
Введение

В данном проекте была поставлена задача разработки оптимизирующего компилятора для языка, описываемого заданной грамматикой.

Для реализации проекта был выбран язык C#. В связи с этим было принято решение использовать генератор синтаксического анализатора Yacc и генератор лексического анализатора Lex.

Структура проекта

Проект состоит из следующих основных компонентов:

- библиотека, реализующая оптимизации;
- консольное приложение;
- реализация GUI (графический интерфейс в рамках IDE)
- проект с тестами

Содержание:

1. Парсер языка и построение AST-дерева.
2. Реализация Pretty Print визитора.
3. Структура для работы с трехадресным кодом.
4. Генерация трехадресного кода по синтаксическому дереву.
5. Выделение базовых блоков в трехадресном коде.
6. Структура графа потока управления (CFG), создание CFG по набору базовых блоков.
7. Накопление Def-Use и Use-Def информации в пределах ББл.
8. Алгоритм выделения “живых” переменных и удаление “мертвого” кода (каскадное).
9. Свертка констант, протяжка констант.
10. Применение алгебраических и логических тождеств.
11. Протяжка копий.
12. Общий алгоритм при наличии вектора оптимизаций (O1, O2, ...).
13. Представление данных для анализа потока данных (DFA).
14. Интерфейс передаточной функции.
15. Передаточная функция для достигающих определений, множества gen
16. и kill.
17. Настроить CI, авторевью кода и тесты.
18. Реализация передаточной функции композицией.
19. Итерационный алгоритм для достигающих определений.
20. Вычисление множеств Use(B) и Def(B) для активных переменных.
21. Реализация общего итерационного алгоритма.
22. Поиск множеств E_GEN и E_KILL для базового блока.
23. Итерационный алгоритм для достигающих выражений.
24. Тестирование итерационного алгоритма для достигающих выражений.
25. Генерация IL-кода.
26. Создание простого аналога среды разработки (IDE).
27. Оптимизация “Распространение констант” между базовыми блоками.
28. Базовые структуры и итерационный алгоритм для распространения констант.
29. Нумерация ББл в порядке “Обращение обратного порядка обхода”.
30. Проверка CFG на приводимость.
31. Расчет глубины CFG.
32. Модифицировать итерационный Алгоритм с нумерацией базовых блоков и подсчётом количества итераций.
33. Примеры неправильных нумераций с max кол-вом итераций.

34. Обобщённый итерационный алгоритм.
35. Классификация рёбер CFG.
36. Перемещение определения как можно ближе к использованию.
37. Оптимизация общих подвыражений.
38. Реализация Def и Use множеств.
39. Реализация итерационного алгоритма для активных переменных.
40. Удаление мёртвого кода между базовыми блоками.
41. Построение дерева доминаторов.

Название задачи

Парсер языка и построение AST-дерева

Постановка задачи

Написать парсер языка на языке C# с использованием GPLex и Yacc.
Реализовать построение синтаксического дерева программы.

Зависимости задач в графе задач

От задачи зависит:

- Базовые визиторы
- PrettyPrinter
- Генерация трёхадресного кода

Теоретическая часть задачи

Для решения данной задачи необходимо реализовать две составляющие: лексер и парсер языка.

Опр. Лексический анализатор (лексер) — это программа или часть программы, выполняющая лексический анализ. Лексер предназначен для разбиения входного потока символов на лексемы - отдельные, осмысленные единицы программы.

Основные задачи, которые выполняет лексер:

- Выделение идентификаторов и целых чисел

- Выделение ключевых слов
- Выделение символьных токенов

Опр. Парсер (или синтаксический анализатор) — часть программы, преобразующей входные данные (как правило, текст) в структурированный формат. Парсер выполняет синтаксический анализ текста. Парсер принимает на вход поток лексем и формирует абстрактное синтаксическое дерево (AST).

Практическая часть задачи (реализация)

Для автоматического создания парсера создаются файлы SimpleLex.lex (описание лексического анализатора) и SimpleYacc.y (описание синтаксического анализатора).

Код лексического и синтаксического анализаторов создаются на C# запуском командного файла generateParserScanner.bat.

Синтаксически управляемая трансляция состоит в том, что при разборе текста программы на каждое распознанное правило грамматики выполняется некоторое действие. Данные действия придают смысл трансляции (переводу) и поэтому мы называем их семантическими. Семантические действия записываются в .y-файле после правил в фигурных скобках и представляют собой код программы на C# (целевом языке компилятора).

Как правило, при трансляции программа переводится в другую форму, более приспособленную для анализа, дальнейших преобразований и генерации кода.

Мы будем переводить текст программы в так называемое синтаксическое дерево. Если синтаксическое дерево построено, то программа синтаксически правильная, и ее можно подвергать дальнейшей обработке.

В синтаксическое дерево включаются узлы, соответствующие всем синтаксическим конструкциям языка. Атрибутами этих узлов являются их существенные характеристики. Например, для узла оператора присваивания `AssignNode` такими атрибутами являются `IdNode` - идентификатор в левой части оператора присваивания и `ExprNode` - выражение в правой части оператора присваивания.

Парсер языка

Парсер был реализован для языка со следующим синтаксисом:

```
a = 777; // оператор присваивания
```

```
// пример арифметических операций
```

```
a = a - b;
```

```
a = a + b;
```

```
a = a * 3;
```

```
a = 5 * b;
```

```
// пример операторов сравнения
```

```
c = a < b;
```

```
c = b > a;
```

```
c = a != b;  
c = a == b;  
// логическое "нет"  
c = !a;
```

```
// полная форма условного оператора  
if (a < b)  
    a = 555;  
else  
{  
    b = 666;  
    c = 777;  
}  
// сокращенная форма условного оператора  
if (b == c)  
    c = 666;
```

```
// операторы циклов  
// цикл while  
while(3)  
{  
    ...  
}  
// цикл for с шагом 2  
for(i = 0, 10, 2)  
{  
    ...  
}
```

```
}  
// цикл for с шагом 1 по умолчанию  
for(i = 0, 10)  
{  
    ...  
}
```

```
// оператор вывода  
print(a);  
print(a, b, c);
```

```
// оператор goto  
goto h;  
// переход по метке  
h: {c = a + b;}
```

Для создания парсера использовались GPLex и Yacc, были созданы соответствующие файлы .lex и .y.

Пример содержимого [.lex файла](#):

```
{ID} {  
    int res = ScannerHelper.GetIDToken(yytext);  
    if (res == (int)Tokens.ID)  
        yylval.sVal = yytext;  
    return res;  
}
```



```
"=" { return (int)Tokens.ASSIGN; }  
";" { return (int)Tokens.SEMICOLON; }
```

Пример содержимого [.y файла](#):

```
%token <iVal> INUM  
%token <sVal> ID  
  
%type <eVal> expr ident W T F  
%type <stVal> assign statement cycle for if  
  
expr : W { $$ = $1; }  
      | expr LT W { $$ = new BinaryNode($1, $3, OperationType.  
Less); }  
      | expr GT W { $$ = new BinaryNode($1, $3, OperationType.  
Greater); }  
      | expr LE W { $$ = new BinaryNode($1, $3, OperationType.  
LessEq); }  
      | expr GE W { $$ = new BinaryNode($1, $3, OperationType.  
GreaterEq); }  
      | expr EQ W { $$ = new BinaryNode($1, $3, OperationType.  
Equal); }  
      | expr NEQ W { $$ = new BinaryNode($1, $3, OperationType.  
NotEqual); }  
      ;
```

Построение AST-дерева

Для построения AST дерева были созданы классы для каждого типа узла:

- [Node.cs](#) - базовый класс для всех узлов
- [ExprNode.cs](#) - базовый класс для выражений
- [AssignNode.cs](#) - операция присваивания
- [BinaryNode.cs](#) - класс для бинарных операций
- [UnaryNode.cs](#) - класс для унарных операций
- [ExprListNode.cs](#) - класс для списка операций
- [IntNumNode.cs](#) - класс для целочисленных констант
- [IdNode.cs](#) - класс для идентификаторов
- [StatementNode.cs](#) - базовый класс для всех операторов
- [BlockNode.cs](#) - класс для блока
- [CycleNode.cs](#) - класс для цикла *while*
- [ForNode.cs](#) - класс для цикла *for*

- [GoToNode.cs](#) - класс для *goto*
- [IfNode.cs](#) - класс для оператора сравнения
- [LabeledNode.cs](#) - класс метки goto
- [PrintNode.cs](#) - класс оператора вывода
- [EmptyNode.cs](#) - класс для пустого узла

Пример кода, описывающего оператор вывода:

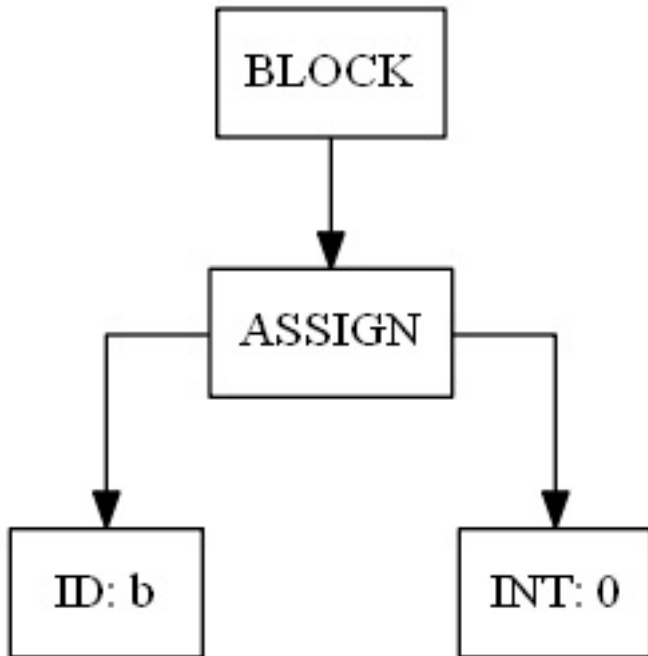
```
namespace Compiler.Parser.AST
{
    public class PrintNode : StatementNode
    {
        public ExprListNode ExprList { get; set; }

        public PrintNode(ExprListNode exprlist)
        {
            ExprList = exprlist;
        }
    }
}
```

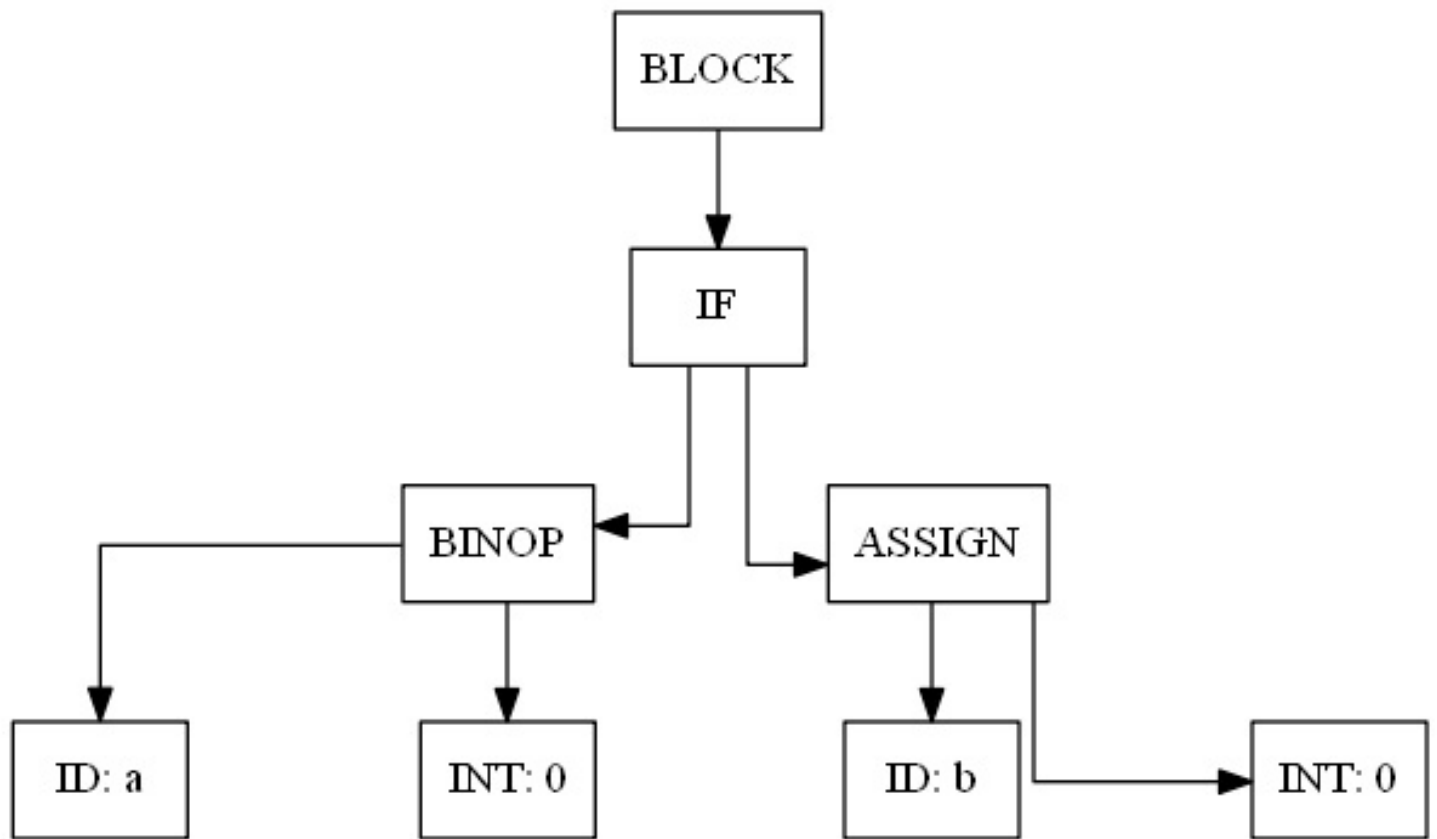
Тесты

1.

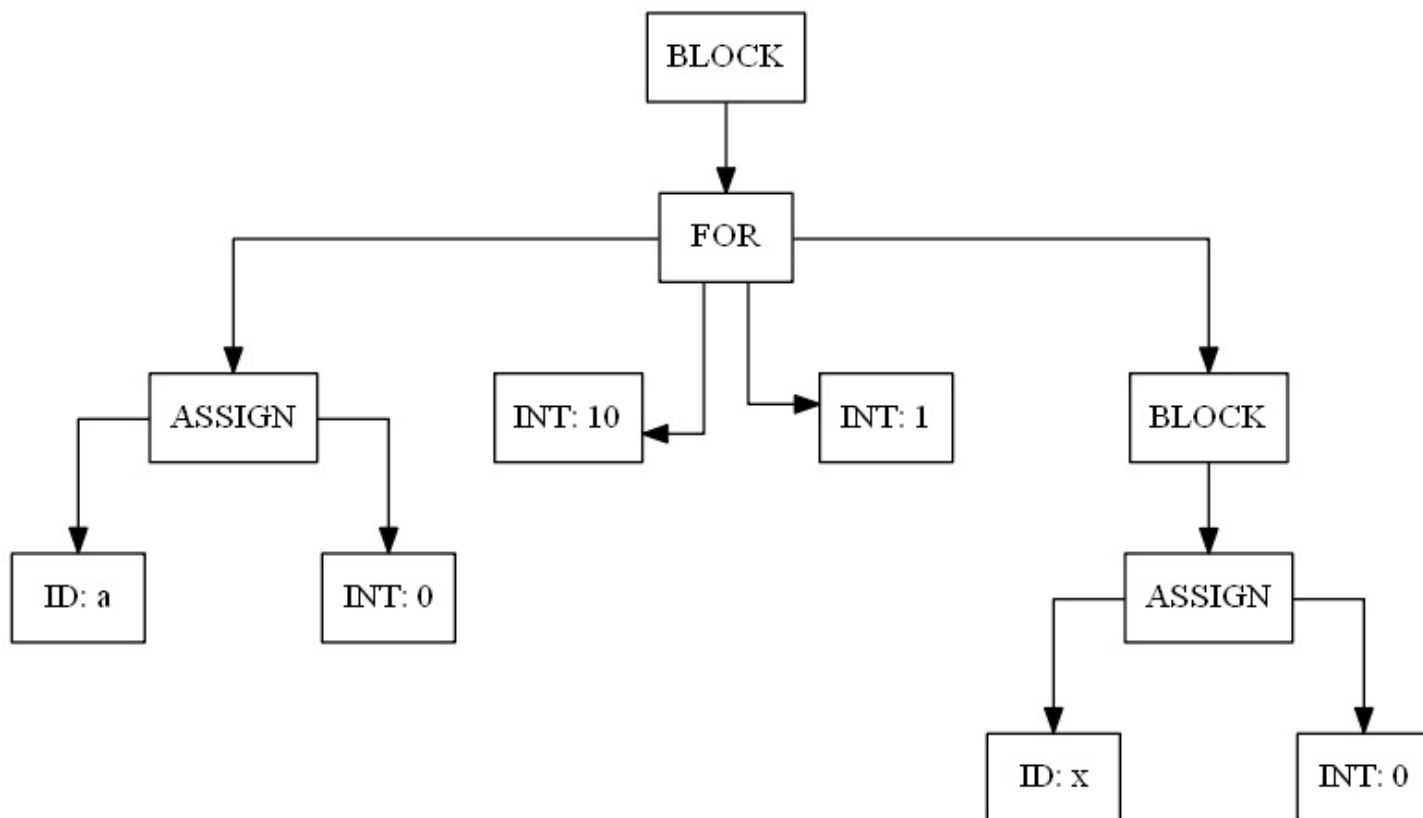
```
b = 0;
```



```
if(a == 0)
    b = 0;
```



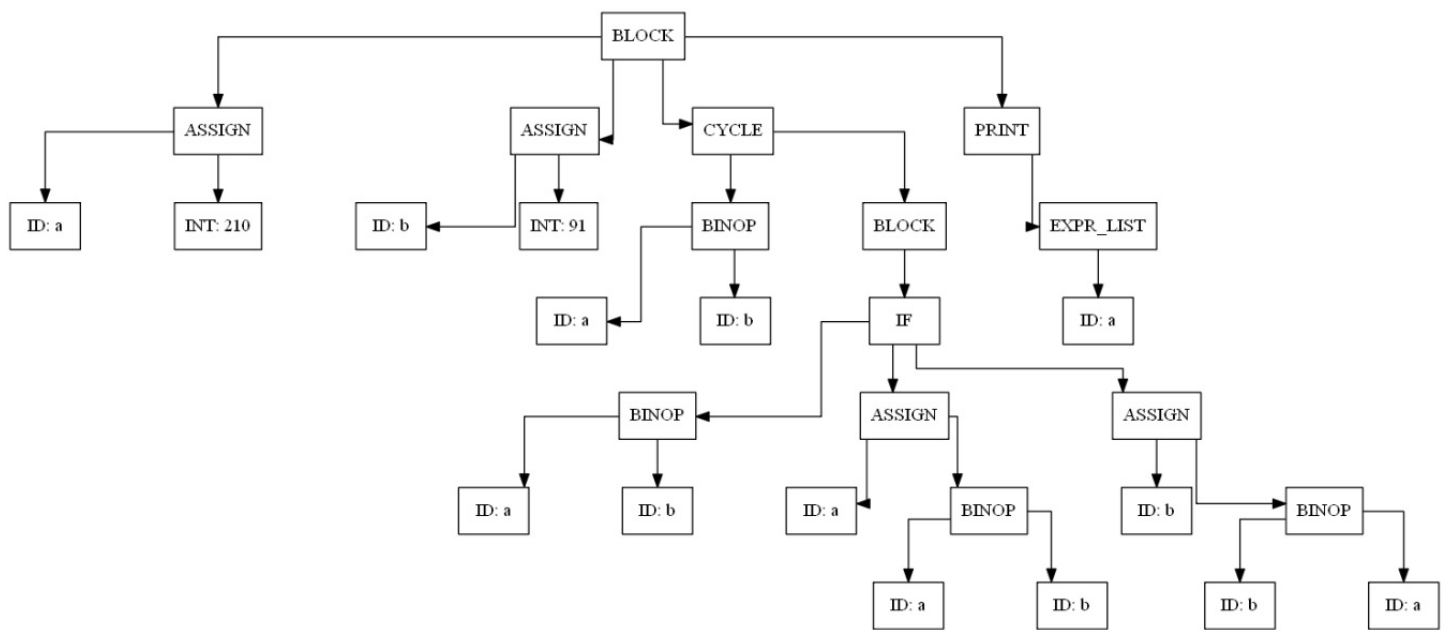
```
for(a = 0, 10, 1) {  
    x = 0;  
}
```



Пример работы.

Алгоритм Евклида.

```
a = 210;  
b = 91;  
  
while (a != b)  
{  
    if (a > b)  
        a = a - b;  
    else  
        b = b - a;  
}  
print (a);
```



Название задачи

Реализация Pretty Print визитора

Постановка задачи

Необходимо реализовать Pretty Print визитор, который по AST восстанавливает отформатированный исходный код программы.

Зависимости задач в графе задач

Зависит от:

- AST-дерево

Теоретическая часть задачи

Для восстановления исходного кода программы по AST будем совершать обход дерева, накапливая код в поле визитора `Text`, при этом учитывая тип посещаемого узла.

Для этого класс `PrettyPrintVisitor` реализует интерфейс `Ivisitor`. Отступы поддерживаются с помощью переменной `Indent`, которая увеличивается на 2 при входе в блок и уменьшается на 2 перед выходом из блока.

Практическая часть задачи (реализация)

Набор методов класса `PrettyPrintVisitor`, реализующих `Ivisitor`:

0. VisitIdNode(IdNode id)
1. VisitIntNumNode(IntNumNode num)
2. VisitUnaryNode(UnaryNode unop)
3. VisitBinaryNode(BinaryNode binop)
4. VisitAssignNode(AssignNode a)
5. VisitCycleNode(CycleNode c)
6. VisitBlockNode(BlockNode bl)
7. VisitPrintNode(PrintNode p)
8. VisitGoToNode(GoToNode g)
9. VisitLabeledNode(LabeledNode l)
10. VisitExprListNode(ExprListNode el)
11. VisitIfNode(IfNode iif)
12. VisitForNode(ForNode w)
13. VisitEmptyNode(EmptyNode w)
14. VisitExprNode(ExprNode s)

Пример реализации `VisitExprListNode` :

```
public override void VisitExprListNode(ExprListNode el) {
    var last = el.ExprList.Last();
    el.ExprList.ForEach(expr =>
    {
        expr.Visit(this);
        if (expr != last)
            Text += ",";
    });
}
```

Пример работы.

Исходный код:

```
if (1 < -3)
    a = 123;
else
    goto h;
for(i = 0, 10)
{
    print(1 >= 3);
    if (1 + 3)
    {
        a = 1;
    }
}
h: {c = a + b;}
```

Исходный код, восстановленный через PrettyPrintVisitor:

```
if ((1 < - 3))
    a = 123;
else
    goto h;
for(i = 0,10,1)
{
    print((1 >= 3));
```

```
if ((1 + 3))  
{  
    a = 1;  
}  
  
h:  
{  
    c = (a + b);  
}
```

Название задачи:

Структура для работы с трехадресным кодом

Постановка задачи:

Реализовать структуру для работы с трехадресным кодом.

Зависимости задач в графе задач

Зависит от:

- AST

От задачи зависит:

- генерация трехадресного кода по синтаксическому дереву
- базовые блоки

Теоретическая часть задачи

Трехадресный код - это последовательность операторов вида:

- `x = op z`
- `goto L`
- `if x goto L`
- `nop`
- `x = y op z`

- `x = y`
- `print`

где `x`, `y`, `z` - имена, константы или сгенерированные компилятором временные объекты, `L` - метка, `op` - бинарный оператор, `print`, `goto` - соответствующие операторы языка.

Трехадресный код представляет собой линеаризованное представление синтаксического дерева или ориентированного ациклического графа, в котором явные имена соответствуют внутренним узлам графа.

Практическая часть задачи (реализация)

Трехадресный код представляет собой иерархию классов:

- Базовый класс строки кода, характеризуется уникальным идентификатором
 - пустая строка (пор, возможно помеченный)
 - оператор печати
 - оператор присваивания (основной оператор кода, может быть унарным и бинарным)
 - оператор безусловного перехода (`goto label`)
 - оператор условного перехода (`if cond goto label`)

Также присутствует набор служебных классов:

- выражения-операнды
 - константы (целые числа)
 - переменные (именованные и не очень)

- декоратор идентификатора - позволяет представить идентификатор(метку) строки в удобном для пользователя виде на печати или при отладке

Пример 1

```
a = 3;  
b = 1 + 1;  
c = a * b;  
print(c);
```

Вывод:

```
l1 : t0 = 3  
l0 : a = 3  
l4 : t2 = 1  
l5 : t3 = 1  
l3 : t1 = 2  
l2 : b = 2  
l8 : t5 = 3  
l9 : t6 = 2  
l7 : t4 = 6  
l6 : c = 6  
l11 : t7 = 6  
l10 : print t7
```

Пример 2

```
for(i = 0, 3, 1 + 1)
    print(1, 2, 3);
```

Вывод:

```
l0 : t0 = 0
l2 : t2 = 1
l3 : t3 = 1
l1 : t1 = 2
l4 : nop
l6 : t4 = t1 >= 0
l7 : if t4 goto l12
l9 : t6 = 3
l8 : t5 = t0 <= 3
l10 : if t5 goto l30
l11 : goto l17
l12 : nop
l15 : t8 = 3
l14 : t7 = t0 >= 3
l16 : if t7 goto l30
l17 : nop
l20 : t9 = 1
l19 : print t9
l22 : t10 = 2
l21 : print t10
```

```
l24 : t11 = 3
```

```
l23 : print t11
```

```
l26 : t13 = 1
```

```
l27 : t14 = 1
```

```
l25 : t12 = 2
```

```
l28 : t0 = t0 + 2
```

```
l29 : goto l4
```


Название задачи:

Генерация трехадресного кода по синтаксическому дереву

Постановка задачи:

Релизовать генератор трехадресного кода по синтаксическому дереву.

Зависимости задач в графе задач

Зависит от:

- AST
- Структура для работы с ТА кодом

От задачи зависит:

- Базовые блоки

Теоретическая часть задачи

Трехадресный код - это последовательность операторов вида:

- `x = op z`
- `goto L`
- `if x goto L`
- `nop`
- `x = y op z`

- `x = y`
- `print`

где `x`, `y`, `z` - имена, константы или сгенерированные компилятором временные объекты, `L` - метка, `op` - бинарный оператор, `print`, `goto` - соответствующие операторы языка.

Трехадресный код представляет собой линеаризованное представление синтаксического дерева или ориентированного ациклического графа, в котором явные имена соответствуют внутренним узлам графа.

Практическая часть задачи (реализация)

Генератор трехадресного кода был выполнен с использованием паттерна *визитор*. С его помощью обходили исходное AST рекурсивно.

Входные данные:

- Синтаксическое дерево

Выходные данные:

- Список команд трехадресного кода

Используемые структуры данных:

- Структура для ТА кода

Пример работы 1

```
a = 3;  
b = 1 + 1;  
c = a * b;  
print(c);
```

Вывод:

```
l1 : t0 = 3  
l0 : a = 3  
l4 : t2 = 1  
l5 : t3 = 1  
l3 : t1 = 2  
l2 : b = 2  
l8 : t5 = 3  
l9 : t6 = 2  
l7 : t4 = 6  
l6 : c = 6  
l11 : t7 = 6  
l10 : print t7
```

Пример работы 2

```
for(i = 0, 3, 1 + 1)  
    print(1, 2, 3);
```

Вывод:

```
l0 : t0 = 0
l2 : t2 = 1
l3 : t3 = 1
l1 : t1 = 2
l4 : nop
l6 : t4 = t1 >= 0
l7 : if t4 goto l12
l9 : t6 = 3
l8 : t5 = t0 <= 3
l10 : if t5 goto l30
l11 : goto l17
l12 : nop
l15 : t8 = 3
l14 : t7 = t0 >= 3
l16 : if t7 goto l30
l17 : nop
l20 : t9 = 1
l19 : print t9
l22 : t10 = 2
l21 : print t10
l24 : t11 = 3
l23 : print t11
l26 : t13 = 1
l27 : t14 = 1
l25 : t12 = 2
l28 : t0 = t0 + 2
l29 : goto l4
```

Название задачи

Выделение базовых блоков в трехадресном коде.

Постановка задачи

Предоставить возможность получить множество базовых блоков по исходному коду программы в формате трехадресного кода.

Зависимости задач в графе задач

- Все оптимизации на уровне базовых блоков
- Построение графа потока управления

Теоретическая часть задачи

Базовый блок – последовательность инструкций или кода, имеющая одну точку входа, одну точку выхода и не содержащая инструкций передачи управления ранее точки выхода. Другими словами, это последовательность инструкций, каждая из которых выполняется тогда и только тогда, когда выполняется первая инструкция из последовательности.

На начало базового блока может указывать одновременно несколько инструкций перехода, конец же блока — либо инструкция передачи управления, либо инструкция, предшествующая переходу.

Базовые блоки являются основной единицей кода, над которой проводятся оптимизации компилятором. Также они являются

вершинами в графе потока управления.

Практическая часть задачи (реализация)

- Был реализован класс `BasicBlock`, который представляет собой простую обертку над массивом узлов трехадресного кода. Дополнительная возможность, которые предоставляет класс – идентификация блока с помощью поля ID.
- Был реализован метод построения списка базовых блоков программы как часть класса `TACode`.
- При создании базового блока у команды, входящей в этот базовый блок, обновляется вспомогательное поле `Block`.

Основная работа алгоритма состоит в поиске лидеров. Лидер – это точка входа в базовый блок.

```
FindLeaders(CodeList)
  add(leaders, CodeList[0])
  for i = 1 to size(CodeList) do
    node = CodeList[i]

    if node is labeled and i not in leaders then
      add(leaders, i)
    if node is GoTo then
      add(leaders, i+1)
```

По полученному списку лидеров можно сделать разбиение на диапазоны участков кода. Т.е. из набора лидеров вида `[a0, a1, a2,`

`a3, ...]` получается набор лидеров вида `[(a0, a1), (a1, a2), (a2, a3), ...]`. При таком подходе для правильной группировки пар требуется добавить последнюю команду в список лидеров:

```
add(leaders, size(CodeList)).
```

Для каждой полученной пары генерируется базовый блок, в который помещаются все команды из диапазона `[a_i, a_{i+1})`

Название задачи

Структура графа потока управления (CFG), создание CFG по набору базовых блоков.

Постановка задачи

Предоставить возможность получить граф потока управления по набору базовых блоков.

Зависимости задач в графе задач

- Все задачи анализа на уровне CFG
- Все оптимизации на уровне CFG
- IDE

Теоретическая часть задачи

Граф потока управления – множество всех возможных путей исполнения программы, представленное в виде графа.

В графе потока управления каждый узел графа соответствует базовому блоку.

В графе потока управления устанавливаются связи между подряд идущими базовыми блоками, кроме случая, когда базовый блок завершается инструкцией перехода, а также между блоками в которых есть безусловный переход и блоками, куда этот переход ведет.

Практическая часть задачи (реализация)

- Был реализован класс `ControlFlowGraph`, который представляет собой простую обертку над массивом базовых блоков.
- В класс `BasicBlock` были добавлены структуры для возможности установления связей между блоками.
- Был реализован метод построения связей между массивом базовых блоков.

Основная работа алгоритма состоит из двух этапов. На первом этапе происходит поиск `GoTo` (безусловных переходов) в последней строчке базового блока.

```
ConnectBasicBlocks(BBList)
  for i = 0 to size(BBList)
    bb = BBList[i]
    if lastLine(bb) is GoTo
      last = lastLine(bb)
      target = getTarget(last)
      targetNode = getNode(target)
      connect(bb, targetNode)
```

Таким образом мы устанавливаем явные связи между блоками с безусловными переходами и блоками, в которые эти переходы ведут.

Во второй части алгоритма происходит установление связей между идущими подряд базовыми блоками.

```
ConnectChainedBasicBlocks(BBList)
  for i = 0 to size(BBList) - 1
    current = BBList[i]
    next = BBList[i+1]
    if lastLine(cur) is GoTo
      continue
    connect(current, next)
```

Если в последней строке есть безусловный переход, то связи между блоками быть не может в силу семантики оператора **GoTo**.

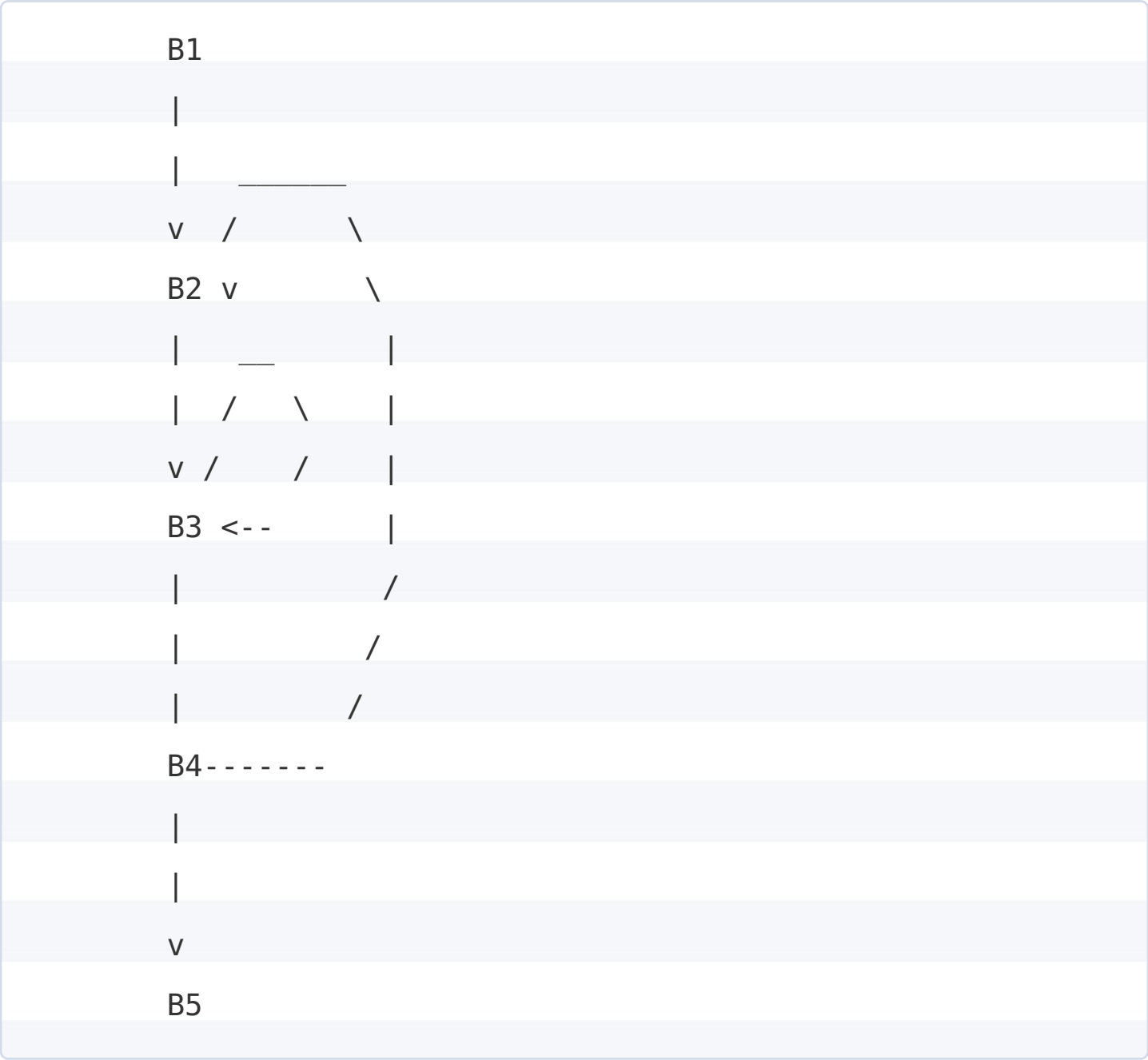
Тесты

Трёхадресный код

1) t1 = 3 - 5	B1
2) t2 = 10 + 2	B2
3) t3 = -1	B3
4) if 1 goto 3)	
5) t4 = t3 + 1999	B4
6) if 2 goto 2)	
7) t5 = 7 * 4	B5
8) t6 = 100 / 2	

Пример работы

Граф потока управления



Название задачи

Накопление Def-Use и Use-Def информации в пределах ББл

Постановка задачи

Для базового блока необходимо реализовать Def-Use и Use-Def цепочки

Зависимости задач в графе задач

Зависит от:

- Трехадресный код

Теоретическая часть задачи

Будем разделять переменные на два типа: def-переменные и use-переменные.

Def-переменная - это определение переменной, когда ей присваивается какое-либо значение.

Use-переменная - это использование переменной, т.е. где-то в программе переменной было присвоено значение, а теперь нас интересует только значение нашей переменной.

Построение этих цепочек будем вести по следующим командам в трехадресном коде:

```
1) Assign: result = left op right
```

В данном варианте, result будет являться def-переменной, а left и right -

use-переменной

(при условии, что left и right - это переменные, которые были определены ранее в коде).

```
2) ifGoto: if (x) goto L1
```

В данном варианте переменная x - будет являться use-переменной (при условии, что x - это переменные, которые были определены ранее в коде).

```
3) Print: print(x)
```

В данном варианте переменная x - будет являться use-переменной (при условии, что x - это переменные, которые были определены ранее в коде).

Структура Def-Use цепочки:

```
0. a = 5
1. x = a
2. a = 4
3. x = 3
4. y = x + 7
5. print(y)
6. print(x)
```

Практическая часть задачи (реализация)

```
    /// Def цепочка
    /// </summary>
    public DList DList { get; }
    /// <summary>
    /// Use цепочка
    /// </summary>
    public UList UList { get; }
    ...
    /// Добавляет Use переменную в DList
    private void AddUseVariable(Expr expr, Guid strId)
    {
        if (expr is Var)
        {
            var variable = expr as Var;

            // Поиск последнего переопределения переменн
            // о
            var index = DList.FindLastIndex(v =>
            {
                return v.DefVariable.Name.Id == variable.
                Id;
            });

            var UVar = new DUVar(variable, N[strId]);

            // Добавление Use переменной
```

```

        if (index != -1)
            DList[index].AddUseVariables(UVar);
        else
            UListNotValid.Add(UVar);
    }
}
...
/// Создает Use цепочку для базового блока
private void BuildUList()
{
    foreach (var dN in DList)
    {
        var dVar = dN.DefVariable;

        foreach (var uVar in dN.UseVariables)
        {
            UList.Add(new UNode(uVar as DUVar, dVar as DUVar));
        }
    }
}
...
/// Добавляет Use переменные
/// <param name="UseVariables">Список используемых переменных</param>
public void AddUseVariables(params DUVar[] UseVariables)
{

```

```

        this.UseVariables.AddRange(UseVariables.ToList())
;
    }

    /// Удаляет Use переменные
    /// <param name="UseVariables">Список используемых пе
    ременных</param>
    public void RemoveUseVariables(params DUVar[] UseVari
    ables)
    {
        for (var i = 0; i < this.UseVariables.Count; i++)
            if (UseVariables.Contains(this.UseVariables[i
            ]))

                this.UseVariables.RemoveAt(i--);
    }

```

Тесты

```

DULists DL = new DULists(B);

// Use цепочка
var useList = new List<UNode> {
    new UNode(new DUVar(a, 1), new DUVar(a, 0)),
    new UNode(new DUVar(x, 2), new DUVar(x, 1)),
    new UNode(new DUVar(x, 3), new DUVar(x, 1)),
    new UNode(new DUVar(a, 3), new DUVar(a, 0)),
    new UNode(new DUVar(y, 4), new DUVar(y, 2)),
    new UNode(new DUVar(a, 5), new DUVar(a, 3))
}

```



```
};
```

```
// Def цепочка
```

```
var defList = new List<DNode> {
```

```
new DNode(new DUVar(a, 0),
          new List<DUVar> {
            new DUVar(a, 1),
            new DUVar(a, 3)
          })
```

```
new DNode(new DUVar(x, 1),
          new List<DUVar> {
            new DUVar(x, 2),
            new DUVar(x, 3)
          })
```

```
new DNode(new DUVar(y, 2),
          new List<DUVar> {
            new DUVar(y, 4)
          }
        ),
```

```
new DNode(new DUVar(a, 3),
          new List<DUVar> {
              new DUVar(a, 5)
          }
        ),
```

};

```
var IsUse = useList.Count == DL.UList.Count;
var IsDef = defList.Count == DL.DList.Count;

if (IsUse)
    for (var i = 0; i < DL.UList.Count; i++)
        IsUse &= useList.Contains(DL.UList[i]);

if (IsDef)
    for (var i = 0; i < DL.DList.Count; i++)
        IsDef &= defList.Contains(DL.DList[i]);
```

Название задачи

Алгоритм выделения “живых” переменных и удаление “мертвого” кода (каскадное)

Зависит от:

- Def/Use Lists

Постановка задачи

Необходимо реализовать алгоритм определения “живых” переменных и удаления “мертвого” кода

Описание

Основная идея заключается в следующем: осуществляется проход по элементам трехадресного кода в рамках одного базового блока.

Переменная может быть активной или неактивной, изначально все переменные считаются активными, поскольку они могут быть использованы в других блоках. Удаляем элементы трехадресного кода, которые являются присваиванием. Переменная, которой присваивается значение, является неактивной. Если же переменная используется в правой части, она помечается как активная.

На основе Def цепочек строятся живые/мертвые переменные. Если переменная в Def цепочки определена, но нигде не используется, то такая переменная объявляется мертвой. В противном случае она становится живой.

Входные данные

Базовый блок

Выходные данные

Списки Живых/Мертвых переменных

Базовый блок без мертвого кода

Ниже представлен фрагмент программы, осуществляющий удаление “мертвого” кода:

```
do
{
    // Вычисляем CFG
    cfg = CreateCFG(code);
    // Вычисляем OUT переменные для всех блоков в
CFG
    this.OUT = (new IterativeAlgorithmAV(cfg)).OUT;
T;

    countRemove = 0;

    // Для каждого блока в cfg
    foreach (var B in cfg.CFGNodes)
    {
        // Удаляем мертвые строки кода
        var newB = RemoveDeadCodeInBlock(B);
        var curCountRem = B.CodeList.Count() - ne
```

```

wB.CodeList.Count();

        if (curCountRem != 0)
        {
            var idxStart = CalculateIdxStart(cfg,
            cfg.IndexOf(B).Value) - countRemove;

            var len = B.CodeList.Count();
            code = ReplaceCode(code, newB.CodeLis
            t.ToList(), idxStart, len);

            countRemove += curCountRem;
        }
    }
}

```

Тесты

Исходный код:

```

// 0: a = 5
// 1: x = a
// 2: a = 4
// 3: x = 3
// 4: y = x + 7
// 5: print(y)
// 6: print(x)

```

Результат:

Step 1:

```
deadVariable = { (x, 1), (a, 2) }
```

```
liveVariable = { (a, 0), (a, 1), (x, 3), (x, 4), (x, 6), (y, 4), (y, 5) }
```

Код:

```
// 0: a = 5
// 1: x = 3
// 2: y = x + 7
// 3: print(y)
// 4: print(x)
```

Step 2:

```
deadVariable = { (a, 0) }
liveVariable = { (x, 1), (x, 2), (y, 2), (y, 3), (x, 4) }
```

Код:

```
// 0: a = 5
// 1: x = 3
// 2: y = x + 7
// 3: print(y)
// 4: print(x)
```

Step 3:

```
deadVariable = { }
liveVariable = { (x, 0), (x, 1), (y, 1), (y, 2), (x, 3) }
```

Код:

```
// 0: x = 3
// 1: y = x + 7
// 2: print(y)
// 3: print(x)
```

Название задачи

Свертка констант, протяжка констант

Постановка задачи

Создать класс, реализующий итерационный алгоритм для задачи распространения констант, и класс, реализующий свертку констант.

Зависимости задач в графе задач

Зависит от:

- Трехадресный код

От задачи зависит:

- Общий алгоритм при наличии вектора оптимизаций (O1, O2, ...)

Теоретическая часть задачи

Распространение констант – хорошо известная проблема глобального анализа потока данных. Цель распространения констант состоит в обнаружении величин, которые являются постоянными при любом возможном пути выполнения программы, и в распространении этих величин так далеко по тексту программы, как только это возможно. Выражения, чьи операнды являются константами, могут быть вычислены на этапе компиляции. Поэтому использование алгоритмов распространения констант позволяет компилятору выдавать более компактный и быстрый код.

Рассмотрим следующий пример:

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

Распространение x возвращает:

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

Далее, свёртка констант и распространение y возвращают следующее:

```
int x = 14;
int y = 0;
return 0;
```

Практическая часть задачи (реализация)

Были реализованы классы `ConstantFolding` и `ConstantPropagation`.

```
public class ConstantFolding : IOptimization
{
    private bool SetNode(Assign node)
    {
```



```
switch (node.Operation)
{
    case OpCode.Plus:
        node.Right = (IntConst)node.Left + (IntConst)node.Right;
        node.Operation = OpCode.Copy;
        node.Left = null;
        break;
    case OpCode.Minus:
        node.Right = (IntConst)node.Left - (IntConst)node.Right;
        node.Operation = OpCode.Copy;
        node.Left = null;
        break;
    case OpCode.Mul:
        node.Right = (IntConst)node.Left * (IntConst)node.Right;
        node.Operation = OpCode.Copy;
        node.Left = null;
        break;
    case OpCode.Div:
        node.Right = (IntConst)node.Left / (IntConst)node.Right;
        node.Operation = OpCode.Copy;
        node.Left = null;
        break;
    default:
        throw new ArgumentOutOfRangeException();
}
```

```

        }
        return true;
    }
    public List<Node> Optimize(List<Node> nodes, out bool
applied)
    {
        var app = false;
        var enumerable = nodes.OfType<Assign>().Where(assign => assign.Left is IntConst && assign.Right is IntConst);

        foreach (var node in enumerable)
            app = SetNode(node);

        applied = app;
        return nodes;
    }
}

```

```

public class ConstantPropagation : IOptimization
{
    public List<Node> Optimize(List<Node> nodes, out bool
applied)
    {
        var app = false;
        for (int i = 0; i < nodes.Count; i++)
        {

```

```
        if (nodes[i] is Assign node && node.Operation
== OpCode.Copy && node.Right is IntConst)
        {
            for (int j = i + 1; j < nodes.Count; j++)
            {
                if (nodes[j] is Assign nextNode)
                {
                    //Если мы встретили объявление эт
ого же элемента
                    if (node.Result.Equals(nextNode.R
esult))
                        break;
                    //Проверка использования Result в
левом операнде другого узла
                    if (node.Result.Equals(nextNode.L
eft))
                    {
                        nextNode.Left = node.Right;
                        nodes[j] = nextNode;
                        app = true;
                    }
                    //Проверка использования Result в
правом операнде другого узла
                    if (node.Result.Equals(nextNode.R
ight))
                    {
                        nextNode.Right = node.Right;
                        nodes[j] = nextNode;
```

```

        app = true;
    }
}
if (nodes[j] is Print printNode)
{
    //Если Result равна тому, что нах
одится в Print и левый операнд node пустой
    if (node.Left is null && node.Res
ult.Equals(printNode.Data))
    {
        printNode.Data = node.Right;
        app = true;
    }
}
}
}
}
applied = app;
return nodes;
}
}

```

Тесты

Свёртка констант:

```
a = b
```

```
c = 20 * 3    -----> c = 60
```

```
d = 10 + 1      -----> d = 11
e = 100 / 50    -----> e = 2
a = 30 - 20     -----> a = 10
k = c + a
```

Протяжка констант:

```
a = 10
c = b - a      -----> c = b - 10
d = c + 1
e = d * a      -----> e = d * 10
a = 30 - 20
k = c + a      -----> k = c + a
```

Пример работы.

Свёртка констант:

```
public void Test1()
{
    var taCodeConstantFolding = new TACode();
    var assgn1 = new Assign()
    {
        Left = null,
        Operation = OpCode.Copy,
        Right = new Var(),
        Result = new Var()
```

```
};  
var assgn2 = new Assign()  
{  
    Left = new IntConst(20),  
    Operation = OpCode.Mul,  
    Right = new IntConst(3),  
    Result = new Var()  
};  
var assgn3 = new Assign()  
{  
    Left = new IntConst(10),  
    Operation = OpCode.Plus,  
    Right = new IntConst(1),  
    Result = new Var()  
};  
var assgn4 = new Assign()  
{  
    Left = new IntConst(100),  
    Operation = OpCode.Div,  
    Right = new IntConst(50),  
    Result = new Var()  
};  
var assgn5 = new Assign()  
{  
    Left = new IntConst(30),  
    Operation = OpCode.Minus,  
    Right = new IntConst(20),  
    Result = assgn1.Result
```

```
};  
var assgn6 = new Assign()  
{  
    Left = assgn2.Result,  
    Operation = OpCode.Plus,  
    Right = assgn5.Result,  
    Result = new Var()  
};
```

```
taCodeConstantFolding.AddNode(assgn1);  
taCodeConstantFolding.AddNode(assgn2);  
taCodeConstantFolding.AddNode(assgn3);  
taCodeConstantFolding.AddNode(assgn4);  
taCodeConstantFolding.AddNode(assgn5);  
taCodeConstantFolding.AddNode(assgn6);
```

```
var optConstFold = new ConstantFolding();  
optConstFold.Optimize(taCodeConstantFolding.CodeList.ToList(), out var aplConstFold);
```

```
Assert.AreEqual(assgn2.Right, 60);  
Assert.AreEqual(assgn3.Right, 11);  
Assert.AreEqual(assgn4.Right, 2);  
Assert.AreEqual(assgn5.Right, 10);  
Assert.True(true);
```

```
}
```

Протяжка констант:

```
public void Test1()
{
    var taCodeConstProp = new TACode();
    var assign1 = new Assign()
    {
        Left = null,
        Operation = OpCode.Copy,
        Right = new IntConst(10),
        Result = new Var()
    };
    var assign2 = new Assign()
    {
        Left = new Var(),
        Operation = OpCode.Minus,
        Right = assign1.Result,
        Result = new Var()
    };
    var assign3 = new Assign()
    {
        Left = assign2.Result,
        Operation = OpCode.Plus,
        Right = new IntConst(1),
        Result = new Var()
    };
    var assign4 = new Assign()
```



```

{
    Left = assign3.Result,
    Operation = OpCode.Mul,
    Right = assign1.Result,
    Result = new Var()
};
var assign5 = new Assign()
{
    Left = new IntConst(30),
    Operation = OpCode.Minus,
    Right = new IntConst(20),
    Result = assign1.Result
};
var assign6 = new Assign()
{
    Left = assign2.Result,
    Operation = OpCode.Plus,
    Right = assign5.Result,
    Result = new Var()
};

taCodeConstProp.AddNode(assign1);
taCodeConstProp.AddNode(assign2);
taCodeConstProp.AddNode(assign3);
taCodeConstProp.AddNode(assign4);
taCodeConstProp.AddNode(assign5);
taCodeConstProp.AddNode(assign6);

```

```
var optConstProp = new CopyPropagation();
optConstProp.Optimize(taCodeConstProp.CodeList.ToList(), out var applCopProp);

Assert.AreEqual(assign2.Right, assign1.Result);
Assert.AreEqual(assign4.Right, assign1.Result);
Assert.AreNotSame(assign6.Right, assign1.Result);
Assert.True(true);
}
```

Название задачи

Применение алгебраических и логических тождеств.

Постановка задачи

Реализовать оптимизацию применение алгебраических и логических тождеств для базового блока.

Зависимости задач в графе задач

Зависит от:

- Трехадресный код

Теоретическая часть задачи

В коде программы могут быть применены следующие тождества для оптимизации:

$x + 0 = x$	$x * 1 = x$
$0 + x = x$	$1 * x = x$
$x - 0 = x$	$x * 0 = 0$
$x - x = 0$	$x / 1 = x$
	$x / x = 1$

Практическая часть задачи (реализация)

Часть кода для оптимизации операции сложения. Полный файл по

ССЫЛКЕ.

```
public List<Node> Optimize(List<Node> nodes, out bool applied
)
{
    var app = false;
    var enumerable = nodes
        .OfType<Assign>()
        .Where(assn => assn.Operation != OpCode.Copy && assn.
Left != null);
    foreach (var node in enumerable)
    {
        switch (node.Operation)
        {
            case OpCode.Plus:
                if (node.Left.Equals(Zero))
                    app = SetLeft(node);
                else if (node.Right.Equals(Zero))
                    app = SetRight(node);
                break;
            ...
        }
    }
    ...
}
```

Тесты

(в трехадресном коде)

```
l0: a = b + 0 => l0: a = b
```

```
l1: b = 1 * a => l1: b = a
```

Пример работы.

(в трехадресном коде)

```
l0: a = b + 0
```

```
l1: a = a * 1
```

```
l2: a = a - 0
```

```
l3: a = a + b
```

```
l4: a = a * 1
```

=>

```
l0: a = b
```

```
l1: a = a
```

```
l2: a = a
```

```
l3: a = a + b
```

```
l4: a = a
```

Название задачи

Протяжка копий

Постановка задачи

Создать класс, реализующий алгоритм протяжки копий.

Зависимости задач в графе задач

Зависит от:

- Трехадресный код

От задачи зависит:

- Общий алгоритм при наличии вектора оптимизаций (O1, O2, ...)

Теоретическая часть задачи

Протяжка копий – еще одна известная проблема глобального анализа потока данных. Цель протяжки копий – заменить переменные их значениями.

Следующий фрагмент кода

```
y = x;  
z = 3 + y
```

после выполнения протяжки копий будет выглядеть вот так:

```
z = 3 + x
```

Практическая часть задачи (реализация)

Был реализован класс `CopyPropagation`.

```
public class CopyPropagation : IOptimization
{
    public List<Node> Optimize(List<Node> nodes, out bool
applied)
    {
        var app = false;
        for (int i = 0; i < nodes.Count; i++)
        {
            if (nodes[i] is Assign node && node.Operation
== OpCode.Copy && !(node.Right is IntConst))
            {
                for (int j = i + 1; j < nodes.Count; j++)
                {
                    if (nodes[j] is Assign nextNode)
                    {
                        //Если мы встретили объявление эт
ого же элемента
                        if (node.Result.Equals(nextNode.R
esult))
```

```

        break;

        //Проверка использования Result в
        левом операнде другого узла
        if (node.Result.Equals(nextNode.L
eft))
        {
            nextNode.Left = node.Right;
            nodes[j] = nextNode;
            app = true;
        }

        //Проверка использования Result в
        правом операнде другого узла
        if (node.Result.Equals(nextNode.R
ight))
        {
            nextNode.Right = node.Right;
            nodes[j] = nextNode;
            app = true;
        }
    }
}

applied = app;
return nodes;
}
}

```


Тесты

Протяжка копий:

```
a = b
c = b - a      -----> c = b - b
d = c + 1
e = d * a      -----> e = d * b
a = 30 - 20
k = c + a      -----> k = c + a
```

Пример работы.

```
public void Test1()
{
    var taCodeCopyProp = new TACode();
    var assgn1 = new Assign()
    {
        Left = null,
        Operation = OpCode.Copy,
        Right = new Var(),
        Result = new Var()
    };
    var assgn2 = new Assign()
    {
        Left = assgn1.Right,
        Operation = OpCode.Minus,
        Right = assgn1.Result,
```

```
        Result = new Var()
    };
    var assign3 = new Assign()
    {
        Left = assign2.Result,
        Operation = OpCode.Plus,
        Right = new IntConst(1),
        Result = new Var()
    };
    var assign4 = new Assign()
    {
        Left = assign3.Result,
        Operation = OpCode.Mul,
        Right = assign1.Result,
        Result = new Var()
    };
    var assign5 = new Assign()
    {
        Left = new IntConst(30),
        Operation = OpCode.Minus,
        Right = new IntConst(20),
        Result = assign1.Result
    };
    var assign6 = new Assign()
    {
        Left = assign2.Result,
        Operation = OpCode.Plus,
        Right = assign5.Result,
```

```
        Result = new Var()  
    };  
  
    taCodeCopyProp.AddNode(assign1);  
    taCodeCopyProp.AddNode(assign2);  
    taCodeCopyProp.AddNode(assign3);  
    taCodeCopyProp.AddNode(assign4);  
    taCodeCopyProp.AddNode(assign5);  
    taCodeCopyProp.AddNode(assign6);  
  
    var optCopyProp = new CopyPropagation();  
    optCopyProp.Optimize(taCodeCopyProp.CodeList.ToList(), out var applCopProp);  
  
    Assert.AreEqual(assign2.Right, assign1.Right);  
    Assert.AreEqual(assign4.Right, assign1.Right);  
    Assert.AreNotSame(assign6.Right, assign1.Right);  
    Assert.True(true);  
}
```

Название задачи

Общий алгоритм при наличии вектора оптимизаций (O1, O2, ...)

Постановка задачи

Необходимо реализовать алгоритм, который применял бы все возможные оптимизации для базового блока.

Зависимости задач в графе задач

Зависит от:

- Все реализованные оптимизации для базового блока

Теоретическая часть задачи

Требуется реализовать алгоритм, который применил бы к базовому блоку все имеющиеся оптимизации до тех пор, пока это возможно делать.

Практическая часть задачи (реализация)

Был реализован класс `AllOptimizations` и функция `ApplyAllOptimizations`, применяющая все оптимизации к трехадресному коду.

```
public class AllOptimizations
{
    private List<IOptimization> BasicBlockOptimizationLis
```

```

t()
{
    List<IOptimization> optimizations = new List<IOptimization>();

    optimizations.Add(new CopyPropagation());
    optimizations.Add(new ConstantFolding());
    optimizations.Add(new ConstantPropagation());
    optimizations.Add(new DeclarationOptimization());
    optimizations.Add(new AlgebraicOptimization());
    optimizations.Add(new SubexpressionOptimization());

    return optimizations;
}

private List<IOptimization> o2OptimizationList()
{
    return new List<IOptimization>();
}

public TACode ApplyAllOptimizations(TACode code)
{
    List<IOptimization> o1Optimizations = BasicBlockOptimizationList();

    var canApplyAny = true;

```

```
while (canApplyAny)
{
    canApplyAny = false;
    var blocks = code.CreateBasicBlockList().ToList();

    var codeList = new List<Node>();

    foreach (var b in blocks)
    {
        var block = b.CodeList.ToList();
        for (int i = 0; i < o10optimizations.Count; i++)
        {
            block = o10optimizations[i].Optimize(block, out var applied);
            canApplyAny = canApplyAny || applied;
        }
        codeList.AddRange(block);
    }

    code = new TACode();
    code.CodeList = codeList;

    foreach (var line in code.CodeList)
        code.LabeledCode[line.Label] = line;
}
```

```
        return code;
    }
}
```

Тесты

```
a = b
c = b - a    -----> c = 0
n = 20
c = 20 * 3   -----> c = 60
d = 10 + n   -----> d = 30
```

Пример работы.

```
public void Test1()
{
    var taCodeAllOptimizations = new TACode();
    var assign1 = new Assign()
    {
        Left = null,
        Operation = OpCode.Copy,
        Right = new Var(),
        Result = new Var()
    };
    var assign2 = new Assign()
    {
        Left = assign1.Right,
```

```
        Operation = OpCode.Minus,
        Right = assgn1.Result,
        Result = new Var()
    };
    var assgn3 = new Assign()
    {
        Left = null,
        Operation = OpCode.Copy,
        Right = new IntConst(20),
        Result = new Var()
    };
    var assgn4 = new Assign()
    {
        Left = new IntConst(20),
        Operation = OpCode.Mul,
        Right = new IntConst(3),
        Result = new Var()
    };
    var assgn5 = new Assign()
    {
        Left = new IntConst(10),
        Operation = OpCode.Plus,
        Right = assgn3.Result,
        Result = new Var()
    };
    taCodeAllOptimizations.AddNode(assgn1);
    taCodeAllOptimizations.AddNode(assgn2);
    taCodeAllOptimizations.AddNode(assgn3);
```



```
taCodeAllOptimizations.AddNode(assign4);
```

```
taCodeAllOptimizations.AddNode(assign5);
```

```
var allOptimizations = new AllOptimizations();
```

```
allOptimizations.ApplyAllOptimizations(taCodeAllOptimizations);
```

```
Assert.AreEqual(assign2.Right, 0);
```

```
Assert.AreEqual(assign4.Right, 60);
```

```
Assert.AreEqual(assign5.Right, 30);
```

```
Assert.True(true);
```

```
}
```

Название задачи

Представление данных для анализа потока данных (DFA).

Постановка задачи

Предоставить инфраструктуру для написания алгоритмов анализа графа потока управления.

Зависимости задач в графе задач

- Все оптимизации на уровне CFG

Теоретическая часть задачи

Под анализом потоков данных понимают совокупность задач, нацеленных на выяснение некоторых глобальных свойств программы, то есть извлечение информации о поведении тех или иных конструкций в некотором контексте. Такая постановка задачи возможна по той причине, что язык программирования и вычислительная среда определяют некоторую общую “безопасную”, семантику конструкций, которая годится “на все случаи жизни”. Учет же контекстных условий позволяет делать более конкретные, частные заключения о поведении той или иной конструкции; при этом такие заключения, вообще говоря, перестают быть верными в другом контексте.

Например, общая семантика присваивания заключается в вычислении выражения, стоящего в правой части, и присваивании полученного

значения в переменную, стоящую в левой части. Однако в случае, когда выражение в правой части не имеет побочных эффектов, а переменная в левой части более нигде не используется, данный оператор становится эквивалентен пустому. Понятно, что на смысл каждой конструкции может оказывать влияние любая конструкция, из которой в этом графе достижима данная.

Отсюда следует, что для правильного учета контекста необходимо принять во внимание влияние всех путей до данной вершины: сначала определив влияние каждого пути, а затем выделив общую часть. Задача осложняется тем, что при наличии циклов множество всех путей в графе управления становится бесконечным.

Итерационный алгоритм

Логически процесс решения задачи анализа потоков данных состоит из двух стадий, выполняемых одновременно. Локальная стадия заключается в учете влияния отдельного оператора (группы операторов в узле графа управления) в предположении, что уже имеется решение задачи анализа потоков данных перед этим оператором. На глобальной стадии происходит решение задачи анализа для каждого пути, ведущего в данную вершину, и затем выделение общей части всех таких решений.

Основной проблемой подхода является проблема остановки алгоритма. Действительно, в какой момент процесс уточнения разметки должен прекратиться? Очевидно, в тот момент, когда получено решение задачи анализа потоков данных. Однако поскольку решение задачи неизвестно, то и воспользоваться этим наблюдением напрямую

оказывается невозможным. Поэтому для определения завершаемости алгоритма используется другой принцип -- принцип достижения неподвижной точки.

Частично-упорядоченное множество X будем называть множеством конечной высоты N тогда и только тогда, когда длины всех строго возрастающих последовательностей элементов X ограничены N . Это означает, что для произвольной возрастающей последовательности начиная с некоторого места все элементы становятся одинаковыми. Рассмотрим теперь функцию перехода F , удовлетворяющую соотношению $F(\mu) \geq \mu$ для произвольной разметки μ . Понятно, что при таком условии при итерировании F начиная с некоторого места будет достигнута ее неподвижная точка. Множество X и функция перехода F подбираются таким образом, чтобы эта неподвижная точка являлась решением задачи анализа потоков данных.

Практическая часть задачи (реализация)

- Был реализован интерфейс `IAlgorithm`, предками которого обязаны быть все алгоритмы на графе потока управления. Интерфейс содержит единственный метод `Analyze`, который возвращает словарь с парами вида <узел графа потока управления, кортеж `IN` и `OUT` массивов>, полученных в ходе анализа. На вход методу подается тройка <граф потока управления, набор операций сбора алгоритма, передаточная функция>.
- Был реализован интерфейс `ILatticeOperations`, который представляет собой набор операций сбора, определяющих

поведение алгоритма. Интерфейс включает:

- нижнюю и верхнюю границы,
- бинарный оператор, устанавливающий порядок между двумя элементами **IN / OUT** ,
- оператор сбора, который пользователь должен определить сам исходя из потребностей алгоритма.

Название задачи

Интерфейс передаточной функции

Постановка задачи

Определить интерфейс передаточной функции.

Зависимости задач в графе задач

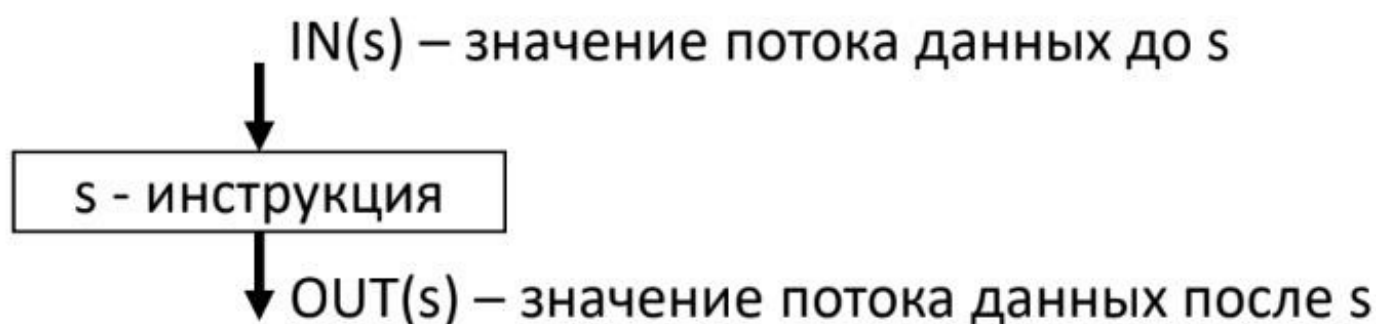
Зависит от:

- Структуры базовых блоков

От задачи зависит:

- Итерационный алгоритм
- DFA-анализ

Теоретическая часть задачи



Передаточная функция:

$OUT[S] = f_{\{S\}}(IN[S])$, где $f_{\{S\}}$ задает семантику инструкции

В задаче обратного потока данных:

$$IN[S] = f_{\{S\}}(OUT[S])$$

Очевидно, что

$$IN[S_{i+1}] = OUT[S_i]$$

Передаточная функция ББЛ - композиция передаточных функций инструкций

$$f_{\{B\}} = f_{\{S1\}} \circ f_{\{S2\}} \circ \dots \circ f_{\{SN\}}$$

Полурешётка — полугруппа, бинарная операция в которой коммутативна и идемпотентна.

С точки зрения теоретико-множественного подхода, полурешётка определяется как частично упорядоченное множество, для каждой пары элементов которого определена точная верхняя грань (верхняя полурешётка) или точная нижняя грань (нижняя полурешётка).

Множество, являющееся одновременно верхней и нижней полурешёткой является решёткой.

Практическая часть задачи (реализация)

```
public interface ITransferFunction<T>
{
    T Transfer(BasicBlock basicBlock, T input, ILatticeOperations<T> ops);
}
```

где basicBlock - базовый блок, input - множество IN, ops - набор

операций над полурешеткой.

Операции над полурешеткой реализуют интерфейс:

```
public interface ILatticeOperations<T>
{
    T Operator(T a, T b);

    bool? Compare(T a, T b);

    T Lower { get; }

    T Upper { get; }
}
```

Тесты

Не подразумеваются.

Пример работы.

Не подразумевается.

Название задачи

Передающая функция для достигающих определений, множества gen и kill.

Постановка задачи

Реализовать передающую функцию для достигающих определений, вычисление множеств gen и kill.

Зависимости задач в графе задач

Зависит от:

- Структура базовых блоков
- Интерфейс передающей функции

Теоретическая часть задачи

Определение: Будем говорить, что определение d достигает точки r , если существует путь от точки, непосредственно следующей за d , к точке r , такой, что d не уничтожается вдоль этого пути.

Привести пример

Анализ должен быть консервативным: если не знаем, есть ли другое присваивание на пути, то считаем, что существует.

Достигающие определения используются при:

1. Является ли x константой в точке r ? (если r достигает одно определение x , и это – определение константы)

2. Является ли x в точке p неинициализированной? (если p не достигает ни одно определение x)

genB – множество определений, генерируемых базовым блоком B.

killB – множество остальных определений переменных, определяемых в определениях genB, в других ББл.

Практическая часть задачи (реализация)

Ниже представлена реализация передаточной функции для достигающих определений и метод вычисления множеств gen и kill.

```
public class TransferFunction : ITransferFunction<HashSet<Guid>>
{
    private TACode taCode;

    public TransferFunction(TACode ta) => taCode = ta;

    public HashSet<Guid> Transfer(BasicBlock basicBlock,
        HashSet<Guid> input, ILatticeOperations<HashSet<Guid>> ops)
    {
        var (gen, kill) = GetGenAndKill(basicBlock, ops);
        var inset = new HashSet<Guid>(input);
        return new HashSet<Guid>(inset.Except(kill).Union
            (gen));
    }
}
```

```

    public (HashSet<Guid>, HashSet<Guid>) GetGenAndKill (
BasicBlock basicBlock, ILatticeOperations<HashSet<Guid>> ops)
    {
        var gen = new HashSet<Guid>(basicBlock.CodeList.W
here(x => x is Assign).Select(x => x.Label));
        var vars = basicBlock.CodeList
            .Where(x => x is Assign)
            .Select(x => ((x as Assign).Result as Var).Id
)
            .ToList();
        var ad = taCode.CodeList
            .Where(x => !gen.Contains(x.Label) && x is As
sign)
            .Cast<Assign>()
            .Where(x => vars.Contains((x.Result as Var).I
d))
            .Select(x => x.Label);
        var kill = new HashSet<Guid>(ad);
        return (gen, kill);
    }
}

```

Тесты

```

l1: a = 3 - 5
l2: b = 10 + 2
l3: c = -1
l4: if 1 goto l3

```

```
l5: d = c + 1999
l6: if 2 goto l2
l7: e = 7 * 4
l8: f = 100 / 25
```

```
gen(B1) = { l1 }
```

```
kill(B1) = { }
```

```
gen(B2) = { l2 }
```

```
kill(B2) = { }
```

```
gen(B3) = { l3 }
```

```
kill(B3) = { }
```

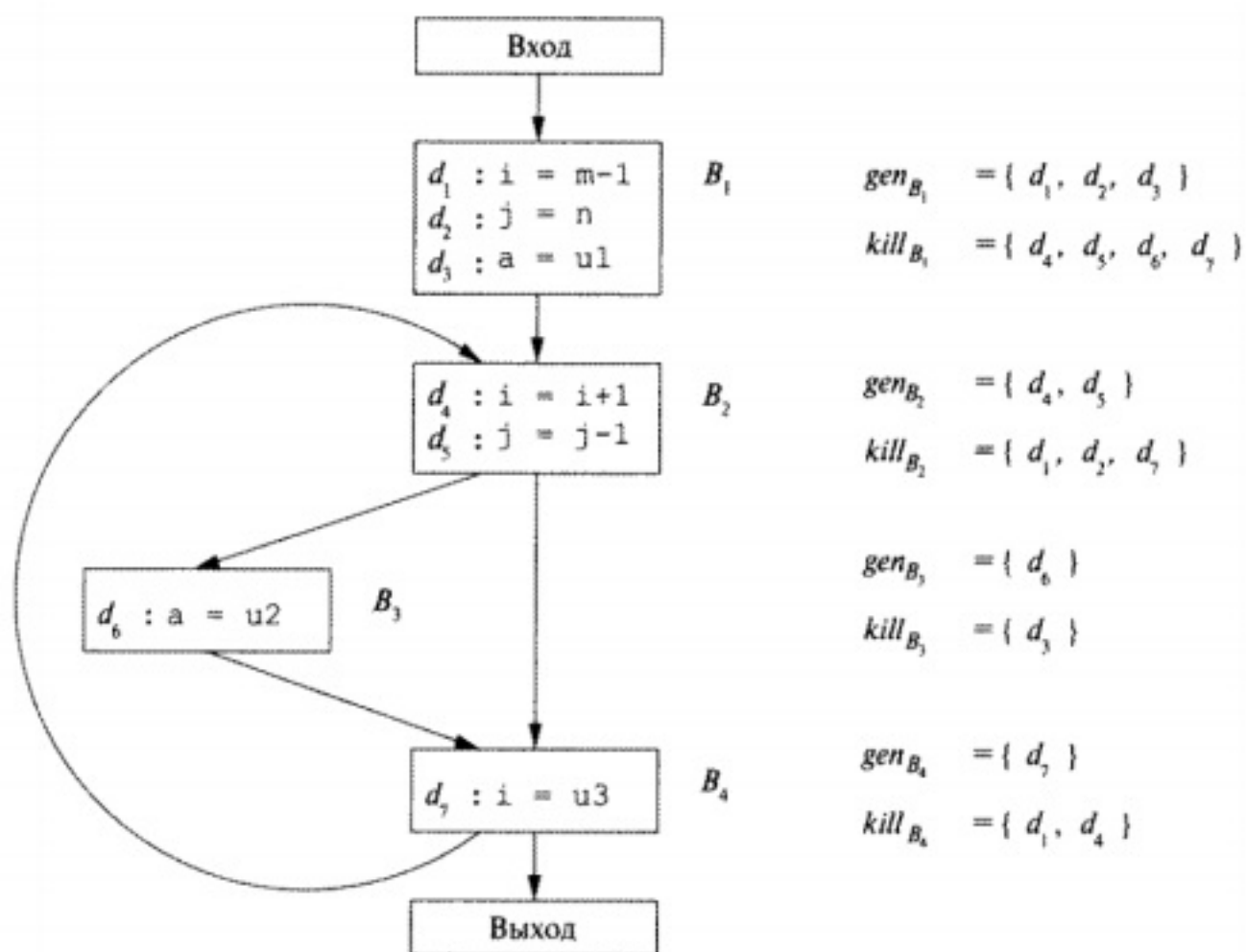
```
gen(B4) = { l5 }
```

```
kill(B4) = { }
```

```
gen(B5) = { l7, l8 }
```

```
kill(B4) = { }
```

Пример работы.



Название задачи

Настроить CI, авторевью кода и тесты

Постановка задачи

С помощью облачных сервисов:

- настроить прекоммит хуки на проверку оформления кода по заданным параметрам
- настроить прекоммит хуки на сборку проекта с указанной конфигурацией
- выбрать библиотеку для тестов

Зависимости задач в графе задач

Отсутствуют

Теоретическая часть задачи

Прекоммит хуки не позволяют принимать, например, пулл реквест, пока не выполнены некоторые условия.

Для удаленной сборки проекта был выбран сервис Travis-CI, так как в нем есть поддержка C#. Для анализа code climate был выбран сервис Codacy.

Практическая часть задачи (реализация)

Привер конфигурации для сборки проекта

```
language: csharp
```

```
solution: Compiler.Parser.sln
```

```
mono:
```

```
- latest
```

```
script:
```

```
- xbuild /p:Configuration=WithoutIDE Compiler.Parser.sln
```

Пример работы.

- Окно мониторинга сборки проекта

The screenshot displays the Travis CI web interface. At the top, the Travis CI logo and navigation links (About Us, Blog, Status, Help) are visible on the left, and the user profile 'Alexander Svetly' is on the right. The main content area shows the repository 'Lucky112 / mmcs-optimizing-compiler-spring-2018' with a 'build passing' status. Below this, the 'Current' build is detailed, showing the commit 'f053daa' and the branch 'doc'. The build log is visible, showing the installation of Mono and the execution of the build script. The log includes the following text:

```
1
2 C# support for Travis-CI is community maintained.
3 Please open any issues at https://github.com/travis-ci/travis-ci/issues/new and cc @joshua-anderson @akoeplinger @nterry
4 Installing Mono
5 Executing: /tmp/tmp.09KQrVmzVb/gpg.1.sh --keyserver
6 keyserver.ubuntu.com
7 --recv-keys
8 3FA7E0328081BFF6A14DA29AA6A19B38D3D831EF
9 gpg: requesting key D3D831EF from hkp server keyserver.ubuntu.com
10 gpg: key D3D831EF: public key "Xamarin Public Jenkins (auto-signing) <releng@xamarin.com>" imported
11 gpg: Total number processed: 1
12 gpg:      imported: 1 (RSA: 1)
13 W: http://ppa.launchpad.net/couchdb/stable/ubuntu/dists/trusty/Release.gpg: Signature by key 158668AFD9BCC4F3C1E0DFC7D69548E1C17E/ 7,895
14 uses weak digest algorithm (SHA1)
15 Extracting templates from packages: 100%
```

Название задачи

Реализация передаточной функции композицией

Постановка задачи

$f_s(X)$ - для кода

$f_b(X)$ - для блока

X - множество из DFA

$OUT[B] = f_b(IN[B])$

Нужно уметь находить суперпозицию $f_b = f_{s1} . f_{s2} . \dots . f_{sn}$

Зависимости задач в графе задач

Зависит от:

- Базовый блок

Теоретическая часть задачи

Отсутствует

Практическая часть задачи (реализация)

Набор методов класса `PrettyPrintVisitor`, реализующих `Ivisitor`:

```
public static class TransferFunctionUtils
{
    public static ITransferFunction<T> Compose<T>(this ITrans
ferFunction<T> f1, ITransferFunction<T> f2) {}
```



```
private class TransferFuctionComposition<T> : ITransferFu
nction<T>
{
    private readonly ITransferFunction<T> f1, f2;
    public TransferFuctionComposition(ITransferFunction<T
> f1, ITransferFunction<T> f2) {}
    public T Transfer(BasicBlock basicBlock, T input, ILa
tticeOperations<T> ops) {}
}
}
```

Пример работы.

Отсутствует

Название задачи

Итерационный алгоритм для достигающих определений.

Постановка задачи

Реализовать итерационный алгоритм для достигающих определений.

Зависимости задач в графе задач

Зависит от:

- Интерфейс передаточной функции
- Передаточная функция и генерация множеств `gen` и `kill`
- Структура базовых блоков

От задачи зависит:

- Тестирование итерационного алгоритма

Теоретическая часть задачи

Определение: Будем говорить, что определение d достигает точки p , если существует путь от точки, непосредственно следующей за d , к точке p , такой, что d не уничтожается вдоль этого пути.

Анализ должен быть консервативным: если не знаем, есть ли другое присваивание на пути, то считаем, что существует.

Достигающие определения используются при:

1. Является ли x константой в точке p ? (если p достигает одно

определение x , и это – определение константы)

2. Является ли x в точке p неинициализированной? (если p не достигает ни одно определение x)

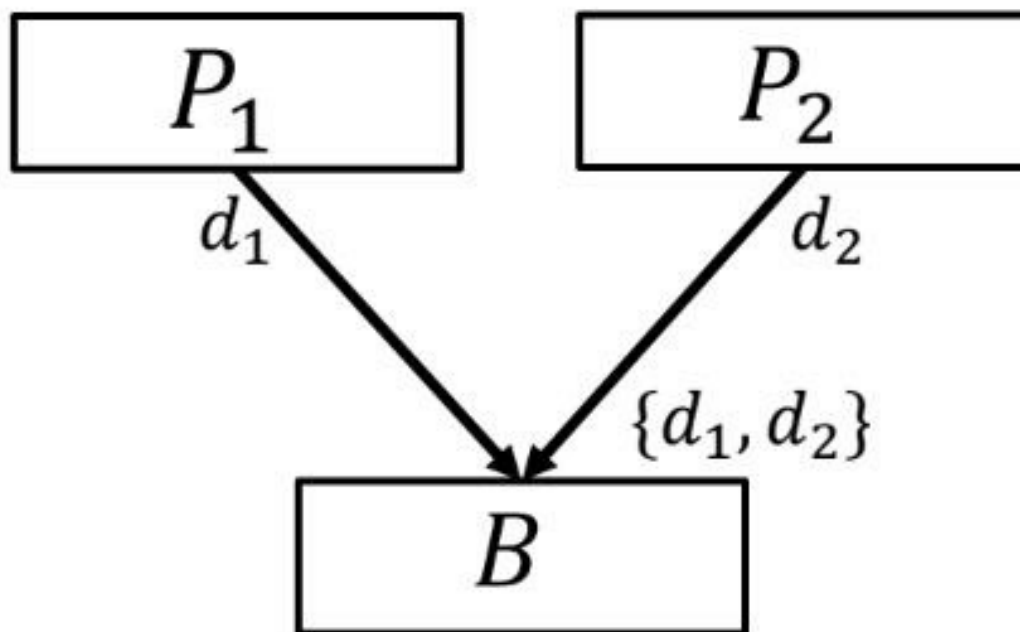
Передаточная функция в общем случае для достигающих определений:

$f_B(X) = gen_B \cup (X - kill_B)$, где

$kill_B = kill_1 \cup kill_2 \cup \dots \cup kill_n$

$gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \dots$
 $\cup (gen_1 - kill_2 - \dots - kill_n)$

Оператор сбора для достигающих определений:



$$IN[B] = \bigcup_{P-\text{предшественник } B} OUT[P]$$

Уравнения для достигающих определений:

$$OUT[Вход] = \emptyset$$

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$

$$IN[B] = \bigcup_{P-\text{предш } B} OUT[P]$$

Итеративный алгоритм:

Вход: граф потока управления, в котором для каждого ББл вычислены gen_B и $kill_B$

Выход: Множества достигающих определений на входе $IN[B]$ и на выходе $OUT[B]$ для каждого ББл B

$OUT[Вход] = \emptyset;$

for (каждый базовый блок B , отличный от входного) $OUT[B] = \emptyset;$

while (внесены изменения в OUT)

for (каждый базовый блок B , отличный от входного) {

$IN[B] = \bigcup_{P-\text{предшественник } B} OUT[P];$

$OUT[B] = gen_B \cup (IN[B] - kill_B);$

 }

Сходимость алгоритма: на каждом шаге $IN[B]$ и $OUT[B]$ не уменьшаются для всех B и ограничены сверху, поэтому алгоритм

СХОДИТСЯ.

Практическая часть задачи (реализация)

Алгоритм реализован в соответствии со схемой, приведенной выше.

```
public class IterativeAlgorithm : IAlgorithm<HashSet<Guid>>
{
    public InOutData<HashSet<Guid>> Analyze(ControlFlowGraph
graph, ILatticeOperations<HashSet<Guid>> ops, ITransferFunction<HashSet<Guid>> f)
    {
        var data = new InOutData<HashSet<Guid>>
        {
            [graph.CFGNodes.ElementAt(0)] = (ops.Lower, ops.L
ower)
        };
        foreach (var node in graph.CFGNodes)
            data[node] = (ops.Lower, ops.Lower);
        var outChanged = true;
        while (outChanged)
        {
            outChanged = false;
            foreach (var block in graph.CFGNodes)
            {
                var inset = block.Parents.Aggregate(ops.Lower
, (x, y)
                => ops.Operator(x, data[y].Item2));
```

```

        var outset = f.Transfer(block, inset, ops);
        if (outset.Except(data[block].Item2).Any())
        {
            outChanged = true;
            data[block] = (inset, outset);
        }
    }
}
return data;
}
}

```

Тесты

```

l1: a = 3 - 5
l2: b = 10 + 2
l3: c = -1
l_: if 1 goto l3
ass4 = l5: d = c + 1999
l_: if 2 goto l2
l7: e = 7 * 4
l8: f = 100 / 25

```

```
IN[0] = { }
```

```
OUT[0] = { l1 }
```

```
IN[1] = { l1, l2, l3, l5 }
```

OUT[1] = { l1, l2, l3, l4 }

IN[2] = { l1, l2, l3, l5 },

OUT[2] = { l1, l2, l4, l5 }

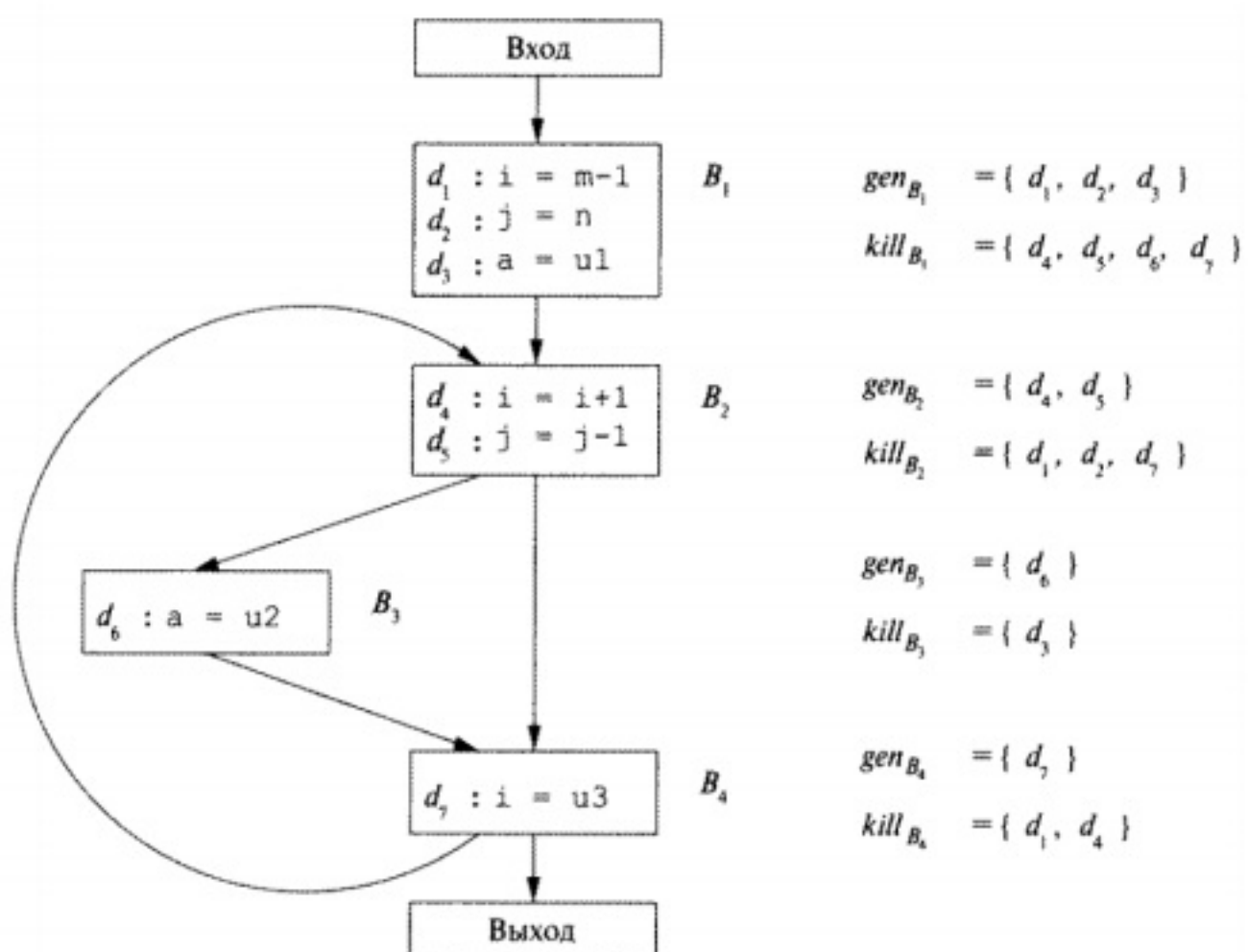
IN[3] = { l1, l2, l3, l5 }

OUT[3] = { l1, l2, l3, l5 }

IN[4] = { l1, l2, l3, l5 }

OUT[4] = { l1, l2, l3, l5, l7, l8 }

Пример работы.



БЛОК B	$\text{OUT}[B]^0$	$\text{IN}[B]^1$	$\text{OUT}[B]^1$	$\text{IN}[B]^2$	$\text{OUT}[B]^2$
B_1	0000000	0000000	1110000	0000000	1110000
B_2	0000000	1110000	0011100	1110111	0011110
B_3	0000000	0011100	0001110	0011110	0001110
B_4	0000000	0011110	0010111	0011110	0010111
ВЫХОД	0000000	0010111	0010111	0010111	0010111

Название задачи

Реализация общего итерационного алгоритма

Зависит от:

CFG

Постановка задачи

Необходимо создать класс для реализации общего итерационного алгоритма

Описание

Для всех узлов, за исключением входного, определяем множества OUT , как множество всех узлов, и множество IN , которые являются пустым для всех узлов.

Осуществляем итеративный проход по всем узлам, в цикле множество IN для узла определяется как пересечение множеств OUT для всех предшественников в узла, а множество OUT для узла определяется как IN , объединенное с текущим узлом.

Предыдущий цикл выполняется до тех пор, пока множество IN и OUT изменяются, множества OUT для узлов являются результатом.

Реализация

```
public IterativeAlgorithmAV(ControlFlowGraph CFG)
{
    this.CFG = CFG;
    StartSettings();
    Algorithm();
}
```

```
/// <summary>
/// Класс для итеративного алгоритма
/// активных переменных
/// </summary>
/// <param name="CFG"></param>
public IterativeAlgorithmAV(List<Node> listNode)
{
    var TA = new TACode();
    TA.CodeList = listNode;
    this.CFG = new ControlFlowGraph(TA);
    StartSettings();
    Algorithm();
}
```

```
/// <summary>
/// Стартовые настройки алгоритма
/// </summary>
private void StartSettings()
{
```

```
IN = new Dictionary<Guid, ActiveVar>();
OUT = new Dictionary<Guid, ActiveVar>();
DefSet = new Dictionary<Guid, DSet>();
UseSet = new Dictionary<Guid, USet>();
```

```
foreach (var B in CFG.CFGNodes)
{
    IN.Add(B.BlockId, new ActiveVar());
    OUT.Add(B.BlockId, new ActiveVar());

    var duSets = new DUSets(B);

    DefSet.Add(B.BlockId, duSets.DSet);
    UseSet.Add(B.BlockId, duSets.USet);
}
}
```

```
/// <summary>
```

```
/// Сравнение двух словарей
```

```
/// </summary>
```

```
/// <param name="obj1"></param>
```

```
/// <param name="obj2"></param>
```

```
/// <returns></returns>
```

```
public bool EqualIN(Dictionary<Guid, ActiveVar> obj1,
Dictionary<Guid, ActiveVar> obj2)
{
    var IsEqual = true;
```

```

        foreach (var v in obj1)
        {
            if (v.Value.Count != obj2[v.Key].Count)
                return false;

            for (var i = 0; i < v.Value.Count; i++)
                IsEqual &= v.Value.ElementAt(i) == obj2[v
.Key].ElementAt(i);
        }

        return IsEqual;
    }

    /// <summary>
    /// Копирование словаря
    /// </summary>
    /// <returns></returns>
    private Dictionary<Guid, ActiveVar> CopyIN()
    {
        var oldSetIN = new Dictionary<Guid, ActiveVar>();

        foreach (var elem in IN)
        {
            ActiveVar AV = new ActiveVar();

            foreach (var v in elem.Value)
                AV.Add(v);
        }
    }

```

```

        oldSetIN.Add(elem.Key, AV);
    }

    return oldSetIN;
}

/// <summary>
/// Базовый итеративный алгоритм
/// </summary>
private void Algorithm()
{
    Dictionary<Guid, ActiveVar> oldSetIN;

    do
    {
        oldSetIN = CopyIN();

        foreach (var B in CFG.CFGNodes)
        {
            var idB = B.BlockId;

            // Первое уравнение
            foreach (var child in B.Children)
            {
                var idCh = child.BlockId;
                OUT[idB].UnionWith(IN[idCh]);
            }
        }
    }
}

```

```
        var subUnion = new ActiveVar(OUT[idB]);
        subUnion.ExceptWith(DefSet[idB]);

        // Второе уравнение
        IN[idB].UnionWith(UseSet[idB]);
        IN[idB].UnionWith(subUnion);
    }
}
while (!EqualIN(oldSetIN, IN));
}
}
}
```

Название задачи

Поиск множеств **E_GEN** и **E_KILL** для базового блока

Постановка задачи

Реализовать алгоритм поиска множеств **E_GEN** и **E_KILL** по базовому блоку

Зависимости задач в графе задач

- Базовые блоки
- Трехадресный код
- Реализация итеративного алгоритма для достигающих выражений

Теоретическая часть задачи

Множество **E_GEN** – это такое множество, которое содержит все выражения сгенерированные базовым блоком **B**. Причем переменные из выражений не переопределяются до конца блока.

Множество **E_KILL** – это такое множество, которое содержит все выражения из программы уничтожаемые базовым блоком **B**

Блок генерирует выражение **x + y**, если он вычисляет **x + y** и потом не переопределяет **x** и **y**

Блок уничтожает выражение **x + y**, если он присваивает **x** или **y** и

потом не перевычисляет $x + y$

Практическая часть задачи (реализация)

- Был релизован класс `TransferFunction`, который в свою очередь унаследован от интерфейса `ITransferFunction`
- Был релизован метод `GetEGenEKill` для поиска множеств `e_gen` и `e_kill`
- Был реализован метод `Transfer`, который по переданному ему базовому блоку и входному множеству `IN` находит множества `e_gen`, `e_kill` и применяет передаточную функцию для вычисления множества `OUT`.

Алгоритм состоит из двух этапов. На первом этапе происходит поиск множества `e_gen` для базового блока

```
node_index = 0
foreach node in basicBlock.Nodes:
    if (node is expression as expr):
        redefinition = false
        for next_node in basicBlock.Skip(node_index+1):
            if (next_node is Assign as ass):
                if (ass.Left is Var as lv) && (lv.Id == a
ss.Result.Id):
                    redefinition = true
                if (ass.Right is Var as rv) && (rv.Id ==
ass.Result.Id):
                    redefinition = true
```



```

        if !redefinition:
            e_gen.Add(expr.Label)
    node_index++

```

На втором этапе происходит поиск множества `e_kill`.

- Сначала находим список всех выражений-присвоений базового блока(не обязательно только `x + y`).
- Далее, из всех выражений-присвоений(вида `x + y`) трёхадресного кода исключаем выражения-присвоения базового блока.
- Находим метки выражений-присвоений базового блока.
- Ищем множество `e_kill` для всех выражений-присвоений текущего базового блока

```

foreach node in basicBlock.Nodes:
    if node is Assign as ass:
        basicBlockAssignNodes.Add(ass)

exceptedAssignNodes = AllAssignNodes.Except(basicBlockAssignNodes)

marksBasicBlockAssignNodes = basicBlockAssignNodes.Select(
    (ass_node => ass_node.Result.Id)

foreach ean in exceptedAssignNodes:
    contains = false
    if (ean.Left is Var as lv) && marksBasicBlockAssignNodes.contains(lv.Id):

```

```
        contains = true
        if (!contains) && (ean.Right is Var as rv) && marksBasicBlockAssignNodes.contains(rv.Id):
            contains = true
        if contains:
            e_kill.Add(ean.Label)
```

Метод **Transfer**

```
def Transfer(basicBlock, IN):
    (e_gen, e_kill) = GetEGenEKill(basicBlock)
    outset = Union(IN \ e_kill, e_gen)
```

Название задачи

Итерационный алгоритм для достигающих выражений

Постановка задачи

Реализовать алгоритм поиска **IN/OUT** множеств для всех базовых блоков программы.

Зависимости задач в графе задач

- Базовые блоки
- Трехадресный код
- Граф потока управления
- Поиск множеств **E_GEN** и **E_KILL**
- Оптимизации доступных выражений

Теоретическая часть задачи

Выражение вида **$x + y$** доступно в точке **p** , если любой путь от входа к **p** вычисляет **$x + y$** и после последнего вычисления до достижения **p** нет присваиваний **x и y** .

Оператор сбора – операция над множествами. Для алгоритма достигающих выражений оператор сбора это пересечение.

U – универсальное множество.

Универсальное множество – множество, содержащее все выражения-присвоения вида **$x + y$** программы.

В отличие от алгоритма о достигающих определениях в алгоритме достигающих выражений осуществляется прямой проход(сверху вниз),

а не обратный.

Смысл алгоритма поиска достигающих выражений в том, чтобы найти достигающие выражения на входе(IN) и на выходе(OUT) базового блока для всех базовых блоков в программе

Практическая часть задачи (реализация)

- Был релизован класс `IterativeAlgorithm`, который в свою очередь унаследован от интерфейса `IAlgorithm`
- Был релизован метод `Analyze`, который по переданному ему графу потоку управления, оператора сбора и передаточной функции находит `IN/OUT` множества для всех базовых блоков
- Так же был реализован класс `Operations`, который унаследован от интерфейса `ILatticeOperations`. Класс `Operations` инкапсулирует нахождение верхней границы полурешётки(универсального множества выражений-присвоений трёхадресного кода) и метод `Operator`, который реализует оператор сбора.

Псевдокод алгоритма.

```
def Analyze(CFG, op, tf)
    IN[B0] = {}
    OUT[B0] = tf.Transfer(B0, IN[B0], op)

    foreach block in CFG.Blocks.Skip(1):
        IN[block] = {}
```

```
OUT[block] = U
```

```
changed = true
```

```
while changed:
```

```
    changed = false
```

```
    foreach block in CFG.Blocks.Skip(1):
```

```
        IN[block] = op(block.Parents)
```

```
        OUT[block] = tf.Transfer(block, IN[block], op  
)
```

```
        if OUT[block] \ OUT_prev[block] != {} :
```

```
            changed = true
```

```
            data[block] = (IN[block], OUT[block])
```

Название задачи

Тестирование итерационного алгоритма для достигающих выражений

Постановка задачи

Провести тестирование алгоритма достигающих выражений

Зависимости задач в графе задач

- Базовые блоки
- Трехадресный код
- Граф потока управления
- Поиск множеств **E_GEN** и **E_KILL**
- Итерационный алгоритм поиска достигающих выражений

Практическая часть задачи (реализация)

- Были произведены автоматические тесты для алгоритма поиска достигающих выражений

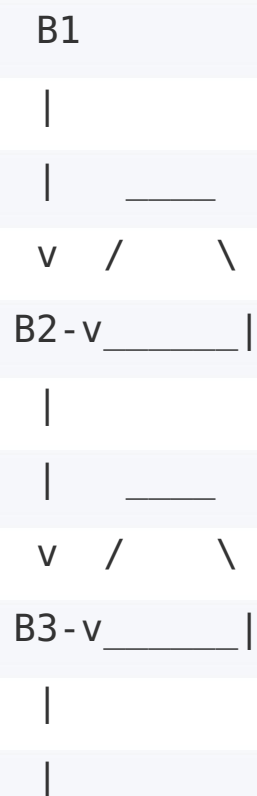
Тесты

Трёхадресный код теста

0)	$t_0 = 3$		
1)	$t_1 = 5 + t_0$		B1
2)	$t_2 = t_1 + t_0$		
3)	$t_3 = 4 + t_2$		

4)	t1 = 10	B2
5)	if (1) goto 3)	
6)	t4 = t1 + 5	
7)	t5 = t3 + t0	
8)	t0 = 100	B3
9)	if (2) goto 6)	
10)	t6 = t5 + 10	
11)	t7 = t6 + 10	
12)	t8 = t6 + t7	B4
13)	t6 = 3	
14)	t5 = 100	

Граф потока управления трёхадресного кода



v

B4

Пример работы

B1

`e_gen = {l1, l2}, e_kill = {l3, l6, l7}`

`IN = {}, OUT = {l1, l2}`

B2

`e_gen = {l3}, e_kill = {l2, l6, l7}`

`IN = {l1, l2}, OUT = {l1, l3}`

B3

`e_gen = {l6}, e_kill = {l1, l2, l7, l10}`

`IN = {l1, l3}, OUT = {l3, l6}`

B4

`e_gen = {}, e_kill = {l10, l11, l12}`

`IN = {l3, l6}, OUT = {l3, l6}`

Название задачи

Генерация IL-кода #46

Постановка задачи

Необходимо создать транслятор трёхадресного кода в IL-код, а также иметь возможность вывести его или выполнить.

Зависимости задач в графе задач

Зависит от: Трёхадресный код

Теоретическая часть задачи

IL-код – промежуточный язык, используемый в платформе .NET, так же называемый «высокоуровневый ассемблер» виртуальной машины .NET, в который переводятся все программы написанные на этой платформе. IL-код состоит из операций работы со стеком (положить на стек значение, сложить два числа с вершины стека и т.д.) и операций передачи управления (операции перехода по меткам). Так как операции и il-коде очень просты, необходимо было каждую инструкцию трёхадресного кода перевести в набор инструкций из IL-кода.

Практическая часть задачи (реализация)

Из-за объёма приведём только интерфейс.

Класс транслятор

```
public class TAcod2ILcodeTranslator
{
    public void Translate(TACode tACode)
    public void RunProgram()
    public string PrintCommands()
}
```

Удобная обёртка над командами генерации il-кода.

```
class GenCodeCreator
{
    public void Emit(OpCode op)
    public void Emit(OpCode op, int num)
    public void Emit(OpCode op, LocalBuilder lb)
    public void Emit(OpCode op, Label l)
    public LocalBuilder DeclareLocal(Type t)
    public Label DefineLabel()
    public void MarkLabel(Label l)
    public void EmitWriteLine()
    public void EndProgram()
    public void RunProgram()
    public void WriteCommandsOn()
    public void WriteCommandsOff()
}
```

Пример работы.

Код программы:

```
a=10;
b=20;
if (a<b)
{
c=30;
}
else
{
c=40;
}
print(c);
```

Трёхадресный код:

```
l1 : t0 = 10
l0 : a = 10
l3 : t1 = 20
l2 : b = 20
l6 : t3 = 10
l7 : t4 = 20
l5 : t2 = 10 < 20
l4 : if t2 goto l11
l9 : t5 = 40
l8 : c = 40
l10 : goto l15
```

```
l11 : nop
l14 : t6 = 30
l13 : c = 30
l15 : nop
l18 : t7 = c
l17 : print t7
```

IL-код:

```
nop
DefineLabel Label0
DefineLabel Label1
DeclareLocal var0: System.Int32
ldc.i4 10
stloc var0
DeclareLocal var1: System.Int32
ldc.i4 10
stloc var1
DeclareLocal var2: System.Int32
ldc.i4 20
stloc var2
DeclareLocal var3: System.Int32
ldc.i4 20
stloc var3
DeclareLocal var4: System.Int32
ldc.i4 10
stloc var4
```

```
DeclareLocal var5: System.Int32
```

```
ldc.i4 20
```

```
stloc var5
```

```
DeclareLocal var6: System.Int32
```

```
ldc.i4 10
```

```
ldc.i4 20
```

```
clt
```

```
stloc var6
```

```
DefineLabel Label2
```

```
ldloc var6
```

```
brfalse Label2
```

```
br Label0
```

```
nop
```

```
MarkLabel Label2
```

```
DeclareLocal var7: System.Int32
```

```
ldc.i4 40
```

```
stloc var7
```

```
DeclareLocal var8: System.Int32
```

```
ldc.i4 40
```

```
stloc var8
```

```
br Label1
```

```
nop
```

```
MarkLabel Label0
```

```
nop
```

```
DeclareLocal var9: System.Int32
```

```
ldc.i4 30
```

```
stloc var9
```

```
ldc.i4 30
```

```
stloc var8
```

```
nop
```

```
MarkLabel Label1
```

```
nop
```

```
DeclareLocal var10: System.Int32
```

```
ldloc var8
```

```
stloc var10
```

```
ldloc var10
```

```
WriteLine
```

```
ret
```

Название задачи

Создание простого аналога среды разработки (IDE)

Постановка задачи

Создать среду для проверки работы компилятора со следующими возможностями:

- интерактивный ввод и загрузка исходного кода из файла;
- сборка программы в исполняемый файл, в т.ч.:
 - запуск синтаксического разбора программы;
 - запуск генерации трехадресного кода;
 - запуск генерации графа потока управления;
 - запуск генерации IL-кода;
- вывод информации об успешности или неудачности какой-либо операции этапа сборки проекта;
- возможность запустить сборку программы с оптимизациями, которые задаются пользователем в интерактивном режиме;
- запуск итоговой программы с возможности вывода результата;
- вывод вспомогательной информации, такой как:
 - сгенерированный трехадресный код;
 - сгенерированный IL-код;
 - визуальное представление абстрактного семантического дерева программы;
 - визуальное представление графа потока управления программы;
 - результаты работы итеративных алгоритмов.

Зависимости задач в графе задач

Нет зависимостей.

Теоретическая часть задачи

Нет теоретической части.

Практическая часть задачи (реализация)

Для написания программы был выбран `WinForms32`. При проектировании программы был применен паттерн `MVC`. Данное решение позволило сделать код чистым и легко редактируемым, так как вся логика была вынесена в отдельные обработчики, которые связывались с визуальным представлением при помощи механизма событий.

Для вывода графических представлений AST и CFG был использован сторонний инструмент `Graphviz`. Данная программа (разработка ведется с 1988 года) давно зарекомендовала себя как мощный инструмент, с помощью которого можно достаточно быстро получить визуальное представление для графов любой сложности. Граф строится по текстовому описанию, известному как `DOT` формат.

В случае с CFG генерация текстового описания графа достаточно прямолинейная и простая. Вершины представляют собой базовые блоки, а дуги берутся из поля `Children`.

Для генерации AST файла был написан класс `AstGraphvizVisitor`,

реализующий интерфейс `IVisitor`, в котором при посещении каждого узла происходит запись информации в объект `StringBuilder`, в котором, по окончании обхода, содержится контент `DOT` файла.

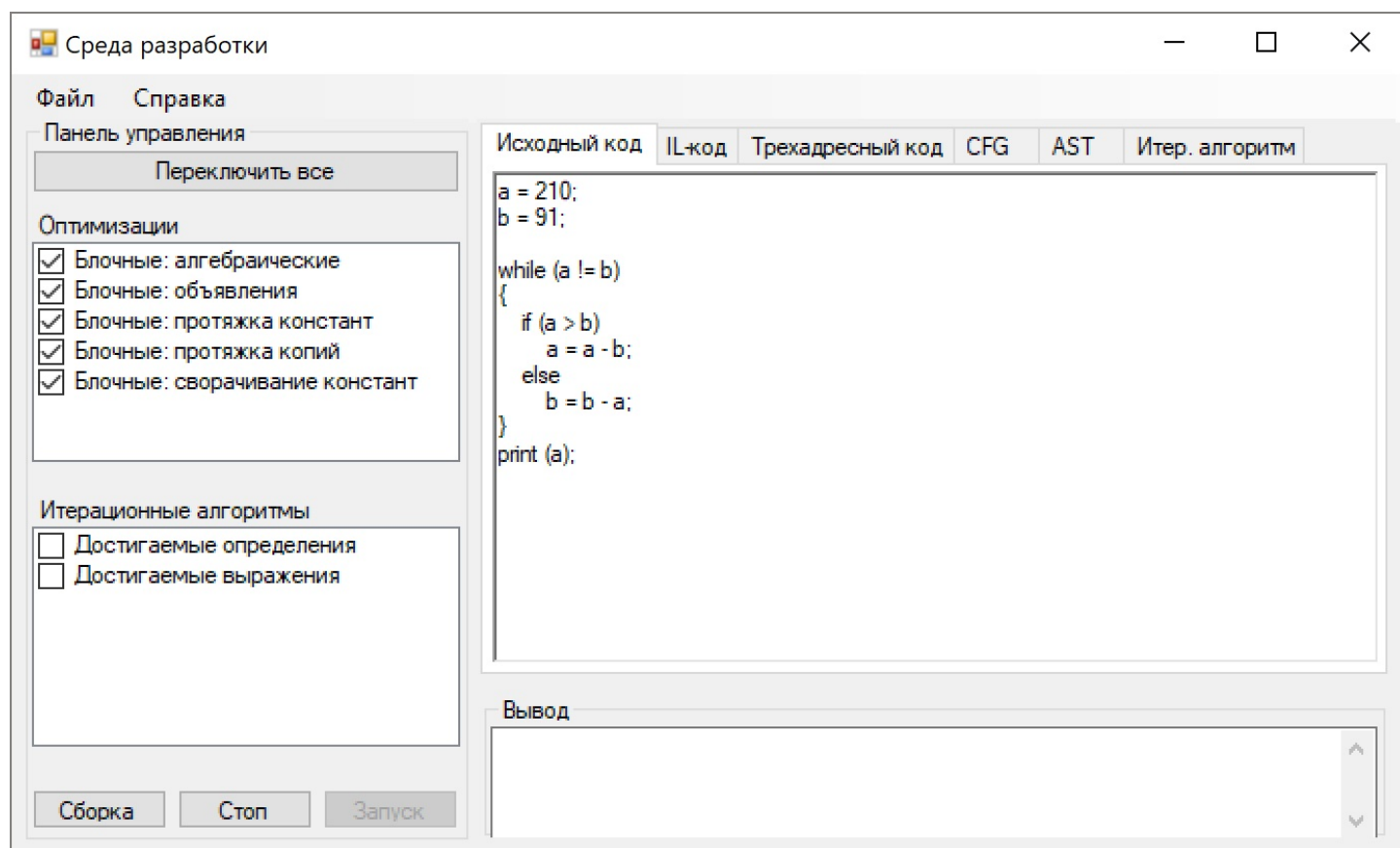
Для вызова `Graphviz` была применена легковесная обертка `GraphvizWrapper.NET`. Использование обертки позволило избежать ручного вызова процессов `Graphviz` с передачей `DOT` файла и перехватом потока вывода процесса.

Тесты

В силу простоты программы применялось ручное тестирование

Пример работы

Окно программы



Вывод CFG

Среда разработки

Файл Справка

Панель управления

Переключить все

Оптимизации

- ☒ Блочные: алгебраические
- ☒ Блочные: объявления
- ☒ Блочные: протяжка констант
- ☒ Блочные: протяжка копий
- ☒ Блочные: сворачивание констант

Итерационные алгоритмы

- ☐ Достигаемые определения
- ☐ Достигаемые выражения

Сборка Стоп Запуск

Исходный код IL-код Трехадресный код **CFG** AST Итер. алгоритм

```
graph TD; B10["10 : goto 131"] --> B14["14 : nop  
t3 = a  
t4 = b  
t2 = a := b  
if t2 goto 111"]; B14 --> B11["11 : nop  
t6 = a  
t7 = b  
t5 = a > b"]; B11 --> B14;
```

Save

Вывод

Граф потока управления построен
Создание трехадресного кода завершено

Вывод AST

Среда разработки

Файл Справка

Панель управления

Переключить все

Оптимизации

- ☒ Блочные: алгебраические
- ☒ Блочные: объявления
- ☒ Блочные: протяжка констант
- ☒ Блочные: протяжка копий
- ☒ Блочные: сворачивание констант

Итерационные алгоритмы

- ☐ Достигаемые определения
- ☐ Достигаемые выражения

Сборка Стоп Запуск

Исходный код IL-код Трехадресный код CFG **AST** Итер. алгоритм

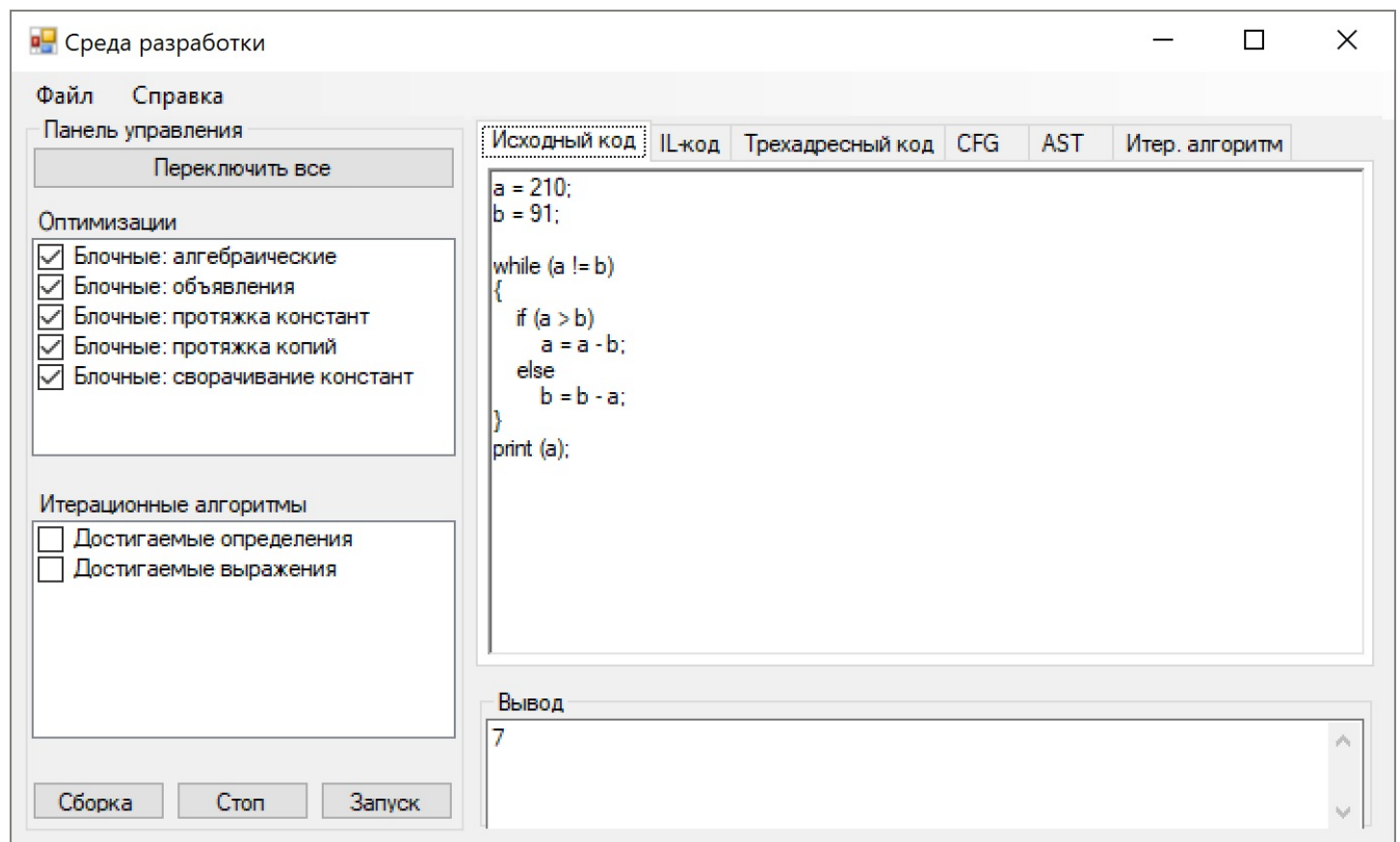
```
graph TD; Root[Block] --> A1[Assign a = expr]; Root --> A2[Assign b = expr]; Root --> C[Cycle]; Root --> P[Print. ex]; A1 --> I1a[Id: a]; A1 --> I1v[Int: 210]; A2 --> I2b[Id: b]; A2 --> I2v[Int: 91]; C --> B1[BinOp: NotEqual]; C --> B2[Block]; B1 --> I3a[Id: a]; B1 --> I3b[Id: b]; B2 --> B3[If: else is present]; B2 --> B4[BinOp: Greater]; B3 --> A3[Assign a = expr];
```

Save

Вывод

Граф потока управления построен
Создание трехадресного кода завершено

Вывод результата запуска



Название задачи

Оптимизация “Распространение констант” между базовыми блоками

Постановка задачи

Реализовать протяжку констант, но уже между базовыми блоками.

Зависимости задач в графе задач

Зависит от:

- Базовые структуры и итерационный алгоритм для распространения констант

Теоретическая часть задачи

Необходимо разрешить оптимизацию распространения констант между базовыми блоками. Для этого нужно получить данные из итерационного алгоритма для распространения констант, а затем правильно их применить для трёхадресного кода, чтобы в итоге код стал оптимизированным. Итерационный алгоритм присваивает переменным одно из трёх значений: Not A Constant, Undefined или IsConstant.

Практическая часть задачи (реализация)

```
public InOutData<Dictionary<Guid, VarValue>> TempFunc
(TACode taCode, ControlFlowGraph cfg)
{
```

```

        Operations ops = new Operations(taCode);
        TransferFunction f = new TransferFunction();

        IterativeAlgorithm itAlg = new IterativeAlgorithm
        ();

        var result = itAlg.Analyze(cfg, ops, f);

        return result;
    }
    public TACode Optimize(TACode taCode, out bool applied)
    {
        var app = false;
        var visited = new Dictionary<Guid, bool>();
        ControlFlowGraph cfg = new ControlFlowGraph(taCode);

        var ioData = TempFunc(taCode, cfg);

        foreach (var node in taCode.CodeList.ToList().OfType<Assign>())
            visited[node.Result.Id] = false;

        for (int j = taCode.CodeList.Count() - 1; j > 0; j--)
        {
            var node = taCode.CodeList.ElementAt(j) as Assign;

            if (node != null)

```

```

        {
            for (int i = 0; i < cfg.CFGNodes.Count();
i++)
            {
                if (ioData[cfg.CFGNodes.ElementAt(i)]
.Item1.ContainsKey(node.Result.Id) && ioData[cfg.CFGNodes.Ele
mentAt(i)].Item1[node.Result.Id].varType is VarValue.Type.CON
ST)
                {
                    if (visited[node.Result.Id] == tr
ue)
                        break;
                    visited[node.Result.Id] = true;
                    node.Right = ioData[cfg.CFGNodes.
ElementAt(i)].Item1[node.Result.Id].value;
                    node.Left = null;
                    node.Operation = OpCode.Copy;
                    taCode.CodeList[j] = node;
                }
            }
        }
    }

    applied = app;
    return taCode;
}

```

Тесты

Например, есть такой фрагмент кода, состоящий из двух базовых блоков:

```
a = 91;  
b = 5;  
  
goto h;  
  
h: c = b - 1;
```

После глобальной оптимизации распространения констант должно получиться следующее:

```
a = 91;  
b = 5;  
  
goto h;  
  
h: c = 5 - 1;
```

Пример работы.

Большой пример, демонстрирующий алгоритм.

Название задачи

Базовые структуры и итерационный алгоритм для распространения констант

Постановка задачи

Реализовать структуры для задачи распространения констант и применить итерационный алгоритм

Зависимости задач в графе задач

- Интерфейс передаточной функции
- Передаточная функция и генерация множеств `gen` и `kill`
- Структура базовых блоков

Теоретическая часть задачи

Каждая переменная в некоторой таблице имеет одно из значений в полурешетке - *UNDEF* (undefigned), *const*, *NAC* (not a const). Таблица является декартовым произведением полурешеток, и следовательно, сама полурешетка.

Таким образом, элементом данных будет отображение m на соответствующее значение полурешетки.

1. Если s не является присваиванием, то f тождественна, т.е. $m = m'$
2. Если s присваивание, то для каждого $v \neq x$: $m'(v) = m(v)$
3. Если s присваивание константе, то $m'(x) = \text{const}$
4. Если $s: x = y + z$, то
 - $m'(x) = m(y) + m(z)$, если $m(y)$ и $m(z)$ - const
 - $m'(x) = \text{NAC}$, если $m(y)$ или $m(z)$ - NAC
 - $m'(x) = \text{UNDEF}$ в остальных случаях

Функция монотонна, потому итерационный процесс решает поставленную задачу.

Практическая часть задачи (реализация)

Необходимо описать операции, передаточную функцию и итерационный алгоритм. Для этого предварительно был реализован класс *VarValue*, который хранит тип переменных (*UNDEF*, *const*, *NAC*), их значения если это константа, и операции над ними.

```
public class VarValue
{
    public enum Type { UNDEF, CONST, NAC };

    public Type varType; // Операнд определяющий тип переменной

    public IntConst value; // Операнд со значением константы, если тип переменной CONST
```

```
// Три вида конструктора для инициализации значений в  
различных ситуациях
```

```
public VarValue()  
{  
    varType = Type.UNDEF;  
    value = null;  
}
```

```
public VarValue(IntConst c)  
{  
    varType = Type.CONST;  
    value = c;  
}
```

```
public VarValue(Var v)  
{  
    varType = Type.NAC;  
    value = null;  
}
```

```
//реализация оператора сбора
```

```
public VarValue CollectionOperator(VarValue right)  
{  
    if (right == null || this == right)  
        return this;  
    if (this.varType == Type.NAC || right.varType ==  
Type.NAC)  
        return new VarValue(new Var());
```

```
        if (this.varType == Type.CONST && right.varType == Type.CONST)

            return new VarValue(new Var());

        if (this.varType == Type.CONST)

            return this;

        return right;

    }
```

// функция-обертка для применения операций над переменными

```
    public static VarValue UseOperation(VarValue left, VarValue right, OpCode code)

    {

        if (left.varType == Type.CONST && right.varType == Type.CONST)

            return new VarValue(ApplyOperation(left.value, right.value, code));

        return OperationUnderNotConst(left, right);

    }
```

// применение операций над не константными значениями

```
    private static VarValue OperationUnderNotConst(VarValue left, VarValue right)

    {

        if (left.varType == Type.NAC || right.varType == Type.NAC)

            return new VarValue(new Var());

        if (left.varType == Type.UNDEF || right.varType ==
```

```
= Type.UNDEF)
```

```
    return new VarValue();
```

```
    return null;
```

```
}
```

```
// применение операций над константами
```

```
private static IntConst ApplyOperation(IntConst left,  
IntConst right, OpCode op)
```

```
{
```

```
    switch (op)
```

```
    {
```

```
        case OpCode.Plus: return left + right;
```

```
        case OpCode.Minus: return left - right;
```

```
        case OpCode.Mul: return left * right;
```

```
        case OpCode.Div: return left / right;
```

```
        default: return left;
```

```
    }
```

```
}
```

```
// функция сравнения для перегрузки операторов сравне  
ния
```

```
public override bool Equals(object obj)
```

```
{
```

```
    if (ReferenceEquals(null, obj)) return false;
```

```
    if (ReferenceEquals(this, obj)) return true;
```

```
    if (obj.GetType() != this.GetType()) return false
```

```
;
```

```
    if (varType == ((VarValue)obj).varType)
```

```

        {
            if (varType == Type.CONST && value != ((VarValue)obj).value)
                return false;
            return true;
        }
        return false;
    }

    public static bool operator ==(VarValue left, VarValue right)
    {
        return Equals(left, right);
    }

    public static bool operator !=(VarValue left, VarValue right)
    {
        ...
    }

```

Далее описываются классы *Operations*, *TransferFunction*, *IterativeAlgorithm*, которые реализуют интерфейс *ILatticeOperations*, *ITransferFunction* и *IAlgorithm* соответственно.

Класс *Operations* описывает структуру полурешетки: в качестве таблиц используется структура *Dictionary<Guid, VarValue>* ставящая в соответствие каждой переменной ее состояние (текущий тип). В

качестве нижней границы все значения словаря инициализируются как NAC, а в качестве верхней UNDEF. Так же реализован метод применения оператора сбора:

```
public Dictionary<Guid, VarValue> Operator(Dictionary<Guid, VarValue> a, Dictionary<Guid, VarValue> b)
{
    var aCopy = a.ToDictionary(entry => entry.Key, entry => entry.Value);
    var bCopy = b.ToDictionary(entry => entry.Key, entry => entry.Value);
    Dictionary<Guid, VarValue> result = new Dictionary<Guid, VarValue>();
    foreach (var key in aCopy.Keys)
    {
        result[key] = a[key].CollectionOperator(b[key]);
        if (bCopy.ContainsKey(key))
            bCopy.Remove(key);
    }
    foreach (var key in bCopy.Keys)
        result.Add(key, bCopy[key]);
    return result;
}
```

Класс *TransferFunction* непосредственно реализует функцию f описанную в теоретической части, *IterativeAlgorithm* описывается как

и базовый, с учетом заданной структуры данных.

Название задачи

Нумерация ББЛ в порядке “Обращение обратного порядка обхода”.

Постановка задачи

Реализовать возможность нумерации ББл в порядке “Обращение обратного порядка обхода”.

Зависимости задач в графе задач

Зависит от:

- Control Flow Graph

От задачи зависит:

- Модифицированный итерационный алгоритм с нумерацией базовых блоков и подсчётом количества итераций

Теоретическая часть задачи

Важный для анализа графа потока вариант - *упорядочивание в глубину* (depth-first ordering), которое представляет собой обращение обратного порядка обхода. Иначе говоря, при упорядочивании в глубину мы посещаем узел, затем обходим его крайний слева узел-преемник, после этого - узел, затем обходим его крайний справа узел-преемник, после этого - узел, расположенный слева от него, и т.д. Однако, перед тем как строить дерево для графа потока, следует выбрать, какой из преемников является крайним справа, какой - его левым соседом и т.д.

Практическая часть задачи (реализация)

```
// Методы расширения для удобного использования нумерации базовых блоков

public static class GraphNumExt
{
    // Обращает порядок нумерации
    public static IGraphNumerator Reverse(this GraphNumerator n, ControlFlowGraph g)
        => new ReverseNum(g, n);

    // Нумерация в обратном порядке
    public static NumeratedGraph BackOrder(this TACode code)
    {
        var graph = new NumeratedGraph(code, null);
        graph.Numerator = GraphNumerator.BackOrder(graph);
        ;

        return graph;
    }

    // Нумерация в прямом порядке
    public static NumeratedGraph StraightOrder(this TACode code)
```

```

    {
        var graph = new NumeratedGraph(code, null);
        graph.Numerator = GraphNumerator.BackOrder(graph)
.Reverse(graph);
        return graph;
    }

    // Реализация IGraphNumerator для обращения порядка н
умерации
    private class ReverseNum : IGraphNumerator
    {
        private readonly ControlFlowGraph _graph;
        private readonly IGraphNumerator _num;

        public ReverseNum(ControlFlowGraph graph, IGraphN
umerator num)
        {
            _graph = graph;
            _num = num;
        }

        public int? GetIndex(BasicBlock b)
        {
            var ind = _num.GetIndex(b);
            if (ind == null) return null;

            return _graph.CFGNodes.Count() - ind.Value;
        }
    }

```

```
}
```

```
}
```

```
// Реализация нумерации в обратном порядке
```

```
public class GraphNumerator : IGraphNumerator
```

```
{
```

```
    public static GraphNumerator BackOrder(ControlFlowGraph graph)
```

```
    {
```

```
        var root = graph.CFGNodes.ElementAt(0);
```

```
        var num = new GraphNumerator();
```

```
        var index = 0;
```

```
        var openSet = new HashSet<BasicBlock>();
```

```
        void Iter(BasicBlock node)
```

```
        {
```

```
            openSet.Add(node);
```

```
            var children = node.Children;
```

```
            foreach(var c in children.Where(x => !openSet  
.Contains(x)))
```

```
                Iter(c);
```

```
            num._num[node] = index++;
```

```
        }
```

```
        Iter(root);
```

```
        return num;
```

```
}
```

```
        private readonly Dictionary<BasicBlock, int> _num = new Dictionary<BasicBlock, int>();
```

```
        public virtual int? GetIndex(BasicBlock b)
        => _num.TryGetValue(b, out var res) ?
            new int?(res) : null;
    }
```

```
// Граф с нумерованными базовыми блоками
```

```
public class NumeratedGraph : ControlFlowGraph
{
    public NumeratedGraph(TACode code, IGraphNumerator numerator) : base(code)
    {
        Numerator = numerator;
    }

    public IGraphNumerator Numerator { get; set; }

    public int? IndexOf(BasicBlock b) => Numerator.GetIndex(b);

    private string NodeToString(BasicBlock n)
    {
        var blockName = TACodeNameManager.Instance[n.BlockId];

        var index = Numerator?.GetIndex(n);
```

```

        return $"({index}:{blockName})";
    }

    public override string ToString()
    {
        var s = new StringBuilder();

        foreach (var n in CFGNodes)
            s.AppendLine(
                $"{NodeToString(n)} : [{ String.Join(
                    ", ", n.Children.Select(NodeToString)) }]"
            );

        return s.ToString();
    }
}

// Интефейс для любых нумераций CFG
public interface IGraphNumerator
{
    int? GetIndex(BasicBlock b);
}

```

Тесты

```

a = 1;
goto h;
h: b = 1;

```

```
goto h2;
h2: c = 1;
d = 1;
```

```
numer[0] == cfg[0]
numer[1] == cfg[1]
numer[2] == cfg[2]
```

где **numer** - нумератор, **cfg** - граф потока управления.

Результат работы - перенумерованные блоки от 0 до 2 сверху вниз.

Пример работы

```
var cfg = new ControlFlowGraph(tacodeInstance);
var numer = GraphNumerator.BackOrder(cfg);
Assert.AreEqual(0, numer.GetIndex(cfg.CFGNodes.ElementAt(0)))
;
Assert.AreEqual(1, numer.GetIndex(cfg.CFGNodes.ElementAt(1)))
;
Assert.AreEqual(2, numer.GetIndex(cfg.CFGNodes.ElementAt(2)))
;
```

Название задачи

Проверка CFG на приводимость

Постановка задачи

Необходимо проверить CFG на приводимость

Зависимости задач в графе задач

Зависит от:

- Классификация ребер CFG по Глубинному остовному дереву
- Дерево доминаторов

Теоретическая часть задачи

Алгоритм проверки основан на следующей теореме:

Теорема о приводимости графа потока управления:

Граф потока управления является приводимым, если для любого глубинного остовного дерева этого графа все отступающие(retreating) ребра являются обратными(back).

Практическая часть задачи (реализация)

Алгоритм проверки CFG на приводимость:

- Получить список всех отступающих(retreating) ребер CFG по Глубинному остовному дереву
- Построить дерево доминаторов
- Проверить, что конец всех отступающих ребер доминирует над их началом с помощью дерева доминаторов

Пример работы.

Исходный код программы с неприводимым CFG:

```
if (1 < -3)
    a = 123;
else
    goto h;
b = 777;
for(i = 0, 10)
{
    print(1 >= 3);
    h: {c = a + b;}
    if (1 + 3)
    {
        a = 1;
    }
}
```

Ответ: false.

Исходный код программы с приводимым CFG:


```
a = 2;
while(3)
{
    b = 10;
    goto h;
}
for(i = 0, 10, 1 + 1)
    print(1,2,3);
h: {c = a + b;}
for(i = 0, 10)
{
    print(1 >= 3);
    if (1 + 3)
    {
        a = 1;
    }
}
```

Ответ: true.

Название задачи

Расчет глубины CFG

Постановка задачи

Необходимо рассчитать глубину CFG

Зависимости задач в графе задач

Зависит от:

- Классификация ребер CFG по Глубинному остовному дереву

Теоретическая часть задачи

Глубина графа потока равна наибольшему числу отступающих ребер вдоль любого ациклического пути.

Практическая часть задачи (реализация)

Алгоритм нахождения глубины дерева:

- Запустить классификатор ребер
- Выполнить обход графа в глубину по вершинам, увеличивая значение максимальной глубины для текущего пути на один при встрече отступающего ребра из текущей вершины
- Вернуть максимальное значение глубины для всех путей

Пример работы.

Отсутствует

Название задачи

Модифицировать итерационный Алгоритм с нумерацией базовых блоков и подсчётом количества итераций.

Постановка задачи

Модифицировать итерационный Алгоритм с нумерацией базовых блоков и подсчётом количества итераций.

Зависимости задач в графе задач

Задача зависит от:

- Обобщенный итерационный алгоритм
- От задачи зависит:
- Тестирование

Теоретическая часть задачи

При использовании порядка “вглубь” количество проходов, необходимое для распространения любого достигающего определения вдоль любого ациклического пути, не более чем на единицу превышает число ребер пути, идущих от блока с большим номером к блоку с меньшим. Эти ребра являются отступающими, так что необходимое количество проходов на единицу больше глубины. Предыдущему алгоритму для того, чтобы определить достижение всех определений, требуется один дополнительный проход, так что верхняя граница количества проходов, необходимых алгоритму с упорядочиванием

блоков вглубь, в действительности на 2 превышает глубину. Изучение показало, что средняя глубина типичного графа потока равна 2.75. Таким образом, алгоритм сходится очень быстро.

Практическая часть задачи (реализация)

```
public class OptimizedGenericIterativeAlgorithm<T> : IAlgorithm<T>
{
    public Func<T, T, bool> Comparer { get; set; }
    public Func<(T, T)> Fill { get; set; }
    public Func<(T, T), (T, T), bool> Finish { get; set; }
}

// Нумерация графа (для оптимизации передается в порядке обратного обхода)
public IGraphNumerator Numerator { get; set; }

// Счётчик операций
public int OpsCount { get; set; }

public InOutData<T> Analyze(
    ControlFlowGraph graph,
    ILatticeOperations<T> ops,
    ITransferFunction<T> f)
{
    var data = new InOutData<T>
    {
```

```

        [graph.CFGNodes.ElementAt(0)] = Fill()
    };

    foreach (var node in graph.CFGNodes)
        data[node] = Fill();
    OpsCount = 0;
    var outChanged = true;
    while (outChanged)
    {
        outChanged = false;
        foreach (var block in graph.CFGNodes.OrderBy(
x => Numerator.GetIndex(x))) // упорядочивание базовых блоков
        {
            OpsCount++; // увеличение счетчика операций

            var inset = block.Parents.Aggregate(ops.Lower, (x, y)
=> ops.Operator(x, data[y].Item2));
            var outset = f.Transfer(block, inset, ops);

            if (!Finish((inset, outset), data[block]))
            {
                outChanged = true;
            }
            data[block] = (inset, outset);
        }
    }

```

```
    }  
    return data;  
  }  
}
```

Тесты

Тестирование проводилось в другом задании.

Название задачи

Примеры неправильных нумераций с max кол-вом итераций #66

Постановка задачи

Привести несколько примеров неправильной нумерации базовых блоков, из-за которой число итераций какого-либо итерационного алгоритма будет максимальным.

Зависимости задач в графе задач

Зависит от: Трехадресный код

Теоретическая часть задачи

Проблема нумерации базовых блоков естественным образом всплывает, в тот момент, когда мы начинаем задумываться о эффективности итерационного алгоритма. Действительно если рассмотреть, например, такую нумерацию 3->5->12->2->6->8->7->15, где стрелки показывают, как блоки идут друг за другом в коде, а номера тот порядок, в котором эти блоки будут обходиться. Тогда на первой итерации верные значения распространятся только в блоки 3,5,12. На второй в 3,5,12,2,6,8. На третьей во все. И четвёртая итерация понадобится, чтобы проверить, что распространившиеся значения не поменялись.

Максимальное количество итераций, которое может сделать алгоритм равно произведению числа базовых блоков на высоту полурешётки. Но

эта оценка неточная, поэтому, например, для алгоритма достигающих определений, максимальным числом итераций будет максимальная длина пути распространения некоторого определения в базовых блоках, среди всех определений всех переменных. Пронумеровав блоки вдоль этого пути в обратном порядке, мы заставим итерационный алгоритм выполняться максимальное число итераций.

В примере, показанном ниже оказалось достаточно пронумеровать все вершины в обратном порядке, чтобы получить максимальное число итераций 6, увеличив его с изначальных трёх.

Практическая часть задачи (реализация)

```
var taCode = new TACode();
var ass1 = new Assign
{
    Left = new IntConst(10),
    Operation = OpCode.Copy,
    Result = new Var()
};
var ass2 = new Assign
{
    Left = new IntConst(10),
    Operation = OpCode.Mul,
    Right = new IntConst(3),
    Result = new Var()
};
ass2.IsLabeled = true;
```

```
var ass3 = new Assign
{
    Left = ass2.Result,
    Operation = OpCode.Plus,
    Right = ass1.Result,
    Result = new Var()
};
ass3.IsLabeled = true;
var ass4 = new Assign
{
    Operation = OpCode.Copy,
    Right = new IntConst(0),
    Result = new Var()
};
var ass5 = new Assign
{
    Operation = OpCode.Copy,
    Right = new IntConst(14),
    Result = ass4.Result
};
var ass6 = new Assign
{
    Operation = OpCode.Minus,
    Left = ass3.Result,
    Right = ass2.Result,
    Result = ass4.Result
};
ass6.IsLabeled = true;
```

```
var ass7 = new Assign
{
    Operation = OpCode.Plus,
    Right = new IntConst(1),
    Left = ass1.Result,
    Result = ass1.Result
};
ass7.IsLabeled = true;
var ifgt1 = new IfGoto
{
    Condition = new IntConst(1),
    TargetLabel = ass6.Label
};
var ifgt2 = new IfGoto
{
    Condition = new IntConst(2),
    TargetLabel = ass3.Label
};
var ifgt3 = new IfGoto
{
    Condition = new IntConst(3),
    TargetLabel = ass2.Label
};
var goto1 = new Goto
{
    TargetLabel = ass7.Label
};
var empty = new Empty { };
```

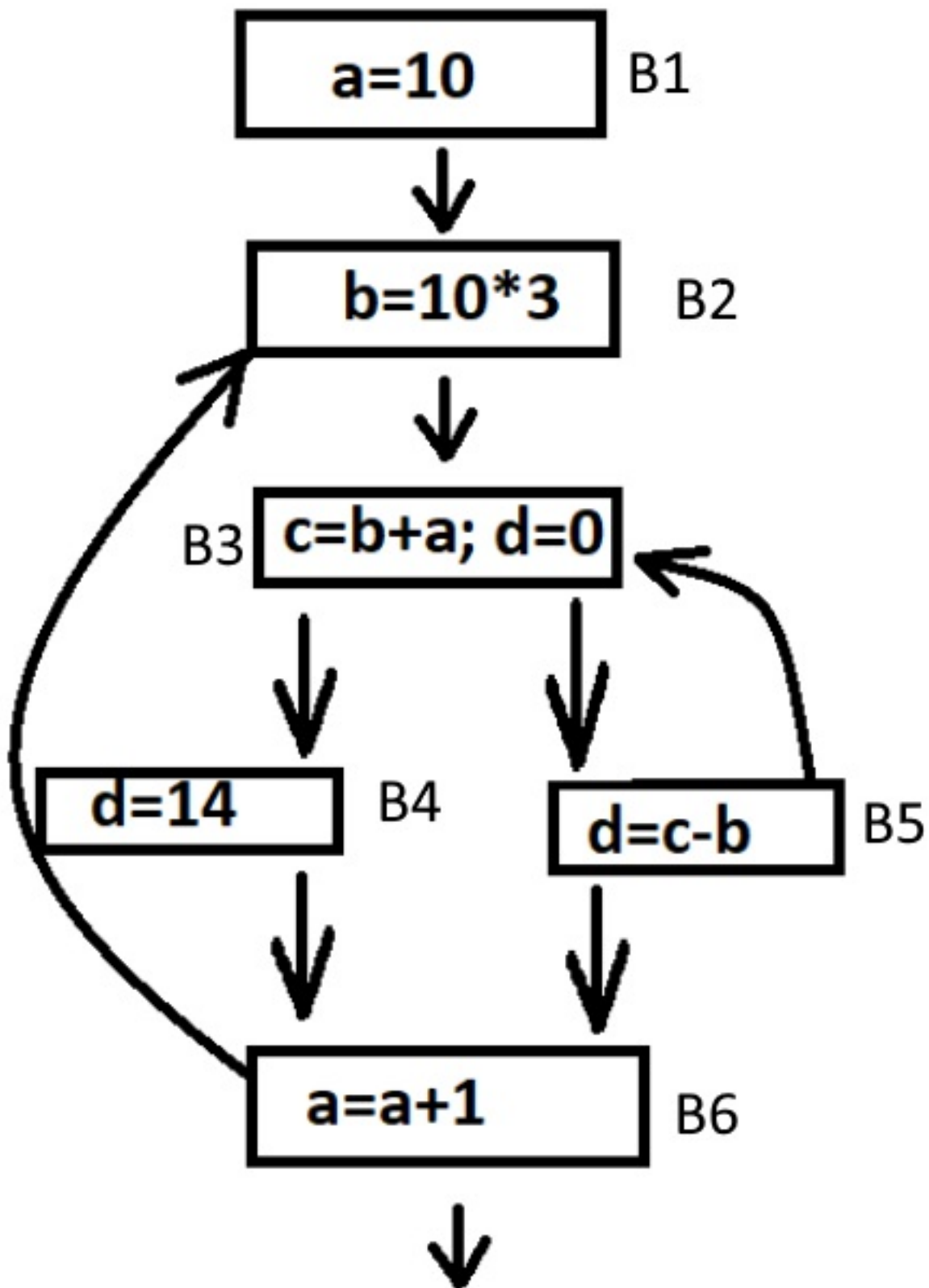
```
taCode.AddNode(ass1);
taCode.AddNode(ass2);
taCode.AddNode(ass3);
taCode.AddNode(ass4);
taCode.AddNode(ifgt1);
taCode.AddNode(ass5);
taCode.AddNode(goto1);
taCode.AddNode(ass6);
taCode.AddNode(ifgt2);
taCode.AddNode(ass7);
taCode.AddNode(ifgt3);
taCode.AddNode(empty);
var cfg = new ControlFlowGraph(taCode);
taCode.CreateBasicBlockList();
var op = new Operations(taCode);
var tf = new TransferFunction(taCode);
var algo = new GenericIterativeAlgorithm<HashSet<Guid>>()
{
    Finish = (a, b) =>
    {
        var (a1, a2) = a;
        var (b1, b2) = b;

        return !a2.Except(b2).Any();
    },
    Comparer = (x, y) => !x.Except(y).Any(),
    Fill = () => (op.Lower, op.Lower),
    DebugToString = (x) => x.Aggregate("", (s, y) => s + ", "
```

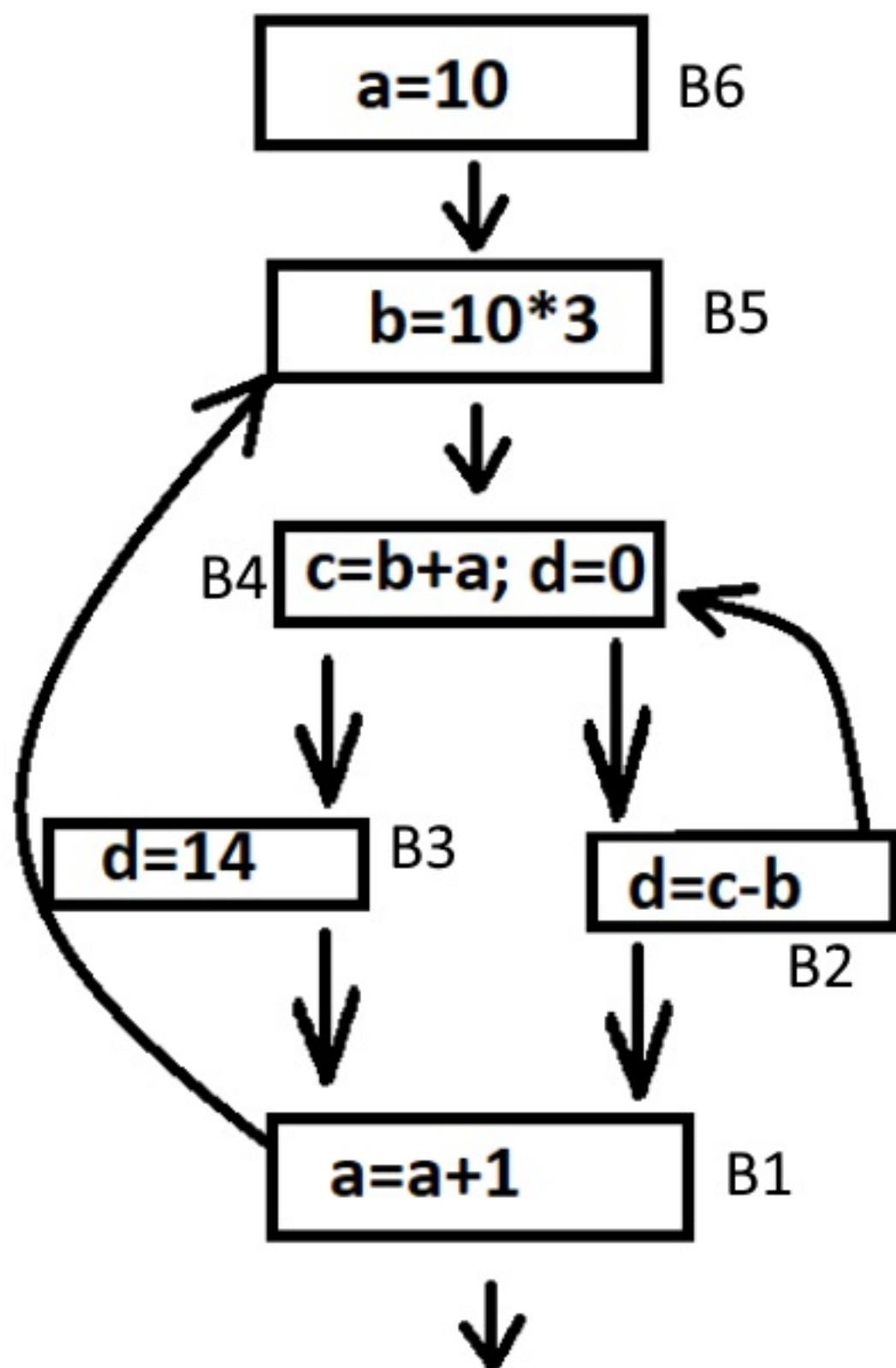
```
+ TACodeNameManager.Instance[y])  
};  
var inout = algo.Analyze(cfg, op, tf);  
Assert.True(algo.CountOfDoneIterations == 3);  
cfg.ReverseCFGNodes();  
inout = algo.Analyze(cfg, op, tf);  
Assert.True(algo.CountOfDoneIterations == 6);
```

Пример работы.

Граф программы, правильная нумерация блоков с числом итераций - 3.



Неправильная нумерация блоков с максимальным числом итераций - 6.



Название задачи

Обобщённый итерационный алгоритм

Постановка задачи

Реализовать обобщённый итерационный алгоритм

Зависимости задач в графе задач

Задача зависит от:

- Control Flow Graph
- Интерфейс передаточной функции

Теоретическая часть задачи

- 1) $OUT[V_{XOD}] = v_{Вход};$
- 2) **for** (каждый базовый блок B , отличный от входного) $OUT[B] = T;$
- 3) **while** (внесены изменения в OUT)
- 4) **for** (каждый базовый блок B , отличный от входного) {
- 5) $IN[B] = \bigwedge_{P-\text{предшественник } B} OUT[P];$
- 6) $OUT[B] = f_B(IN[B]);$
- }

а) Итеративный алгоритм для прямой задачи потока данных

- 1) $IN[V_{YXOD}] = v_{Выход};$
- 2) **for** (каждый базовый блок B , отличный от выходного) $IN[B] = T;$
- 3) **while** (внесены изменения в IN)
- 4) **for** (каждый базовый блок B , отличный от выходного) {
- 5) $OUT[B] = \bigwedge_{S-\text{преемник } B} IN[S];$
- 6) $IN[B] = f_B(OUT[B]);$
- }

б) Итеративный алгоритм для обратной задачи потока данных

Рис. 9.23. Прямая и обратная версии итеративного алгоритма

Практическая часть задачи (реализация)

```
public class GenericIterativeAlgorithm<T> : IAlgorithm<T>
{
    public Func<T, T, bool> Comparer { get; set; }
    public Func<(T, T)> Fill { get; set; } // начальное з
начение (значение первого узла)
    public Func<(T,T), (T,T), bool> Finish { get; set; }
// условие завершения итерационного алгоритма

    public InOutData<T> Analyze(
        ControlFlowGraph graph,
```

```

ILatticeOperations<T> ops,
ITransferFunction<T> f)
{
    var data = new InOutData<T>();
    data[graph.CFGNodes.ElementAt(0)] = Fill();

    foreach (var node in graph.CFGNodes)
        data[node] = Fill();

    var outChanged = true;
    while (outChanged)
    {
        outChanged = false;
        foreach (var block in graph.CFGNodes)
        {
            var inset = block.Parents.Aggregate(ops.L
ower, (x, y)
                => ops.Operator(x, data[y].Item2));
            var outset = f.Transfer(block, inset, ops
);
            if (!Finish((inset,outset), data[block]))
            {
                outChanged = true;
            }
            data[block] = (inset, outset);
        }
    }
    return data;
}

```

```
}  
}
```

Тесты

Для тестирования использовался случай достигающих определений.

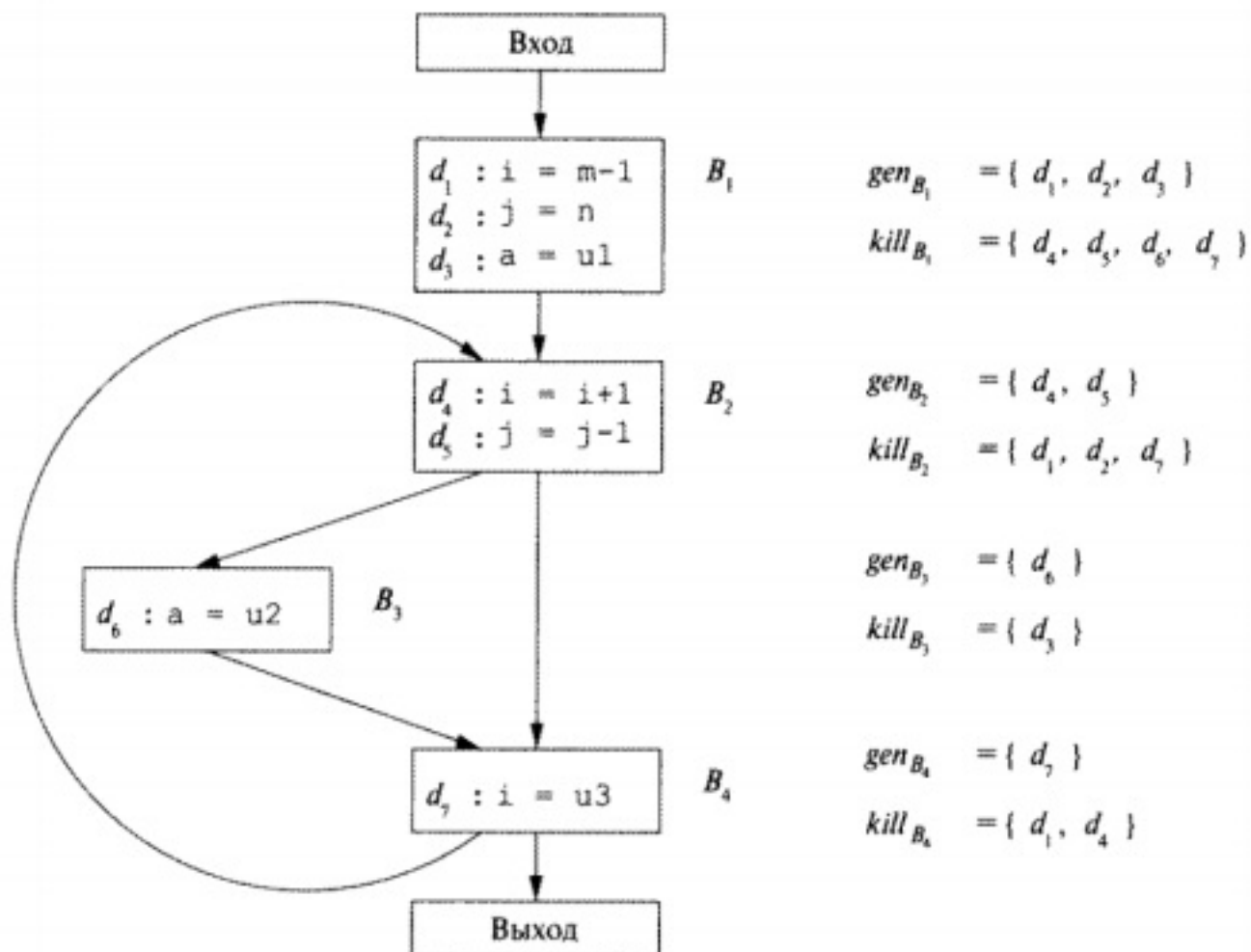
```
l1: a = 3 - 5  
l2: b = 10 + 2  
l3: c = -1  
l_: if 1 goto l3  
ass4 = l5: d = c + 1999  
l_: if 2 goto l2  
l7: e = 7 * 4  
l8: f = 100 / 25
```

```
IN[0] = { }  
OUT[0] = { l1 }  
  
IN[1] = { l1, l2, l3, l5 }  
OUT[1] = { l1, l2, l3, l4 }  
  
IN[2] = { l1, l2, l3, l5 },  
OUT[2] = { l1, l2, l4, l5 }  
  
IN[3] = { l1, l2, l3, l5 }  
OUT[3] = { l1, l2, l3, l5 }
```

IN[4] = { l1, l2, l3, l5 }

OUT[4] = { l1, l2, l3, l5, l7, l8 }

Пример работы.



Блок B	$OUT[B]^0$	$IN[B]^1$	$OUT[B]^1$	$IN[B]^2$	$OUT[B]^2$
B_1	0000000	0000000	1110000	0000000	1110000
B_2	0000000	1110000	0011100	1110111	0011110
B_3	0000000	0011100	0001110	0011110	0001110
B_4	0000000	0011110	0010111	0011110	0010111
ВЫХОД	0000000	0010111	0010111	0010111	0010111

Название задачи

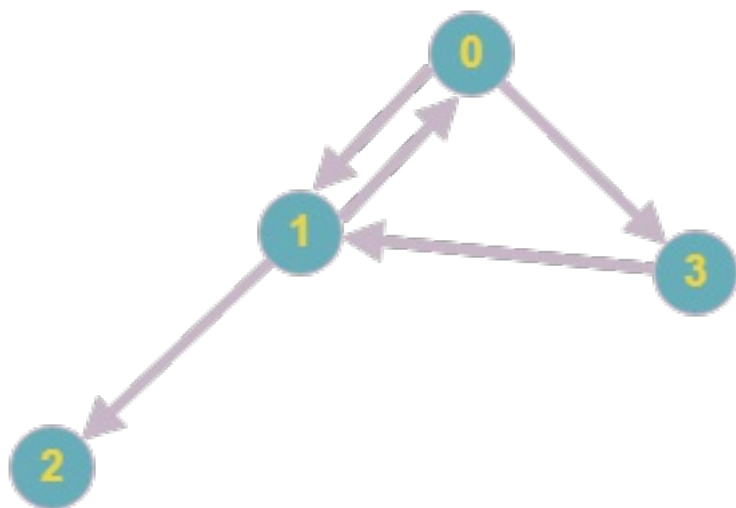
Классификация рёбер CFG.

Постановка задачи

Дан ControlFlowGraph. Все его рёбра необходимо классифицировать на три группы:

1. Наступающие (coming) рёбра идут от узла к его истинному потомку.
2. Отступающие (retreating) рёбра идут от узла к его предку.
3. Поперечные (coming) - все остальные рёбра.

Пример классификации:



Рёбра $0 \rightarrow 1$ и $1 \rightarrow 2$ являются наступающими, $1 \rightarrow 0$ - отступающее, и $3 \rightarrow 1$ поперечное.

Зависимости задач в графе задач

Задачи, от которых зависит текущая задача:

1. Построение ControlFlowGraph.
2. Построение глубинного остовного дерева.

Задачи, зависящие от текущей:

1. Проверка CFG на приводимость

Теоретическая часть задачи

Для решения данной задачи строится глубинное остовное дерево - обход “поиск в глубину” вершин ControlFlowGraph, начиная с первой вершины. Отметим, что в процессе обхода графа вершину нумеруются согласно Те рёбра, которые попали в данный граф, являются наступающими.

Ребро из вершины x в вершину y будет являться отступающим в том случае, если вершина y является предком вершины x (или номер вершины x больше номера вершины y).

Практическая часть задачи (реализация)

Вначале был создан класс-перечисление, в котором определяются классы рёбер:

```
public enum EdgeType
{
    Coming = 1,
    Retreating = 2,
    Cross = 3
}
```

```
}
```

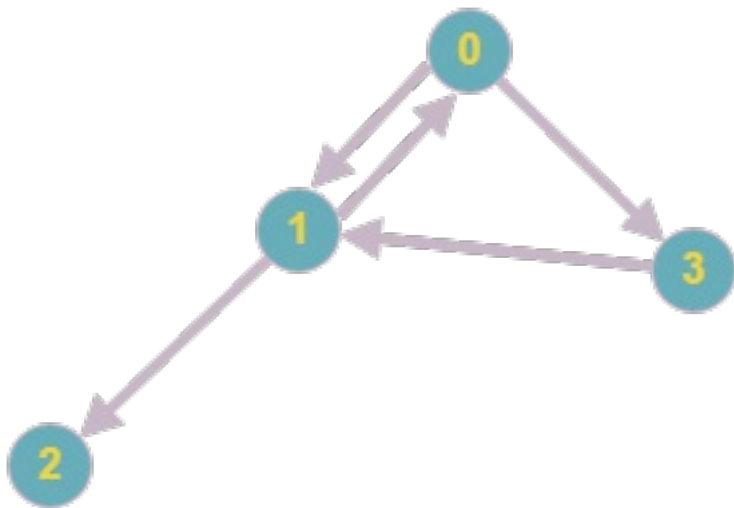
Дополнительно был создан класс `EdgeTypes`, являющийся словарём, где ключ - ребро `ControlFlowGraph`, а значение - тип ребра.

```
public class EdgeTypes : Dictionary<Edge<BasicBlock>, EdgeType>
{
    public override string ToString()
    {
        return string.Join("\n", this.Select(ed => $"[{ed.Key.Source.ToString()} -> {ed.Key.Target.ToString()}]: {ed.Value}"));
    }
}
```

Наконец, в самом классе `ControlFlowGraph` добавлено поле `EdgeTypes`, где и хранится результат классификации.

Тесты

Возьмём `ControlFlowGraph` из примера выше:



Результат работы программы:

0 → 1 : Coming

1 → 2: Coming

0 → 3: Coming

1 → 0: Retreating

3 → 1: Cross

Пример работы.

Приведём алгоритм работы данной задачи:

```
public void ClassificateEdges()  
{  
    var depthTree = new DepthSpanningTree(this);  
    foreach (var edge in CFGAuxiliary.Edges)  
    {  
        if (depthTree.SpanningTree.Edges.Any(e => e.Target.Equals(edge.Target) && e.Source.Equals(edge.Source)))  
        {
```



```

        EdgeTypes.Add(edge, EdgeType.Coming);
    }
    else if (depthTree.FindBackwardPath(edge.Source, edge.Target))
    {
        EdgeTypes.Add(edge, EdgeType.Retreating);
    }
    else
    {
        EdgeTypes.Add(edge, EdgeType.Cross);
    }
}
}

```

Метод для вызова в IDE:

```

public string PrintCFGEdgeClassification(ControlFlowGraph controlFlowGraph)
{
    controlFlowGraph.ClassificateEdges();
    return controlFlowGraph.EdgeTypes.ToString();
}

```

Название задачи

Перемещение определения как можно ближе к использованию.

Постановка задачи

Реализовать оптимизацию перемещения определений к использованию.

Зависимости задач в графе задач

Зависит от: Трехадресный код

Теоретическая часть задачи

Перемещение кода

```
x := b
. . .
y := x + z
```

Если в этой области переменная x – живая и нет других использований переменной x , то определение переменной x можно переместить как можно ближе к месту первого использования.

Если в этой области переменная x – живая и нет других использований переменной x , то определение переменной x можно переместить как можно ближе к месту первого использования.

Практическая часть задачи (реализация)

Часть кода для оптимизации операции сложения. Полный файл по [ссылке](#).

```
bool app = false;
var currentVariables = new List<Guid>();
var usedVariables = new List<Guid>();
var nodes = inputNodes;
for (int currentIndex = nodes.Count - 1; currentIndex >= 0; currentIndex--)
{
    if (nodes[currentIndex] is Assign assignCurrent)
    {
        if (assignCurrent.Left is Var currentLeft && !usedVariables.Contains(currentLeft.Id))
        {
            currentVariables.Add(currentLeft.Id);
        }
        if (assignCurrent.Right is Var currentRight && !usedVariables.Contains(currentRight.Id))
        {
            currentVariables.Add(currentRight.Id);
        }
        if (currentVariables.Count == 0) { continue; }
        for (var i = currentIndex - 1; i >= 0; i--)
        {
            if (currentVariables.Count == 0)
            {
```

```

        break;
    }
    if (nodes[i] is Assign iAssign)
    {
        var id = iAssign.Result.Id;
        // If current result id in currentVariables,
try to move it
        if (currentVariables.Contains(id) && !iAssign
.IsLabeled)
        {
            int j = i + 1;
            while (j < currentIndex)
            {
                try
                {
                    if (nodes[j + 1] is Assign nnAssign && nodes[j] is Assign nAssign)
                    {
                        if (nnAssign.Left is Var nnLeft && nnAssign.Right is Var nnRight)
                        {
                            if (nnLeft.Id == id && nnRight.Id == nAssign.Result.Id)
                            {
                                break;
                            }
                            if (nnLeft.Id == nAssign.Result.Id && nnRight.Id == id)

```

```

        {
            break;
        }
    }
}
}
catch (Exception)
{
}
if (nodes[j] is Assign jAssign)
{
    // If left or right is Variable,
stop moving
    if (jAssign.Left is Var || jAssign.Right is Var)
    {
        if (jAssign.Left is Var jLeft)
        {
            if (id == jLeft.Id)
            {
                break;
            }
        }
        else
        {
            if (id == (jAssign.Right
as Var).Id)

```

```

        {
            break;
        }
    }
}

// else move node
var tmp = nodes[i];
nodes[i] = nodes[j];
nodes[j] = tmp;
j++;
i++;
}

// remove from currentVariables and add to
used variables
currentVariables.Remove(id);
usedVariables.Add(id);
}
}
}
}

applied = app;
return nodes;

```

Тесты

(в трехадресном коде)

```
b = 1
a = b
c = b - a
n = 20
c = 20 * 3
d = 10 + n
```

Получим

```
b = 1
a = b
c = b - a
c = 20 * 3
n = 20
d = 10 + n
```

Пример работы.

(в трехадресном коде)

```
l0: b = 1
l1: a = b + 0
l2: c = 0
l3: d = a + b
l4: h = c * 1
=>
l0: b = 1
```

```
l1: a = b + 0
```

```
l3: d = a + b
```

```
l2: c = 0
```

```
l4: h = c * 1
```


Название задачи

Оптимизация общих подвыражений.

Постановка задачи

Построение орграфа и восстановлению по нему трёхадресного кода.

Зависимости задач в графе задач

- Трёхадресный код и связанные с ним задачи

Теоретическая часть задачи

Оптимизация общих подвыражений – оптимизация компилятора, которая ищет в программе вычисления, выполняемые более одного раза на рассматриваемом участке, и удаляет вторую и последующие одинаковые операции, если это возможно и эффективно. Данная оптимизация требует проведения анализа потока данных для нахождения избыточных вычислений.

Подвыражение в программе называется **общим подвыражением**, если существует другое такое же подвыражение, которое всегда вычисляется перед данным, и операнды не изменяются в промежутке между вычислениями.

Для реализации используется ориентированный ациклический граф, построенный по участку трёхадресного кода:

- в качестве вершин используется структура содержащая операции и переменные которым присваиваются значения
- трехадресный код генерируется от корня орграфа рекурсивно
- если вершина помечена несколькими переменными, то код генерируется для первой переменной, а для остальных генерируются команды копирования
- если орграф состоит из нескольких несвязных подграфов, то код по ним можно генерировать, начиная с любого подграфа

Практическая часть задачи (реализация)

Для реализации графа была реализована структура описывающая вершины: *ExpressionNode*

```
class ExpressionNode
{
    public ExpressionNode LeftNode { get; set; } // Левый
операнд
    public ExpressionNode RightNode { get; set; } // Прав
ый операнд
    public OpCode OpCode { get; set; } // Производимая оп
ерация
    public List<Expr> AssigneeList { get; } // Хранилище
операндов

    public ExpressionNode(Expr expression)
    {
        AssigneeList = new List<Expr>();
    }
}
```

```

        AssigneeList.Add(expression);
    }

    public bool IsList()
    {
        if (RightNode == null && LeftNode == null)
            return true;
        return false;
    }
}

```

А так же структура для деревьев для более эффективного поиска нужного операнда

```

class ExpressionTree
{
    public List<ExpressionNode> Nodes { get; set; } // Но
ды дерева
    public List<Expr> AllAssignees { get; set; } // Списо
к всех присвоений в данном дереве

    public ExpressionTree()
    {
        Nodes = new List<ExpressionNode>();
        AllAssignees = new List<Expr>();
    }
}

```

```

public void AddNode(ExpressionNode node)
{
    Nodes.Add(node);
    AllAssignees = AllAssignees.Concat(node.AssigneeList).Distinct().ToList();
}
}

```

Был описан класс оптимизации *SubexpressionOptimization*, реализующий интерфейс *IOptimization* с переопределенным методом *Optimize*.

Сама оптимизация представляет собой следующий алгоритм:

- выбираем ноды представляющие собой операцию присваивания

```

var nodes = inputNodes.OfType<Assign>().Where(assn => assn.Operation != OpCode.Copy && assn.Left != null);

```

- проверяем правый и левый операнд - существует ли дерево содержащее один из них

```

foreach(var node in nodes)
{
    currentTree = null;
    var leftNode = FindOrInitializeExpressionNode(node.Left, out ExpressionNode leftParentNode);
    var rightNode = FindOrInitializeExpressionNode(node.Right

```

```
, out ExpressionNode rightParentNode);
```

```
.....
```

- если дерево не нашлось, создаем новый ExpressionNode в качестве листа дерева

```
var root = SeekRootTree(expr);  
parentNode = null;  
initializedNewNode = true  
if (root == null)  
{  
    ExpressionNode newNode = new ExpressionNode(expr);  
    if (currentTree == null)  
    {  
        currentTree = new ExpressionTree();  
        currentTree.AddNode(newNode);  
        exprForest.Add(currentTree);  
    }  
    return newNode;  
}
```

- если дерево нашлось, запускаем поиск в ширину от корня, чтобы найти самое актуальное вхождение переменной в дереве

```
var lastExpression = BFS(root, expr, out parentNode);  
if (lastExpression == null)  
    return new ExpressionNode(expr);
```

- проверяем, есть ли у полученных узлов единый предок и если есть, то добавляем результат присвоения к нему в AssignList, если нет то создаем ExpressionNode с результатом присваивания

```
if (leftParentNode == null && rightParentNode == null)
{
    var resultNode = new ExpressionNode(node.Result);
    resultNode.LeftNode = leftNode;
    resultNode.RightNode = rightNode;
    resultNode.OpCode = node.Operation;
    currentTree.AddNode(resultNode);
}
else
{
    leftParentNode.AssigneeList.Add(node.Result);
    applied = true;
}
```

- после того как обходим все узлы, применяем оптимизацию - строим новые выражения восстанавливая дерево
- обходим все деревья и узлы исходного трехадресного кода и находим нужное присваивание из каждого ExpressionNode

```
var node = inputNodes.OfType<Assign>().FirstOrDefault(x => x.Node.AssigneeList.Contains((x as Assign).Result));
```

- для каждого элемента данного узла заменяем значения

последующих присваиваний на значение найденного элемента

```
foreach (var optExpr in expNode.AssigneeList)
{
    var extraNode = inputNodes.OfType<Assign>().FirstOrDefault(x => (x as Assign).Result == optExpr as Var);
    extraNode.Left = null;
    extraNode.Right = node.Result;
    extraNode.Operation = OpCode.Copy;
}
```

Тесты

```
public void SubexpressionOptimizationTest()
{
    var taCode = new TACode();

    var ass1 = new Assign
    {
        Left = new Var(),
        Operation = OpCode.Plus,
        Right = new Var(),
        Result = new Var()
    };

    var ass2 = new Assign
    {
```

```
        Left = ass1.Result,  
        Operation = OpCode.Minus,  
        Right = new Var(),  
        Result = ass1.Left as Var  
    };  
  
    var ass3 = new Assign  
    {  
        Operation = OpCode.Plus,  
        Left = ass2.Result,  
        Right = ass1.Right,  
        Result = ass1.Right as Var  
    };  
  
    var ass4 = new Assign  
    {  
        Left = ass1.Result,  
        Operation = OpCode.Minus,  
        Right = ass2.Right,  
        Result = ass2.Right as Var  
    };  
  
    var ass5 = new Assign  
    {  
        Operation = OpCode.Mul,  
        Left = new IntConst(2),  
        Right = new Var(),  
        Result = new Var()  
    };  
  
    var ass6 = new Assign
```



```

{
    Left = ass5.Result,
    Operation = OpCode.Minus,
    Right = new IntConst(1),
    Result = ass5.Result as Var
};

```

```

/*
    a = b + c
    b = a - d
    c = b + c
    d = a - d  -----> d = b
    e = 2 * n
    e = e - 1
*/

```

```

taCode.AddNode(ass1);
taCode.AddNode(ass2);
taCode.AddNode(ass3);
taCode.AddNode(ass4);
taCode.AddNode(ass5);
taCode.AddNode(ass6);

```

```

Console.WriteLine("SUBEXPRESSION TEST");
Console.WriteLine($"TA Code:\n{taCode.ToString()}

```

```

");

```

```

var subexpOpt = new SubexpressionOptimization();

```

```
        Console.WriteLine("Optimised TA Code:");  
        foreach (var node in subexpOpt.Optimize(taCode.CodeList.ToList(), out var applied))  
            Console.WriteLine($"{node}");  
        Console.ReadKey();  
    }  
}
```

Название задачи

Общий алгоритм при наличии вектора оптимизаций (O1, O2, ...)

Постановка задачи

Необходимо реализовать алгоритм, который применял бы все возможные оптимизации для базового блока.

Зависимости задач в графе задач

Зависит от:

- Все реализованные оптимизации для базового блока

Теоретическая часть задачи

Требуется реализовать алгоритм, который применил бы к базовому блоку все имеющиеся оптимизации до тех пор, пока это возможно делать.

Практическая часть задачи (реализация)

Был реализован класс `AllOptimizations` и функция `ApplyAllOptimizations`, применяющая все оптимизации к трехадресному коду.

```
public class AllOptimizations
{
    private List<IOptimization> BasicBlockOptimizationLis
```

```

t()
{
    List<IOptimization> optimizations = new List<IOptimization>();

    optimizations.Add(new CopyPropagation());
    optimizations.Add(new ConstantFolding());
    optimizations.Add(new ConstantPropagation());
    optimizations.Add(new DeclarationOptimization());
    optimizations.Add(new AlgebraicOptimization());
    optimizations.Add(new SubexpressionOptimization());
};

return optimizations;
}

private List<IOptimization> o2OptimizationList()
{
    return new List<IOptimization>();
}

public TACode ApplyAllOptimizations(TACode code)
{
    List<IOptimization> o1Optimizations = BasicBlockOptimizationList();

    var canApplyAny = true;

```

```
while (canApplyAny)
{
    canApplyAny = false;
    var blocks = code.CreateBasicBlockList().ToList();

    var codeList = new List<Node>();

    foreach (var b in blocks)
    {
        var block = b.CodeList.ToList();
        for (int i = 0; i < o10optimizations.Count; i++)
        {
            block = o10optimizations[i].Optimize(block, out var applied);
            canApplyAny = canApplyAny || applied;
        }
        codeList.AddRange(block);
    }

    code = new TACode();
    code.CodeList = codeList;

    foreach (var line in code.CodeList)
        code.LabeledCode[line.Label] = line;
}
```

```
        return code;
    }
}
```

Тесты

```
a = b
c = b - a    -----> c = 0
n = 20
c = 20 * 3   -----> c = 60
d = 10 + n   -----> d = 30
```

Пример работы.

```
public void Test1()
{
    var taCodeAllOptimizations = new TACode();
    var assign1 = new Assign()
    {
        Left = null,
        Operation = OpCode.Copy,
        Right = new Var(),
        Result = new Var()
    };
    var assign2 = new Assign()
    {
        Left = assign1.Right,
```

```
        Operation = OpCode.Minus,
        Right = assgn1.Result,
        Result = new Var()
    };
    var assgn3 = new Assign()
    {
        Left = null,
        Operation = OpCode.Copy,
        Right = new IntConst(20),
        Result = new Var()
    };
    var assgn4 = new Assign()
    {
        Left = new IntConst(20),
        Operation = OpCode.Mul,
        Right = new IntConst(3),
        Result = new Var()
    };
    var assgn5 = new Assign()
    {
        Left = new IntConst(10),
        Operation = OpCode.Plus,
        Right = assgn3.Result,
        Result = new Var()
    };
    taCodeAllOptimizations.AddNode(assgn1);
    taCodeAllOptimizations.AddNode(assgn2);
    taCodeAllOptimizations.AddNode(assgn3);
```

```
taCodeAllOptimizations.AddNode(assign4);
```

```
taCodeAllOptimizations.AddNode(assign5);
```

```
var allOptimizations = new AllOptimizations();
```

```
allOptimizations.ApplyAllOptimizations(taCodeAllOptimizations);
```

```
Assert.AreEqual(assign2.Right, 0);
```

```
Assert.AreEqual(assign4.Right, 60);
```

```
Assert.AreEqual(assign5.Right, 30);
```

```
Assert.True(true);
```

```
}
```


Название задачи

Реализация Def и Use множеств

Зависит от:

- Def/Use Lists

Постановка задачи

Необходимо определить Def и Use множества

для базового блока

Теоретическая часть задачи

Определение. Переменная x активна в точке p если значение x из точки p может использоваться вдоль некоторого пути, начинающегося в p .

Определение.

def_B — множество переменных, определённых в B до любого их использования

use_B — множество переменных, значения которых могут использоваться в B до любого их определения

Пример.

$$\begin{aligned} i &= k + 1 \\ j &= l + 1 \\ k &= i \\ l &= j \end{aligned}$$

$$\begin{aligned} def_B &= \{i, j\} \\ use_B &= \{k, l\} \end{aligned}$$

Любая переменная из use_B — активная на входе в B

Любая переменная из def_B — мёртвая на входе в B

Спорная ситуация

Пример.

$i = i + 1$
 $j = j + 1$

 $def_B = \{ \}$
 $use_B = \{i, j\}$

Но можно использовать другое определение def_B :

def_B — множество переменных, определённых в В ~~до любого их использования~~. Тогда

$i = i + 1$
 $j = j + 1$

 $def_B = \{i, j\}$
 $use_B = \{i, j\}$

Как станет понятно в дальнейшем, это неважно

В данной реализации был выбран второй вариант определения Def множества.

Входные данные

Базовый блок

Выходные данные

Def и Use множества

Реализация

Ниже приведена часть алгоритма:

```
/// <summary>
```

```
/// Создает Def и Use множества для базового блока
/// </summary>
private void BuildDUSets()
{
    var duLists = new DULists(Block);

    foreach (var d in duLists.DList)
        DSet.Add(d.DefVariable.Name);

    foreach (var u in duLists.UListNotValid)
        USet.Add(u.Name);
}
```

Тесты

Приведем примененеие этого алгоритма для нескольких базовых блоков

Исходный код:

```
// 0: i = k + 1
// 1: j = l + 1
// 2: k = i
// 3: l = j
```

Результат:

```
Def = { i, j, k, l }
Use = { k, l }
```

Название задачи

Реализация итерационного алгоритма для Активных переменных

Зависит от:

- Def/Use Lists
- LiveAndDead Variables
- Delete Dead Code In Single Block
- Def/Use Sets

Постановка задачи

Необходимо определить активные переменные для каждого базового блока

в Control Flow Graph

Теоретическая часть задачи

$$IN[\text{Выход}] = \emptyset$$

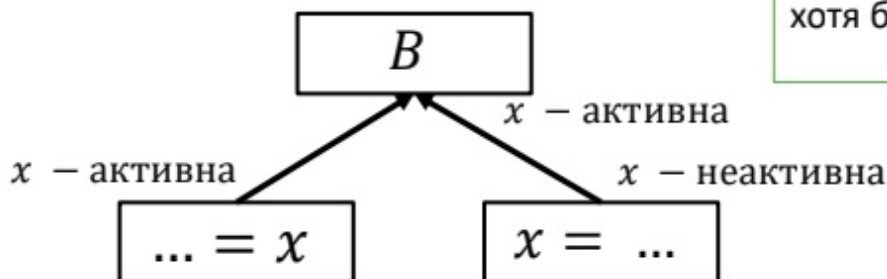
$$IN[B] = use_B \cup (IN[B] - def_B)$$

$$\begin{matrix} i = i + 1 \\ j = j + 1 \end{matrix}$$

$$\begin{matrix} def_B = \{ \} \\ use_B = \{i, j\} \\ \text{или} \\ def_B = \{i, j\} \\ use_B = \{i, j\} \end{matrix}$$

$$OUT[B] = \bigcup_{S-\text{преемник } B} IN[S]$$

Переменная активна на выходе из B если она активна на входе хотя бы в одном из преемников



Вход: граф потока управления, в котором для каждого ББл вычислены def_B и use_B

Выход: Множества переменных, активных на входе $IN[B]$ и на выходе $OUT[B]$ для всех ББл B

$IN[\text{Выход}] = \emptyset;$

for (каждый базовый блок B, отличный от выходного) $IN[B] = \emptyset;$

while (Внесены изменения в IN)

for (каждый базовый блок B, отличный от выходного) {

$OUT[B] = \bigcup_{S-\text{преемник } B} IN[S];$

$IN[B] = use_B \cup (OUT[B] - def_B);$

 }

Сходимость алгоритма: на каждом шаге $IN[B]$ и $OUT[B]$ не уменьшаются для всех B и ограничены сверху, поэтому алгоритм сходится

Входные данные

Control Flow Graph исходной программы

Выходные данные

IN и OUT множества

Реализация

Ниже приведена часть алгоритма:

```
/// <summary>
    /// Базовый итеративный алгоритм
    /// </summary>
    private void Algorithm()
    {
        Dictionary<Guid, ActiveVar> oldSetIN;

        do
        {
            oldSetIN = CopyIN();

            foreach (var B in CFG.CFGNodes)
            {
                var idB = B.BlockId;

                // Первое уравнение
                foreach (var child in B.Children)
                {
                    var idCh = child.BlockId;
                    OUT[idB].UnionWith(IN[idCh]);
                }
            }
        } while (oldSetIN != IN);
    }
```

```

    }

    var subUnion = new ActiveVar(OUT[idB]);
    subUnion.ExceptWith(DefSet[idB]);

    // Второе уравнение
    IN[idB].UnionWith(UseSet[idB]);
    IN[idB].UnionWith(subUnion);
}
}
while (!EqualIN(oldSetIN, IN));
}

```

Тесты

Исходный код:

```

// 0:      a = 2
// 1:      b = 3
// 2: (1) : c = a + b
// 3: (2) : a = 3
// 4:      b = 4
// 5: (3) : c = a
// 6:      print(c)

```

Разбиение на блоки:

B0:

```

// 0:      a = 2
// 1:      b = 3

```

B1:

```
// 0: (1) : c = a + b
```

B2:

```
// 0: (2) : a = 3
```

```
// 1:      b = 4
```

B3:

```
// 0:      print(c)
```

Результат:

IN(B0) = { }

OUT(B0) = { a, b }

IN(B1) = { a, b }

OUT(B1) = { }

IN(B2) = { }

OUT(B2) = { a }

IN(B3) = { a }

OUT(B3) = { }

Название задачи

Удаление мертвого кода между базовыми блоками

Зависит от:

- Def/Use Lists
- LiveAndDead Variables
- Delete Dead Code In Single Block
- Def/Use Sets
- Iteractive Algorithm For Active Variables

Теоретическая часть задачи

Мы применяем итерационный алгоритм для CFG исходной программы и получаем OUT-переменные.

Далее для каждого блока в этом графе мы удаляем строки мертвого кода следующим образом:

подается блок, для него мы вычисляем живые/мертвые переменные.

Мы делаем следующее, если в списке мертвых переменных встретилась переменная, которая

является out (т.е. нужна другому блоку в программе), то мы находим последнее присваивание

этой переменной и перемещаем его в список живых переменных.

Также, мы рассматриваем OUT переменные предков данного блока.

Если в родительских блоках

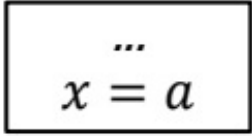
определены переменные, которые используются в текущем блоке (но не определяются), то такие

переменные становятся живыми.


После этого мы строим граф для нашего нового кода и повторяем все шаги, пока на каком-то этапе мы не удалим ни одной строки кода.

Основная оптимизация – удаление мёртвого кода

Уточнения:

- 

x – мёртвая

Если мы знаем, что в $OUT[B]$ x – мёртвая, то последнее присваивание без использования – мёртвый код
- 

x – живая

Если мы знаем, что в $IN[B]$ x – живая, то для неё можно отвести регистр

Входные данные

Код программы в трехадресном виде

Входные данные

Код программы в трехадресном виде без мертвых строк кода

Реализация

Ниже приведена часть алгоритма:

```
/// <summary>
```

```
/// Удаление мертвого кода в CFG
```

```
/// </summary>
```

```
/// <returns></returns>
private TACode RemoveDeadCodeInCFG()
{
    var code = new TACode();
    code.CodeList = CodeIN.CodeList.ToList();
    ControlFlowGraph cfg;
    int countRemove;

    do
    {
        // Вычисляем CFG
        cfg = new ControlFlowGraph(code);
        // Вычисляем IN и OUT переменные для всех бло
ков в CFG

        this.OUT = (new IterativeAlgorithmAV(cfg)).OUT;

        countRemove = 0;

        // Для каждого блока в cfg
        foreach (var B in cfg.CFGNodes)
        {
            // Удаляем мертвые строки кода
            var newB = RemoveDeadCodeInBlock(B);
            var curCountRem = B.CodeList.Count() - newB.CodeList.Count();

            if (curCountRem != 0)
            {
```

```

        var idxStart = CalculateIdxStart(B, code.CodeList);

        var len = B.CodeList.Count();
        code = ReplaceCode(code, newB.CodeList.ToList(), idxStart, len);
        countRemove += curCountRem;
    }
}
}
while (countRemove != 0);

return code;
}

```

Тесты

Исходный код:

```

// 0:      a = 2
// 1:      b = 3
// 2: (1) : c = a + b
// 3: (2) : a = 3
// 4:      b = 4
// 5: (3) : c = a
// 6:      print(c)

```

Результат:

```

// 0: (2) : a = 3
// 1: (3) : c = a

```

```
// 2:      print(c)
```

Название задачи

Построение дерева доминаторов.

Постановка задачи

Построить дерево доминаторов по Control Flow Graph (CFG)

Зависимости задач в графе задач

Зависит от: Классификации ребер в CFG

Теоретическая часть задачи

Дерево доминаторов — вспомогательная структура данных, содержащая информацию об отношениях доминирования. Дуга от узла M к узлу N идет тогда и только тогда, если M является непосредственным доминатором N .

Алгоритм построения дерева доминаторов можно представить следующим образом:

1. Строим матрицу доминаторов (DM). Изначально это матрица, в которой во всех строках стоят 1, кроме первой строки. В ней 1 только в первом столбце.
2. Определяем 2 операции над строками матрицы доминаторов:
 - Объединение:

$$i \cup j = \begin{cases} i, & i = j \\ 1, & \text{else} \end{cases}, \text{ где } i \text{ и } j - \text{ строки матрицы доминаторов}$$

- Пересечение

$$i \cap j = i * j, \text{ где } i \text{ и } j - \text{ строки матрицы доминаторов}$$

3. Для каждого узла x , кроме $root$, определяем множество доминаторов следующим образом:

$$dom(x) = \{x\} \cup \{dom(i_1) \cap dom(i_2) \cap \dots \cap dom(i_k)\}$$

где i_1, i_2, \dots, i_k - предки узла x .

4. Повторяем шаг 3, пока матрица доминаторов не перестанет изменяться после прохода всех узлов.

Практическая часть задачи (реализация)

Часть кода для оптимизации операции сложения. Полный файл по [ссылке](#).

```
// Инициализируем переменные
int N = CFG.CFGNodes.Count;
bool changed = true;

// Заполняем матрицу смежности для дерева доминаторов единиц
```

ами

```
foreach (var bb in CFG.CFGNodes)
{
    var item = new DomRow
    {
        BasicBlock = bb
    };
    foreach (var bb1 in CFG.CFGNodes)
    {
        item.ItemDoms.Add(new DomCell
        {
            BasicBlock = bb1,
            HasLine = true
        });
    }
    _matrix.Add(item);
}

// По правилу все ячейки должны быть 1, кроме 1 строки в промежутке от 2ой до последней ячейки
// Пример
/* [
 * 1 0 0 0,
 * 1 1 1 1,
 * 1 1 1 1,
 * 1 1 1 1,
 * ]
 */
for (int i = 1; i < N; i++)
```



```

{
    _matrix[0].ItemDoms[i].HasLine = false;
}
// Считаем матрицу смежности для дерева доминаторов
while (changed)
{
    changed = false;
    for (int i = 1; i < N; i++)
    {
        // Заполняем значение {x} в формуле. {x} является узле
        л, который доминирует только сам над собой
        List<DomCell> blockRow = new List<DomCell>();
        foreach (var bb in CFG.CFGNodes)
        {
            // Если id ББ-ка совпадает с текущим, ставим 1 в
            противном случае 0.
            blockRow.Add(new DomCell
            {
                BasicBlock = bb,
                HasLine = bb.BlockId == CFG.CFGNodes[i].Block
                Id
            });
        }
        // Список = { dom(i_1) && dom(i_2) && ... && dom(i_k)
    }

    List<bool> results = new List<bool>();
    for (var j = 0; j < N; j++)
    {

```

```

        results.Add(true);
    }

    foreach (var parent in CFG.CFGNodes[i].Parents)
    {
        // Находим dom каждого предка
        var domParent = _matrix.Single(row => row.BasicBlock.BlockId == parent.BlockId);
        for (int j = 0; j < N; j++)
        {
            // Поэлементно находим конъюнкцию всех предков
            results[j] &= domParent.ItemDoms[j].HasLine;
        }
    }
    // Сохраняем строку до изменения
    var oldRow = new List<bool>();
    foreach(var item in _matrix[i].ItemDoms)
    {
        oldRow.Add(item.HasLine);
    }
    // Получили dom(i) = {i} || results. Следовательно мы
    // полняем поэлементную дизъюнкцию
    for (var j = 0; j < N; j++)
    {
        // Выбираем строку с текущим блоком, и меняем все
        // его значения
        _matrix[i].ItemDoms[j].HasLine = blockRow[j].HasL

```

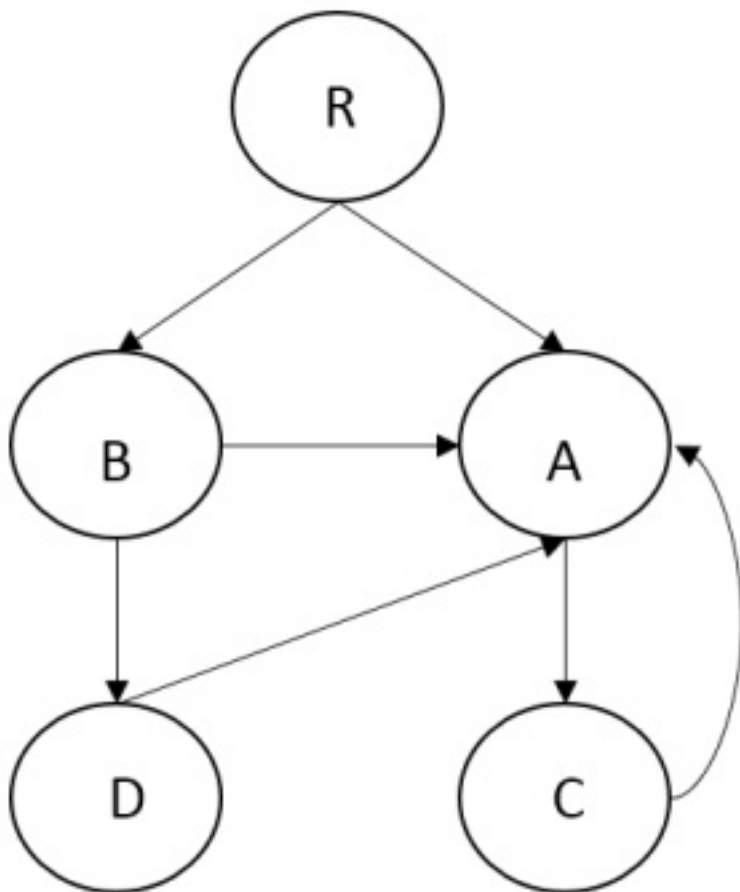
```

ine || results[j];
    }
    // Проверяем изменилась ли строка
    for (var j = 0; j < N; j++)
    {
        changed |= oldRow[j] != _matrix[i].ItemDoms[j].HasLine;
    }
}
}
}

```

Пример работы.

Имеем такой граф:



Строим матрицу доминаторов.

	R	A	B	C	D
R	1	0	0	0	0
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1

Применяем алгоритм построения

•
$$dom(A) = \{A\} \cup \{dom(R) \cap dom(B) \cap dom$$

Подставив строки из матрицы доминирования в эту формулу, получаем:

$$dom(A) = 01000 \cup \{10000 \cap 11111 \cap 11111 \cap$$

что эквивалентно:

$$dom(A) = 01000 \cup \{10000\} = \mathbf{11000}$$

• $dom(B) = 00100 \cup \{10000\} = \mathbf{10100}$

• $dom(C) = 00010 \cup \{11000\} = \mathbf{11010}$

• $dom(D) = 00001 \cup \{10100\} = \mathbf{10101}$

После этих процедур наша матрица будет иметь вид:

DM	R	A	B	C	D

R	1	0	0	0	0
A	1	1	0	0	0
B	1	0	1	0	0
C	1	1	0	1	0
D	1	0	1	0	1

j доминирует над i

