

**K10.1**

# Dynamische Programmiersprachen

# Dynamische Programmiersprachen

- Der Begriff *Dynamischen Programmiersprache* ist nicht präzise bzw. formal definiert.
- Typischerweise versteht man darunter Sprachen ...
  - ... bei denen zur **Laufzeit** Tätigkeiten ausgeführt werden, die bei „statischen Sprachen“\* zur Compilezeit ausgeführt werden, z.B.:
    - Dynamische Typüberprüfung
    - Dynamische Bestimmung des Typs anhand struktureller Eigenschaften
    - Modifikation von Programmen zur Laufzeit
  - ... die Interpretiert ausgeführt werden (**Skriptsprachen**)
  - ... die automatische Speicherverwaltung beinhalten
- Motivation:
  - Produktivität bei der Programmierung erhöhen und/oder
  - Erlernbarkeit der Programmiersprache vereinfachen

\* Auch dieser Begriff ist nicht präzise definiert.

# Dynamische Typüberprüfung (i)

- Typüberprüfung erfolgt zur Laufzeit (im Gegensatz zur statischen Typprüfung zur Compilezeit).
    - Vertreter: Groovy, JavaScript, Lisp, Lua, Perl, PHP, Ruby, Python, ...
  
  - Charakteristische Eigenschaft:
    1. Werte/Objekte haben Typ
      - Ergo: Typinformation muss zusätzlich mit dem Wert/Objekt geführt werden.
    2. Variablen haben keinen Typ
      - Ergo: Variable kann während ihrer Lebenszeit Werte beliebigen Typs annehmen.
- Es können **weniger bis gar keine Garantien zur Typkompatibilität** zur Laufzeit gegeben werden.

# Dynamische Typüberprüfung (ii)

- Nachteile:
  - Typüberprüfung zur Laufzeit hat gewissen Aufwand (Overhead).
  - Laufzeitfehler durch Typinkompatibilitäten.
    - Auftretens- und Ursacheort im Quelltext u.U. weit voneinander entfernt
  - Erhöhter Testaufwand; volle Abdeckung aller möglichen Ausführungen aber oft nicht mit sinnvollem Aufwand machbar.

# Duck Typing

(i)

- **Idee:** Ein Objekt ist kompatibel zu einem Typ  $T$  wenn es zur Laufzeit die Methoden (und Attribute) von  $T$  besitzt.
  - Analog: zwei Objekte sind typkompatibel wenn sie zur Laufzeit gleiche Methoden (und Attribute) besitzen.
  - Ergo: Objekt muss im Grunde genommen überhaupt keinen statischen Typ besitzen. Der Typ wird dynamisch (zur Laufzeit) anhand seiner **Struktur**, nämlich der vorhandenen Methoden (und Attribute), bestimmt.

Name geht zurück auf ein Gedicht von J.W. RILEY:

*„When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.“*

# Duck Typing

(ii)

Beispiel (Python):

```
class Vogel:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return
        self.__class__.__name__
        + ' ' + self.name

class Ente(Vogel):
    def quak(self):
        print str(self)+' : quak'

class Frosch:
    def quak(self):
        print str(self)+' : quak'
```

```
tiere = [Vogel('Gustav'),
         Ente('Donald'),
         Frosch('Kermit')]

for t in tiere:
    try:
        t.quak()
    except AttributeError:
        print 'Kann nicht quaken:', t
```

Ausgabe:

```
Kann nicht quaken: Vogel Gustav
Ente Donald: quak
Frosch Kermit: quak
```

# Duck Typing – Gegenüberstellung (iii)

- Duck Typing ist dynamische Form des **Structural Typing**. Bei letzterem wird Typkompatibilität i. Allg. statisch zur Compilezeit bestimmt.
- Die Typsysteme der bisher in der Vorlesung behandelten Sprachen (C++, Java, Haskell) sind **nominativ**:
  - Typkompatibilität bzw. -äquivalenz bestimmt durch **explizite** Typdeklarationen und/oder Typ**namen**.

```
// explizite Deklaration äquivalenter  
// Typen durch Typalias in C++
```

```
typedef unsigned char BYTE;
```

```
-- explizite Deklaration äquivalenter  
-- Typen durch Typsynonyme in Haskell
```

```
type String = [Char];
```

```
// unterschiedliche Typen in C++  
// mit identischer Struktur
```

```
struct Foo {  
    int id; char name[80];  
};
```

```
struct Bar {  
    int id; char name[80];  
};
```

# Mixins

(i)

- **Idee:** Ein **Mixin** deklariert und **definiert** eine Menge von Methoden die man (zur Laufzeit) beliebig vielen Klassen hinzufügt – „beimischt“, ohne dass es dabei eine Superklasse sein muss (was damit den Unterschied zur Vererbung darstellt).
- Direkt unterstützt u.a. in: Go, Ruby, Scala

Beispiel (Ruby):

```
# Mixin das Logging-Methoden bereitstellt
module Logging
  def logInfo # Methode für Info-Ausgabe
    # Implementierung ...
  end
  def logErr # Methode für Fehler-Ausgabe
    # Implementierung ...
  end
end
```

```
# Verwendung
class Foo
  include Logging
  # ...
end
class Bar
  include Logging
  # ...
end
```



# Mixins

(ii)

- Konzept in der objektorientierten Programmierung:
  - Keine Form der Spezialisierung (Vererbung) sondern ein Mittel um Funktionalität (dynamisch) einer Klasse hinzuzufügen.
- Motivation: don't repeat yourself (dry)
- Vergleichbar mit:
  - **Mehrfachvererbung**: Klasse kann (alle) Methoden über ein oder mehrere Mixins erhalten.
    - Unterschied: eliminiert Probleme des mehrfachen Erbens von Methoden (siehe auch Diamond-Problem bzw. virtuelle Mehrfachvererbung in C++)
  - **Java Interface**
    - Unterschied: Methoden sind implementiert.
- Manche Sprachen ermöglichen das „beimischen“ von Methoden in eine Klasse zur Laufzeit, wobei Mixins selbst zur Compilezeit definiert sind. Dies ist eine Form des *late binding*.

# Metaprogrammierung

- Mit **Lisp** wurde erstmalig die Möglichkeit eingeführt, ein in ein Programm als Daten eingebettetes **Programmfragment**  $p$  (welches i.A. nicht notwendigerweise in derselben Sprache geschrieben ist) **auszuwerten** und das Ergebnis zurückzuliefern (Ausdruck) bzw. **auszuführen** (Anweisung(en)).
- Auswertung/Ausführung von  $p$  entweder zur:
  - Compilezeit: **Statische** Metaprogrammierung  
z.B.: C/C++ Makros und C++ Templates
  - Laufzeit: **Dynamische** Metaprogrammierung
- Auswertung zur Laufzeit verfügbar u.a. in Lua, Perl, Python, Ruby, JavaScript.

# Metaprogrammierung – eval

- Egal ob Auswertung zur Compile- oder Laufzeit stattfindet, es wird **Evaluierungsfunktion**:  $eval(p)$  benötigt:
  - $p$  liegt in Form von **Daten** für  $eval$  vor.  
Demzufolge unterscheidet  $eval$  bzgl.  $p$  **nicht** zwischen Programm und Daten.
  - Auswertung/Ausführung zur Laufzeit meist interpretiert.
  - Kompilierte Auswertung/Ausführung benötigt Zugriff auf einen Compiler durch das ausführbare Programm.
- Syntax bzgl.  $eval$  variiert zwischen den Programmiersprachen; z.B. implizite Kennzeichnung durch Präfix **#** bei C/C++ Makros.

# eval - Beispiele

Java Script:

```
// Auswertung eines Ausdrucks
x = 3;
alert(eval('x + 3'));

// Ausführung von Anweisungen
x = 3;
eval('x += 3; alert(x);');
```

Perl:

```
// Auswertung eines Ausdrucks
$x = 3;
print eval('$x + 3'), "\n";

// Ausführung von Anweisungen
$x = 3;
eval('$x += 3; print "$x\n";');
```

Python (interaktiver Modus >>>):

```
>>> # Auswertung eines Ausdrucks
>>> x = 3
>>> eval('x + 3')
6
```

```
>>> x = 3
>>> y = 4
>>> exec "x += 3; y += 4"
>>> x
6
>>> y
8
```

# eval und Quines

- **eval** lässt sich verwenden um sogenannte **Quines** zu implementieren:
  - Definition: Ein Quine ist ein Programm das keine Eingaben hat und bei Ausführung exakt seinen eigenen Quelltext ausgibt.
  - Auf dieser Basis ist ein Quine damit eine Spezialform eines Metaprogrammes.

Beispiel (Ruby):

```
eval s=%q(puts"eval s=%q(#{s})")
```

Ausprobierbar ohne dass man eine Ruby-Laufzeitumgebung  
Installieren muss auf <http://codepad.org/>

# Metaprogrammierung: Dynamische Programmmodifikation

- Ebenfalls mit **Lisp** wurde erstmalig die Möglichkeit eingeführt dass ein Programm sich selbst zur Laufzeit modifiziert.
- **Programm benutzt sich selbst als Daten**. Durch Modifikation dieser Daten kann es sich selbst zur Laufzeit modifizieren.
  - Ob dies in der Folge zu einer unmittelbaren Änderung der Programmausführung führt, ist abhängig davon, ob die Modifikation direkt den Maschinencode (im Codesegment) ändert, bzw. ob sie so geschieht, dass ein Interpreter, der das aktuelle Programm interpretiert, die Modifikation „sieht“.

# Dyn. Programmmodifikation – Beispiel

- Ruby: Hinzufügen einer neuen Methode zu einer Klasse und Ausführung dieser:

```
class Foo
  # initiale Definition der Klasse ...
end

# füge neue Methode 'hello' zur Klasse 'Foo' hinzu
Foo.class_eval("def hello; return 'hello'; end")

# erzeuge neue Instanz und führe 'hello' aus
Foo.new.hello
```

- Ähnlich kann man mit **remove\_method** in Ruby eine Methode  $m^C$  einer Klasse  $C$  zur Laufzeit entfernen. Beachte: hat  $m^C$  jedoch  $m^{Sup}$  einer Superklasse  $Sup$  überschrieben, dann kann danach immer noch  $m^{Sup}$  auf Instanzen von  $C$  aufgerufen werden (da ererbt).

**K10.2**

# Domänenspezifische Programmiersprachen



# Unterscheidung

- Alle bisher betrachteten Sprachen zählen zu den **general purpose** Sprachen: sie sind **universell**, d.h. für jegliche Programmieraufgaben (mehr oder weniger gut) benutzbar.
  - Programmiersprachen sind nicht *per se* gut oder schlecht; sie sind nur mehr oder weniger gut geeignet für einen bestimmten Zweck.
- Eine domänenspezifische Sprache (DSL) ist für einen bestimmten, eingegrenzten Problembereich entworfen.
  - Ziel: durch Domänenexperten ohne Zusatzwissen nutzbar: „*weniger technisch*“.
  - Beispiele für Nicht-Ziele, als bewusste Designentscheidung:
    - Turing-Vollständigkeit
    - Ausführbarkeit – Modellierungssprachen wie z.B. UML

# Typen von DSLs

(i)

Man unterscheidet zwischen zwei Typen von DSLs:

## 1. Interne DSL:

- Definiert mittels einer Wirtsprogrammiersprache (host language).
- DSL („Programm“) ist eingebettet in ein Wirtsprogramm.
- Freiheitsgrade bzgl. der DSL daher durch Wirtssprache beschränkt.

Beispiele interner DSLs:

- **Gradle**: Build-Sprache, auf Basis von Groovy definiert.
- **Rake**: Build-Sprache, auf Basis von Ruby definiert.

# Typen von DSLs

(ii)

## 2. Externe DSL:

- Nicht mittels einer bestehenden Programmiersprache definiert
  - Definition der Syntax als formale Grammatik meist mit entsprechenden Tools
- Freiheitsgrade „nur“ durch Editoren und Gesetze bzw. Grenzen der maschinellen Verarbeitung beschränkt (Berechenbarkeit, Ausführbarkeit, Komplexität).
- Bisher hoher Aufwand für Entwicklung externer DSLs. Neue Frameworks zur Entwicklung/Spezifikation erleichtern dies erheblich, z.B.:  
**Xtext** [<http://www.eclipse.org/Xtext>]
  - *Erweiterte Backus-Naur-Form* (EBNF) zur Beschreibung der Grammatik
  - Parser, Compiler und Texteditor werden automatisch erzeugt

# Beispiele externer DSLs

- **Csound**: Software-Synthesizer zur Klang-/Musikerzeugung
- **LaTeX**: Textsatz
- **PROMOS**: Modellierung von (finanz-)mathematischen Eigenschaften von Produkten, z.B. Versicherungsprodukte
  - Struktur, Varianten, Kostenmodell, Regeln
- **R**: statistisches Rechnen und Erzeugung statistischer Grafiken
- **Sieve**: Definition von E-Mail-Filtern
- **SQL**: Anfragesprache für relationale Datenbanken
- **Verilog, VHDL**: Hardwarebeschreibungssprachen
- ... und viele, viele mehr

# Ausblick

Wo geht die Entwicklung von Programmiersprachen hin? Werden wir neue Paradigmen sehen? Was sind aktuelle Trends?

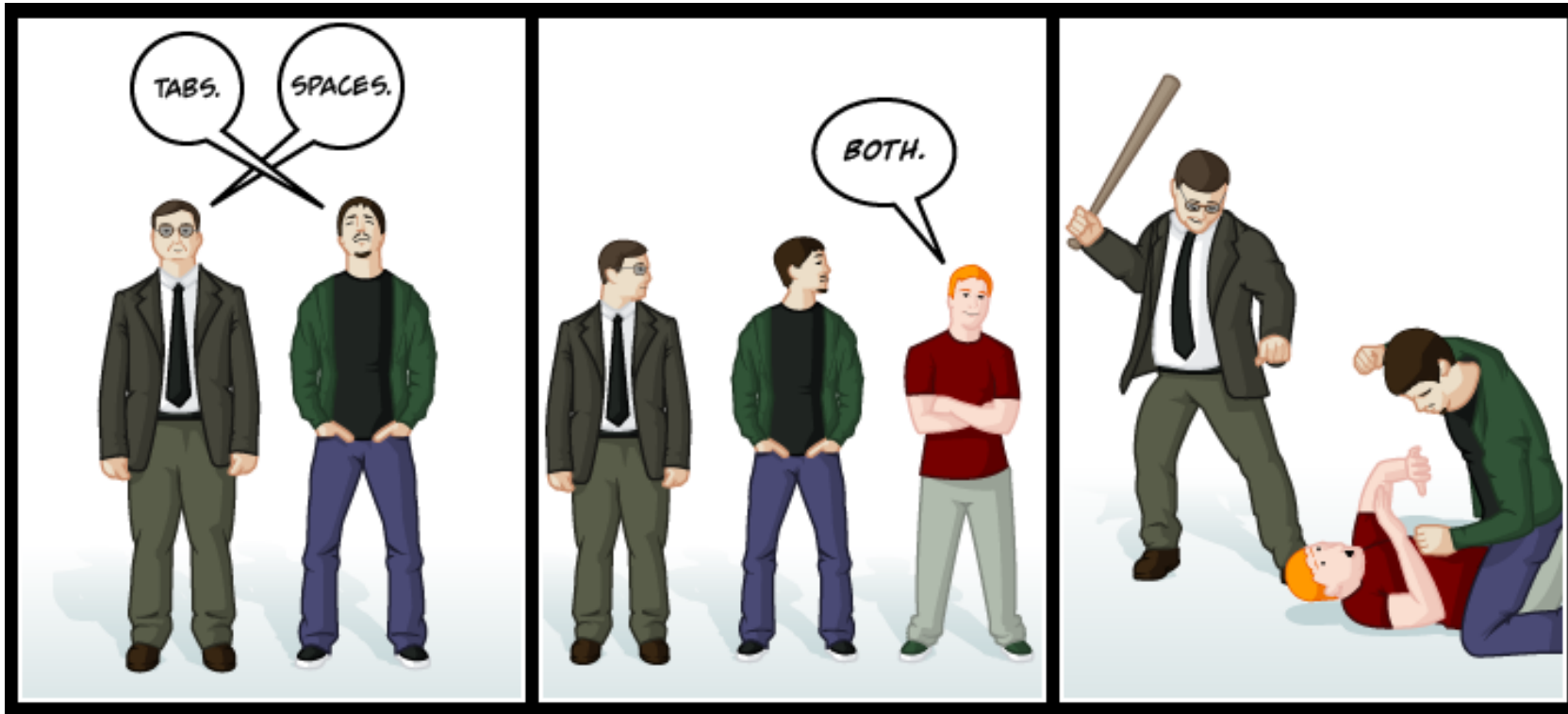
Trends:

- **Multi-Paradigmen-Sprachen**
  - Imperativ, objektorientiert, funktional, generisch, und seltener auch logikorientiert
- **Polyglott-Programmierung**
  - Verwendung und enge Kopplung mehrerer Programmiersprachen
- Neue Sprachen auf Grund **neuer Berechnungsmodelle**
  - Probabilistische Programmiersprachen, Quantencomputing
- **Modulare, erweiterbare Sprachen\***
  - Selbstdefinierte Module (Erweiterungen) für domänenspezifische Belange
- **Typsysteme**
  - Dependent Types: Polymorphe Typen, die dynamisch von Daten bestimmt werden.

\* Sind eine logische Konsequenz dessen was auch in natürlichen Sprachen passiert: sie verändern sich, da jeder sie verändern kann.

# ... Paradigmen ... ???

Was hat dieses Comic mit (Programmier-) Paradigmen zu tun?



Antwort: In der Praxis der Programmierung bzw. Softwareentwicklung findet man mindestens ein **nichttechnisches** Paradigma, welches nahezu unüberwindbar ist.