Теормин по курсу Конструирование Компиляторов

Написано пчелом с АСВК по лекциям деда,
старым теорминам и драгонбуку.

Учтите, что некоторые формулировки деду
не нравились, даже если они были взяты 1 в 1
с драгонбука или его же лекций.

Внимание: данный PDF включает в себя все LaTeX исходники.
Чтобы извлечь прикрепленные файлы, воспользуйтесь:

- Linux: утилитой `pdfdetach`
- Windows/Macos: `AdobeAcrobat`.

Пожалуйста – исправляйте, дополняйте, улучшайте :)

15 Января 2023 г.

Оглавление

1 Промежуточные представления высокого, среднего, низкого уровня. . . . 3
2 Базовый блок. . . . 3
3 Граф потока управления. . . . 3
4 Локальная оптимизация. . . . 4
5 Глобальная оптимизация. . . . 4
6 Ориентированный ациклический граф. . . . 4
7 Остовное дерево. . . . 4
8 Поток данных. . . . 4
9 Точка программы. . . . 4
10 Состояние программы. . . . 4
11 Передаточная функция инструкции. . . . 5
12 Передаточная функция базового блока. . . . 5
13 Определение переменной. . . . 5
14 Использование переменной. . . . 5
15 Достигающее определение. . . . 5
16 Живая переменная. . . . 5
17 Доступное выражение. . . . 5
18 Избыточные вычисления. . . . 5
19 Полурешетка с операцией «сбор». . . . 6
20 Верхний и нижний элементы полурешетки. . . . 6
21 Наибольшая нижняя граница двух элементов полурешетки. . . . 6
22 Диаграмма полурешетки. . . . 6
23 Структура потока данных. . . . 6
24 Монотонная структура анализа потока данных. . . . 7
25 Дистрибутивная структура анализа потока данных. . . . 7
26 Фиксированная точка системы уравнений. . . . 7
27 Максимальная фиксированная точка системы уравнений. . . . 7
28 Решение сбором по всем выполнимым путям. . . . 7
29 Решение сбором по всем путям. . . . 8
30 Консервативность анализа. . . . 8
31 Доминатор. . . . 8
32 Дерево доминаторов. . . . 8
33 Постдоминатор. . . . 8
34 Дерево постдоминаторов. . . . 8
35 Граница доминирования. . . . 9
36 Обратная граница доминирования. . . . 9
37 Зависимость по данным. . . . 9
38 Зависимость по управлению. . . . 10
39 Эквивалентность по управлению. . . . 10
40 Сворачивание констант. . . . 10
41 Распространение копий. . . . 10
42 Остовное ребро. . . . 10
43 Прямое ребро. . . . 10
44 Обратно направленное ребро. . . . 10
45 Обратное ребро. . . . 10
46 Естественный цикл. . . . 11
47 Инвариант цикла. . . . 11
48 Инструкция, инвариантная относительно цикла. . . . 11
49 Индуктивная переменная. . . . 11
50 Семейство индуктивных переменных. . . . 11
51 Частичная раскрутка циклов. . . . 11
52 Гнездо циклов. . . . 12

53 Выравнивание циклов. . . . 12
54 Слияние циклов. . . . 12
55 Бесполезный код. . . . 12
56 Недостижимый код. . . . 12
57 Форма статического единственного присваивания (SSA-форма). . . . 13
58 φ-функция. . . . 13
59 Максимальная SSA-форма. . . . 13
60 Частично усеченная SSA-форма. . . . 13
61 Критическое ребро. . . . 13
62 Суперблок. . . . 13
63 Область графа потока управления. . . . 14
64 Дерево управления. . . . 14
65 Сводимость ГПУ 14
66 Профилирование. . . . 15
67 Инструментирование и семплирование. . . . 15
68 Межпроцедурный анализ. . . . 15
69 Граф вызовов. . . . 16
70 Контекст. . . . 16
71 Чувствительность к контексту. . . . 16
72 Функции скачка. . . . 16
73 Машинно-ориентированная оптимизация. . . . 17
74 Режимы адресации. . . . 17
75 Выбор команд. . . . 17
76 Переписывание дерева. . . . 18
77 Что такое распределение регистров. . . . 18
78 Интервалы жизни переменных. . . . 18
79 Расщепление интервалов жизни. . . . 18
80 Слияние интервалов жизни. . . . 19
81 Граф конфликтов. . . . 19
82 Слив (сброс) интервалов жизни. . . . 19
83 Планирование кода. . . . 19
84 Граф зависимостей по данным. . . . 20
85 Граф зависимостей программы. . . . 20
86 Топологический порядок. . . . 20
87 Топологическая сортировка. . . . 20
88 Программная конвейеризация. . . . 21
89 Ресурсы. Виды ресурсов. . . . 21
90 Таблица резервирования ресурсов. . . . 21
91 Пространственная локальность данных. . . . 21
92 Временна́я локальность данных. . . . 22
93 Для чего необходимо обеспечивать локальность данных? . . . 22

1. Промежуточные представления высокого, среднего, низкого уровня.

Промежуточные представления высокого уровня (HIR) – это атрибутированное абстрактное синтаксическое дерево.

Промежуточные представления среднего уровня (MIR, Middle-level Intermediate Representation) включает в себя

- Трехадресный код (трехадресные инструкции, "четверки"):
  - Инструкции присваивания: `x` $\leftarrow$ `op`, `y`, `z`.
  - Инструкции перехода: `goto, ifTrue, ifFalse`
  - Процедуры: `call, return, param`
- Таблицу символов: переменные, их имена в программе и атрибуты, какие как область видимости, тип, для имен функций – число параметров и т.п.

Промежуточные представления низкого уровня (LIR) – используется для машинно-зависимых задач вроде распределения регистров и выбора подходящих команд.

2. Базовый блок.

Базовым блоком (ББ) называется последовательность следующих друга за другом инструкций MIR, обладающая следующими свойства:

1. поток управления может входить в базовый блок только через его первую инструкцию (т.е. в программе нет переходов в средину базового блока);
2. поток управления покидает базовый блок без останова или ветвления, кроме, возможно последней инструкции базового блока.

Начало базового блока (НББ) – это либо

- первая инструкция программы,
- помечанная инструкция программы (`L1: smth smth;`)
- инструкция, следующая за инструкцией перехода (`goto L2; smth smth;`)

3. Граф потока управления.

Граф потока управления (ГПУ) такой граф, что:

- Вершины графа – базовые блоки
- Дуги соединяют базовые блоки по инструкциям перехода и последовательно. Например, если в конце базового блока стоит инструкция условного перехода, то из него будет выходить две дуги. Если же первая инструкция ББ имеет метку, то в него будет входить несколько дуг: из тех ББ, в которых последняя инструкция – (без)условный переход на эту метку.

4. Локальная оптимизация.

Оптимизация - выполнение следующих преобразований:

- Удаление общих выражений (инструкций, повторно вычисляющих уже вычисленные значения).
- Удаление мертвого кода (инструкций, вычисляющих значения, которые впоследствии не используются).
- Сворачивание констант (выражение `2 * 3.14` можно заменить значением `6.28`).
- Изменение порядка инструкций, там, где это возможно (чтобы сократить время хранения временного значения на регистре).
- Локальное снижение стоимости вычислений, т.е. замена более дорогих операций более дешевыми (например, `2*x -> x+x`, `x/2 -> x*0.5`).

Все вышеперечисленные преобразования можно выполнить за один просмотр базового блока, представив его в виде ориентированного ациклического графа.

5. Глобальная оптимизация.

Оптимизация в пределах процедуры, которая учитывает потоки данных между базовыми блоками.

6. Ориентированный ациклический граф.

Ориентированный ациклический граф – ориентированный граф, в котором отсутствуют направленные циклы, но могут быть параллельные пути.

7. Остовное дерево.

Остовное дерево – ацикличный подграф (дерево) графа, который состоит из минимального подмножества ребер, таких, что из одной вершины графа можно попасть в любую другую вершину двигаясь по этим ребрам.

8. Поток данных.

Поток данных – множество состояний на входе и выходе в базовый блок.

9. Точка программы.

Точка программы – место (строка), между двумя инструкциями программы.

10. Состояние программы.

Состояние программы – множество значений всех переменных включая значения переменных в стеке.

11. Передаточная функция инструкции.

Состояния во входной и выходной точках фрагмента определяет передаточные функции этого фрагмента:

- Передаточная функция *прямого обхода* (f) переводит состояние во входной точке в состояние в выходной точке.
- Передаточная функция *обратного обхода* (f^b) переводит состояние в выходной точке в состояние во входной точке.

12. Передаточная функция базового блока.

Передаточная функция базового блока равна композиции передаточных функций его инструкций.

13. Определение переменной.

- *Определением переменной* `x` называется инструкция, которая присваивает значение переменной `x` ($\texttt{x} \leftarrow \texttt{smth}$).
- Каждое определение переменной `x` *убивает* все другие ее определения.
- Каждая переменная может иметь несколько определений.

14. Использование переменной.

Использованием переменной `x` является инструкция, одним из операндов которой является переменная `x` ($\texttt{z} \leftarrow \texttt{x} + \texttt{y}$).

15. Достигающее определение.

Определение `d` *достигает* точки `p`, если существует путь от точки, непосредственно следующей за `d`, к точке `p`, т.ч. вдоль этого пути `d` не убивается.

16. Живая переменная.

Живая переменная – переменная, вычисленная в данном базовом блоке, а далее используемая в других базовых блоках.

17. Доступное выражение.

Доступным выражением в точке $\texttt{p}_1$ называется такое выражение `e`, что если `e` вычисляется на любом пути от точки $\texttt{p}_2$, где $\texttt{p}_2$ – произвольная, до точки $\texttt{p}_1$, то переменные, входящие в состав `e`, не переопределяются между последним таким вычислением и точкой $\texttt{p}_1$.

18. Избыточные вычисления.

Избыточные вычисления – повторный или неоднократный пересчет одного и того же.

19. Полурешетка с операцией «сбор».

Полурешетка – это абстрактная алгебраическая структура, над элементами которой определена абстрактная операция «сбор», обладающая свойствами операций пересечения и объединения. (Пояснение: полурешетки используются для обобщения алгоритмов анализа потока данных.)

Операция «сбор» ($\wedge$) – операция, определенная на множестве полурешетки L, т.ч. $\forall x, y, z \in L$:

- $x \wedge x = x$ – однозначность (идемпотнетность).
- $x \wedge y = y \wedge x$ – коммутативность.
- $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ – ассоциативность.

20. Верхний и нижний элементы полурешетки.

Верхний элемент $\top \in L$, т.ч. $\forall x \in L$ выполняется $\top \wedge x = x$.

Нижний элемент $\bot \in L$, т.ч. $\forall x \in L$ выполняется $\bot \wedge x = \bot$.

21. Наибольшая нижняя граница двух элементов полурешетки.

Наибольшей нижней границей элементов $x, y \in L$ полурешетки L, называется такой элемент $g \in L$, что не существует любого другого элемента $z \in L$, больше него.

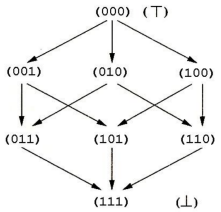
$$g = \overline{\inf}(x, y) \;\Leftrightarrow\; g \le x,\; g \le y \;\text{ и }\; \forall z \in L : z \le x,\; z \le y \;\Rightarrow\; z \le g$$

Утв. Если $x, y \in L$, $\langle L, \wedge \rangle$ – полурешетка L с операцией сбора $\wedge$, то $\overline{\inf}(x, y) = x \wedge y$.

22. Диаграмма полурешетки.

Диаграмма полурешетки $\langle L, \wedge \rangle$ представляет собой граф, узлами которого являются элементы L, а ребра направлены от `x` к `y`, если $y \le x$.

Пример для $\langle U, \cup \rangle$, $|U| = 8$, элемент множества U представляется битовым 3-вектором. (Здесь $\cup$ – операция объединения).



23. Структура потока данных.

Структурой потока данных – называется четверка $\langle D, F, L, \wedge \rangle$, где D – направление анализа (Forward/Backward), F – семейство передаточных функций, L – поток данных, $\wedge$ – операция сбора.

Семейство передаточных функций называется *замкнутым*, если:

1. F – содержит тождественную функцию I: $\forall x \in L : I(x) = x$.
2. F – замкнуто относительно композиции $\forall f, g \in F \Rightarrow h(x) = g(f(x)) \in F$.

24. Монотонная структура анализа потока данных.

Структура потока данных $\langle D, F, L, \wedge \rangle$, называется *монотонной*, если:

$$\forall x, y \in L,\ \forall f \in F \quad f(x \wedge y) \leq f(x) \wedge f(y)$$

или

$$\forall x, y \in L,\ \forall f \in F \quad x \leq y \ \Rightarrow\ f(x) \leq f(y)$$

25. Дистрибутивная структура анализа потока данных.

Структура потока данных $\langle D, F, L, \wedge \rangle$, называется *дистрибутивной*, если:

$$\forall x, y \in L,\ \forall f \in F \quad f(x \wedge y) = f(x) \wedge f(y)$$

Утв. Из дистрибутивности следует монотонность.

26. Фиксированная точка системы уравнений.

Фиксированная точка (FP) системы уравнений называется такая точка, которая является решением:

$$\texttt{In}[B] = \bigwedge_{P\ \in\ \texttt{Pred}(B)} f_P(\texttt{In}[P])$$

27. Максимальная фиксированная точка системы уравнений.

Максимальная фиксированная точка (MFP) системы уравнений $\texttt{In}[B] = \wedge f_p(\texttt{In}[P]),\ p \in \texttt{Pred}(B)$, представляет собой решение $\{\texttt{In}[B_i]^{max}\}$ этой системы, обладающее тем свойством, что для любого другого решения $\{\texttt{In}[B_i]\}$ выполняется условия $\texttt{In}[B_i] \leq \texttt{In}[B_i]^{max}$, где $\leq$ – полурешеточное отношение частичного порядка.

28. Решение сбором по всем выполнимым путям.

Выполнимым путем называется такой путь P, что существует такое выполнение программы (наличие таких входных данных), которое следует в точности по этому пути.

Идеальным решением системы уравнений потока данных для $\texttt{In}[B_k]$ будет:

$$\texttt{Ideal}[B_k] = \bigwedge_{p \in P} f_p(\texttt{In}[l_{\texttt{Entry}}]),\ P = \{P_1, P_2, \ldots\}$$

Здесь P – множество *всех выполнимых* путей от `Entry` до B_k, $l_{\texttt{Entry}}$ – граничное условие.
Решение названо *идеальным*, так как:

- оно наиболее точное
- вычислить его почти никогда не удается, так как поиск всех возможных путей выполнения – задача неразрешимая.

29. Решение сбором по всем путям.

Решение сбором по всем путям (MOP-решение) от `Entry` до входа в B_k определяется соотношением

$$\texttt{MOP}[B_k] = \bigwedge_{P \in Q} f_P(l_{\texttt{Entry}})$$

где $Q = \{P_1, P2, \ldots\}$ – множество всех путей от `Entry` до входа в B_k.

Пути, рассматриваемые в MOP-решении – это надмножество всех выполнимых путей: MOP-решение собирает значения потоков данных как для всех выполнимых путей, так и для путей, которые не могут быть выполнены. Следовательно, для всех B_k выполняется соотношение $\texttt{MOP}[B_k] \leq \texttt{Ideal}[B_k]$.

30. Консервативность анализа.

Решение FP системы уравнений потока данных называется *консервативным*, если выполняется условие FP $\leq$ MFP.

31. Доминатор.

В ГПУ вершина `d` является *доминатором* вершины `n`, если любой путь от вершины `Entry` до вершины `n` проходит через вершину `d`. Каждая вершина является доминатором самой себя. Обозначение: d `dom` n или $d = \texttt{Dom}(n)$.

Строгим доминатором называется такой доминатор, который не совпадает с самим собой. Обозначение: d `sdom` n.

Непосредственным доминатором называет такой доминатор i вершины n, что

$$\nexists m,\ m \neq i,\ m \neq n\ :\ i\ \texttt{dom}\ m,\ m\ \texttt{dom}\ n$$

Другими словами, это ближайший строгий доминатор к вершине n. Обозначение: $i = \texttt{IDom}(n)$.

32. Дерево доминаторов.

Если соединить дугами каждый блок с его непосредственным доминатором (`SDom`), получится *дерево доминаторов*.

33. Постдоминатор.

В ГПУ вершина `p` называется *постдоминатором* вершины `n`, если каждый путь из `n` в `Exit` проходит через вершину `p`. Обозначение: $p = \texttt{Postdom}(n)$.

Постдоминаторы – это доминаторы обратного ГПУ.

Строгий и *непосредственный* постдоминаторы определяется аналогично тому, как они определены для доминаторов.

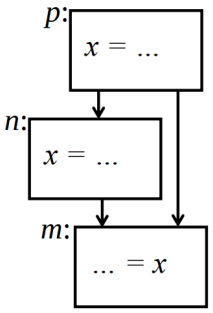
34. Дерево постдоминаторов.

Аналогично дереву доминаторов.

35. Граница доминирования.

Граница доминирования вершины n – множество, которое содержит все первые узлы, достижимые из n, но над которыми n не доминирует, причем это верно для любого пути ГПУ (*первый узел* – тот, который следует непосредственно за узлом n).

Рассмотрим узлы ГПУ p, n и m. Если выполняются условия:



- $n, p \in \texttt{Pred}(m)$
- $p \in \texttt{Dom}(n)$
- $p \in \texttt{Dom}(m)$
- n может быть только нестрогим доминатором m, т.е. если $n \neq m$, то $n \notin \texttt{Dom}(m)$.

тогда m входит в состав *границы доминирования* вершины n. Обозначение: $m \in \texttt{DF}(n)$.

В случае цикла из одного блока вершина может являться своей границей доминирования.

36. Обратная граница доминирования.

Обратной границей доминирования $\texttt{RDF}(n)$ вершины n называется граница доминирования в обратом ГПУ.

37. Зависимость по данным.

Две команды называются *зависимыми по данным*, если изменение порядка их выполнения может привести к изменению результата вычислений.

Виды зависимостей:

- *Истинная зависимость*: чтение после записи – если команда `A` записывает значение в ячейку, а команда `B` считывает значение из этой же ячейки, то они зависимы.
- *Анти-зависимость*: запись после чтения – если команда `A` считывает значение из ячейки, а команда `B` записывает в эту же ячейку, то они зависимы.
- *Зависимость по выходу*: запись после записи – если команды `A` и `B` записывают значения в одну и ту же ячейку, то они зависимы.

Только истинная зависимость является неустранимой, две другие можно разрешить использованием разных ячеек памяти.

В общем случае задача выявления зависимости алгоритмически неразрешима: компилятор обязан предполагать, что две команды могут обращаться к одним и тем же ячейкам, если он не может установить обратное (**консервативность**).

38. Зависимость по управлению.

Две команды *зависимы по управлению*, если результат выполнения одной из команд определяет будет ли выполнена вторая.

Вершина m ГПУ *зависит по управлению* от вершины n если:

- $\exists T$ – не пустой путь от n до m, т.ч. $\forall k \in T \setminus \{n\} : m = \texttt{Postdom}(k)$. То есть если выполнение программы пошло по пути T, то чтобы достичь `Exit`, оно обязательно пройдет через m.
- m не обязательно является строгим постдоминатором n: у n может быть несколько выходов, так что помимо T возможны и другие пути, проходящие через n, но потом ведущие не в m, а в другие вершины.

Обратная граница доминирования (RDF) позволяет определять границы зависимостей по управлению.

39. Эквивалентность по управлению.

Два базовых блока B_i и B_j *эквивалентны по управлению*, если B_i выполняется $\Leftrightarrow$ выполняется B_j.

Утв. Если $B_i = \texttt{Dom}(B_j)$ и $B_j = \texttt{Postdom}(B_i)$, то эти блоки эквивалентны по управлению.

40. Сворачивание констант.

Сворачивание констант – процесс вычисления константных выражений на этапе компиляции и замена их на результат (`2 * 3.14` заменить на `6.28`).

41. Распространение копий.

Распространение копий – процесс замены использования переменной x на y, если ранее встречалась операция копирования $\texttt{x} \leftarrow \texttt{y}$.

42. Остовное ребро.

Дуги (ребра) ГПУ, являющегося дугами и его остовного дерева называются *остновными*.

43. Прямое ребро.

Дуги (ребра) ГПУ, не являющееся дугами его остовного дерева, но имеющие такое же направление, что и остовные, называются *прямыми*.

44. Обратно направленное ребро.

Дуги (ребра) ГПУ, направленные противоположно остовным называются *обратно направленными*.

45. Обратное ребро.

Обратно направленная дуга (ребро) ГПУ $\langle B_i, B_k \rangle$ называется *обратной*.

46. Естественный цикл.

Естественный цикл (натуральный), цикл со следующими свойствами:

- Имеет *заголовок* – единственный входной узел.
- Существует обратное ребро, ведущее в заголовок цикла.

47. Инвариант цикла.

Инвариантом цикла называются:

- Константы.
- Переменные, все определения которых находятся вне цикла.
- Переменные цикла, но определенные в нем один раз И ее определение находится в ББ, являющемся доминатором всех выходов из цикла.

48. Инструкция, инвариантная относительно цикла.

Инструкция, инвариантная относительно цикла – такая инструкция, что все ее операнды являются инвариантами этого цикла.

49. Индуктивная переменная.

Индуктивная переменная – это переменная цикла, т.ч. на каждой его итерации она меняется на значение переменной (или константы), являющейся инвариантом этого цикла.

50. Семейство индуктивных переменных.

Семейство индуктивных переменных – это множество, в которое входит основная индуктивная переменная `i` и все ее производные индуктивные переменные.

Основной индуктивной переменной называется такая индуктивная переменная цикла `i`, у которой определение имеет вид $\texttt{i} \leftarrow \texttt{i} + \texttt{c}$, где `c` – инварианта цикла (необязательно константа).

Производными индуктивными переменными от основной индуктивной переменной `i` называют такие переменные `j`, которые зависят от значения `i` и, как привило (но не обязательно), являются линейными функциями от `i`: `j = c * i + d`, где `c` и `d` – инварианты цикла. Обозначение: `j = <i, c, d>`.

51. Частичная раскрутка циклов.

Частичная раскрутка цикла (по `m`) – это операция при которой компилятор копирует тело цикла (`m` раз) для нескольких (`m`) итерация и корректирует вычисление индексов соотв. образом.

Пример.

```
// Изначальный цикл
for (j = 1; j <= nj; j++) {
  for (i = 1; i <= ni; i++) {
    y[i] += x[j] * m[i][j];
```

```
  }
}
//=========================================//
// раскрутка (по m = 4):
for (j = 1; j <= nj; j++) {
  // вычисления для цикла-пролога
  mod_i = ni % 4;
  if (mod_i >= 1) {
    // цикл-пролог
    for (i = 1; i <= mod_i; i++) {
      y[i] += x[j] * m[i][j];
    }
  }
  // результат раскрутки
  for (i = mod_i + 1; i <= ni; i += 4) {
    y[i] += x[j] * m[i][j];
    y[i + 1] += x[j] * m[i + 1][j];
    y[i + 2] += x[j] * m[i + 2][j];
    y[i + 3] += x[j] * m[i + 3][j];
  }
}
```

Цикл-пролог – специальный цикл вставленный компилятором на тот случай, если `ni` не кратно `m` (`m = 4` в данном случае). Его назначение – гарантировать, что количество итераций цикла раскручиваемого по `m`, кратно `m`.

52. Гнездо циклов.

Гнездо циклов – набор циклов, расположенных таким образом, что один цикл является телом другого цикла. Проще говоря: вложенные циклы.

53. Выравнивание циклов.

Если не удается раскрутить цикл по `m` из-за того, что условие цикла `ni` не кратно `m`, то вводится *цикл-пролог*, который работает от начала до `ni mod m`, а раскрученный, выравненный цикл уже начинает с `ni mod m`, а не с начала.

54. Слияние циклов.

Слияние циклов – объединение двух циклов, у которых одинаковый шаг и условия на индекс, в один.

55. Бесполезный код.

Бесполезный код – те инструкции, которые не влияют на результат вычислений.

56. Недостижимый код.

Недостижимый код – инструкции, которые не содержатся ни в одном реально пути выполнения. Проще говоря: инструкции, которые ни при каких условиях не буду исполнены.

57. Форма статического единственного присваивания (SSA-форма).

Форма статического единственного присваивания (SSA – Static Single Assignment) – промежуточное представление, облегчающее некоторые оптимизации кода (такие как распространение констант). У всех переменных в SSA-форме разные имена.

SSA-форма позволяет в каждой точке программы объединить

- информацию об имени переменной
- информацию о текущем значении этой переменной
  (т.е. какое из определений данной переменной определяет ее текущее значение в данной точке в данный момент)

Хотелось бы, чтобы программа в SSA-форме удовлетворяла условиям:

1. каждое определение переменной имеет индивидуальное имя
2. каждое использование переменной достигало только одно определение

58. φ-функция.

φ-функция – такая функция, которая может выдавать значение переменной в зависимости от того из какого блока мы попали в текущий блок (то есть выдает SSA имя переменной в зависимости от пути). В общем случае φ-функция является n-местной ($n \geq 2$). При входе в базовый блок все его φ-функции выполняются одновременно и до любого другого операнда.

59. Максимальная SSA-форма.

Максимальной SSA-формой называется такая SSA-форма, что она содержит намного больше φ-функций, чем на самом деле необходимо: φ после каждой точки сбора.

60. Частично усеченная SSA-форма.

Частично усеченная SSA-форма – такая SSA-форма, построенная с помощью алгоритма, размещающего φ-функции только на границах доминирования. Содержит меньше φ-функций, чем максимальная SSA-форма (но не обязательно минимальное).

61. Критическое ребро.

Критическая дуга (ребро) – это такая дуга, которая выходит из вершины с несколькими потомками ($|\texttt{Succ}| \geq 1$), и входит в вершину с несколькими предками ($|\texttt{Pred}| \geq 1$). Критические дуги обычно исключают, разрывая их и вставляя в разрыв дополнительные вершины.

62. Суперблок.

Суперблок (или расширенный базовый блок) – это множество E базовых блоков $B_1, B_2, \ldots, B_n$, где только у блока B_1 может быть несколько предшественников, а каждый из блоков $B_i,\ i \in [2, n]$ имеет в единственного предшественника.

Блоки $B_i \in E$ формируют дерево с корнем B_1. В суперблока E может быть несколько выходов на блоки (или суперблоки), не входящие в состав E.

63. Область графа потока управления.

Областью графа потока называется его подграф $R = \langle N_R, E_R \rangle$ такой, что:

1. Существует узел $h \in N_R$, доминирующий над всеми узлами в R; этот узел называется *заголовком области* R;
2. Если из некоторого узла r_2 ГПУ можно достичь узла $r_1 \in R$, минуя заголовок h, то и $r_2 \in R$;
3. Множество E_R включает все ребра ГПУ между любыми узлами $r_1, r_2 \in R$, за исключением некоторых ребер, входящих в заголовок h;
4. Единственным входом в область является ее заголовок

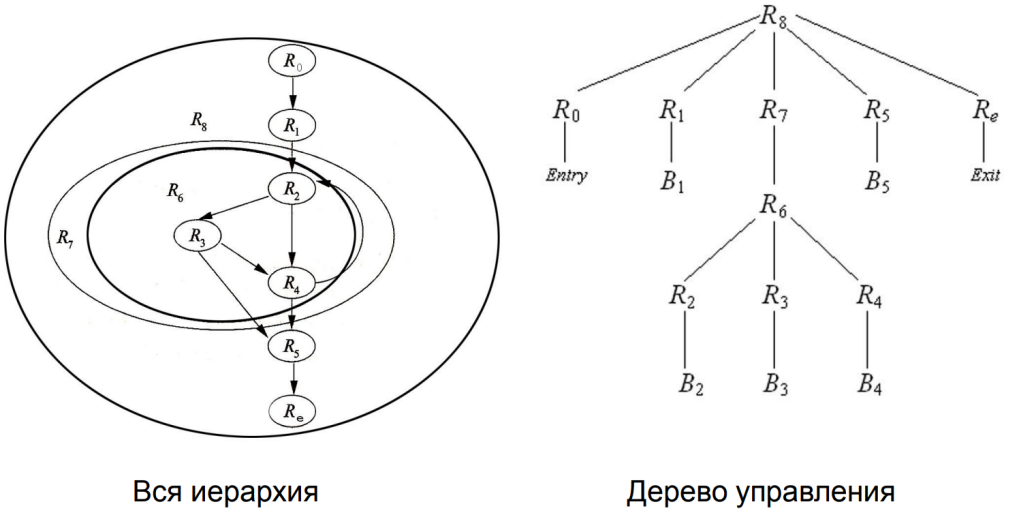
Область-лист – область вида $R = \langle \{B\}, \varnothing \rangle$, где B – произвольный ББ.

Область-тело – такая область, которая может заменить тело самого внутреннего цикла гнезда циклов (все узлы и ребра, за исключением обратных ребер к заголовку цикла).

Область-цикл – область-тело, где к телу соответствующего цикла идет обратное ребро к заголовку этого цикла.

64. Дерево управления.

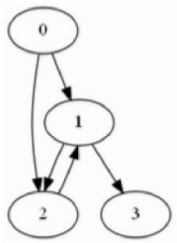
Дерево управления – иерархия всех областей в виде дерева.
Пример:



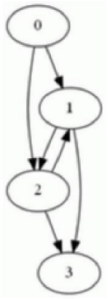
Вся иерархия Дерево управления

65. Сводимость ГПУ

Сводимость ГПУ – признак такого ГПУ, для которого можно построить дерево управления.
Пример несводимых графов:



Пример с GOTO



Классический пример

Построение областей и дерева управления позволяет ускорить обход графа потока управления во время выполнения различных алгоритмов анализа потока данных. В алгоритмах анализа потока данных, в которых в качестве сбора используется объединение, удается, заменив каждый цикл узлом типа область-цикл, оставить только ациклические пути для распространения атрибутов. Это позволяет сократить время выполнения соответствующего анализа потока данных.

66. Профилирование.

Профилирование – это численная оценка времени выполнения каждого фрагмента (базового блока, области) процедуры. В результате профилирования получается временной, или частотный, или еще какой-нибудь профиль процедуры.

Профиль – результат работы профилирования (временная, частотная или еще какая оценка).

67. Инструментирование и семплирование.

Инструментирование – в различные точки процедуры вставляются вызовы соответствующего инструмента (счетчика количества обращений к фрагменту, секундомера и т.п.), выбираются наборы исходных данных, обеспечивающие достаточно полное покрытие процедуры и исследуемая процедура запускается на этих наборах, в результате чего получается требуемый профиль.

Семплирование – процесс, в котором профилировщик с помощью механизма прерываний прерывает программу с некоторой периодичностью, анализирует ее состояние (трассы потоков) и на основе этого строит статическую модель (обновляет ее).

Программное семплирование – механизм прерываний реализован программно (виртуальная среда/машина, интерпретатор).
Аппаратное семплирование – механизм прерываний реализован на уровне ядра ОС.

Как правило, семплирование меньше замедляет программу, чем инструментирование, но дает менее точный профиль.

68. Межпроцедурный анализ.

Межпроцедурный анализ – анализ, позволяющий определить оптимизации для процедур.
Деление программы на процедуры имеет много достоинств, так как существенно упрощает составление и отладку программ. Но такое деление имеет и недостатки, усложняющие оптимизацию программ во время компиляции:

- Накладные расходы на вызов процедур (прологи, эпилоги и т.п.) существенно снижают эффективность (быстродействие) программы.
- Ограничение возможностей компилятора понимать, что происходит внутри вызова. Рассмотрим фрагмент процедуры

```
x = 15;
p = &x;
m = n + x;
q = f(p, m + n);
m = n + x;
```

Вопрос: можно ли исключить одно из присваиваний `m = n + x;`?

69. Граф вызовов.

Взвешенный граф вызовов (ВГВ) – граф, который задается пятеркой

$$G = \langle N,\ p_N,\ E,\ p_E,\ \texttt{main} \rangle$$

- N – множество вершин (представляющих процедуры программы), с каждой вершиной связан ее "вес".
- p_N – целое число, равное количеству инструкций соответствующей процедуры
- E – множество ребер (каждое ребро соединяет вызываемую и вызывающую процедуры), с каждым ребром связан его "вес".
- p_E – количество выполнений соответствующего вызова
- `main` – имя процедуры, выполняемой первой.

70. Контекст.

Контекст – это текущее состояние глобальных переменных и параметров программы из-за которых процедуры могут вести себя по разному.

71. Чувствительность к контексту.

Процедуру называют *чувствительной к контексту* если она работает по-разному с разными контекстами.

72. Функции скачка.

Функция скачка – такая функция, которая позволяет анализатору определить влияние константных значений при входе в процедуру, на множество константных значений в каждой точке вызова. (Короче хрень, которая позволяет анализатору на понять какие переменные можно вычислить на этапе компиляции.)

Для каждого фактического параметра строится своя функция скачка, так что точке вызова s с n параметрами соответствует вектор функций скачка $\mathcal{J}_s = (\mathcal{J}_s^{p_1}, \mathcal{J}_s^{p_2}, \ldots, \mathcal{J}_s^{p_n})$, где $p_i,\ i \in [1, n]$ –

фактические параметры вызова в точке s.

Каждая функция скачка $\mathcal{J}_s^{p_i}$ может зависеть не только от p_i, но и от некоторого подмножества других формальных параметров вызова. Это подмножество будем обозначать $Support(\mathcal{J}_s^{p_i})$. Требуется, чтобы функция возвращала значение $Undef$, если хотя бы один параметр из $Support(\mathcal{J}_s^{p_i})$ имеет значение $Undef$.

73. Машинно-ориентированная оптимизация.

- Учет регистровой структуры вычислительной аппаратуры (оптимальное распределение регистров, передача параметров в процедуры и функции через регистры).
- Удаление лишних команд.
- Оптимизация потока управления и удаление недостижимых участков программ.
- Снижение "стоимости" программы (есть «дорогие» и «дешевые» команды с точки зрения времени их выполнения процессором).
- Использование «машинных идиом» (например: `xor eax, eax` для обнуления регистра).
- Слияние, дробление и развертывание циклов, иногда требующееся из-за технических особенностей аппаратуры.
- Учет векторных и конвейерных свойств архитектуры.

74. Режимы адресации.

- *Режим прямой адресации* – адрес задается непосредственно значением `x`.
- *Режим индексированной адресации* – `a(r)`, здесь: `a` – переменная, `r` – регистр. Адрес `a(r)` вычисляется путем прибавления к l-значению `a` значения из регистра `r`. Этот режим адресации полезен для обращения к массивам: `a` – базовый адрес массива, `*r` – смещение.
- *Режим косвенной адресации* – `*r` ссылка на ячейку памяти, находящуюся по адресу `contents(r)` `* c(r)` ссылка на ячейку памяти, находящуюся по адресу `c + contents(r)`. Здесь `c` – константа, `contents` – операция разыменования.
- *Режим адресации с использованием констант*, для указания которых используется префикс `#`. Например, команда `LD r, #n` загружает в регистр `r` целое число `n`.

75. Выбор команд.

Выбор команд – это процесс отображение инструкций промежуточного представления в команды целевого процессора.

Пример для команды $\mathtt{x} \leftarrow +, \mathtt{y}, \mathtt{z}$, где память переменных статическая:

```
LD  R0, y       ; загрузить y в R0
LD  R1, z       ; загрузить z в R1
ADD R0, R0, R1  ; сложить R0 и R1
ST  x, R0       ; сохранить сумму в x
```

76. Переписывание дерева.

Правилом преобразования дерева называется инструкция вида

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

Здесь `replacement` – отдельный узел, `template` – шаблон (поддерево), `action` – действие (фрагмент генерируемого кода).

Схема трансляции дерева – множество правил преобразования дерева.

Свертка входного дерева – процесс генерации целевого кода путем последовательного применения правил преобразования.

Трансляция состоит из (возможно, пустой) последовательности машинных команд, которые генерируются действием (`action`), связанным с шаблоном (`template`).

Переписывание дерева – процесс генерации целевого кода сверткой входного дерева.

77. Что такое распределение регистров.

Распределение регистров – процесс отображения неограниченного множества имен (псевдорегистров) на конечное множество физических регистров целевой машины (NP-полная задача).

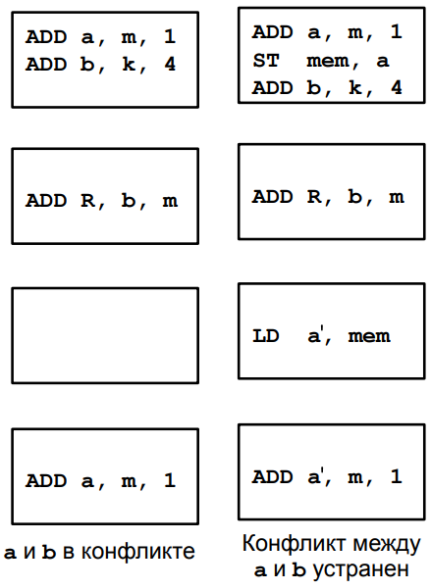
Назначение регистров выполняется после распределения регистров; оно отображает множество имен регистров на доступные регистры целевого процессора. Для решения этой задачи известно несколько алгоритмов полиномиальной сложности.

78. Интервалы жизни переменных.

Интервалом жизни переменной `v` называется множество команд программы, начиная с команды, в которой переменная `v` впервые определяется; и кончая последней командой, в которой переменная `v` используется.

79. Расщепление интервалов жизни.

Расщепление интервалов жизни – процесс, в котором при попытке одновременной записи в один регистр (*конфликт*), вычисления временно сохраняются в память.



- На левом рисунке $\texttt{LR}_a$ и $\texttt{LR}_b$ полностью пересекаются. Разделив $\texttt{LR}_a$ на два LR, удается устранить конфликт.
- При этом команды ST и LD применяются только здесь, а не перед всеми a.

80. Слияние интервалов жизни.

Слияние интервалов жизни – процесс, в котором команды копирования вида $\texttt{LR}_\texttt{i} = \texttt{LR}_\texttt{j}$ (где оба операнда не находятся в конфликте) удаляются, а все $\texttt{LR}_\texttt{j}$ заменяются на $\texttt{LR}_\texttt{i}$. В результате их ИЖ как бы сливаются.

81. Граф конфликтов.

Граф конфликтов – граф, у которого вершины, это интервалы жизни, а ребра соединяют те ИЖ, которые находятся в конфликте.

82. Слив (сброс) интервалов жизни.

Сбросом интервалов жизни называется процесс, когда регистров становится недостаточно и некоторые переменные перемещены (сброшены) в специальные ячейки оперативной памяти.

Операция сброса имеет свою стоимость, она складывается из трех компонент:

1. Стоимость вычисления адресов при сбросе.
2. Стоимость операции доступа к памяти.
3. Оценка частоты выполнения.

83. Планирование кода.

Планирование кода – процесс организации команд таким образом, чтобы не меняя семантики программы обеспечить близкое к оптимальному использование особенностей архитектуры целевого процессора. Прежде всего необходимо обеспечить использование возможностей параллельного выполнения команд.

84. Граф зависимостей по данным.

Граф зависимостей по данным – это такой граф $G = (N, E)$, который отражает все зависимости по данным, имеющиеся в блоке; и где

- N – множество узлов, представляющих операции в командах базового блока.
- E – множество ориентированных ребер, представляющих зависимости по данным между операциями.

Граф зависимостей по данным строиться так:

- Каждой операции $n \in N$ ставится в соответствие таблица резервирования ресурсов ($\texttt{RT}_n$), определяемая в соответствии с типом операции n.
  Каждое ребро $e \in E$ помечается задержкой d_e, указывающей, что целевой узел может быть выполнен не ранее, чем через d_e тактов после выполнения исходного узла.
- Пусть за операцией n_1 следует операция n_2, причем обе обращаются к одной и той же ячейке памяти с запаздыванием l_1 и l_2 соответственно, тогда в множество E нужно включить ребро $n_1 \to n_2$, помеченное задержкой $l_1 - l_2$.

85. Граф зависимостей программы.

Граф зависимостей программы – это граф, узлами которого являются инструкции присваивания программы, а ребра указывают зависимости данных между инструкциями и их направлениями. Ребро от инструкции s_1 к инструкции s_2 имеется $\Leftrightarrow$ имеется зависимость данных между некоторым динамическим экземпляром s_1 и более поздним динамическим экземпляром s_2.

86. Топологический порядок.

Топологическим порядком являет обход графа после его топологической сортировки.

Такой порядок гарантирует, что ББ не будет планироваться до тех пор, пока не будут спланированы все команды, от которых он зависит.

87. Топологическая сортировка.

Топологическая сортировка – нумерация вершин графа следующим образом:

1. Строится остовное дерево по этому графу.
2. Начинается проход с вершины `Entry` слева-направо вглубь.
3. Вершине присваивается номер (начиная с 1) при выходе из нее.
4. По окончанию нумерации, граф перенумеруется по правилу $n-i+1$, где n – количество вершин графа (без `Entry` и `Exit`), i – текущий номер вершины.

Такая сортировка позволяет, чтобы каждая вершина была обработана до тех вершин, на которые она указывает.

88. Программная конвейеризация.

Универсальные циклы – циклы, итерации которых совершенно независимы одна от другой. К таким циклам можно применить возможности параллельной обработки процессора.

Пример:

```
for (i = 0; i < n; i++)
  D[i] = A[i] * B[i] + c;
```

Программная конвейеризация – метод организации развертки универсального цикла и последовательного планирования команд таким образом, чтобы максимально задействовать параллельные возможности процессора, но при этом не раздуть код до слишком больших размеров, что это могло бы повлиять на общую производительность программы.

Программная конвейеризация предоставляет удобный способ достичь оптимального использования ресурсов одновременно с компактным кодом. Арифметические команды хорошо конвейеризуемы.

(СТАРЫЙ ТЕОРМИН)
Метод планирования кода, который позволяет в результате планирования процедур, содержащих универсальные циклы, получать достаточно компактный код, который будет как можно более полно использовать возможности компьютера по параллельному выполнению команд, или (для компьютеров классов VLIW и EPIC) операций (частей команд).

89. Ресурсы. Виды ресурсов.

Модель машины можно представить как $M = \langle R, T \rangle$, где:

- T – множество типов операций: загрузка, сохранение, арифметические операции и т.д.
- R – вектор $R = [r_1, r_2, \ldots]$ аппаратных ресурсов, где r_i – количество доступных единиц i-того ресурса.
  Типичные *виды ресурсов*: модули обращения к памяти (МОП), арифметически-логическое устройство (АЛУ), модули для работы с плавающей точкой.

90. Таблица резервирования ресурсов.

Таблица резервирования ресурсов (resource-reservation table) – это двумерная таблица, где для каждой машинной команды t моделируется ее использование ресурсов. Обозначение: RT_t.

Ширина этой таблицы равна количеству видов ресурсов машины M, а длина – продолжительность использования ресурсов в операции. Например $RT_t[i, j]$ указывает на количество единиц j-того ресурса, используемого операцией типа t спустя i тактов после начала ее выполнения. Для удобства $RT_t[i, j] = 0$ если i указывает на несуществующий элемент.

$$\forall t, i, j \quad RT_t[i, j] \leq R[j]$$

То есть $RT_t[i, j]$ не должно превышать количества ресурсов типа j в этой машине ($R[j]$).

91. Пространственная локальность данных.

Пространственная локальность данных – за небольшой промежуток времени выполняется обращение к данным, находящимся рядом друг с другом.

92. Временна́я локальность данных.

Временна́я локальность данных – использование некоторых данных несколько раз за короткий промежуток времени.

93. Для чего необходимо обеспечивать локальность данных?

Локальность данных *важна*, так как повышает производительность программы (меньше промахов в кешах).