Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Теоретическая информатика и компьютерные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ ПО КУРСУ «КОНСТРУИРОВАНИЕ КОМПИЛЯТОРОВ» НА ТЕМУ:

Разработка оптимизирующего компилятора

Студент _____ _____

подпись, дата *фамилия, и.о.*

Научный руководитель _____ _____

подпись, дата *фамилия, и.о.*

2025 г

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компиляторы являются ключевым звеном в современном программировании, обеспечивая преобразование исходного кода на высокоуровневых языках в эффективный машинный код, понятный оборудованию. Они скрывают от разработчика аппаратные детали, повышают переносимость программ и позволяют концентрироваться на логике задачи, а не на низкоуровневых особенностях конкретной архитектуры. Кроме того, компиляторы выполняют статический анализ кода, выявляя ошибки ещё на этапе разработки, что снижает стоимость их исправления и повышает надёжность создаваемых программных систем. Оптимизирующие возможности компиляторов напрямую влияют на производительность, энергопотребление и ресурсную эффективность программного обеспечения, что особенно важно в условиях роста объёмов данных и требований к быстродействию.

Актуальность разработки простого оптимизирующего компилятора обусловлена тем, что он позволяет, с одной стороны, исследовать и демонстрировать базовые принципы оптимизаций, а с другой — создавать практический инструмент, пригодный для обучения и прототипирования. В условиях, когда промышленные компиляторы чрезвычайно сложны и громоздки, наличие упрощённой, но функциональной реализации даёт возможность глубже понять внутренние механизмы трансляции и оптимизации программ.

Целью данной работы является разработка оптимизирующего компилятора для модельного языка с промежуточным представлением в виде SSA формы. Оптимизирующими проходами были выбраны следующие оптимизационные проходы: продвижение констант (англ. sparse conditional constant propogation), удаление мёртвого кода и вынос инвариантных инструкций из циклов.

1. Основные понятия

Компилятор — это программа, которая переводит предложения одного языка в эквивалентное предложение на другом языке. Язык, с которого осуществляется перевод, называется *исходным языком*, а язык, на который переводят — *целевым языком*. Обычно, исходный язык — это некоторый высокоуровневый язык, на котором программистам удобно описывать алгоритмы (например, C++ или FORTRAN), а целевой язык — это низкоуровневый язык, точно описывающий команды для выполнения компьютером (например, ассемблер процессора x86 или байткод JVM). Компилятор также может переводить один высокоуровневый язык в другой высокоуровневый. Тогда такой компилятор принято называть *транспилятором*. Самым популярным транспилятором на сегодняшний день является компилятор языка TypeScript, производящий трансляцию исходного языка в JavaScript.

В задачи компилятора входит проверка поданной последовательности на соответствие правилам языка. Если исходная последовательность соответствует этим правилам, то компилятор обязан принять её и выдать соответствующее предложение на целевом языке. Если же компилятор обнаружил несоответствие, то он обязан выдать соответствующее сообщение об ошибке.

В задачи компилятора также зачатую входит *оптимизация* поданной программы. Для этого, компилятор совершает некоторые преобразования, превращая исходную программу в другую, *эквивалентную* ей. Эквивалентность двух программ означает, что они производят одинаковые результаты на всех возможных входных данных, сохраняя видимое поведение. От качества оптимизаций напрямую зависит скорость работы программы на целевой платформе.

1.1 Архитектура компилятора

Стандартный компилятор обычно состоит из нескольких модулей - *фронтенда* и *бэкенда*. Модуль фронтенда занимается анализом предложения на исходном языке и представлением его в некотором промежуточном

представлении, пригодном для последующей обработки бэкендом. Бэкенд же занимается оптимизацией поданного промежуточного представления и порождением целевого языка из поданного промежуточного представления[1].

Данное архитектурное решение хорошо тем, что два модуля могут разрабатываться независимо друг от друга, увеличивая общую скорость разработки, благодаря общему промежуточному представлению. Промежуточное представление программы также позволяет поддерживать несколько исходных языков - достаточно лишь написать соответствующий модуль фронтенда. Подобным же образом можно заменять бэкенды компилятора для поддержки других целевых языков. Так устроены многие проекты, например GCC и LLVM.

1.2 Фронтенд

Модуль фронтенда состоит из нескольких видов анализов: лексического, синтаксического и семантического. На фазе лексического анализа исходный текст программы преобразуется в последовательность токенов. На фазе синтаксического анализа токены формируются в синтаксические блоки, описывающие структуру программы. На базе разобранных блоков строится абстрактное синтаксическое дерево, над которым будет проводится семантический анализ, цель которого — проверка на соответствие получившегося дерева, описывающего программу, правилам языка (например, проверка на соответствие типов данных переменных или проверка того, что используемая переменная была объявлена).

1.2.1 Лексический анализ

В задачу лексического анализа входит чтение потока символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называемыми лексемами[2].

Существует несколько подходов к реализации фазы лексического анализа. Один подход строит лексический распознаватель, основываясь на детерминированном конечном автомате. Для этого необходимо выделить

лексические домены, провести факторизацию алфавита, составить таблицу переходов конечного автомата и провести его детерминизацию.

Второй подход предполагает объектно-ориентированный подход, который несколько проще в реализации и поддержании. Он также более гибок, давая разработчику больше контроля.

1.2.2 Синтаксический анализ

У исходного языка существует некоторая грамматика, которая описывает его. Формулировка синтаксического анализа[2] заключается в следующем: имея предложение на исходном языке необходимо восстановить его дерево разбора, на основе которого будет построено абстрактное синтаксическое дерево, описывающее структуру программы.

Существует множество методов разбора потока токенов. Из них выделяются следующие типы алгоритмов разбора: нисходящие и восходящие.

Нисходящие алгоритмы начинают со стартового символа грамматики (от корня дерева разбора). К таким алгоритмам относится, например, рекурсивный спуск (LL(k)).

Восходящие алгоритмы разбирают последовательность начиная с листьев дерева разбора и основаны на использовании стэка. К этим алгоритмам относятся, например, LR(k), LALR(1).

В большинстве компиляторах (например, clang[6], rustc[7]), в основном используется самописный разбор рекурсивным спуском. Это обусловлено следующим:

- можно обрабатывать контекстно-зависимые конструкции в языке;
- более детальная диагностика ошибок;
- проще отлаживать процесс разбора.

1.2.3 Семантический анализ

Задача семантического анализа[2] — проверить абстрактное синтаксическое дерево на соответствие правилам языка. К этим правилам относятся, например, проверка на соответствие типов переменной и

присваиваемому ей значения, проверка существования переменной в точке её использования.

Результатом семантического анализа являются таблицы символов для каждой области видимости, в которых хранятся переменные и функции, определенные в данной области, а также их типы данных. Таблицы крепятся к соответствующим узлам абстрактного синтаксического дерева для дальнейшего использования.

1.2.4 Порождение промежуточного представления

В императивных языках программирования в основном используется промежуточное представление в виде SSA формы.

SSA форма характеризуется тем, что в ней у каждой переменной существует лишь одно определение[1]. Данное промежуточное представление значительно упрощает поиск цепочек определение-использования, делая многие оптимизации в бэкенде, таких как удаление мёртвого кода или продвижения констант, значительно легче.

SSA промежуточное представление организованно в виде *графа потока управления* (англ. control flow graph, CFG), в котором узлы — базовые блоки, а ребра между блоками указывают передачу управления. Базовый блок — это набор инструкций, у которого только одна точка входа — его начало, и одна точка выхода — его конец. Базовый блок кончается *инструкциями-терминаторами*, то есть инструкциями, которые осуществляют передачу управления. К таким инструкциям относятся инструкция возврата и инструкция (без-)условного перехода. Вызов функции при этом не считается терминатором, так как выполнение функции Любой граф потока управления имеет единственный начальный блок и единственный выходной блок. В случае раннего возврата из функции, базовый блок, содержащий возврат, логически связывается с выходным блоком.

В SSA вместо перезаписи переменной компилятор создает новые версии значения (например, x_1, x_2, x_3), чтобы каждое определение было уникальным.

Когда управление может прийти в блок из разных ветвей (`if`/`else`, циклы), применяются *фи-узлы*, которые выбирают правильное входное значение в зависимости от того, откуда пришло управление. Это позволяет сохранить правило «одно присваивание на значение», даже если на исходном языке переменная менялась в разных ветвях.

Стоит определить основные определения, связанные с отношением доминирования одного базового блока другим[3]:

- базовый блок A является *доминатором* базового блока B, если все пути из входного блока в выходной, проходящие через блок B, обязательно проходят и через блок A (обозначается как $A \succcurlyeq B$);
- базовый блок является A является *строгим доминатором* блока B, если A является доминатором B и $A \neq B$ (обозначается как $A \succ B$);
- базовый блок A является *непосредственным доминатором* блока B, если A является доминатором блока B и ни один блок, лежащий между A и B не является доминатором блока B;
- базовый блок B принадлежит *фронту доминирования* базового блока A, если A доминирует некоторого предка B и не является строгим доминатором B:

$$DF = \{B | \exists P \in pred(B) : X \succcurlyeq P \wedge A \not\succ B\}$$

Стандартный алгоритм построения SSA формы[1] расставляет фи-узлы используя итерированный фронт доминирования: для каждого блока, в котором была определена данная переменная, находится его соответствующий фронт доминирования, куда будут проставлены фи-узлы для данной переменной. После вставки фи-узла в новое место появилась новая точка определения переменной, поэтому процесс расстановки продолжается уже с этого блока. Такой процесс продолжается пока не закончатся все блоки фронта доминирования.

После расстановки фи-узлов используется подход с обходом дерева доминаторов для расстановки версий переменных.

Обрезанная SSA форма (англ. pruned SSA) строится похожим на стандартный подход образом, но в ней при расстановке фи-узлов, помимо фронта доминирования, используется результат анализа живости (англ. liveness analysis): фи-узлы расставляются только в те блоки фронта доминирования, в которых данная переменная находится во множестве `LiveIn` данного блока, то есть необходимо чтобы эта переменная «была жива» в рамках блока. Такой подход не расставляет бесполезные фи-узлы, отсюда и название подхода. Расстановка версий происходит аналогично стандартному алгоритму.

1.3 Бэкенд

Бэкенд компилятора занимается оптимизацией промежуточного представления и порождением оптимального кода. Обычно, оптимизации организованы в виде независимых *проходов*, которые можно запускать в любом порядке. Оптимизационные проходы формируют *конвейер*, где каждый проход выполняется один за другим. Иногда, одна и та же оптимизация может выполняется несколько раз, но в разных частях конвейера. Это обусловлено тем, что некоторые оптимизации облегчают работу других оптимизационных проходов, после которых запускаются другие проходы, обеспечивающие больше оптимизационных возможностей первым.

Основное правило, которого должны придерживаться оптимизирующие компиляторы — это то, что оптимизация обязана сохранить наблюдаемое поведение программы на любых входных данных. Данное условие обязывает быть крайне внимательным к семантике операций языка. Например, при мёртвого кода, нельзя удалить операцию деления, если статически нельзя доказать что делитель отличен от нуля, ведь деление на ноль порождает исключение, что является внешне наблюдаемым поведением.

Оптимизационные проходы принято классифицировать по области их действия:

- peephole оптимизация: рассматривается несколько соседних инструкций;

- локальная оптимизация: рассматривается один базовый блок;
- внутрипроцедурная оптимизация: рассматривается одна подпрограмма;
- оптимизация циклов: рассматривается один или несколько вложенных циклов;
- межпроцедурная оптимизация: рассматривается весь исходный код программы.

1.3.1 Распространение констант

Целью продвижения констант является замена выражений, требующих для их вычисления только констант, значением вычисленного выражения. Существует и более агрессивный вид данной оптимизации, умеющий определять недостижимые блоки графа потока управления, который базируется на SSA представлении.

Данная оптимизация уменьшает количество вычисляемых выражений, заменяя вычисления сразу на его результат, ускоряя тем самым работу программы.

При распространении констант стоит учесть, что некоторые крайние случаи, когда для вычисления хоть и требуются только константные выражения, но, тем не менее, нельзя заменить вычисление на его результат. Например, при делении одного числа на другое нельзя распространить результат вычисления при делении на ноль.

1.3.2 Удаление мёртвого кода

Целью удаления мёртвого кода является удаление нигде не используемых, или *мёртвых*, выражений. Удаляя ненужные выражения, очевидно, ускоряет работу программы.

Как и в случае продвижения констант, существуют некоторые краевые случаи. Например, если не получилось доказать, что знаменатель всегда отличен от нуля, то удалять такую инструкцию будет ошибкой, даже если результат деления нигде не используется, так как деление на ноль вызывает исключение и является наблюдаемым поведением. Удаление такой инструкции

удалит и порождение исключения, что нарушит семантику программы. Также, нельзя удалять вызовы функций, так как функции могут порождать побочные эффекты.

Данный оптимизационный проход хорошо сочетается с проходом распространения констант: сначала продвигаются константы, и многие инструкции становятся инструкциями присваивания константе. Далее, многие такие инструкции можно удалить, так как их значения уже были распространены по графу потока управления и, по сути, данные присваивания стали бесполезными.

1.3.3 Вынос инвариантного кода из циклов

Целью данной оптимизации является обнаружение и вынос инвариантных относительно цикла инструкций. Это даёт прирост к производительности, так как инвариантные инструкции не вычисляются лишние разы во время исполнения цикла, а вычисляется заранее единожды.

Данная оптимизация требует, однако, чтобы циклы программы были приведены к некоторому специальному виду: к каждому циклу добавляется специальный блок, называемый *предзаголовком* (англ. preheader), который предшествует первому блоку цикла. Именно в этот блок будут выноситься инвариантные инструкции данного цикла.

2. Разработка оптимизирующего компилятора

Компилятор должен работать по следующему принципу: на вход ему поступает код исходной программы, после чего на выходе должно получиться оптимизированное промежуточное представление программы.

2.1 Спецификация исходного языка

Исходный язык представляет собой C-подобный императивный язык программирования с поддержкой целочисленного типа данных и целочисленных массивов любой размерности.

Язык должен поддерживать следующие структуры:

- объявление и присваивание целочисленных переменных;
- объявление и присваивание массивов произвольной размерности;
- вызов функций;
- циклы о счётчиком;
- безусловный цикл;
- механизм прерывания цикла `break` и завершения текущий итерации цикла `continue`;

2.2 Фронтенд

Фронтенд компилятора принимает на вход текст программы на исходном языке, а на выходе отдает специальную построенный граф потока управления. В этом графе каждый узел, называемым *базовым блоком*, представляет собой набор инструкций промежуточного представления. Основным свойством базового блока является то, что у него лишь одна точка входа — его начало, и одна точка выхода — конец этого блока, где происходит передача управления к другим блокам графа.

2.2.1 Лексический анализ

Лексическим анализом занимается сущность `Lexer`. `Lexer` принимает на вход исходный текст программы и трансформирует его в список разобранных токенов.

У Lexer есть несколько полей: source, в котором находится текст исходной программы, pos — позиция курсора в тексте программы, а также row и col — строка и столбец положения курсора.

Lexer на каждом шаге своей работы в жадном режиме находит самый длинный префикс исходной последовательности, принадлежащий одному из нескольких лексических доменов. Всего таких доменов несколько:

- домены ключевых слов (по одному на каждый): "func", "int", "void", "let", "if", "else", "for", "return", "break", "continue";
- домены операторов (по одному на каждый тип оператора): "+", "-", "*", "/", "%", "==", "!=", "<", "<=", ">", ">=", "&&", "||", "!", "=", "->";
- домены пунктуации (по одному на каждый): "(", ")", "{", "}", "[", "]", ";", ",";
- домен идентификаторов: непрерывная последовательность латинских букв, цифр и знака «_», причем которая обязательно начинается либо с «_», либо с буквы. домен целых чисел: непрерывная последовательность, состоящая из цифр.
- Служебный домен, означающий конец входной последовательности: EOF.

Лексический анализатор реализует функционал преобразования исходного текста программы в последовательность *токенов*. Токен — это набор из лексемы и её координат в исходном тексте программы. Место в исходном тексте определяется номером строки и номером столбца, в котором оказалась данная лексема. Также, у токенов есть атрибут, зависящий от лексического домена куда принадлежит его лексема. В случае нахождения ошибки, анализатор обязан отдать специальный токен домена ERROR.

2.2.2 Синтаксический анализ

Синтаксическим анализом занимается сущность Parser. Parser обрабатывает поданную последовательность токенов, восстанавливая дерево разбора грамматики языка и строя абстрактное синтаксическое дерево[4]. Простота исходного языка позволяет использовать для анализа

последовательности токенов классический алгоритм анализа, основанного на рекурсивном спуске.

`Parser` обязан работать следующим образом: начать разбор со стартового нетерминала грамматики и углубляться рекурсивно вниз по грамматике, проверяя синтаксическую структуру языка и параллельно строя абстрактное синтаксическое дерево. В случае ошибки, синтаксический анализатор обязан вызвать исключение синтаксического разбора и завершить его без попытки восстановления.

Грамматика разбираемого языка представлена в листинге А.1.

2.2.3 Семантический анализ

Семантическим анализом будет заниматься сущность `SemanticAnalyzer`. Данная сущность анализирует полученное абстрактное синтаксическое дерево с фазы синтаксического анализа начиная с его корня и продолжая рекурсивно вниз по дереву.

Семантический анализатор обязан предусмотреть следующее:

- все переменные должны быть предварительно определены перед их использованиями;
- функции могут определяться в любом порядке;
- типы переменных и присваиваемых им значений должны совпадать;
- количество и типы аргументов вызываемых функций должны соответствовать заявленным при объявлении;
- индексировать массив можно только если размерность индекса совпадает с размерностью типа массива;
- нельзя присваивать один массив другому массиву;
- запретить переопределение переменных, разрешается только переприсваивание;
- запретить передачу одного и того же массива в качестве нескольких аргументов для функции — предполагается, что все указатели указывают на непересекающиеся участки памяти (это можно сравнить

с квалификатором `restrict` из языка C).

Область видимости определяется блоком. К блокам относятся вся программа, тело функции, тело циклов и блоки оператора ветвления.

В случае успешного прохождения семантического анализа в узлы абстрактного синтаксического дерева, соответствующих блокам, будут вставлены их таблицы символов. В случае какой-либо неудачи, анализатор попытается восстановиться от ошибки и продолжит анализ.

2.2.4 Порождение промежуточного представления

Как было указано ранее, промежуточное представление представлена в виде графа потока выполнения, состоящий из базовых блоков. Графом в программе представляется сущность `CFG`. В качестве полей она имеет имя функции, которую данный граф описывает, а также ссылки на входной и выходной базовые блоки.

Сами базовые блоки описываются сущностью `BasicBlock`. У неё имеются следующие поля:

- метка, на которую нужно совершить переход для передачи управления данному базовому блоку;
- информация, описывающая что это за блок;
- таблица символов области видимости, в которую попадает блок;
- набор инструкций базового блока;
- набор фи-узлов базового блока;
- список предшественников и преемников блока;

Построением по абстрактному синтаксическому дереву `CFG` занимается сущность `CFGBuilder`. У сущности существует несколько полей, а именно:

- счетчик количества блоков, необходимый для порождения имен блоков;
- счетчик количества временных переменных, нужный для проставления их имен;
- ссылка на текущий заполняемый блок;
- стэк ссылок на базовые блоки, отвечающий на вопрос куда передать

управление после инструкции `break`;

- стэк ссылок на базовые блоки, отвечающий на вопрос куда передать управление после инструкции `continue`.

Каждый синтаксический блок — это набор идущих друг за другом утверждений. Каждое из этих утверждений последовательно преобразуется в одну или более инструкций, вставляемых в базовые блок. Каждая инструкция — это некоторая операция над одним или двумя операндами с одним выходным значением. Исключением является лишь вызов функции, в котором количество операндов может быть неограниченным.

Каждое утверждение разбирается рекурсивно. Некоторые сложные утверждения создают новые базовые блоки, такие как, например, циклы или операторы ветвления.

При построении оператора ветвления, создаются от двух до трёх новых базовых блоков, в зависимости от того, есть ли у него ветвь «иначе» или нет. Один базовый блок создается как блок схождения управления.

Каждый цикл со счётчиком строится как цикл постусловием, с дополнительным условием перед циклом, проверяющее, запустится ли самая первая итерация данного цикла или нет. При несоблюдении этого условия управление перейдёт в блок, находящийся сразу после цикла. Вторым блоком является предзаголовком. Он необходим для оптимизационного прохода выноса инвариантных инструкций цикла и изначально пуст. Третьим блоком будет само тело цикла, в которое будут заноситься его инструкции. Четвертым блоком является блок обновления индуктивных переменных, называемый защёлкой (англ. latch)[1]. Отдельный блок нужен чтобы при прерывании текущей итерации с помощью инструкции `continue` сработало изменение индуктивных переменных. Основным свойством защёлки является то, что это единственный блок в цикле, обладающий обратной дугой. Это свойство упрощает поиск циклов в графе. Пятым блоком является хвостовой блок цикла. В этот блок происходит передача управления по завершении цикла. Этот блок является

служебным, и всегда будет пустым. Данный блок необходим для облегчения обнаружения всех блоков данного цикла. Его важным свойством является то, что все его предшественники обязательно являются блоками соответствующего цикла. Шестым блоком является выходной блок (англ. exit block). Именно в нём будет происходить построение следующих за циклом инструкций.

Для безусловного цикла всё остается почти тем же самым, за исключением того, что блок обновления индуктивных переменных будет пустым, и того, что в этом случае не будет проверки условия запуска первой итерации цикла.

Во время порождения циклов в CFG сохраняется информация о нем, а именно:

- ссылка на блок — предзаголовок;
- ссылка на первый блок тела цикла, также называемым заголовком цикла;
- ссылка на хвостовой блок цикла.

Сохранение данной информации нужно для выноса инвариантного кода из циклов.

Стоит уточнить, что у выходного блока его предшественником может быть блок *не* принадлежащий его циклу. Примером такой ситуации как раз является цикл со счётчиком. Проверка условия запуска первой итерации передаёт управление в выходной блок, но этот блок самому циклу не принадлежит.

Преобразованием построенного графа потока управления к обрезанной SSA форме занимается сущность `SSABuilder`. Для этого она выполняет анализ живости для получения *LiveIn* множества для каждого базового блока, а также вычисляет фронт доминирования и дерево доминаторов для расстановки фи-узлов и версионирования соответственно.

В задачу анализа живости[3] входит получение множеств *LiveIn* и *LiveOut* для каждого базового блока. Они получаются по следующим

формулам:

$$\mathrm{LiveOut}[c] = \bigcup_{s \in succ(c)} \mathrm{LiveIn}[s]$$

$$\mathrm{LiveIn}[c] = \mathrm{Uses}[c] \cup (\mathrm{LiveOut}[c] \setminus \mathrm{Defs}[c])$$

где

- $Uses[c]$ — переменные, используемые в блоке c при вычислении выражений;
- $Defs[c]$ — переменные, определенные в блоке c .

В начале алгоритма данные множества являются пустыми. Процесс вычисления множеств проводится для каждого базового блока до схождения к неподвижной точке.

Нахождение доминаторов каждого базового блока выполняется итеративным алгоритмом до схождения до неподвижной точки. Множество доминаторов данной вершины вычисляется по формуле [3]

$$\mathrm{DOM}(n) = \{n\} \cup \left(\bigcap_{m \in \mathrm{preds}(n)} \mathrm{DOM}(m) \right)$$

В самом начале, в множество доминаторов для каждой вершины графа потока управления, кроме начальной, являются все вершины графа. Для начальной вершины множество доминаторов равно ей самой.

Для нахождения фронта доминирования используется следующий алгоритм: для каждой вершины графа B, берётся её предшественник P. Если предшественник является непосредственным доминатором данной вершины, то этот предшественник пропускается. Иначе, к фронту доминаторов предшественника P добавляется вершина B. Затем, берётся непосредственный доминатор P, P’, и уже для него выполняются данные проверки. Данный алгоритм продемонстрирован на листинге 1.

Расстановка фи-узлов происходит по следующему принципу: для каждой переменной x и для каждого базового блока B, в котором данной переменной было присвоено некоторое значение, обходится фронт доминирования данного блока DF[B]. В этих блоках ставится фи-узел для переменной x, если его там

уже не было и если переменная x является живой в данном блоке. Далее, так как вставка фи-узла порождает новую точку определение, процедура повторяется и для данного блока. Алгоритм представлен на листинге 2.

Листинг 1 — вычисление фронта доминирования

```
 1  proc ComputeDominanceFrontierGraph(cfg, idom_tree)
 2      DF ← ассоциотивный массив со значением по умолчанию = пустое
множество
 3
 4      for each node in cfg do
 5          for each pred in node.predecessors do
 6              current ← pred
 7              while current ≠ idom_tree.idom(node) do
 8                  DF[current].add(node)
 9                  current ← idom_tree.idom(current)
10              end
11          end
12      end
13
14      return DF
15  end
```

Для расстановки версий начинает обходиться дерево доминаторов данного графа потока управления. Для переименовывания алгоритм хранит по стэку для каждой переменной, хранящий их версии. Изначально переименовываются фи-узлы данного блока. Затем, переименовываются и инструкции блока. Во время переименования сначала переименовывается правая часть выражения, а потом левая. При переименовании правой части выражения, номера версий берутся со стэка; при переименовании левой части порождается новая версия и она кладется в соответствующий стэк. После обработки инструкций, производится вставка операндов фи-узлов в блоках последователях. После этого, процедура вызывается для всех потомков данного блока в дереве доминаторов. В самом конце, со стэка снимаются версии переменных, определенных в данном блоке.

К алгоритму расстановки версий добавлена дополнительная деталь: во время его выполнения для каждой переменной определяется, является ли данная переменная указателем смещенным относительно какого-либо другого

указателя. Это необходимо для корректной работы удаления мёртвого кода с массивами.

Листинг 2 — расстановка фи-узлов

```
 1  proc put_phis(defs_by_var)
 2      for each (var, def_blocks) in defs_by_var do
 3          has_phi ← empty set of BasicBlock
 4          work ← list(def_blocks.defining_blocks)
 5
 6          while work is not empty do
 7              n ← work.pop()
 8
 9              for each y in DF[n] do
10                  if y ∈ has_phi then
11                      continue
12                  end
13
14                  if var ∉ live_in[y] then
15                      continue
16                  end
17
18                  y.insert_phi(var)
19                  add y to has_phi
20
21                  if y ∉ defs_by_var[var].defining_blocks then
22                      work.push(y)
23                  end
24              end
25          end
26      end
27  end
```

Для представления промежуточного представления программы в текстовом виде, у сущностей `BasicBlock`, `CFG` и каждой сущности, описывающей соответствующий тип инструкции, существуют методы `to_IR()`. При вызове данного метода у `BasicBlock`, сначала последовательно вызывается `to_IR()` у всех фи-узлов блока, а затем и у простых инструкций. Вызов метода у `CFG` приводит к обходу всех базовых блоков графа потока управления и вызова у них метода `to_IR()`.

2.3 Бэкенд

Оптимизационные проходы представляются собой как независимые сущности, каждая из которых модифицирует поданный граф потока управления

некоторым образом.

2.3.1 Распространение констант

Распространением констант[1] занимается сущность SCCP.

Данный алгоритм основан на следующей полурешетке:

- «Не константа» (верхний элемент полурешетки): значение переменной — не константа.
- «Константа»: значение переменной — константа;
- «Не определено» (нижний элемент полурешетки): значение переменной неопределено;

Основной операцией на элементах полурешётки является операция объединения.

В начальный момент времени считается, что все базовые блоки, за исключением входного, являются недостижимыми. Алгоритм поддерживает два списка — список достижимых базовых блоков для обхода и список переменных, изменившие своё значение по решётке.

Алгоритм заключается в следующем: постоянно поочерёдно обрабатываются достижимые блоки, затем измененные переменные, пока оба списка не опустеют.

При обработке базового блоки, сначала обрабатываются его фи-узлы, вычисляя значение переменной по полурешётке по всем достижимым предшественникам блока. Затем, поочерёдно обходятся инструкции данного блока, присуждая каждой присвоенной переменной блока значение полурешетки.

Для арифметических операций получение её значения по полурешетке делается следующим образом: если хоть какая-то переменная, используемая при вычислении выражения, не является константой, то и всё вычисленное выражение константой быть не может, поэтому данной переменной присваивается значение «не константа». Если хоть один операнд выражения является неизвестным, то и всё выражение будет неизвестным. Если же оба

являются константами, то выражение можно вычислить до конца, присвоив результату вычисленный результат в качестве значения по решётке.

Есть некоторые инструкции, которые всегда считаются неконстантными. Например, вызов результат вызова функции, объявление массива или загрузка значения по адресу.

Инструкции безусловного перехода всегда помечают ребро, символизирующее передачу управления от одного блока к другому, как выполнимое. Если данное ребро ещё не было помеченным выполнимым, то оно помечается, а сам базовый блок добавляется в список достижимых блоков, если он ранее не был помечен достижимым ранее. После этого, обновляются значения по полурешётке для каждого фи-узла следующего блока.

Инструкция условного перехода интерпретируется следующим образом: если оба выражения — константы, но можно однозначно понять, в какую ветвь произойдет передача управления. Если же хоть одно выражение — не константа по полурешётке, тогда необходимо принять консервативное решение и пометить обе ветви выполнимыми. Иначе, всё ещё недостаточно информации для принятия решения о выполнимости ветвей, и оно откладывается до момента продвижения информации к данной инструкции.

Изначально, все переменные имеют значение «не определено» по полурешётке. При присваивании нового значения по полурешетке данной переменной, происходит объединение старого и нового значений. Если значение переменной изменилось, тогда данная переменная помещается в список переменных к обработке.

Когда список базовых блоков к обработке опустел, начинают браться переменные из второго списка — списка переменных, у которых изменилось значение по полурешётке. Этот список отвечает за продвижение информации о константности по графу потока управления. Для каждой переменной из списка, все переменные, которые используют данную переменную для вычисления своего значения, пересчитывают своё значение по полурешётке по алгоритму,

описанному ранее.

2.3.2 Удаление мёртвого кода

Удалением мёртвого кода [5] занимается сущность `DCE`. Логика алгоритма заключается в следующем: нужно пометить все инструкции, которые гарантированно являются живыми, а также все инструкции, нужные для вычисления живых инструкций. Все остальные инструкции можно удалить. К таким инструкциям относятся инструкции ветвления, вызова функции, а также инструкция возврата. Также, нельзя удалять инструкции записи по адресу, связанной с аргументом функции являющимся указателем. Данный алгоритм является разновидностью алгоритмов пометки (англ. mark and sweep).

На первом этапе, обрабатывается каждая инструкция каждого базового блока графа потока управления, ища корневые инструкции, которые всегда помечаются живыми. К таким относятся инструкции переходов, возврата, вызов функции, а также операция присваивания результата деления или взятия остатка от деления, если делитель — не константное выражение или ноль. Затем, процесс повторяется для переменных, участвующие в вычислении данной переменной, и так далее. Процесс продолжается, пока не будет помечен весь граф вычисления. Если один из операндов — смещенный указатель на массив, то начинается обход графа потока управления в обратном порядке, помечая все выражения, записывающие значения используя данный указатель.

Второй этап заключается в переписывании графа путём удаления мёртвых инструкций.

2.3.3 Вынос инвариантного кода из циклов

Выносом инвариантных инструкций из циклов[5] занимается сущность `LICM`.

В начале работы алгоритма, для каждого цикла в графе производится поиск их базовых блоков. Делается это следующим образом: от хвостового блока берутся все его предшественники, затем их предшественники, и так далее, до предзаголовка цикла. Так как предзаголовок цикла доминирует над

всеми внутренними блоками цикла, процесс захватит только эти блоки и никакие другие.

Следующим этапом является нахождение инвариантных инструкций цикла. Инструкция называется инвариантной, если все операнды, требующиеся для вычисления инструкции, являются инвариантными. Операнд является инвариантным, если он является константным выражением или был определен вне цикла. Также требуется, чтобы такая инструкция доминировала все точки своего использования в рамках данного цикла, а также, чтобы она доминировала хвостовой блок цикла. Первое условие предотвращает неверный вынос инструкции, так как если точка использования не доминируется данной переменной, то существует некоторый путь в графе потока управления, который обходит стороной присваивание переменной, и если бы инструкция выносилась, то зависимая переменная получила бы неверное значение. Второе условие необходимо для обработки выхода из цикла: если определение переменной не доминирует хвостовой блок, то существует путь, по которому происходит выход из цикла, причём не проходящий через данное присваивание. Примеры данных условий продемонстрированы на рисунке 1.

Стоит отметить, что данный алгоритм способен работать только выносом арифметических операций. Он не способен работать с выносом операций, работающих с памятью, так как работа с памятью требует дополнительного анализа. Пример такого кода представлен в листинге 3. Также нельзя выносить и вызовы функций, так как функции в общем случае могут порождать побочные эффекты.

Листинг 3 — пример программы без инвариантных инструкций

```
1  func main() -> void {
2      let arr [10]int = {};
3      for (let i int = 0; i < 10; i = i + 1) {
4          arr[i] = i;
5          arr[5] = 0; // нельзя выносить из цикла
6      }
7  }
```

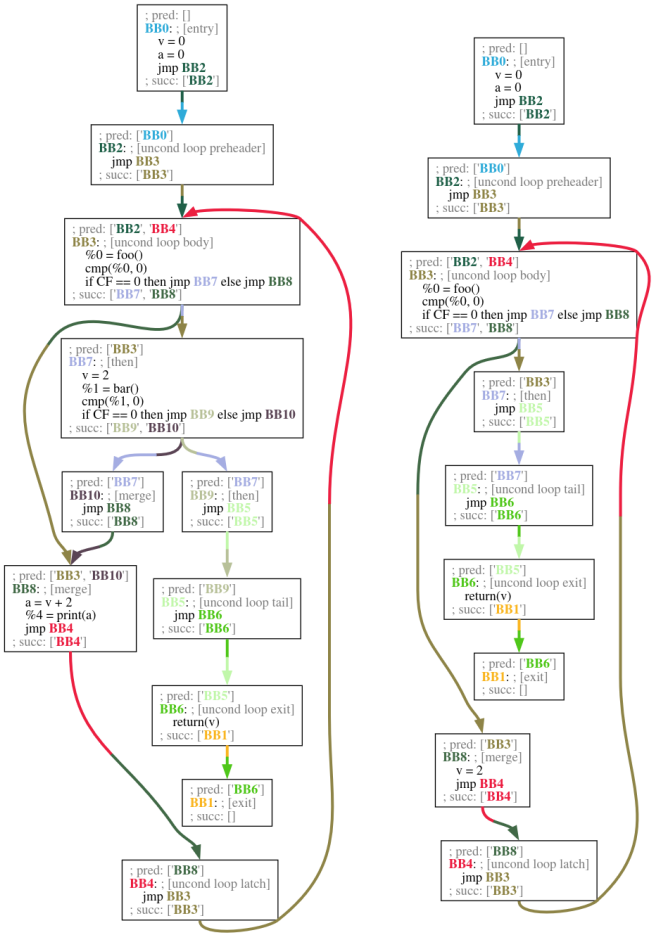

Рисунок 1: первое условие (слева) и второе условия выноса (справа)

2.4 Средства отладки

Для более удобного тестирования графов потока исполнения, необходимо обеспечить возможность их визуализации в формате `dot`[8]. Также, необходимо обеспечить возможность строить граф из текстового представления кода промежуточного представления. Это позволит разрабатывать тесты корректности оптимизаций, сравнивая полученное промежуточное представление и ожидаемое в графическом формате, что значительно удобнее текстового. Для большего удобства, необходимо ввести цветовое кодирование ребер и меток для быстрого распознавания ошибочных переходов при передачи управления в графе.

3. Реализация

Языком реализации был выбран язык Python из-за скорости прототипирования и системы типов, обеспеченную опциональной типизацией.

3.1 Фронтенд

Модуль фронтенда в коде поделён на два модуля: модуль анализа, лежащий в папке `src/parsing` и модуль порождения промежуточного представления, находящийся в папке `src/ssa`. Код в первом модуле ответственен за лексический, синтаксический и семантический анализы. Код во второй — за построение графа потока управления и SSA формы над этим графом.

3.1.1 Лексический анализ

Лексический анализатор представлен в коде классом `Lexer`. В своём конструкторе он принимает исходный текст программы. Основным методом у данного класса является метод `tokenize()`, разбивающий текст программы на последовательность токенов. Сам токен представлен в программе в виде класса `Token`.

Метод `tokenize()` устроен следующим образом он постоянно вызывает вспомогательный метод `_next_token()`, до тех пор, пока не обнаружит, что он вернул `EOF`-токен. В свою очередь Метод `_next_token()` жадно выделяет префикс от оставшегося текста программы, стараясь так получить токен максимальной длины. В ходе разбора, `Lexer` также отслеживает свою позицию в тексте программы.

3.1.2 Синтаксический анализ

В программе сущность `Parser` представлена одноименным классом. Данный класс реализует разбор методом рекурсивного спуска. Сам разбор осуществляется набором методов, по одному на каждый нетерминал грамматики.

Основным методом данного класса является метод `parse()`, результатом

которого является корень абстрактного синтаксического дерева, описывающего структуру программы.

Узлы абстрактного синтаксического дерева представляются различными классами в программе. Основным классом является класс `ASTNode`, являющийся абстрактным классом, от которого наследуются другие. У данного класса несколько полей, а именно `line` и `column` — строка и столбец в тексте программы, соответствующие положению узла в коде.

Всего, существует два типа узлов: утверждения и выражения. Обе категории представлены собственным абстрактным классом, `Statement` и `Expression` соответственно, от которых наследуются другие.

К утверждениям относятся те узлы, которые не порождают никакого значения. К ним относятся присваивание (`Assignment`), переприсваивание (), условный оператор (`Condition`), цикл со счётчиком (`ForLoop`), цикл без условия (`UnconditionalLoop`), вызов функции без получения результата (`FunctionCall`), возврат из функции (`Return`), прерывание цикла (`Break`), пропуск текущей итерации цикла (`Continue`).

К выражениям относятся те узлы, которые вычисляют некоторое значение. К ним относятся бинарное выражение (`BinaryOp`), унарное выражение (`UnaryOp`), идентификатор (`Identifier`), число (`IntegerLiteral`), вызов функции (`CallExpression`), взятие элемента по индексу (`ArrayAccess`).

Есть также узлы, не попадающие ни к утверждениям, ни к выражениям. Это классы `Program`, олицетворяющий программу как набор функций, `Function`, представляющий функцию как набор утверждений, и `Argument`, обозначающий аргумент функции.

В случае ошибки разбора, `Parser` вызывает исключение `ParseError`.

3.1.3 Семантический анализ

Семантический анализатор представлен классом `SemanticAnalyzer`. Основной целью данного класса является заполнение таблицы символов для

каждой области видимости. Сама таблица символов представлена классом SymbolTable. У SymbolTable несколько полей:

- variables — словарь, отображающий имена переменных, определенных в данной области видимости, в информацию об их типе данных;
- functions — словарь, отображающий имена функций, определенных в данной области видимости, в информацию об их типе;
- ссылка на родительскую таблицу символов.

Во время запроса информации о типе переменной или функции, сначала производится поиск информации в текущей таблице символов. В случае неуспеха поиск продолжается в родительских таблицах.

3.1.4 Порождение промежуточного представления

Построение промежуточного представления вынесено в отдельный модуль, находящийся по пути src/ir.

Граф потока управления представляется в коде классом CFG. Данный класс содержит в себе имя функции name, которую он описывает, две ссылки на входной и выходной блоки графа, entry и exit, а также список информации о построенных в нём циклах loops_info.

Информация о цикле представляется классом LoopInfo, которая включает в себя ссылку на предзаголовок preheader, ссылку на первый блок тела цикла, или заголовок цикла, header, а также ссылку на хвостовой блок цикла tail. Также, данный класс содержит множество блоков, которые составляют цикл, blocks, но при конструировании экземпляра класса оно не заполняется. Заполняется оно на этапе оптимизационного прохода выноса инвариантных инструкций из циклов.

Базовые блоки описываются классом BasicBlock. BasicBlock содержит в себе исполняемые инструкции, представленные абстрактным классом Instruction, а также набор фи-узлов данного блока. В своих вычислениях, инструкции работают с ssa-значениями — переменными или

константами. В коде они представлены в виде классов-наследников абстрактного класса SSAValue — SSAConstant и SSAVariable соответственно. SSAVariable в качестве полей класса имеет имя описываемой переменной name, номер версии переменной version, и идентификатор указателя, с которым связана данная переменная base_pointer. version и base_pointer устанавливаются на этапе построения SSA формы.

Операции над переменными представлены абстрактным классом Operation, у которого несколько наследников:

- OpCall — вызов функции;
- OpUnary — некоторая унарная операция;
- OpBinary — некоторая бинарная операция;
- OpLoad — загрузка значения по адресу;

Всего типов инструкций несколько:

- InstAssign — инструкция присваивания либо ssa-значения, либо операции над значениями;
- InstCmp — инструкция сравнения, включающая в себе условный переход;
- InstUncondJump — инструкция безусловного перехода;
- InstReturn — инструкция возврата;
- InstPhi — инструкция, символизирующая фи-узел;
- InstArrayinit — инициализация массива;
- InstStore — запись значения по адресу;
- InstGetArgument — служебная инструкция, символизирующая получение i-ого аргумента функции.

Построением графа потока управления занимается класс CFGBuilder, основным методом которого является метод build(), принимающий корень абстрактного синтаксического дерева Program на входе и отдающий экземпляр класса CFG на выходе. Данный класс рекурсивно обходит узлы абстрактного

синтаксического, заполняя базовые блоки инструкциями, соответствующими данному узлу дерева. Стоит отметить, что в данном промежуточном представления нет инструкций, вычисляющих несколько операций за раз. Все такие длинные выражения разбиваются на подвыражения, результат вычисления которых хранится во временных переменных.

Расстановкой фи-узлов и версионированием переменных занимается класс `SSABuilder`. Основной метод данного класса, `build()`, принимает на входе экземпляр класса `CFG`, который будет модифицироваться в хорде выполнения метода.

В рамках своей работы, `CFGBuilder` вычисляет множества `LiveIn` и `LiveOut` для каждого, строит множества доминаторов и фронт доминирования для каждого базового блока. Далее, используя эту информация, происходит расстановка фи-узлов, после чего следует и версионирование. При установке новой версии переменой, вычисляется её `base_pointer`, получая информацию о типе переменной из таблицы символов, закрепленной за базовым блоком. Если же такой информации в таблице не нашлось, как в случае со временными переменными, то базовый указатель выводится из операндов, которые используются при вычислении переменной. Результат работы алгоритма представлена на рисунке А.1.

3.2 Бэкенд

Модуль, отведенный под оптимизирующие проходы, расположен по пути `src/optimizations`. В рамках данного модуля было разработано 3 класса: `SCCP`, `DCE` и `LICM`, реализующие распространение констант, удаление мёртвого кода и вынос инвариантного кода из циклов соответственно. Каждый из классов является наследником абстрактного класса `OptimizationPass`, который обязывает каждый класс реализовать метод `run()`, принимающий на входе экземпляр класса CFG на входе.

Класс `SCCP` реализует метод `run()` следующим образом: в начале собирается информация о связях «определение-использования» в графе потока

управления, после чего запускается основной алгоритм продвижения информации по графу, основанный на двух списках — списке блоков и переменных. После того, как оба списка опустели, граф переписывается — удаляются недостижимые блоки, а в константных выражениях вычисления заменяются на их результаты. Код метода представлен на листинге 4.

Листинг 4 — метод `run()` класса `SCCP`

```
class SCCP(OptimizationPass):
    @override
    def run(self, cfg: CFG):
        self.cfg = cfg
        self._build_metadata(cfg)

        self._mark_block_executable(cfg.entry)
        self.executable_blocks.add(cfg.exit)

        while self.block_worklist or self.var_worklist:
            while self.block_worklist:
                bb = self.block_worklist.popleft()
                self._process_block(bb)

            while self.var_worklist:
                var_key = self.var_worklist.popleft()
                self._process_variable_users(var_key)

        self._rewrite_cfg()
        self._fold_constants()
```

Класс `DCE` реализует метод `run()` так: собирается информация о связях «определение-использования» в графе потока управления, после чего запускается фаза пометки инструкций как живыми. Следом идет фаза переписывания графа путем удаления мёртвых инструкций. Стоит отметить, что именно здесь и используется поле `base_pointer`, присваиваемое ssa-переменным на фазе построения SSA формы. Оно используется чтобы понять, какие переменные являются на самом деле адресами, и какие из операций следует сохранить. Реализация метода представлена на листинге 5.

Листинг 5 — реализация метода `run()` класса `DCE`.

```
class DCE(OptimizationPass):
    @override
    def run(self, cfg: CFG):
        self.cfg = cfg
        self._build_metadata(cfg)
        self._mark(cfg)
        self._sweep(cfg)
```

Класс `LICM` реализует метод `run()` следующим образом: в начале строятся множества доминаторов для каждого базового блока. После, собирается информация о связях «определение-использования», а также собирается информация о том, в каком базовом блоке была определена та или иная переменная. Затем, запускается процедура нахождения базовых блоков каждого построенного цикла. В самом конце, запускается сама процедура выноса инструкций из циклов. Реализация метода представлена на листинге 6.

Листинг 6 — реализация метода `run()` класса `LICM`.

```
class LICM(OptimizationPass):
    @override
    def run(self, cfg: CFG):
        self.cfg = cfg
        self.dom_tree = compute_dominator_tree(cfg)
        self._index_definitions(cfg)
        self._collect_loop_blocks(cfg)
        for loop in cfg.loops_info:
            self._hoist_loop(loop)
```

4. Тестирование

Для тестирования программы использовался модуль `unittest`. В нём для описания каких либо тестов необходимо создать класс, наследующийся от класса `unittest.TestCase`. Далее, все тесты описываются как методы.

В рамках тестирования целевыми показателями качества ставились процент покрытия кода и качество обработки краевых случаев. Покрытие кода тестами следующее:

- модуль анализа: 89% покрытия;
- модуль построения промежуточного представления: 93% покрытия;
- модуль оптимизаций: 95% покрытия.

В рамках тестирования лексического анализатора была проверка корректного распознавания им всех лексических доменов. В качестве тестов были выдвинуты распознавание как последовательностей, верно разбиваемых на токены, так и те, которые разбить нельзя. Модуль с тестами находится по путь `tests/test_lexer.py`.

В рамках тестирования синтаксического анализатора на вход ему подавались синтаксически разнообразные тексты программ, проверяющие работоспособность рекурсивного разбора и учитывающего приоритет операций: вложенные циклы и операторы ветвления, а также сложные арифметические выражения. Модуль с тестами находится по путь `tests/test_parser.py`.

Тестирование семантического анализатора представляет собой проверку, накладываемых на анализатор в секции разработки. Тестирование проводилось как на семантически верных, так и на неверных программах. В рамках тестирования было важно протестировать и работу таблиц символов: работает ли иерархический поиск переменных по областям видимости или нет. Модуль с тестами находится по путь `tests/test_semantic.py`.

Тестирование промежуточного представления, а также оптимизационных проходов, строится по следующей схеме: в каждом тесте создаётся 2

переменные: `src`, представляющий текст программы к обработке, и `expected_ir`, представляющий ожидаемое текстовое представление промежуточного представления. Затем, вызывается метод вспомогательный `assert_ir()`, порождающий промежуточное представление из `src`, и сравнивающий его с `expected_ir`. Если они не совпадают, то в качестве ошибки выдаются два графа потока управления в `dot` формате, отвечающие ожидаемому и полученному промежуточным представлениям соответственно. Пример теста продемонстрирован на листинге 7.

Листинг 7 — пример теста промежуточного представления.

```
def test_consecutive_assignments(self):
    src = """
        func main() -> int {
            let a int = 0;
            let b int = 0;
            b = a;
            return b;
        }
    """

    expected_ir = textwrap.dedent("""
        ; pred: []
        BB0: ; [entry]
            a_v1 = 0
            b_v1 = 0
            b_v2 = a_v1
            return(b_v2)
        ; succ: [BB1]

        ; pred: [BB0]
        BB1: ; [exit]
        ; succ: []
    """).strip()

    self.assert_ir(src, expected_ir)
```

В начале написания теста описывается лишь программа на исходном языке в `src` и `expected_ir` остается пустым. Далее, тест запускается и, он, очевидно, проваливается, однако в сообщении об ошибке показывается построенный программой промежуточное представление в виде графа потока управления. И если он соответствует ожиданиям, то полученное

промежуточное представление в текстовом виде ставится в переменную `expected_ir`.

Тестирование SSA формы происходило одновременно с тестированием оптимизационных проходов. Сами тесты расположены по путям `tests/test_ssa.py`, `tests/test_sccp.py`, `tests/test_dce.py`, `tests/test_licm.py`, `tests/test_sccp_dce.py`, `tests/test_sccp_licm.py`.

В рамках тестирования SSA формы была проверена корректность построения основных синтаксических конструкций — циклов со счётчиком и безусловных циклов, а также операторов ветвления. Вместе с ними были также протестированы инструкции прерывания циклов `break` и пропуска итерации `continue`. Также был проверен алгоритм расстановки фи-узлов, задействующий анализ живости и фронт доминирования.

В рамках тестирования оптимизационного прохода продвижения констант были проверены как тривиальные случаи продвижения внутри одного блока, так и нетривиальные, требующие доказательства достижимости базовых блоков.

При тестировании удаления мёртвого кода протестировано определение различения живых инструкций от мёртвых. Особое внимание было выделено тестированию определения инструкций записи по адресу как мёртвой или живой.

Тестирование выноса инвариантного кода требовало тестирования условий на инвариантность выносимых инструкций. Было необходимо удостовериться, что если условия на доминирование хвостового блока и всех точек использования нарушается, то инструкция не выносилась. Также было необходимо удостовериться, что выносимыми инструкциями являются только стандартные присваивания.

Были протестированы и комбинации оптимизационных проходов. Например, так как оптимизационный проход продвижения констант делает

множество константных инструкций мёртвыми, необходимо было удостовериться, что последующий проход удаления мёртвого кода их удалил бы. Также было необходимо удостовериться в продолжении работы оптимизации выноса инвариантного кода из циклов после оптимизации продвижения констант.

ЗАКЛЮЧЕНИЕ

В рамках курсовой был реализован оптимизирующий компилятор Си-подобного языка программирования, базирующегося на промежуточном представлении в виде SSA формы. Были реализованы и протестированы модули анализа входной программы, а также 3 оптимизирующих прохода: продвижение констант, удаление мёртвого кода и вынос инвариантного кода из циклов.

Основной проблемой данного проекта является то, что исходный язык несколько не выразителен — в нём отсутствуют многие типы данных, такие как числа с плавающей точкой или строки. В нём также нет возможности создавать пользовательские типы данных.

Существует несколько направлений развития проекта. Можно дополнить выразительность языка новыми типами данных, как встроенными, так и пользовательскими. Можно добавить другие оптимизационные проходы к уже имеющимся. Также, можно реализовать этап преобразования промежуточного представления в некоторый низкоуровневый код.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Владимиров, К. Оптимизирующие компиляторы. Структура и алгоритмы / К. Владимиров. — Москва: Издательство АСТ, 2024. — 272 с. — (Программирование для всех). — ISBN 978-5-17-167965-1.
2. Ахо А. В., Лам М. С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструментарий / пер. с англ. — 2-е изд. — М.: Вильямс, 2008. — 1175 с. — ISBN 978-5-8459-1349-4
3. Cooper K. D., Torczon L. Engineering a compiler / K. D. Cooper, L. Torczon. — 3rd ed. — Cambridge, MA: Morgan Kaufmann (Elsevier), 2022. — 824 p. — ISBN 978-0-12-815412-0.
4. Nystrom R. Crafting Interpreters / R. Nystrom. — Genever Benning, 2021. — 640 p. — ISBN 978-0-9905829-3-9.
5. Muchnick S. S. Advanced compiler design and implementation / S. S. Muchnick. — San Francisco: Morgan Kaufmann Publishers, 1997. — 856 с. — ISBN 1-55860-320-4
6. How Clang handles the "type-variable name" ambiguity of C/C++ [Электронный ресурс] / E. Ilio. — 2012. — URL: https://eli.thegreenplace.net/2012/07/05/how-clang-handles-the-type-variable-name-ambiguity-of-cc/ (дата обращения: 19.10.2025).
7. What type of parser does the Rust compiler use? [Электронный ресурс] / RedDocMD. — 2022. — URL: https://users.rust-lang.org/t/what-type-of-parser-does-the-rust-compiler-use/71430/3 (дата обращения: 19.10.2025).
8. Graphviz. Documentation [Электронный ресурс]. — 2025. — URL: https://graphviz.org/documentation/ (дата обращения: 04.11.2025)

ПРИЛОЖЕНИЕ А

Листинг А.1 — грамматика исходного языка

```
 1  PROGRAMM ::= FUNCTION* EOF
 2
 3  FUNCTION ::= "func" IDENTIFIER "(" ARG_LIST ")" "->" TYPE "{"
BLOCK "}"
 4
 5  ARG_LIST ::= EPSILON | ARG ("," ARG)*
 6
 7  ARG ::= IDENTIFIER TYPE
 8
 9  BLOCK ::= "{" STATEMENTS "}"
10
11  STATEMENTS ::= STATEMENT*
12
13  STATEMENT ::= ASSIGNMENT ";"
14                | REASSIGNMENT ";"
15                | CONDITION
16                | LOOP
17                | FUNCTION_CALL ";"
18                | RETURN ";"
19                | "break" ";"
20                | "continue" ";"
21                | BLOCK
22
23  ASSIGNMENT ::= "let" IDENTIFIER TYPE "=" (EXPR | "{}")
24
25  REASSIGNMENT ::= EXPR_LVALUE "=" EXPR
26
27  EXPR_LVALUE ::= IDENTIFIER ("[" EXPR "]")*
28
29  CONDITION ::= "if" "(" EXPR ")" BLOCK ["else" BLOCK]
30
31  LOOP ::= for BLOCK
32           | for "(" ASSIGNMENT ("," ASSIGNMENT)* ";" EXPR ";"
REASSIGNMENT ("," REASSIGNMENT)* ")" BLOCK
33
34  FUNCTION_CALL ::= IDENTIFIER "(" EXPR_LIST ")"
35
36  EXPR_LIST ::= EPSILON | EXPR ("," EXPR)*
37
38  RETURN ::= "return" [EXPR]
39
40  EXPR ::= EXPR_OR
41
42  EXPR_OR ::= EXPR_AND ("||" EXPR_AND)*
43
44  EXPR_AND ::= EXPR_COMP ("&&" EXPR_COMP)*
45
46  EXPR_COMP ::= EXPR_ADD (("==" | "!=" | "<" | "<=" | ">" | ">=")
```

```
EXPR_ADD)*
47
48  EXPR_ADD ::= EXPR_MUL (("+" | "-") EXPR_MUL)*
49
50  EXPR_MUL ::= EXPR_UNARY (("*" | "/" | "%") EXPR_UNARY)*
51
52  EXPR_UNARY ::= EXPR_ATOM
53                 | "+" EXPR_UNARY
54                 | "-" EXPR_UNARY
55                 | "!" EXPR_UNARY
56
57  EXPR_ATOM ::= IDENTIFIER ("[" EXPR "]")*
58                | INTEGER
59                | "(" EXPR ")"
60                | FUNCTION_CALL
61
62  TYPE ::= int | ("[" INTEGER "]")+ int | void
```

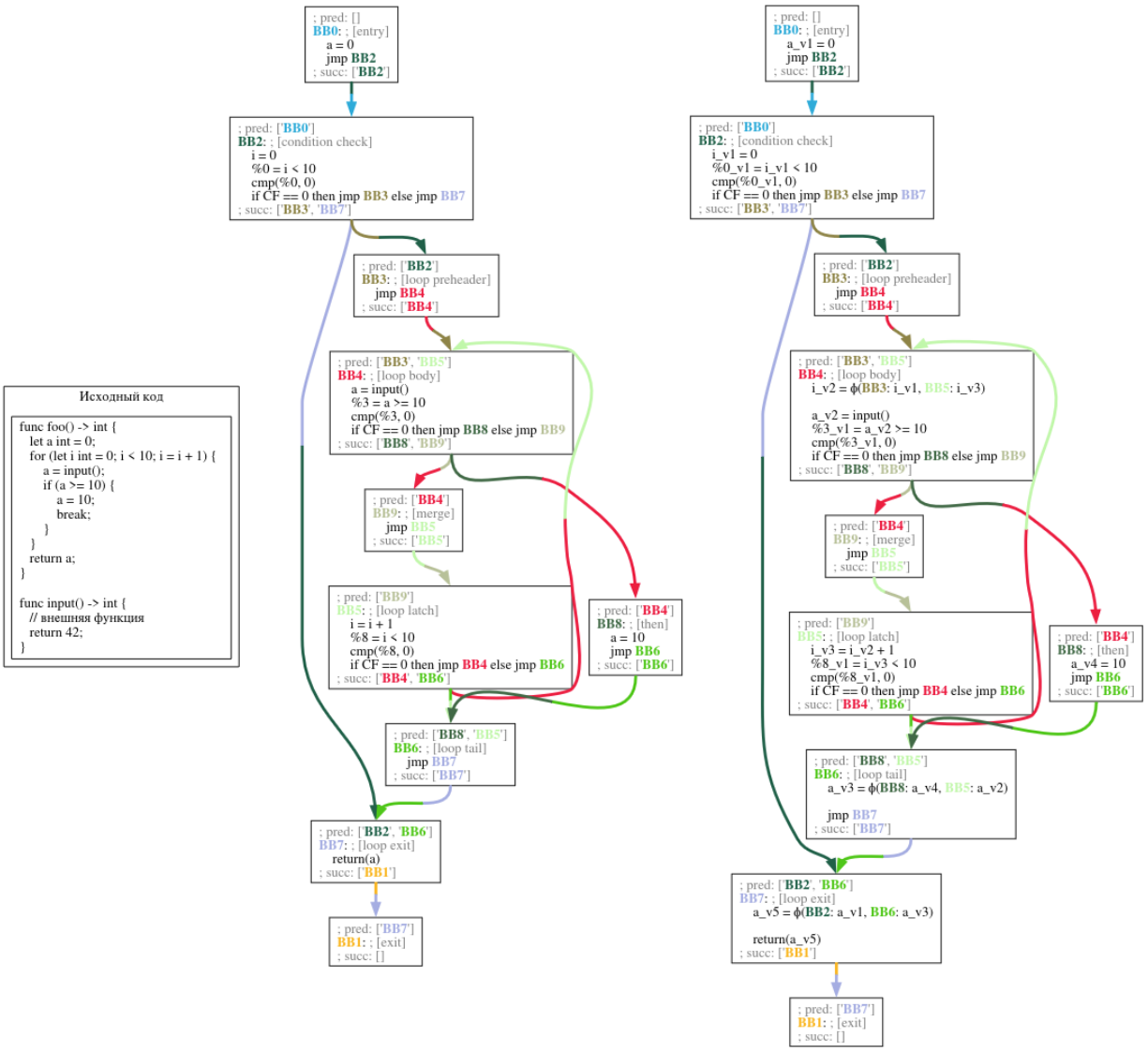

Рисунок А.1: граф потока выполнения до и после работы `SSABuilder`