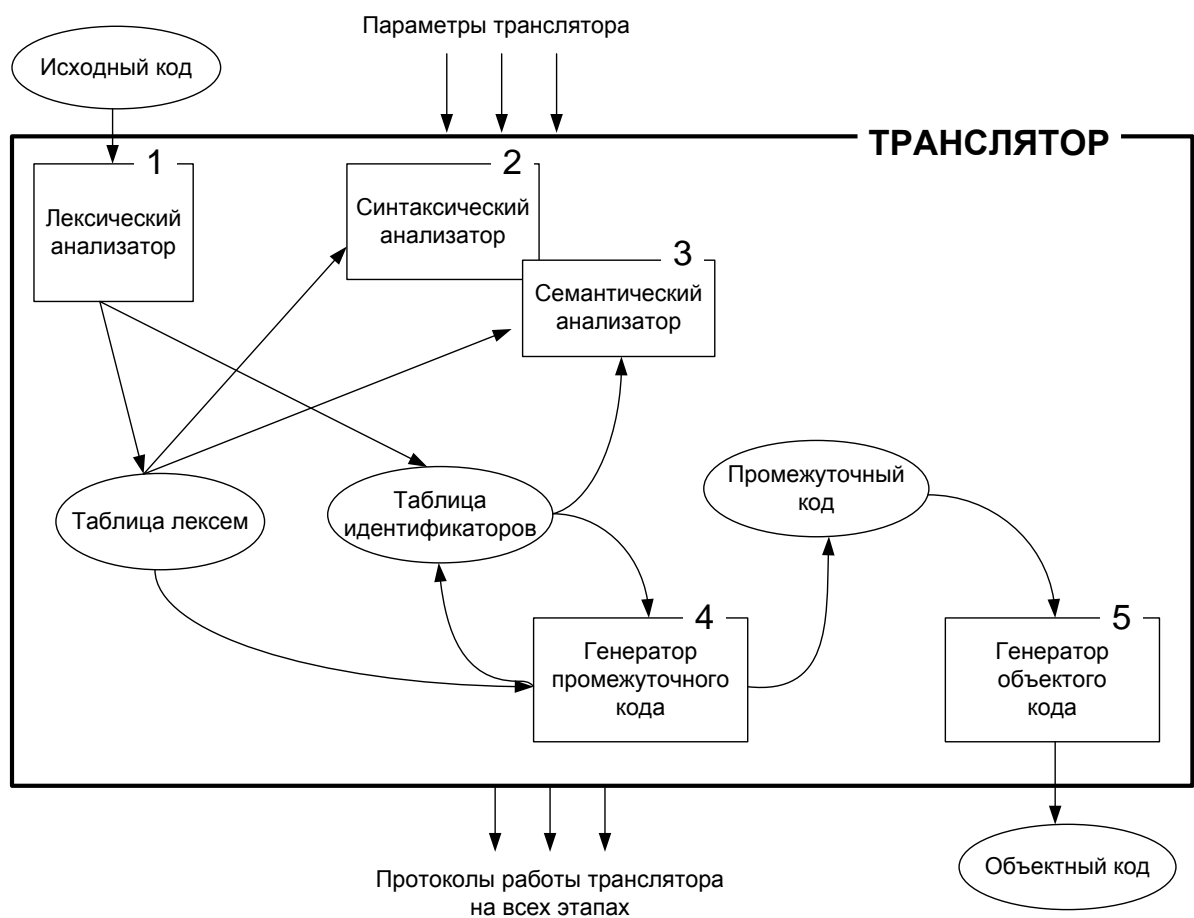


## Генерация и оптимизация кода

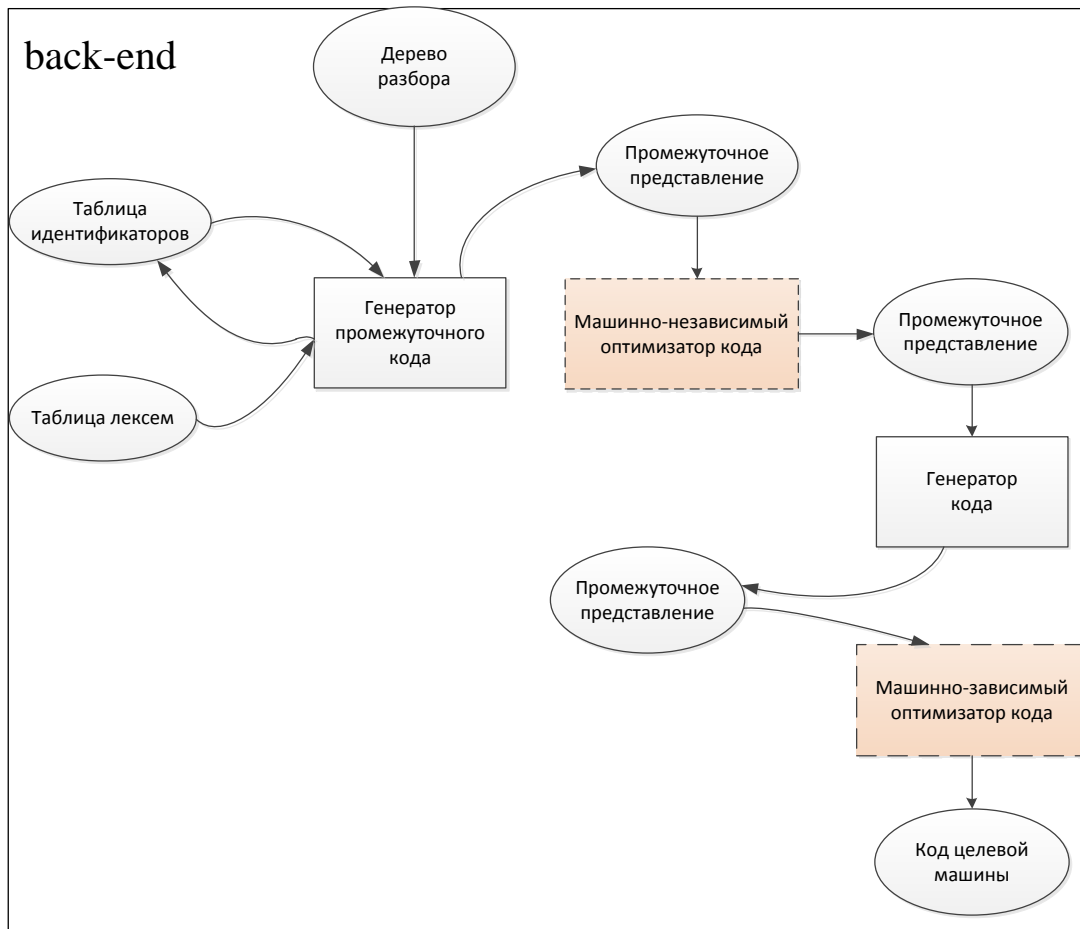
Большое интервью с Клиффом Кликом — отцом JIT-компиляции в Java

<https://habr.com/ru/company/jugru/blog/458718/>

## 1. Общая структура транслятора



## 2. Оптимизация выполняется на этапах подготовки к генерации и непосредственно при генерации объектного кода.



Оптимизация — это процесс преобразования части кода в другую функционально эквивалентную часть для улучшения одной или более характеристик кода.

Критерии оптимизации:

- скорость выполнения кода;
- размер кода;
- время компиляции кода;
- минимизация энергопотребления;
- получение более компактного кода;
- уменьшение количества операций ввода-вывода.

Два основных вида оптимизирующих преобразований:

- преобразования исходной программы (в форме ее внутреннего представления в компиляторе), не зависящие от результирующего объектного языка;
- преобразования результирующей объектной программы.

Классификация оптимизации:

- локальные (оператор, последовательность операторов, базовый блок);
- внутрипроцедурные;
- межпроцедурные;
- внутримодульные;
- глобальные (оптимизация всей программы, «оптимизация при сборке», межпроцедурная оптимизация).

**Peephole-оптимизация** (щелевая оптимизация) –рассматривает несколько соседних инструкций промежуточного кода («смотрит в глазок») на код для определения возможных преобразований с целью оптимизации.

*Например*, удвоение числа эффективнее выполнить с использованием операции левого сдвига или сложением числа с таким же числом.

*Например*, некоторые инструкции могут быть заменены одной инструкцией или более короткой последовательностью инструкций.

**Внутрипроцедурная оптимизация** — оптимизации, выполняемые в рамках единицы трансляции (например, функции или процедуры).

**Методы оптимизации кода** – зависят от типов синтаксических конструкций исходного языка:

- линейных участков программы;
- логических выражений;
- вызовов процедур и функций;
- других конструкций входного языка.

Во всех случаях могут использоваться как машинно-зависимые, так и машинно-независимые методы оптимизации.

### 3. Машинно-независимая оптимизация.

Примеры оптимизации линейных участков программы.

#### 3.1 Генерация более эффективных команд для частных случаев

В лекции 24 рассматривали генерацию в промежуточный код в виде тетрад. Рассмотрим пример, с использованием следующих тетрад:

<code>+(p1,p2,p3)</code>	Вычислить сумму двух integer-значений, результат поместить в стек <code>p1</code> = значение 1 <code>p2</code> = значение 2 <code>p3</code> = адрес результата в стеке
<code>store(p1,p2,p3)</code>	Скопировать данные <code>p1</code> = адрес источника <code>p2</code> = адрес получателя <code>p3</code> = null

*Пример:*

вычислить `c = a + b`, для частного случая `b = 0`

Выражение	Запись в виде тетрад до оптимизации	Запись в виде тетрад после оптимизации
<code>c = a + b</code>	<code>+(a, b, T1)</code> <code>store(T1, c, null)</code>	<code>store(a, c, null)</code>

*Пример:*

замена операций эквивалентными с меньшей стоимостью исполнения.

Выражение	После прямого преобразования
<code>y = y * 2;</code> <code>x = x * 4;</code> <code>x = x / 2;</code>	<code>y = y + y;</code> <code>x = x &lt;&lt; 2;</code> <code>x = x &gt;&gt; 1;</code>

### 3.2 Удаление недостижимого кода

До оптимизации

После оптимизации

**if (1) s1; else s2;      ➔    s1;**

если предикат условного оператора (в примере 1) всегда принимает значение «истина», то else-выражение будет недостижимым => условный оператор можно заменить оператором s1;

*Пример.* Пусть

**if (1) then a:=1 else a:=2**

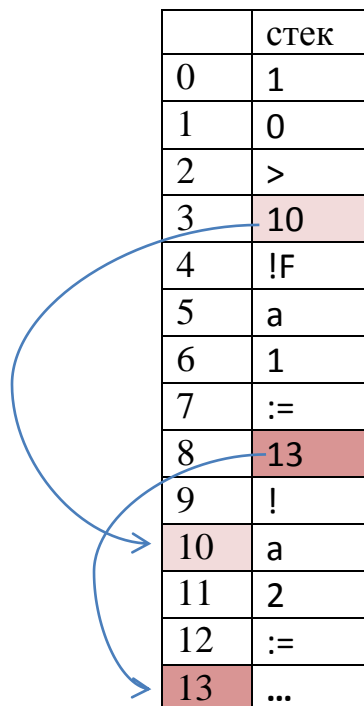
ПОЛИЗ этого условного оператора:

**В p<sub>1</sub> !F S<sub>1</sub> p<sub>2</sub> ! S<sub>2</sub> ...**

где В – условие. Если операнды в условии – константы, то результат известен, тогда, сделав соответствующие проверки во время генерации кода, получим:

До оптимизации

После оптимизации



	стек
0	1
1	0
2	>
3	10
4	!F
5	a
6	1
7	:=
8	13
9	!
10	a
11	2
12	:=
13	...

	стек
0	a
1	1
2	:=
3	...

### 3.3 Оптимизация линейных участков программы:

#### а) Удаление бесполезных присваиваний

для последовательности выражений:

```
a=b*c;  
d=b+c;  
a=d*c;
```



получим:

```
d=b+c;  
a=d*c;
```

Выражение	Запись в виде тетрад до оптимизации	Запись в виде тетрад после оптимизации
a=b*c;	$*(b,c,T1)$ store(T1,a,null)	
d=b+c;	$+(b,c,T2)$ store (T2, d, nul)	$+(b,c,T2)$ store (T2, d, nul)
a=d*c;	$*(d,c,T3)$ store(T3,a,null)	$*(d,c,T3)$ store(T3, a, null)

**б) Исключение избыточных вычислений:**

для последовательности выражений:

**d=d+b\*c;**  
**a=d+b\*c;**  
**c=d+b\*c;**



получим:

**t=b\*c;**  
**d=d+t;**  
**a=d+t;**  
**c=a;**

Выражение	Числа зависимости переменных				До оптимизации	dep (i)	Дополнительный шаг	После оптимизации
	a	b	c	d				
d=d+b*c;	0	0	0	0	1 <del>*(b,c,T1)</del>	1	1 <del>*(b,c,T1)</del>	<del>*(b,c,T1)</del>
	0	0	0	0	2 <del>+(T1,d,T2)</del>	2	2 <del>+(T1,d,T2)</del>	<del>+(T1,d,T2)</del>
	0	0	0	3	3 store(T2,d,nul)	3	3 store(T2,d,nul)	store(T2,d,nul)
a=d+b*c;	0	0	0	3	4 <del>*(b,c,T3)</del>	1	1 <del>*(b,c,T1)-&gt;s(b,c,T1)</del>	<del>*(b,c,T1)</del>
	0	0	0	3	5 <del>+(T3,d,T4)</del>	4	2 <del>+(T1,d,T4)</del>	<del>+(T1,d,T2)</del>
	6	0	0	3	6 store(T4,a,nul)	5	3 store(T4,a,nul)	store(T2,a,nul)
c=d+b*c;	6	0	0	3	7 <del>*(b,c,T5)</del>	1	1 <del>*(b,c,T1)-&gt;s(b,c,T1)</del>	<del>*(b,c,T1)</del>
	6	0	0	3	8 <del>+(T5,d,T6)</del>	6	8 <del>+(T1,d,T6)</del>	<del>+(T1,d,T2)</del>
	6	0	9	3	9 store(T6,c,nul)	10	9 store(T6,c,nul)	store(T2,c,nul)

**Правила присвоения чисел зависимости операндам (dep):**

- 1) изначально для каждой переменной ее число зависимости равно 0;
- 2) для i-й тетрады, в которой переменной A присваивается некоторое значение, число зависимости переменной A (dep(A)) получает значение i (т.е. значение A теперь зависит от i-й тетрады);
- 3) число зависимости i-й тетрады (dep(i)) принимается равным значению *I+<максимальное\_из\_чисел\_зависимости\_операндов>*.

**Алгоритм** исключения лишних операций на исключении тетрады особого вида s (SAME):

- если i-я тетрада идентична j-й тетраде и  $j < i$ , то i-я тетрада считается лишней в том и только том случае, когда  $dep(i) = dep(j)$ ;
- введем новую тетраду SAME вида:  
s(<операнд1>,<операнд2>,<результат>) для замещения идентичной тетрады. Если тетрада SAME встречается в позиции с номером i, то тетрада i идентична тетраде j и ее можно исключить.

**с) Свёртка объектного кода.**

Производится во время компиляции только для тех операций, для которых операнды уже известны.

для последовательности выражений:

**i=2+1;**  
**j=6\*i+i;**



получим:

**i=3;**  
**j=21;**

Выражение	До оптимизации	Шаг 1	Шаг 2	После оптимизации
i=2+1;	+(2,1,T1) store(i,T1,nul)	<b>C(i,3,nul)</b> store(i,3,nul)	store(i,3,nul)	store(i,3,nul)
j=6*i+i;	*(6,3,T2) +(T2,3,T3) store(j,T3,nul)	<b>C(T2,18,nul)</b> store(T2,18,nul) +(18,3,T3) store(j,T3,nul)	<b>C(T3,21,nul)</b> store(j,T3,nul)	store(j,21,nul)

Чтобы выполнить свёртку объектного кода, создадим таблицу **T**, которая содержит пары (<переменная>,<константа>) для всех переменных, значения которых уже известны, по правилам:

для тетрады присваивания вида **store(b, a, nul)**:

- если **b** — константа, то **a** со значением константы заносится в таблицу **T** (при этом, если в ней уже существует значение для **a**, то это старое значение заменяется на новое);
- если **b** — не константа, то **a** исключается из таблицы **T**, если оно там есть.

Введем тетраду **C** (const) специального вида: **C(i, k, null)**,  
где **i** – имя переменной,  
**k** – значение константы,  
**null** – не используется.

**Алгоритм свёртки объектного кода:**

1. Если операнд является переменной, которая содержится в таблице **T**, то операнд заменяется на соответствующее значение константы.
2. Если тетрада имеет тип **C(i, k, null)**, то операнд **i** заменяется на значение константы **k**.
3. Если все операнды тетрады являются константами, то тетрада может быть свернута.



**d) Перестановка операций:**

Выражение		После перестановки операций
$a=2*b*3*c;$	$\rightarrow$	$a=(2*3)*(b*c);$
$a=(b+c)+(d+c);$	$\rightarrow$	$a=(b+(c+(d+c)));$

Перестановка операций – изменение порядка следования операций для повышения эффективности программы.

**e) Алгебраические преобразования:**

Выражение		После преобразования
$a=b*c+b*d;$	$\rightarrow$	$a=b*(c+d);$
$a*1;$	$\rightarrow$	$a;$
$a*0;$	$\rightarrow$	$0;$
$a+0;$	$\rightarrow$	$a;$

Основаны на известных алгебраических и логических тождествах.

Арифметические преобразования представляют собой изменение порядка следования и выполнение операций на основании известных алгебраических тождеств:  $a*1 \equiv a$ ;  $a*0 \equiv 0$ ;  $a+0 \equiv a$ ;

Арифметические преобразования:

- замена возведения в степень умножением;
- замена целочисленного умножения на константу, кратную 2, выполнением операций сдвига.

**f) Оптимизация вычисления логических выражений:**

$a \parallel b \parallel c \parallel d \rightarrow a$ , если  $a=true$ ;

В случае  $a \parallel f(b) \parallel g(c)$  – логическое выражение сохраняется безизменения, т.к функции могут иметь побочные эффекты.

Операция называется предопределенной для некоторого значения операнда, если ее результат зависит только от этого операнда и остается неизменным (инвариантным) относительно значений других операндов.

Операция логического сложения является предопределенной для логического значения «истина» (true).

Операция логического умножения — предопределена для логического значения «ложь» (false).

### **3.4 Оптимизация передачи параметров в процедуры и функции.** **Оптимизация вызова процедур и функций** (обычно параметры передаются через стек).

#### ***a) Передача параметров через регистры.***

В C++ есть специальный спецификатор хранения `register`, который используется для разрешения хранения параметра в регистре ЦП.

#### ***b) Подстановка кода функции (вместо вызова функции).***

Компиляторы выполняют подстановку не только для макросов, но и для функций с разрешения пользователя (***inline***-функции).

### 3.5 Оптимизация циклов.

#### а) Вынесение инвариантных вычислений из циклов

До оптимизации	После оптимизации
for (i=1; i<=10; i++) a[i]=b*c*a[i];	d=b*c; for (i=1; i<=10; i++) a[i]=d*a[i];

**Инвариант цикла** — утверждение, всегда истинное перед началом выполнения итерации цикла.

#### б) Замена операций с индуктивными переменными (перменными, образующими арифметическую прогрессию).

До оптимизации	После оптимизации
for (i=1; i<=N; i++) a[i]=i*10;	t=10; i=1; while (i<=N) {a[i]=t; t=t+10; i++;}
S=10; for (i=1; i<=N; i++) {r=r+f(S); S=S+10; }	S=10; m=N; while (S<=m) {r=r+f(S); S=S+10; }

#### с) Слияние и развёртывание циклов.

Слияние:

До оптимизации	После оптимизации
for (i=1; i<=N; i++) for (j=1; j<=M; j++) a[i][j]=0;	K=M*N; //(остаётся 1 цикл) for (i=1; i<=K; i++) a[i]=0;

Развёртывание:

До оптимизации	После оптимизации
for (i=1; i<=3; i++) a[i]=i;	a[1]=1; a[2]=2; a[3]=3;

#### 4. Машинно-зависимые методы оптимизации

Машинно-зависимые методы оптимизации ориентированы на конкретную архитектуру целевой вычислительной системы, на которой будет выполняться результирующая программа.

Понятие «архитектура» включает в себя особенности и аппаратных, и программных средств целевой вычислительной системы.

##### **Машинно-зависимые преобразования:**

- распределение регистров процессора;
- оптимизация кода для процессора, допускающая распараллеливание вычислений;
- выбор команд (отображение команд внутреннего представления на систему команд процессора целевой машины).

Простейшие машинно-зависимые методы оптимизации обычно основываются на особенностях системы команд процессоров целевой машины.

Например, для процессоров Intel команда загрузки нулевого значения в регистр аккумулятора `eax`

```
mov eax, 0
```

выполняется дольше и имеет большую длину, чем команда очистки регистра `eax`, выполняемая с помощью операций `xor` (исключающее или):

```
xor eax, eax
```

Если необходимо загрузить значение, равное 1, то порождается пара команд:

```
xor eax, eax  
inc eax
```

для значения -1 – пара команд:

```
xor eax, eax  
dec eax
```

Результирующий код будет более эффективным, если использовать ассемблерные команды увеличения и уменьшения значения регистра на 1 (команды `inc` и `dec`):

- для операции сложения порождается команда `inc` вместо команды `add`, если один из операндов равен **1**;
- для операции сложения порождается команда `dec` вместо команды `add`, если один из операндов равен **-1**.