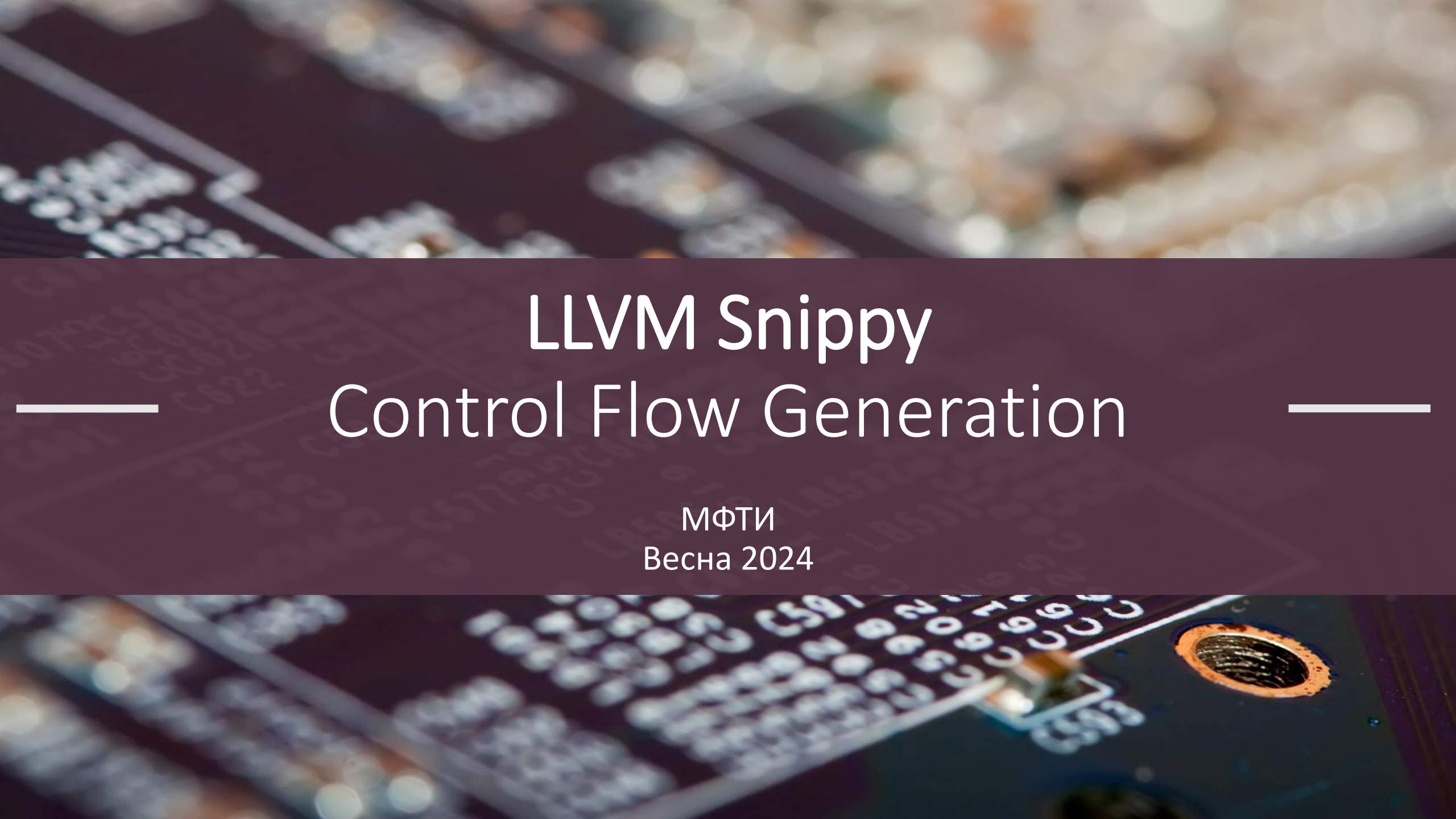


LLVM Snippy
Control Flow Generation

МФТИ
Весна 2024

Теорминимум: поток управления

Поток управления (**Control Flow** или **CF**) – порядок выполнения инструкций в *императивной* программе

Императивное программирование – парадигма программирования, при которой программа состоит из команд, выполняемых компьютером

В момент выполнения инструкции она *управляет* процессором

Передача управления – переход к выполнению следующей инструкции

Последовательность *передач управления* в процессе выполнения программы формирует ее *поток управления*

Теорминимум: виды передачи управления

Какие виды передачи управления вы знаете?

Как вы оцените их стоимость?

Теорминимум: виды передачи управления

- Прямой переход
  - Безусловный jump/goto
  - Условный jump/branch
- Косвенный переход
  - Вызов функции по указателю
  - Вычисляемый jump/goto
  - Прерывания

Теорминимум: граф потока управления

Базовый блок – прямолинейный участок кода, не содержащий в себе ни операций передачи управления, ни операций, на которые управление передается из других частей программы

Граф потока управления (**Control Flow Graph** или **CFG**) – представление потока управления в виде графа

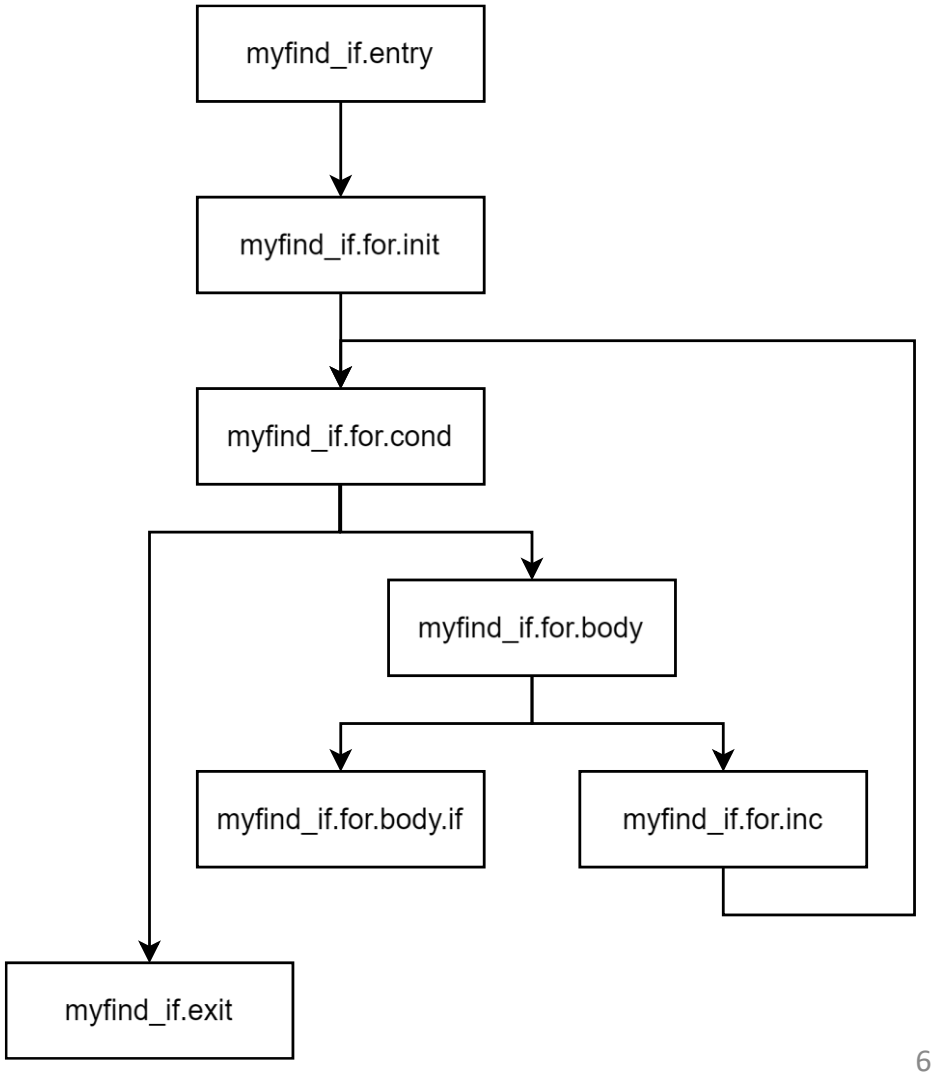
Вершины CFG соответствуют базовым блокам

Дуги CFG соответствуют инструкциям потока управления

Теорминимум: пример CFG

```
template<typename T>
T *myfind_if(T *beg, T *end, T x)
{
  for (T *it = beg; it != end; ++it)
  {
        if (*it == x)
          return it;
  }

  return end;
}
```



Теорминимум: структурированный поток управления

Структурированный поток управления состоит только из трех видов управляющих структур:

- Последовательность
- Ветвление
- Цикл

Теорема Бёма — Якопини

> *Любая вычисляемая программа может быть вычислена со структурированным потоком управления.*

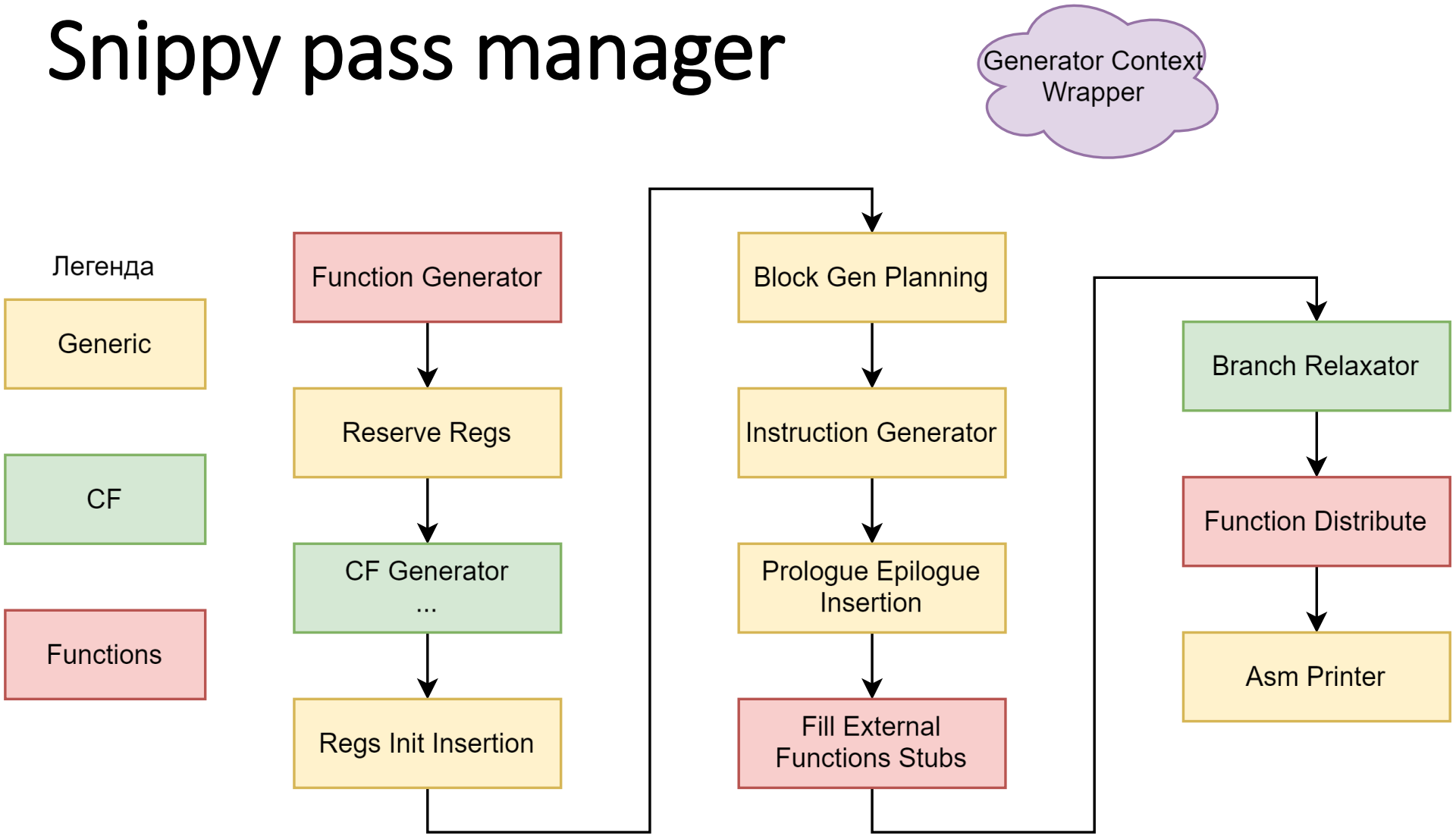
Теорминимум: сводимый CFG

CFG является сводимым, если он не содержит ни одного сильно связанного подграфа с двумя и более вершинами

Теорема: Следующие утверждения о CFG эквивалентны:

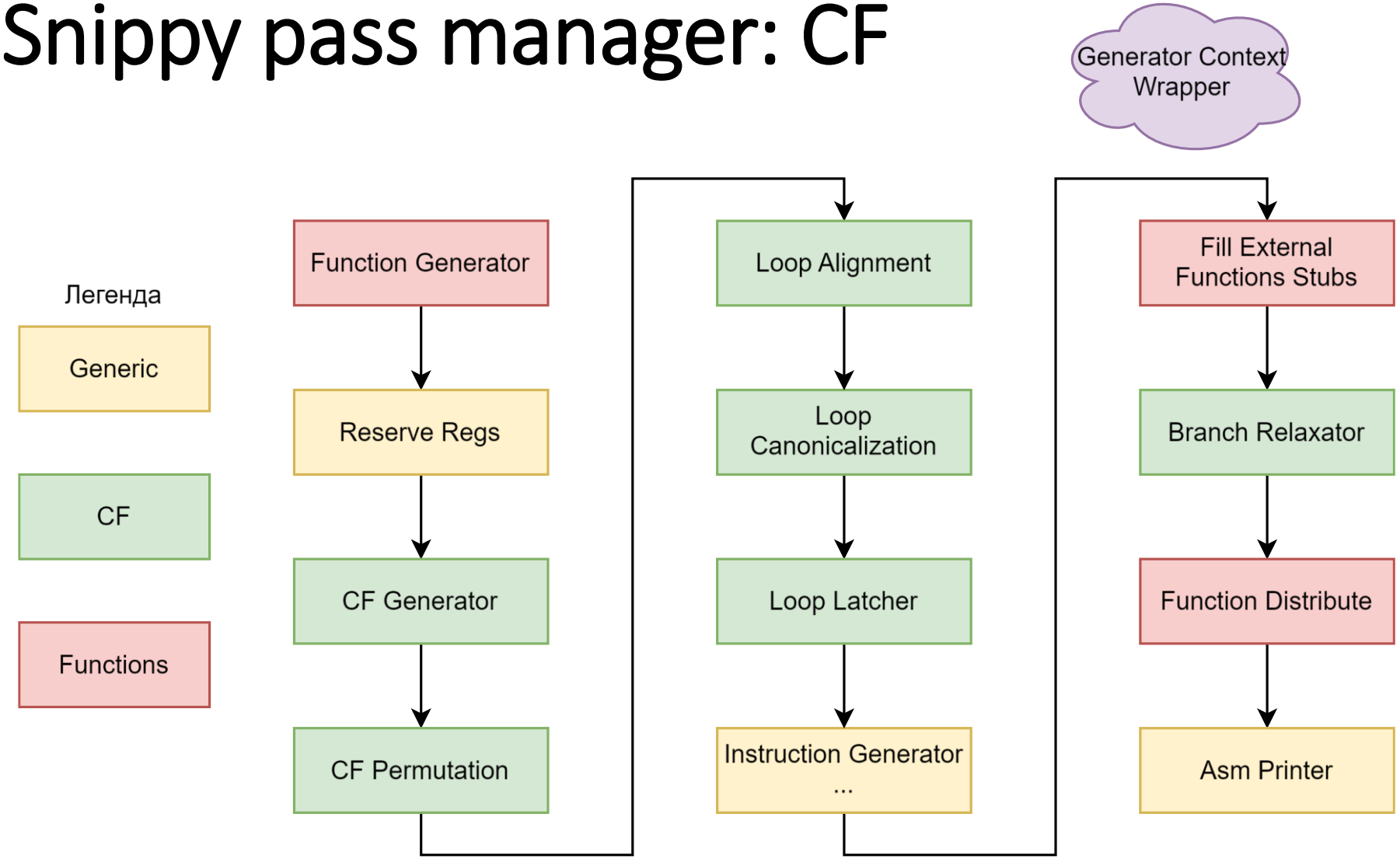
- CFG является сводимым
- CFG может быть трансформирован в одну вершину повторяющимися применениями T_1 и T_2 преобразований:
  - T_1: Убрать ребро-цикл
  - T_2: Выбрать не входную вершину y, которая имеет только одно входящее ребро $x \to y$ и склеить вершины x и y
- Соответствующий CF является структурированным

Snippy pass manager



[llvm-snippy/lib/FlowGenerator.cpp]

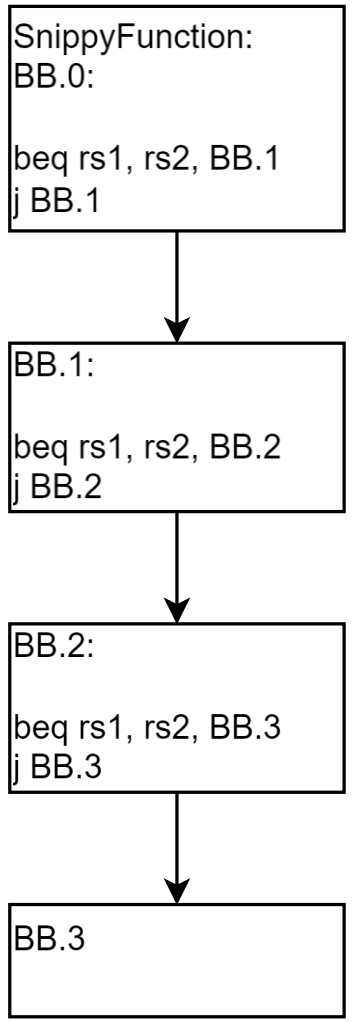
Snippy pass manager: CF



[llvm-snippy/lib/FlowGenerator.cpp]

CF Passes: CFGenerator

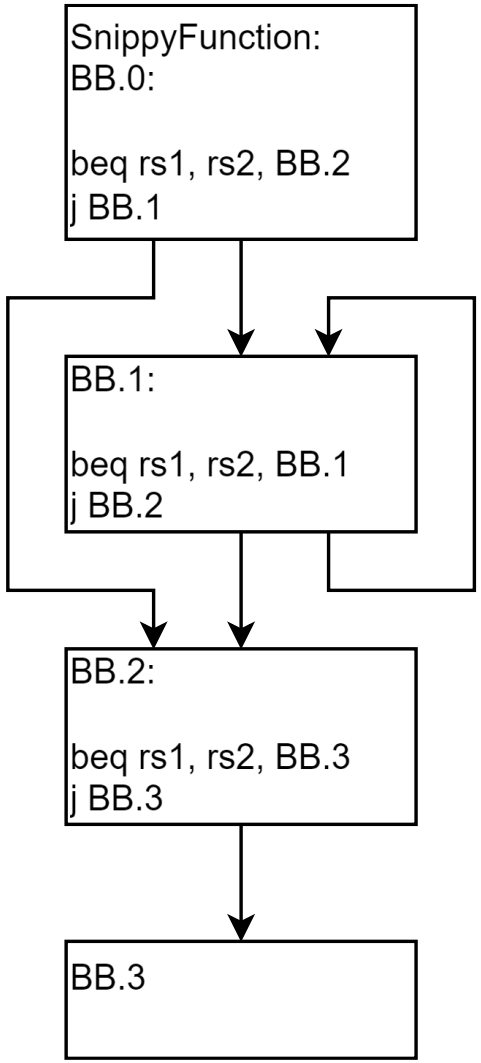
- Генерирует CF инструкции из гистограммы
- Генерирует по одной CF инструкции на базовый блок
- Для каждого условного перехода генерируется fallback jump



llvm-snippy/lib/CFGeneratorPass.cpp

CF Passes: CFPermutation

- Запутывает сгенерированный CF, сохраняя его структурность
- Запутывает шаг за шагом, пока больше не может запутать



llvm-snippy/lib/CFPermutationPass.cpp

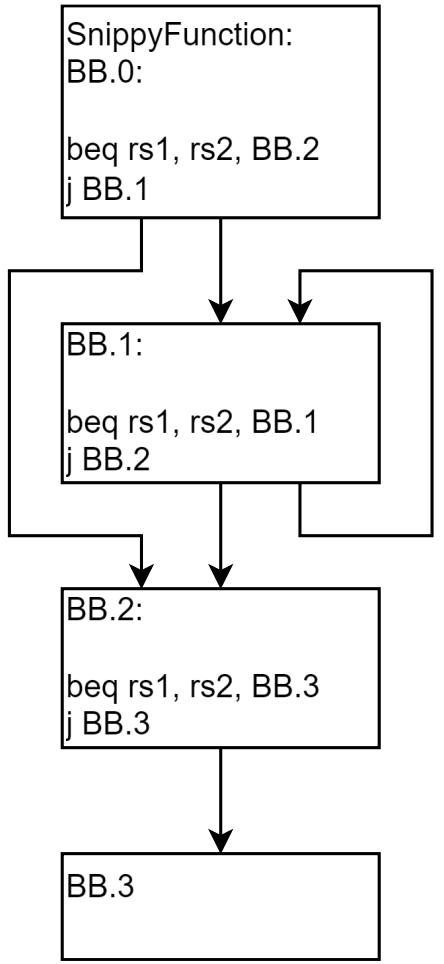
CF Passes: Алгоритм CFPermutation

1. Для каждой вершины CFG создается набор дуг, которые могут быть созданы без нарушения структурности
2. Случайно выбирается незапутанная вершина
3. Для выбранной вершины выбирается случайная дуга из набора
4. Для всех вершин обновляются наборы возможных дуг:
    1. Для всех вершин до начала новой дуги из набора удаляются все вершины внутри дуги
    2. Для всех вершин после новой дуги из набора удаляются все вершины внутри дуги
    3. Для всех вершин внутри дуги из набора удаляются все вершины извне дуги
5. Пункты 2-4 повторяются пока остается хоть одна вершина с непустым набором возможных дуг

llvm-snippy/lib/CFPermutationPass.cpp

CF Passes: Проблемы CFPermutation

Какие вы видите проблемы со сгенерированным CFG?



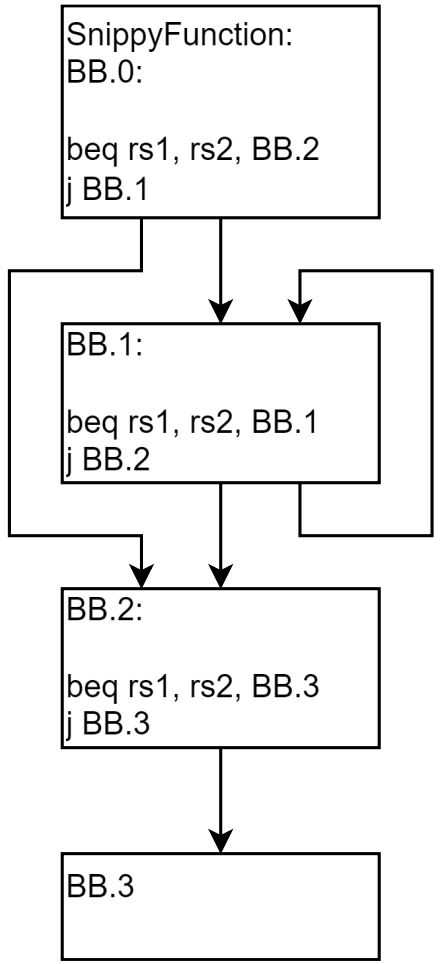
[llvm-snippy/lib/CFPermutationPass.cpp]

CF Passes: Проблемы CFPermutation

1. BB.1 – *бесконечный цикл*

2. InstructionGenerator может сгенерировать *слишком много* инструкций – дальность прыжка может не поместиться в кодировку

Рассмотрим как они решаются



llvm-snippy/lib/CFPermutationPass.cpp

Проблема бесконечных циклов в llvm-snippy

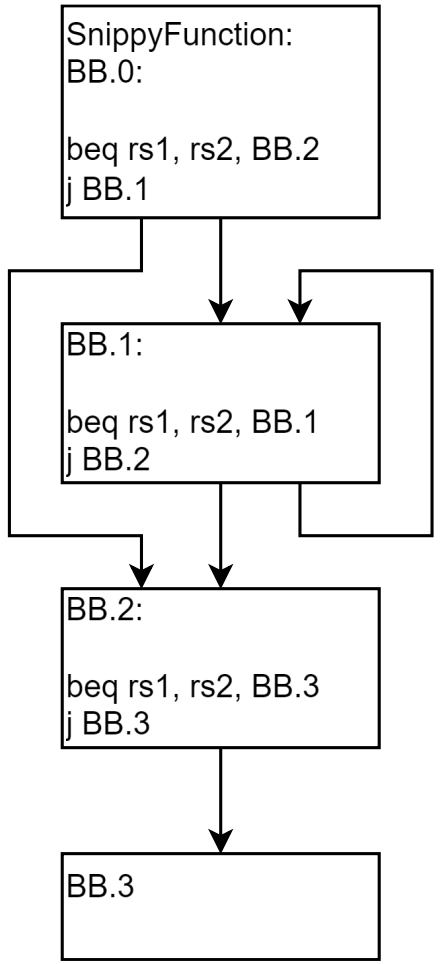
Возможное решение:

Генерация счетчиков цикла:

- Генерация инициализации счетчика
- Генерация инкрементирования счетчика

Генерация счетчиков на лету вместе с запутыванием CF кажется переусложнением

Но как находить циклы в других пассах?



Поиск циклов

- Поиск циклов является довольно классической задачей в компиляторах
- В LLVM информацию о циклах можно получить с помощью специального анализа:
  - LoopInfo для LLVM IR
  - MachineLoopInfo для LLVM MIR
- А как LLVM строит этот анализ?

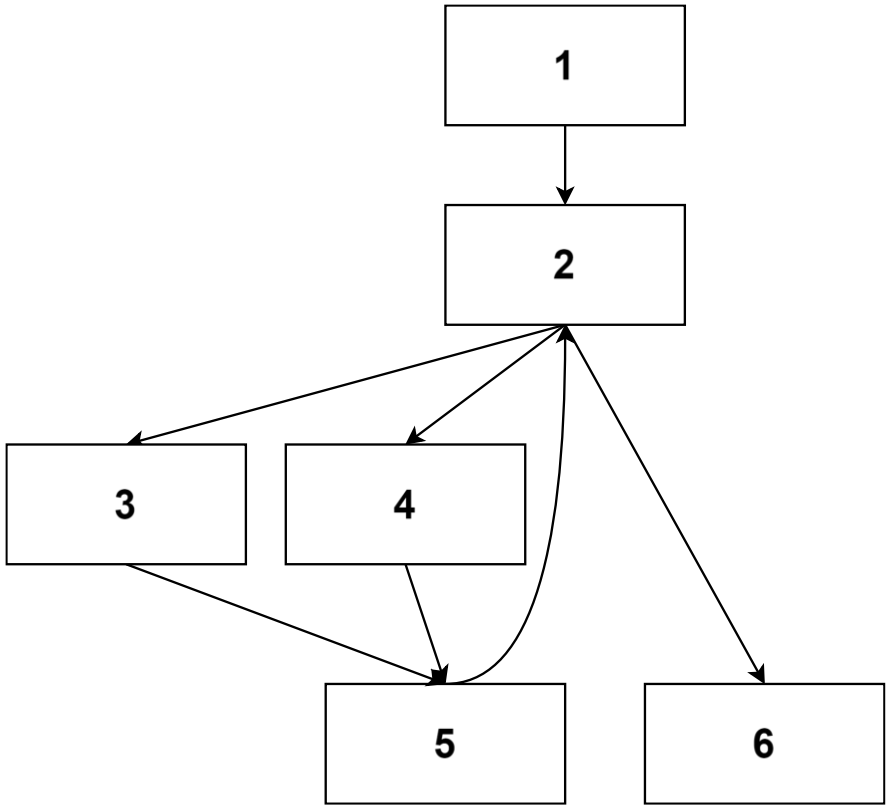
Еще немного теории: отношение доминации

Вершина *d* графа потока управления **доминирует** вершину *n*, если любой путь от входной вершины до *n* проходит через *d*

Для вершины *n* вершина *d* является **доминатором**

Какие вершины доминирует:

- **1**?
- **2**?
- **4**?

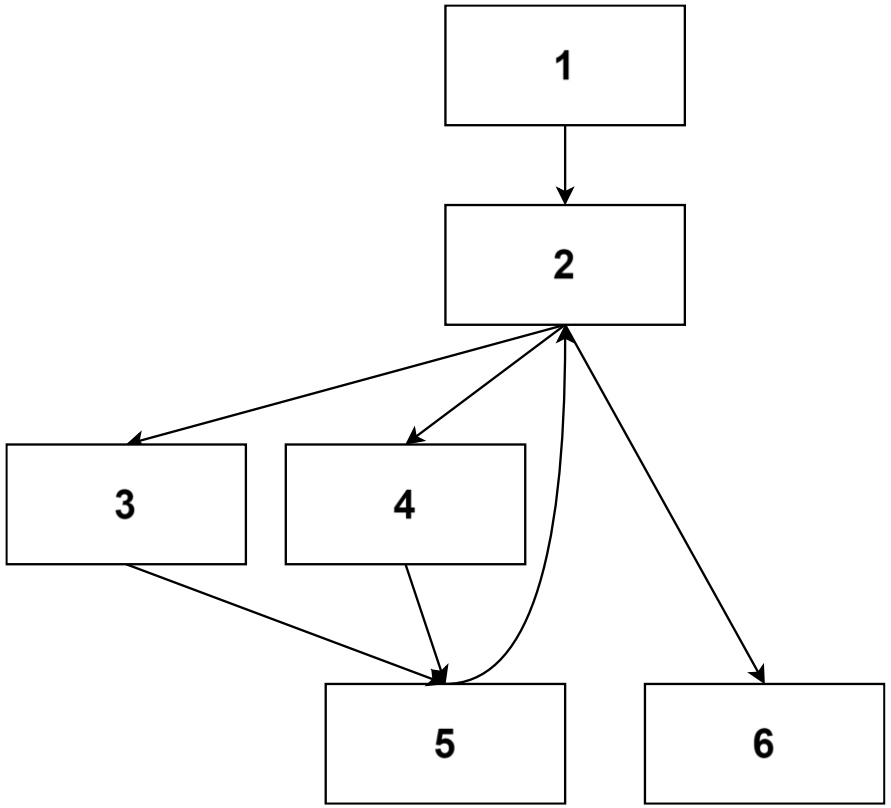


Еще немного теории: отношение доминации

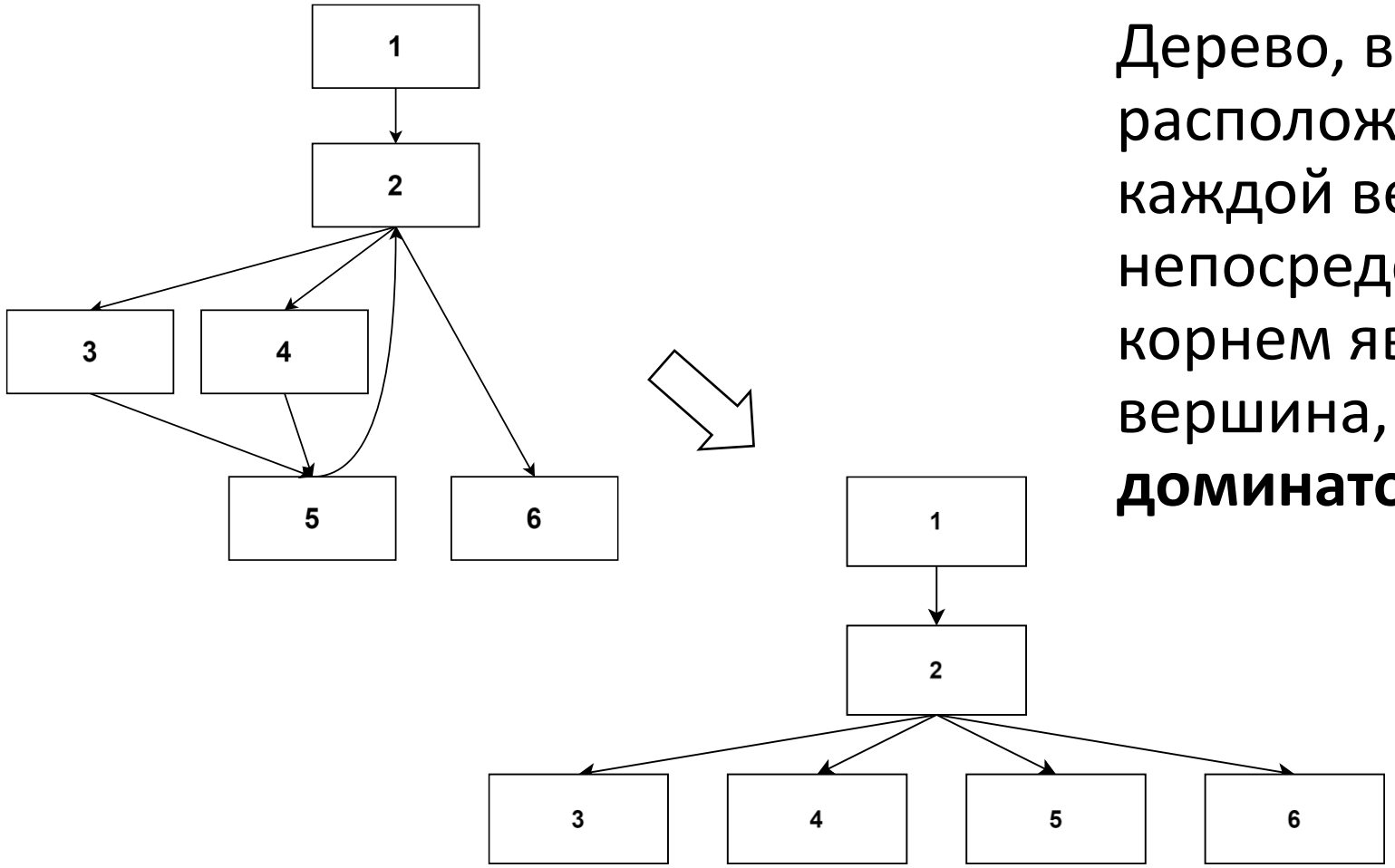
Вершина *d* **строго доминирует** вершину *n*, если *d* доминирует *n* и *d* не является *n*

Непосредственным доминатором вершины *n* является доминатор, находящийся ближе всего к *n* вдоль любого ациклического пути от входной вершины до *n*

Какие вершины непосредственно доминирует **1**, **2**, **4**?



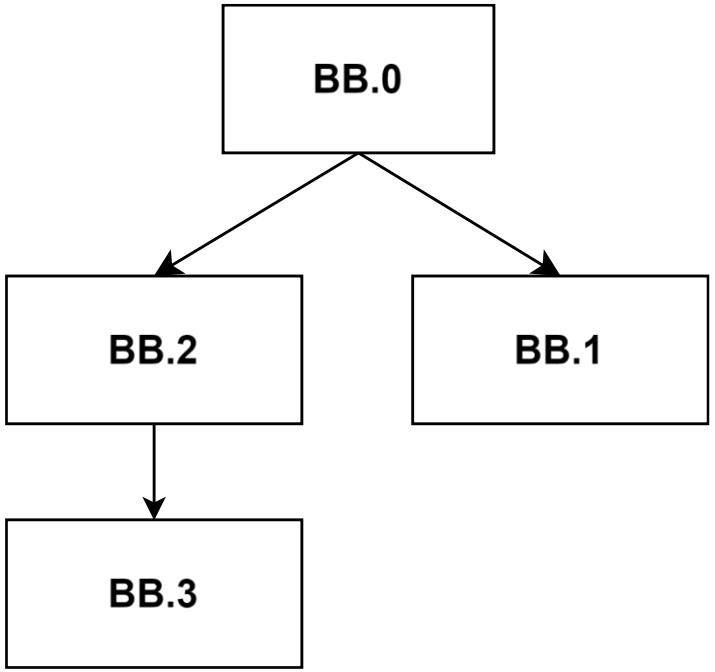
Еще немного теории: дерево доминаторов

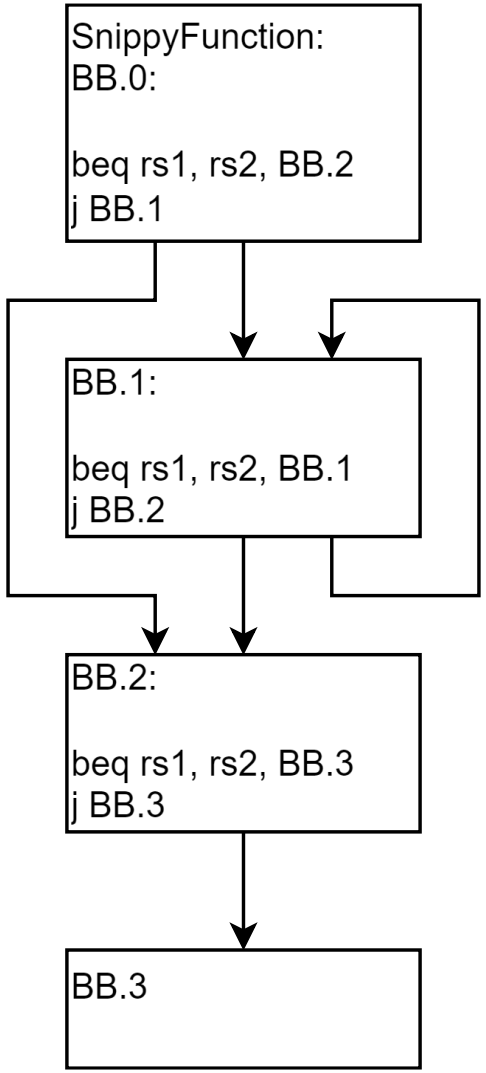


Дерево, в котором все вершины расположены так, что предком каждой вершины является ее непосредственный доминатор, а корнем является входная вершина, называется **деревом доминаторов**

Так как найти циклы?

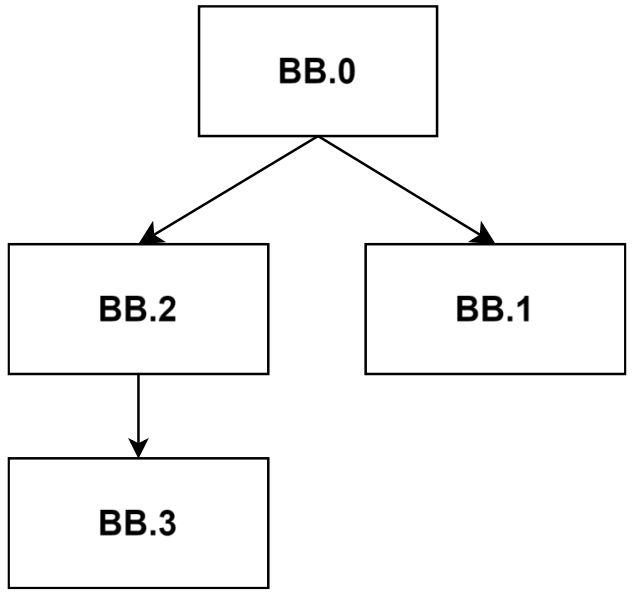
Построим дерево доминаторов для этого CFG:

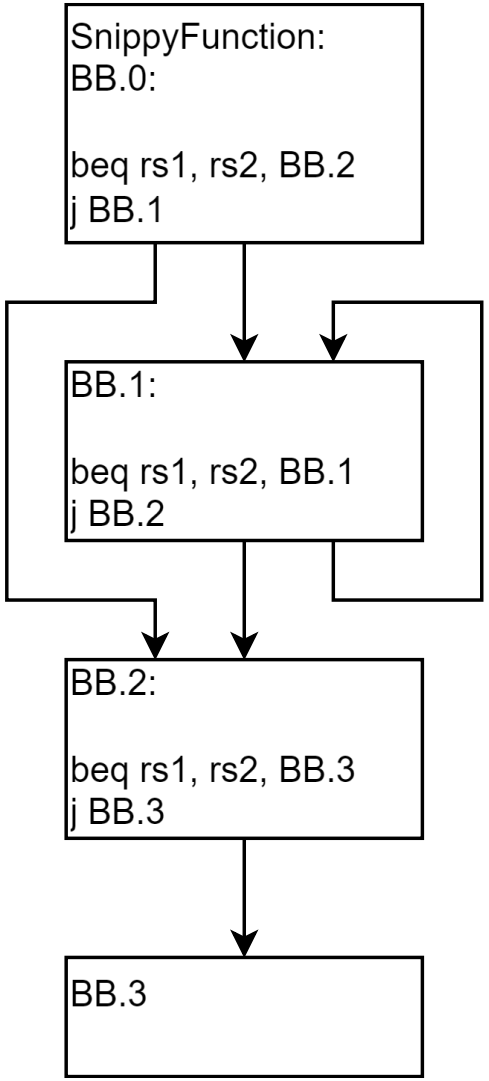




Так как найти циклы?

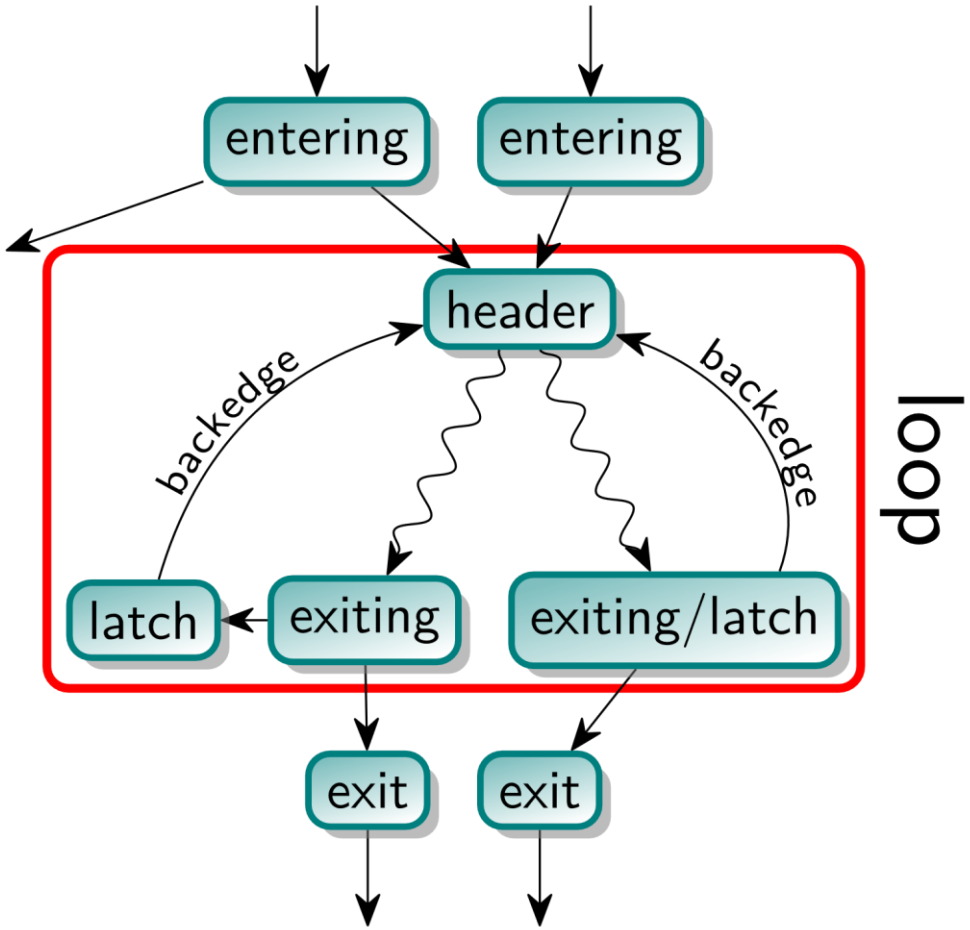
- Идем по дереву в обратном порядке (post-order)
- Для текущего блока выделяем два множества:
  1. Блоки-предшественники, из которых есть ребра в текущий блок
  2. Блоки, доминируемые текущим блоком
- Если пересечений этих множеств непустое, то текущий блок – заголовок цикла





Терминология циклов

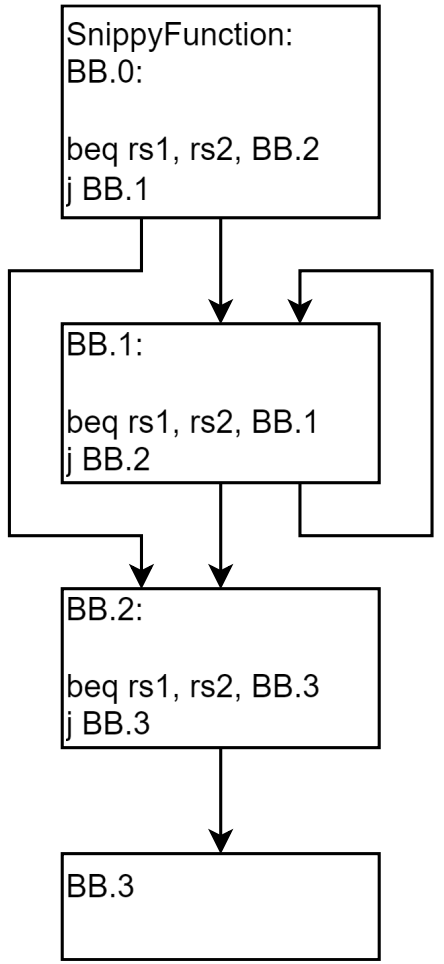
- **Entering** – блок вне цикла, из которого можно попасть в цикл
- **Header** – блок, принадлежащий циклу, который доминирует все блоки цикла
- **Latch** – блок, принадлежащий циклу, из которого идет обратная дуга в header
- **Exiting** – блок, принадлежащий циклу, из которого есть дуга вне цикла
- **Exit** – блок вне цикла, в который есть дуга из exiting блока
- Если entering блок только один, и его единственная дуга – дуга в header, то такой блок называется **preheader**



LLVM Loop Terminology

Проблема бесконечных циклов в llvm-snippy: Инициализируем счетчик

Согласно рассмотренной терминологии циклов, в каком блоке необходимо инициализировать счетчик?

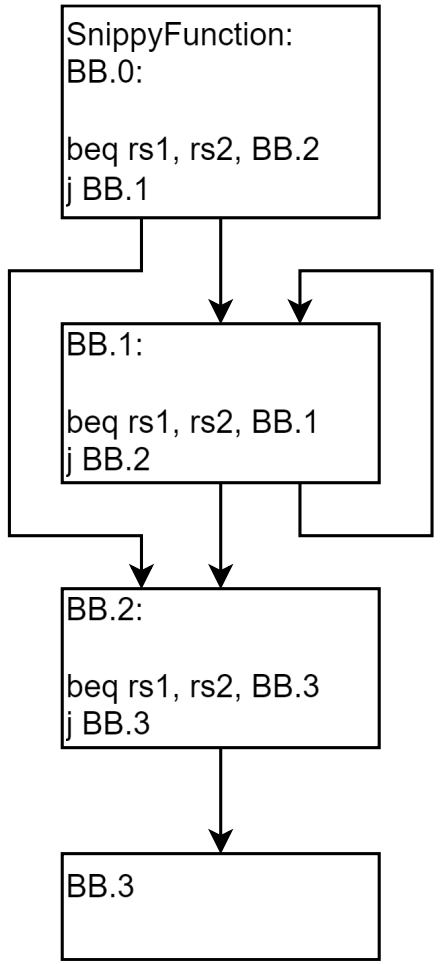


Проблема бесконечных циклов в llvm-snippy: Инициализируем счетчик

Согласно рассмотренной терминологии циклов, в каком блоке необходимо инициализировать счетчик?

Инициализировать счетчик нужно в едином месте до цикла – выбираем **preheader**

Где в данном CFG preheader цикла?



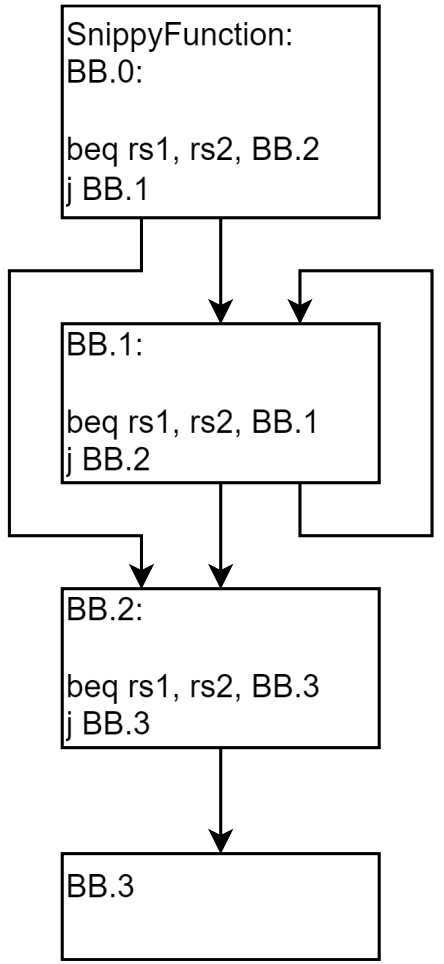
Проблема бесконечных циклов в llvm-snippy: Инициализируем счетчик

Согласно рассмотренной терминологии циклов, в каком блоке необходимо инициализировать счетчик?

Инициализировать счетчик нужно в едином месте до цикла – выбираем **preheader**

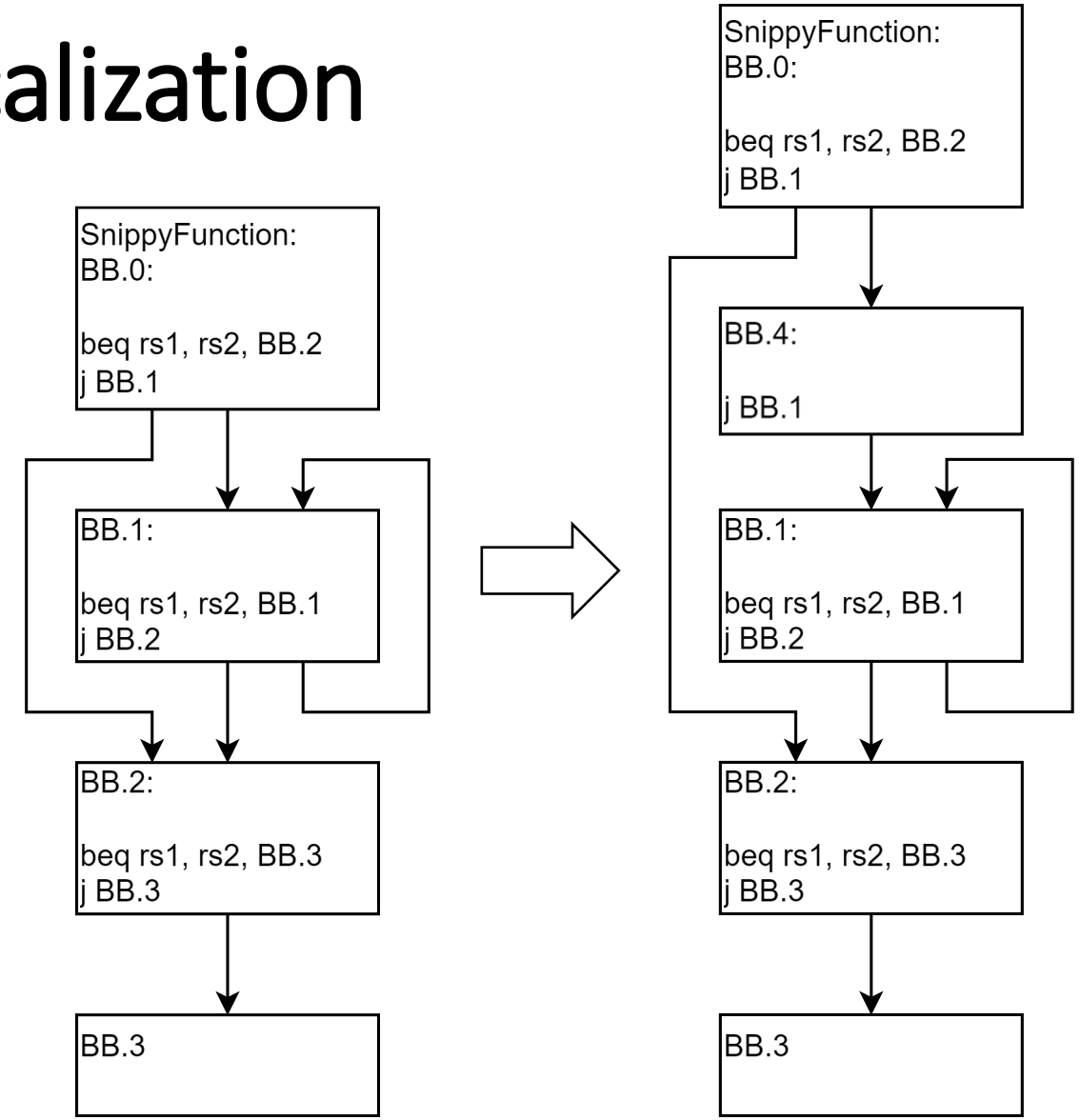
Где в данном CFG preheader цикла?

У данного цикла *нет* preheader блока



CF Passes: LoopCanonicalization

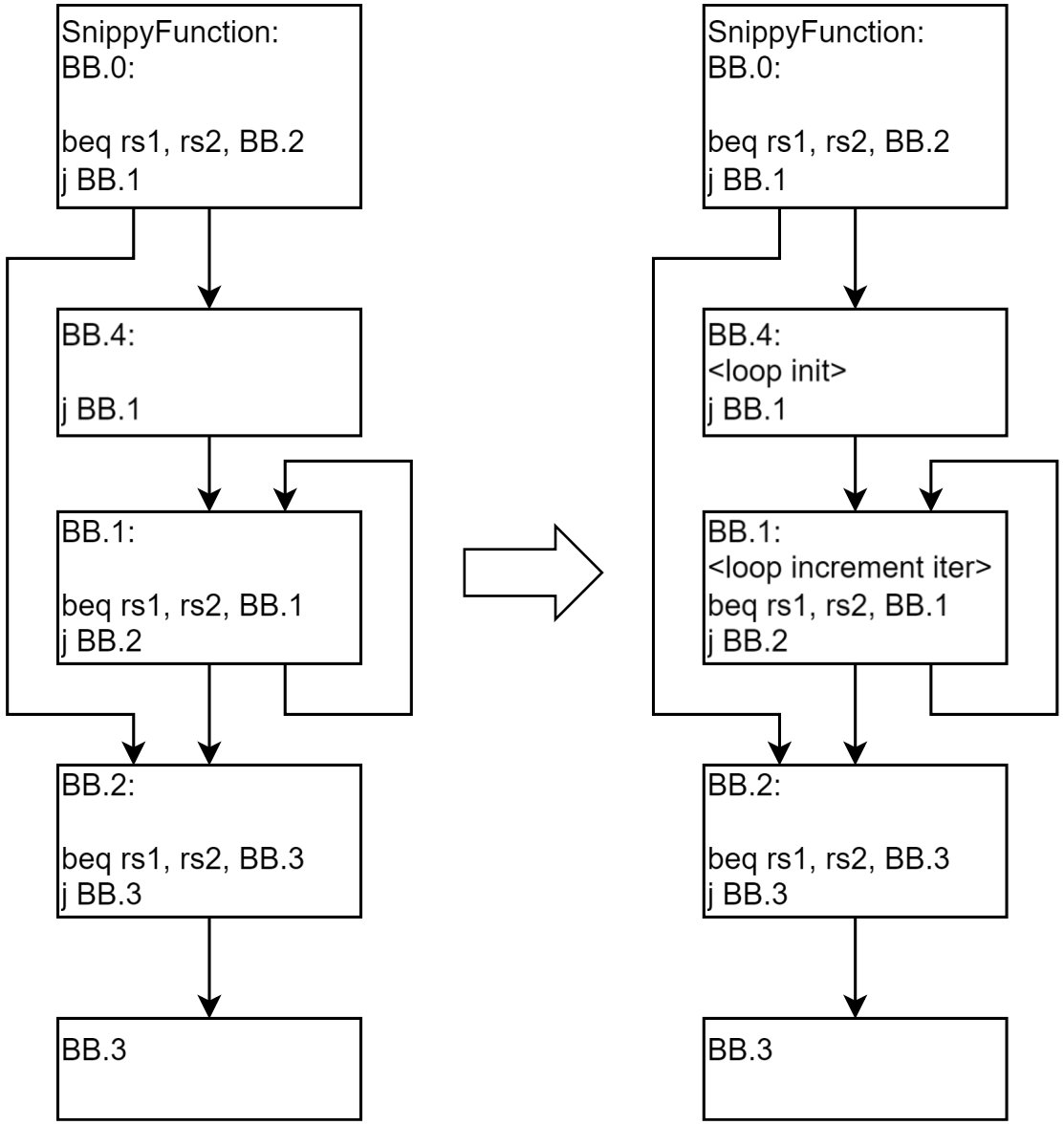
- Проходится по всем циклам и проверяет, есть ли у них preheader
- Если preheader отсутствует, то создает его
- Использует анализ LLVM для поиска циклов



llvm-snippy/lib/LoopCanonicalizationPass.cpp

CF Passes: LoopLatcher

- Для каждого цикла:
  - Вставляет инициализацию цикла в preheader
  - Вставляет инкрементацию счетчика в latch блок
  - Резервирует регистры используемые для расчета условия выхода из цикла
- Расстановка счетчиков target-специфична и варьируется даже внутри разных опкодов одной целевой архитектуры



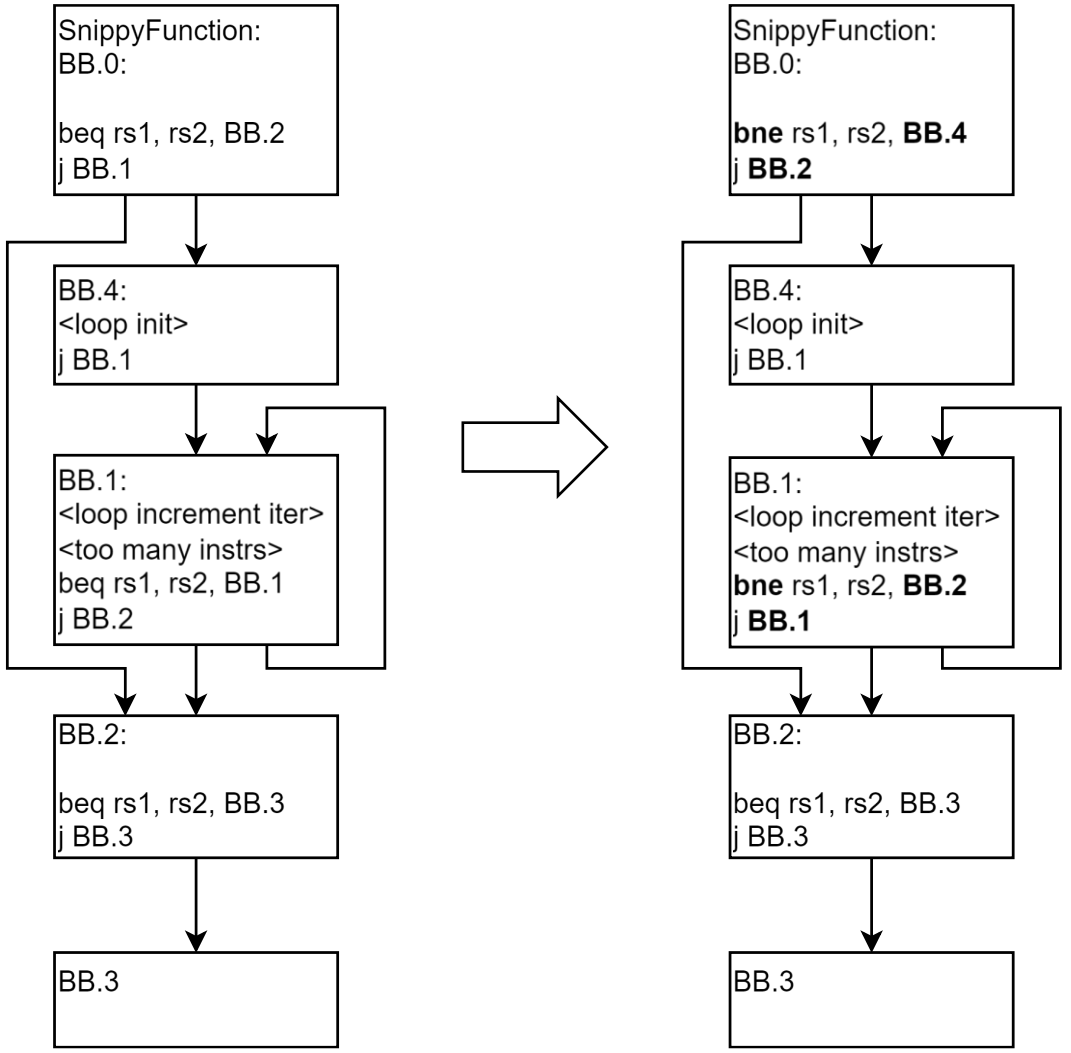
[llvm-snippy/lib/LoopLatcherPass.cpp]

Проблема слишком длинных бранчей в llvm-snippy

- Структурированный поток управления сгенерирован
- Циклы всегда выполняются за конечное число итераций
- Но в теле цикла может оказаться слишком много инструкций – в RISC-V branch инструкции могут прыгать максимум на ±4Kb, то есть ±1000 инструкций

CF Passes: BranchRelaxator

- Для каждой CF инструкции:
  - Проверяем как далеко происходит переход
  - Если длина перехода не помещается в кодировку, делаем – делаем релаксацию:
    - Compressed branch заменяем на обычный
    - Обычный бранч заменяем на противоположный и меняемся целями прыжка с fallback jump’ом (у jump больше места в кодировке под immediate)



llvm-snippy/lib/LoopCanonicalizationPass.cpp

To be continued …

На следующем занятии

- Узнаем про еще один вид стандартных компиляторных графов – Call Graph
- Оценим возможность завершения выполнения случайно сгенерированных функций, вызывающих друг друга
- Рассмотрим как в llvm-snippy решена проблема бесконечных рекурсий