# Языки программирования. Метапрограммирование

Алексей Островский

Физико-технический учебно-научный центр НАН Украины

12 декабря 2014 г.

# Языки программирования

## Определение

**Язык программирования** — формальный язык для записи инструкций, выполняемых исполнителем (чаще всего — ЭВМ).

**Составляющие языка:**

- Синтаксис — набор правил, определяющих последовательности символов, составляющих допустимую программу или ее фрагмент.
- Семантика — правила, определяющие значение различных конструкций языка.
- Система выполнения и стандартная библиотека — определяют функциональность, доступную для всех имплементаций языка без подключения внешних модулей.

# Классификация языков программирования

**Способы классификации ЯП:**

- по способу представления инструкций — текстовые, графические, смешанные;
- по конкретизации инструкций — императивные (инструкции определяют последовательность действий), декларативные (инструкции определяют конечный результат, но не способ его достижения);
- по поддерживаемым парадигмам программирования;
- по выразительной силе — полные и неполные по Тьюрингу;
- по области применения — общего назначения и предметно-ориентированные;
- по семантическим характеристикам, напр., особенностям системы типов данных.

# Синтаксис ЯП

**Синтаксис = форма ЯП.**

**Описание синтаксиса:** формы Бэкуса — Наура (БНФ).

**Роль в обработке программы:**

- лексический анализ — группирование символов программы в лексемы (напр., числа, строки, ключевые слова);
- синтаксический анализ — построение дерева разбора на основе лексем.

**Синтаксис БНФ:**[1]

```
<выражение> ::= <альтернатива 1> | <альтернатива 2> | … | <альтернатива n>
```

[1] В БНФ может использоваться синтаксис регулярных выражений (+ = повтор 1 или более раз, [ ] — выбор из нескольких вариантов и т. п.).

# Пример БНФ

**БНФ для списков:**

```
1 expression ::= element | list
2 element    ::= number | variable
3 number     ::= [+-]?['0'-'9']+
4 variable   ::= ['A'-'Z''a'-z']'\w'*
5 list       ::= '[' (expression',')*expression ']' | '[]'
```

**Допустимые списки:**

- `[]`
- `[1, [2, 3], foo, Bar]`
- `[-123456789, [[[]], [+987654321, baZZ]]]`

# Семантика ЯП

**Семантика = содержание или смысл ЯП.**

**Описание семантики:** математические и логические модели.

**Составляющие семантики:**

- статическая семантика — ограничение на инструкции, не формализуемые в БНФ (напр., декларирование переменных, анализ инициализации, ограничения, связанные с системой типов);
- динамическая семантика — определение поведения отдельных конструкций языка, преобразование дерева разбора в машинные инструкции;
- формальная семантика — анализ кода программы для получения ее характеристик, напр., для доказательства корректности (логика Хоара) или оптимизации.

# Различия между синтаксисом и семантикой

**Синтаксические и семанические ошибки в Java:**

```
1  /* Некорректный синтаксис. */
2  int x == 5;
3
4  /* Некорректная семантика — неизвестный тип переменной. */
5  MissingType y;
6
7  /* Некорректная семантика — переменная не определена. */
8  z = 3;
9
10 /* Некорректная семантика — переменная не инициализирована. */
11 String str;
12 while (str.length() < 20) str += " ";
```

# Система типов

## Определение

**Система типов языка программирования** — совокупность правил, определяющих свойство типа для конструкций языка (переменных, выражений, функций, модулей, …).

**Цели системы типов:**

- **основная:** определение интерфейсов для взаимодействия с частями программы и обеспечение их корректного использования с целью устранения ошибок;
- обеспечение функциональности языка (напр., динамическая диспетчеризация в ООП);
- рефлексия;
- оптимизация;
- повышение доступности программы для понимания.

# Классификация типов

**Категории типов в ЯП:**

- примитивные типы (булев тип, целые и вещ. числа);
- ссылки и указатели;
- составные типы (напр., массивы и записи);
- объекты;
- функции;
- типы типов (напр., `Class<?>` в Java).

**Унификация типов в ООП:**

- языки, в которых все типы наследуются от базового типа (C#, Python);
- языки с изолированными примитивными типами (C++, Java, JavaScript).

# Проверка типов в различных ЯП

**Java:**

```
1 String str = "4" + 2; // str == "42"
```

**Python:**

```
1 x = "4" + 2;
2 # TypeError: cannot concatenate 'str' and 'int' objects
```

**PHP:**

```
1 <?php
2 $number = 2;
3 $x = '4' + $number; // $x == 6
4 $x = '4'.$number; // $x == "42"
```

**C:**

```
1 printf("%s\n", "4" + 2);
2 char* nextStr = "next string";
3 // выведет (скорее всего) "next string"
```

# Статическая и динамическая типизация

**Статическая типизация** — определение типов всех конструкций языка на этапе компиляции программы.

**Виды статической типизации:**

- явная — типы конструкций декларируются программистом (напр., при объявлении переменных);
- неявная — тип переменных выводится в процессе компиляции. Примеры:
  - `var x = 5` в C#;
  - `List<> list = new ArrayList<String>()` в Java 7+.

**ЯП со статической типизацией:** C++, Pascal, Java, C#.

**Динамическая типизация** — определение типов некоторых конструкций и проверка соответствующих ограничений во время выполнения программы.

**ЯП с динамической типизацией:** Python, PHP, Perl, JavaScript.

# Утиная типизация

## Определение

**Утиная типизация** (англ. *duck typing*) — вид динамической типизации, при которой корректность использования объекта определяется набором его методов и свойств, а не типом.

**ЯП с утиной типизацией:**

- языки с ООП на основе прототипов (JavaScript, Lua);
- Python;
- Smalltalk.

**Пример динамической не утиной типизации:** подсказки типов (англ. *data hinting*) в PHP.

# Сильная и слабая типизация

**Типобезопасность** (англ. *type safety*) — предотвращение языком программирования ошибок согласования типов (напр., интерпретации целого числа `int` как вещественного `float`).

**Безопасность памяти** (англ. *memory safety*) — предотвращение ЯП доступа к оперативной памяти, не выделенной в ходе выполнения программы.

**Сильная типизация** — высокая степень типобезопасности и безопасности памяти (напр., отсутствие неявных приведений типов, указателей).

**Примеры ЯП с сильной типизацией:** Python, Java, функциональные ЯП.

**Слабая типизация** — возможность обхода безопасности типов и памяти с помощью конструкций ЯП (напр., `void*` в C/C++).

**Примеры ЯП со слабой типизацией:** C, C++, Visual Basic.

# Полиморфизм

## Определение

**Полиморфизм** — использование единого интерфейса для сущностей различных типов.

**Виды полиморфизма:**

- Специальный (ad hoc) полиморфизм — определение различных реализаций для конечного числа фиксированных наборов входных типов (напр., перегрузка функций / методов).
- Параметрический полиморфизм — определение обобщенной реализации для произвольного типа (напр., шаблоны в C++; generics в Java).
- Полиморфизм подтипов — использование интерфейса класса для любого производного от него подкласса (применяется в ООП).

# Реализация ЯП

- **Компилируемые ЯП** — исходный код преобразуется в машинный *до начала* выполнения.

  **Примеры:** C/C++, Pascal.

- **Интерпретируемые ЯП** — преобразование кода в машинные инструкции *по ходу* выполнения.

  **Примеры:** PHP, JavaScript.

- Компиляция в независимый от платформы код и его интерпретация.

  **Примеры:** Java (байт-код), C# (CLR — common language runtime).

# Метапрограммирование

## Определение

**Метапрограммирование** — разработка программ, обращающихся с программами как с данными:

- создание программ, порождающих другие программы (в том числе во время компиляции);
- разработка программ, модифицирующих свой код во время исполнения.

**Цели метапрограммирования:**

- минимизация затрат на разработку;
- оптимизация кода;
- автоматизация разработки за счет использования высокоуровневых абстракций.

# Виды метапрограммирования

1. **Шаблоны** — автоматическая генерация мало различающихся параметризованных фрагментов кода.

   **Примеры параметров:** типы данных, размеры массивов.

   **Примеры:** препроцессоры в C и сходных языках; шаблоны в C++.

2. **Интерпретация** произвольного исходного кода с помощью специальных функций.

   **Примеры:** Функция `eval` в Python, PHP, JavaScript и других интерпретируемых ЯП.

3. Использование **рефлексии** (англ. *reflection*) для самомодификации программы во время выполнения.

4. Использование **предметно-специфичных языков** (англ. *domain-specific languages*) с последующей трансляцией или интерпретацией при помощи ЯП общего назначения.

# Рефлексия

## Определение

**Рефлексия** или **отражение** (англ. *reflection*) — отслеживание и изменение структуры и поведения программы во время ее выполнения.

**Возможности рефлексии:**

- поиск и извлечение информации о типах во время исполнения;
- определение дополнительной информации о классах / методах (напр., данные об аннотациях в Java);
- создание или переопределение классов / методов во время работы программы.

# Варианты использования рефлексии

- **Тестирование:** создание mock-объектов — заглушек, реализующих требуемые функции программного окружения.
- **Контейнеры объектов** (напр., серверы приложений): управление жизненным циклом объектов.
- **Интерпретация предметно-ориентированных языков:** установление связи между объектами ЯП общего назначения и инструкциями DSL.

# Пример рефлексии

**Рефлексия в JavaScript:**

```
1  function Button() { /* ... */ };
2
3  Button.prototype = {
4      onClick: function() { /* ... */ },
5      onDoubleClick: function() { /* ... */ },
6      /* ... */
7
8      /** Заменяет все методы объекта, начинающиеся на 'on', на заглушки. */
9      disable4ever: function() {
10         var noop = function() { alert('I am disabled!'); };
11         for (var member in this) {
12             if ((typeof(this[member]) == 'function')
13                     && (member.substring(0, 2) == 'on')) {
14                 this[member] = noop;
15             }
16         }
17     }
18 };
```

# DSL

## Определение

**Предметно-ориентированный язык** (англ. *domain-specific language*) — компьютерный язык, специализированный для конкретной области применения.

**Примеры ПОЯ**

| **Область** | **Язык(и)** |
|---|---|
| Веб-страницы (отображение) | HTML, CSS |
| Компьютерная верстка | TEX/ LATEX |
| Веб-страницы (генерация) | Языки шаблонов (Twig, Jinja / шаблоны Django, JSP) |
| Матричное программирование | Matlab, Octave |
| Реляционные СУБД | SQL |

# Причины появления DSL

**Задача:** создание представлений (view) в рамках архитектуры MVC.

**Наивное решение:** использование ЯП общего назначения (напр., Python).

### Наивное представление в ЯП Python

```
1  # Список книг, взятый из базы данных.
2  books = [
3      { 'author': 'Ray Bradbury', 'title': '451F' },
4      { 'author': 'Herman Melville', 'title': 'Moby-Dick' }
5  ];
6
7  # HTML-код таблицы с данными о книгах.
8  print '<table>';
9  print '<thead><tr><th>Author</th><th>Title</th></tr></thead>';
10 print '<tbody>';
11 for book in books:
12     print '<tr><td>%s</td><td>%s</td></tr>'
13         % (book['author'], book['title']);
14 print '</tbody>';
15 print '</table>';
```

# Причины появления DSL

**Недостатки использования ЯП общего назначения:**

- громоздкость кода (избыток функций вывода и т. п.);
- соблазн внести код, не относящийся к созданию представления;
- разработчик интерфейса сайта может не знать ЯП общего назначения.

**Решение:** использование языка шаблонов (HTML с дополнительными управляющими конструкциями).

```
<table>
    <thead><tr><th>Author</th><th>Title</th></tr></thead>
    <tbody>
    {% for book in books %}
    <tr>
        <td>{{ book.author }}</td><td>{{ book.title }}</td>
    </tr>
    {% endfor %}
    </tbody>
</table>
```

# Место DSL в разработке ПО

**Варианты использования DSL:**

1. DSL используется для конфигурации компонентов, написанных на ЯП общего назначения;
2. DSL встраивается в ЯП общего назначения (напр., Java Scripting Engine);
3. DSL транслируется (чаще всего — в код на ЯП высокого уровня).

**Инструменты для разработки DSL:**

- средства лексического и синтаксического анализа (yacc, lex, bison, ANTLR);
- инструменты моделирования предметной области (Eclipse Modeling Framework).

# Преимущества DSL

- **Разделение ответственности** (англ. *separation of concerns*) — независимость семантики языка от его реализации;
- **высокий уровень абстракции** DSL — позволяет сократить объем решения, повысить его доступность и упростить отладку;
- упрощение **портирования** — при переносе в другую среду меняется лишь имплементация DSL;
- легкость в освоении использовании **профильными специалистами**.

# Недостатки DSL

- **Затраты** на проектирование, имплементацию и сопровождение языка и соответствующих инструментов (напр., поддержки среды разработки);
- проблема **определения границ** DSL;
- трудности **интеграции** DSL в программную систему;
- проблема **стандартизации** языков одной предметной области;
- **низкая производительность** по сравнению с языками общего назначения.

# Выводы

1. Языки программирования характеризуются синтаксисом (формой) и семантикой (содержанием). Семантика языка может использоваться не только в процессе компиляции / интерпретации, но и при анализе программ, напр., при доказательстве их корректности.
2. Важная часть семантики языка программирования — система типов. ЯП может обладать статической или динамической, строгой или слабой типизацией.
3. Метапрограммирование (порождающее программирование) — парадигма программирования, в которой код программ рассматривается как данные. Основными инструментами метапрограммирования являются рефлексия и использование предметно-ориентированных языков.

# Материалы


- Czarnecki, K.; Eisenecker, U. W.

  Generative Programming.

- Fowler, M.

  Domain Specific Languages.

  http://martinfowler.com/dsl.html

- Лавріщева К. М.

  Програмна інженерія (підручник).

  К., 2008. — 319 с.

# Спасибо за внимание!