СОДЕРЖАНИЕ

Лабораторная работа 7. Преобразование и анализ кода с использованием Clang и LLVM

Цель работы: Познакомиться с инструментами Clang и LLVM, научиться собирать AST и IR-промежуточное представление кода на C/C++, а также извлекать базовую информацию о программе (например, список функций).

Задачи:

1. Установить Clang и LLVM;
2. Скомпилировать простой C-файл с использованием clang и получить его: абстрактное синтаксическое дерево (AST), промежуточное представление LLVM IR;
3. Использовать opt для применения базовой комплексной оптимизации (например, O2);
4. Построить граф потока управления (CFG) для оптимизированной программы;
5. Проанализировать результат, сделать выводы и ответить на контрольные вопросы.

Список контрольных вопросов:

1. Что такое Clang, и какова его роль в процессе компиляции программ?
2. Что представляет собой LLVM и как он используется в современных компиляторах?
3. Чем отличается абстрактное синтаксическое дерево (AST) от промежуточного представления LLVM IR?
4. Для чего необходимо промежуточное представление (IR) в процессе компиляции?
5. Что делает инструкция alloc в LLVM IR, и зачем она используется в функциях?

6. Зачем нужна оптимизация кода в компиляторе, и какие основные цели она преследует?

7. Что такое SSA-форма и почему она важна при оптимизации программ?

8. Что такое граф потока управления (CFG) и как он помогает анализировать поведение программы?

9. Как устроено представление арифметических операций в LLVM IR (например, умножение, сложение)?

10. Почему функции в LLVM IR обычно представляют собой отдельные единицы анализа и оптимизации?

11. Что происходит с функцией в LLVM IR, если она вызывается один раз и очень короткая?

12. Какие преимущества даёт использование IR и CFG для автоматических оптимизаций по сравнению с анализом исходного текста на C?

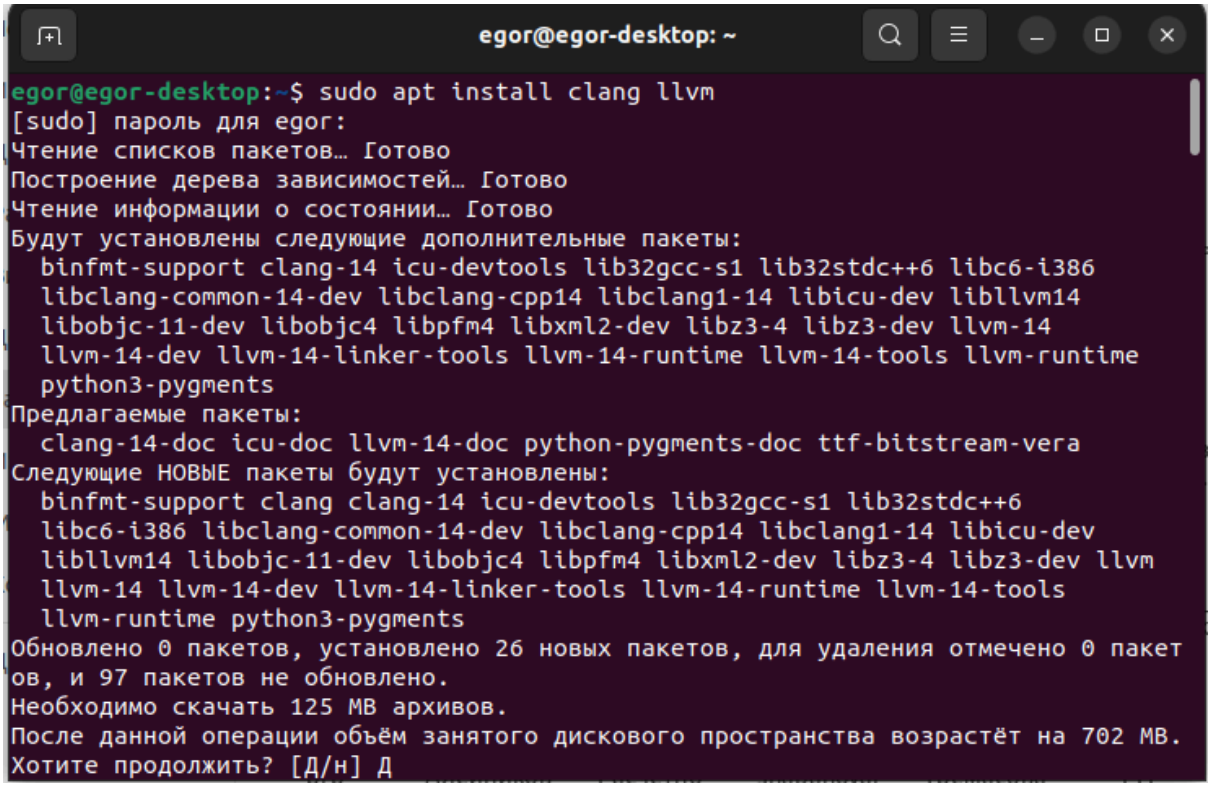
Ход работы

1. Установка и подготовка среды

Работа выполнялась в среде Ubuntu 22.04. Установлены следующие инструменты:

- clang — компилятор языка C/C++;
- llvm — инструменты анализа и оптимизации кода;
- opt — инструмент для работы с LLVM IR и применения оптимизаций;
- Graphviz — инструмент для визуализации кода.

Команда установки: sudo apt install clang llvm



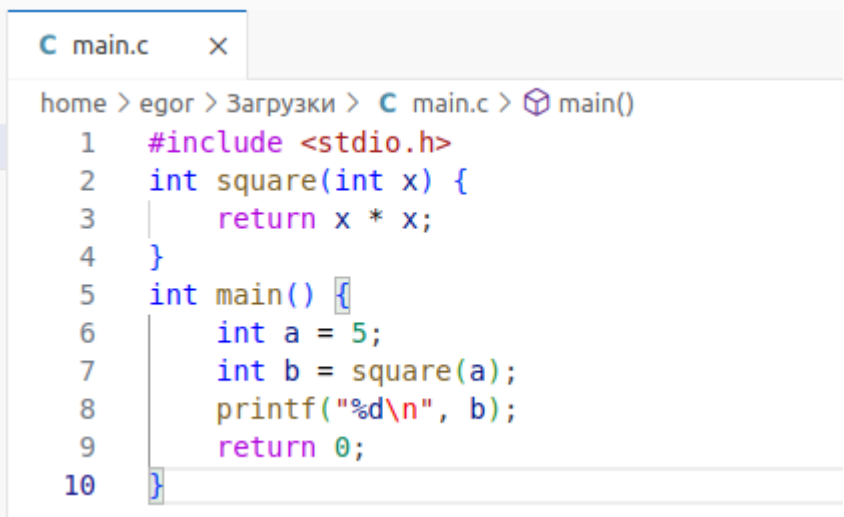
2. Исходный код

Программа на языке C:

```
#include <stdio.h>
int square(int x) {
    return x * x;
}
int main() {
```

```
    int a = 5;
    int b = square(a);
    printf("%d\n", b);
    return 0;
}
```

Сохранена в файл main.c.



3. Получение AST

Команда: clang -Xclang -ast-dump -fsyntax-only main.c

```
|-FunctionDecl 0x2c257d50 <main.c:2:1, line:4:1> line:2:5 used square 'int (int)'
| |-ParmVarDecl 0x2c257cb8 <col:12, col:16> col:16 used x 'int'
| `-CompoundStmt 0x2c257e98 <col:19, line:4:1>
|   `-ReturnStmt 0x2c257e88 <line:3:5, col:16>
|     `-BinaryOperator 0x2c257e68 <col:12, col:16> 'int' '*'
|       |-ImplicitCastExpr 0x2c257e38 <col:12> 'int' <LValueToRValue>
|       | `-DeclRefExpr 0x2c257df8 <col:12> 'int' lvalue ParmVar 0x2c257cb8 'x' 'int'
|       `-ImplicitCastExpr 0x2c257e50 <col:16> 'int' <LValueToRValue>
|         `-DeclRefExpr 0x2c257e18 <col:16> 'int' lvalue ParmVar 0x2c257cb8 'x' 'int'
`-FunctionDecl 0x2c257f00 <line:5:1, line:10:1> line:5:5 main 'int ()'
  `-CompoundStmt 0x2c258868 <col:12, line:10:1>
    |-DeclStmt 0x2c258040 <line:6:5, col:14>
    | `-VarDecl 0x2c257fb8 <col:5, col:13> col:9 used a 'int' cinit
    |   `-IntegerLiteral 0x2c258020 <col:13> 'int' 5
    |-DeclStmt 0x2c2581a0 <line:7:5, col:22>
    | `-VarDecl 0x2c258070 <col:5, col:21> col:9 used b 'int' cinit
    |   `-CallExpr 0x2c258160 <col:13, col:21> 'int'
    |     |-ImplicitCastExpr 0x2c258148 <col:13> 'int (*)(int)' <FunctionToPointerDecay>
    |     | `-DeclRefExpr 0x2c2580d8 <col:13> 'int (int)' Function 0x2c257d50 'square' 'int (int)'
    |     `-ImplicitCastExpr 0x2c258188 <col:20> 'int' <LValueToRValue>
    |       `-DeclRefExpr 0x2c2580f8 <col:20> 'int' lvalue Var 0x2c257fb8 'a' 'int'
    |-CallExpr 0x2c2587c0 <line:8:5, col:21> 'int'
    | |-ImplicitCastExpr 0x2c2587a8 <col:5> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x2c2581b8 <col:5> 'int (const char *, ...)' Function 0x2c23cd28 'printf' 'int (const char *, ...)'
    | |-ImplicitCastExpr 0x2c258808 <col:12> 'const char *' <NoOp>
    | | `-ImplicitCastExpr 0x2c2587f0 <col:12> 'char *' <ArrayToPointerDecay>
    | |   `-StringLiteral 0x2c2581d8 <col:12> 'char[4]' lvalue "%d\n"
    | `-ImplicitCastExpr 0x2c258820 <col:20> 'int' <LValueToRValue>
    |   `-DeclRefExpr 0x2c2581f8 <col:20> 'int' lvalue Var 0x2c258070 'b' 'int'
    `-ReturnStmt 0x2c258858 <line:9:5, col:12>
      `-IntegerLiteral 0x2c258838 <col:12> 'int' 0
```

Функция square принята, содержит параметр x и возвращает x * x.

4. Генерация LLVM IR

Команда: clang -S -emit-llvm main.c -o main.ll


```
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @square(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = mul nsw i32 %3, %4
  ret i32 %5
}

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 5, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = call i32 @square(i32 noundef %4)
  store i32 %5, i32* %3, align 4
  %6 = load i32, i32* %3, align 4
  %7 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 noundef %6)
  ret i32 0
}

declare i32 @printf(i8* noundef, ...) #1

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"Ubuntu clang version 14.0.0-1ubuntu1.1"}
```

5. Оптимизация IR

Команда: clang -O0 -S -emit-llvm main.c -o main_O0.ll

Стоит отметить, что в файле с IR до оптимизации:

Все переменные (a, b, x.addr) размещены в памяти через alloca;

Множество операций load и store;

square вызывается как отдельная функция.


```
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @square(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = mul nsw i32 %3, %4
  ret i32 %5
}

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 5, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = call i32 @square(i32 noundef %4)
  store i32 %5, i32* %3, align 4
  %6 = load i32, i32* %3, align 4
  %7 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 noundef %6)
  ret i32 0
}

declare i32 @printf(i8* noundef, ...) #1

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"Ubuntu clang version 14.0.0-1ubuntu1.1"}
```

Команда: clang -O2 -S -emit-llvm main.c -o main_O2.ll

Команда -O2 – комплексная оптимизация среднего уровня. Она применяет более 30 различных оптимизаций:

- -inline – встраивание небольших функций (встраивает square в main, если она вызывается один раз);
- -constprop – подставит значение square(5) → 25, если функция встроена и всё известно на этапе компиляции;
- -mem2reg – перевод переменных из памяти в регистры (SSA);
- -instcombine – объединение и упрощение инструкций (упростит арифметику, например x * x может быть преобразовано в shl при x = 2^n);
- -simplifycfg – оптимизирует структуру блоков (Упростит граф управления, если после inlining останутся лишние блоки);
- -reassociate, -gvn, -sroa, -dce и другие.

В файле с IR после оптимизации:

Вся функция square исчезла – она была встроена (-inline) и затем вычислена (оптимизация -constprop);

Никаких переменных, alloca, store, load – всё удалено (оптимизации -mem2reg, -dce);

Остался только вызов printf(25).

Команда: diff main_O0.ll main_O2.ll

Сравнение двух файлов:

```
egor@egor-desktop:~$ clang -O0 -S -emit-llvm main.c -o main_O0.ll
egor@egor-desktop:~$ clang -O2 -S -emit-llvm main.c -o main_O2.ll
egor@egor-desktop:~$ diff main_O0.ll main_O2.ll
8,15c8,11
< ; Function Attrs: noinline nounwind optnone uwtable
< define dso_local i32 @square(i32 noundef %0) #0 {
<   %2 = alloca i32, align 4
<   store i32 %0, i32* %2, align 4
<   %3 = load i32, i32* %2, align 4
<   %4 = load i32, i32* %2, align 4
<   %5 = mul nsw i32 %3, %4
<   ret i32 %5
---
> ; Function Attrs: mustprogress nofree norecurse nosync nounwind readnone uwtable willreturn
> define dso_local i32 @square(i32 noundef %0) local_unnamed_addr #0 {
>   %2 = mul nsw i32 %0, %0
>   ret i32 %2
18,29c14,16
< ; Function Attrs: noinline nounwind optnone uwtable
< define dso_local i32 @main() #0 {
<   %1 = alloca i32, align 4
<   %2 = alloca i32, align 4
<   %3 = alloca i32, align 4
<   store i32 0, i32* %1, align 4
<   store i32 5, i32* %2, align 4
<   %4 = load i32, i32* %2, align 4
<   %5 = call i32 @square(i32 noundef %4)
<   store i32 %5, i32* %3, align 4
<   %6 = load i32, i32* %3, align 4
<   %7 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 noundef %6)
---
> ; Function Attrs: nofree nounwind uwtable
> define dso_local i32 @main() local_unnamed_addr #1 {
>   %1 = tail call i32 (i8*, ...) @printf(i8* noundef nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 noundef 25)
33c20,21
< declare i32 @printf(i8* noundef, ...) #1
---
> ; Function Attrs: nofree nounwind
> declare noundef i32 @printf(i8* nocapture noundef readonly, ...) local_unnamed_addr #2
35,36c23,25
< attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "
target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
< attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+s
se2,+x87" "tune-cpu"="generic" }
---
> attributes #0 = { mustprogress nofree norecurse nosync nounwind readnone uwtable willreturn "frame-pointer"="none" "min-legal-vector-width"="0" "no-trapping-math"="
true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
> attributes #1 = { nofree nounwind uwtable "frame-pointer"="none" "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cp
u"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
> attributes #2 = { nofree nounwind "frame-pointer"="none" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+
fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
```

Стоит отметить, что после оптимизации произошли следующие изменения:

- Переменные типа alloca были удалены;
- Код переведён в SSA-форму;
- Оптимизация улучшила читаемость и упростила поток управления.

6. Граф потока управления программы

Команда для генерации оптимизированного LLVM IR: clang -O2 -S -emit-llvm main.c -o main.ll

Команда для генерации .dot-файлов CFG для функций: opt -dot-cfg -disable-output main.ll

```
egor@egor-desktop:~$ opt -dot-cfg -disable-output main.ll
egor@egor-desktop:~$ find . -name "*.dot"
./.main.dot
./.square.dot
```

Эта команда создаст DOT-файлы: .main.dot – для функции main; .square.dot – для square, если она не была удалена оптимизацией.

Команда для установки библиотеки Graphviz: sudo apt install graphviz

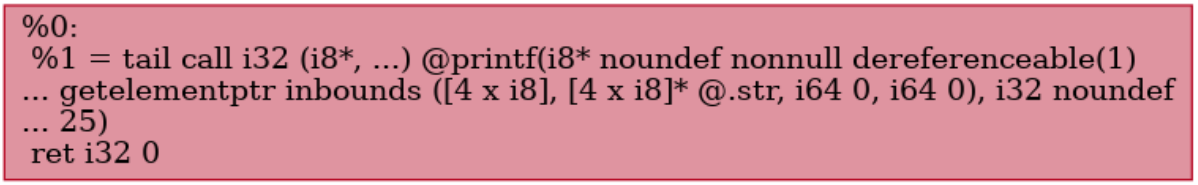
Команды для преобразования файлов с расширением .dot в .png с помощью Graphviz:

dot -Tpng .main.dot -o cfg_main.png

dot -Tpng .square.dot -o cfg_square.png

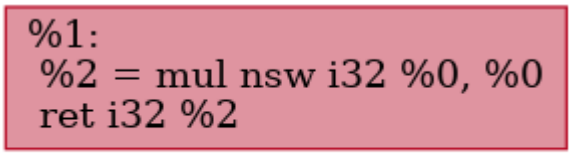
Команды для просмотра файлов с CGF:

xdg-open cfg_main.png



CFG for 'main' function

xdg-open cfg_square.png



CFG for 'square' function

Стоит отметить, что в LLVM каждый граф потока управления (CFG) строится на уровне функции, поскольку структура управления всегда локальна для тела функции. Для получения полного представления о программе, нужно построить CFG для всех функций и анализировать их совокупность. Автоматическое объединение всех CFG в один граф не предусмотрено в LLVM по умолчанию.

Выводы

- С помощью Clang можно получить полную структуру AST и IR, а также CGF;
- LLVM предоставляет гибкие инструменты анализа и оптимизации;
- Промежуточное представление кода удобно для написания компиляторных трансформаций.

Дополнительное задание

Вариант 1. Объявление комплексного числа с инициализацией на языке C++

Задание: Напишите программу, в которой создается объект типа std::complex<double> z(3.0, 4.0);, и выведите его модуль. Постройте AST и LLVM IR, проанализируйте, как компилятор обрабатывает вызов конструктора и вычисление модуля.

Вариант 2. Объявление и определение структуры на языке C

Задание: Определите структуру struct Point { int x; int y; };, инициализируй объект Point p = {2, 3};, вызовите функцию, принимающую Point как аргумент. Проанализируйте представление структуры в IR и передачу по значению.

Вариант 3. Объявление ассоциативного массива языка C++

Задание: Используйте std::map<std::string, int> с инициализацией 2–3 пар. Рассмотрите в AST и IR, как происходит вызов конструктора и вставка элементов через insert или {}.

Вариант 4. Объявление прототипа функции на языке C/C++

Задание: Создайте прототип int sum(int a, int b); и функцию, использующую его. Проанализируйте, как Clang представляет прототип в AST и как IR строит вызов.

Вариант 5. Объявление целочисленной константы с инициализацией на языке C/C++

Задание: Определите const int LIMIT = 100; и используй в условии. Проверьте, была ли подставлена константа при оптимизации -constprop или -O2.

Вариант 6. Объявление вещественной константы с инициализацией на языке C/C++

Задание: Напишите const double PI = 3.1415; и используйте в арифметике. Проанализируйте поведение с -constprop или -O2.

Вариант 7. Объявление массива символов с инициализацией строковой константой на языке C

Задание: Создайте char msg[] = "Hello"; и выведите msg[1]. Посмотрите, как Clang хранит строку в IR и как обращается к символам.

Вариант 8. Объявление перечисления на языке C++

Задание: Создайте enum Color { Red, Green, Blue }; и переменную Color c = Green;. Проанализируйте, как enum представлен в IR.

Вариант 9. Создание функции языка C/C++

Задание: Напишите функцию double average(int a, int b) и вызовите её из main(). Постройте её CFG и определите, где можно провести оптимизации.

Вариант 10. Лямбда-выражения языка C++

Задание: Создайте auto f = [](int x) { return x * x; }; и вызовите f(3). Проанализируйте, как компилятор реализует лямбду (через структуру или замыкание) и какие блоки появляются в CFG.

Вариант 11. Исследование дополнительных оптимизаций LLVM и их влияния на код

Выберите 2–3 оптимизации LLVM, не использованные в основной части лабораторной работы (например: -loop-unroll, -sroa, -licm, -dse, -indvars, -adce, -slp-vectorizer, -loop-vectorize и др.).

Прочитайте документацию по выбранным оптимизациям.

Примените их к предложенному примеру кода:

```
#include <stdio.h>
int sum_array(int* arr, int n) {
    int sum = 0;
    for (int i = 0; i < n; ++i)
        sum += arr[i];
    return sum;
}
int main() {
```

```
    int data[5] = {1, 2, 3, 4, 5};
    int total = sum_array(data, 5);
    printf("Sum = %d\n", total);
    return 0;
}
```

Сравните IR до и после оптимизации, опишите изменения.

Постройте граф потока управления (CFG) до и после оптимизации, если он изменился.

Сделайте выводы, какие преобразования были выполнены и зачем.