**Конструляторы -- QBE IL -- all.h**

**Околодокументация от работяг.**

**Мытищи, 20!8**

[**QBE IL**](#_ai1ayeal05mn) **3**

[О чём?](#_y51kpoilmqp1) 3

[Чё с ним делать?](#_9mafunob1v9q) 3

[**All.h**](#_9jfq5iwv12g1) **4**

[Введение](#_ow3tpjlfn587) 4

[Околодокументация](#_y67a6y3da8w) 5

[Внутренности all.h](#_2ktwucnohpbj) 6

[void parse(FILE \*f, char \*path,  
 void data(Dat \*), void func(Fn \*))](#_qwu6fq4vxpe5) 6

[void printfn(Fn \*fn, FILE \*f)](#_hj4glanomspm) 6

[struct Fn](#_8411jy34ir8u) 6

[struct Tmp](#_226zq711pn90) 7

[unsigned long long Tmp0](#_e025j3x4u6xk) 7

[struct Blk](#_bt1qpyjbcrbl) 7

[struct Ins](#_ntw6emtnm0oo) 8

[extern Op optab[NOp]](#_bzcrermt55m0) 8

[struct Ref](#_5ja2apwg37eg) 8

[struct Op](#_vgjl678bp48o) 9

[static inline int req (Ref a, Ref b) {](#_mks2bog5di91) 9

[struct Use](#_v4r791b7kbrs) 9

[**Алгоритмы**](#_5mcalymbmdyw) **10**

[gen, kill](#_glnjjxii3542) 10

[def, use](#_ihzk3lw07cwh) 11

[reaching definitions](#_dmc9mk2vl1qg) 12

[live variables](#_nnawk5pemf9p) 12

[dominators](#_51hl6ovz25a6) 13

[dominance frontier](#_8746egelenf9) 13

[dead code elimination](#_lixjs863wl5e) 15

[Теория](#_1vinbgl4u7fy) 15

[Разбор функции Mark из книжки.](#_9yq38vf226ha) 15

[Функция Sweep](#_gd4a4gxfqb1n) 16

[Практика](#_4699f886hbip) 16

[Функции к задаче 3dce](#_o85100uvkrwk) 17

# 

# QBE IL

## О чём?

QBE IL (Questionable Backend Intermediate Language) — язык промежуточного представления, нужен, по всей видимости, для того, чтобы окончательно добить студентов на 4 курсе, либо обучить всяким невероятно нужным вещам, вроде внутреннего представления языка компилятором.

## Чё с ним делать?

<https://c9x.me/compile/doc/il.html> — читать текст, можно даже загнать в гугл транслейт, если слишком лень.

*если кто-то хочет, может запилить сюда какой-нить текст про него*

# All.h

## Введение

Но самое интересное, это header all.h, любезно предоставленный ИСП РАН для учебных целей. Если в него заглянуть, можно увидеть что-то неприличное, а всё, что о нём написано это следующие строки:

|  |
| --- |
| Неофициальный C-интерфейс QBE представлен файлом all.h в корневой директории проекта.  Для реализации анализов и трансформаций на уровне промежуточного представления вам понадобятся функции parse и printfn для чтения и записи QBE IL в текстовом формате.  Программа на QBE IL является последовательностью определений функций, данных и типов. Определения функций представлены структурой Fn, содержащий указатель на entry-блок Fn::start.  Базовые блоки представлены структурой Blk, содержащей указатель на следующий блок в некотором порядке обхода Blk::link, указатели на последующие блоки Blk::s1 и Blk::s2, указатели на предшествующие блоки Blk::pred, а также на начало массива инструкций Blk::ins.  Инструкции представлены структурой Ins, содержащий код операции Ins::op (который может использоваться в качестве ключа в массиве optab), и ссылки Ins::to и Ins::arg на результат операции и её операнды.  Используйте функцию freeall для освобождения памяти. |

Достаточно исчерпывающе, не правда ли? Поэтому далее попытаемся запилить кое-какую инфу про эти структуры.

## Околодокументация

Предоставленный в условии контестов пример использования:

|  |
| --- |
| #include <qbe/all.h>  #include <stdio.h>  static void readfn (Fn \*fn) {  for (Blk \*blk = fn->start; blk; blk = blk->link) {  printf("@%s", blk->name);  printf("\n\tgen = ");  if (blk->nins && Tmp0 <= blk->ins->to.val) {  printf("@%s%%%s", blk->name, fn->tmp[blk->ins->to.val].name);  }  printf("\n\tkill =\n\n");  }  }  static void readdat (Dat \*dat) {  (void) dat;  }  int main () {  parse(stdin, "<stdin>", readdat, readfn);  freeall();  } |

Для использования хедера в программах на C++ (пиздоглазым посвящается):

|  |
| --- |
| #define export exports extern "C" {  #include "qbe/all.h" } #undef export |

Компиляция: g++ -lm объектник.o qbe/obj/libqbe.a -o программа

Можно: g++ исходник.c qbe/obj/libqbe.a -o программа

CMakeLists для использования CLion:

|  |
| --- |
| cmake\_minimum\_required(VERSION 3.9)  project(task3)  set(CMAKE\_CXX\_STANDARD 11)  include\_directories(/*<DESTDIR при вызове make install>*/include)  add\_executable(task3 main.cpp)  target\_link\_libraries(task3 -L/*<DESTDIR при вызове make install>*/lib -lqbe) |

## Внутренности all.h

|  |  |
| --- | --- |
| void parse(FILE \*f, char \*path,  void data(Dat \*), void func(Fn \*)) Мистическая функция, которая занимается тем, что парсит QBE файл во внутреннее представление, с которым происходит взаимодействие внутри некоторой функции (внутри крутится вечный цикл со свитчами, которые еще что-то парсят).  В примере считывание происходит, очевидно с stdin, работать с внутренним представлением функции предлагают в func, data не используется. | |
| f | хендл файла, откуда происходит чтение программы |
| path | путь к этому файлу (используется только для сообщений об ошибках, так что неважен) |
| data | функция, вызываемая для каждого распаршенного куска данных |
| func | функция, вызываемая для каждой распаршенной функции |

|  |
| --- |
| void printfn(Fn \*fn, FILE \*f) [Функция](https://github.com/8l/qbe/blob/d2075d5e131e181820d8c39ad74322f970d562aa/parse.c#L1150) для печати всей структуры Fn\* в файл FILE\* в виде QBE представления. Сильно помогает разобраться, как именно соотносятся все структуры между собой. |

|  |  |
| --- | --- |
| struct Fn Структура, которая является интерфейсом к внутреннему представлению QBE. | |
| Полезные поля | |
| Blk \*start | Указатель на первый в цепочке (т.е. вычисленный) базовый блок |
| uint nblk | число блоков в цепочке |
| Tmp \*tmp | массив структур, хранящих информацию о переменных. |
| int ntmp | размер массива tmp |

|  |  |
| --- | --- |
| struct Tmp Хранит информацию о переменных. Tmp - от temporary values, то есть те самые “переменные”, с которыми мы и работаем (у нас же intermediate representation). | |
| Полезные поля | |
| char \*name | название того, чего хранится в структуре |
| Use \*use | ”использования” переменной, массив |
| uint nuse | их число |

|  |
| --- |
| unsigned long long Tmp0 Число регистров, а по совместимости - индекс первой переменной, не являющейся регистром, в массивах fn->tmp. Например, в самом qbe итерация по переменным выглядит как: for (t=Tmp0; t<fn->ntmp; t++).  Также: if ( … && blk->ins->to.type == RTmp && Tmp0 <= blk->ins->to.val) — такой штукой мы понимаем, что там лежит, во-первых, переменная, а во-вторых — именованная переменная, а не регистровая, например |

|  |  |
| --- | --- |
| struct Blk Непосредственно информация о базовом блоке. | |
| Полезные поля | |
| Ins \*ins | массив инструкций, входящих в этот блок |
| uint nins | количество инструкций в блоке (можно писать for) |
| Blk \*s1, Blk \*s2 | указатели на последующие блоки (по ходу программы). Если s1 == null и s2 == null, то это значит, что блок около exit находится |
| Blk \*link | указатель на следующий блок (вычисленный, вовсе не обязательно следующий по ходу программы!)  **ВНИМАНИЕ**: если s1 == null && s2 == null, то совсем необязательно, что link == null. Есть тест, что в базовом блоке стоит ret 0, но при этом link != null.  Итерацию по блокам делать через link, выяснять, какие блоки конечные - через s1 и s2 |
| Blk \*\*pred | массив указателей на предшествующие (по ходу программы) блоки |
| uint npred | их количество |
| Blk \*idom | указатель на непосредственный доминатор |

|  |  |
| --- | --- |
| struct Ins Структура, хранящая информацию об инструкциях. | |
| Полезные поля | |
| Ref to | информация о результате переменной, куда запишется результат операции |
| Ref arg[2] | информация об операндах |
| uint op:30 | код операции для массива optab (см. ниже) |

|  |
| --- |
| extern Op optab[NOp] Массив, содержащий таблицу всех операций языка QBE IL, обращаясь к i-ому элементу этого массива, мы обратимся к i-ой операции этого языка. (Очевидно, число i обычно заранее задается парсером) |

|  |  |
| --- | --- |
| struct Ref Структура, хранящая описание переменных (не очень подробное, более подробное почему-то в Tmp) | |
| Полезные поля | |
| uint type | тип переменной, один из: RTmp (переменная == Tmp), RCon (константа), RType, RSlot, RCall, RMem.  *возможно полезное поле* |
| uint val | индекс переменной в массиве tmp структуры Fn, если тип — RTmp.  Поэтому, для получения информации об этой переменной требуется писать весьма интересную конструкцию вида:  fn->tmp[blk->ins->to.val].name  здесь будет получено имя переменной, в которую записывается результат операции, стоящей первой в списке инструкций ins блока blk. |

|  |  |
| --- | --- |
| struct Op Структура, хранящая информацию об операции | |
| Полезные поля | |
| char \*name | название операции |
| short argcls[2][4] | “классы” аргументов, их ширина то бишь. Один из Kx (равный -1), Kw, Kl, Ks или Kd  *возможно полезное поле* |
| int canfold | можно ли делать fold над такой операцией |

|  |  |
| --- | --- |
| static inline int req (Ref a, Ref b) { return a.type == b.type && a.val == b.val;  } | потенциально полезная функция |

|  |  |
| --- | --- |
| struct Use Информация об одном “использовании” переменной. | |
| Полезные поля | |
| type | кто использует переменную: инструкция (UIns), переход (UJmp) или φ-функция (UPhi) |
| bid | id блока, использовавшего переменную |
| u | union  u.ins - ссылка на инструкцию  u.phi - ссылка на φ-функцию |

# Алгоритмы

В презентациях-2015 очень много опечаток.

[Драгонбука на русском, по которой можно грепать](https://drive.google.com/open?id=0B0X-oQW4pjUUMm9ma0dzMXRqSzg); понимать, что хотели сказать слайды, хорошо именно по ней зачастую

[Engineering a Compiler, 2nd ed](https://compilers.ispras.ru/materials/Engineering_a_Compiler_2nd_edition_by_Cooper_and_Torczon.pdf); можно грепать; на нее ссылается условие 3dce и вообще неплохая книга, если что-то непонятно из драгонбука

[Все тесты на задачи 2\* в виде гуглдока (226 страниц)](https://docs.google.com/document/d/1AaOLXUFfItG2FJZ3Uolh7R5jJkMIPTX5XnL3ssEkPDg/edit?usp=sharing)

[Все тесты на sieve, все 2\*, 3dom и 3df в виде github gist](https://gist.github.com/anonymous/467670ec40737623339544f268134381)

## gen, kill

Представим переменную с именем “name” в блоке с названием “block” в качестве пары (“name”, “block\_name”). Тогда:

* множество **gen** для блока -- это просто множество всех переменных, которые объявляются в этом блоке;
* множество **kill** для блока “block” -- смотрим на gen блока “block”, и для каждой переменной “name” в этом множестве проверяем: если во множестве gen любого другого блока “block2”, **не обязательно** предшествующего блоку “block”, встречается переменная с таким же именем “name”, то пара (“name”, “block2”) войдет во множество kill блока “block”.

[Тесты](https://gist.githubusercontent.com/anonymous/467670ec40737623339544f268134381/raw/2gk)

## def, use

[Алгоритм здесь](https://github.com/wisestump/OptimizingCompiler/blob/4105258f3865a6ebed70a1445fbf1f1df264a61c/documentation/PiedPiper/PiedPiper-DefUseCalculation.md).

**Но есть нюанс**: в примерах типа

|  |
| --- |
| export function w $f(w %n, w %m) {  @nn  %a =w add 1, 1  @ret  ret %a  } |

переменная %a должна входить в use(ret). Для проверки этого был добавлен 16 тест. Однако ret не считается инструкцией и в blk->ins не входит.

Достаточно модифицировать [оригинальный алгоритм](https://github.com/wisestump/OptimizingCompiler/blob/4105258f3865a6ebed70a1445fbf1f1df264a61c/documentation/PiedPiper/PiedPiper-DefUseCalculation.md) так: найти возвращаемую переменную, и если её имя не пусто (а при ret 0 оно будет пусто, например) и если её нет в def последнего блока, то добавить её в его use.

Примерно так:

|  |
| --- |
| // Зачем делаем проверку, что это последний блок?  if ((blk->s1 == NULL) && (blk->s2 == NULL)) {  std::string ret\_var\_name = fn->tmp[blk->jmp.arg.val].name;  if (ret\_var\_name.length() != 0) {  ...  }  } |

Казалось бы, как отслеживать использования переменных в таких случаях:

%t =w call $printf(l $fmt, w %x)

ведь аргументом функции call является printf. На самом деле после парсинга эта строчка сама раскладывается на последовательность инструкций:

arg(...)

arg(x)

t = call(printf)

То есть все аргументы вызываемой функции используются в предшествующих инструкции call инструкциях arg и заботиться ни о чем не надо.

[Тесты](https://gist.githubusercontent.com/anonymous/467670ec40737623339544f268134381/raw/2du); [тест 16](https://gist.githubusercontent.com/anonymous/08d151d8d4e59e760a2ba8f535231406/raw/2du%2520%252316).

## reaching definitions

Нужно уметь строить **gen** и **kill** для блока, задача 2gk.

Нормально оформленный алгоритм из лекции:

|  |
| --- |
| for (каждый блок B)  Out[B] = ∅  In[B] = ∅  changed = true  while (changed) {  changed = false  for (каждый блок B, кроме Entry) {  NewOut[B] = Gen[B] ∪ (In[B] \ Kill[B])  In[B] = ∪ Out[P] по всем блокам P, предшествующим B  if (NewOut[B] != Out[B]) {  Out[B] = NewOut[B]  changed = true  }  }  } |

В алгоритме “каждый блок, кроме Entry” — это каждый, кроме фиктивного Entry, то есть все упомянутые в коде.

В итоге rd блока - это построенный **In** блока.

[Тесты](https://gist.githubusercontent.com/anonymous/467670ec40737623339544f268134381/raw/2rd)

## live variables

rd наоборот. Алгоритм из драгонбука:

|  |
| --- |
| for (каждый базовый блок B) {  In[B] = ∅  }  while (внесены изменения в In) {  for (каждый базовый блок, отличный от выходного) {  Out[B] = ⋃In[S], S - преемник В  In[B] = use[B] ∪ (Out[B] \ def[B])  }  } |

В итоге lv блока - это его **Out**.

После появления 16 теста требуется изменение в алгоритм вычисления use: см. [примечание к du.](#_ihzk3lw07cwh) Для проверки этого был добавлен 16 тест.

[Тесты (в том числе 16 тест)](https://gist.githubusercontent.com/anonymous/467670ec40737623339544f268134381/raw/2lv)

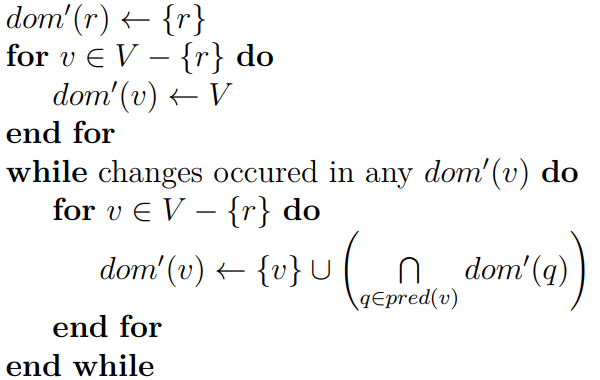
## dominators

Определение:

|  |
| --- |
| Узел d графа потока доминирует над узлом n, если любой путь от входного узла графа потока к n проходит через d. Заметим, что при таком определении каждый узел доминирует над самим собой. |

Если очень не хочется делать, как просят, по определению, и для каждой пары вершин проверять любой путь, то вот такой алгоритм зашел; вроде, это тот же, что дается в слайдах и драгонбуке, но для тех, кому лень собирать из кусочков. ~~Надеюсь, он достаточно “наивный”.~~ Ок, он ни черта не наивный, но за написание слишком крутого мудреного алгоритма тут, вроде, никого не ругали.

V - множество всех вершин, r - entry-вершина, V-{r} - все вершины, кроме entry.



Следует заметить, что **множество надо вывернуть**: в этом алгоритме в dom получается не “над кем доминирует”, а “кто доминирует над”.

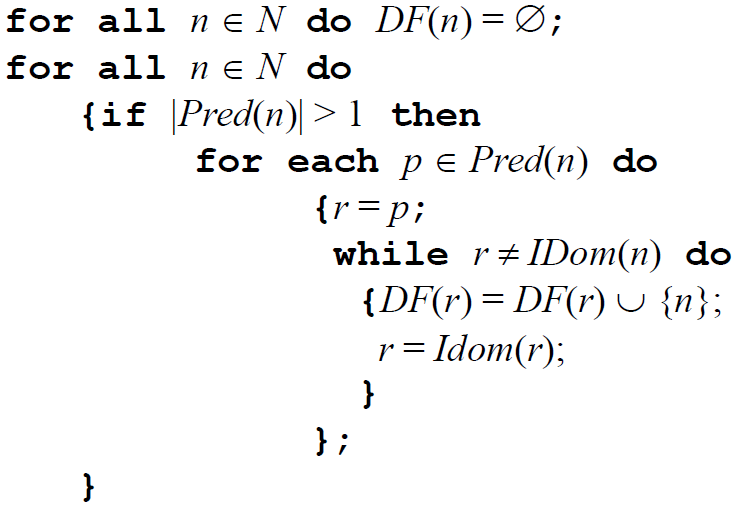
[Тесты](https://gist.githubusercontent.com/anonymous/467670ec40737623339544f268134381/raw/3dom)

## dominance frontier

Наивно, по определению - можно и тупым перебором. Определение, кстати:

|  |
| --- |
| Множество узлов m, удовлетворяющих условиям:   1. n является доминатором предшественника m  **p ∈ Pred(m) & n ∈ Dom(p)** 2. n **не** является строгим доминатором m  **n ∉ (Dom(m) \ {m})**   называется границей доминирования n и обозначается DF(n) |

Раз уж есть алгоритм из лекций для прошлой задачи, пусть будет и для этой.



(IDom - непосредственный доминатор, DF - граница доминирования)

Почему то, что он делает, правда? Потому, что есть

**Свойства границы доминирования** (почти из слайдов)

1. Если узел0 – граница доминирования, то в него входит больше одного ребра (из определения, очевидно)
2. Если предок узла0 не является его доминатором, то узел0 – граница доминирования для этого предка (из определения, очевидно)
3. Если узел0 – граница доминирования узла1, то узел0 – граница доминирования всех доминаторов узла1, не являющихся доминаторами узла0 (из определения: дом0 является доминатором предшественника – узла1 – но не является доминатором самого узла0)

**Рандомный факт о доминаторах №1** - на любом пути от узла до entry встречаются все его доминаторы (очевидно, из определения доминатора)

Как найти непосредственный доминатор? Один из способов - использовать

**Рандомный факт о доминаторах №2** - Если дом0 – непосредственный доминатор узла0, то все доминаторы узла0 являются доминаторами дом0 (из 1 и определения доминатора). Поэтому достаточно проверить для каждого доминатора узла, являются ли его доминаторами все остальные (что очень удобно с использованием множеств dom из прошлой задачи, “невывернутых”)

[Тесты](https://gist.githubusercontent.com/anonymous/467670ec40737623339544f268134381/raw/3df)

## dead code elimination

### **Теория**

В этой задаче разрешается пользоваться чуть б**о**льшим множеством предоставляемых ими функций. В частности, SSA-форму строят за нас (см. заготовку в контесте). Предлагается использовать mark and sweep: сначала функция Mark находит и помечает все “полезные” операции, затем функция Sweep удаляет бесполезные ветвления и операции.

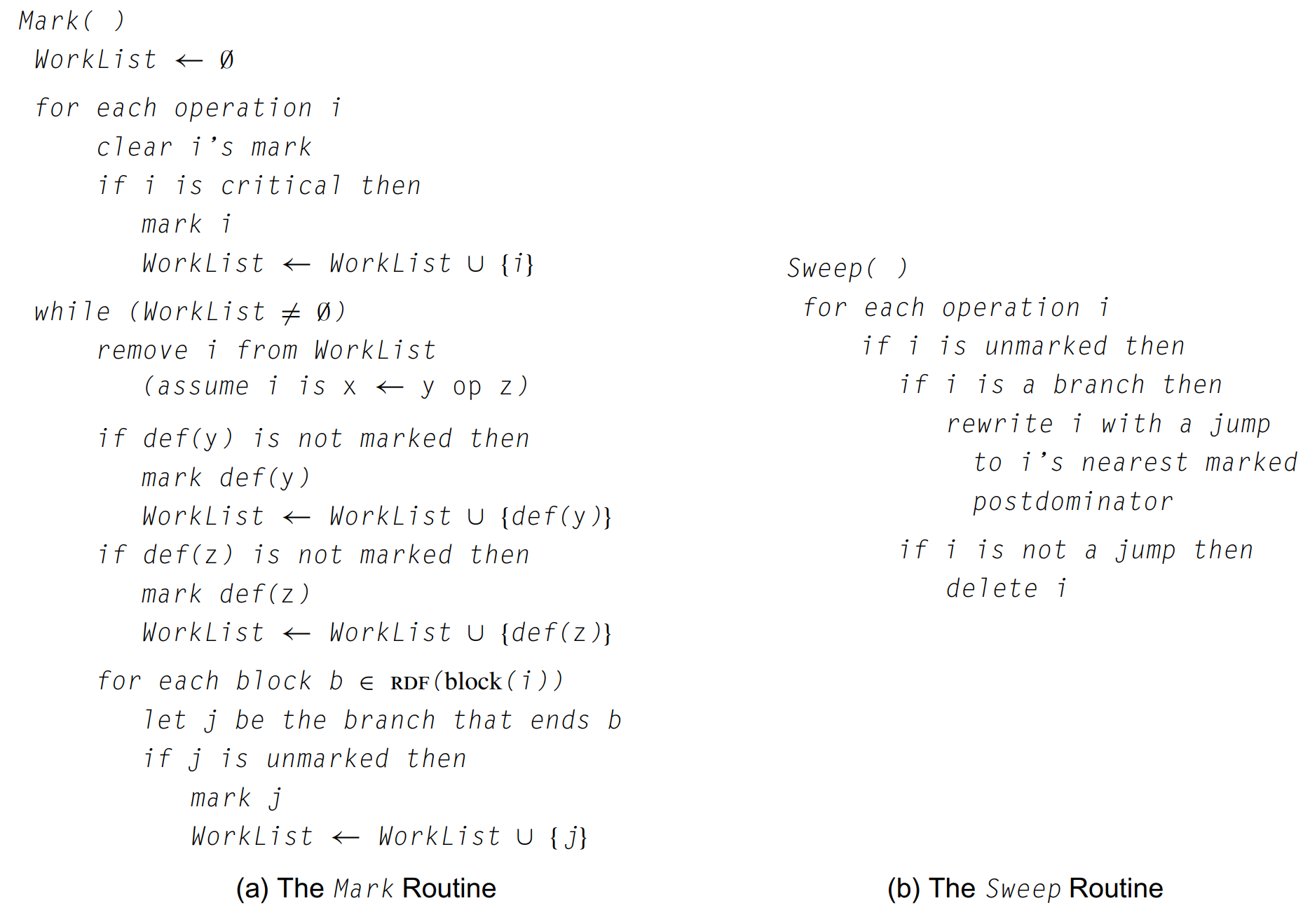
Mark relies on reverse dominance frontiers, which derive from the dominance frontiers used in the ssa construction.

*Reverse dominance frontiers* (*обратные границы доминирования*) are simply dominance frontiers computed on the reverse cfg.

*Reverse CFG* (*обратный граф потока*) is the CFG with its edges reversed.

Итого, предначертано:

* Привести программу в SSA-форму с помощью заготовки.
* Посчитать RDF.
  + Значит, сначала построить доминаторы, а потом границы доминирования, поменяв в алгоритмах preds <-> s1,s2, чтобы было как в обратном графе.
* Запомнить, в какой инструкции определяется каждая переменная (по определению SSA-формы к переменным присваивается значение только единожды), это пригодится для Mark.
* Пометить полезные инструкции (Mark).
* Удалить бесполезные бранчи и инструкции (Sweep).



#### **Разбор функции Mark из книжки**.

Сжато, алгоритм - это добавление операций в множество полезных, начиная с тех, что непосредственно влияют на побочные эффекты или возвращаемое значение функции, и рекурсивно распространяя “полезность” на задействованные в этом переменные.

* **WorkList:** операции, недавно помеченные как полезные. В начале - все критические. Итерирование происходит, пока он не будет пуст.
* **critical**: Критическими считайте инструкции записи в память, инструкции возврата из функции и любые вызовы функций.
* **def(operand)**: операция, где определяется operand. Если операцию пометили полезной, то ее операнды полезны, значит, полезны инструкции, где эти операнды определяются.
* **Последний цикл:** операции ветвления являются полезными только если от их наличия зависит, выполнится ли другая полезная операция. Так вышло, что RDF(block) - это множество блоков, в которых присутствуют операции ветвления, которые полезны, потому что какие-то операции в block полезны. Ну, их надо обойти и пометить эти ветвления как полезные.

#### **Функция Sweep**

В условии сказано “Бесполезные инструкции следует заменять инструкциями nop без операндов”, а также “Неиспользуемые фи-функции нужно просто удалить (не заменять на nop)”, поэтому функция Sweep будет 1) заменять на nop бесполезные инструкции, 2) удалять бесполезные фи-функции, 3) а с ветвлениями делать вот что:

Sweep replaces any unmarked branch with a jump to its first postdominator that contains a marked operation. To find the nearest useful postdominator, the algorithm can walk up the postdominator tree until it finds a block that contains a useful operation. Since, by definition, the exit block is useful, this search must terminate.

Алсо, [дерево постдоминаторов == дереву доминаторов обратного графа](https://www.cs.utexas.edu/~pingali/CS380C/2016-fall/lectures/Dominators.pdf), а мы его как раз все равно строим, пока ищем RDF.

Осторожно, тест номер 9 проверяет, не зациклится ли этот поиск.

### **Практика**

Из текста задания:

* Структура [Ins](https://c9x.me/git/?p=qbe.git;a=blob;f=all.h;h=24a175526ffea1d56d6084f24d1ef496be716dd7;hb=HEAD#l198) не описывает инструкции, встречающиеся на границах базовых блоков: [терминаторы (переходы)](https://c9x.me/git/?p=qbe.git;a=blob;f=all.h;h=24a175526ffea1d56d6084f24d1ef496be716dd7;hb=HEAD#l218) и [фи-функции](https://c9x.me/git/?p=qbe.git;a=blob;f=all.h;h=24a175526ffea1d56d6084f24d1ef496be716dd7;hb=HEAD#l205).  
  Это означает то, что “нормальные” инструкции хранятся в Blk->ins, фи-функции лежат в Blk->phi (в отличие от ins это не массив, а список), отдельно их положили как минимум потому, что все фи-функции выполняются перед остальными инструкциями; а инструкция-терминатор в блоке всего одна, в конце, и она в Blk->jmp.
* Для описания обобщённых инструкций удобно использовать структуру [Use](https://c9x.me/git/?p=qbe.git;a=blob;f=all.h;h=24a175526ffea1d56d6084f24d1ef496be716dd7;hb=HEAD#l242) (поле bid в ней предназначено для хранения идентификатора блока, [идентификаторы блоков](https://c9x.me/git/?p=qbe.git;a=blob;f=all.h;h=24a175526ffea1d56d6084f24d1ef496be716dd7;hb=HEAD#l226) заполняются функцией fillrpo).  
  Здесь была попытка донести мысль, что с тремя различными сущностями (просто инструкциями, переходами и фи-функциями) мудохаться неудобно, поэтому их для удобства можно обернуть все в Use, и в алгоритмах свитчиться по Use->type.
* Инструкции–терминаторы, в свою очередь, делятся на инструкции безусловного перехода (Jjmp), ветвления (Jjnz) и возврата (используйте макрос [isret](https://c9x.me/git/?p=qbe.git;a=blob;f=all.h;h=24a175526ffea1d56d6084f24d1ef496be716dd7;hb=HEAD#l179) для проверки на принадлежность к данному классу).  
  Т.к. функции могут принимать произвольное число аргументов, инструкции вызова в QBE не имеют операндов, вместо чего они предваряются нужным числом инструкций–аргументов (из-за того, что инструкция во внутреннем представлении QBE может иметь не более двух операндов). Cм. обработку инструкций Ocall и Ovacall [здесь](https://c9x.me/git/?p=qbe.git;a=blob;f=amd64/sysv.c;h=014452aeace761e7fff2b8c5b3800c16e50631ea;hb=HEAD#l671).  
  Эти проверки нужна для определения критичности инструкции. Список “запись в память, возврат или вызов функции” в итоге проверяется функциями isret(), isstore() и проверкой на Ocall, Ovacall и isarg() как в примере по ссылке.

[**Тесты**](https://pastebin.com/raw/KVMwMxAc)

### Функции к задаче 3dce

Объяснение того, что делают некоторые функции из заготовки:

|  |  |
| --- | --- |
| fillrpo(fn) | Обход дерева блоков в reverse postorder порядке, вычисленные номера блоков записываются в blk->id.Нужно для ssa(), но еще идентификаторами блоков удобно пользоваться, например, при заполнении Use. Во второй раз она вызывается затем, что недостижимые блоки она удаляет. |
| fillpreds(fn) | Заполнение полей pred и npred во всех блоках функции. Нужно как для ssa(), так и для printfn() (во второй раз - потому что в Sweep мы пересоединяем блоки, изменяя ветвления). |
| filluse(fn) | Заполнение структур fn->tmp для функции fn и структур tmp.use для всех fn->tmp.Нужно для ssa(). |
| ssa(fn) | Построение SSA-формы. По пути заполняет ~~dominance frontier, кладет в blk->fron (размер множества - в blk->nfron); заполняет живые переменные, но хуй знает куда кладет~~ кучу всякой не нужной в этом задании фигни; главное - она меняет начинку функции, т.е. строит SSA-форму in place. |
| printfn(fn, stdout) | Печать функции fn в виде кода на QBE. |