

Лекция 7.

# **Интерпретаторы и компиляторы**

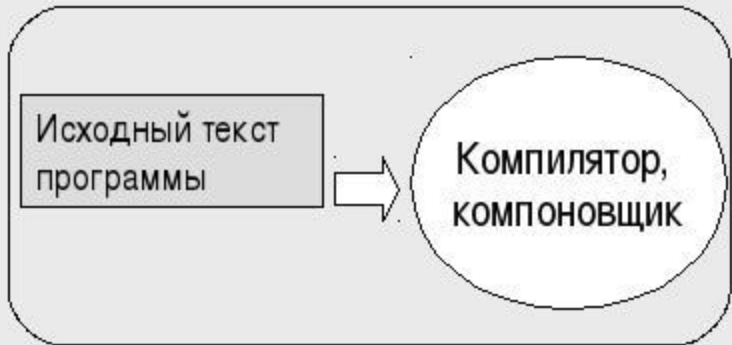
## Абстрактные машины и реальные процессоры

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

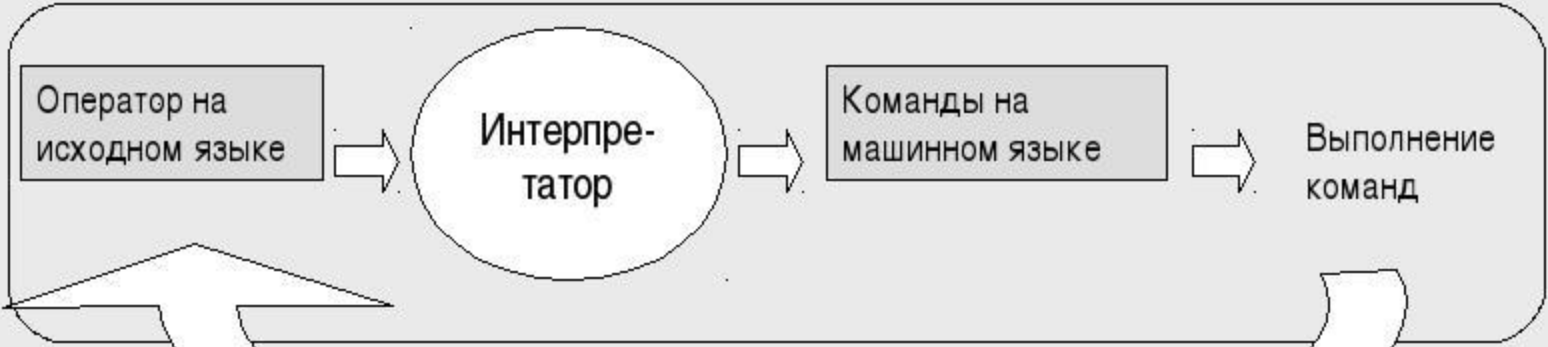
Цель трансляции – реализация возможности выполнения реальным компьютером кода, написанного для абстрактной машины.

# Трансляция

## Компиляция



## Интерпретация



## 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:

- Строка представляет собой *выражение*.
- Выражение состоит из:
  - *команды* (оператора)
  - *значений* (операндов), которыми оперирует команда.
- Допустимые команды: set, add, print.

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

---

Опишем синтаксис этого языка с помощью БНФ.

# Лексика, синтаксис и семантика

Для описания ЯП нужно определить:

- **алфавит** – набор символов, используемый при записи программ;
- **синтаксис** – правила, задающие внешний вид программы и определяющие допустимые тексты в языке;
- **семантику** – правила, определяющие действия, которые выполнит компьютер под управлением программы.

# Формулы БНФ

- **Терминальные символы** — это отдельные символы или их последовательности, имеющие конкретные известные значения и являющиеся неразрывным целым.
- **Нетерминальные символы (метапеременные)** – элементы языка, не имеющие заранее известного значения, которые раскрываются правилам грамматики.
- ::= "есть по определению"
- | "или"
- {a} повтор a 0 или более раз
- [a] a входит 0 или 1 раз

**<идентификатор> ::= выражение**

- **идентификатор** – имя метапеременной
- **выражение** – комбинация терминальных символов и метапеременных.

# БНФ для нашего ЯП

Простейший случай

`<expression> ::= <set> | <add> | <print>`

`<set> ::= 'set' <var> <integer>`

`<add> ::= 'add' <var> <var> <var>`

`<print> ::= 'print' <var>`

`<var> ::= <char> { <char> | <digit> }`

`<char> ::= 'a' | 'b' | ... | 'z' | 'A' | ... | 'Z'`

`<digit> ::= '0' | '1' | '2' | ... | '9'`

`<integer> ::= <digit> { <digit> }`

---

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

Усложним синтаксис языка:

$\langle \text{expression} \rangle ::= \langle \text{set} \rangle \mid \langle \text{add} \rangle \mid \langle \text{print} \rangle$

$\langle \text{set} \rangle ::= \text{'set'} \langle \text{var} \rangle \{ \langle \text{integer} \rangle \mid \langle \text{var} \rangle \}$

$\langle \text{add} \rangle ::= \text{'add'} \{ \langle \text{var} \rangle \mid \langle \text{integer} \rangle \} \{ \langle \text{var} \rangle \mid \langle \text{integer} \rangle \} \langle \text{var} \rangle$

$\langle \text{print} \rangle ::= \text{'print'} \{ \langle \text{var} \rangle \mid \langle \text{integer} \rangle \}$

$\langle \text{var} \rangle ::= \langle \text{char} \rangle \{ \langle \text{char} \rangle \mid \langle \text{digit} \rangle \}$

$\langle \text{char} \rangle ::= \text{'a'} \mid \text{'b'} \mid \dots \mid \text{'z'} \mid \text{'A'} \mid \dots \mid \text{'Z'}$

$\langle \text{digit} \rangle ::= \text{'0'} \mid \text{'1'} \mid \text{'2'} \mid \dots \mid \text{'9'}$

$\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \}$

## Интерпретация: пример

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



Лучше день потерять,  
потом за 5 минут  
долететь...

## 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.
```

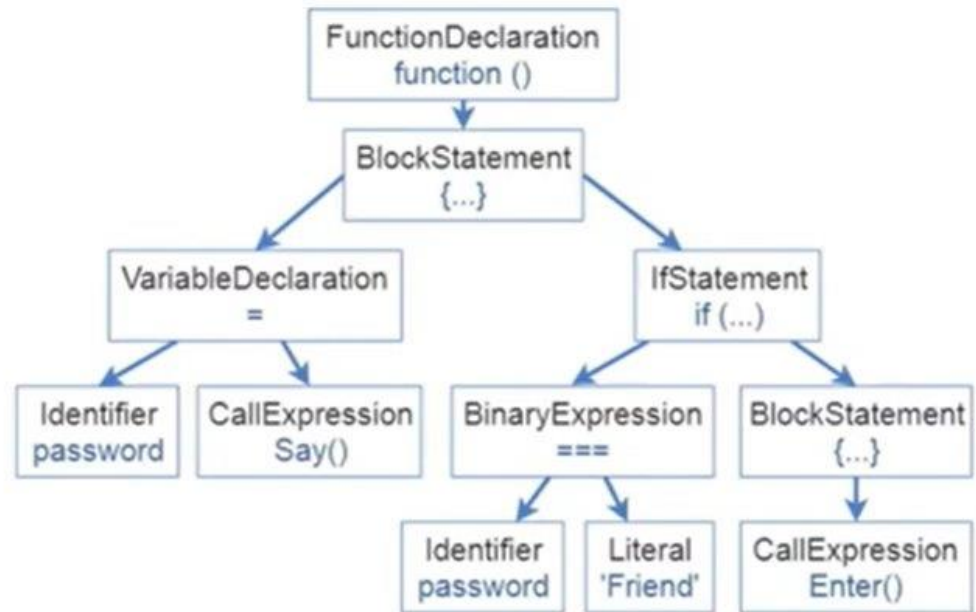
Компилятор при оптимизации уберет этот пустой цикл.

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.

```
function DoorsCheck() {  
    const password = Say();  
    if (password === 'Friend') {  
        Enter();  
    }  
}
```



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



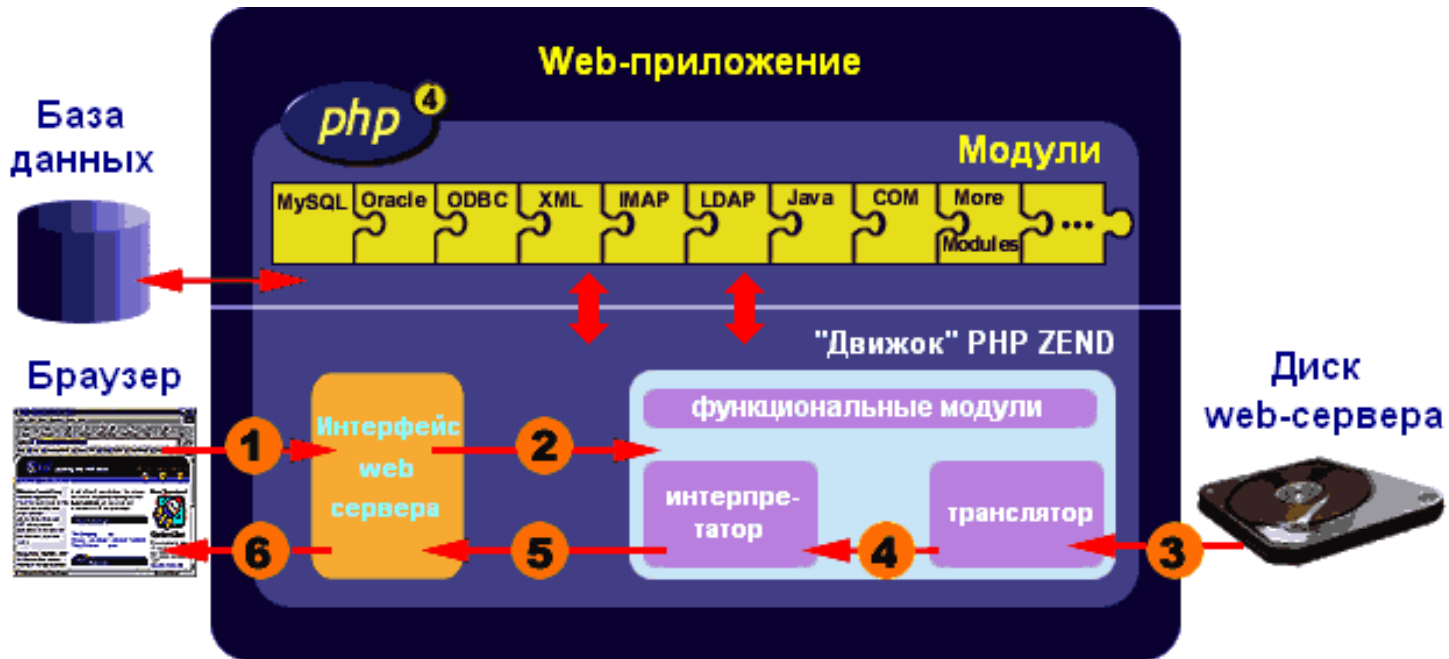
Компилятор создает байт-код на промежуточном языке, понимаемом некоторой виртуальной машиной (интерпретатором байт-кода).

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 кэшировать 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 (значение лексемы и ее тип).

### 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

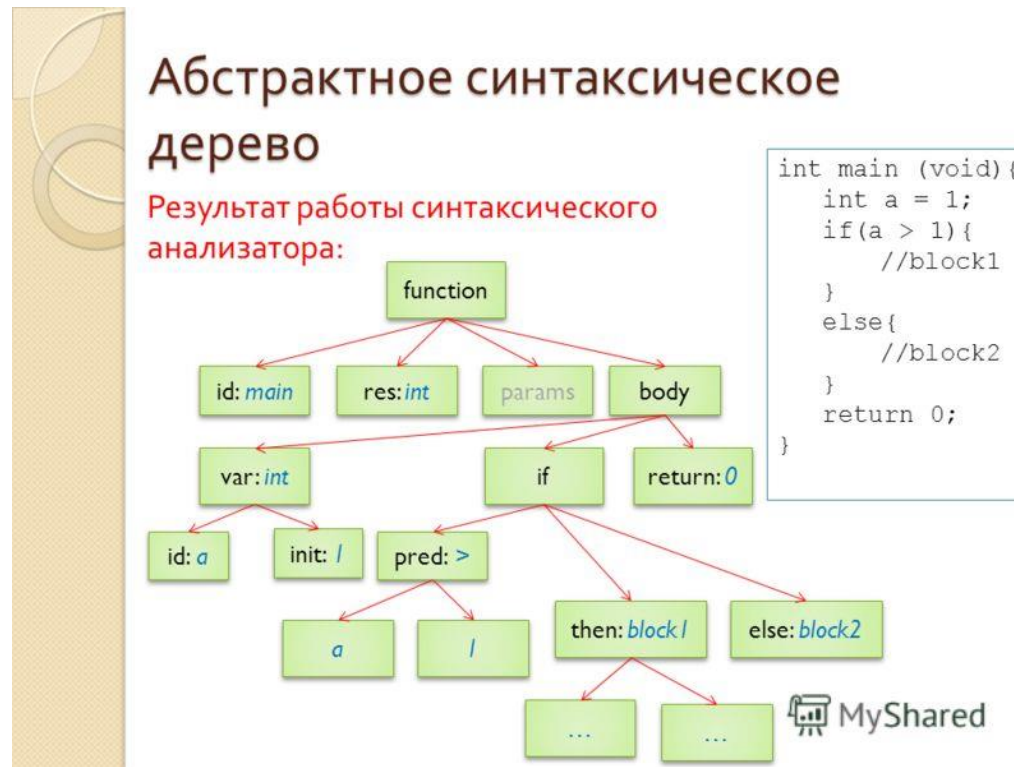
## 3. Абстрактное синтаксическое дерево

Древовидная структура, в которой абстрактно представлена вся программа.

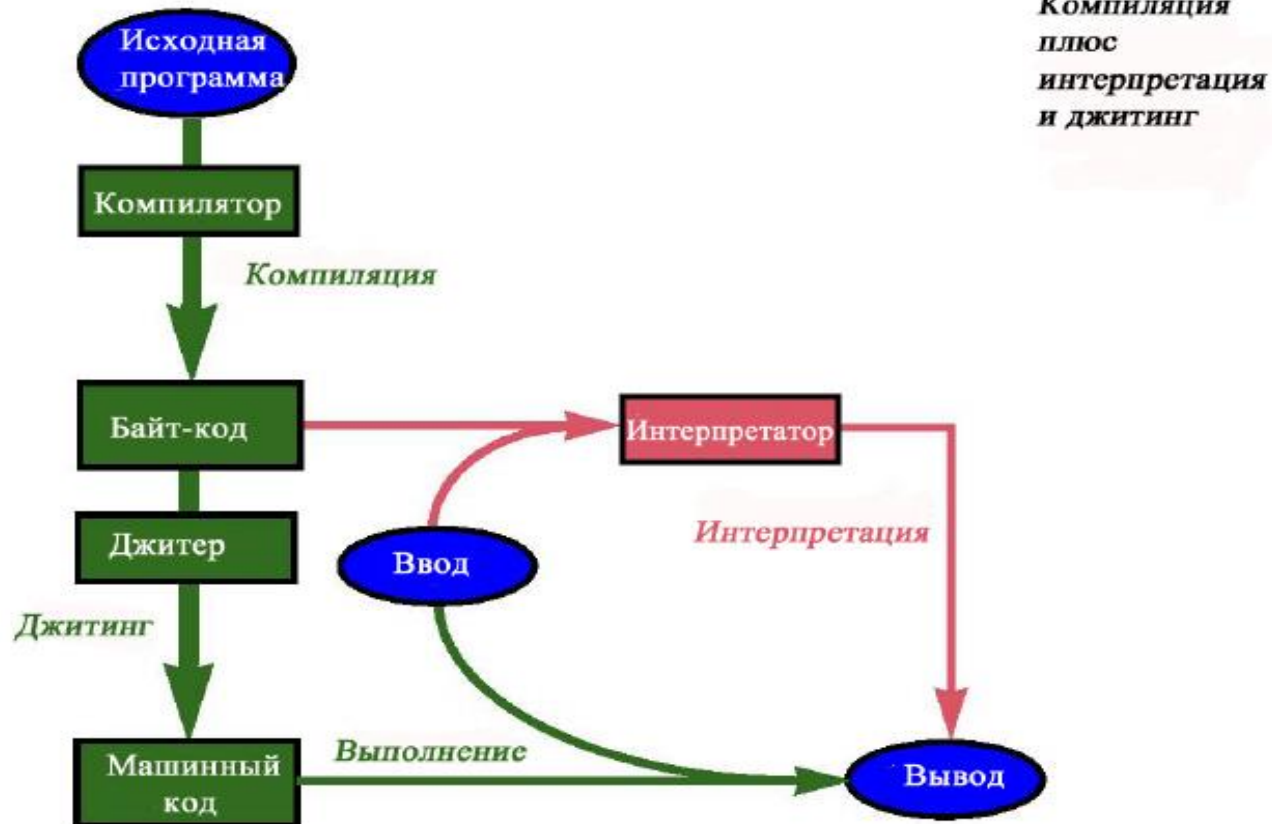
Преимущества использования AST:

- 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



Компиляция  
плюс  
интерпретация  
и джитинг

## Преимущества над обычной компиляцией:

- Более компактный код.
- Сокращение времени компиляции.

## Недостаток:

- Компиляция становится частью процесса выполнения программы.