Perl 6, genau jetzt!

Ingo Blechschmidt
<iblech@web.de>

LUGA

4. Mai 2005

Inhalt

Parrot?

- Registerbasierte virtuelle Maschine
- Plattformunabhängiger Bytecode
- „One bytecode to rule them all“

BASIC Forth
Lisp Parakeet
Perl 5 Perl 6
PHP Python
Ruby Scheme
TCL ...

↔ Parrot ↔

BASIC Forth
Lisp Parakeet
Perl 5 Perl 6
PHP Python
Ruby Scheme
TCL ...

Parrot?

- Verdammt schnelle registerbasierte virtuelle Maschine
- Plattformunabhängiger Bytecode
- „One bytecode to rule them all“

Parrot?

- Verdammt schnelle registerbasierte virtuelle Maschine
- Plattformunabhängiger Bytecode
- „One bytecode to rule them all“

BASIC	Forth			BASIC	Forth
Lisp	Parakeet			Lisp	Parakeet
Perl 5	Perl 6	$\leftrightarrow$	Parrot $\leftrightarrow$	Perl 5	Perl 6
PHP	Python			PHP	Python
Ruby	Scheme			Ruby	Scheme
TCL	...			TCL	...

Architektur

- Je 64 Register der Typen `I`, `N`, `S` und `P`

  `I`: Int
  `N`: Num
  `S`: String
  `P`: PMC – Parrot Magic Cookie
- Bereitstellung von Garbage Collection, Subroutinen, Coroutines, Continuations, Klassen, Methoden
- Just In Time-Compiler für meist benutzte Plattformen $\Rightarrow$ Speed!
- HLL $\rightarrow$ PIR $\rightarrow$ Bytecode $\rightarrow$ natives Programm, C oder direkte Ausführung

Beteiligungsmöglichkeiten

- Mailingliste:
  `perl6-internals@perl.org`,
  `gmane.comp.lang.perl.perl6.internals`
- IRC: `#parrot` auf `irc.perl.org`
- Auch Newbies gern gesehen
- Viele Beteiligungsmöglichkeiten, nicht nur für Programmierer
- Weitere Informationen: `http://www.parrotcode.org/`

„Die Wasserbett-Theorie“

- Glattes Wasserbett
- Eindrücken an einer Stelle (Vereinfachung) $\Rightarrow$ Hochkommen an anderen Stellen (Komplizierung)

Gleichgewicht

- Sehr viele Operatoren: `+@= /%^ !@=` $\leftrightarrow$
- Sehr wenig Operatoren: `set(x, add(5, 3))`

„Die Wasserbett-Theorie“

- Glattes Wasserbett
- Eindrücken an einer Stelle (Vereinfachung) ⇒ Hochkommen an anderen Stellen (Komplizierung)

Gleichgewicht

- Sehr viele Operatoren: `+@= /%^ !@=` ↔
- Sehr wenig Operatoren: `set(x, add(5, 3))`

„Die Wasserbett-Theorie“

- Glattes Wasserbett
- Eindrücken an einer Stelle (Vereinfachung) $\Rightarrow$ Hochkommen an anderen Stellen (Komplizierung)

Gleichgewicht

- Sehr viele Operatoren: `+@= /%^ !@=` $\leftrightarrow$
- Sehr wenig Operatoren: `set(x, add(5, 3))`

Huffmann-Komprimierung

- Häufig genutzte Features $\longrightarrow$
  kurze Namen (z.B. `say`, `cat`)
- Selten genutzte Features $\longrightarrow$
  lange Namen (z.B. `gethostbyname`,
  `nautilus-file-management-properties`)

Freiheit und Anpassung

- Kein Zwänge, nur angemessene Standards
- Bräuche, keine Gesetze
- „Nur tote Sprachen ändern sich nicht.“

Möglichkeiten der Anpassung

- Überschreiben/Erweitern von Builtins und Operatoren
- C-like und Lisp-like Makros
- Neudefinition der Grammatik:

```
use Grammar::Python;
x = [ foo(), bar() ]
x.push(3)
```

Freiheit und Anpassung

- Kein Zwänge, nur angemessene Standards
- Bräuche, keine Gesetze
- „Nur tote Sprachen ändern sich nicht.“

Möglichkeiten der Anpassung

- Überschreiben/Erweitern von Builtins und Operatoren
- C-like und Lisp-like Makros
- Neudefinition der Grammatik:

```
use Grammar::Python;
x = [ foo(), bar() ]
x.push(3)
```

Freiheit und Anpassung

- Kein Zwänge, nur angemessene Standards
- Bräuche, keine Gesetze
- „Nur tote Sprachen ändern sich nicht.“

Möglichkeiten der Anpassung

- Überschreiben/Erweitern von Builtins und Operatoren
- C-like und Lisp-like Makros
- Neudefinition der Grammatik:

```
use Grammar::Python;
x = [ foo(), bar() ]
x.push(3)
```

DWIM – Do What I Mean

- Do What I Mean
- Nicht immer gleich Do What You Mean

Beispiele

```
5 + 3*2          # 5 + (3*2)
s:2nd/foo/bar/ # Zweites "foo" → "bar"
s:2th/foo/bar/ # ditto
```

DWIM – Do What I Mean

- Do What I Mean
- Nicht immer gleich Do What You Mean

Beispiele

```
5 + 3*2          # 5 + (3*2)
s:2nd/foo/bar/ # Zweites "foo" → "bar"
s:2th/foo/bar/ # ditto
```

„Ausleihen“

- „Ausleihen“ von Features aus anderen Sprachen
- Zusammenarbeit zwischen den Sprachenentwicklern

Beispiele

- `.` als Methodenaufrufoperator (von Python, Ruby und vielen anderen Sprachen)
- Pragmas (ursprünglich von C)
- Klammern: „Oha! Hier ist etwas anders als normal!“ (Perl 5, Ruby, Mathematik)
  - $5 + (3 \cdot 2) \rightarrow 5 + 3 \cdot 2$
  - `say(...)` → `say ...`

„Ausleihen“

- „Ausleihen“ von Features aus anderen Sprachen
- Zusammenarbeit zwischen den Sprachenentwicklern

Beispiele

- . als Methodenaufrufoperator (von Python, Ruby und vielen anderen Sprachen)
- Pragmas (ursprünglich von C)
- Klammern: „Oha! Hier ist etwas anders als normal!“ (Perl 5, Ruby, Mathematik)
  - $5 + (3 \cdot 2) \rightarrow 5 + 3 \cdot 2$
  - `say(...)` → `say ...`

„Hello World“

Perl 6: `say "Hello, World!";`
Perl 5: `print "Hello, World!\n";`

C:
```
printf("%s", "Hello, World!\n");
```
Haskell:
```
putStrLn "Hello, World!"
```
Lua:
```
print "Hello, World!";
```
PHP:
```
print("Hello, World!\n");
```
Python:
```
print "Hello, World!"
```
Ruby:
```
puts "Hello, World!"
```
Shell:
```
echo "Hello, World!"
```
Tcl:
```
puts "Hello, World!"
```

Variablen

- Haupttypen:
  `$scalar`, `@array`, `%hash`, `&code`
- Dereferenzierung:
  `@array[3]`,
  `%hash{"key"}`, `%hash<key>`,
  `&code(argumente)`
- Slices:
  `@array[23, 42]`,
  `%hash{"ingo", "larry"}`

Variablen

- Haupttypen:
  `$scalar`, `@array`, `%hash`, `&code`
- Dereferenzierung:
  `@array[3]`,
  `%hash{"key"}`, `%hash<key>`,
  `&code(argumente)`
- Slices:
  `@array[23, 42]`,
  `%hash{"ingo", "larry"}`

Variablen

- Haupttypen:
  `$scalar`, `@array`, `%hash`, `&code`
- Dereferenzierung:
  `@array[3]`,
  `%hash{"key"}`, `%hash<key>`,
  `&code(argumente)`
- Slices:
  `@array[23, 42]`,
  `%hash{"ingo", "larry"}`

Operatoren

- Standard-Operatoren:
  - `+ * - / ~`
  - `[] {}`
  - `.`
  - `...` (Yadda-Yadda)
- Chained Operators: `23 < $a < 42`
- Kontextspezifizierer:
  - `+@array` (Anzahl Elemente in `@array`)
  - `~@array` (`@array` als Zeichenkette)
  - `?@array` („Ist `@array` nicht leer?“)
- Hyper-Operatoren:

```
[1, 2, 3] »*« 2          # [2, 4, 6]
[1, 2, 3] »+« [4, 5, 6] # [5, 7, 9]
```

Operatoren

- Standard-Operatoren:
  - `+ * - / ~`
  - `[] {}`
  - `.`
  - `...` (Yadda-Yadda)
- Chained Operators: `23 < $a < 42`
- Kontextspezifizierer:
  - `+@array` (Anzahl Elemente in `@array`)
  - `~@array` (`@array` als Zeichenkette)
  - `?@array` („Ist `@array` nicht leer?“)
- Hyper-Operatoren:

```
[1, 2, 3] »*« 2          # [2, 4, 6]
[1, 2, 3] »+« [4, 5, 6] # [5, 7, 9]
```

Operatoren

- Standard-Operatoren:
  - `+ * - / ~`
  - `[] {}`
  - `.`
  - `...` (Yadda-Yadda)
- Chained Operators: `23 < $a < 42`
- Kontextspezifizierer:
  - `+@array` (Anzahl Elemente in `@array`)
  - `~@array` (`@array` als Zeichenkette)
  - `?@array` („Ist `@array` nicht leer?“)
- Hyper-Operatoren:

```
[1, 2, 3] »*« 2          # [2, 4, 6]
[1, 2, 3] »+« [4, 5, 6] # [5, 7, 9]
```

Operatoren

- Standard-Operatoren:
  - `+ * - / ~`
  - `[] {}`
  - `.`
  - `...` (Yadda-Yadda)
- Chained Operators: `23 < $a < 42`
- Kontextspezifizierer:
  - `+@array` (Anzahl Elemente in `@array`)
  - `~@array` (`@array` als Zeichenkette)
  - `?@array` („Ist `@array` nicht leer?“)
- Hyper-Operatoren:

```
[1, 2, 3] »*« 2          # [2, 4, 6]
[1, 2, 3] »+« [4, 5, 6] # [5, 7, 9]
```

Junctions

„Wenn dies, dies oder dies wahr ist, dann...“

Perl 5:
```
if($a == 3 || $a == 5 || $a == 7) {...}
```
Perl 6:
```
if $a == 3|5|7 {...}
```

„Wenn alle mindestens 18 sind, dann...“

Perl 5:
```
if(grep({ $_->age < 18 } @leute) == 0) {...}
```
Perl 6:
```
if all(@leute».age) >= 18 {...}
```

„Wenn mindestens einer mindestens 18 ist, dann...“

Perl 5:
```
if(grep { $_->age >= 18 } @leute) {...}
```
Perl 6:
```
if any(@leute».age) >= 18 {...}
```

Junctions

„Wenn dies, dies oder dies wahr ist, dann...“

Perl 5:
```
if($a == 3 || $a == 5 || $a == 7) {...}
```
Perl 6:
```
if $a == 3|5|7 {...}
```

„Wenn alle mindestens 18 sind, dann...“

Perl 5:
```
if(grep({ $_->age < 18 } @leute) == 0) {...}
```
Perl 6:
```
if all(@leute».age) >= 18 {...}
```

„Wenn mindestens einer mindestens 18 ist, dann...“

Perl 5:
```
if(grep { $_->age >= 18 } @leute) {...}
```
Perl 6:
```
if any(@leute».age) >= 18 {...}
```

Junctions

„Wenn dies, dies oder dies wahr ist, dann...“

Perl 5:
```
if($a == 3 || $a == 5 || $a == 7) {...}
```
Perl 6:
```
if $a == 3|5|7 {...}
```

„Wenn alle mindestens 18 sind, dann...“

Perl 5:
```
if(grep({ $_->age < 18 } @leute) == 0) {...}
```
Perl 6:
```
if all(@leute».age) >= 18 {...}
```

„Wenn mindestens einer mindestens 18 ist, dann...“

Perl 5:
```
if(grep { $_->age >= 18 } @leute) {...}
```
Perl 6:
```
if any(@leute».age) >= 18 {...}
```

Junctions

„Wenn dies, dies oder dies wahr ist, dann...“

Perl 5:
```
if($a == 3 || $a == 5 || $a == 7) {...}
```
Perl 6:
```
if $a == 3|5|7 {...}
```

„Wenn alle mindestens 18 sind, dann...“

Perl 5:
```
if(grep({ $_->age < 18 } @leute) == 0) {...}
```
Perl 6:
```
if all(@leute».age) >= 18 {...}
```

„Wenn genau einer einer mindestens 18 ist, dann...“

Perl 5:
```
if(grep({ $_->age >= 18 } @leute) == 1) {...}
```
Perl 6:
```
if one(@leute».age) >= 18 {...}
```

Smartmatching („Extreme DWIM“)

```
# Enthält $str "foo"?
if $str ~~ m/foo/   {...}

# Enthält @array "ingo"?
if "ingo" ~~ @array {...}

# Gibt es einen Key "ingo" in %hash?
if "ingo" ~~ %hash  {...}

# Sind @foo und @bar identisch?
if @foo ~~ @bar     {...}
```

Smartmatching („Extreme DWIM“)

```
# Enthält $str "foo"?
if $str ~~ m/foo/   {...}

# Enthält @array "ingo"?
if "ingo" ~~ @array {...}

# Gibt es einen Key "ingo" in %hash?
if "ingo" ~~ %hash  {...}

# Sind @foo und @bar identisch?
if @foo ~~ @bar     {...}
```

Smartmatching („Extreme DWIM“)

```
# Enthält $str "foo"?
if $str ~~ m/foo/   {...}

# Enthält @array "ingo"?
if "ingo" ~~ @array {...}

# Gibt es einen Key "ingo" in %hash?
if "ingo" ~~ %hash  {...}

# Sind @foo und @bar identisch?
if @foo ~~ @bar     {...}
```

Smartmatching („Extreme DWIM“)

```
# Enthält $str "foo"?
if $str ~~ m/foo/   {...}

# Enthält @array "ingo"?
if "ingo" ~~ @array {...}

# Gibt es einen Key "ingo" in %hash?
if "ingo" ~~ %hash  {...}

# Sind @foo und @bar identisch?
if @foo ~~ @bar     {...}
```

Subroutines – Definition

Perl 6:
```
sub foo(Num $i) { say $i + 3 }
```
Perl 5:
```
sub foo { my $i = shift; print $i + 3, "\n" }
```

C:
```
void foo(float i) { printf("%f\n", i + 3); }
```
Haskell:
```
foo i = putStrLn . show $ i + 3
```
Lua:
```
function foo(i) print(i + 3) end
```
PHP:
```
function foo($i) { print($i + 3); }
```
Python:
```
def foo(i): print i + 3
```
Ruby:
```
def foo(i) puts i + 3 end
```
Shell:
```
function foo { echo $(($1 + 3)); }
```
Tcl:
```
proc foo {i} { puts [expr $i + 3] }
```

Subroutines – Aufruf

Perl 6: `foo 42;` ∨ `foo i => 42;` ∨ `foo :i(42);`
Perl 5: `foo 42;`

C: `foo(42);`
Haskell: `foo 42`
Lua: `foo(42)`
PHP: `foo(42);`
Python: `foo(42)`
Ruby: `foo 42`
Shell: `foo 42`
Tcl: `foo 42`

Klassendefinition und -instantiierung in anderen Sprachen

Perl 5

```
package Foo;
sub new          { bless {}, shift }
sub hallo        { "Hallo " . $_[1] . "!" }
sub bar :lvalue { $_[0]->{foo} }



# Dann:
my $obj = Foo->new;
$obj->bar = 42;
print $obj->hallo("Ingo");
```

Klassendefinition und -instantiierung in anderen Sprachen

PHP

```
class Foo {
  var $bar;
  function hallo($name) {
    return "Hallo $name!";
  }
}

# Dann:
$obj = new Foo();
$obj->bar = 42;
print $obj->hallo("Ingo");
```

Klassendefinition und -instantiierung in anderen Sprachen

Python

```
class Foo:
  bar = None
  def hallo(self, name):
    return "Hallo %s!" % name



# Dann:
obj = Foo()
obj.bar = 42
print obj.hallo("Ingo")
```

Klassendefinition und -instantiierung in anderen Sprachen

Ruby

```ruby
class Foo
  attr_accessor :bar
  def hallo(name)
    return "Hallo #{name}!"
  end
end

# Dann:
obj = Foo.new
obj.bar = 42
puts obj.hallo("Ingo")
```

Klassendefinition und -instantiierung in Perl 6

Perl 6

```
class Foo {
  has $.bar;
  method hallo(Str $name) {
    return "Hallo {$name}!";
  }
}

# Dann:
my $obj = Foo.new;
say $obj.bar;
say $obj.hallo("Ingo");
```

Klassendefinition und -instantiierung in Perl 6

Perl 6

```
class Foo {
  has $.bar is rw;
  method hallo(Str $name) {
    return "Hallo {$name}!";
  }
}

# Dann:
my $obj = Foo.new;
$obj.bar = 42;
say $obj.hallo("Ingo");
```

Klassendefinition und -instantiierung in Perl 6

Perl 6

```
class Foo {
  has $.bar is rw;
  method hallo(Str $name) {
    return "Hallo {$name}!";
  }
}

# Dann:
my $obj = Foo.new;
$obj.bar = 42;
say hallo $obj: "Ingo";
```

Klassendefinition und -instantiierung in Perl 6

Perl 6

```
class Foo {
  has $.bar is rw;
  method hallo(Str $name) {
    return "Hallo {$name}!";
  }
}

# Dann:
my Foo $obj = Foo.new;
$obj.bar = 42;
say hallo $obj: "Ingo";
```

Klassendefinition und -instantiierung in Perl 6

Perl 6

```
class Foo {
  has $.bar is rw;
  method hallo(Str $name) {
    return "Hallo {$name}!";
  }
}

# Dann:
my Foo $obj .= new;
$obj.bar = 42;
say hallo $obj: "Ingo";
```

Klassendefinition und -instantiierung in Perl 6

Perl 6

```
class Foo is Baz {
  has $.bar is rw;
  method hallo(Str $name) {
    return "Hallo {$name}!";
  }
}

# Dann:
my Foo $obj .= new;
$obj.bar = 42;
say hallo $obj: "Ingo";
```

Rollen

```
role Logger::Mail {
  method log(Str $message) {...}
}

role Logger::Logfile {
  method log(Str $message) {...}
}

class NormaleKlasse {...}

my NormaleKlasse $normales_obj .= new(...);
$normales_obj does Logger::Mail;
$normales_obj.log(...);
```

Rollen

```
role Logger::Mail {
  method log(Str $message) {...}
}

role Logger::Logfile {
  method log(Str $message) {...}
}

class NormaleKlasse does Logger::Mail {...}

my NormaleKlasse $normales_obj .= new(...);

$normales_obj.log(...);
```

Mitgestaltungsmöglichkeiten

- Mailingliste:
  `perl6-language@perl.org`,
  `gmane.comp.lang.perl.perl6.language`
- IRC: `#perl6` auf Freenode
- Auch Newbies gern gesehen
- Viele Beteiligungsmöglichkeiten, nicht nur für Programmierer
- Weitere Informationen: `http://dev.perl.org/perl6/`

„Perl 6 ist ja schön und gut, aber das dauert doch noch Jahre, bis es fertig ist!“

- Nur tote Produkte sind „fertig“.
- Parrot steht bereits.
- Seit dem 1. Februar gibt es nun auch einen Perl 6-Compiler.

Pugs

- Ursprünglich Haskell-Projekt von Autrijus Tang „als Übung“
- Projektbeginn: 1. Februar 2005
- Nun fast 100 Entwickler
- Version 6.2.2: Alles außer Objektorientierung (!)

Entwicklung

- „Test-driven development“ –
- Camelfolks: Schreiben von Tests in Perl 6 für noch nicht implementierte Features:

```
is 23 + 42, 64, "Einfache Rechnungen funzen.";
is ~[1, 2, 3], "1 2 3",
  "Arrays wandeln sich richtig in Strings um.";
is +[1, 2, 3], 3,
  "Arrays wandeln sich richtig in Ints um.";
```

- Lambdafolks: Implementierung dieser Features
- Ergebnis der Zusammenarbeit:
  Über 4.000 funktionierende Tests

Entwicklung

- „Test-driven development“ –
- Camelfolks: Schreiben von Tests in Perl 6 für noch nicht implementierte Features:

```
is 23 + 42, 64, "Einfache Rechnungen funzen.";
is ~[1, 2, 3], "1 2 3",
  "Arrays wandeln sich richtig in Strings um.";
is +[1, 2, 3], 3,
  "Arrays wandeln sich richtig in Ints um.";
```

- Lambdafolks: Implementierung dieser Features
- Ergebnis der Zusammenarbeit:
  Über 4.000 funktionierende Tests

Weitere Pläne

Pugs 6.0	Erstes Release
Pugs 6.2	Grundlegende IO- und Kontrollflusselemente, veränderbare Variablen
Pugs 6.28	Klassen
Pugs 6.283	Rules und Grammars
Pugs 6.2831	Rollen
Pugs 6.28318	Makros
Pugs 6.283185	Portierung von Pugs von Haskell nach Perl 6
Pugs 2π	Vollendung

Beteiligungsmöglichkeiten

- Mailingliste:
  `perl6-compiler@perl.org`,
  `gmane.comp.lang.perl.perl6.compiler`
- IRC: `#perl6` auf Freenode
- Auch Newbies gern gesehen
- Schreiben von Tests (Perl 6), Implementierung (Haskell), Schreiben von Dokumentation, Portierung von Perl 5—Python—Ruby—. . .-Modulen nach Perl 6, . . .
- Weitere Informationen: `http://www.pugscode.org/`

Ausblick

- Perl 6 ist verdammt cool. :)
- Parrot steht bereits und bringt einiges an Geschwindigkeit für viele Sprachen.
- Dank Pugs kann man schon genau jetzt in Perl 6 programmieren.