Лекция 9.

Интерпретаторы и компиляторы

Абстрактные машины и реальные процессоры

Язык программирования – машинный код некоторого абстрактного компьютера, отличающегося от реальных процессоров.

Цель трансляции – реализация возможности выполнения реальным компьютером кода, написанного для абстрактной машины.


Трансляция
Компиляция
Интерпретация
Исходный текст программы
Компилятор, компоновщик
Программа на машинном языке
Выполнение программы
Оператор на исходном языке
Интерпре-татор
Команды на машинном языке
Выполнение команд

Базовые схемы (без ввода данных)

- **Компилятор** транслирует переданную ему программу в целевую форму – машинный код, который уже может быть непосредственно выполнен на компьютере.
- **Интерпретатор** непосредственно выполняет программу в исходном коде. Он должен определять эффект выполнения каждой конструкции языка программирования.

Базовые схемы (без ввода данных)

Интерпретатор поднимает машину до уровня пользовательской программы.

Компилятор опускает пользовательскую программу до уровня машинного языка.

Интерпретация и компиляция

```
x := 1;
y := x;
```

Интерпретатор должен хранить таблицу всех используемых переменных и связанных с ними значений.

Переменная	Значение
x	1
y	1

Компилятор генерирует машинный код, используя команды компьютера и адреса памяти.

	x			y			
	1			1			

Интерпретация: пример

Простой язык программирования:

- каждая строка представляет собой *выражение*,
- каждое выражение состоит из *команды* (оператора)
- и любого количества *значений* (операндов), которыми оперирует команда.

```
set a 1
set b 2
add a b c
print c
```

Интерпретация: пример

Простой язык программирования:

- каждая строка представляет собой *выражение*,
- каждое выражение состоит из *команды* (оператора)
- и любого количества *значений* (операндов), которыми оперирует команда.

```
set a 1
set b 2
add a b c
print c
```

Напишем интерпретатор – программу, которая считывает каждое «выражение», находит оператор и операнды, а затем что-то с ними делает, в зависимости от конкретного оператора.

Интерпретация: пример (PHP)

```
<?php
$lines = file($argv[1]);

$linenr = 0;
foreach ($lines as $line) {
    $linenr++;
    $operands = explode(" ", trim($line));
    $command = array_shift($operands);

    switch ($command) {
      case 'set' :
            $vars[$operands[0]] = $operands[1];   // $vars['a']=1, $vars['b']=2
            break;
    case 'add' :
            $vars[$operands[2]] = $vars[$operands[0]] + $vars[$operands[1]];
                    // $vars['c'] = $vars['a']+$vars['b'];
            break;
    case 'print' :
            print $vars[$operands[0]] . "\n";   // print $vars['c'] . "\n";
            break;
    default :
            throw new Exception(sprintf("Unknown command in line %s\n", $linenr));
    }
}
```

Интерпретация: пример (PHP)

Написать интерпретатор легко, но выполняться программа будет очень медленно.

Придется обрабатывать каждую строку и проверять:

- Какой оператор нужно выполнить?
- Это правильный оператор?
- Есть ли у него нужное количество операндов?
- Какой тип у этих операндов?
- …

Базовые схемы с вводом данных

У интерпретатора два источника ввода – исходная программа и входные данные.

Компилятору на вход подается только программа.

Интерпретация и компиляция: что лучше?

Интерпретация и компиляция: что лучше?

- Обработка информации в том виде, как она есть
- Предварительное преобразование к более удобной форме

Интерпретация и компиляция: что лучше?

- Обработка информации в том виде, как она есть
- Предварительное преобразование к более удобной форме

Интерпретация и компиляция: что лучше?

- Скорость выполнения
- Удобство и скорость разработки
- Надежность программы

Интерпретация и компиляция: что лучше?

Скорость выполнения – лучше компиляторы.

- Машинный код выполняется быстрее.
- При создании машинного кода компилятор может применять оптимизацию.
- Интерпретация кода требует при выполнении каждого оператора его предварительной обработки.

Интерпретация и компиляция: что лучше?

Скорость выполнения – лучше компиляторы.

- Машинный код выполняется быстрее.
- При создании машинного кода компилятор может применять оптимизацию.
- Интерпретация кода требует при выполнении каждого оператора его предварительной обработки.

Удобство и скорость разработки – лучше интерпретаторы.

- При интерпретации выполнение начинается сразу.
- REPL-среда (read-evaluate-print loop). Очень удобна при изучении нового языка, так как предоставляет пользователю быструю обратную связь.

Интерпретация и компиляция: что лучше?

Скорость выполнения – лучше компиляторы.

- Машинный код выполняется быстрее.
- При создании машинного кода компилятор может применять оптимизацию.
- Интерпретация кода требует при выполнении каждого оператора его предварительной обработки.

Удобство и скорость разработки – лучше интерпретаторы.

- При интерпретации выполнение начинается сразу.
- REPL-среда (read-evaluate-print loop). Очень удобна при изучении нового языка, так как предоставляет пользователю быструю обратную связь.

Надежность программы – лучше компиляторы.

- В процессе компиляции делаются различные проверки

Процесс компиляции

Написание компилятора – одна из труднейших задач в информатике. Получаемый код зависит от платформы.

Процесс компиляции

Написание компилятора – одна из труднейших задач в информатике. Получаемый код зависит от платформы.

1. **Лексический анализ.** Последовательность символов исходного файла преобразуется в последовательность лексем (токенов).

Процесс компиляции

Написание компилятора – одна из труднейших задач в информатике. Получаемый код зависит от платформы.

1. **Лексический анализ.** Последовательность символов исходного файла преобразуется в последовательность лексем (токенов).
2. **Синтаксический анализ.** Последовательность лексем преобразуется в дерево разбора.

Процесс компиляции

Написание компилятора – одна из труднейших задач в информатике. Получаемый код зависит от платформы.

1. **Лексический анализ.** Последовательность символов исходного файла преобразуется в последовательность лексем (токенов).
2. **Синтаксический анализ.** Последовательность лексем преобразуется в дерево разбора.
3. **Семантический анализ.** Дерево разбора обрабатывается для установления его семантики (смысла): привязка идентификаторов к определениям, проверка совместимости, определение типов выражений и т.д. Результат – промежуточное представление/код.

Процесс компиляции

Написание компилятора – одна из труднейших задач в информатике. Получаемый код зависит от платформы.

1. **Лексический анализ.** Последовательность символов исходного файла преобразуется в последовательность лексем (токенов).
2. **Синтаксический анализ.** Последовательность лексем преобразуется в дерево разбора.
3. **Семантический анализ.** Дерево разбора обрабатывается для установления его семантики (смысла): привязка идентификаторов к определениям, проверка совместимости, определение типов выражений и т.д. Результат – промежуточное представление/код.
4. **Оптимизация.** Удаление лишних конструкций и упрощение кода с сохранением его смысла.
5. **Генерация машинного кода.**

Нахождение ошибок в программе

```
Var
   a: integer;
   b: string;
begin
   a:=1;
   b;=”a”;
end.
```

Нахождение ошибок в программе

```
Var
   a: integer;
   b: string;
begin
   a:=1;
   b;=”a”;
end.
```

Здесь компилятор обнаружит ошибку на этапе лексического анализа.

Нахождение ошибок в программе

```
Var
   a: integer;
   b: string;
begin
   a:=1;
   b:=”a”;
   begin
end.
```

Нахождение ошибок в программе

```
Var
   a: integer;
   b: string;
begin
   a:=1;
   b:=”a”;
   begin
end.
```

Это синтаксическая ошибка.

Нахождение ошибок в программе

```
Var
   a: integer;
   b: string;
begin
   a:=1;
   b:=”a”;
   a:=b;
end.
```

Нахождение ошибок в программе

```
Var
   a: integer;
   b: string;
begin
   a:=1;
   b:=”a”;
   a:=b;
end.
```

Здесь компилятор обнаружит ошибку на этапе семантического анализа.

Нахождение ошибок в программе

```
Var
  a: integer;
  b: string;
begin
  a:=1;
  b:=”a”;
  for a:=1 to 100000 do
    begin
    end;
end.
```

Нахождение ошибок в программе

```
Var
  a: integer;
  b: string;
begin
  a:=1;
  b:=”a”;
  for a:=1 to 100000 do
    begin
    end;
end.
```

Компилятор при оптимизации уберет этот пустой цикл.

Как можно увеличить скорость выполнения приложения на интерпретируемом языке программирования?

Транскомпиляция (source-to-source компиляция)

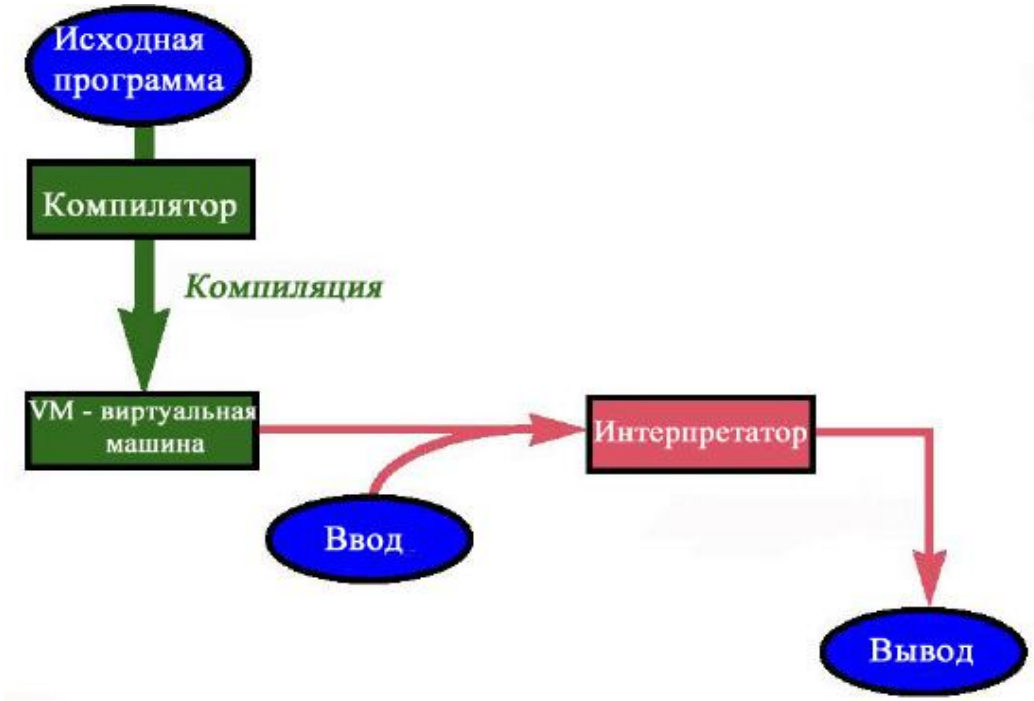
Исходный текст программы на одном языке автоматически переводится в программу на другом языке, для которого уже существует компилятор.

Система HipHop в Facebook – преобразование языка PHP в C++.

Транскомпиляция повышает производительность приложения, так как позволяет напрямую выполнять двоичный код.

Обратная сторона: прежде чем выполнить код, нам сначала нужно провести транскомпиляцию, а затем настоящую компиляцию. Но это нужно делать только во время разработки.

Комбинирование компиляции и интерпретации



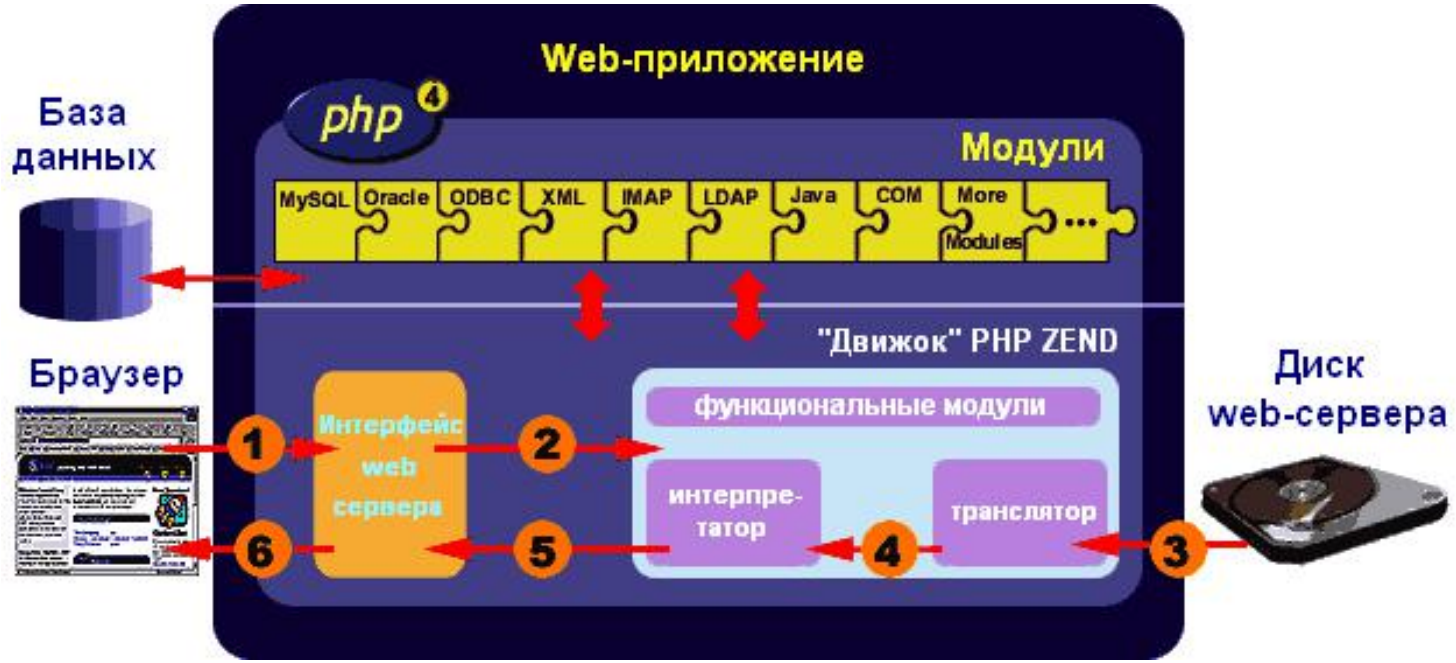
Компилятор создает байт-код на промежуточном языке, понимаемом некоторой виртуальной машиной (интерпретатором байт-кода).

Преимущества:

- **Кроссплатформенность**, так как VM-код не зависит от специфики процессоров;
- **Повышение эффективности**, так как создаваемый промежуточный код легко интерпретируется.

Комбинирование компиляции и интерпретации

PHP, Ruby, Python, Java.



1. Считать скрипт (PHP) целиком в память.
2. Целиком преобразовать/компилировать скрипт в байт-код.
3. Выполнить байт-код посредством интерпретатора (PHP).

Вместо шагов 1-2 можно кешировать байт-код.

Байт-код

Программа на PHP:

```
$a = 3;
echo "hello world";
print $a + 1;
```

Байт-код для этой программы:

```
line     #* E I O op    fetch   ext  return  operands
-------------------------------------------------------
   2     0  E >   ASSIGN                     !0, 3
   3     1        ECHO                       'hello+world'~
   4     2        ADD                ~1      !0, 1
         3        PRINT              ~2      ~1
         4        FREE               ~2
         5      > RETURN
```

- Каждая строка – команда с набором операторов.
- Вместо имен переменных – числа.
- Выполняется быстрее, чем классическая интерпретация

Получение байт-кода из текста программы

1. Лексический анализ (токенизация).

Представление исходного кода в виде массива токенов (без учета их значений).

Исходный код

```
$a = 3;
echo "hello world";
print $a + 1;
```

Получение байт-кода из текста программы

1. Лексический анализ (токенизация).

Представление исходного кода в виде массива токенов (без учета их значений).

Исходный код

```
$a = 3;
echo "hello world";
print $a + 1;
```

Токены

```
T_VARIABLE $a
T_WHITESPACE
=
T_WHITESPACE
T_LNUMBER 3 ;
T_WHITESPACE
T_ECHO echo
T_WHITESPACE
T_CONSTANT_ENCAPSED_STRING "hello world"
;
```

Получение байт-кода из текста программы

2. Синтаксический анализ (парсинг).

Проверяем токены на соответствие правилам, которые определяют язык программирования.

Получение байт-кода из текста программы

2. Синтаксический анализ (парсинг).

Проверяем токены на соответствие правилам, которые определяют язык программирования.

Например, присваивание может состоять из любого T_VARIABLE, после которого идёт символ =, а затем T_LNUMBER, T_VARIABLE или T_CONSTANT_ENCAPSED_STRING.
Тогда $a = 1, или $a = $b, или $a = 'foobar' являются корректными выражениями, а 1 = $a – недопустимо.

После токенизации и парсинга либо сразу генерируется байт-код (так было в PHP до версии 5.6), либо сначала строится **абстрактное синтаксическое дерево**.

Получение байт-кода из текста программы

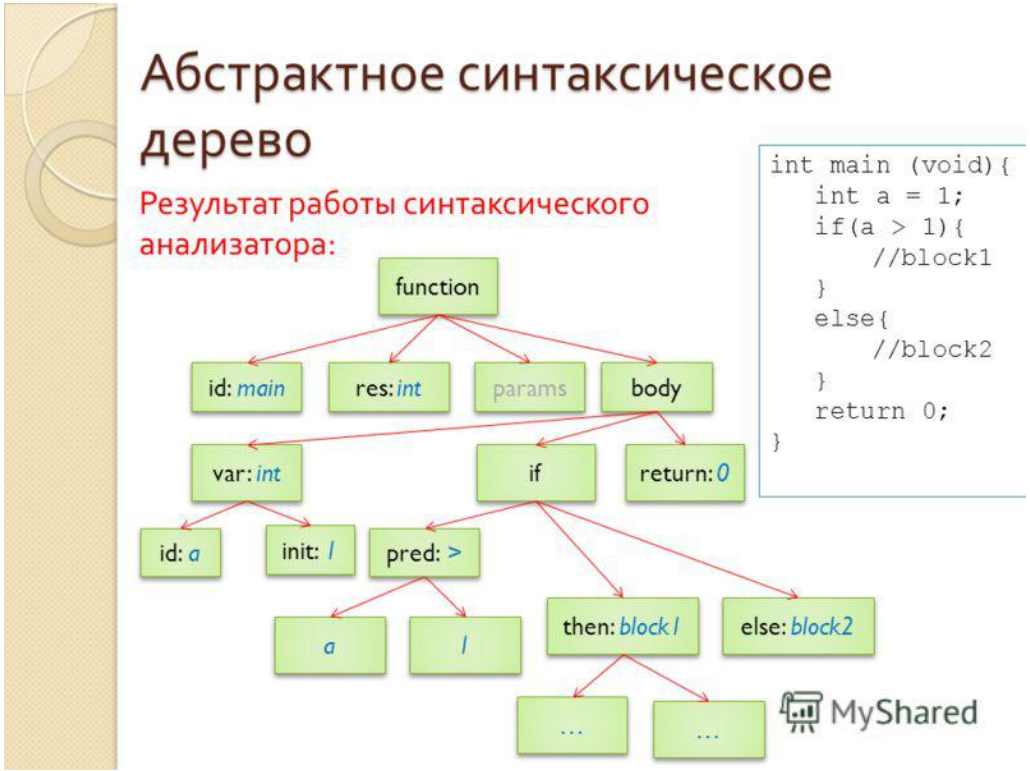
3. Абстрактное синтаксическое дерево

Древовидная структура, в которой абстрактно представлена вся программа.

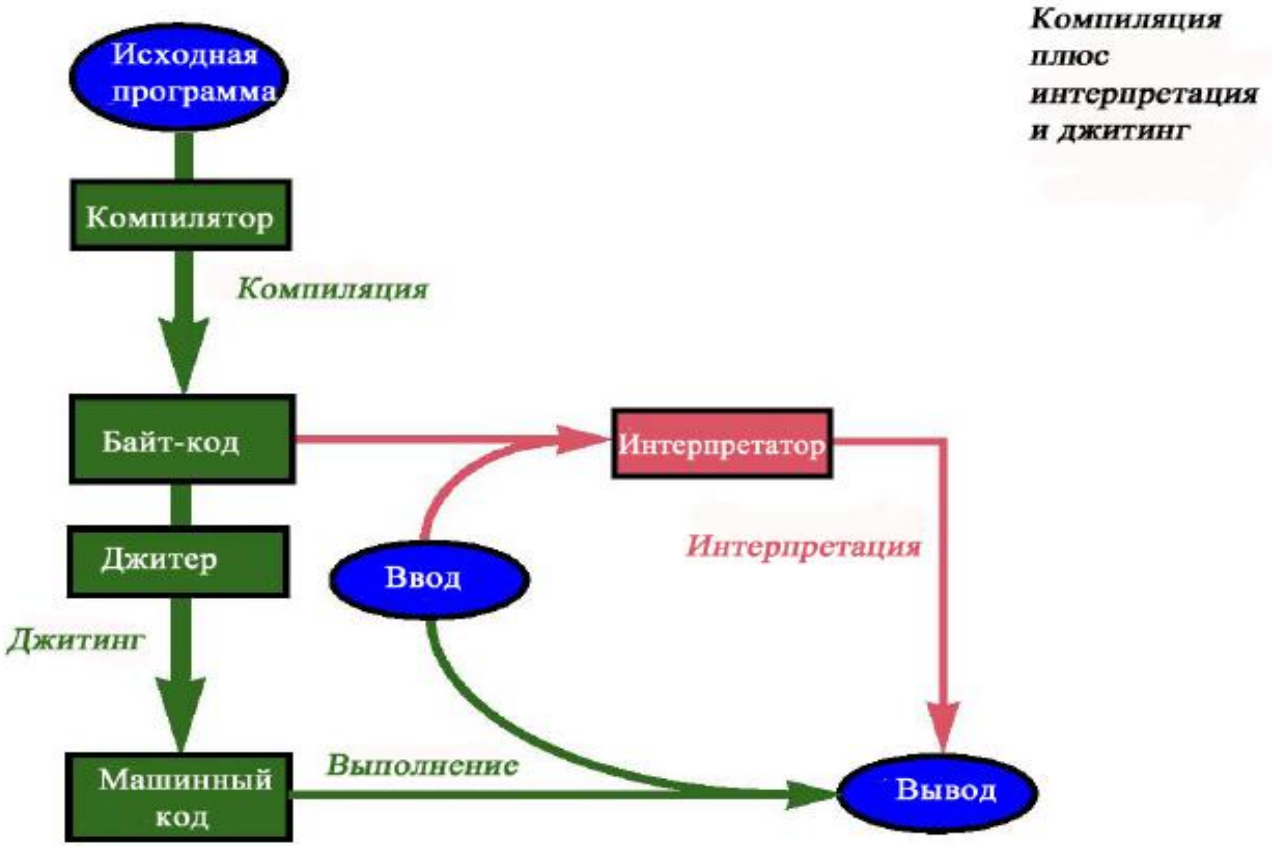
Преимущества использования AST:

- Упрощает генерирование байт-кода.
- Можно оптимизировать код. Статический анализ.

PHP 7.0



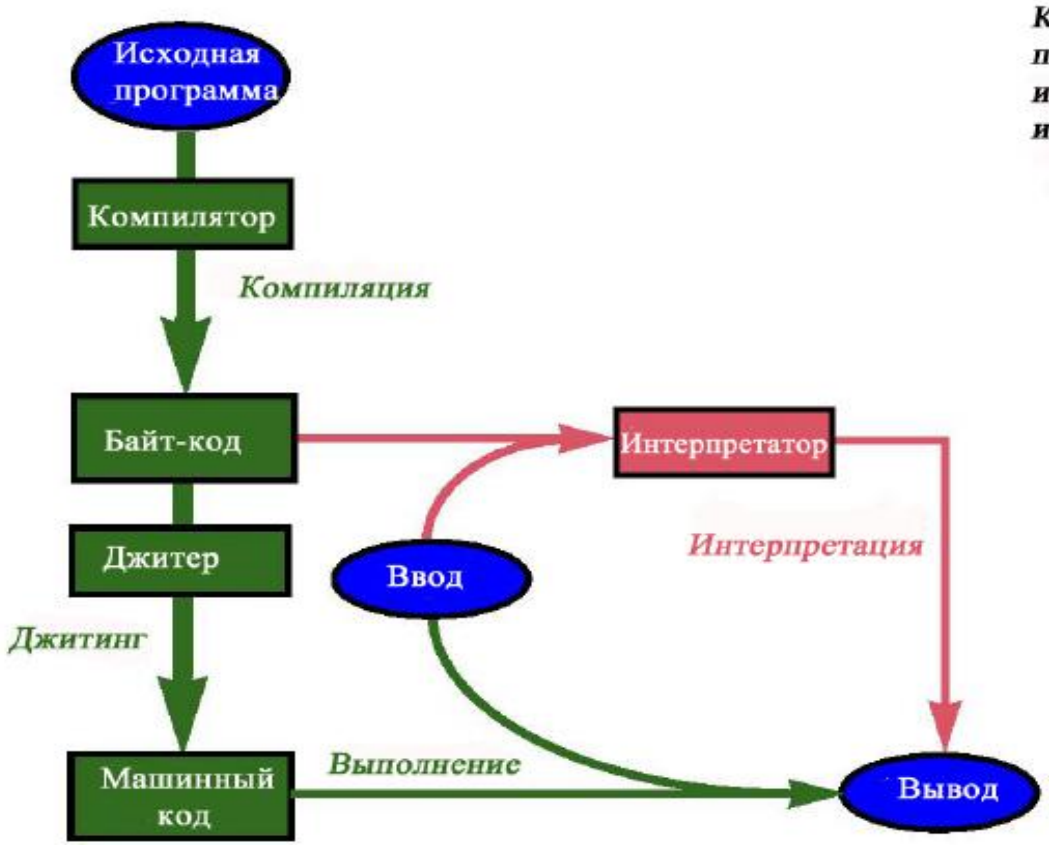
Комбинирование компиляции и интерпретации



JIT-компиляторы (just-in-time, точно в срок).

- Программа может стартовать немедленно, не нужно ждать завершения компилирования.
- В двоичный код преобразуются только самые эффективные части кода, так что процесс компилирования автоматизируется и ускоряется.

Комбинирование компиляции и интерпретации



Преимущества над обычной компиляцией:

- Более компактный код.
- Сокращение времени компиляции.

Недостаток:

- Компиляция становится частью процесса выполнения програмы.