

Perl 6 für Perl 5-Programmierer



Moritz Lenz

moritz.lenz@gmail.com

2016-12-01 – 2016-12-02

Erwartungen an die Schulung?

Über den Schulungsleiter

- Maintainer der Perl 6 Testsuite (2011-2013)
- Initiator des Perl 6 Documentation Projects (<https://doc.perl6.org>)
- Autor mehrerer iX-Artikel über Perl 6
- Software-Entwickler und Architekt (Perl 5 und Python)

Themen

- Perl 6 Basics
- Wichtige Typen (Zahlen, Strings, Container)
- Subroutinen, Multiple Dispatch, Operatoren
- Objektorientierung
- Regexes und Grammatiken
- Diverses: IO, Parallelität, Junctions, Whatever
Priming

Wo findet man Hilfe?

- <https://doc.perl6.org/>
- <http://de.perl6intro.com/>
- IRC: #perl6 auf irc.freenode.org
- perl6-users@perl.org
- Die üblichen Verdächtigen: Google, perlmonks, stackoverflow

Perl 6

- Sprachspezifikation separat vom Compiler
- Spezifikation durch Testsuite und Design Documents (<https://design.perl6.org/>)
- Früher gab es mehrere Compiler (Pugs, Niecza), jetzt Rakudo
- Es gibt eine eigene VM (MoarVM) für Rakudo, JVM und JS-Backends sind in Arbeit

Perl 6 – die Sprache

- Nicht abwärtskompatibel mit Perl 5
- ... aber Kooperation über `Inline::Perl5` möglich
- Objektorientierung im Kern der Sprache; eingebaute Datentypen sind jetzt Klassen
- Optionale Typedeklaration
- Regexes verbessert, mit Erweiterung Grammatiken.
- Parallelität mit high-level-Abstraktionen

Rakudo Perl 6

- `perl6 --version`
- `perl6 -e 'say "Hallo, Welt"'`
- `perl6 hello.p6`

Perl 6: Syntax

- Identifier: - und ' in zwischen Buchstaben erlaubt: "sub don't-hang(\$x) { ... }"
- Methodenaufrufe mit . statt ->
- Indirekter Methodenaufruf: \$obj . "\$name" ()
- Klammern direkt nach einem Identifier sind (fast) **immer** ein Funktionsaufruf
- Keine Klammern mehr um if und while-Bedingungen mehr nötig

Perl 6 Syntax: Beispiel

```
use v6;
```

```
my $x = 42;
```

```
if $x < 0 {  
    say "negative";  
}
```

```
elsif 0 <= $x < 10 {  
    say "small";  
}
```

```
else {  
    say "Not quite as small";  
}
```

Perl 6 Syntax: Beispiel

```
use v6;  
  
my $x = 42;  
  
if $x < 0 {  
    say "negative";  
}  
elsif 0 <= $x < 10 {  
    say "small";  
}  
else {  
    say "Not quite as small";  
}
```

Bessere Fehlermeldung,
wenn man es aus Versehen
mit perl 5 ausführt

Perl 6 Syntax: Beispiel

```
use v6;
```

```
my $x = 42;
```

```
if $x < 0 {  
    say "negative";  
}
```

```
elsif 0 <= $x < 10 {  
    say "small";  
}
```

```
else {  
    say "Not quite as small";  
}
```

Keine Klammern
um die Bedingung

Perl 6 Syntax: Beispiel

```
use v6;  
  
my $x = 42;  
  
if $x < 0 {  
    say "negative";  
}  
elsif 0 <= $x < 10 {  
    say "small";  
}  
else {  
    say "Not quite as small";  
}
```

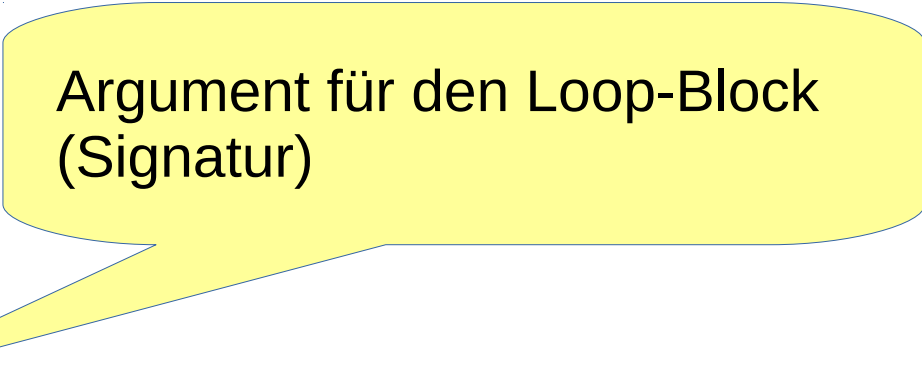
Chained Conditionals, wird
wie `0 <= $x && $x < 10`
ausgewertet

Perl 6 Syntax: Loops

```
use v6;
```

```
my @numbers = 1..9;
```

```
for @numbers -> $n {  
    say $n if $n.is-prime;  
}
```



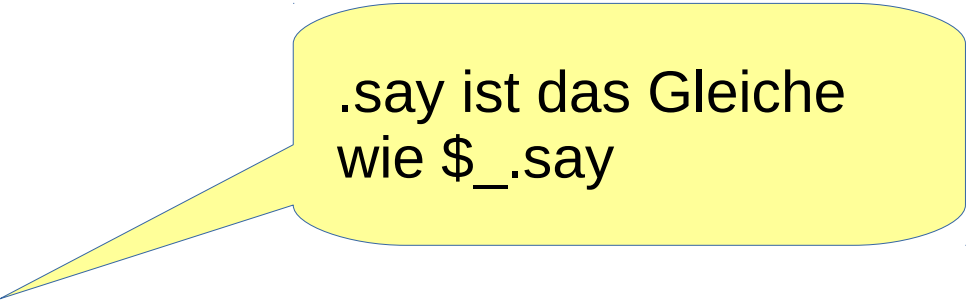
Argument für den Loop-Block
(Signatur)

Perl 6 Syntax: Loops

```
use v6;
```

```
my @numbers = 1..9;
```

```
for @numbers -> $n {  
    say $n if $n.is-prime;  
}
```



.say ist das Gleiche
wie \$_.say

```
.say for @numbers.grep:  
-> $n { $n.is-prime }
```

Perl 6 Syntax: () sind ein sub-Aufruf

- `foo()`: rufe sub foo auf
- `if($x){}`: rufe sub if auf
- `qw(...)`: rufe sub qw auf

Perl 6 Syntax: () nach Identifier

- `foo()`: rufe sub `foo` auf
- `if($x){}`: rufe sub `if` auf
- `qw(a b c)`: rufe sub `qw` auf

Stattdessen:

- `if $x { ... }`
- `<a b c>` oder `qw/a b c/` oder `qw<a b c>`

Perl 6 Syntax: Sigils

- Sigils sind invariant: Zugriff auf Element von my @a mit @a [0]
- %hash{ expression() } und %hash<literal>
- &mysub ist eine Variable, mysub und mysub () sind Aufrufe
- my \$scalar = @array; automatische Referenz, nicht Anzahl der Elemente (.elems)

Übung: Newton's Methode für sqrt()

- Iterative Methode, um die Wurzel von x zu berechnen
- $r_0 = x$
- $r_{n+1} = (r_n + x / r_n) / 2$
- Berechne r_5 für $x=8$, vergleiche mit $\text{sqrt}(8)$

Operatoren-Reform: Infix

.	Methoden-Aufruf
~	Strings zusammenführen
~~	Smart Match
+& + +^	AND, OR, XOR numerisch bitwise
~& ~ ~^	AND, OR XOR string bitwise
?& ? ?^	AND, OR, XOR logisch

Smart Matching

Rechte Seite	Bedeutung
Zahl	Numerischer Vergleich
String	String-Vergleich
Typ	Typ-Check
Regex	Matcht die Regex den String?
Code	Liefert der Code True zurück?

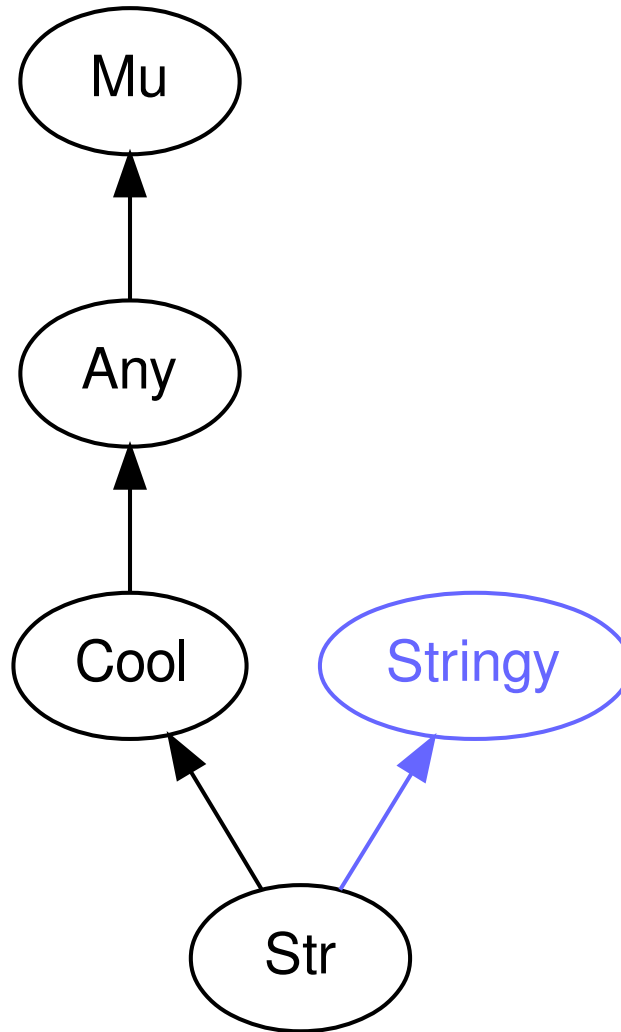
Operatoren-Reform: Prefix

~	String-Contet
+	Numerischer Context
?	Logischer/Boolean Context

Operatoren-Reform: Noch mehr

cond ?? v1 !! v2	Ternary Operator
x	String-Wiederholung
xx	Listen-Wiederholung

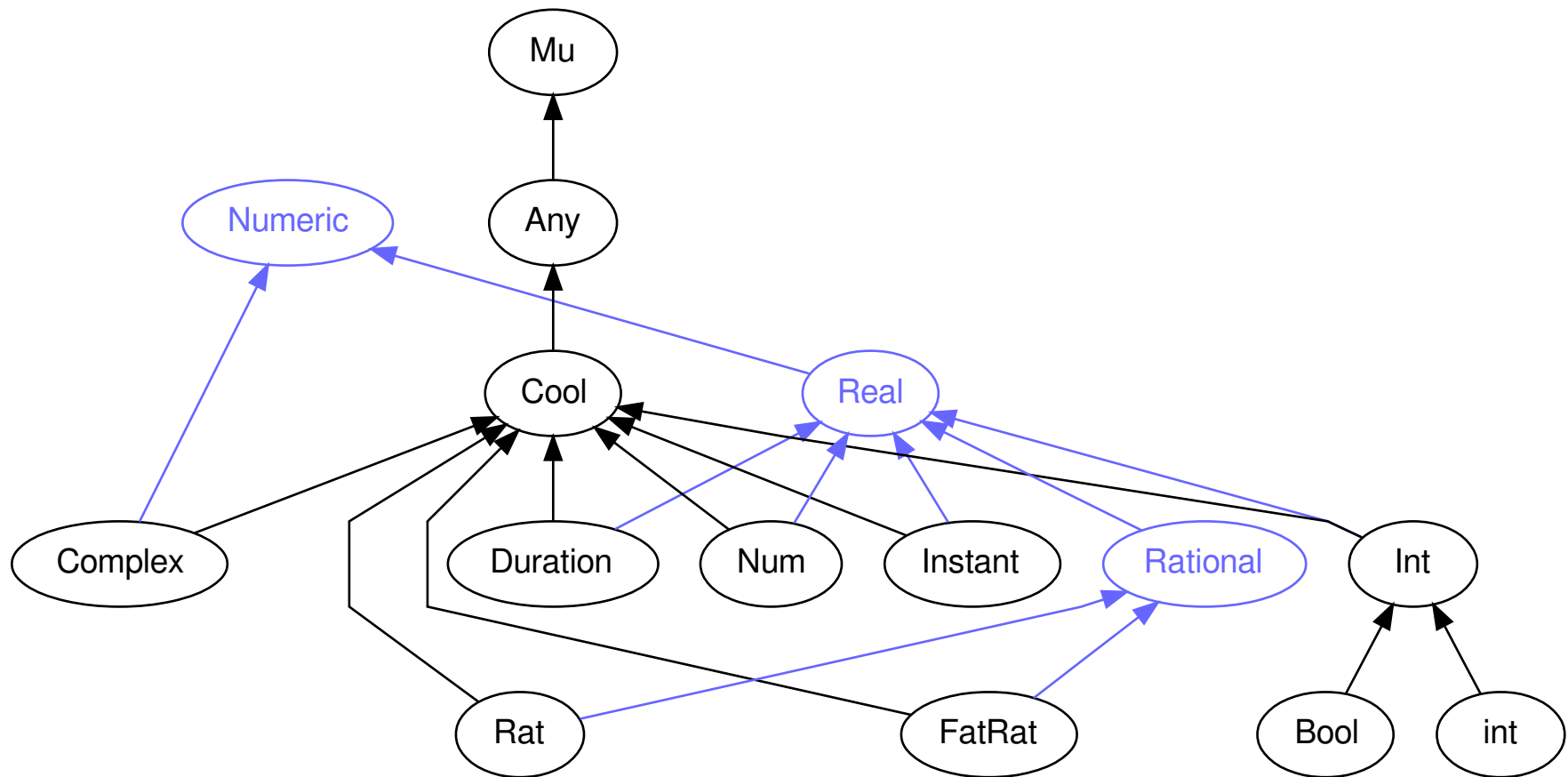
Typen: Strings



Typen: Strings

- Text, keine Binärdaten (-> Blob, Buf)
- Grapheme-basiert
- length => chars
- Neue Methoden: words, lines, wordcase, comb, starts-with, ends-with, indent, trim, trim-leading, trim-trailing, uniname

Typen: Numerisch



Typen: Numerisch

- Int: Big Integer, .say if (2**\$_ - 1).is-prime for ^100
- Rat: Bruchzahl, die bei Überlauf auf Num zurückfällt: 0.1 + 0.2 == 0.3
- FatRat: Brüche ohne Überlauf
- Num: Floating point-Zahlen, 3.01e308
- Complex: Zahlen aus Real- und Imaginärteil (Num)

Rat -> Num Überlauf

```
my $x = 8;
```

```
my $r = $x;
```

```
for 1..6 {  
    $r = ($r + $x / $r) / 2;  
    say $r.^name, ': ', $r.perl;  
}
```

Rat: 4.5

Rat: <113/36>

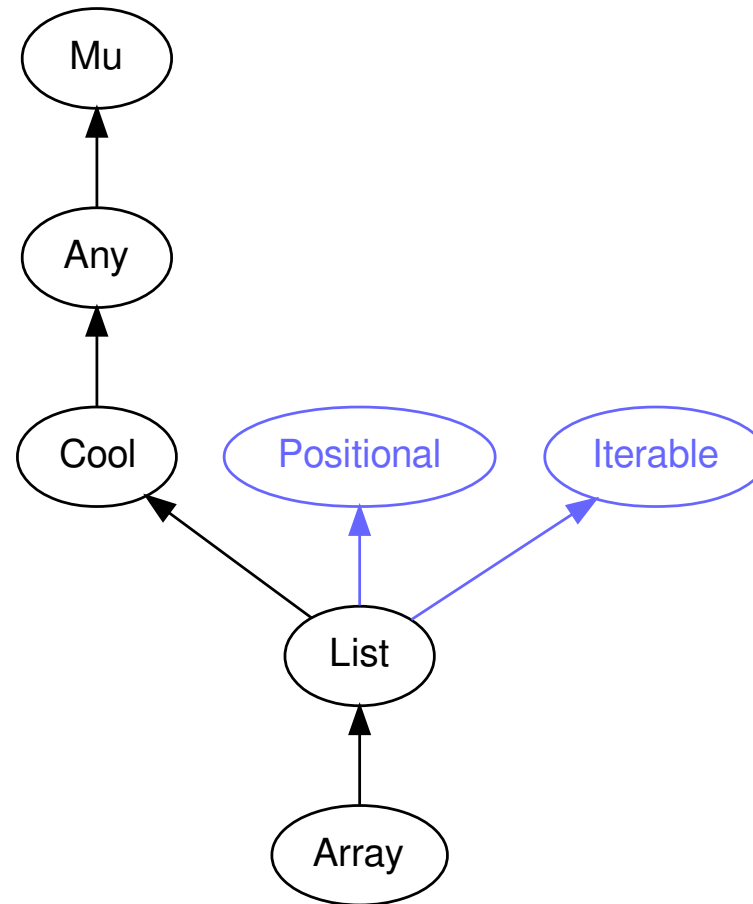
Rat: <23137/8136>

Rat: <1064876737/376485264>

Rat: <2267891697076964737/801820798913807136>

Num: 2.82842712474619e0

Typen: Array/List



Array/List: Methoden

- grep, map, reduce, first, permutations, combinations, head, tail, roll, pick, unique, rotor

Array/List: Methoden

- grep, map, reduce, first, permutations, **combinations**, head, tail, roll, pick, unique, rotor

```
say .join('|') for <a b c>.permutations
# a|b|c
# a|c|b
# b|a|c
# b|c|a
# c|a|b
# c|b|a
```

Array/List: Methoden

- grep, map, reduce, first, permutations, **combinations**, head, tail, roll, pick, unique, rotor

```
say .join('|') for <a b c>.combinations(2);  
# a|b  
# a|c  
# b|c
```


Array/List: Methoden

- grep, map, reduce, first, permutations, combinations, head, tail, **roll**, **pick**, unique, rotor

```
> say ("a".. "z").pick(5); # keine Duplikate  
(l i n p e)  
> say ("a".. "z").roll(5); # Duplikate  
(m a b o m)  
> say (1..10).pick(*)  
(4 3 5 9 1 6 7 8 10 2)
```

Array/List: Methoden

- grep, map, reduce, first, permutations, combinations, head, tail, roll, pick, unique, **rotor**

```
> say ('a'..'h').rotor(3).join('|');  
a b c|d e f
```

```
> say ('a'..'h').rotor(3, :partial).join('|');  
a b c|d e f|g h
```

```
> say ('a'..'h').rotor(3 => -1).join('|');  
a b c|c d e|e f g
```

Array/List: push vs. append

```
my @a = 1, 2;  
my @b = 3, 4;
```

```
@a.push(@b);  
dd @a;          # Array @a = [1, 2, [3, 4]]
```

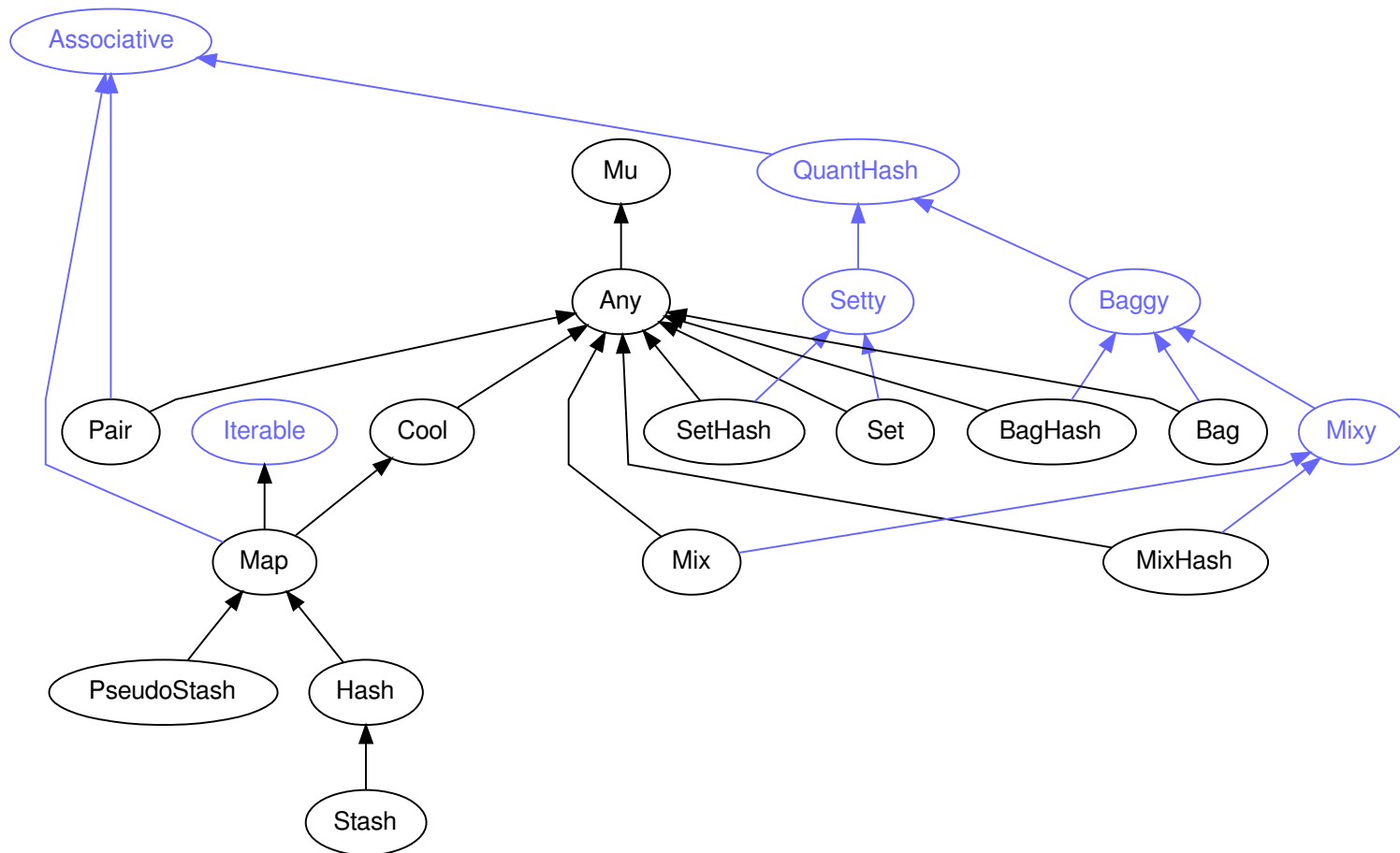
```
@a.append(@b);  
dd @a;          # Array @a = [1, 2, 3, 4]
```

```
# vorne: unshift / prepend
```

Übung: Passwort-Generator

- Schreibe ein Programm, das zufällige Passwörter erzeugt
- Zwischen 8 und 12 Zeichen
- Ein Sonderzeichen
- Mindestens je ein Groß und Kleinbuchstabe und eine Ziffer

Typen: Hash/Set/Bag



Typen: Hash/Set/Bag

Unveränderbar	Veränderbar	Aufgabe
Map	Hash	Key => Wert
Set	SetHash	Key => Existenz (Bool)
Bag	BagHash	Key => Zählen (Real)

Typen: Hash

```
use v6;
```

```
my %h = a => 1, b => 2;
```

```
my $key = 'a';
```

```
say %h{ $key };
```

```
say %h<a>;
```

```
say %h<a b>;
```

Bag: Dinge Zählen

- Wie häufig kommt "a" in "abracadabra" vor?

```
$ perl6 -e 'say bag("abracadabra".comb)<a>'  
5
```


Bag: Selbst zählen

```
use v6;
```

```
my %bag := bag( open($?FILE).comb );
```

```
for %bag.pairs.sort({ - .value }) -> $p {  
    say $p.key, "\t", $p.value;  
}
```

BagHash: In %-Variable speichern

```
my @chars = "abracadabra".comb;  
my %b := BagHash.new( @chars );  
say %b;  
    # BagHash.new(a(5), c, r(2), b(2), d)  
  
%b<d>--;  
say %b;  
    # BagHash.new(a(5), c, r(2), b(2))
```

Set und Bag: Operatoren

ASCII-Op	Unicode-Op	Bedeutung
(elem)	\in	Ist Element in?
(\leq)	\subseteq	Teilmenge?
($<$)	\subset	Strikte Teilmenge?
($<+$)	\supseteq	Bag: Teilmenge inkl. Gewichtung
($ $)	\cup	Vereinigung
($\&$)	\cap	Schnittmenge
($-$)	\setminus	Differenz

Subroutinen

```
sub distance(Int $x1, Int $y1,  
             Int $x2, Int $y2) {  
    return sqrt ($x2-$x1)**2 + ($y2-$y1)**2;  
}  
say distance 2, 0, 6, 3;    # 5
```

Subroutinen: Default-Werte

```
sub logarithm($x, $base = e) {  
    $x.log / $base.log  
}  
say logarithm 8, 2;           # 3  
say logarithm 5;             # 1.6094379124341
```

Subroutinen: Optionale Argumente

```
sub check-password(Str $password,  
                  Str $user?) {  
    if $user.defined  
        && $password.index($user).defined {  
        die "You cannot use your username in your  
password";  
    }  
    ...  
}
```

Subroutinen: Benannte Argumente

```
sub myjoin($sep, *@elems) {  
    my $result = @elems.shift;  
    for @elems -> $e {  
        $result ~= $sep ~ $e;  
    }  
    return $result;  
}  
say myjoin ' ', 'a', 'b', 'c';    # a,b,c
```

Subroutinen: r/o-Argumente

```
sub nowrite($x) {  
    $x = 42;      # Cannot assign to a readonly  
                  # variable or a value  
}
```


Subroutinen: r/w-Argumente

```
sub swap($x is rw, $y is rw) {  
    ($x, $y) = ($y, $x);  
}
```

```
my ($a, $b) = 1..2;  
swap $a, $b;  
say $a, " ", $b;      # 2 1
```

Subroutinen: Benannte Argumente

```
# Optional per Default, mit
# ! nicht mehr optional
sub draw-circle(:$cx = 0, :$cy = 0, :$radius!) {
    say qq[<circle cx="$cx" cy="$cy"
              r="$radius"/> ];
}
draw-circle radius => 10, cy => 5;
draw-circle :radius(42), :cy(5);

my $radius = 42;
draw-circle :$radius;    # wie radius => $radius
```

Beliebige benannte Argumente

```
sub named(*%all) {  
    for %all.keys.sort -> $k {  
        say $k, "\t", %all{$k};  
    }  
}
```

```
named x => 'y', foo => 'bar';
```

foo	bar
x	y

Subtypen

```
subset Positive of Real where {  $\$_$   $>$  0 };  
sub sqrt(Positive  $\$x$ ) { ... }
```

oder

```
sub sqrt(Real  $\$x$  where  $\$x$   $>$  0) {  
    ...  
}
```

Type Smilies

- `sub f(Int $x) { ... }` erlaubt auch das Typen-Objekt `Int`, nicht nur Instanzen vom Typ `Int`
- `sub f(Int:D $x) { ... }` schränkt es auf Instanzen ein
- `:U` für Typen-Objekte (`U = Undefined`), `:_` für explizit alles (`Int:_ === Int`)

Subroutinen: MAIN

- Argumente von der Kommandozeile
- `sub MAIN(Int $x, Bool :$force)`
 `{ ... }`
- `perl6 myscript --force 42`

Übung Subroutinen

- Schreibe eine binäre Suche, die ein sortiertes Array von Zahlen und eine Suchzahl akzeptiert
- Falls die Suchzahl im Array vorkommt, soll der Index zurückgegeben werden, sonst -1
- Teile das Array in der Mitte, vergleiche den Wert in der Mitte mit der Suchzahl. Benutze das Ergebnis, um in der richtigen Hälfte des Arrays weiter zu suchen. Wiederhole bis fertig.

Multiple Dispatch

- Ein Name
- Mehrere Implementationen mit verschiedenen Signaturen

Multis: Anzahl der Argumente

```
multi sub log($number) { ... }  
multi sub log($number, $base) { ... };
```

```
log(42);           # log($number)  
log(42, 2);        # log($number, $base)
```

Multis: Typ der Argumente

```
multi sub identify(Int $number) {  
    say "Int $number";  
}  
multi sub identify(Str $str) {  
    say "Str $str";  
}  
multi sub identify(@array) {  
    say "Array @array[]";  
}  
identify 1;                # Int 1  
identify 'Perl';           # Str Perl  
identify [1, 'x'];         # Array 1 x
```

Multis: Benannte Argumente

```
multi sub go($destination, :$car!) { ... }  
multi sub go($destination, :$walk!) { ... }  
multi sub go($destination, :$train!) { ... }  
  
go 'home', :train;
```

Übung: to-json

- Erzeuge vereinfachtes JSON: Array, String, Int Objekte/Hashe
- Kein String Escaping

```
say to-json([1, [2, {a => 'b'}]]);  
# [1,[2,{"a":"b"}]]
```

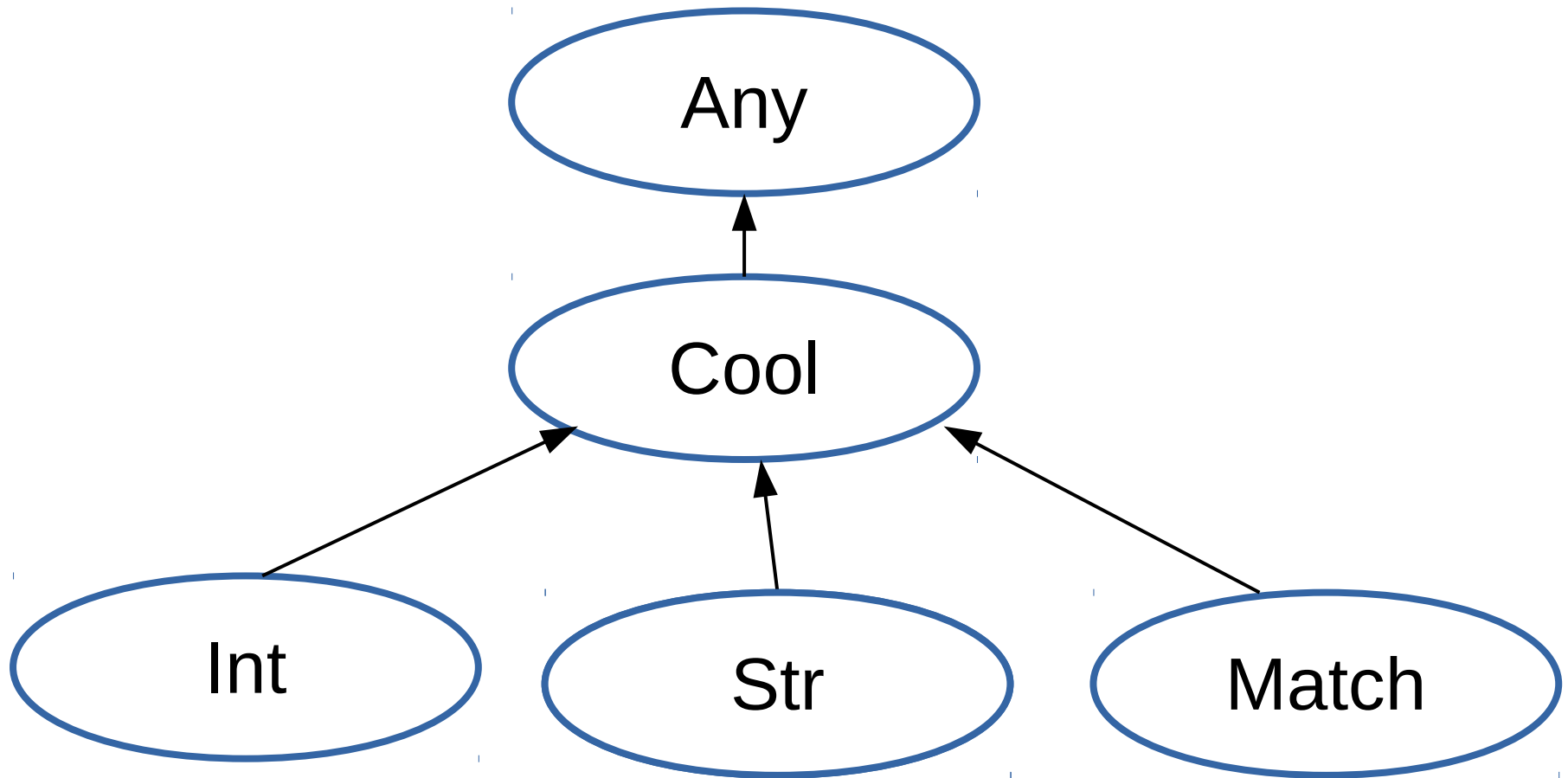
Wie funktionieren Multis?

```
multi a(Int) { say 1 }  
multi a(Str) { say 2 }  
multi a(Any) { say 3 }
```

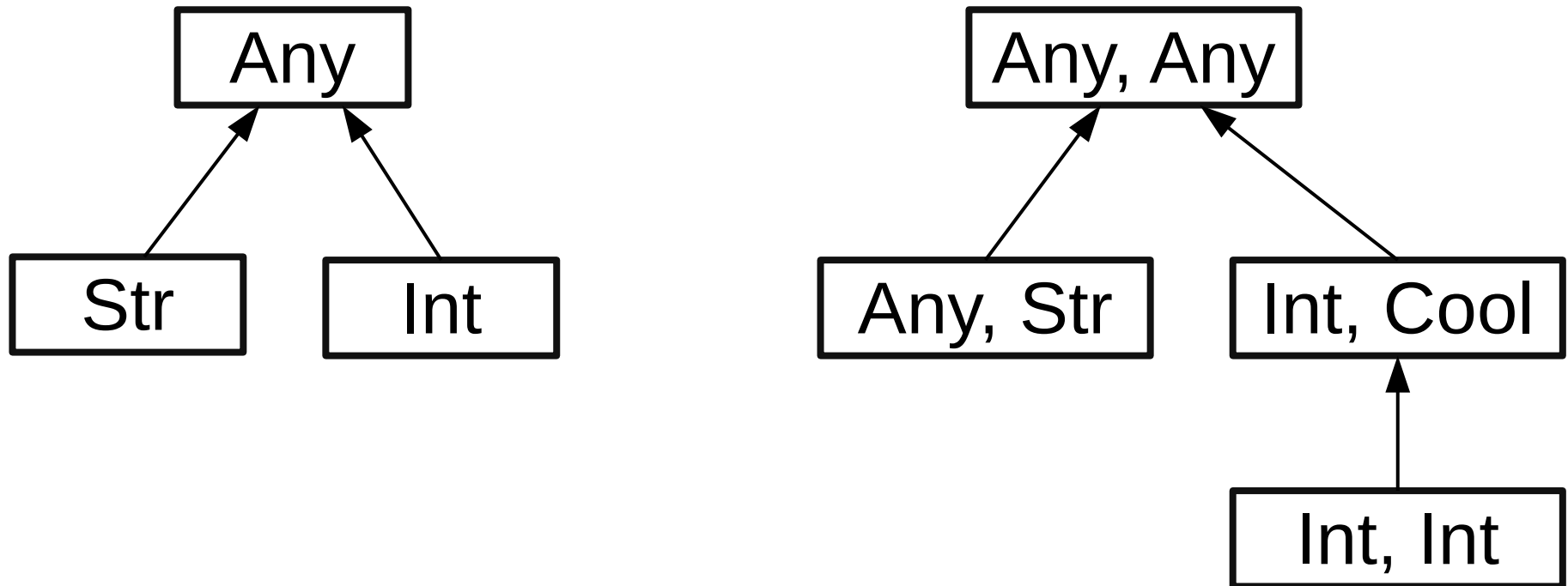
```
multi a(Any, Any) { say 4 }  
multi a(Any, Str) { say 5 }  
multi a(Int, Int) { say 6 }  
multi a(Int, Cool) { say 7 }
```

```
a 2, 'x'
```

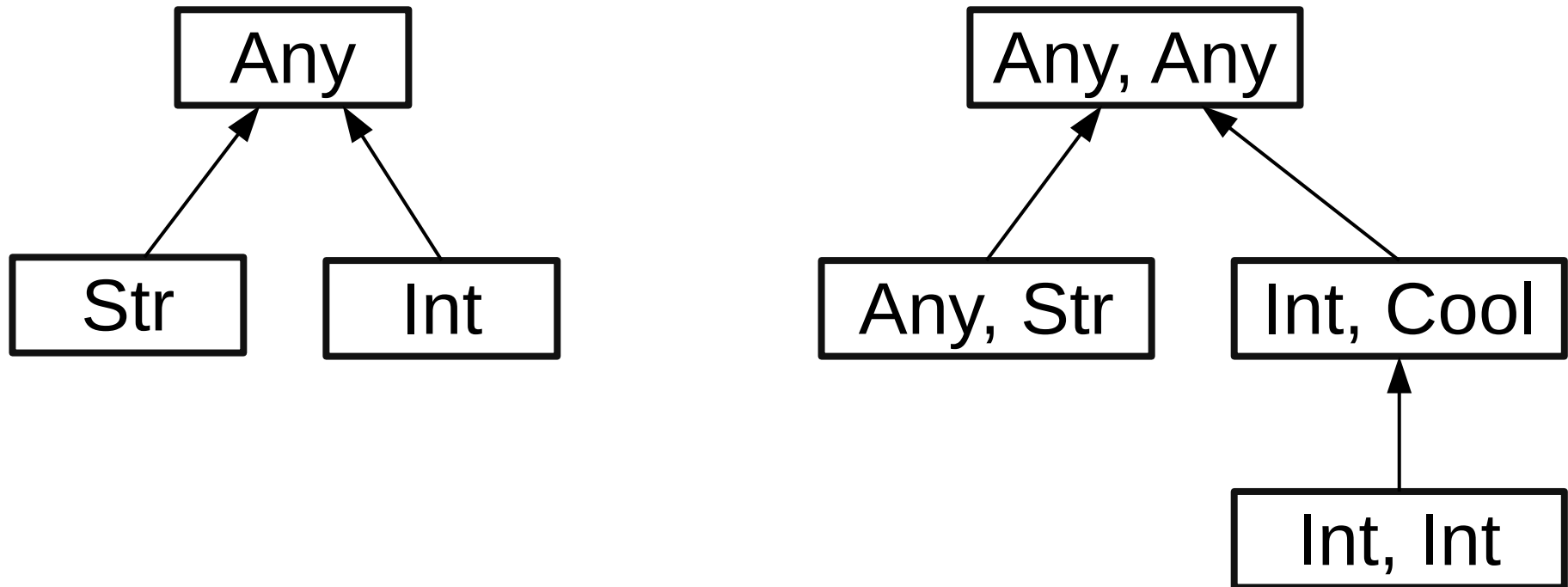
Multis: Zusammenhang Typen



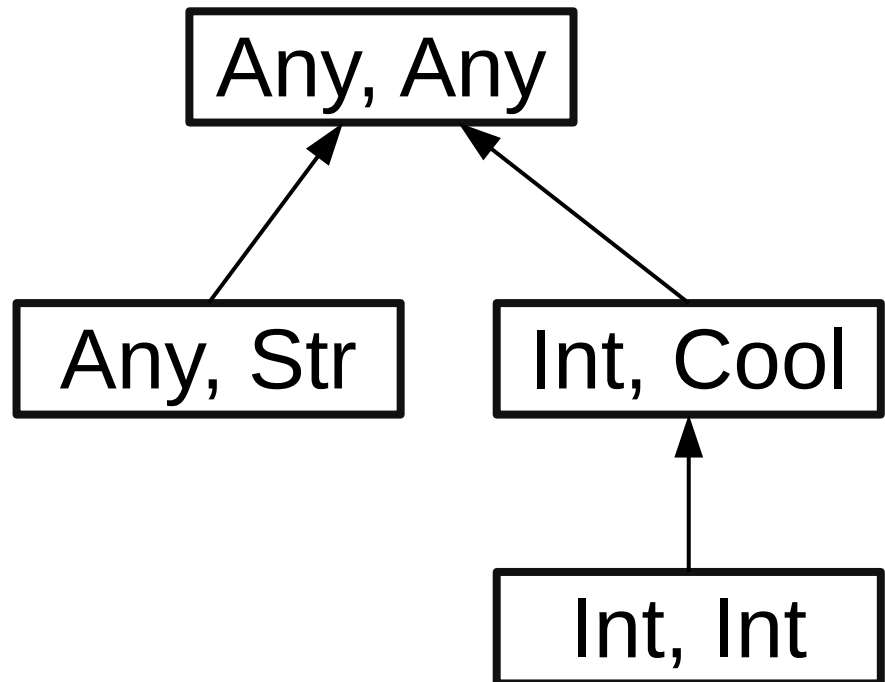
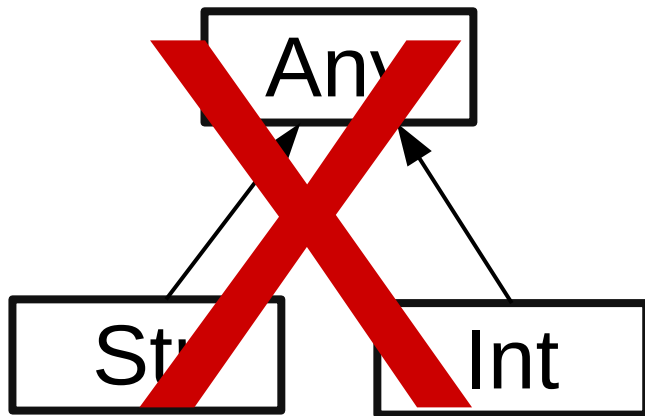
Signaturen topologisch sortiert



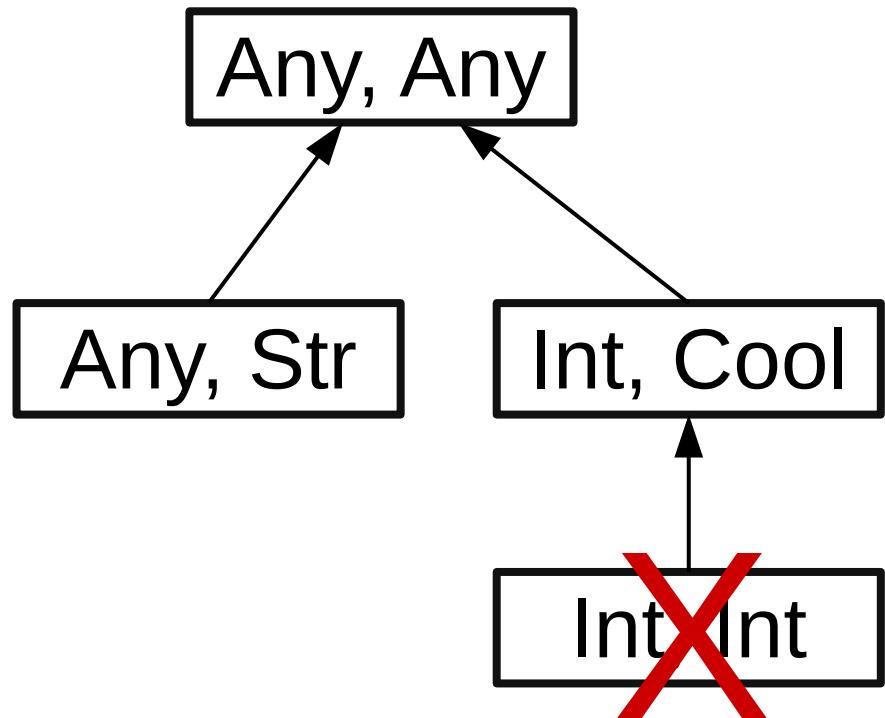
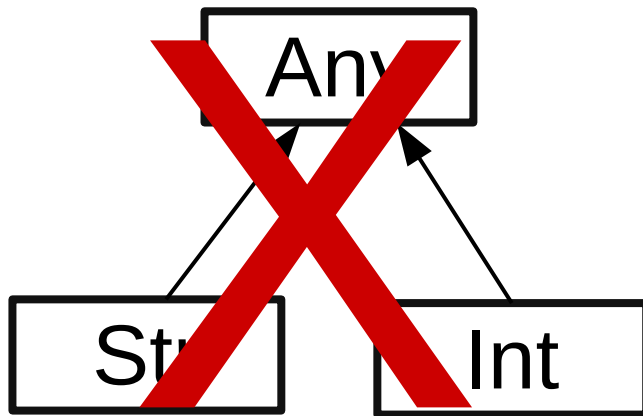
Aufruf mit (Int, Match)



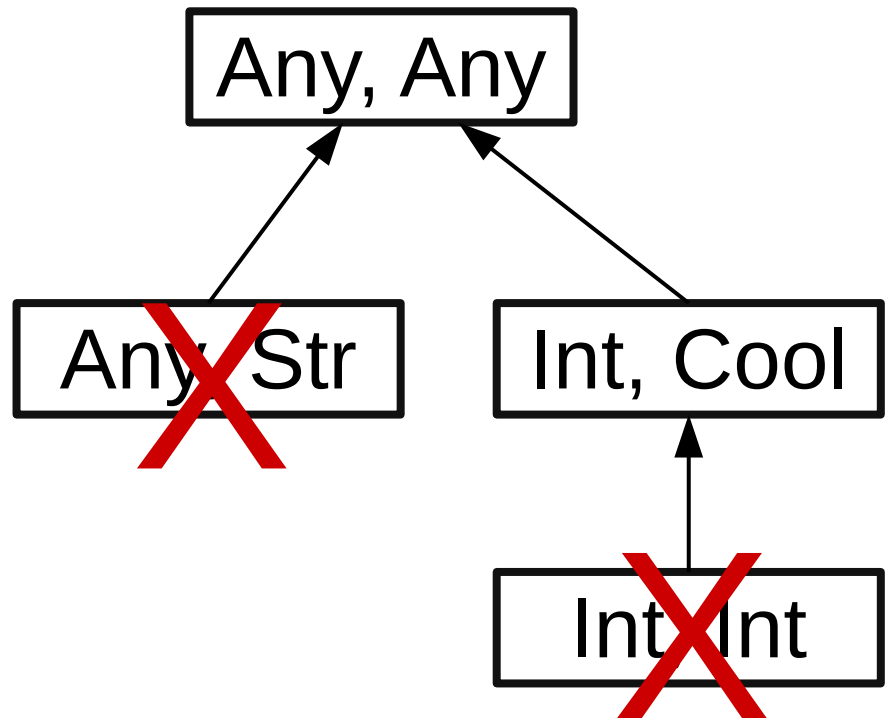
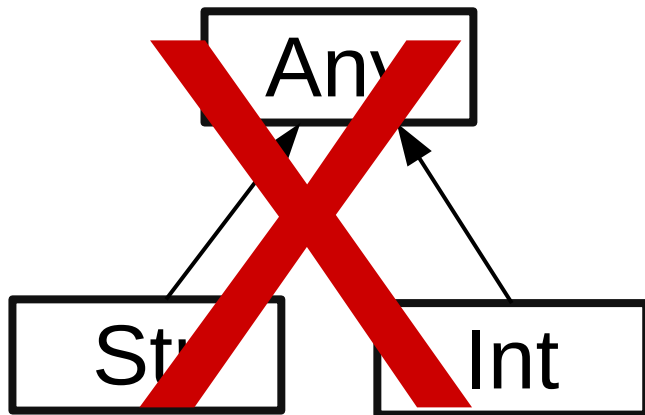
Aufruf mit (Int, Match)



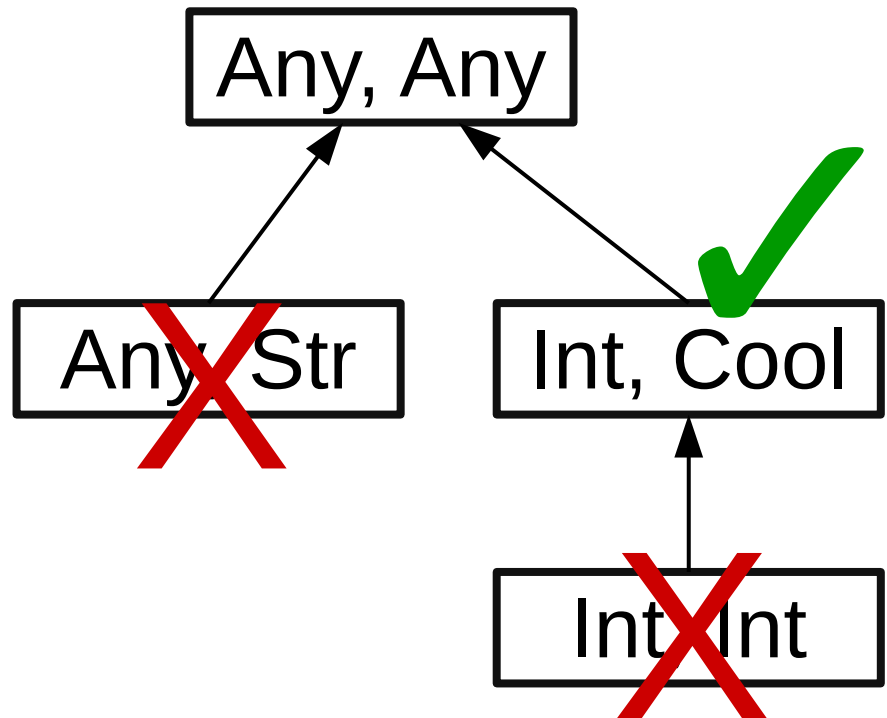
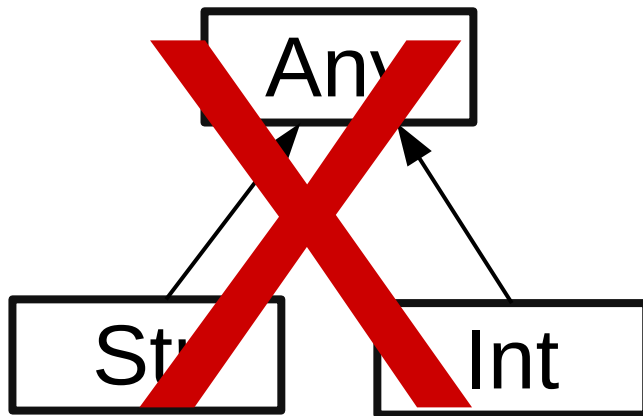
Aufruf mit (Int, Match)



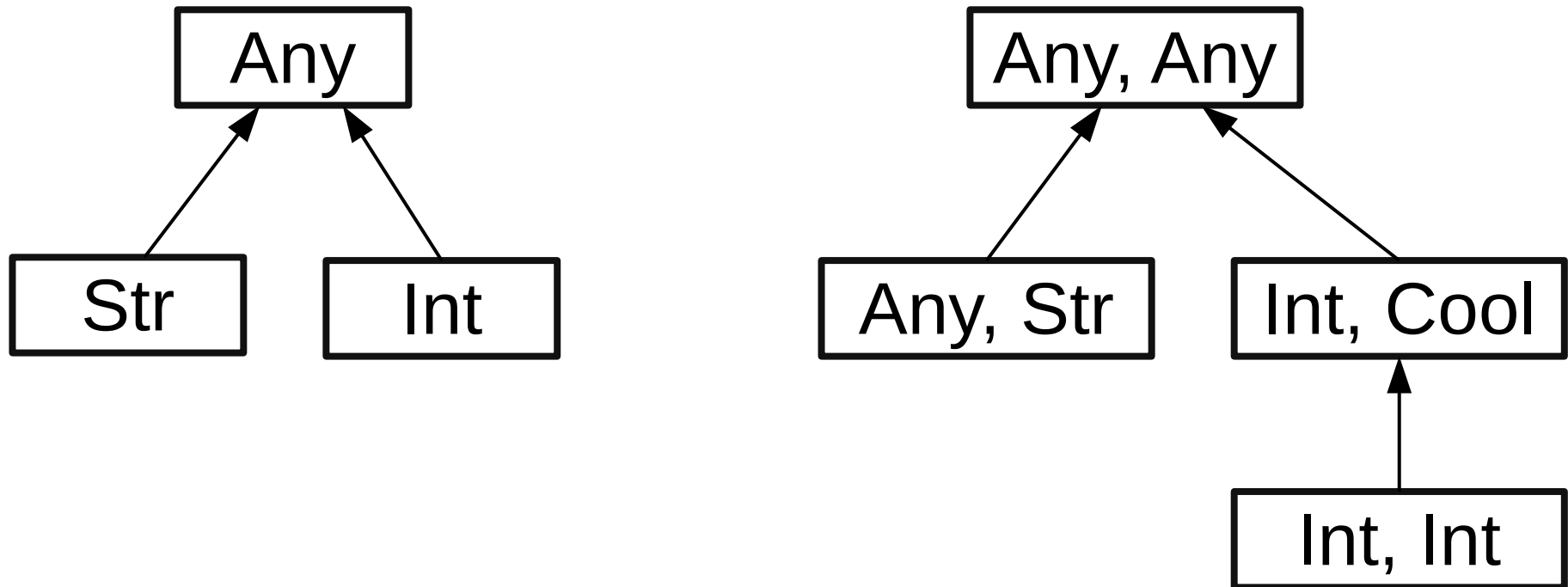
Aufruf mit (Int, Match)



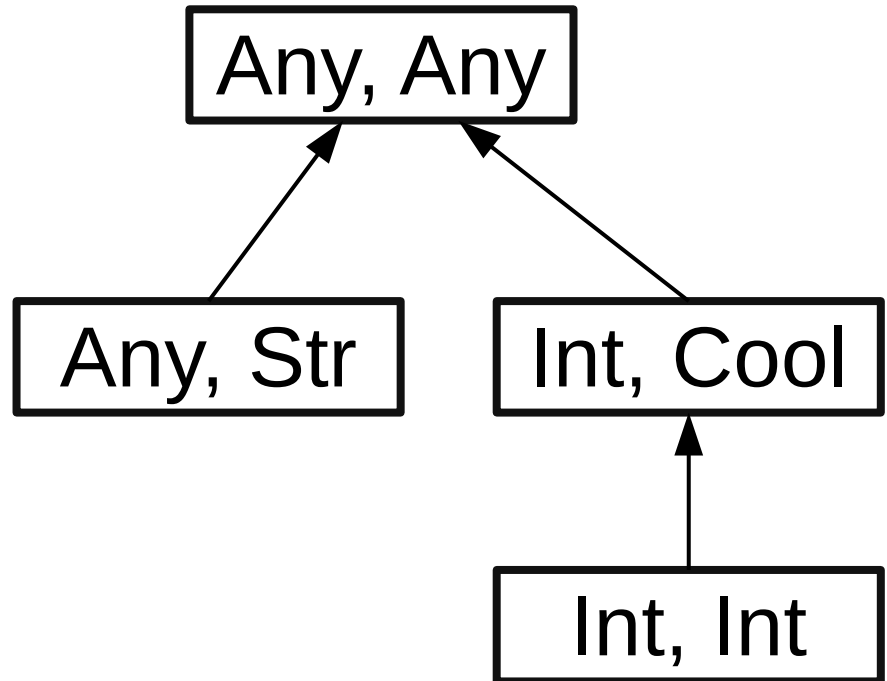
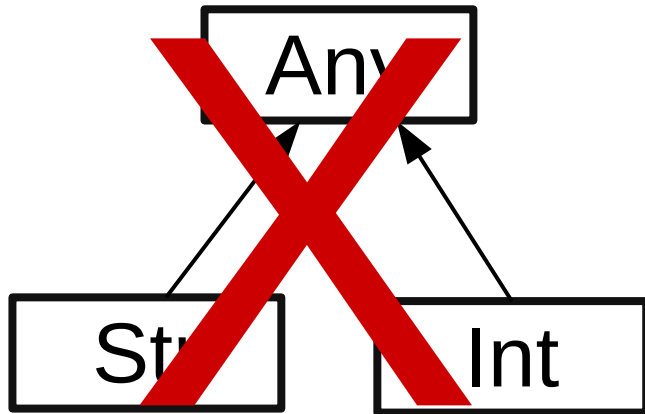
Aufruf mit (Int, Match)



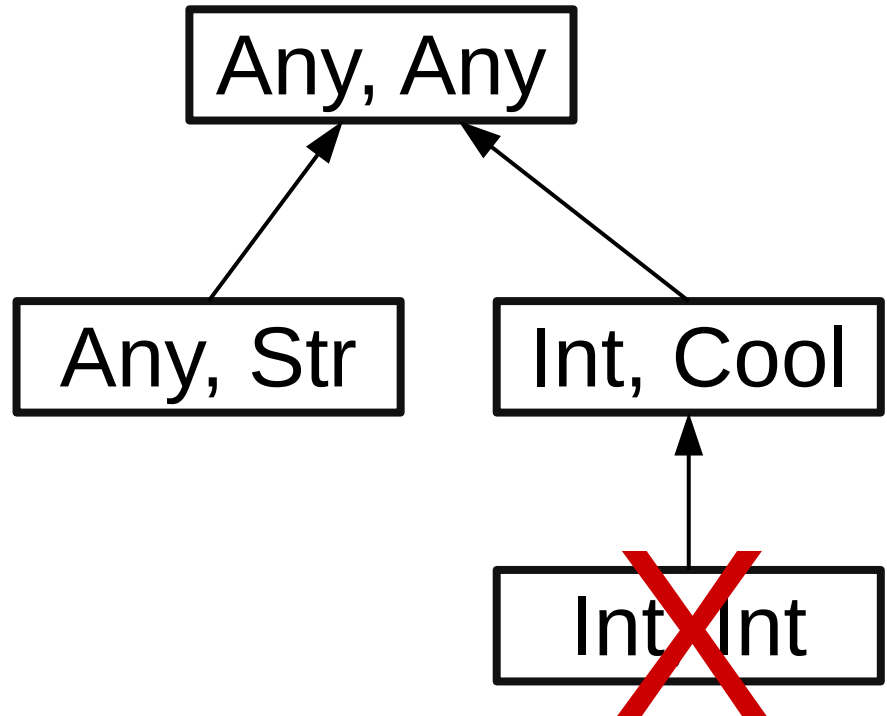
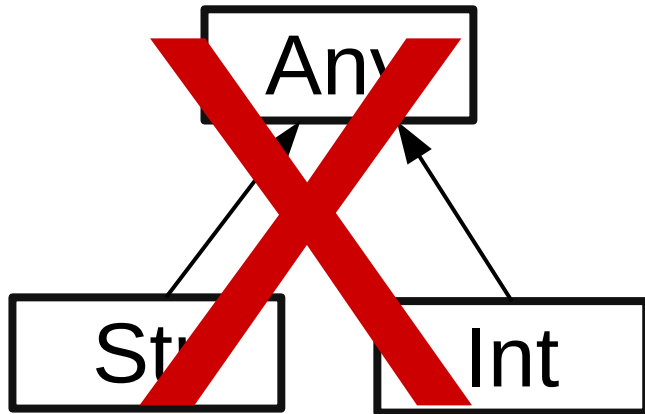
Aufruf mit (Int, Str)



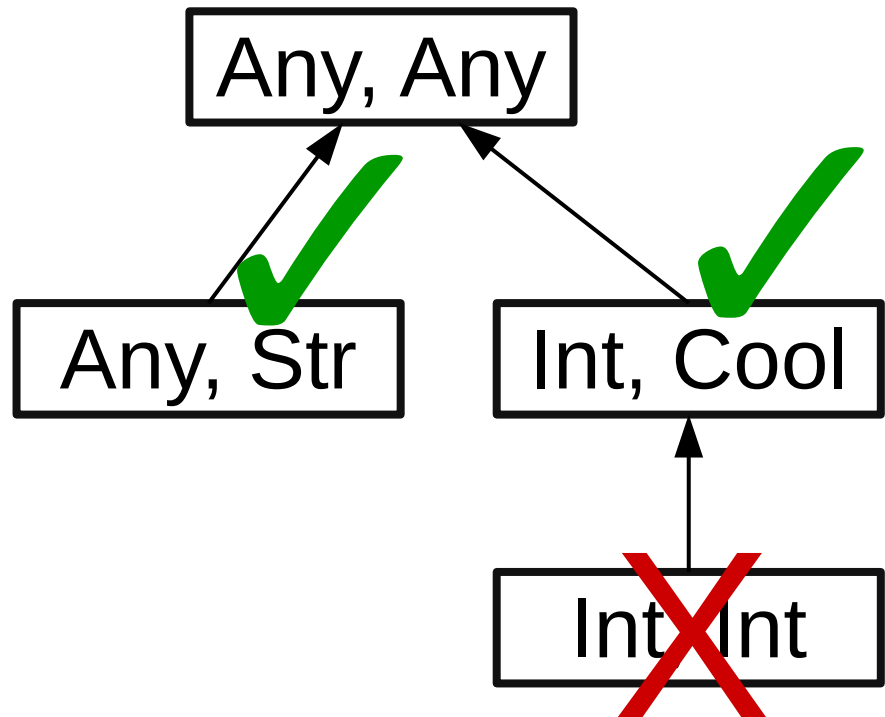
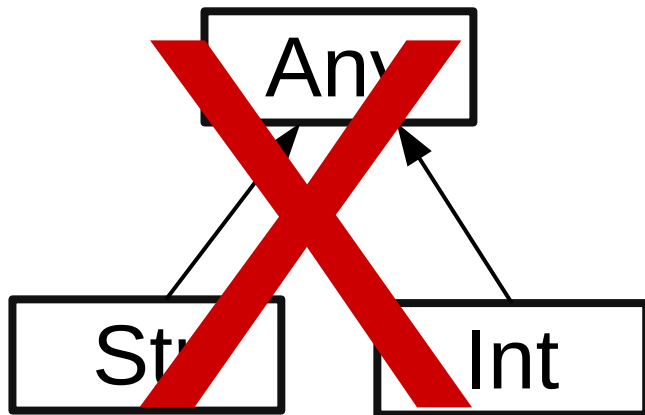
Aufruf mit (Int, Str)



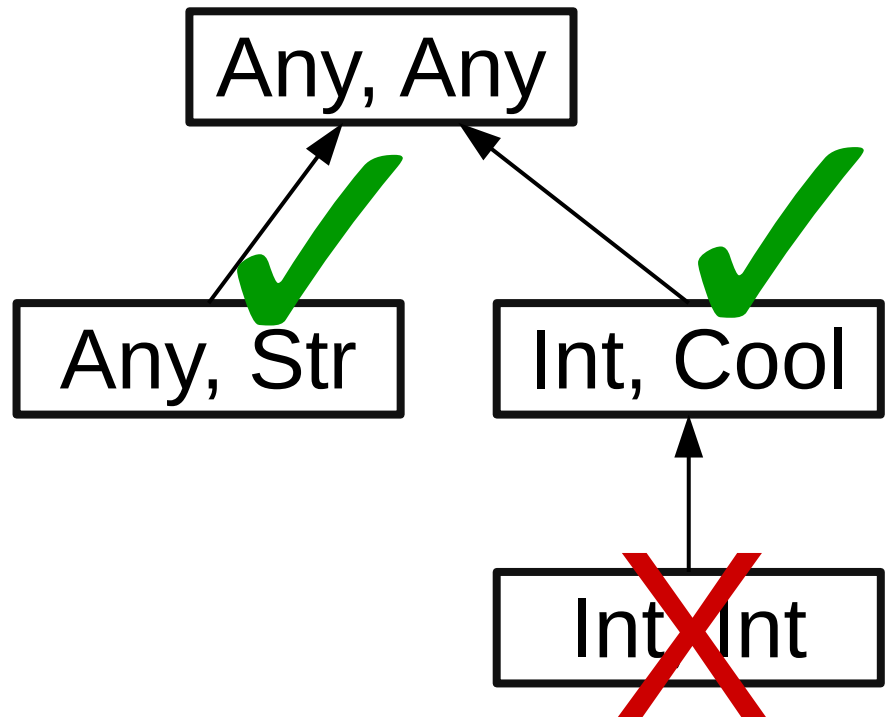
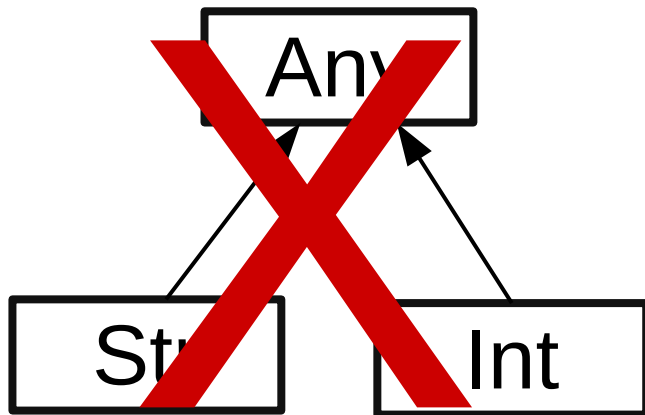
Aufruf mit (Int, Str)



Aufruf mit (Int, Str)



Aufruf mit (Int, Str)



Und jetzt? mögliche Verfeinerungen

- Benannte Parameter
- where-Constraints
- `is default`

Wenn es keine gibt: "Ambiguous Dispatch"

Operatoren

- Operatoren sind Subroutinen mit speziellen Namen
- `+` ist implementiert in `infix:<+>`
- Können auch Multiple Dispatch
- Lexikalisches Scoping, wie Subroutinen

Operatorentypen

Name	Bedeutung
prefix	Vor dem Ausdruck
infix	zwischen Ausdrücken
postfix	Nach einem Ausdruck
circumfix	Um einen Ausdruck herum
postcircumfix	Nach einem Ausdruck, um einen anderen herum (z.B. <code>\$foo[bar]</code>)

Operator überladen

```
class Vector2D {  
  has Real $.x;  
  has Real $.y;  
}  
  
multi sub infix:<*>(Vector2D $a, Real $b) {  
  Vector2D.new(  
    x => $a.x * $b,  
    y => $a.y * $b,  
  );  
}  
  
say Vector2D.new(x => 1, y => 2) * 2;  
# Vector2D.new(x => 2, y => 4)
```

Eigene Operatoren

```
sub postfix:<!>(Int $x where $x >= 0) {  
    [*] 1...$x;  
}
```

```
say 10!;      # 3628800
```

```
sub prefix:<loud>($x) {  
    uc $x;  
}
```

```
say loud 'hello', 'world';  # HELLOworld
```

Multi Dispatch: proto

- Multi = proto + Kandidaten
- proto wird implizit erzeugt
- Man kann auch explizit einen proto schreiben, und mit `{*}` an die Kandidaten delegieren
- Gut für gemeinsame Aufgaben wie Argumente aufbereiten, Logging, Fehlerbehandlung, Konfiguration nachladen.

Multi Dispatch: proto

```
proto sub get($protocol, $url, *%) {  
    say "Getting $url via $protocol";  
    my $result = try {  
        {*};  
    }  
    if $! {  
        say "Got error '$!' for $url"  
    }  
    return $result;  
}  
multi sub get('http', $url) { say "http $url" }  
multi sub get('ftp', $url) { say "ftp $url" }  
  
get('http', "http://perl6.org/");
```


Übung: eigene Operatoren

- Schreibe einen Präfix-Operator `twice`, der einen String doppelt schreibt und eine Zahl numerisch verdoppelt
- Schreibe einen Infix-Operator `+~` der zwei Zahle addiert und einen kleinen Fehler einfügt (`$base.rand - 0.5 * $base`)

Meta-Operatoren

Operatoren, die anderen Operatoren
modifizieren

Meta-Operatoren: Reduce

```
my @numbers = 1, 2, 7;  
say 1 + 2 + 7;           # 10  
say [+] @numbers;        # 10  
  
say [+] ();              # 0  
say [*] ();              # 1  
  
say [\+] @numbers;       # (1 3 10)
```

Meta-Operatoren: Zip

```
say @numbers Z 1..10; # ((1 1) (2 2) (7 3))
```

```
say @numbers Z+ 1..10; # (2 4 10)
```

```
say @numbers Zxx 1..3; # ((1) (2 2) (7 7 7))
```

Meta-Operatoren: Kreuzprodukt

```
say <a b> X 1..3;  
# ((a 1) (a 2) (a 3) (b 1) (b 2) (b 3))  
say <a b c> X~ 1..3;  
# (a1 a2 a3 b1 b2 b3 c1 c2 c3)
```

Objektorientierung

- Moose ist Produkt der Perl 6-Entwicklung
- Viele gemeinsame Konzepte
- Klassen, Rollen, Objekte, Meta Object Protocol

Klassen, Attribute, Methoden

```
class Task {  
  has &.action;  
  has @.dependencies;  
  method run() {  
    .run for @.dependencies;  
    &!action();  
  }  
}  
  
my $t = Task.new(  
  action => { say 'Frühstück essen' },  
  dependencies => Task.new(  
    action => { say 'aufstehen' },  
  ),  
);  
$t.run;
```

Defaults für Attribute

```
class Rectangle {  
  has $.width;  
  has $.height = self.width;  
  method area() { $.width * $.height }  
}  
  
say Rectangle.new( width => 5 ).area;
```


Öffentliche Attribute

```
class A {  
    has $.x;  
}
```

```
class A {  
    has $!x;  
    method x() { $x };  
}
```

Öffentliche Attribute

- Private Attribute: `$!attr`
- Öffentliche Attribute: `$.attr = $!attr +`
accessor method
- Read-Only
- Schreibzugriff innerhalb der Klasse über das
private Attribut

Öffentliche Attribute: rw

```
class B {  
    has $.x is rw;  
}  
class B {  
    has $!x;  
    method x() is rw { $!x }  
}
```

Private Attribute initialisieren

```
class Encapsulated {  
  has $!private;  
  submethod BUILD(:$private) {  
    $!private = $private;  
  }  
}
```

oder kürzer:

```
class Encapsulated {  
  has $!private;  
  submethod BUILD(:$!private) {  
  }  
}
```

Eigener Konstruktor

```
class Task {  
  method new(&action, *@dependencies) {  
    self.bless(:&action, :@dependencies);  
  }  
  ...  
}  
  
my $t = Task.new(  
  { say 'Frühstück essen' },  
  Task.new( { say 'aufstehen' } ),  
);  
$t.run;
```

Rollen

- Können Attribute und Methoden enthalten
- Werden in Klassen kopiert
- Mehr Checks zur Compile-Zeit
- Parametrierbar

Rollen: Beispiel

```
role Storable {  
    method store($filename) {  
        spurt $filename, self.perl;  
    }  
}
```

```
class Account does Storable {  
    has Str $.username;  
    has Bool $.is-valid = True;  
}
```

```
my $ac = Account.new(username => 'mlenz');  
$ac.store('mlenz.p6');
```

Rollen als Interfaces

```
role Drawable {  
  method draw($canvas) {  
    ...  
  }  
}
```

Drei Punkte im Quellcode
(kein Platzhalter)

```
class Square does Drawable {  
  has ($.x, $.y, $.width, $.height);  
  method draw($canvas) {  
    my $x1 = $.x + $.width;  
    my $y1 = $.y + $.height;  
    $canvas.line($.x, $.y, $x1, $.y);  
    # ...  
  }  
}
```


Rollen als Interfaces

```
class Borken does Drawable { };
```

```
# Method 'draw' must be implemented by Borken  
because it is required by a role
```

Rollen-Konflikt

```
role Drinker {  
  method go-to-bar {  
    say 'Going to the bar to drink';  
  }  
}  
role Gymnast {  
  method go-to-bar {  
    say 'Going to the bar to exercise';  
  }  
}  
  
class DrunkenGymnast does Drinker does Gymnast {}  
  
# Method 'go-to-bar' must be resolved by class DrunkenGymnast  
# because it exists in multiple roles (Gymnast, Drinker)
```

Rollen-Konflikt auflösen

```
class DrunkenGymnast does Drinker does Gymnast {  
  method go-to-bar {  
    say 'staggering to the bar, undecided';  
  }  
}  
DrunkenGymnast.new.go-to-bar;
```

Parametrisierte Rollen

```
role BinaryTree[::T] {  
  has T $.item;  
  has BinaryTree[T] $.left;  
  has BinaryTree[T] $.right;  
  method visit-preorder(&c) {  
    c($!item);  
    $!left.visit-preorder(&c) if $!left;  
    $!right.visit-preorder(&c) if $!right;  
  }  
}
```

Parametrisierte Rollen - Benutzung

```
my $t = BinaryTree[Int].new(  
    item => 5,  
    left => BinaryTree[Int].new( item => 4 ),  
    right => BinaryTree[Int].new( item => 8 ),  
);  
$t.visit-preorder(&say);
```

Mixin: Rollen zur Laufzeit anwenden

```
role Storable {  
    method store($filename) {  
        spurt $filename, self.perl;  
    }  
}
```

```
my $x = 5;  
$x does Storable;  
$x.store('stored-int');
```

Mixin mit anonymer Rolle

```
my $y = 5;  
$y does role {  
    method square { self * self }  
};  
say $x.square;
```

Regexes und Grammatiken

Regexes: Neue Syntax

- Whitespace und Kommentare insignifikant (/x by default)
- Buchstaben und Zahlen: Literale
- Alles andere: metasyntaktisch
- Ein Backslash dreht es herum
- Quotes ' . . . ' und " . . . " funktionieren wie ausserhalb von Regexes
- Matching mit ~~ (smart match)

Quoting in Aktion

```
say 'a,c' ~~ /a,c/;
```

Unrecognized regex metacharacter , (must be quoted to match literally)

```
say 'a,c' ~~ /a\,c /;      # 「a,c」
```

```
say 'a,c' ~~ /'a,c' /;     # 「a,c」
```

Unverändert: Quantifier

	normal	so wenig wie möglich	gebe nichts auf
0 oder 1	?	??	?+
beliebig viele	*	*?	*+
min. ein	+	+?	++

Generischer Quantifier

```
my $x = 3;
```

```
say 'abcdef' =~ /. ** 1..3 /;
```

```
say 'abcdef' =~ /. ** {1..$x} /;
```

Backslash-Sequenzen

<code>\d</code>	<code>\D</code>	Ziffer
<code>\w</code>	<code>\W</code>	Ziffer oder Buchstabe
<code>\s</code>	<code>\S</code>	Whitespace
<code>\n</code>	<code>\N</code>	Zeilenumbruch
<code>\h</code>	<code>\H</code>	Horizontaler Whitespace
<code>\v</code>	<code>\V</code>	Vertikaler Whitespace

Unicode Properties, <:prop>

L	Letter	Buchstabe
Lu	LetterUppercase	Großbuchstabe
M	Mark	Akzente etc.
N	Number	Ziffer
P	Punctuation	Satzzeichen
S	Symbol	Symbole

https://docs.perl6.org/language/regexes.html#Unicode_properties

Zeichenklassen

```
say '0xDEADBEEF' == /'0x' <[ 0..9 A..F ]> + /;  
say '0xDEADBEEF' == /'0x' <[:digit + [A..F]]> + /;
```

Anker

^	Anfang des Strings
^^	Anfang einer Zeile
\$	Ende des Strings
\$\$	Ende einer Zeile
<<, >>, «, »	Wortgrenze

Alternativen

- | matcht die längste Alternative
- || matcht die erstmögliche Alternative

```
say 'abc' ~~ / ab | .+ / ; # 「abc」  
say 'abc' ~~ / ab || .+ / ; # 「ab」
```

<(und)> schränken Match ein

```
say '0xDEADBEEF' ~~ / '0x' <( <[ 0..9 A..F ]>  
+ /;  
# 「DEADBEEF」
```

Captures und Match

- [] Gruppierung ohne Capture
- () Gruppierung mit Capture
- Nummerierung der Matches fängt bei 0 an
- Gesamter Match in `$/`
- `$0` ist ein Alias für `$/[0]`, `$1` für `$/[1]` etc.
- Matches von Captures wieder Match-Objekte

Match Objects

```
'abcdefg' ~~ /(a)((b)(.))/;  
say $/;
```

```
# 「abcd」  
# 0 => 「a」  
# 1 => 「bcd」  
# 0 => 「b」  
# 1 => 「d」
```

Match Objects

```
say $/.^name;      # Match
say $/.from;       # 0
say $/.to;         # 4
say $/.Str;        # abcd
say $/.list.elems; # 2
say $0;           # 「a」
```

Modifier

- Vor die Regex
- Als Colonpairs / Adverbe
- `s:i/a/b/`
- `s:2nd:i/a/b/`

Regex Adverbs

<code>:i</code>	<code>:ignorecase</code>	Groß- und Kleinschreibung ignorieren
<code>:m</code>	<code>:ignoremark</code>	Akzente etc. ignorieren
<code>:r</code>	<code>:ratchet</code>	Backtracking deaktivieren
<code>:s</code>	<code>:sigspace</code>	Whitespace signifikant

```
rx/ [:i insensitiv] sensitiv /
```

Match Adverbs

<code>:g</code>	<code>:global</code>	Alle matches
<code>:ov</code>	<code>:overlap</code>	Auch überlappend
<code>:ex</code>	<code>:exhaustive</code>	Wirklich alle Matches
<code>:c(5)</code>	<code>:continue(5)</code>	Suche ab Pos. 5 (Default \$/ .to)
<code>:p(5)</code>	<code>:pos(5)</code>	Wie :c, aber verankert
<code>:x(5)</code>		Mindestens 5 Matches
<code>:th, :st, :nd, :rd</code>		Offset

: sigspace

- Ersetzt Whitespace in der Regex durch `<.ws>`
Aufrufe
- `token ws { <!ww> \s* }`
- In Grammatiken überschreibbar
- `rule x { .. }` entspricht `regex x`
`:sigspace :ratchet { .. }`

Übung: Regexes

- Schreibe ein kleines Programm, das Einheiten (cm in m und umgekehrt) konvertiert:
`./autoconvert "12cm in m"`
- Bonus-Punkte für Support von nicht-Integern
`("12.5cm in m")`

Ini-Datei Parsen

```
[heading]  
key=value  
key2=more value
```

```
[perl6]  
style=mixed  
feel=awesome
```

Ini-Datei Parsen

[heading]
key=value
key2=more value

[perl6]
style=mixed
feel=awesome

' [' .+? '] ' \n+

Ini-Datei Parsen

[heading]
key=value
key2=more value

'[' .+? ']' \n+

[perl6]
style=mixed
feel=awesome

(\w+) \s*
'=' \s* (\N+)
\n+

Der Parser nimmt Form an

```
my regex header {  
    '[' ( .+? ) ']' \n+  
}  
my token pair {  
    (\w+) \s* '=' \s* (\N+) \n+  
}  
  
say "[perl6]\n" ~~ /<&header>/;  
say "feel=awesome\n" ~~ /<&pair>/;
```

Der Parser nimmt Form an

```
my regex header {  
    '[' ( .+? ) ']' \n+  
}  
my token pair {  
    (\w+) \s* '=' \s* (\N+) \n+  
}
```

Regex ohne
Backtracking

```
say "[perl6]\n" ~~ /<&header>/;  
say "feel=awesome\n" ~~ /<&pair>/;
```

Zusammenfügen

```
my token ini { <&section>* }
```

```
my token section {  
  <&header>  
  <&pair>*  
}
```

```
say slurp('foo.ini') ~~ /<ini>/;
```


Grammatiken

- Bisher: Code Reuse wie Subroutinen
- Wäre es nicht toll, wenn man Klassen oder Rollen dafür nehmen könnte?
- Klasse mit Regexes drin == Grammatik
- Implizit Parent-Klasse Grammar

Grammatik

```
grammar Ini {  
    token TOP { <section>* }  
    token section { <header> <pair>* }  
    regex header {  
        '[' ( .+? ) ']' \n+  
    }  
    token pair {  
        (\w+) \s* '=' \s* (\N+) \n+  
    }  
}  
  
say Ini.parsefile("foo.ini");
```

Werte Extrahieren

- Das Match-Objekt hat alle interessanten Werte
- Durch das Objekt gehen ist eher anstrengend, viel kann schief gehen
- Alternative: Action Objects
- Werden aufgerufen, wenn eine Regex erfolgreich war
- können `&make` aufrufen, um `$/ .made` zu setzen

Action Objects I

```
class IniSection {  
  has $.heading;  
  has @.pairs;  
}
```

```
class IniAction {  
  method pair($/) {  
    make ~$0 => ~$1;  
  }  
  method header($/) {  
    make ~$0  
  }  
}
```

Action Objects II

```
method section($/) {  
    make IniSection.new(  
        heading => $<header>.made,  
        pairs   => $<pair>.map: { .made },  
    )  
}  
method TOP($/) {  
    my %sections;  
    for $<section>.list -> $s {  
        %sections{$s.made.heading} = $s.made;  
    }  
    make %sections;  
}  
}
```

Action Objects III

```
my $parsed = Ini.parsefile("foo.ini",  
    :actions(IniAction.new));  
say $parsed.made.perl;
```

```
{:heading(IniSection.new(heading =>  
"heading", pairs => [:key("value"),  
:key2("more value")])),  
:perl6(IniSection.new(heading => "perl6",  
pairs => [:style("mixed"),  
:feel("awesome")]))}
```

JSON-Grammatik I

```
use v6;  
unit grammar JSON::Tiny::Grammar;  
  
token TOP { \s* <value> \s* }  
rule object { '{' ~ '}' <pairlist> }  
rule pairlist { <pair> * % \, }  
rule pair { <string> ':' <value> }  
rule array { '[' ~ ']' <arraylist> }  
rule arraylist { <value> * % [ \, ] }
```

JSON-Grammatik II

```
proto token value {*};
token value:sym<number> {
    ' - '?
    [ 0 | <[1..9]> <[0..9]>* ]
    [ \. <[0..9]>+ ]?
    [ <[eE]> [\+|\-]? <[0..9]>+ ]?
}
token value:sym<true>      { <sym>      };
token value:sym<false>     { <sym>      };
token value:sym<null>      { <sym>      };
token value:sym<object>    { <object>   };
token value:sym<array>     { <array>    };
```


JSON-Grammatik III

```
token value:sym<string> { <string> }
```

```
token string {  
    \" ~ \" [ <str> | \\ <str=.str_escape> ]*  
}
```

```
token str { <-[\"\\\\t\\n]>+ }
```

```
token str_escape {  
    <[\"\\\\/bfnrt]> | 'u' <utf16_codepoint>+ % '\\u'  
}
```

```
token utf16_codepoint { <.xdigit>**4 }
```

JSON-Grammatik IV

```
class X::JSON::Tiny::Invalid is Exception {  
    has $.source;  
    method message { "Input ($.source.chars()  
characters) is not a valid JSON string" }  
}
```

```
sub from-json($text) is export {  
    my $a = JSON::Tiny::Actions.new();  
    my $o = JSON::Tiny::Grammar.parse($text,  
:actions($a));  
    unless $o {  
        X::JSON::Tiny::Invalid.new(source =>  
$text).throw;  
    }  
    return $o.made;  
}
```

Grammatik: Vererbung

```
use JSON::Tiny::Grammar;  
use JSON::Tiny::Actions;  
  
grammar JSON::Inf::Grammar  
    is JSON::Tiny::Grammar {  
        token value:sym<Inf> { 'Inf' }  
    }  
  
say JSON::Inf::Grammar.parse(  
    '{ "a": Inf }',  
);
```

Grammatik-Vererbung: Output

```
「 { "a": Inf} 」  
value => 「 { "a": Inf} 」  
object => 「 { "a": Inf} 」  
pairlist => 「 "a": Inf 」  
pair => 「 "a": Inf 」  
string => 「 "a" 」  
str => 「 a 」  
value => 「 Inf 」
```

Übung Grammatiken: CSV-Parser

- Schreibe einen CSV-Parser
- Separator ist , Quoting mit ", kein Escaping
- Erste Zeile Input ist die Kopfzeile
- Ausgabe: Array of Hashes

Input/Output

open()

```
my $fh = open(IO::Path() $path, :$r, :$w,  
              :$a, :$rw, :$bin, :$enc,  
              :$nl-in, :$nl-out, :$chomp=True)
```

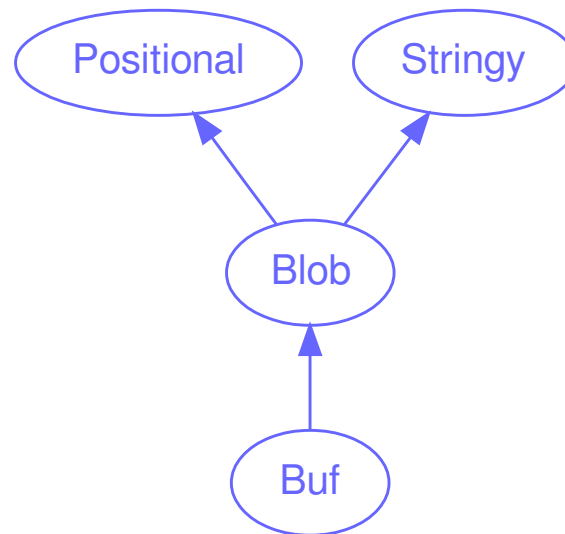
```
my $fh = open($?FILE);  
for $fh.lines.kv -> $k, $v {  
    say "$k: $v";  
}
```

\$filename.IO

- Liefert ein `IO::Path`-Objekt
- Damit kann man File-Tests als Methoden aufrufen (`.e`, `.r`, `.w`, `.d`, ...)
- Pfad-Manipulationen wie `basename`, `dirname`, `parts`, `watch`, `is-absolute`
- Daten auslesen mit `.get`, `.lines`, `.slurp-rest`

<https://docs.perl6.org/type/IO::Path>

Binärdaten



- `$fh.read($count), $fh.write($Blob)`
- `Str.encode($encoding),
Blob.decode($encoding)`

Externe Programme ausführen

```
my $proc = run 'echo', 'Hallo world', :out;  
my $captured-output = $proc.out.slurp-rest;  
say "Output was $captured-output.perl()";  
    # Output was "Hallo world\n"
```

```
my $p1 = run 'echo', 'Hello, world', :out;  
my $p2 = run 'cat', '-n', :in($p1.out), :out;  
say $p2.out.get;
```

<https://docs.perl6.org/type/Proc>

Parallele Ausführung

Parallele Ausführung

- Primitive: Promises, Warteschlangen, Streams
- Threads, Threadpools, Locks eher im Hintergrund
- Zugriff auf "normale" Datenstrukturen i.A. nicht thread save, muss man synchronisieren

Parallele Ausführung: Promise

```
sub number-of-primes(Int $upto) {  
    (1..$upto).grep(&is-prime).elems  
}  
  
my $promise = start number-of-primes(4000);  
say $promise.status;           # Planned  
  
await $promise;  
say $promise.status;           # Kept  
say $promise.result;           # 550
```

Warteschlangen

```
my $source = Channel.new;
my @worker = (1..4).map: -> $i {
  start loop {
    my $v = $source.receive;
    say $v x 5; # Arbeit hier machen
    CATCH {
      when X::Channel::ReceiveOnClosed {
        say "Shutting down worker $i";
        last;
      }
    }
  }
}
$source.send(('a' .. 'z').pick)
for 1..100;
```

Parallele Datenströme

```
sub MAIN(Str $path = '.') {  
  react {  
    whenever IO::Notification  
      \ .watch-path($path)  
      \ .stable(0.2) -> $event {  
      my $path = $event.path.IO;  
      if $path.s {  
        say "$path: ", $path.s;  
      }  
    }  
  }  
  whenever signal(SIGINT) {  
    say "Shutting down";  
    done;  
  }  
}  
}
```

Junctions

Junctions

```
my @input = 1, 4, 3;

if any(@input) == 3 {
    say "Da ist eine drei drin";
}

if 0 <= all(@input) < 10 {
    say 'Alles gut.';
}

if sqrt(none(@input)) == 2 {
    say "Keine 4 dabei";
}
```

Junction-Typen

	any	Mindestens ein True
&	all	Kein False
^	one	Genau ein True
	none	Kein True

Junction Autothreading

- `2 + all(2, 3)` wird `all(2 + 2, 2 + 3)`
aka `all(4, 5)`
- `4 == all(4, 5)` wird `all(4 == 4, 4 == 5)` wird `all(True, False)`
- Im Boolean Kontext wird ein einzelner
True/False-Wert daraus:
`?all(True, False) == False`

Wie Funktioniert's?

- Sub- und Operator-Parameter sind Typ Any
- Junction !~~ Any
- Als Fallback findet Autothreading statt
- Mit explizitem Typ Junction oder Mu unterbindbar

Übung IO, Junctions

- Schreibe ein Programm, das Zahlen von STDIN liest, und überprüfe, ob alle einem vorgegebenem Minimum und Maximum entsprechen
- `echo "1\n2.5\n3" | ./numbercheck --min=1 --max=2`

Phaser

- Code-Blöcke, die zu bestimmten Zeiten automatisch aufgerufen werden
- Aus Perl 5 bekannt: BEGIN, END

Phasers I

BEGIN	Anfang Compile-Zeit
CHECK	Ende Compile-Zeit
INIT	Anfang Laufzeit
END	Ende Laufzeit

Beachte: Bei Modulen können Compile- und Laufzeit getrennt sein.

Phasers II

ENTER	Bei jedem Eintritt in den Block
LEAVE	Bei jedem Verlassen des Blocks
KEEP	Bei erfolgreichem Verlassen des Blocks
UNDO	Bei fehlerhaftem Verlassen des Blocks
CATCH	Bei Exceptions
CONTROL	Bei Control-Exceptions

Phasers III

FIRST	Vor erstem Durchlauf einer Schleife
NEXT	Vor Fortführen einer Schleife
LAST	Nach der letzten Iteration einer Schleife
PRE	Precondition
POST	Postcondition

Phaser: Beispiele

```
say elems (1..20).grep({ (2 ** $_ - 1).is-  
prime});  
say now - INIT now;
```

```
my $fh = open 'output', :w;  
LEAVE $fh.close if $fh;  
$fh.say: 'initial data';  
die "OH NOEZ";
```

Whatever-Priming

- Erzeugt aus einem Ausdruck ein Code-Objekt
- Ersetze ein Objekt, auf dem man eine Methode aufruft, durch einen *
- Oder ersetze einen Operand eines Operators durch einen *
- Bestimmte Operatoren ausgenommen: . . , . . . , ~ ~

Whatever-Priming

```
say grep { $_.is-prime }, 1..10;  
say grep *.is-prime, 1..10;
```

```
say map { $_ + 5 }, 1..3;  
say map * + 5, 1..3;
```

Übung Whatever-Priming

- Gehe durch vorherige Beispiele, und probiere, Whatever-Priming einzusetzen. Teste, ob der Code immer noch funktioniert :-)

Das war's

Feedback?

Vielen Dank!