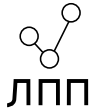


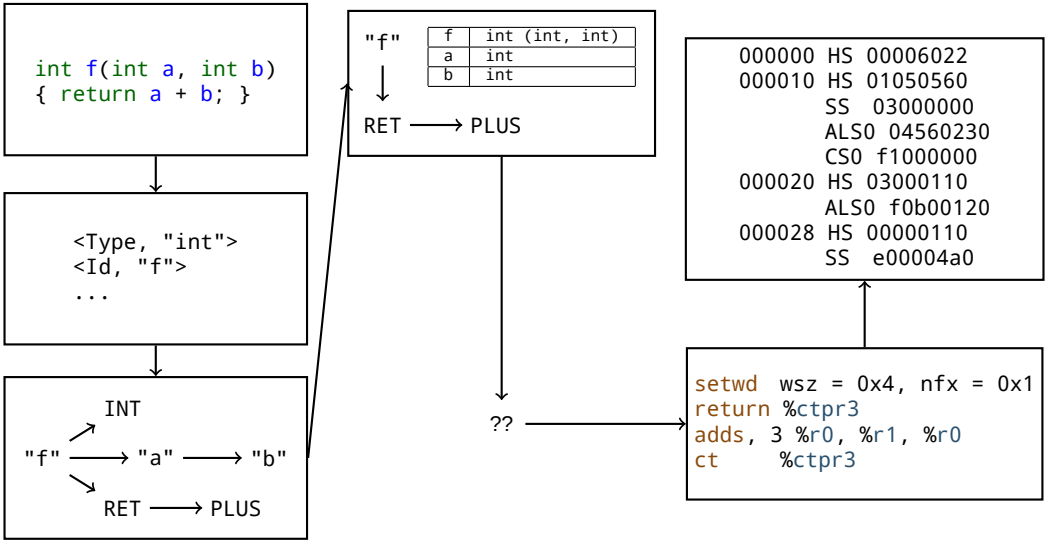
Проектирование компиляторов

Лекция 5.
Внутреннее представление программы в компиляторе.

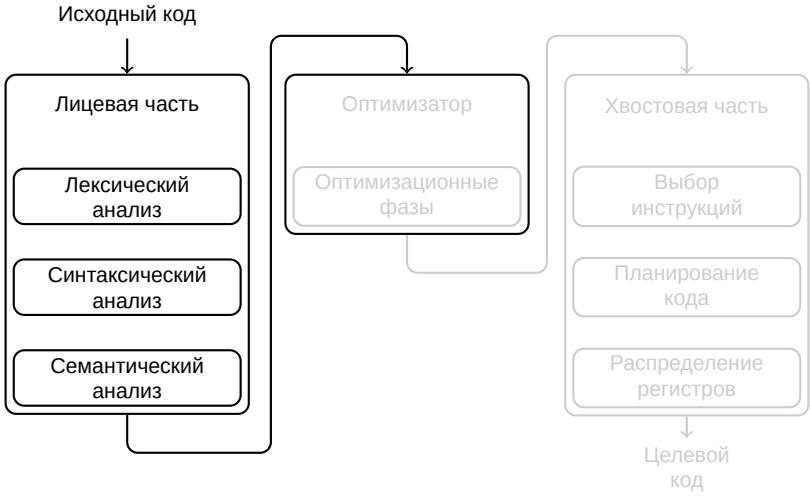
Маркин А. Л.
master@lab-pp.ru

2023

Компиляция программы



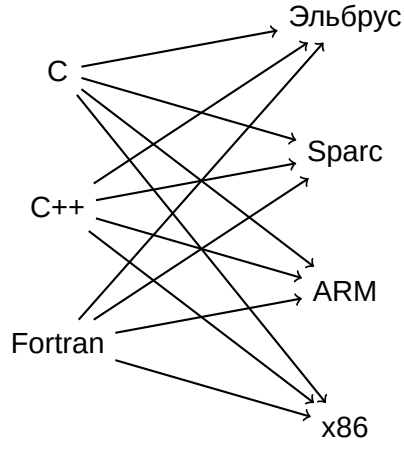
Компиляция программы



Проблема

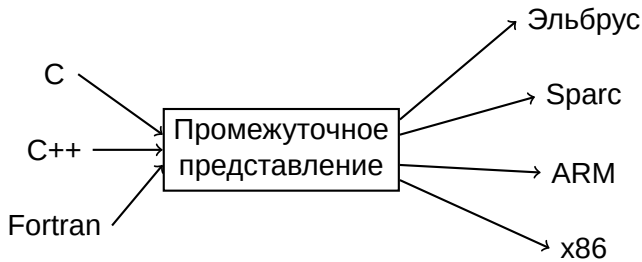
Деревья абстрактного синтаксиса имеют слишком высокий уровень абстракции и слишком зависят от свойств языка:

- разные языки программирования требуют разных реализаций лицевой части
- разные процессоры требуют разных реализаций хвостовой части



Промежуточное представление

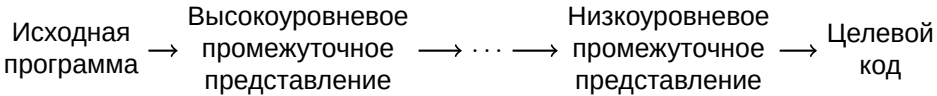
Для уменьшения объёма работ различные лицевые части приводят программы к единому виду: **промежуточному представлению**.



Промежуточное представление (IR, Intermediate Representation) — версия кода оригинальной программы, созданная для абстрактной вычислительной машины с целью проведения анализов и преобразований.

Промежуточное представление

В процессе трансляции компилятор может строить последовательность различных промежуточных представлений:



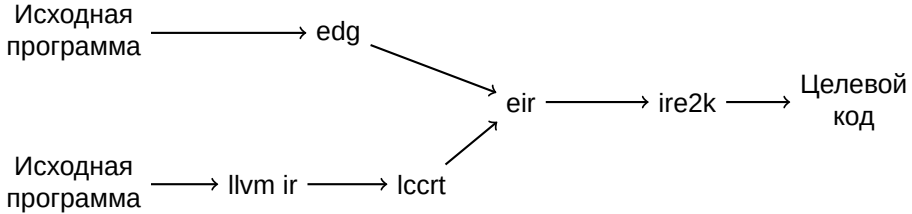
На высоком уровне возможно делать оптимизации и анализы, основанные на свойствах языка программирования.

На низком уровне возможно делать оптимизации, основанные на особенностях целевой машины

Промежуточное представление

Пример различных представлений в компиляторе gcc:

Пример различных представлений в компиляторе lcc:



Промежуточное представление

Промежуточные представления в компиляторе Эльбруса:

- **eir** (Elbrus Intermediate Representation) — высокоурвневое промежуточное представление.
  - Архитектурно-независимое
  - Оперирует объектами программы
  - Обладает информацией о типах и прочей информацией, приходящей из языка
- **ire2k** (Intermediate Representation for Elbrus 2000) — низкоуровневое промежуточное представление.
  - Архитектурно-зависимое, близкое к ассемблеру Эльбруса
  - Оперирует регистрами
  - Теряет семантическую информацию, приходящую из языка
  - Позволяет генерировать целевой ассемблер

Промежуточное представление

Для компилятора важно качественно спроектировать промежуточное представление. В зависимости от решаемых задач на первый план могут выходить различные свойства представления:

- простота создания кода
- простота преобразования в целевой код
- простота преобразований
- объём кода
- уровень абстракции

Часто для одного представления невозможно совместить все свойства.

Виды промежуточных представлений

Структурные представления:

- Ориентированы на представление в виде графов, списков или деревьев
- Широко применяются для трансляции из одного языка в другой
- Обычно занимают большой объём памяти

Примеры:

- Деревья разбора
- Деревья абстрактного синтаксиса
- Направленные ациклические графы

Виды промежуточных представлений

Линейные представления:

- Представляют псевдокод для абстрактной машины
- Уровень абстракции может меняться
- Легче перестраивать

Примеры:

- Одноадресный код, эмулирующий стековую машину
- Трёхадресный код, эмулирующий RISC машину
- Псевдоассемблер для генерации кода конкретной машины

Виды промежуточных представлений

Смешанные:

- Представляют комбинацию графов и линейного кода
- Позволяют проводить большое количество оптимизаций

Примеры:

- Граф потока управления
- Граф определений и использований
- Граф зависимостей

Трёхадресный код

В качестве представления часто используют **трёхадресный код**, состоящий из операции, адресов двух аргументов и адреса результата.

Трёхадресный код представляет собой линеаризованное представление синтаксического дерева.

Операции трёхадресного кода имеют следующий формат:

```
результат = аргумент1 операция аргумент2
результат = операция аргумент
```

Трёхадресный код

Трёхадресный код для выражения `a = b * -c + b * -c` может выглядеть следующим образом:

```
t1 = -c
t2 = b * t1
t3 = -c
t4 = b * t3
t5 = t2 + t4
a  = t5
```

Переменные `t*`, созданы компилятором как временное хранилище промежуточных значений.

Трёхадресный код

Внутри компилятора трёхадресный код может быть представлен структурой из четвёрок:

```
t1 = -c
t2 = b * t1
t3 = -c
t4 = b * t3
t5 = t2 + t4
a  = t5
```

Операция	Аргумент 1	Аргумент 2	Результат
SUB	c		t_1
MUL	b	t_1	t_2
SUB	c		t_3
MUL	b	t_3	t_4
ADD	t_2	t_4	t_5
MOV	t_5		a

Трёхадресный код

Или в виде троек:

```
t1 = -c
t2 = b * t1
t3 = -c
t4 = b * t3
t5 = t2 + t4
a  = t5
```

№	Операция	Аргумент 1	Аргумент 2
0	SUB	*c*	
1	MUL	*b*	(0)
2	SUB	*c*	
3	MUL	*b*	(2)
4	ADD	(1)	(3)
5	MOV	*a*	(4)

Аргументами в данном случае являются либо адреса переменных, либо ссылки на результат предыдущих операций.

Представление основных языковых конструкций в промежуточном коде

Примеры кода

Некоторые операции eir:

- `READ, RD_FIELD` — чтение скалярного объекта и поля структуры.
- `WRITE, WR_FIELD` — запись в скалярный объект и в полне структуры.
- `ADD_I, SUB_I, ...` — целочисленные арифметические операции.
- `N2NS, N2ND` — получение адреса поля структуры и элемента массива.
- `EQ_I, GT_I, LT_I, ...` — целочисленные сравнения.
- `BRANCH, CALL, RETURN, RETVAL` — условный переход, вызов процедуры, возврат из процедуры без значения и со значением.

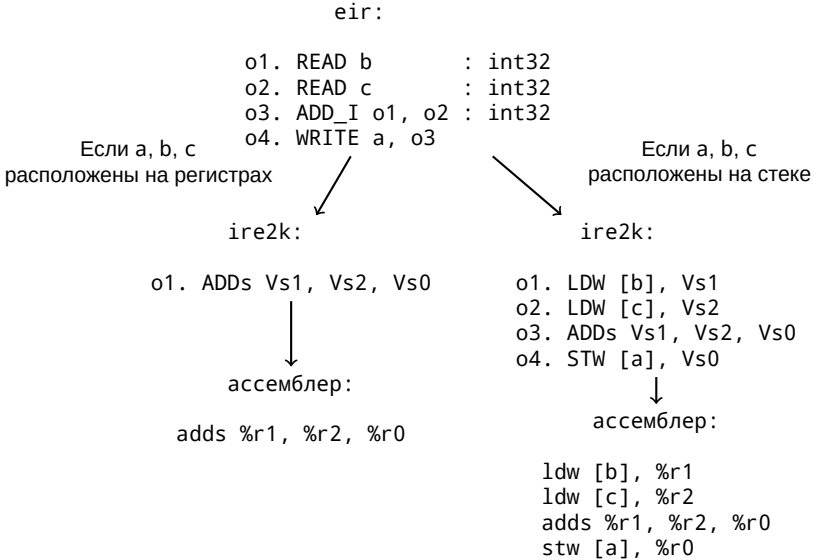
Примеры кода

Некоторые операции ire2k:

- `LDW, LDD, ...` — загрузка слова и двойного слова.
- `STW, STD, ...` — запись слова и двойного слова
- `ADDs, SUBs, ...` — целочисленные арифметические операции
- `CMPEs, CMPBs, ...` — целочисленные сравнения одинарных слов
- `BRANCH, CALL, RETURN` — условный переход, вызов процедуры, возврат из процедуры.

Примеры кода

Выражение `a = b + c;`



Примеры кода

Выражение `b = a[4];`

eir:

```
o1. CONST 4      : int32
o2. N2ND a, o1   : * int32
o3. READ o2      : int32
o4. WRITE b, o3
```

ire2k:

```
o1. MULs 0x4, 0x4, Vs0
o2. LDW [a], Vs0, Vs1
```

ассемблер:

```
muls 0x4, 0x4, %r0
ldw [a], %r0, %r1
```

Выражение `a[5] = 0;`

eir:

```
o1. CONST 5      : int32
o2. N2ND a, o1   : * int32
o3. CONST 0      : int32
o4. WRITE o2, o3
```

ire2k:

```
o1. MULs 0x5, 0x4, Vs0
o2. STW [a], Vs0, 0x0
```

ассемблер:

```
muls 0x5, 0x4, %r0
stw [a], %r0, 0x0
```

Примеры кода

Выражение `a = s.f;`

```
eir:

o1. N2NS s.f  : * int32
o2. READ o1   : int32
o3. WRITE a, o2
```

```
ire2k:

o1. LDW [s], 0x0, Vs0
```

```
ассемблер:

ldw [s], 0x0, %r0
```

Выражение `s.f = 5;`

```
eir:

o1. CONST 5     : int32
o2. N2NS s.f    : * int32
o3. WRITE o2, o1
```

```
ire2k:

o1. STW [s], 0x0, 0x5
```

```
ассемблер:

stw [s], 0x0, 0x5
```

Примеры кода

Выражение `i = 0; do { i++; } while( i < 10);`

eir:

```
o1.  CONST 0       : int32
o2.  WRITE i, o1
o3.  CONST 1       : int32
o4.  READ i        : int32
o5.  ADD_I o4, o3  : int32
o6.  WRITE i, o5
o7.  CONST 10      : int32
o8.  READ i        : int32
o9.  LT_I o8, o7
o10. BRANCH o9, (t: o3, f: o11)
o11. ...
```

→

ire2k:

```
o1. MOVs    0x0, Vs0
o2. ADDs    0x1, Vs0, Vs0
o3. CMPBs   Vs0, 0xa, P0
o4. CT      o2, C0
o5. BRANCH  C0, P0 [T]
...
```

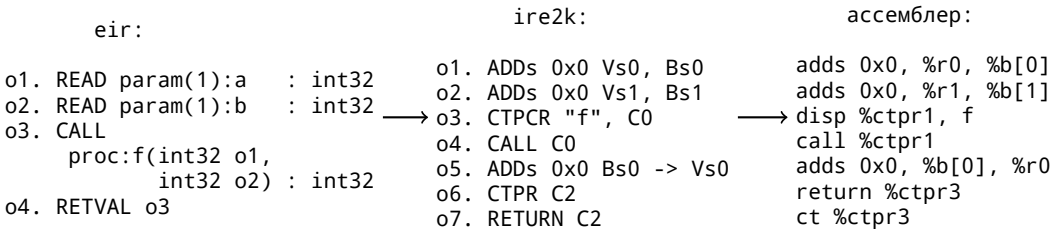
⟶

ассемблер:

```
adds 0x0, 0x0, %r0
.l1:
adds 0x1, %r0, %r0
cmpbsb %r0, 0xa, %p0
disp %ctpr0, .l1
ct %ctpr0 | %p0
...
```

Примеры кода

Выражение `int g(int a, int b) {return f(a, b);}`



Представление основных языковых конструкций в промежуточном коде

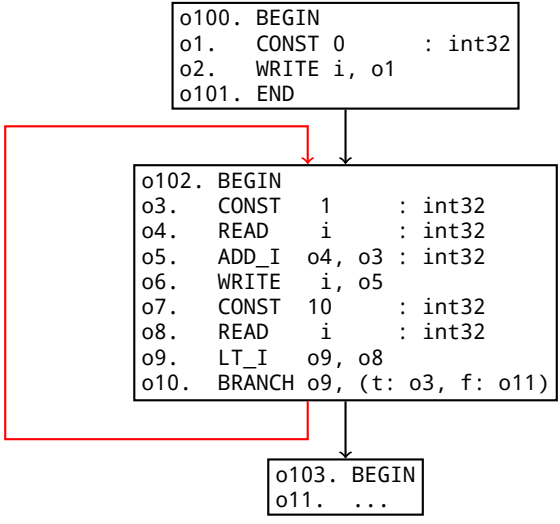
Проблема

По линейному представлению кода сложно определять реальную последовательность исполнения операций. Также сложно понимать где и какие циклы расположены:

```
o1.  CONST 0      : int32
o2.  WRITE i, o1
o3.  CONST 1      : int32
o4.  READ i       : int32
o5.  ADD_I o4, o3 : int32
o6.  WRITE i, o5
o7.  CONST 10     : int32
o8.  READ i       : int32
o9.  LT_I o9, o8
o10. BRANCH o9, (t: o3, f: o11)
o11. ...
```

Проблема

Для удобства на основе операций процедуры строится граф потока управления:



Граф потока управления

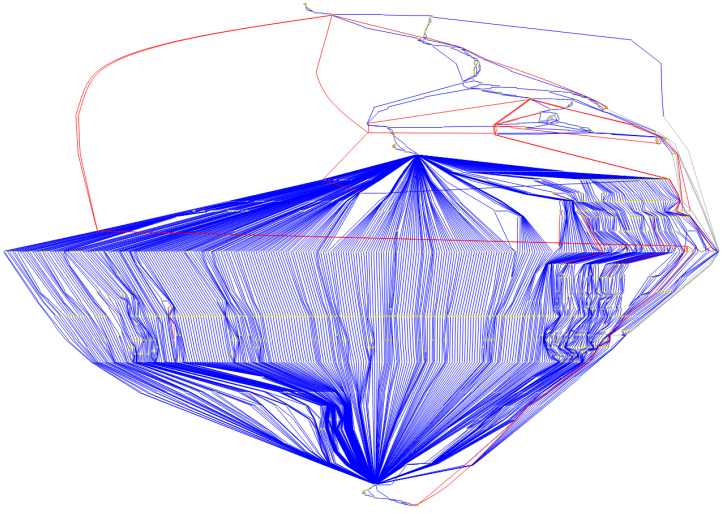
Линейный код разрезается на участки таким образом чтобы последней операцией участка была операция перехода. Первой операцией участка будет ближайшая к концу участка операция, на которую совершается переход. Такие участки называются **линейным участками (базовыми блоками)**.

В линейном участке:

- не может быть более одной операции вызова
- переходом может быть только одна операция

Граф потока управления

Пример графа потока управления функции yyparse из 176.gcc:



Граф потока управления

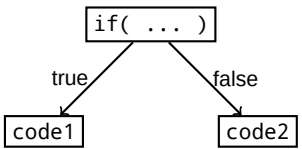
Граф Потока Управления (CFG — Control Flow Graph) — направленный граф $G = (N, E)$. Каждый узел $n \in N$ представляет из себя линейный участок. Каждая дуга $e = (n_i, n_j) \in E$ представляет возможный переход от одного линейного участка к другому.

Граф потока управления отображает структуру каждой процедуры и является необходимой структурой в оптимизирующем компиляторе.

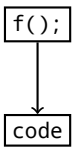
При построении графа принято делать точкой входа фиктивный входной (**START**) узел, а все выходы из процедуры сводить к фиктивному выходному узлу (**STOP**).

Основные языковые конструкции на графе

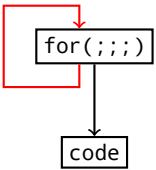
Условие



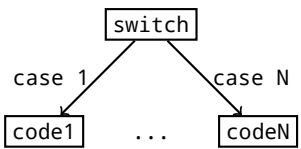
Вызов



Цикл



Коммутация



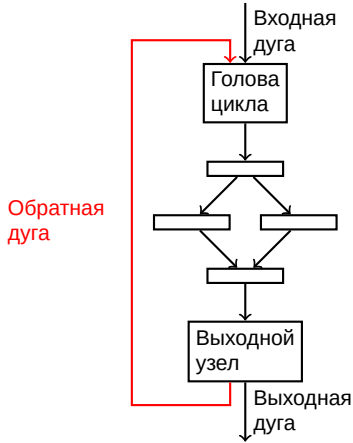
Циклы

Цикл представляет из себя **ССК — Сильно связанную компоненту**, т.е. такой подграф $G = (N_S, E_S)$ что каждый узел из N_S достижим из каждого другого узла по пути, включающем только дуги из E_S.

Цикл состоит из **входных дуг** — внешние дуги, по которым производится вход в цикл; **головы цикла** — узла, с которого цикл начинает своё исполнение; **обратных дуг** — дуг, ведущих из узла цикла в голову; **выходных дуг** — дуг, выводящих управление из цикла и **выходных узлов** — узлов, в которых есть выходные дуги.

Циклы

Пример графа простого цикла:



Циклы

Цикл может иметь только одну голову, как минимум одну обратную дугу и обычно имеет как минимум одну выходную дугу.

Вход в цикл может быть как через голову, так и через другие узлы. Вход в цикл не через голову называется **боковым**.

Цикл, содержащий боковые входы называется **несводимым**. Такие циклы накладывают большие ограничения на возможности их оптимизации.

Дерево циклов

Если циклы A и B обладают головами a и b соответственно, $a \neq b$ и $b \in A$, то узлы цикла B являются подмножеством узлов цикла A. В этом случае говорят что цикл B **вложен** в цикл A.

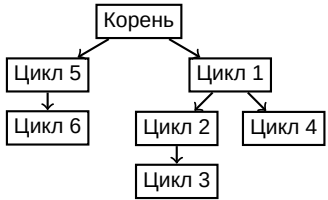
Дерево циклов отображает вложенность циклов друг в друга.

Дерево циклов

```
for() // Цикл 1
{
    for() // Цикл 2
    {
        for() // Цикл 3
        {
        }
    }

    for() // Цикл 4
    {
    }
}

for() // Цикл 5
{
    for() // Цикл 6
    {
    }
}
```



Профиль программы

Профиль программы

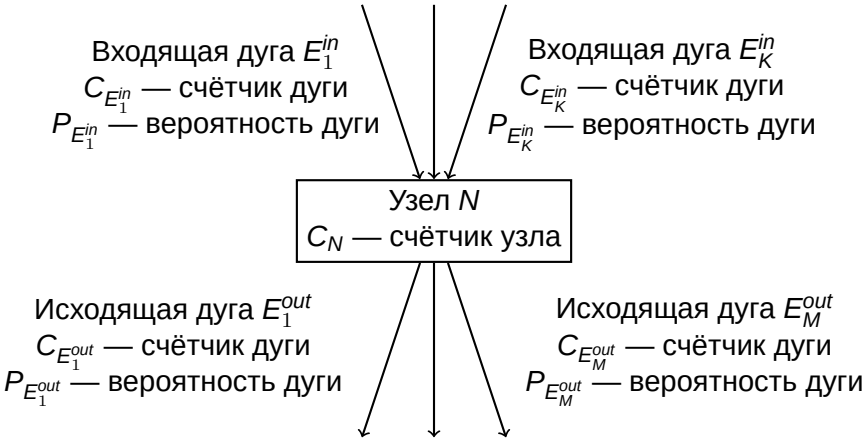
Для качественной работы оптимизатора компилятору необходимо знать информацию о количестве проходов по каждому узлу графа потока управления. Такая информация называется **профильной**.

Счётчики исполнения — значения, указывающие на то сколько раз, программа исполнит данный линейный участок, или перейдёт по данной дуге.

Вероятность перехода — значения, указывающие на то с какой вероятностью будет совершён переход из данного линейного участка по данной дуге.

Профиль программы

Проифильная информация узла графа потока управления:



Профиль программы

Один из способов получения профильной информации — посмотреть фактическое поведение программы и записать количество проходов по каждому линейному участку. Для этого применяется двупроходная схема компиляции:

1. **Инструментирующая компиляция** — компиляция программы с автоматическим добавлением кода, отслеживающего поведение программы.
2. Исполнение программы на обучающей выборке. По итогам этого исполнения программа создаст данные о своём поведении.
3. **Профилирующая компиляция** — компиляция программы с профильными данными, используемыми оптимизатором.

Профиль, полученный таким способом называется **реальный (или динамический)**.

Профиль программы

Использование реального профиля связано с рядом трудностей, поэтому применяется такой профиль редко.

Если реальной профилирующей информации в компиляторе нет, то компилятор пытается угадать поведение программы и самостоятельно формирует её профиль на основе некоторых признаков и догадок. Такой профиль называется **предсказанным (или статическим)**.

Литература

- *А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман* Компиляторы: принципы, технологии и инструменты.
- *S. S. Muchnick* Advanced Compiler Design and Implementation.
- *K. D. Cooper, L. Torczon* Engineering a compiler.
- *R. Allen, K. Kennedy* Optimizing Compilers for Modern Architectures: A Dependence-based Approach.