**Document Title**

Subject

Author

# Выбор инструкций в компиляторах

Ниже мы уделим внимание финальной части компилятора, которая занимается выбором инструкций целевой архитектуры (англ. instruction selection). После этой фазы для получения исполняемого кода компилятору останется сделать только переупорядочивание инструкций (instruction scheduling) и распределение регистров (англ. register allocation).

Информация подготовлена в основном на основе работ [[Blindell, 2016](#_2bba4496a65363b84b3ee0cc6f056aad), [Blindell, 2018](#_5f2c06eb469ea7e58856a664dad1ab58)]. В тех местах, где это уместно, были приведены примеры использования RISC-V инстукций.

## Введение

Исполнитель, для которого собирается программа обычно называют **целевой машиной** (англ. target machine). Она состоит в первую очередь из процессора, который постоянно исполняет некоторый **машинный код**. Код состоит из **инструкций**, а множество доступных инструкций и их поведение задается с помощью архитектуры набора команд (англ. Instruction Set Architecture, ISA). Машинный код, приготовленный для конкретной ISA должен корректно работать на всех машинах и процессорах, которые поддерживают данную ISA. Конкретные инструкции бывают разнообразные, простые и сложные.

Теоретически, программировать можно и в машинных кодах. Но сейчас принято использовать **язык ассемблера**, который преобразуется в машинные коды с помощью программы, называемой **ассемблером**.

Обычно, ассемблерные инструкции являются более простыми, чем синтаксические конструкции языков высокого или низкого уровня. Но встречаются такие инструкции, которым соответствуют в языке программирования целые функции. Для процессоров, где такие «хитрые» инструкции поддерживаются, естественно использовать именно их. Если таких инструкций нет, то необходимо порождать несколько более простых инструкций. Вообще, задачей преобразования кода на языке программирования в машинные инструкции занимается **компилятор**.

### Компилятор

Обычно, в архитектуре компилятора можно выделить нескольких частей. Всё начинается с синтаксического анализа, где разбирается исходный текст программы, отлавливаются синтаксические ошибки, ошибки типизации и программа преобразовывается в некоторое промежуточное представление (англ. intermediate representation). За это отвечает часть компилятора именуемая frontend.

Затем (middle-end) представление программы проходит множество фаз оптимизации. Формально, эта часть не особенно нужна, чтобы получить работающую программу, но для получения эффективно работающей программы без неё не обойтись. Часть компилятора, связанная с оптимизациями, может запросто стать самой большой частью компилятора.

После оптимизаций представление программы передается в back-end, где происходит порождение кода, обычно в три этапа.

* **Выбор инструкций** (англ. instruction selection), которые будут исполняться в процессоре.
* **Переупорядочивание инструкций** (англ. instruction scheduling), чтобы задействовать все мощности параллелизма конвейера процессора.
* **Распределение регистров** (англ. register allocation) для использования встроенных в процессоров регистров и сокращения нагрузки на память.

### Выбор инструкций

Для некоторого куска P представления программы задачей выбора инструкций является подбор машинных инструкций таким образом, чтобы они демонстрировали поведение такое же как у P. Задача осложняется тем, чтобы зачастую существуют последовательности инструкций которые справляются с симулированием поведения P лучше, чем другие последовательности инструкций. Особенно это касается специализированный процессоров (например, DSP), где заготовлены специальные инструкции для конкретных задач из реального мира.

Семантику инструкций можно представлять с помощью так называемых шаблонов, а задачу выбора инструкций сводить к двум подзадачам.

* **Поиск шаблонов** (англ. pattern matching). Здесь мы ищем все последовательности-кандидаты в порождаемый код. Обычно доступные инструкции пересекаются по демонстрируемому поведению, из-за чего кандидатов может получаться много.
* **Выбор шаблонов** (англ. pattern selection) заключается в непосредственном выборе из кандидатов.

Обычно вторая задача формулируется как задача оптимизации, где мы пытаемся либо минимизировать размер кода, либо минимизировать суммарное время исполнения инструкций, чтобы максимизировать производительность программы.

### Сравнение разных методов выбора инструкций

В ISA описаны множества инструкций и они разные по сложности. В самых первых процессорах простыми считались инструкции, работающие с регистрами, а сложными — работающие с памятью. В литературе упоминаются различные схемы адресации, которые позволяют сокращать размер кода и улучшать производительность. Например, предположим, что нам надо из массива байт загрузить некоторый элемент. По сути у нас есть *базовый адрес* начала массива и некоторое *смещение*, и нам нужно сложить эти два адреса и загрузить из памяти по адресу суммы. Для RISC-V мы это должны сделать буквально, а AMD64 имеет специальные инструкции для *индексированной адресации*. Поддержка такого вида адресации в современных компиляторах давно реализована, поэтому в современной литературе сложными считаются инструкции, у которых много результатов, или те, которые можно использовать только в определенных ситуациях.

### Классификация инструкций

Чтобы проще сравнивать различные подходы к выбору инструкций, давайте введем классы инструкций.

**С единичным результатом** (англ. single-output instructions). Такие инструкции производят только один наблюдаемый результат, который можно прочитать другими инструкциями в ассемблерном коде. Сюда относятся большинство инструкций в современных процессорах, например, сложение и умножения, загрузка из памяти с учетом индекса выше, в том числе сложные инструкции типа cpop из RISCV, которая считает количество единиц в битовом представлении числа.

Обычно, из таких простых инструкций состоят ISA RISC процессоров, например, MIPS или RISC-V с базовым набором инструкций.

**С множественными результатами** (англ. multi-output instructions) имеют более одного наблюдаемого результата. Классическим примером будут инструкции, которые сразу вычисляют и остаток, и частное, или арифметические инструкции, выставляющие флаги переполнения. Большинство архитектур предоставляют такого рода инструкции, в том числе и AMD64, и RISC-V (например, [расширение atomic](https://msyksphinz-self.github.io/riscv-isadoc/html/rv64a.html)).

**С не пересекающимися результатами** (англ. disjoint-output instructions) порождают из набора входных данных набор выходных. От предыдущего вида они отличаются тем, что тут результат не зависит от всех входных данных, и входы и результаты сгруппированы в виде некоторых шаблонов, которые не пересекаются. Сюда относятся SIMD-инструкции (англ. single-instruction, multiple-data), которые запускают одновременно несколько однотипных действий над данными. Для AMD64 такие инструкции есть в расширения SSE и AVX, для ARM — в NEON, в RISC-V — векторные инструкции.

**Межблоковые** инструкции получаются из нескольких блоков графа потока управления высокоуровневого языка. Каночиным примером будет арифметика с насыщением, например max для из [RISC-V с расширением bitmanip](https://github.com/riscv/riscv-bitmanip/blob/main/bitmanip/insns/max.adoc).

**Зависимые между собой** инструкции обычно встречаются в специализированных архитектурах типа DSP. Зависимости заключаются в том, что некоторые инструкции не могут стоять рядом с другими в зависимости от используемого способа адресации. Современным методам такие инструкции даются тяжело, обычно потому что они нарушают некоторые предположения, которые вшиты в методы выбора инструкций.

### Что такое порождение «оптимальных» инструкций?

Говоря про «оптимальный выбор инструкций» часто подразумевают следующее определение. Для некоторого набора I инструкций, где каждая инструкция имеет стоимость , алгоритм выбора инструкций дает оптимальный результат, если для любой входной программы P он находит набор (с повторами) S из I такой, что S реализует P, и не существует другого такого набора , что он тоже реализует программу P, и при этом .

У этого определения есть несколько недостатков. Во-первых, алгоритмы выбора инструкций могут иметь различные наборы поддерживаемых инструкций, игнорируя некоторые, которые в перспективе могут дать более эффективный код. Алгоритмы, использующие похожий набор инструкций, можно сравнивать. Алгоритмы с непохожими наборами инструкций могут быть каждый по-своему оптимальными, но при это различаться в разы по производительности. Поддерживать все возможные инструкции ISA тоже затруднительно, так как алгоритмы порождения инструкций становятся полуоптимальны.

Во-вторых, два сравнимых подхода к выбору инструкций могут породить код, который после фаз переупорядочивания инструкций и распределения регистров будет непохожего качества. Например, нам нужно породить инструкции, которые независимы друг от друга. ISA предлагает два варианта: использовать две инструкции со стоимостью 2 каждая, либо использовать одну инструкцию со стоимостью 3. Согласно критерию выше, нужно выбирать второй подход, так как там суммарная стоимость будет меньше. Но если целевая архитектура умеет исполнять несколько инструкций параллельно, то лучше первый подход.

## Раскрытие макросов (Macro Expansion)

Раскрытие макросов исторически является первым и достаточно простым подходом к порождению инструкций. Зачастую реализация разделяется на две части: непосредственно макросы-шаблоны и процедура, которая применяет эти макросы к коду (macro expander). За счет этого разделения первая часть может быть специализирована под различные архитектуры, в то время как вторая может быть написана один раз для всех архитектур.

**Преимущества**: просто и прямолинейно.

Пример раскрытия макросов для архитектуры RISC-V. Одной инструкции языка Си слева соответствуют от 1 до 3 инструкций ассемблера.

|  |  |
| --- | --- |
| int a = 1; | li r1, 1 |
| int b = a+4; | addi r2, r1, 4 |
| p[4] = b; | lw r3, @p ; адрес начала массива addi r4, r3, 4\*8 sw r4, r5 |

### Наивное раскрытие макросов

Одной из первых работ по порождению кода с помощью макросов является SIMCMP (SIMple CoMPiler) [[Orgass and Waite, 1969](#_48d36f13e273e0746062b731405ed69b)]. В этом проекте код программы читался строчка за строчкой, и на ходу порождался машинный код. Сделано это для того, чтобы писать компилятор языка на самом этом языке (англ. bootstraping).

Ниже можно найти пример спецификации в системе SIMCMP [[Orgass and Waite, 1969](#_48d36f13e273e0746062b731405ed69b)].

Объявление макроса в SIMCMP.

\* = CAR.\*.  
 I = CDR('21)  
 CDR('11) = CAR(I).  
.X

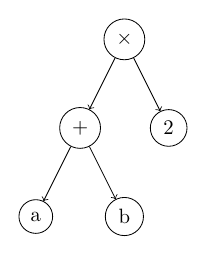
Строка программы, которую компилируем.

A = CAR B.

Порожденный код

I = CDR(38)  
CDR(36) = CAR(I)

Другой пример — GCL [[Elson and Rake, 1970](#_b9b92e9ac3092da0c49c94acc4a4c019)], который использовался в компиляторе PL/1 и код порождался из деревьев абстрактного синтаксиса (англ. abstract syntax tree, AST). По сравнению с чтением программы построчно, AST гарантирует, что программа написана без синтаксических ошибок, что упрощает задачу порождения кода.



Дерево выражений

Пример кода на RISCV для простого выражения и его схема компиляции для RISC-V. Значения переменных a и b хранятся в регистрах r1 и r2 соответственно.

add t0, r1, r2  
mulw t0, t0, 2

### Промежуточные представления вместо деревьев абстрактного синтаксиса

Первые компиляторы занимались порождением кода непосредственно на основе команд на языке программирования. Это прямолинейный подход, который не может анализировать исходную программу в целом, а только по отдельным инструкциям. К тому же оно привязывает порождение кода (т.е. компилятор) к конкретному языку программирования.

Более удачным вариантом является порождение кода из деревьев абстрактного синтаксиса. В наши дни из AST порождается из специальное представления программ, в которых совершаются различные оптимизации. Примерами таких представлений могут быть ANF, SSA и C--.

Одно из первых промежуточных представлений было разработано [[Wilcox, 1971](#_152ba26c3f82b8ab85518f91910152cc)] для компилятора PL/C, где AST преобразовывалось в SLM-инструкции (англ. source level machine). Порождатель кода отображает SLM-инструкции в машинные, используя правила на языке ICL (Interpretative Codeing Language). На практике оказалось, что такие правила очень сложно писать, потому что много тонкостей (разные виды адресации, местоположения данных) надо поддерживать вручную.

Макрос для сложения чисел на языке ICL [[Wilcox, 1971](#_152ba26c3f82b8ab85518f91910152cc)]

ADDB BR A,ADDB1 Если A в регистре, переход на ADDB1  
 BR B,ADDB2 Если B в регистре, переход на ADDB2  
 LGPR A Породить код, загружающий A в регистр  
  
ADDB1 BR B,ADDB3 Если B в регистре, переход на ADDB3  
 GRX A,A,B Породить A+B  
 B ADDB4 Слияние  
  
ADDB3 GRR AR,A,B Породить A+B  
ADDB4 FREE B Освободить ресурсы, связанные с B  
ADDB5 POP 1 Удалить дескриптор для B со стэка  
 EXIT  
  
ADDB2 GRI A,B,A Породить A+B  
 FREE A Освободить ресурсы, связанные с A  
 SET A,B Удалить дескриптор для A со стэка  
 B ADDB5 Слияние

### Порождение макросов из описания целевой машины

Реалистичные компиляторы с какого-то момента времени должны начать поддерживать несколько целевых машин. Проблемы с рукописными макросами начинаются, если машины начинают существенно различаться между собой. Например, бывают разные классы регистров (TODO ссылка), в которые можно класть только данные определенного вида, или которые нельзя использовать одновременно, или некоторые архитектуры могут не иметь подходящих команд, и для выполнения операции над данными из DRAM необходимо задействовать дополнительный регистр.

Доступ к данным по указателю на стеке для RISC-V64 и AMD64

|  |
| --- |
| Код на Си  x = \*a; |
| AMD64  *; AMD64* mov 8(%rsp), %rax  *; RISCV64* addi t0, sp, 8 lw a0, t0 |

В примере выше мы обращаемся к элементу на расстоянии 8 байт от вершины стека. В архитектуре AMD64 мы можем сделать это непосредственно, в RISCV64 необходимо пользоваться промежуточным регистром. При генерации кода с помощью макросов приходится одновременно заниматься и распределением регистров, что усложняется задачу порождения оптимального кода.

Писать макросы руками сложно, хотелось бы иметь генератор, который по описанию машины порождает соответствующие макросы. Одна из первых попыток [[Miller, 1971](#_53c486ae432af5ce6508b73e0c2ad455)] сделать это была система Dmacs. Она предлагала два проприетарных языка: первый (Machine-Independent Macro Language (MIML)) определят 2-адресные команды, которые являлись представлением программы, а второй (Object Machine Macro Language (OMML)) декларативный язык использовался, чтобы преобразовывать MIML команды в ассемблерный код.

Представление арифметического выражения A[I] = B + C[J] \* D с помощью команд MIML. Команда SS используется, чтобы переслать данные между разными источниками. На аргументы ссылаются либо по имени, либо по номеру строки, где он использовался.

1: SS C,J  
2: IMUL 1,D  
3: IADD 2,B  
4: SS A,I  
5: ASSG 4,3

Часть описания компьютера IBM-360 на языке OMML [[Miller, 1971](#_53c486ae432af5ce6508b73e0c2ad455)]. Команда rclass описывает виды регистров, а rpath — разрешенные способы пересылки между видами регистров и памятью.

rclass REG: r2, r3, r4, r5, r6  
rclass FREG: fr0, fr2, fr4, fr6  
...  
rpath WORD -> REG: L REG,WORD  
rpath REG -> WORD: ST REG,WORD  
rpath FREG -> WORD: LE FREG,WORD  
rpath WORD -> FREG: STE FREG,WORD  
...  
ISUB s1 ,s2  
from REG(s1),REG(s2) emit SR s1 ,s2  
from REG(s1),WORD(s2) emit S s1 ,s2  
resultresultREG(s1)  
REG(s2)  
FMUL m1, m2 (commutative)  
from FREG(m1),FREG(m2) emit MER m1 ,m2  
from FREG(m1),WORD(m2) emit ME m1 ,m2  
resultresultFREG(m1)  
FREG(m1)

### Использование peephole-оптимизаций

Основным недостатком подхода на основе раскрытия макросов является то, что отдельные части IR раскрываются без учета рядом находящихся частей IR. Попытаться обойти этот недостаток можно с помощью peephole (в перевода на русский — «глазок») оптимизаций. Их суть заключается в том, что выбирается «окно» небольшого размера, которое двигают по порожденному коду и пытаются объединить видимые инструкции. Данный метод может применяться и в отрыве от выбора инструкций, к уже порожденному коду. Одним из самых известных применений являются «супер оптимизаторы» [[Massalin, 1987](#_be857b05265dd381628994ac1f500246)], например Souper [[Sasnauskas *et al.*, 2018](#_d08d610dd0281f85eec2b9439be016e1)]. Идея подхода заключается кодировании семантики текущего набора инструкций в представление, понятное SMT-решателям, и затем нахождение минимальной программы с такой же семантикой с помощью синтеза программ (англ. Counter Example Guided Inductive Synthesis, CEGIS). К сожалению, Souper поддерживает набор инструкций размером только в несколько десятков, и масштабирование этого подхода на разнообразные архитектуры является предметом дальнейших исследований.

Оптимизации методом peephole можно использовать [[Davidson and Fraser, 1984](#_6191d3127ad90424f6bfc16f329e1fda)] и в контексте выбора инструкций, такой подход используется в компиляторе GCC [[Stallman, 1988](#_8f4493640f5210c5778f98a5dbe1021c)]. Суть подхода заключается в том, что раскрытие макросов порождает не код целевой машины, а некоторое описание на языке RTL (англ. Register Transfer List). В примере ниже трехадресная инструкций сложения складывает константу imm с регистром и сохраняет результат в , выставляя флаг нуля .

В предлагаемом подходе представление программы с помощью правил RTL превращается в описание «эффекта» этой программы. В отличие других подходов на основе макросов распределения регистров не происходит. Все используемые регистры — виртуальные, предполагается, что их бесконечно много. После раскрытия макросов и до распределения регистров запускается так называемый комбинатор (англ. combiner), который пытается объединить несколько RTL описаний в большее RTL-описание, соответствующее какой-то инструкции целевой архитектуры. Чтобы такой подход работал, надо поддерживать инвариант, что все RTL-описания выразимы с помощью одной инструкции целевой архитектуры.

Теоретически, такой подход позволяет порождать код, рассматривая не одну команду языка программирования, а сразу несколько, даже лежащих в разных блоках потока управления. Сложность порожденных инструкций сильно зависит от размера «окна» оптимизатора, так, например, не получится породить инструкции, соответствующие трём RTL, если мы смотрим только на два RTL.

## Покрытие деревьев

Одним из основных ограничений раскрытия макросов является то, что в нём порождается код, рассматривая только одну инструкцию или только один узел промежуточного представления. Из-за этого порождается код плохого качества. Другой сложностью является то, что поиск кандидатов в порожденный код и выбор наилучшего осуществляется за один шаг, что делает задачу исследования разных комбинаций инструкций затруднительной. Эти недостатки решает порождение кода с помощью деревьев.

Суть идеи заключается в том, что нам дано некоторое дерево, которое представляется собой программы, а также некоторый шаблон древовидных шаблонов (англ. pattern). Задача порождения кода сводится к задаче покрытия нашего дерева подмножеством этих шаблонов оптимальным образом, т.е. задача разбивается на поиск всех возможных покрытий и выбор оптимального покрытия шаблонами-образцами. Для большинства архитектур шаблоны будут пересекаться, и поэтому различных покрытий будет много. Обычно, мы будет стараться воспользоваться минимальным количеством шаблонов:

* Предпочитая крупные шаблоны мы будет использовать специализированные инструкции, которые, как правило, исполняются быстрее.
* С меньшим количеством шаблонов они будут меньше пересекаться, а значит меньше данных будет пересчитываться заново, что приведет к улучшению производительности и размера кода.

В общем случае, оптимальное решение сводится к минимизации не количества использованных шаблонов, а к снижению суммарной *стоимости* этих шаблонов, хотя существует сильная корреляция между количеством шаблонов и их суммарной стоимостью. Также стоит отметить, что выбор оптимальных шаблонов не всегда приводит к оптимальному коду (в том числе из-за участия других фаз компиляции). Но постановка задачи выбора оптимального покрытия шаблонами, гораздо менее спорна, чем задача порождения эффективного кода, так как мы всегда выбираем из фиксированного набора шаблонов, порожденных из ISA.

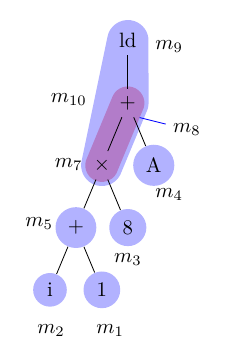
Пример кода на Си

x = A[i + 1];

Пример: простое выражение, которое загружает по индексу i+1 из массива чисел A. Предполагается, что индекс i находится в регистре, A — в памяти, а числе 8байтные. Всего три полных покрытия дерева шаблонами: , и ,

Инструкции-шаблоны, построенные на основе ISA. Астериск обозначает взятие из памяти по адресу.

mv r <- var  
add r <- s + t  
mul r <- s × t  
muladd r <- s × t + u  
load r <- ∗s  
maload r <- ∗(s × t + u)



Дерево выражений и его покрытие шаблонами

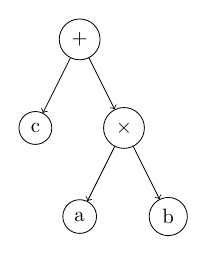
### Использование синтаксического анализа

В попытке преодолеть «наколеночность» методов с раскрытием макросов, были предложены подходы к выбору инструкций с использованием формализмов. Одним из них может быть использование формальных грамматик и подходов на основе синтаксического анализа языков. Было предложено [[Glanville and Graham, 1978](#_b1ba65d00a199e8ea0e6a895fde8871c)] описывать промежуточное представление программы с помощью контекстно-свободных грамматик, где правила аргументирована стоимостью операций и некоторым действием (англ. action code), которое будет заниматься непосредственно порождением кода.

Грамматика для порождения кода для арифметических выражений

|  |  |  |
| --- | --- | --- |
| Инструкция | Стоимость | Действие |
|
| r1 <- r1 + r2 | 1 | emit add r1,r1,r2 |
|
| r1 <- r1 × r2 | 1 | emit mul r1,r1,r2 |
|
| r3 <- Int | 1 | emit li r1, I |
|

В грамматике используются так называемые терминальные символы (в нашем примере названия арифметических действий и числа), и нетерминальные символы (названия регистров-локаций)



Пример выражения, для которого будем порождать инструкции с помощью синтаксического анализа

Во время анализа на стеке накапливаются текущие терминалы и нетерминалы. После получения нового терминала и добавления его на стек, анализатор может сделать два действия:

* shift — продолжить чтение терминалов и оставив стек без изменений;
* reduce — выбрать правило грамматики, снять с вершины стека нетерминалы из правой части правила, и заменить на левую часть правила; вместе с этим сгенерировать некоторый код на ассемблере.

Таким образом для входа , где — целые числа, мы породим примерно такой код, совершив следующие действия: , где — shift, а — reduce по правилу N.

li R1, a  
li R2, b  
mul R1, R1, R2  
li R3, c  
add R1, R1, R3

В правилах у регистров есть индексы, которые позволяют выражать случаи, когда вход и выход инструкций приходятся на один и тот же регистр.

Основной сложностью такого вида синтаксического анализа, является то, что не всегда очевидно, когда предпочитать shift, а когда reduce. Обычно это решается переписыванием грамматики так, чтобы конфликтные случаи не случались. Но для больших грамматик делать это вручную затруднительно. В изначальном подходе конфликт между shift и reduce всегда разрешался в пользу shift, а если на стеке получалось слишком много терминалов, то применялись ad hoc правила, чтобы сгенерировать код как-нибудь и исправить (почти) аварийное состояние. В случае reduce/reduce конфликта, выглядит разумным пытаться применить самое длинное правило. (Случаи, когда два правила одинаковой длины конфликтуют, можно задетектировать до запуска синтаксического анализа.)

**Преимущества**. В процессе синтаксического анализ снизу вверх строится таблица состояний с переходами, которая позволяет вести анализ за время пропорциональное размеру входа. Также такой вид синтаксического анализа выступает в роли формальной теории, чтобы, например, обосновывать полноту грамматики инструкций

**Недостатки**. Во-первых, из-за использования грамматик в момент синтаксического анализа мы не имеем доступа к конкретным значениям, например, констант. Из-за этого невозможно выразить какие-то ограничения на диапазоны констант и т.п. Так же, если инструкции имеют много видов адресации операндов (эта проблема должна обойти RISC-V стороной), то появляется много похожих правил, специализированных под местонахождение операндов. Так для CISC архитектуры VAX, грамматика разрослась до миллионов правил [[Graham *et al.*, 1982](#_5e0e3f7324b4e3501971ad3226ef505b)]. Методы рефакторинга и упрощения грамматик известны, но их в данном случае надо применять с осторожностью, чтобы не повредить качеству порождаемого кода.

В контексте RISC-V можно привести такой пример. Существуют расширения, которые позволяют сделать [сложение-со-сдвигом](https://github.com/riscv/riscv-bitmanip/blob/main/bitmanip/insns/sh3add.adoc), c помощью них можно реализовать умножение на некоторые константы. Например, можно mul r0, r1, 9 заменить на sh3add r0, r1, r1, за счет соотношения r\*9 = r + r lsl 3.

Во-вторых, такой вид синтаксического анализа порождает код для левого операнда, а потом для правого, не откатываясь назад. Таким образом, код левого операнда не знает о содержимом правого операнда, что может привести к плохому порожденному коду.

### Порождение кода путём анализа сверху вниз

Анализ сверху вниз вначале выбирает правило порождения кода, а уже потом проталкивает вниз все необходимые ограничения для операндов паттерна. Таким образом можно выражать, например, ограничения на константы, которые учавсвуют в операндах. При выборе правила можно не угадать, что приведет к невозможности породить код для операндов. В этих случаях процесс возвращается назад (англ. backtracking) и пробует применить другое правило. К сожалению большое количество возвратов назад, негативно влияет на производительности, из-за чего и первые испытания такого подхода [[Newcomer, 1975](#_ffe5d469e319e16090e0e08847243471)], и последующие [[Nymeyer *et al.*, 1995](#_cbee765de4274301e0c35b4655ca2365)] не сыскали широкого распространения.

Отличительной чертой подходов сверху вниз является сопоставление представления программы с шаблонами с учетом некоторых аксиом (например, not (E1<=E2) заменяется на E1>E2, E+0 на E, и т.п.), чтобы получать более эффективный результат.

### Отделение сопоставления с образцами-шаблонами и порождения кода

В предыдущих подходах выбор шаблонов и порождение кода делались вместе, что позволяет порождать код за один проход и получать более быстрый компилятор. Но при этом, при порождении кода сложно учесть влияние разных комбинаций шаблонов. Поэтому можно исследовать идею разделения фаз покрытия дерева образцами-шаблонами и порождение кода по этим шаблонам.

В литературе также встречаются исследования по оптимизации поиска подходящих шаблонов для дерева. Они заключаются в сведении задачи сопоставления с образцом к задаче поиска подстроки в строке [[Aho and Corasick, 1975](#_a1b93cf82a14cb82ab3e59c33142ddf8)], также построение таблиц для сопоставления с образцом, и последующее сжатие их. Основным достижением этих подходов является поиск всех возможных корректных сочетаний шаблонов за линейное время от размера программы. В данном документе они не освещены.

### Динамическое программирование

С появлением возможности получения всех подходящих сочетаний шаблонов за линейное время, начали появляться идеи выполнения выбора инструкций также за линейное время. Первые идеи [[Ripken, 1977](#_fa74e00839bb85cdd537a7e4d7db13a4)] использования динамического программирования позже привели к появлению генератора компиляторов Twig [[Aho *et al.*, 1989](#_c9e5362677843cac3458a71a5be467cf)], которые принимал на вход описание архитектуры на языке CGL (Code Generator Language) и дерево компилируемой программы, и порождал код за три прохода.

* Проход сверху вниз, который находил для каждого узла дерева множества подходящих шаблонов.
* Снизу вверх вычислялась стоимость выбора соответствующего шаблона для каждого узла.
* Последний проход сверху вниз выбирал покрытие наименьшей стоимости, и по дороге порождал код.

Такой подход имеет преимущества, по сравнению с подходом на основе синтаксического LR анализа. Основным является то, что конфликты теперь сами разрешаются путём вычисления стоимости применения конкретных шаблонов. Также описания шаблонов для архитектур становятся существенно короче.

К сожалению, подход динамического программирования предполагает, что задача может быть разбита на подзадачи, которые могут быть решены оптимально по-отдельности, и потом скомбинированы. На практике, задача порождения кода не обладает такими свойствами.

### Ограничения покрытия деревьев

Основным недостатком работы с деревьями выражений является то, что одинаковые подвыражения должны быть разделены по рёбрам и продублированы при построении дерева. Такие преобразования известны в литературе как edge splitting и node duplication. В зависимости от набора инструкций, не разделяя подвыражения можно добивать лучшего качества кода.

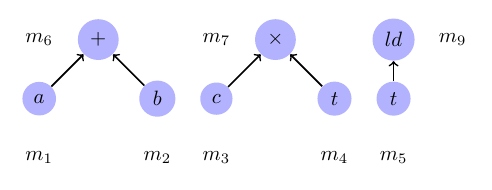
В примере ниже общее выражение для вычисления значения t было разделено, что приводит к покрытию m1,…m7,m9 со стоимостью 0+…+0+2+3+5=10. Если представить дерево как граф без циклов, то его можно покрывать шаблонами m8 и m10, что даст стоимость 0+…+0+4+5=9.

Пример. Инструкции и их стоимость. Нотация \*s означает получения данных по адресу в памяти.

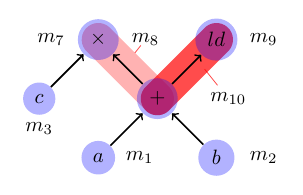
|  |  |
| --- | --- |
| Инструкция | Стоимость |
|
| add r <- s + t | 2 |
|
| mul r <- s × t | 3 |
|
| addmul r <- (s + t) × u | 4 |
|
| load r <- \* s | 5 |
|
| addload r <- \* (s + t) | 5 |
|

Пример кода на Си для которого будет порождать код с помощью деревьев не вполне эффективно

t = a + b;  
x = c \* t;  
y = \*(( int \*) t);



Деревья выражений после совершения деления рёбер (англ. edge splitting).



Представление программы в виде графа без циклов (вместо деревьев).

Также деревья ограничивают разнообразие поддерживаемых инструкций процессора. Так как у дереве всегда один корень, то инструкции с большим количеством выходов (англ. multi-output instructions ) не представимы, так как требуют больше одного корня. Даже инструкции с не пересекающимися выходами непредствимы, так как алгоритм выбора инструкций рассматривает деревья по одному.

В-третьих, представление с помощью деревьев не может моделировать граф потока управления. Цикл for требует циклический путь в графе, что не ложится в деревья. По этой причине представление с помощью деревьев годится только для выбора инструкций внутри базового блока (англ. basic block) графа потока управления. Это не позволяет выбирать инструкции процессора, которые соответствуют коду сразу в нескольких базовых блоках, что может негативно влиять на производительность.

В итоге, представление с помощью деревьев позволяет получить более качественный код по сравнению с наивным раскрытием макросов, но для современных архитектур инструкций нужны более сложные представления.

## Покрытие ациклических графов (DAGов)

Если ослабить ограничение, что у одного узла дерева — максимум один родитель, то вместо деревьев мы получим представление с помощью направленных ациклических графов (англ. Directed Acyclic Graph). За счет наличия нескольких родителей можно представлять значения, которые являются аргументами нескольких других выражений одновременно. Шаблонам теперь также разрешено иметь несколько корней, что позволяет осуществить поддержку инструкций со множественными результатами.

Так как DAGи менее ограничительны чем деревья, то для них можно применять новые подходы для порождения кода. Основных два

* Разделить DAG на деревья, породить код и объединить получившиеся результаты.
* Сопоставлять с образцам непосредственно граф, используя алгоритмы изоморфизма графов. В общем случае алгоритмы экспоненциально сложны, но зачастую они работают за линейное время.

### Сложность

Задача оптимального порождения кода по представлению в форме DAG NP-полна [[Koes and Goldstein, 2008](#_da128a2e3233fc29135a8c0a6c5be3e0)]. Доказать это можно сведя (за полиномиальное время) задачу SAT к задаче выбора шаблона в DAG .

### Жадные подходы

Порождение кода на основе DAG применяется в компиляторе LLVM, но исследование деталей затруднено тем, что основная документация — исходный код. Согласно [[Bendersky, n.d.](#_37fdf172628f4c87ba7c5648f6025c34)], порождение кода состоит из последовательного переписываться DAG, где инструкции промежуточного представления заменяются на машинные инструкции.

В LLVM шаблоны-деревья записываются на специальном языке, который компилируются в специальный предметно-ориентированные языки (bytecode), который осуществляет анализ представления программы. Все шаблоны перед компиляцией сортируются:

* по убыванию сложности, на которую влияет размер паттерна, и специальные константы, которые эвристически дают приоритет некоторым паттернам;
* по возрастанию стоимости порожденного кода;
* по возрастанию размера подграфа, который покрывается шаблоном.

Так как в шаблонах участвуют только деревья, то инструкции с множественными выходами генерировать не получится, для них нужен отдельный ad hoc алгоритм порождения. Также, за счет особенностей жадных алгоритмов, они не могут претендовать на оптимальность.

Также в LLVM присутствуют два других подхода к выбору инструкций: FastISel и GlobalISel, который позволяет порождать также и межблоковые инструкции.

### Методы выбора инструкций для DAGов

Методы можно условно разделить на те, которые адаптируют подходы для деревьев, и на все остальные. Можно придумывать эвристики, которые преобразуют граф без циклов в деревья так, чтобы затраты на копирование узлов были незначительны или отсутствовали. Также можно адаптировать методы на основе динамического программирования.

Также существуют методы, специфические для задачи оптимального выбора инструкций для графов без циклов. Они могут быть основаны на сведение задачи выбора к задаче оптимизации какой-либо предметной области. Были попытки сведения к задаче линейного программирования, MWIS (англ. maximum weighted independent set) проблемам, а также задаче программирования в ограничениях (англ. constraint programming), и др. В [[Beg, 2013](#_4e7b72996a9e35d7b46432d0172bb6c0)] исследовалось введение *глобальных ограничений* для решения задачи оптимального порождения кода с помощью программирования в ограничениях, и пришли к выводу, что для простых архитектур (MIPS и ARM) оптимальные решения примерно так же эффективны как и полуоптимальные на основе LLVM. Скорее всего для RISC-V можно ожидать таких же результатов.

### Ограничения покрытия DAGов

Графы без циклов являются обобщением деревьев. С помощью них можно непосредственно моделировать общие подвыражения, и большее разнообразие инструкций, а именно с множественными выходами и непересекающимися выходами, что существенно улучшает производительность и размеры кода. Подходы на основе покрытия DAGов сейчас наиболее распространенные.

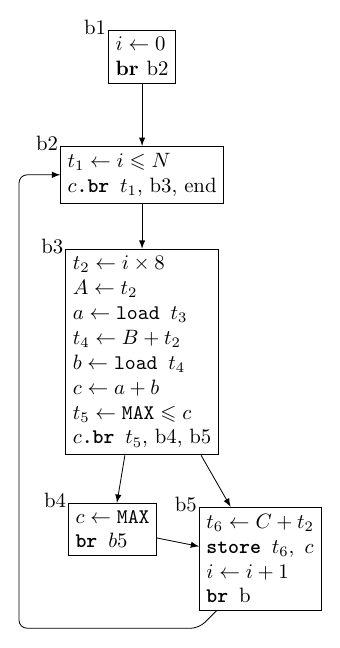
Цена этому заключается в том, что оптимальный результат больше не получить за линейное время, так как задача становится NP-полной. В то же время, DAGи недостаточно выразительны, чтобы промоделировать все аспекты программ. Например, циклы for не представимы как ациклические графы, что не позволяет моделировать инструкции, затрагивающие сразу несколько блоков графа потока управления программ.

## Покрытие графов

Некоторые конструкции языков программирования, например циклы, не ложатся в представление с помощью DAGов. Поэтому существует наиболее общая форма представления программ с помощью графов, где присутствует информация и о данных, и о потоке управления программы. Порождение инструкций для таких графов называется *глобальным порождением инструкций* (англ. global instruction selection), потому что учитывается информация не только в одном базовом блоке программы, а в нескольких блоках сразу. К тому же, появляются возможности передвигать инструкции из одного блока в другой (англ. global code motion), и выбирать межблоковые инструкции. Это делает графы наиболее мощным инструментом для порождения кода для архитектур, где много специализированных инструкций (например, различные DSP).

Пример кода на C, который складывает (с насыщением) массивы A и B, c получением массива C. Предполагается, что массивы равной длины, и размер элемента — 8 байт. Переменные N и MAX обозначают длину и верхнюю границу.

int i = 0;  
**while** (i < N) {  
 int a = A[i];  
 int b = B[i];  
 int c = a + b;  
 **if** (MAX < c)  
 c = MAX;  
 C[i] = c;  
 i++;  
}



Граф потока управления для вычисления насыщенной суммы двух массивов.

Пример выше посвящен использованию межблоковых инструкций, а именно операции взятия максимума двух чисел, доступной в том числе [для RISC-V](https://msyksphinz-self.github.io/riscv-isadoc/html/rvfd.html#fmax-d). Одна такая инструкция могла бы заменить сравнение с максимумом, ветвление и полностью убрать блок b4, что сократило бы размер кода с 16 до 13 инструкций (почти 25%).

В программе выше также присутствуют четыре сложения, которые не зависят друг от друга. Если начать передвигать инструкции между блоками, можно добиться применения векторных (или SIMD) инструкции, которая сделает четыре сложения одновременно. Если затраты на копирование и подготовку данных (англ. data copying ) для векторных инструкций незначительны, то автовекторизация ещё больше сократит затрачиваемые такты.

### Решение задачи поиска проверки через изоморфизм графов

Методы для DAG не масштабируются для графов, поэтому для графов нужны свои алгоритмы поиска подходящих шаблонов. Для выбора оптимального шаблона можно использовать алгоритмы, подходящие для DAGов. *Задача изоморфизма графов* проверяет, можно ли исходный грaф поворачивать, перекручивать или зеркально отображать так, чтобы в нём нашелся искомый подграф. Эта задача является обобщением поиска шаблонов для DAG при наличии разумных ограничений. Например, шаблоны для коммутативных операций (сложение или умножение) можно зеркально отображать, чтобы операнды поменялись местами, а для вычитания или деления — нет.

В литературе задача изоморфизма графов встречается в различных областях и известны методы её решения. Например, алгоритм Ульмана [[Ullmann, 1976](#_1f2a373b9a5f5f95343764898500d537)] имеет сложность в худшем случае , а алгоритм VF2 [[Cordella *et al.*, 2001](#_4e2b26a88bcdef13f703976041f5ced5)] — .

### Промежуточные представления на основе Sea-of-Nodes

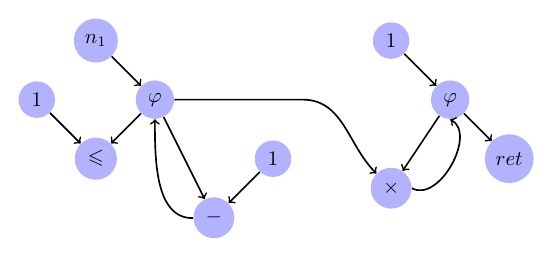
Функции, так как в них используется граф потока управления, мы вынуждены представлять с помощью графов. По соглашению, представления для них называются sea-of-nodes.

#### Static Single Assignment

Если каждая переменная присваивается только один раз, то можно говорить, что программа находится в SSA-форме [[Cytron *et al.*, 1991](#_a1f16c072c175c5e54b1c69027a2ee88)]. Проведение оптимизаций в такой форме более удобно, чем без неё. Например, в программе можно исследовать промежутки активности переменных (англ. live range), которые неформально обозначают места для в программе, где значения переменных нужны и их нельзя удалять. Для SSA формы эти промежутки непрерывны и по сути упрощаются до одного промежутка (за счет размножения количества переменных).

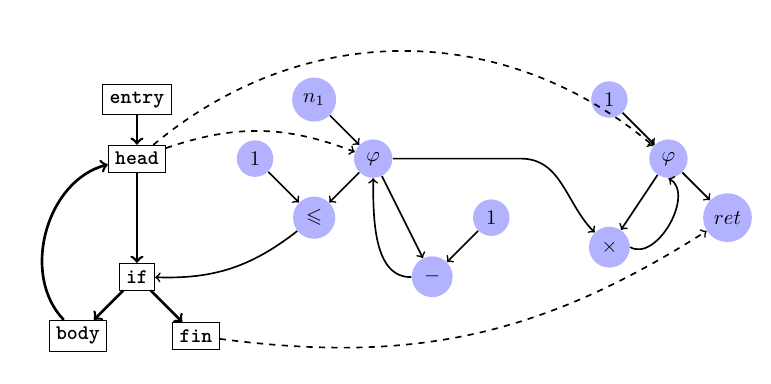
В примере ниже приведена реализация и SSA-форма факториала на языке Си. В ней используются так называемые φ-функции, которые присваивают значение переменной в зависимости от того, из какого блока к данной точке программы пришло исполнение. На основе SSA-представления функций можно строит SSA-графы [[Gerlek *et al.*, 1995](#_45672766386f8afe6c82d98348635253)], которые напоминают графы потока данных. Каждой операции соответствует узел графа, а рёбра обозначают поток данных, игнорируя факты того, что данные могут быть в разных базовых блоках графа потока управления. Такие SSA-графы не являются самостоятельными объектами в компиляторах, их используют вместе с графами потока управления для представления программ.

|  |
| --- |
| Реализация факториала на Си  int factorial (int n) {  **entry**:  int f = 1;  **head**:  **if** (n <= 1) **goto** end;  **body**:  f = f \* n;  n = n - 1;  **goto** head;  **end**:  **return** f; } |
| Код в SSA форме  int factorial (int n1 ) {  **entry**:  int f1 = 1;  **head**:  int f2 = φ(f1: entry, f3: body);  int n2 = φ(n1: entry, n3: body);  **if** (n2 <= 1) **goto** end;  **body**:  int f3 = f2 \* n2;  int n3 = n2 - 1;  **goto** head;  **end**:  **return** f2;  } |



Пример SSA-графа для факториала

Также существует представление [[Click and Paleczny, 1995](#_d0f781d397e6df89309ca59d7df75075)], объединяющее SSA граф и граф потока управления. Такое представление используется в *Java Hotspot Server Compiler (JHSC)*, где граф разбивается на, возможно, пересекающиеся деревья выражений. Корни деревьев выбираются так, чтобы они представляли собой общие подвыражения, или операции у которых есть побочный эффект, который не может быть раскопирован. А сами деревья выбираются так, чтобы попытаться их представить одной машинной инструкцией. Учитывая, что операции всё ещё представлены деревьями, инструкции с множественными результатами так породить не получится.



Пример графа Клика-Палечны, соответствующий факториалу. Тонкие линии объединяют операции над данными и поток данных. Толстые линии обозначают рёбра графа потока управления. Пунктирные линии обозначают принадлежность операций блокам.

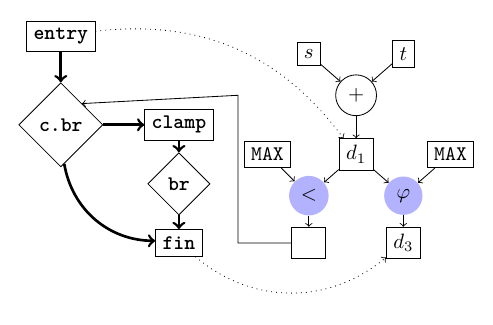
#### Универсальное представление

Одной из последних работ по выбору инструкций является подход [[Blindell, 2018](#_5f2c06eb469ea7e58856a664dad1ab58)] на основе *универсального порождения инструкций* (англ. Universal Instruction Selection). Оно является дальнейшим усложнением графов Клика-Палечны, что делает его достаточно полным, чтобы на нём проводить выбор инструкций. В частности, туда добавляются:

* Операции для явного изменения потока управления в графе потока управления.
* В граф потока данных добавляются узлы для конкретных значений (англ. value nodes), к уже имеющимся узлам для вычислений (англ. computation nodes)
* Операции над данными соединяются с блоками, где они происходят.
* Объявления новых переменных с помощью φ-функций также соединяются с блоками, где они происходят.
* Узлы для конкретных констант, используемых в операциях. Такие узлы раскопируются, если они используются в разных блоках, потому что присутствует ограничение, что одно и то же значение не может быть использовано одновременно в разных блоках.
* Так называемые *state nodes*, которые запрещают переставлять некоторые операции с неявными зависимостями, например, вызовы функций с побочными эффектами
* Номера ребер, чтобы упростить задачу поиска шаблонов в графе, так как при упорядоченных рёбрах она решается эффективнее.

По сравнению с графами Клика-Палечны, в универсальном представлении *все* операции надо потоком данных и управления представлены в виде узлов, что дает больше информации. Поиск шаблонов осуществляется с помощью изоморфизма графов. Если для конкретного шаблона находятся несколько подходящих подграфов, то это возможность использования инструкций с дизъюнктными результатами (SIMD или векторные), при условии, что подграфы не пересекаются и нет циклических зависимостей по данным. В предыдущих подходах такое было неосуществимо.

Данный подход был [реализован](https://github.com/unison-code/uni-instr-sel_), как дополнение к LLVM 3.8, и протестирован на DSP процессорах Hexagon. К сожалению, дело не дошло до реальной практической апробации, по видимому, вместо процессора используется его эмулятор, а оценка качества кода дается только статическим вычислением стоимости. Апробация подхода для RISC-V — это задача будущего.



Пример универсального представления для функции сложения с насыщением.

|  |
| --- |
| Сложение с насыщением в SSA форме  int satadd (int s, int t) {  **entry**:  int d1 = s + t;  **if** (d1 > MAX) **goto** clamp;  **clamp**:  int d2 = MAX;  **end**:  int d3 = φ(d3: entry, d2: clamp);  **return** d3; } |

## Заключение

Не смотря на полвека исследований алгоритмов порождения инструкций, компиляция в оптимизированный код является всё ещё не до конца решенной задачей. Существуют разные подходы, каждый из которых не является вполне универсальным. Из-за этого обход этих недостатков обычно делается с помощью отдельной фазы компиляции. Например, если выбор SIMD, NEON и векторных инструкций не поддерживается в фазе порождения кода, то стоит добавлять отдельный проходы, которые порождают такие инструкции, часто с помощью так называемых polyhedral оптимизаций, или используя super-word parallelism [[Larsen and Amarasinghe, 2000](#_9e97cfb0a5d0051c944fe2cb581f0ba0)].

При порождении инструкций для заказных процессоров (англ. Application-specific instruction-set processor, ASIP) задача усложняется другим образом. Так как в процессор можно добавлять пользовательские инструкции, то шаблоны распознавания инструкций больше не становятся статически известными при сборке компилятора.

Как было уже сказано ранее, оптимальное порождения инструкций должен производится вместе с другими фазами порождения кода. Переупорядочивание инструкций ради более грамотного использования конвейера особенно актуально для VLIW-процессоров. Другой проблемой является рематериализация при распределении регистров: иногда значения легче пересчитать заново, чтобы снизить количество занятых регистров и сократить пересылки между процессором и памятью.

Также существуют методы порождения инструкций [[Leather, 2019](#_0881b5b44b5d7d1f5936fe47096c3538)], которые стоят особняком от выше упомянутых, так как они основаны на машинном обучении.

[AC75] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, jun 1975. URL: <https://doi.org/10.1145/360825.360855>, [doi:10.1145/360825.360855](https://doi.org/10.1145/360825.360855).

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, 11(4):491–516, oct 1989. URL: <https://doi.org/10.1145/69558.75700>, [doi:10.1145/69558.75700](https://doi.org/10.1145/69558.75700).

[Beg13] Mirza Omer Beg. *Combinatorial Problems in Compiler Optimization*. PhD thesis, University of Waterloo, Ontario, Canada, 2013. URL: <https://hdl.handle.net/10012/7423>.

[Ben] E. Bendersky. A deeper look into the llvm code generator: part 1. URL: <http://eli.thegreenplace.net/2013/02/25/a-deeper-look-into-the-llvm-code-generator-part-1/>.

[Bli16] G.H. Blindell. *Instruction selection: Principles, methods, and applications*. 01 2016. [doi:10.1007/978-3-319-34019-7](https://doi.org/10.1007/978-3-319-34019-7).

[Bli18] Gabriel Hjort Blindell. *Universal Instruction Selection*. PhD thesis, KTH Royal Institute of Technology, Sweden, 2018. URL: <https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-223599>.

[CP95a] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, 35–49. New York, NY, USA, 1995. Association for Computing Machinery. URL: <https://doi.org/10.1145/202529.202534>, [doi:10.1145/202529.202534](https://doi.org/10.1145/202529.202534).

[CP95b] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. *SIGPLAN Not.*, 30(3):35–49, mar 1995. URL: <https://doi.org/10.1145/202530.202534>, [doi:10.1145/202530.202534](https://doi.org/10.1145/202530.202534).

[CFSV01] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. An improved algorithm for matching large graphs. In 2001. URL: <https://api.semanticscholar.org/CorpusID:15968654>.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, oct 1991. URL: <https://doi.org/10.1145/115372.115320>, [doi:10.1145/115372.115320](https://doi.org/10.1145/115372.115320).

[DF84] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Trans. Program. Lang. Syst.*, 6(4):505–526, oct 1984. URL: <https://doi.org/10.1145/1780.1783>, [doi:10.1145/1780.1783](https://doi.org/10.1145/1780.1783).

[ER70] M. Elson and S. T. Rake. Code-generation technique for large-language compilers. *IBM Systems Journal*, 9(3):166–188, 1970. [doi:10.1147/sj.93.0166](https://doi.org/10.1147/sj.93.0166).

[GSW95] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. Program. Lang. Syst.*, 17(1):85–122, jan 1995. URL: <https://doi.org/10.1145/200994.201003>, [doi:10.1145/200994.201003](https://doi.org/10.1145/200994.201003).

[GG78] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, 231–254. New York, NY, USA, 1978. Association for Computing Machinery. URL: <https://doi.org/10.1145/512760.512785>, [doi:10.1145/512760.512785](https://doi.org/10.1145/512760.512785).

[GHS82a] Susan L. Graham, Robert R. Henry, and Robert A. Schulman. An experiment in table driven code generation. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, 32–43. New York, NY, USA, 1982. Association for Computing Machinery. URL: <https://doi.org/10.1145/800230.806978>, [doi:10.1145/800230.806978](https://doi.org/10.1145/800230.806978).

[GHS82b] Susan L. Graham, Robert R. Henry, and Robert A. Schulman. An experiment in table driven code generation. *SIGPLAN Not.*, 17(6):32–43, jun 1982. URL: <https://doi.org/10.1145/872726.806978>, [doi:10.1145/872726.806978](https://doi.org/10.1145/872726.806978).

[KG08] David Ryan Koes and Seth Copen Goldstein. Near-optimal instruction selection on dags. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, 45–54. New York, NY, USA, 2008. Association for Computing Machinery. URL: <https://doi.org/10.1145/1356058.1356065>, [doi:10.1145/1356058.1356065](https://doi.org/10.1145/1356058.1356065).

[LA00a] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, 145–156. New York, NY, USA, 2000. Association for Computing Machinery. URL: <https://doi.org/10.1145/349299.349320>, [doi:10.1145/349299.349320](https://doi.org/10.1145/349299.349320).

[LA00b] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *SIGPLAN Not.*, 35(5):145–156, may 2000. URL: <https://doi.org/10.1145/358438.349320>, [doi:10.1145/358438.349320](https://doi.org/10.1145/358438.349320).

[Lea19] Hugh Leather. Deep learning for compilers. 2019. URL: <https://www.inf.ed.ac.uk/teaching/courses/copt/lecture-15.pdf>.

[Mas87] Henry Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, 122–126. New York, NY, USA, 1987. Association for Computing Machinery. URL: <https://doi.org/10.1145/36206.36194>, [doi:10.1145/36206.36194](https://doi.org/10.1145/36206.36194).

[Mil71] P. L. Miller. Automatic creation of a code generator from a machine description. Technical Report, USA, 1971.

[New75] Joseph Michael Newcomer. *Machine-independent generation of optimal local code.* PhD thesis, USA, 1975. AAI7521781.

[NKWA95] Albert Nymeyer, Joost-Pieter Katoen, Ymte Westra, and Henk Alblas. Code generation = a\* + burs. In 12 1995. [doi:10.1007/3-540-61053-7\_60](https://doi.org/10.1007/3-540-61053-7_60).

[OW69] Richard J. Orgass and William M. Waite. A base for a mobile programming system. *Commun. ACM*, 12:507–510, 1969. URL: <https://api.semanticscholar.org/CorpusID:8164996>.

[Rip77] K. Ripken. Formale beschreibung von maschinen, implementierungen und optimierender maschinencodeerzeugung aus attributierten program- mgraphen. *Tech. rep. TUM-INFO-7731*, 1977.

[SCC+18] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: a synthesizing superoptimizer. 2018. [arXiv:1711.04422](https://arxiv.org/abs/1711.04422).

[Sta88] R. Stallman. Internals of gnu cc. apr 1988. URL: <https://web.archive.org/web/20120112142027/http://trinity.engr.uconn.edu/~vamsik/internals.pdf>, [doi:10.1145/69558.75700](https://doi.org/10.1145/69558.75700).

[Ull76] J. R. Ullmann. *An Algorithm for Subgraph Isomorphism*. PhD thesis, New York, NY, USA, jan 1976. URL: <https://doi.org/10.1145/321921.321925>, [doi:10.1145/321921.321925](https://doi.org/10.1145/321921.321925).

[Wil71] Thomas Richard Wilcox. *Generating machine code for high-level programming languages*. PhD thesis, USA, 1971.