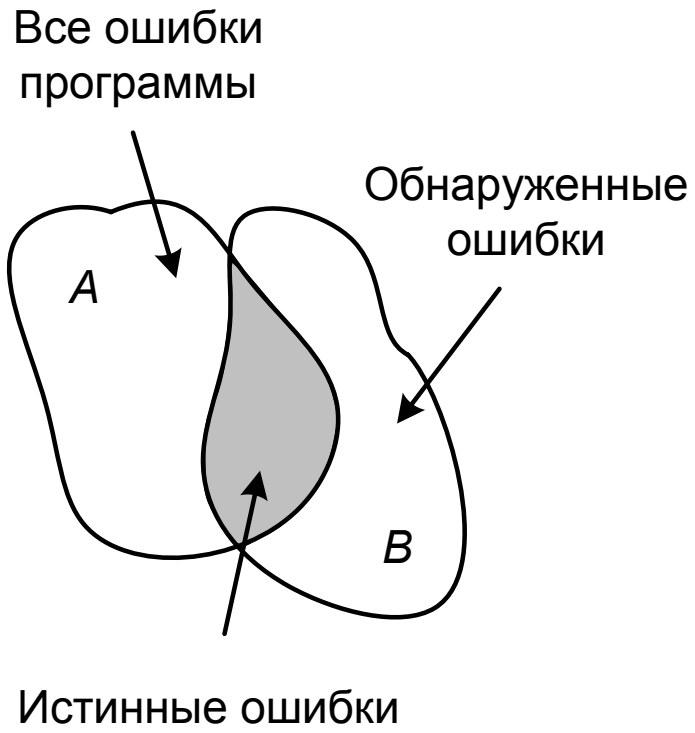
- **Полнота** – доля истинных ошибок среди всех ошибок в программе

$$\frac{|A \cap B|}{|A|}$$

- **Точность** – доля истинных ошибок среди всех обнаруженных ошибок

$$\frac{|A \cap B|}{|B|}$$

- **Ресурсоемкость**
- **Степень автоматизации**



Обнаружение нефункциональных ошибок

- Проверяемые правила определяются стандартами языка и стандартных библиотек функций
- Свойства статического анализа
  - обнаружение основных видов нефункциональных ошибок
  - **процесс обнаружения можно полностью автоматизировать**
  - обладает приемлемой ресурсоемкостью
  - обладает достаточно высокой полнотой/точностью
- Статический анализ одна из развитых техник – проверенные на практике правила реализуются в средах разработки и компиляторах
- Обнаружение ошибок с помощью компиляторов

Обнаружение функциональных ошибок

- Проверяется соответствие исходного кода программы спецификациям на некотором формальном языке
- Примерами таких спецификаций являются контракты для функций программы
  - предусловия – требуются для выполнения функции
  - постусловия – должны обеспечиваться после ее выполнения
  - инварианты – должны сохраняться в процессе выполнения функции
- Свойства статического анализа
  - процесс обнаружения кроме создания спецификаций можно полностью автоматизировать
  - обладает приемлемой ресурсоемкостью
  - полнота зависит от количества спецификаций и полноты самих алгоритмов анализа, точность определяется алгоритмами анализа

Виды статического анализа

- Синтаксический анализ (поиск по шаблонам)
- Анализ потока данных
  - анализ состояний объектов программы
- Анализ потока управления
- Анализ параллельного выполнения программы

Анализ программ на основе шаблонов

- Выполняется синтаксический анализ – рассматриваются одна или несколько последовательных конструкций, **без учета контекста программы**
- Поиск в исходном коде конкретных или параметризируемых шаблонов
- Параметризируемый шаблон представляет собой конструкции программы и переменные в качестве параметров

```
gets(x);
```

```
if (x = y) {
   ...;
}
```

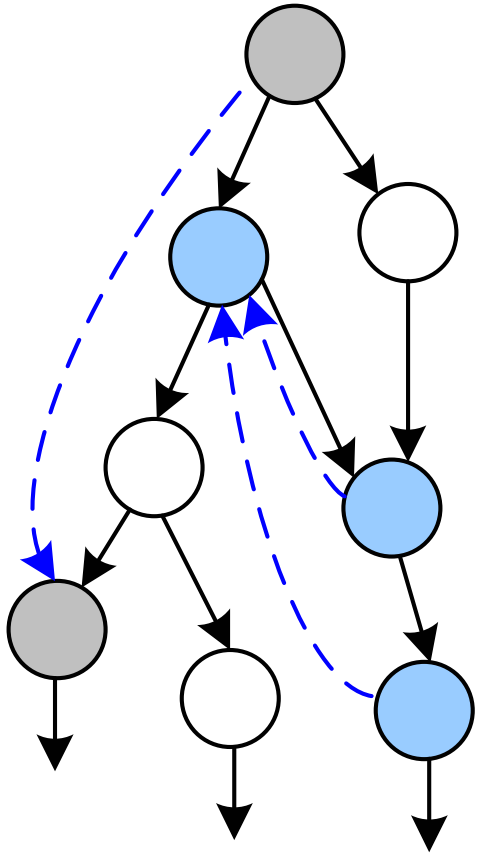
```
for (int x; x < C; x++) {
   ...;
}
```

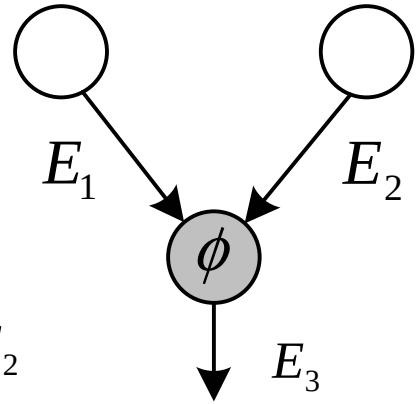
Анализ потока данных

- Анализируются изменения данных в конструкциях программы
- Виды анализа потока данных
  - Reaching Definitions
  - Available Expressions
  - Constant Propagation
  - Very Busy Expressions
  - Live Variables
  - Use-Definition & Definition-Use
  - Анализ состояний объектов программы

Классификация видов анализа

- Направление анализа
  - прямой
  - обратный
- Объединение результатов
  - may-анализ – полные результаты
  - must-анализ – точные результаты





may

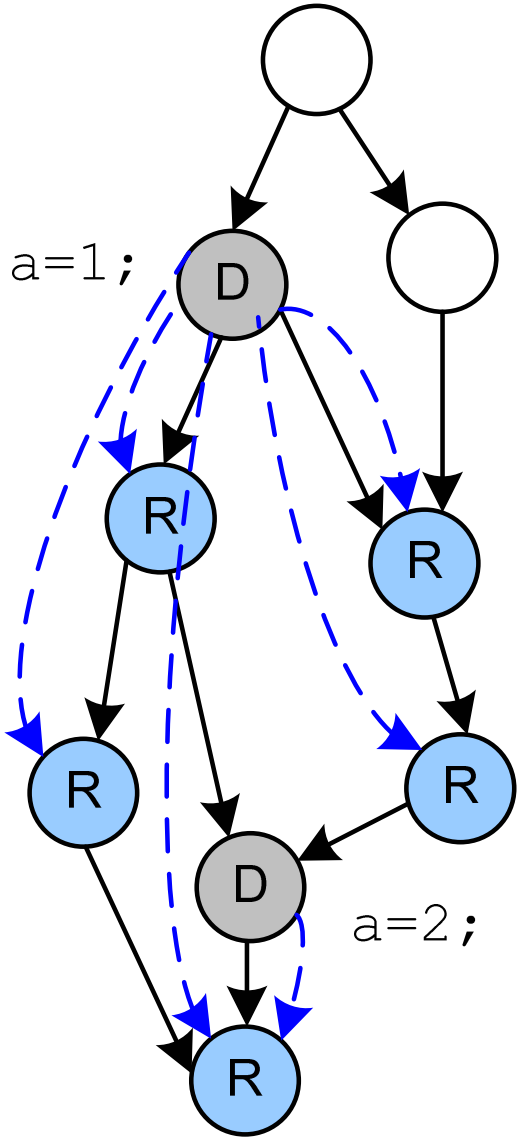
$E_3 = E_1 \cup E_2$

must

$E_3 = E_1 \cap E_2$

Reaching Definitions

- **Reaching Definitions** – определение конструкций, которых может достигать значение переменной присвоенное ей в некоторой конструкции



Reaching Definitions

- RD_j – множество достижимых определений для переменных, с указанием места определения
- $RD_j = (x_{j1},l_{k1}),\ldots(x_{jn},l_{kn})$, x – переменная, l – место определения
- Правила анализа

$$[x = a]^l : RD_l = RD_{l-1} \setminus (x, l_j) \cup (x, l)$$

$$[a]^l : RD_l = RD_{l-1}$$

$$[\phi]^l : RD_l = \bigcup_{\forall i \in \mathrm{Pred}(l)} RD_i$$

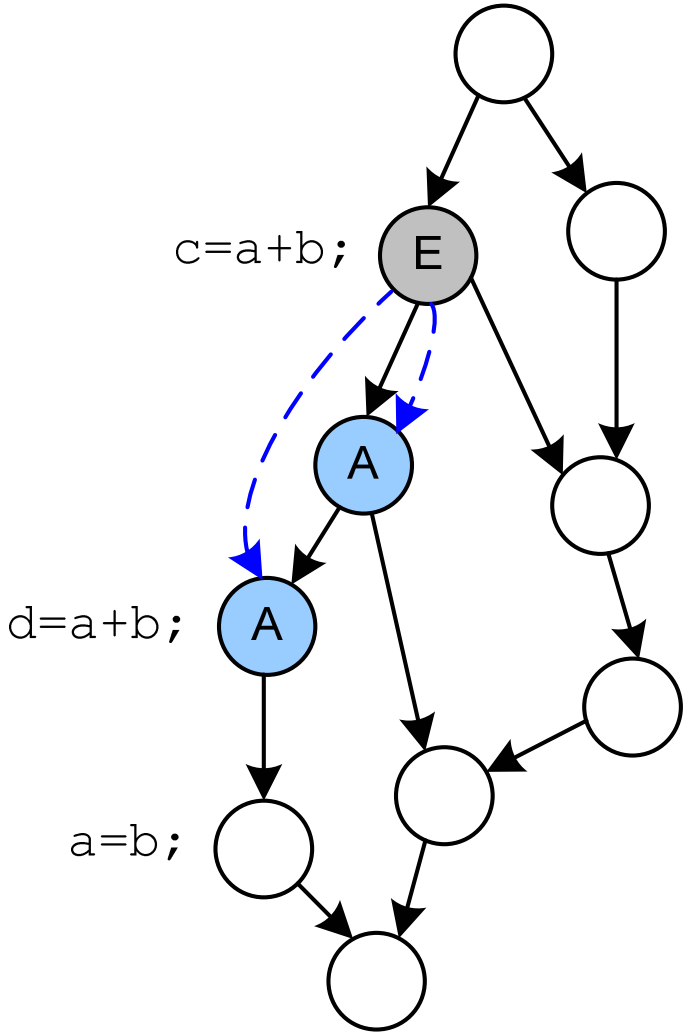
Reaching Definitions

```
0: int a,b;
1: a = 10;
2: b = 20;
3: a = b + 1;
4: if (...) {
5:     a = 3;
6: }
```

```
RD0 = (a,?)(b,?)
RD1 = (a,1)(b,?)
RD2 = (a,1)(b,2)
RD3 = (a,3)(b,2)
RD4 = (a,3)(b,2)
RD5 = (a,5)(b,2)
RD6 = (a,3)(a,5)(b,2)
```

Available Expressions

- **Available Expressions** – анализ ранее предвычисленных выражений, которые не изменились на всех путях до некоторой конструкции программы



Available Expressions

- AE_j – множество предвычисленных выражений
- $AE_j = e_{j1},\ldots e_{jn}$, e – выражение
- Правила анализа

$$[x = a]^l : AE_l = AE_{l-1} \setminus (e : x \in e) \cup (a : x \notin a)$$

$$[a]^l : AE_l = AE_{l-1} \cup (a)$$

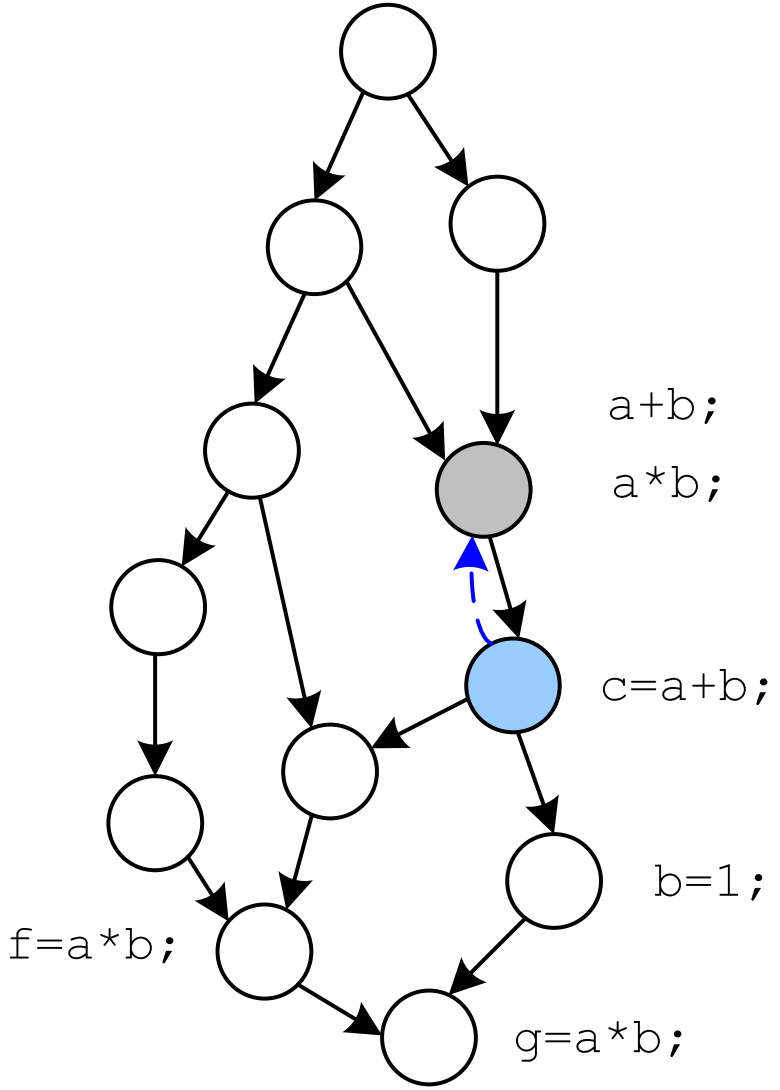
$$[\phi]^l : AE_l = \bigcap_{\forall i \in Pred(l)} AE_i$$

Available Expressions

```
0: int a,b,c;          AE0 = ()
1: a = b + 1;          AE1 = (b+1)
2: c = a * b;          AE2 = (b+1, a*b)
3: b = a + 1;          AE3 = (a+1)
4: if (c < 5){         AE4 = (a+1)
5:     c = a + b;      AE5 = (a+1)(a+b)
6: }                   AE6 = (a+1)
```

Very Busy Expressions

- **Very Busy Expressions** – анализ, определяющий выражения, значения которых гарантированно (на всех путях выполнения) используются до изменения переменных, входящих в это выражение
- Обратный анализ



Very Busy Expressions

- VB_j – множество выражений используемых далее на всех путях выполнения программы
- $VB_j = e_{j1},\ldots e_{jn}$
- Правила анализа

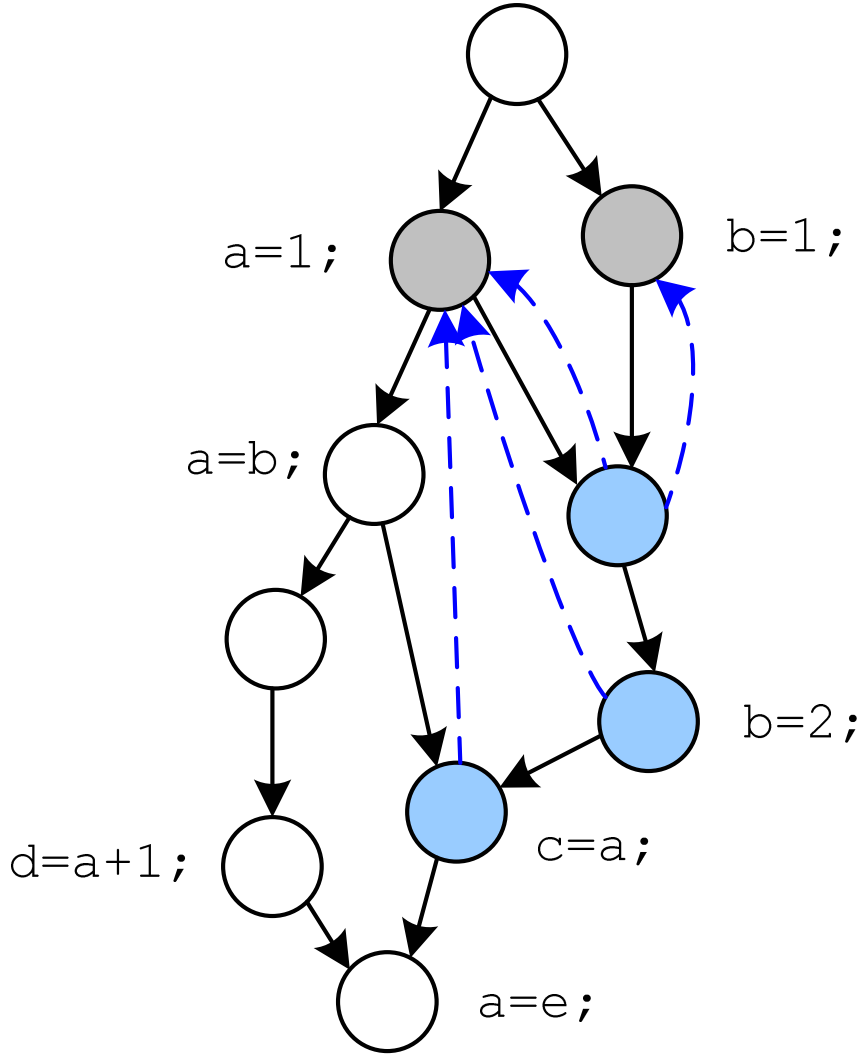
$$[x = a]^l : VB_l = VB_{l+1} \setminus (e : x \in e) \cup (a)$$

$$[a]^l : VB_l = VB_{l+1} \cup (a)$$

$$[if]^l : VB_l = \bigcap_{\forall i \in Succ(l)} VB_i$$

Live Variables

- **Live Variables** – анализ, определяющий переменные в некоторой конструкции, значения которых используется без переопределения хотя бы на одном пути выполнения программы
- Обратный анализ



Live Variables

- LV_j – множество переменных используемых далее хотя бы на одном пути выполнения программы
- $LV_j = v_{j1},\ldots v_{jn}$
- Правила анализа

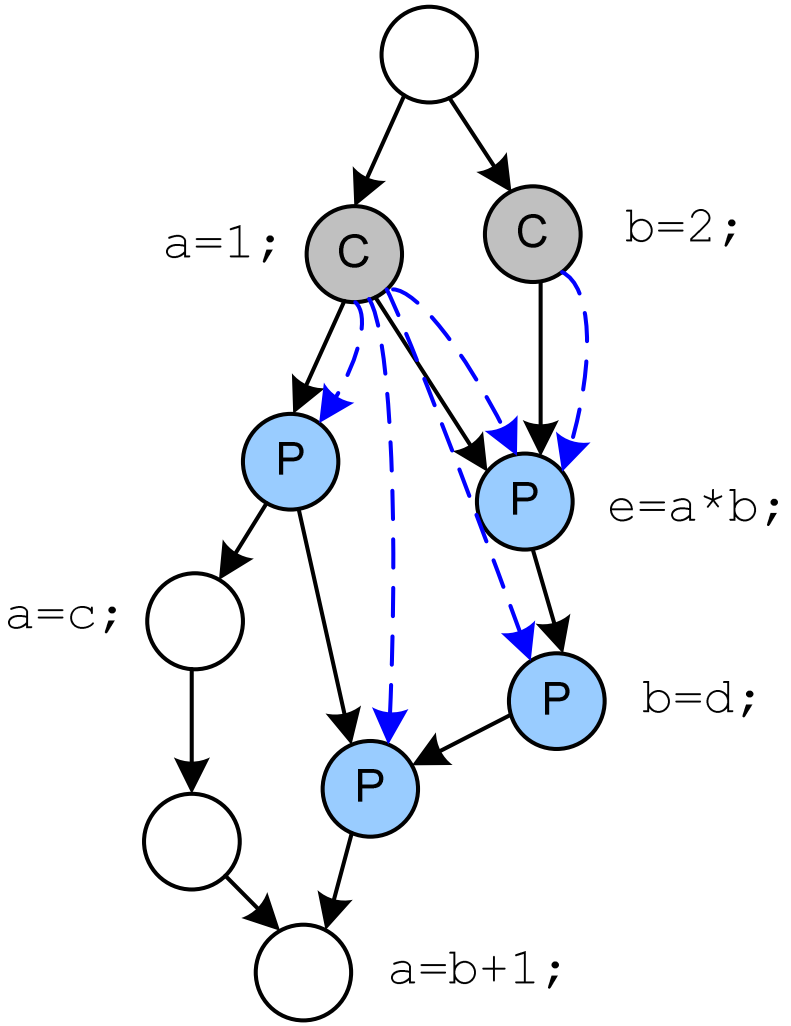
$$[x = a]^l : LV_l = LV_{l+1} \setminus (x) \cup (\forall y \in a)$$

$$[a]^l : LV_l = LV_{l+1}$$

$$[if\ ]^l : VB_l = \bigcup_{\forall i \in Succ(l)} VB_i$$

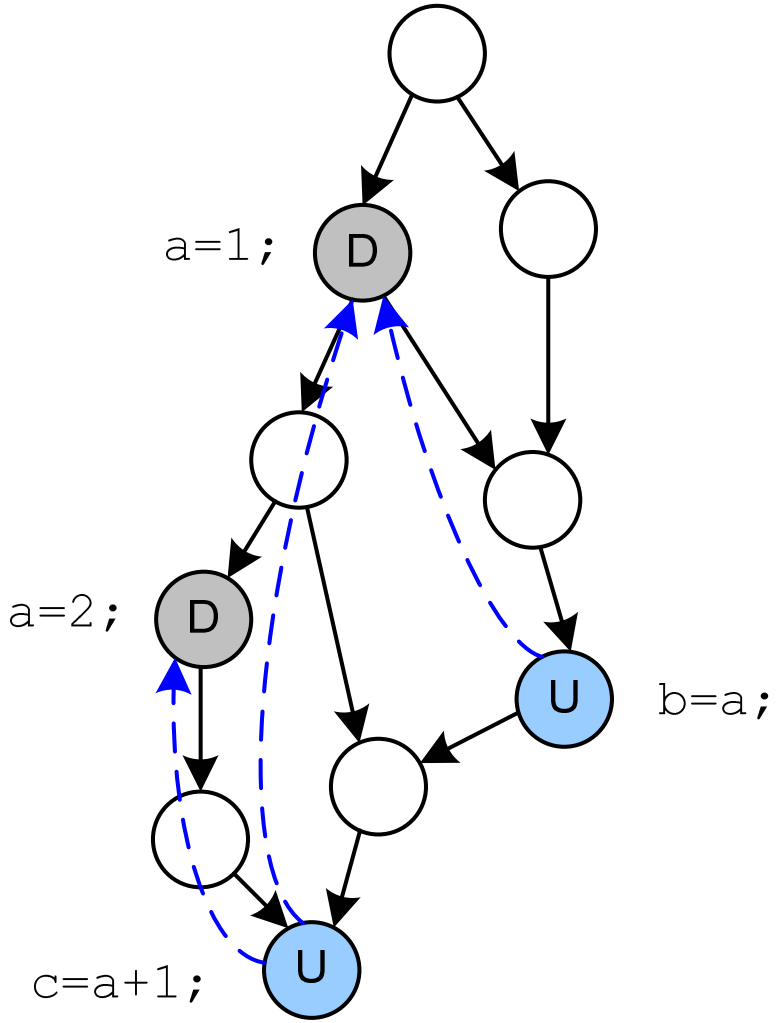
Constant Propagation

- **Constant Propagation** – анализ переменных, которые имеют константные значения в конструкциях программы



Use-Definition

- **Use-Definition** – анализ, определяющий места определения переменной, значение которой используется в некоторой конструкции программы

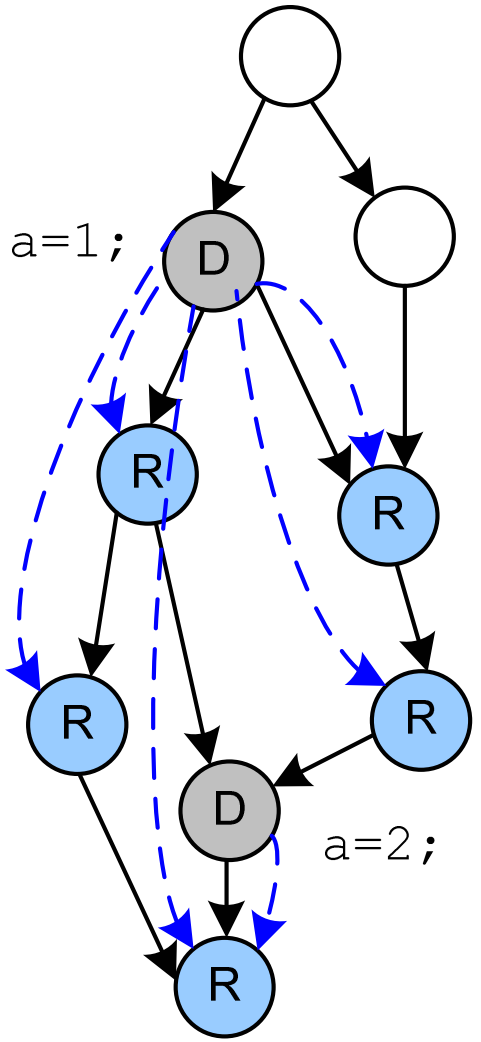


Алгоритмы анализа потока данных

- Алгоритмы извлечения необходимой информации
  - подход на основе системы уравнений или ограничений
  - абстрактная интерпретация
  - системы типов и эффектов

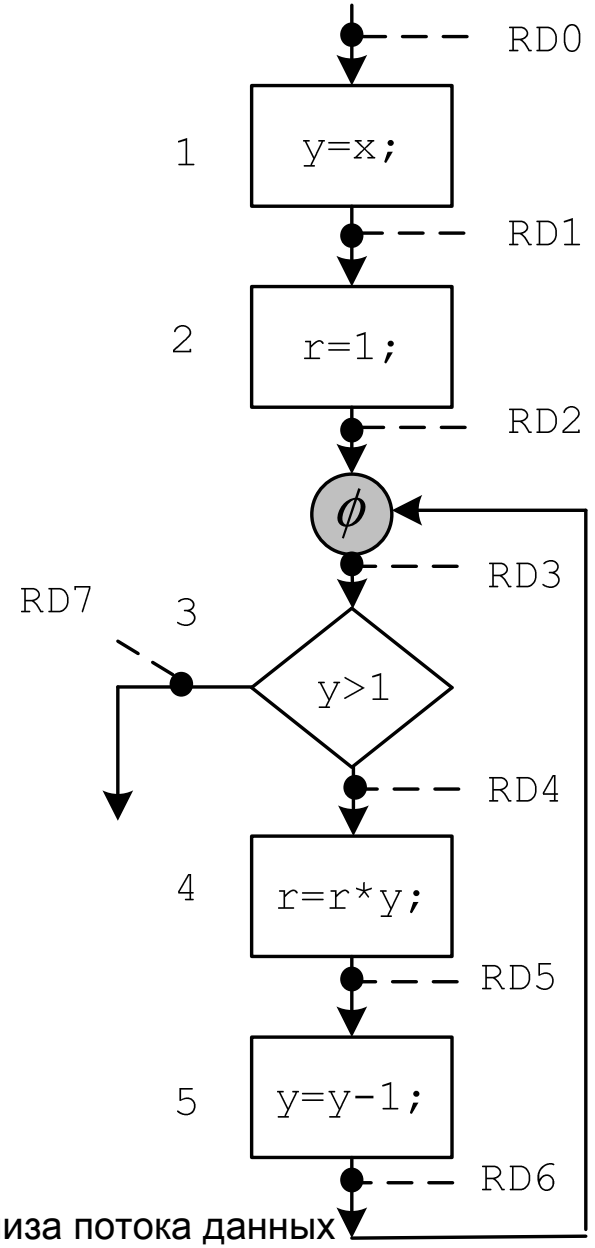
Подход на основе системы уравнений

- На примере RD анализа
- Для каждой конструкции программы строится уравнение
- Решение системы уравнений
- Проблема при наличии циклов

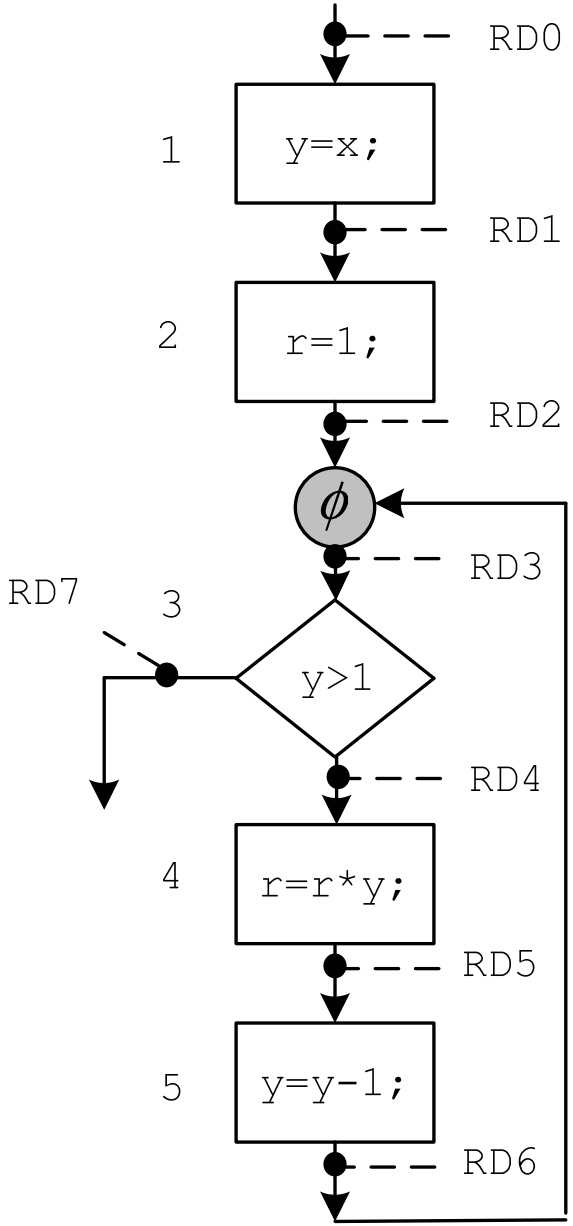


Подход на основе системы уравнений

```
int f(int x) {
   1: int y = x;
   2: int r = 1;
   3: while (y > 1) {
   4:    r = r * y;
   5:    y = y - 1;
   }
   return r;
}
```



Подход на основе системы уравнений



RD0 = (x,?)

RD1 = RD0\(y,i)∪(y,1)

RD2 = RD1\(r,i)∪(r,2)

RD3 = RD2 ∪ RD6

RD4 = RD3

RD5 = RD4\(r,i)∪(r,4)

RD6 = RD5\(y,i)∪(y,5)

RD7 = RD3

RD – множество пар вида (x_{j1}, l_{k1}),

где x_{j1} – переменная программы,

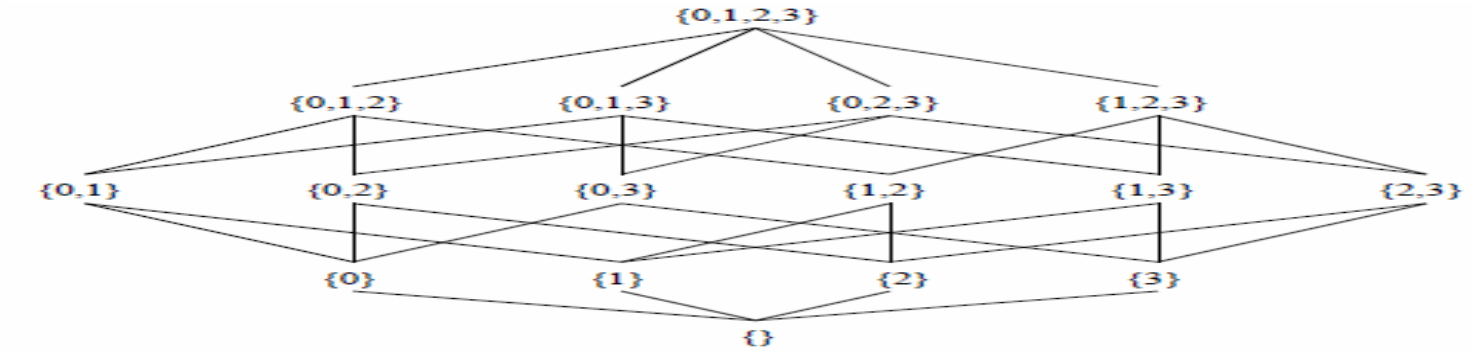
l_{k1} – место последнего определения x_{j1}

Решение системы уравнений

- При отсутствии циклов каждое уравнение достаточно решить один раз
- При наличии циклов используются специальные алгоритмы решения системы уравнений
  - получение решения с требуемыми свойствами (полного решения)
  - получение решения за приемлемое время
- Для описания этих алгоритмов удобно применять математический аппарат теории решеток

Решетка

- Решетка – частично упорядоченное множество, для каждого подмножества которого определена единственная точная верхняя грань и единственная точная нижняя грань



- Решетка, состоит из подмножеств множества (x_i, l_j), отношением порядка является отношение включения подмножеств

$$RD_1 \subseteq RD_2 \Leftrightarrow \forall (x_i, l_j) \in RD_1 \Rightarrow (x_i, l_j) \in RD_2$$

Представление СУ с помощью функции

- Система уравнений состоит из уравнений вида

$$RD_l = f_l(RD_1, \ldots, RD_n)$$

- СУ может быть представлена с помощью функции

$$F(\overline{RD}) = (f_1(\overline{RD}), \ldots, f_n(\overline{RD}))$$

- Функция *F(X)* является монотонной, если справедливо

$$\forall RD_1, RD_2 : RD_1 \subseteq RD_2 \Rightarrow F(RD_1) \subseteq F(RD_2)$$

Наименьшая неподвижная точка (LFP)

- Любая функция *F(X)*, монотонная над элементами конечной решетки, имеет единственную **наименьшую неподвижную точку** (Least Fixed Point), для которой справедливо

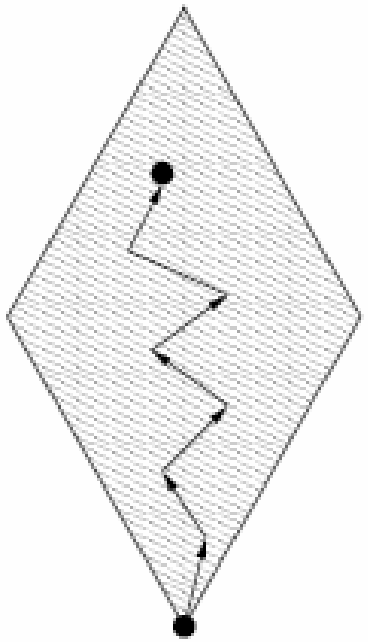
$$F(LFP) = LFP$$

- LFP является минимально возможным решением СУ
- Наименьшая неподвижная точка для заданной монотонной функции может быть получена путем ее многократного применения к нижней грани решетки

$$F^1(\bot) = F(\bot),\ F^2(\bot) = F\left(F^1(\bot)\right), \ldots, F^{n+1}(\bot) = F\left(F^n(\bot)\right)$$

Алгоритмы вычисления LFP

- Алгоритм хаотических итераций (**Chaotic Iteration algorithm**)
- Алгоритм на основе списков зависимых конструкций (**Worklist algorithm**)
- Другие алгоритмы (**Strong Components algorithm**)



Алгоритм хаотических итераций

- Алгоритм хаотических итераций
  1. выполняется решение всех уравнений
  2. если результат решения хотя бы одного уравнения изменился по сравнению с предыдущим решением (итерацией), то п. 1.

$$RD_1 = \varnothing, \ldots, RD_n = \varnothing$$
$$\exists j : RD_j \neq f_j(RD_1, \ldots, RD_n)$$

- Не учитывает зависимостей между уравнениями программы, в результате решается существенно больше уравнений чем требуется

Worklist - алгоритм

- Алгоритм на основе списков зависимых конструкций
  1. выполняется решение всех уравнений
  2. для каждого решенного уравнения определяются уравнения, на которые оно влияет, эти уравнения добавляются в список решаемых
  3. если список решаемых уравнений не пуст, то п. 2

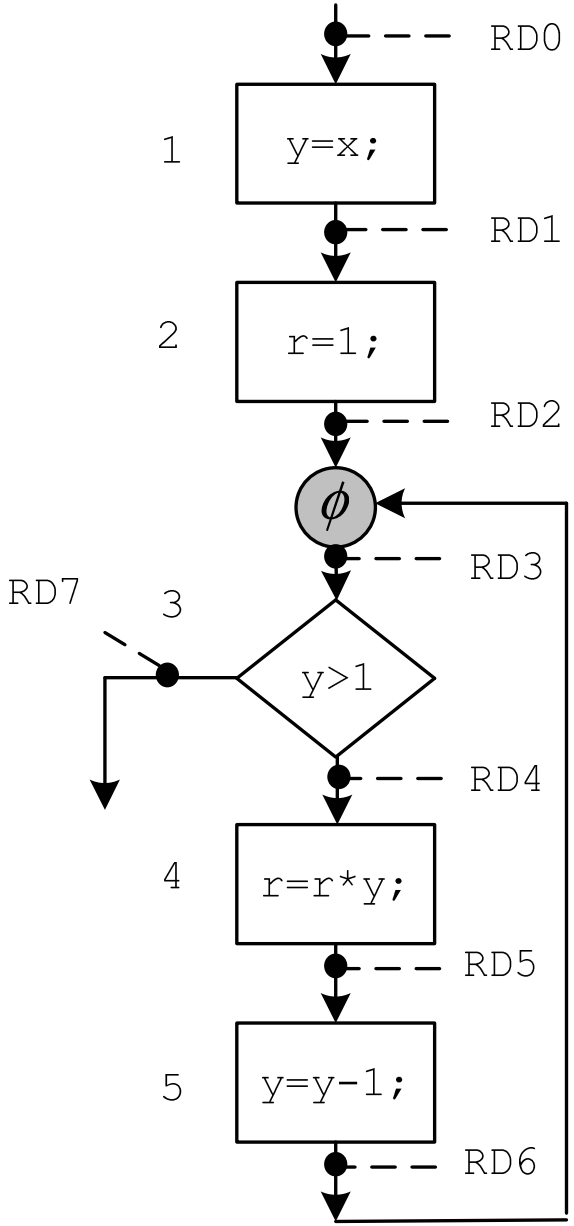
$$RD_1 = \varnothing, \ldots, RD_n = \varnothing, W_k = \varnothing$$
$$W_{k-1} \neq \varnothing : \forall RD_j \in W_{k-1} \Rightarrow$$
$$RD_j \neq f_j(RD_1, \ldots, RD_n), W_k = W_k \cup E_j$$

- На практике бывает сложно определить взаимные влияния уравнений, не обеспечивается оптимальный порядок решения уравнений для вложенных циклов

Strong Components - алгоритмы

- Алгоритмы на основе определения сильносвязанных частей в графе программы (сильносвязанных множеств уравнений)
  1. определяются сильносвязанные множества уравнений
  2. выполняется решение уравнений каждого множества, полученные результаты распространяются на между множествами
  3. процесс повторяется необходимое числа раз
- Многоуровневая иерархия

Решение системы уравнений RD



RD0 = (x,?)

RD1 = RD0\(y,i)∪(y,1)

RD2 = RD1\(r,i)∪(r,2)

RD3 = RD2 ∪ RD6

RD4 = RD3

RD5 = RD4\(r,i)∪(r,4)

RD6 = RD5\(y,i)∪(y,5)

RD7 = RD3

RD1 = (y,1)(r,?)	RD3 = (y,1)(y,5)
RD2 = (y,1)(r,2)	(r,2)(r,4)
RD3 = (y,1)(r,2)	RD5 = (y,1)(y,5)(r,4)
RD5 = (y,1)(r,4)	RD6 = (y,5)(r,4)
RD6 = (y,5)(r,4)	RD7 = (y,1)(y,5)
RD7 = (y,1)(r,2)	(r,2)(r,4)

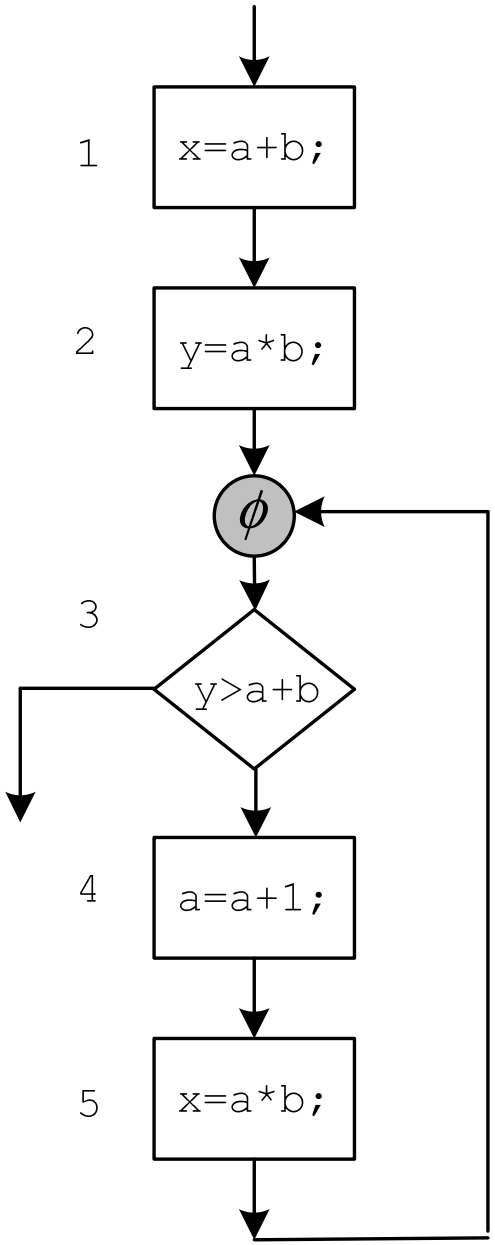
Абстрактная интерпретация

- Общая теория, которая задает способ аппроксимации семантики динамических дискретных систем
- В основе подхода лежит аппроксимация семантики поведения программы для получения вычислимой и при этом достоверной семантики
- Полученная аппроксимация должна обеспечивать проверку частичных свойств программы
- АИ может рассматриваться как частичное выполнение программы с получением требуемой семантической информации

Распространение информации по путям выполнения программы

- Решение СУ путем распространения информации по всем **реализуемым** путям выполнения программы
- Может выполняться как **may**, так и **must** анализ
- Анализ начинается с первой конструкции программы и завершается в после прохождения всех путей
- Для снижения ресурсоемкости используются различные аппроксимации (упрощения)

Решение системы уравнений AE



AE0 = ()

AE1 = AE0\(e:x)∪(a+b)

AE2 = AE1\(e:y)∪(a*b)

AE3 = AE2 ∩ AE6

AE4 = AE3 ∪ (a+b)

AE5 = AE4\(e:a)

AE6 = AE5\(e:x)∪(a+b)

AE7 = AE3

AE1 = (a+b)	AE3 = (a*b)
AE2 = (a+b,a*b)	AE4 = (a*b,a+b)
AE3 = ()	AE5 = ()
AE4 = (a+b)	AE6 = (a*b)
AE5 = ()	AE7 = (a*b)
AE6 = (a*b)	

Анализ состояний объектов программы

- Определяются возможные или гарантированные значения всех видимых объектов в точках программы
- Определяются значения
  - переменных численных типов (интервальных переменных)
  - переменных-указателей на объекты
  - переменных-указателей на функции
  - дескрипторов ресурсов
- Для определения состояний объектов используются те же методы, что для анализа потока данных
- Состояния объектов могут использоваться для обнаружения ошибок

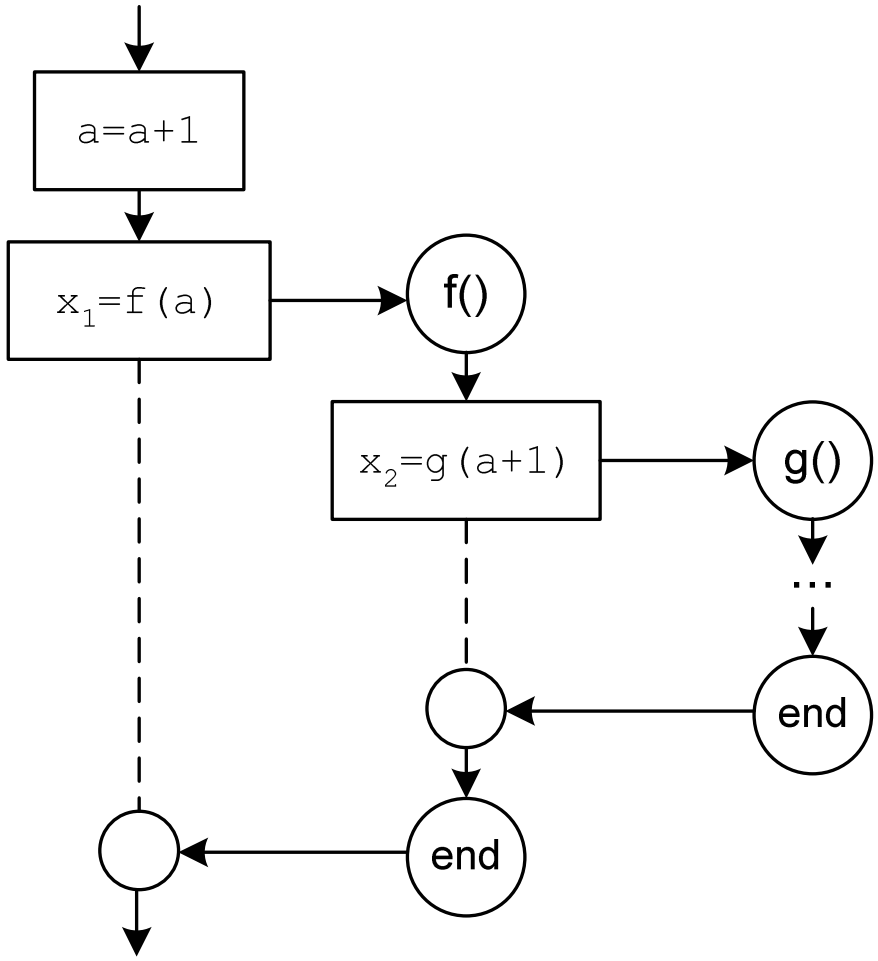
Анализ потока управления

- Связь между анализом потока управления и анализом состояний объектов программы
- Анализ потока управления обеспечивает
  - определение последовательности анализируемых конструкций
  - передачу параметров в вызываемые функции и возвращаемого значения в точку вызова
- Анализ потока данных обеспечивает
  - вызов функций через указатели
  - определение недостижимых переходов

Анализ потока управления

- Раскрываются вызываемые функций, строится полный CFG программы

```
main() {
   a = a + 1;
   return f(a);
}
int f(int a) {
   return g(a+1);
}
int g(int a) {
   ...
}
```



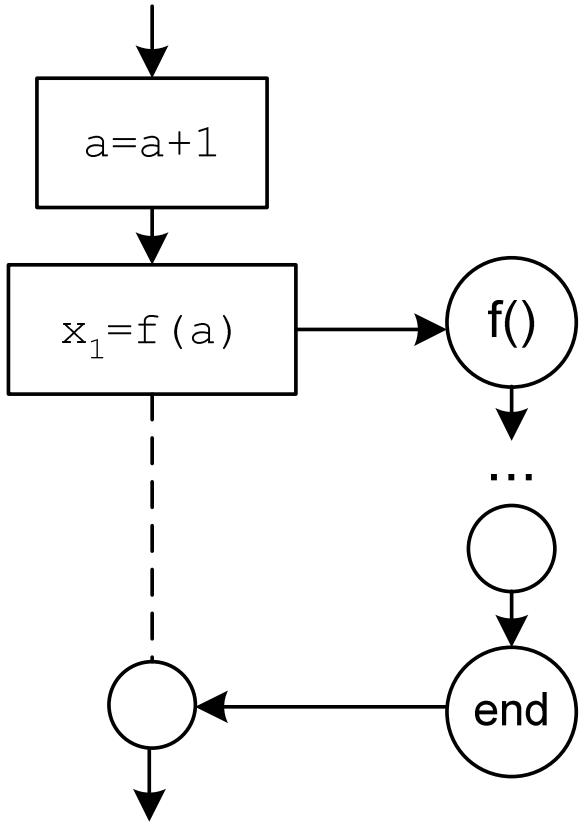
Вызов функций через указатели

```
main() {
   ...
   a = a + 1;
   return *pf(a);
}
int f(int a) {
   return g(a+1);
}
```

Анализ потока данных

```
pf -> f()
```

Анализ потока управления



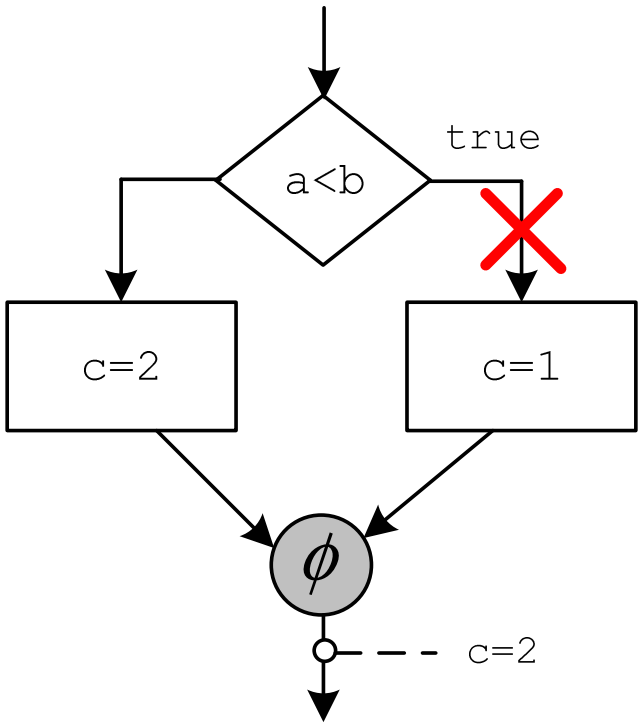
Определение недостижимых переходов

```
main() {
   ...
   a = b + 1;
   if (a < b) {
      c = 1;
   } else {
      c = 2;
   }
}
```

Анализ потока данных

всегда верно, что a > b

Анализ потока управления



Эффективность статического анализа

- **Эффективность СА** – комплексное свойство, отражающее качество получаемых результатов, степень автоматизации процесса анализа и сложность его организации, ресурсоемкость, применимость для программ различного размера и класса
- Показатели эффективности:
  - полнота результатов
  - точность результатов
  - ресурсоемкость
  - возможность автоматизации

Эффективность статического анализа

- Эффективность СА зависит от
  - вида извлекаемой информации
  - свойств языка программирования
  - особенностей алгоритма программы и стиля кодирования
  - степени влияния окружения и внешних воздействий
  - используемых алгоритмов анализа

Извлекаемая информация

- Определяется задачей статического анализа
  - типами обнаруживаемых ошибок
- Для обнаружения нефункциональных ошибок в программах на языке C необходимо определять
  - значения интервальных переменных
  - объекты, на которые указывают указатели
  - функции, на которые указывают указатели
  - состояния ресурсов
- Извлекаемая информация для обнаружения функциональных ошибок определяется проверяемыми спецификациями

Свойства языка программирования

- Наиболее сложные конструкции языка C
  - арифметика указателей
  - сложные типы данных
  - указатели на функции
  - приведение типов
  - рекурсивные функции
- В языке C++ дополнительно
  - анализ исключений
  - полиморфизм и виртуальные функции
  - шаблоны

Особенности алгоритма и стиля кодирования

- Сложность алгоритма и способ его реализации
- Обычно используется подмножество языка программирования
- Архитектура программы
- Стиль кодирования

Входные данные и внешние воздействия

- Виды внешних воздействий
  - входные параметры программы
  - данные, полученные из потока ввода и прочитанные из файлов
  - команды пользователя
  - внешнее окружение программы
- Информация о внешних воздействиях отсутствует в исходном коде программы
- Чем больше влияние входных данных информации, тем ниже эффективность статического анализа

Свойства алгоритмов СА

- Любой алгоритм статического анализа использует те или иные аппроксимации
  - для снижения ресурсоемкости
  - при этом снижается полнота/точность получаемых результатов
- Примеры используемых аппроксимаций
  - совместный анализ путей выполнения программы
  - ограничение на глубину стека вызовов
  - ограничение на число итераций при анализе циклов
  - ограничение на размер анализируемых объектов