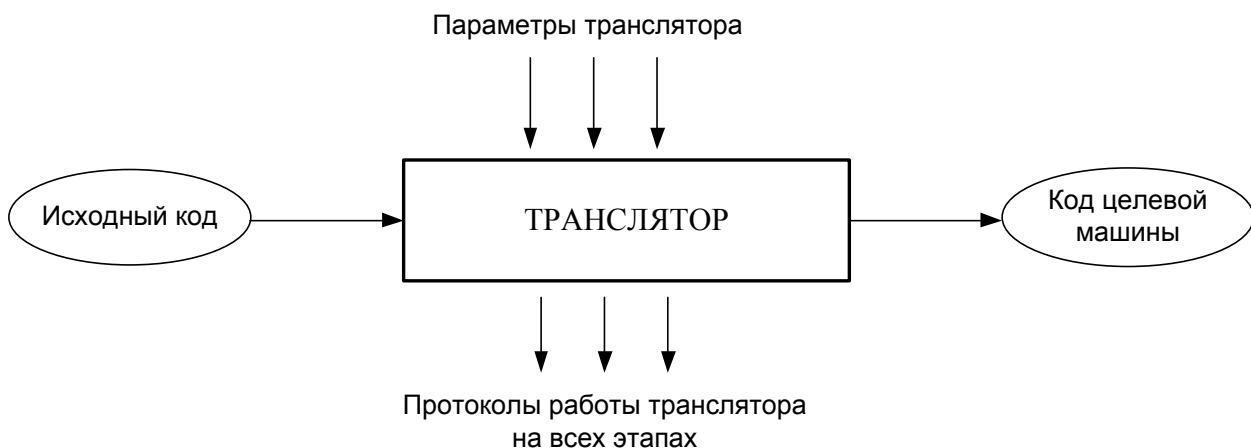


Генерация кода. Jit-компиляторы.

Подходы к разработке трансляторов:

- часть операционной системы;
- для аппаратной платформы (ассемблер);
- реализации для одной программной платформы;
- реализация для одной программной платформы, но для разных процессоров;
- интерпретаторы;
- несколько реализаций для разных платформ;
- кроссплатформенные реализации (Java);
- **компиляторы-интерпретаторы (компиляция + интерпретация);**
- разработка стандарта и стандартизация (Java, C++, C#)

1. Общая схема транслятора:



Исходный код:

- C, C++, Fortran, Java
- LaTeX: текст с макрокомандами
- Компиляторы VHDL
(язык описания устройств на сверх больших интегральных схемах (СБИС))
- SQL
- Естественный язык1 (английский)

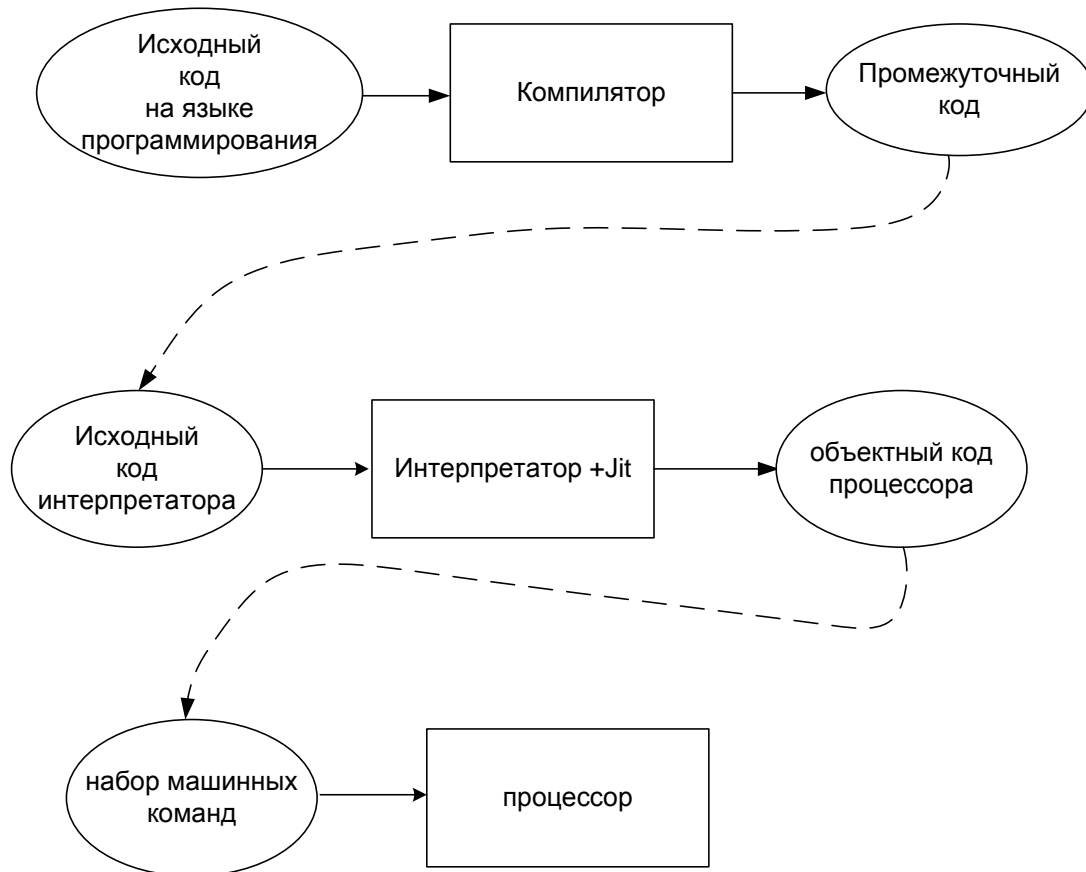
Целевой код:

- ассемблер целевой машины
- .DVI файл
- схема чипа
- план выполнения запроса
- естественный язык2 (русский)

2. Компиляторы-интерпретаторы

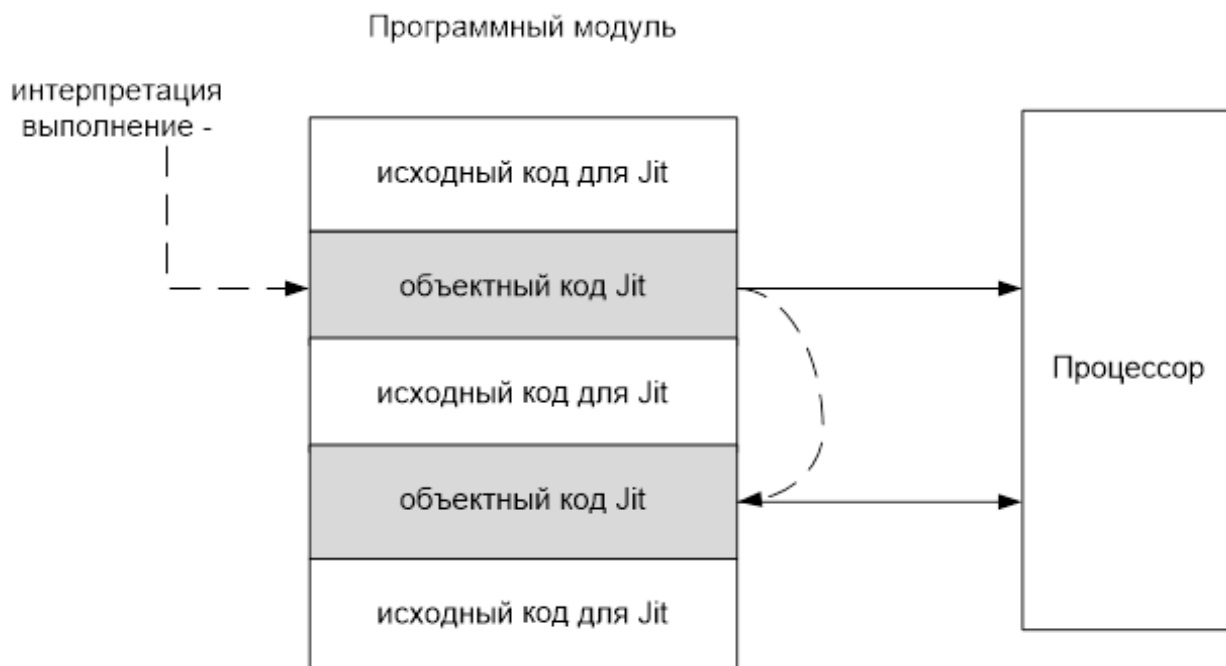
Jit-трансляторы: сначала генерируется промежуточный код, затем он компилируется в объектный код аппаратной платформы.

Jit-трансляторы могут осуществлять частичную трансляцию по мере необходимости.



3. Частичная компиляция

После внесения изменений компилируются только те части программы, которые были модифицированы после предыдущей компиляции.



Шитый код (threaded code) – один из способов реализации промежуточной виртуальной машины при интерпретации языков программирования.

Разновидности шитого кода: **подпрограммная, прямая, косвенная и свернутая**.

Операции промежуточного языка представляются ссылками на подпрограммы, соответствующие этим операциям. Различаются они способом представления ссылок и, соответственно, интерпретатором такой последовательности.

Пример **подпрограммного шитого** кода:

```
call Sub1;  
call Sub2;  
call Sub3;
```

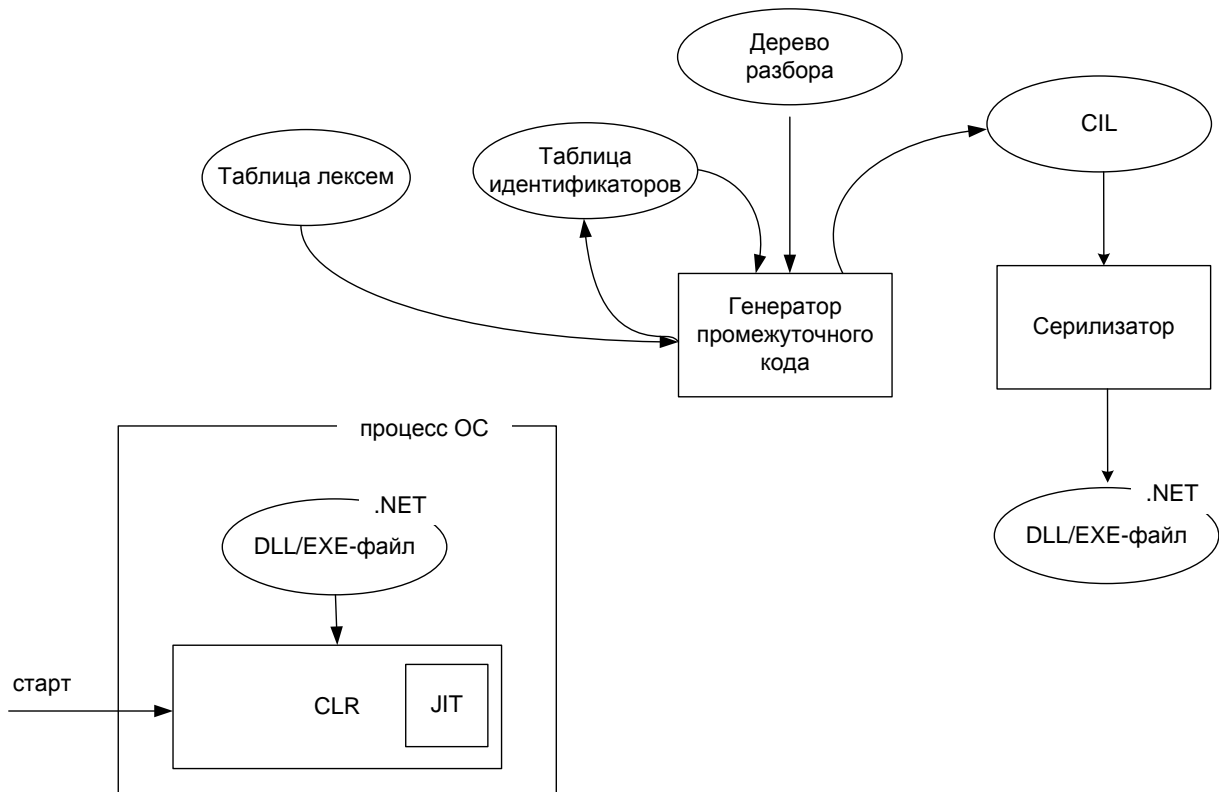
В **прямом шитом коде** определяются последовательность адресов соответствующих подпрограмм (функций). Вызывается интерпретатор управления последовательностью ссылок.

В **косвенном шитом коде** последовательность операций промежуточного языка также состоит из последовательности адресов подпрограмм, разница заключается в том, что управление осуществляется через использование указателей на исполняемые команды, которые хранятся в таблице.

Свернутый шитый код – это набор инструкций по вызову небольших фрагментов низкоуровневого кода, эквивалентных нескольким командам ассемблера или виртуальной машины и в этом случае интерпретатор читает из таблицы код и совершает переход по соответствующему адресу.

Байт-код можно рассматривать как специальный случай свёрнутого шитого кода с адресной таблицей, предоставляющей положение адреса вызываемого кода.

4. Генерация кода языка .NET (C#, VB.NET)

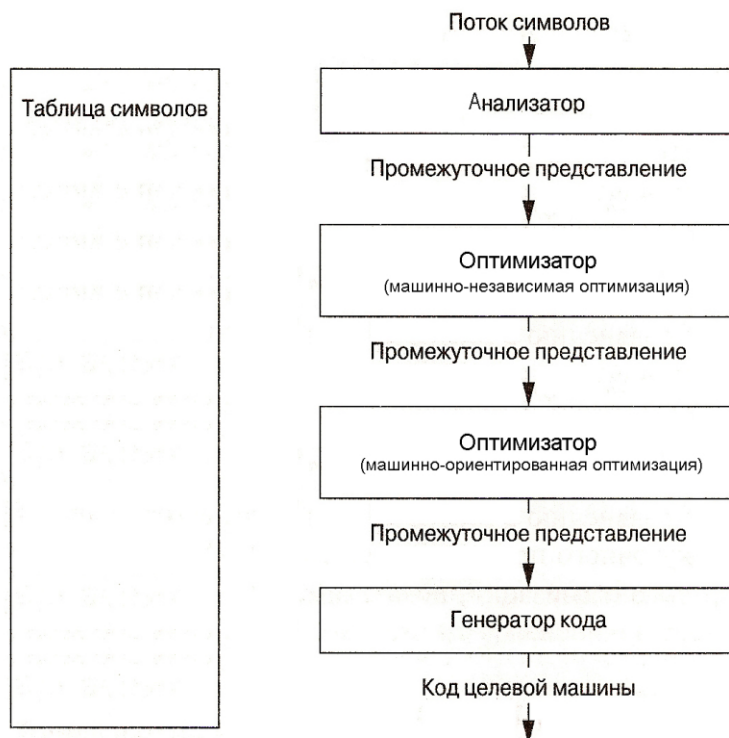


5. Виды компиляции: статическая, динамическая компиляция

- статическая компиляция:
 - AOT (*ahead-of-time*) компиляция;
 - Исходный код → Нативный код;
 - основные действия до выполнения;
- динамическая компиляция:
 - «just-in-time» компиляция;
 - Исходный код → Байт-код → Интерпретатор + JIT;
 - значительная часть работы выполняется во время выполнения.

5.1. Статическая компиляция: оперирует Исходным кодом.

Структура статического оптимизирующего компилятора:



Задачи оптимизирующих компиляторов:

- по коду программы на исходном языке можно построить множество семантически эквивалентных ей программ на целевом языке;
- получить целевую программу, оптимальную по некоторому критерию:
 - скорость выполнения;
 - размер кода (например, для мобильных устройств);
 - минимальное энергопотребление (мобильные устройства,...);
 - и т.п.

5.2.Динамическая компиляция

Профилирование – это сборка информации о коде.

Цель профилировки – исследовать характер поведения приложения во всех его точках.

«Точка» – отдельная машинная команда или конструкция языка высокого уровня (например: функция, цикл или одна строка исходного текста).

Профилировщик определяет:

- общее время исполнения каждой точки программы;
- удельное время исполнения каждой точки программы;
- причины и/или источника конфликтов и пенальти (penalty information);
- количество вызовов той или иной точки программы;
- степени покрытия программы.

6. JIT-компиляция

Java-компилятор `javac` преобразует исходный код на языке Java в переносимый байт-код JVM, который является «инструкциями» для JVM.

Преимущества JIT-компилятора:

- выполняет оптимизацию кода;
- может использовать внутреннее представление, более подходящее для оптимизаций, чем непосредственно байт-код;
- оптимизирует программу под конкретные входные данные (профиль программы собирается «на лету»)
- перекомпилирует программу, если профиль изменился.

Недостатки JIT-компилятора:

- возрастают затраты ресурсов на компиляцию;
- увеличивается время старта;
- ограничен в сложности выполняемых оптимизаций, т.к. не должен задерживать выполнение программы;
- высокая вероятность ошибки, связанной с нарушением семантики пользовательского кода.

Особенности:

- инкапсуляция
- наследование
- полиморфизм
- неопределенное время жизни объектов

Реализуется через вызовы методов для доступа к данным

Особенности систем управляемого исполнения:

- промежуточный код (Byte Code)
 - сокрытие семантики исходного кода
- ленивая загрузка классов
 - неполный граф вызовов во время компиляции
- JIT-компилятор является частью среды
 - возможность перекомпиляции при загрузке новых классов и изменении характера нагрузки
- компиляция происходит параллельно с выполнением
 - требуется учитывать время компиляции
 - за раз компилируется небольшой блок кода

Пример.

Пусть в программе используется метод `Integer.bitCount(int)`, который подсчитывает количество битов в целом числе, установленных в 1 (единицу).

- на многих процессорах есть *готовая инструкция*, которая делает то же самое – это одна машинная команда.
- при использовании *библиотечной функции* `Integer.bitCount()` – JIT подменяет все её вызовы внутренними функциями. Список *intrinsic*-функций (внутренних функций) для каждого процессора свой.

JIT-компилятор собирает информацию об архитектуре, на которой он запущен и на ее основании определяет, что для метода `Integer.bitCount(int)` есть конкретная команда процессора (в JIT есть список *intrinsic*-функций для данного процессора).

```
if (funcName == "Integer.bitCount" && CPU == x86)
    useIntrinsic()
else
    useNormalFunc()
```

Другой пример – Method inlining:

<pre>boolean hasBoobs(String str) { if (str == null) return false else return str.indexOf("Boobs") > 0 } for (i in 1..1000000) { hasBoobs(some string...); }</pre>	→	<pre>for (i in 1..1000000) { if (str == null) return false else // тело метода indexOf() }</pre>
--	---	--

В примере 1 миллион раз вызывается метод `hasBoobs()`, который, в свою очередь, вызывает другой метод – `indexOf()`. Прямо в рантайме код метода `indexOf()` встраивается в `else` часть условия.

Пример. Оптимизация виртуальных вызовов (*virtual call inlining*).

Есть интерфейс `Animal` с методом – `walk()` и две его реализации для классов `Cat` и `Dog`:

```
class Cat implements Animal {  
    void walk() {  
        ...  
    }  
}
```

```
class Dog implements Animal {  
    void walk() {  
        ...  
    }  
}
```

Возможное использование в коде:

```
Animal animal = ...  
animal.walk()
```

Вопрос - метод какого класса надо вызвать `Cat` или `Dog`?

В рантайме JIT-компилятор точно знает экземпляр какого класса в переменной `animal`. Этот объект уже создан и работает. Например: в переменной `animal` хранится экземпляр класса `Cat`, следовательно можно не делать вызов виртуальной функции, а прямо вставить вызов конкретного метода `Cat.walk()`.

Многоуровневый JIT:

- 1-й уровень: быстрый — интерпретатор — выполняет простые оптимизации, после которых сразу же начинает выполнять байт-код и собирать профиль;
- 2-й уровень: оптимизирует только «горячие» места, выполняет более сложные оптимизации по профилю, при этом, собственное внутреннее представление, может быть спекулятивным;
- обеспечивает переключение между уровнями при изменении профиля.

Особенности JIT

Спекулятивный JIT:

- специализирует код для данных, полученных при профилировании (прежде всего типы объектов);
- для обработки остальных случаев предусмотрены проверки, возвращающие выполнение на предыдущий уровень JIT;

Пример.

- целочисленная арифметика и регистры: для *предположительно* целых типов (т.к. работает быстро для целых);
- откат на предыдущий уровень происходит только при переполнении (работает медленно, но и для других, в том числе неатомарных типов).

Пример спекулятивной оптимизации:

<pre>if (cond) foo(); else bar();</pre>	Профилируем →	<pre>if (cond) foo(); // 0 раз else bar(); // 10000 раз</pre>
---	---------------	---

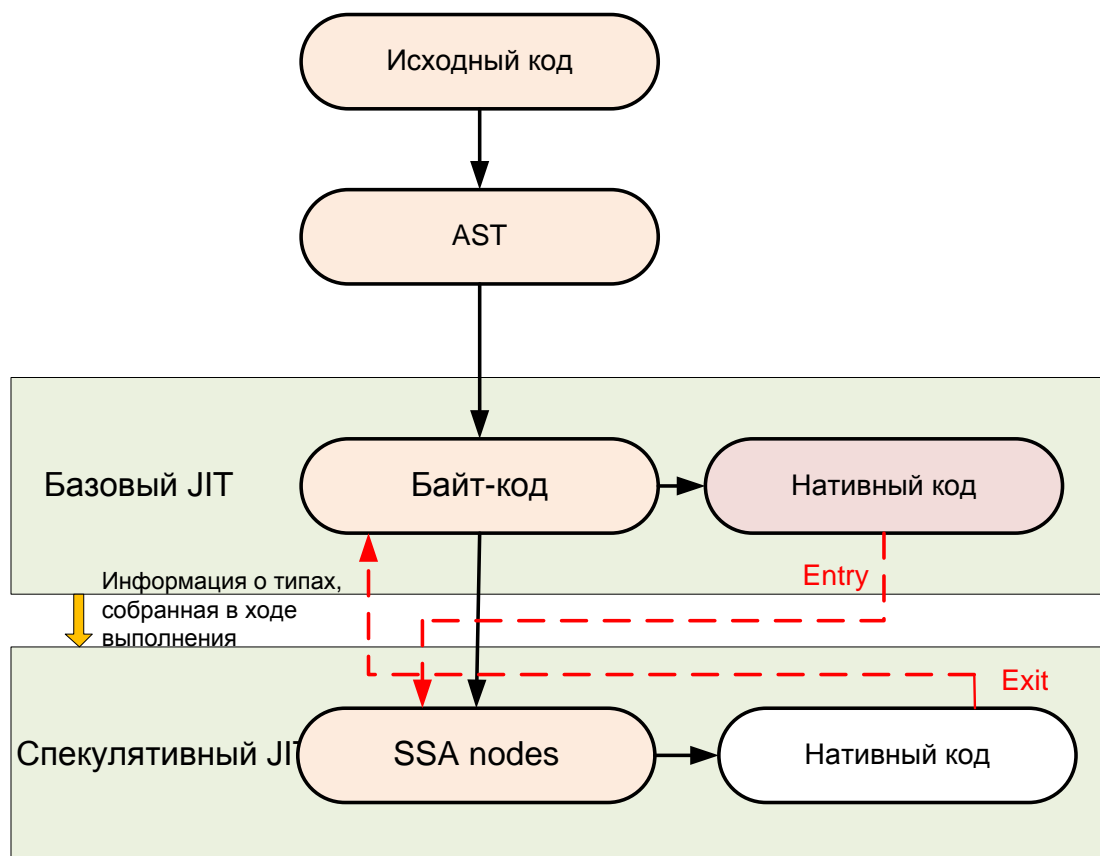
Оказалось: `foo()` никогда не исполнялся!

Вывод (*спекулятивный*): не будем компилировать `foo`:

<pre>if (cond) foo(); else bar();</pre>	→	<pre>if (cond) <uncommon_trap> bar();</pre>
---	---	---

Чтобы избежать ловушек: (**Uncommon Traps**):

```
// Если условие случится, исполнение продолжить в интерпретаторе:
if (cond)
<uncommon_trap>
bar();
```



Переключение между уровнями JIT:

- переключение возможно между командами байт-кода;
- каждая проверка удачной спекуляции содержит свой код отката;
- замена исполняемого кода во время выполнения в стеке.

Пример при переполнении для целочисленной арифметики:

- записать в стек все вычисленные значения как **float**;
- перейти в код базового JIT в место, соответствующее операции, вызвавшей переполнение.

Свойства промежуточного представления высокого уровня:

- сохранение семантики исходного кода
 - контексты блоков и функций;
 - типы данных;
 - общие подвыражения;
 - циклы в явном виде;
- сохранение метаданных;
- упрощение оптимизации;
- граф управления (**control flow graph**):
 - узлы – блоки, участки кода без переходов;
 - дуги – переходы между блоками;
 - возможно параллельное исполнение блоков;
 - хранение дополнительных данных в узлах и дугах;
 - описание блоков.

Температура кода:

- мертвый код: не исполняется никогда;
- холодный код: исполняется редко, не оказывает влияния на скорость работы системы;
- теплый код: исполняется регулярно, влияет на скорость работы системы;
- горячий код: оказывает доминирующее влияние на скорость работы системы.

Удаление «мертвого» кода

Одной из задач оптимизирующего компилятора является удаление «мертвого» кода – кода, не влияющего на результат выполнения программы.

Сборка мусора

Объекты размещаются в куче. По истечении времени жизни таких объектов из них создается мусор. Для уборки мусора запускается сборщик мусора.