Dennis Felsing

2014-06-21

Hello World

```
echo "Hello World"
```

Hello World

```
echo "Hello World"
```

- Willkommen zu einem Vortrag über Shell-Scripting!

Hello World

```
echo "Hello World"
```

- Willkommen zu einem Vortrag über ~~Shell-Scripting~~ Nimrod!

Hello World

```
echo "Hello World"
```

- Willkommen zu einem Vortrag über ~~Shell-Scripting~~ Nimrod!
- Eine neue System-Programmiersprache
  (neben D, Go und Rust)

Hello Nimrod

- statisch typisiert

```
var name: string = "World"
```

- mit Typinferenz

```
var str = "Hello " & name
```

- und klarer Syntax

```
for i in 1..10:
  echo str
```

- Kompiliert zu C als Zwischensprache:

```
proc printf(formatstr: cstring)
  {.header: "<stdio.h>", varargs.}
printf("%s %d\n", str, 5)
```

Nimrod?

- Seit 2008 entwickelt, aktuell Version 0.9.4
- Kleines Entwicklerteam rund um Andreas Rumpf
- Version 1.0 Ende 2014 erwartet
- Kein Unternehmen dahinter
- Compiler größtes Open-Source-Projekt in Nimrod

Ziele

Nimrod combines Lisp's power with Python's readability and C's performance.

- Die drei Es: expressive, efficient, elegant

Expressive:	Keine unnötige Syntax
	Standard-Library in Nimrod
	AST zur Compiletime bearbeiten
Efficient:	Kompiliert zu C mit C-artiger Performance
	Kleine Binaries ohne Dependencies
	Performanter Garbage Collector
Elegant:	Metaprogrammierung zur Compiletime
	Genauso zu schreiben wie Runtime-Code

- Freiheit statt Einschränkungen

Variablen und Konstanten

```
var x = 20
x = 19
x = "foo" # error: type mismatch

let y = 20 # immutable variable
y = 19 # error: cannot assign

const z = 1 + 3

var f1,f2: float = 1e40

var
  a = 1000000
  b: int16 = 0b0111110001000000
  c = 0xFFF8AB12FC000001'u64
```

Variablen und Konstanten

```
var x = 20
x = 19
x = "foo" # error: type mismatch

let y = 20 # immutable variable
y = 19 # error: cannot assign

const z = 1 + 3

var f1,f2: float = 1e40

var
  a = 1_000_000
  b: int16 = 0b0111_1100_0100_0000
  c = 0xFF_F8_AB_12_FC_00_00_01'u64
```

Kontrollstrukturen

```
while true:
  if 2 > 3:
    echo "hi"
  elif 1: # type mismatch: expected bool
    echo "then"
  else:
    echo "bye"
    break

  block loops:
    while true:
      while "foo" != "bar":
        break loops
```

Case-Statements

```
import strutils

# error: not all cases covered
case stdin.readLine.parseInt()
of 1:
  echo "go language"
of 2, 6:
  echo "rust language"
of 3..5, 7..9:
  echo "nimrod language"
```

Case-Statements

```
import strutils


case stdin.readLine.parseInt()
of 1:
  echo "go language"
of 2, 6:
  echo "rust language"
of 3..5, 7..9:
  echo "nimrod language"
else:
  discard
```

Typen

```
type dice = range[0..5]
var d: dice = 4
d = 6 # error: out of range

var ds: array[100, dice]
ds[50] = 5
ds[60..69] = ds[50..59]
for d in ds:
  echo d
ds[105] = 0 # error: index out of bounds

var newDs: seq[dice] = @[]
for i,d in ds:
  if i > 10 and d == 5:
    newDs.add(d)
echo newDs
```

Typen

```
type dice = range[0..5]
var d: dice = 4
d = 6 # error: out of range

var ds: array['a'..'z', dice]
ds['f'] = 5

for d in ds:
  echo d
ds['D'] = 0 # error: index out of bounds

var newDs = newSeq[dice]()
for i,d in ds:
  if i > 'g' and d == 5:
    newDs.add(d)
echo newDs
```

Prozeduren

```
proc dimensions(): tuple[w, h: int] = (15,20)

echo(dimensions().w)

proc sum*(x, y: int): int = # * means exported
  x + y

proc sum(xs: seq[int]): int =
  for x in xs:
    result += x

dimensions() # error: value has to be discarded
```

Prozeduren

```
proc dimensions(): tuple[w, h: int] = (15,20)

echo(dimensions().w)

proc sum*(x, y: int): int = # * means exported
  x + y

proc sum(xs: seq[int]): int =
  for x in xs:
    result += x

discard dimensions()
var (x,y) = dimensions()

echo sum(x,y)
echo x.sum(y)
echo x.sum y
```

Micro-Benchmark

Python

```python
def eratosthenes(n):
  sieve = [1] * 2 + [0] * (n - 1)
  for i in range(int(n**0.5)):
    if not sieve[i]:
      for j in range(i*i, n+1, i):
        sieve[j] = 1
  return sieve

eratosthenes(100000000)
```

Nimrod

```nimrod
import math
proc eratosthenes(n): auto =
  result = newSeq[int8](n+1)
  result[0] = 1; result[1] = 1

  for i in 0 .. int sqrt(float n):
    if result[i] == 0:
      for j in countup(i*i, n, i):
        result[j] = 1

discard eratosthenes(100_000_000)
```

C

```c
#include <stdlib.h>
#include <math.h>

char* eratosthenes(int n)
{
  char* sieve = calloc(n+1,sizeof(char));
  sieve[0] = 1; sieve[1] = 1;
  int m = (int) sqrt((double) n);

  for(int i = 0; i <= m; i++) {
    if(!sieve[i]) {
      for (int j = i*i; j <= n; j += i)
        sieve[j] = 1;
    }
  }
  return sieve;
}

int main() {
  eratosthenes(100000000);
}
```

Micro-Benchmark

Python - 35.1s

```python
def eratosthenes(n):
  sieve = [1] * 2 + [0] * (n - 1)
  for i in range(int(n**0.5)):
    if not sieve[i]:
      for j in range(i*i, n+1, i):
        sieve[j] = 1
  return sieve

eratosthenes(100000000)
```

Nimrod

```nimrod
import math
proc eratosthenes(n): auto =
  result = newSeq[int8](n+1)
  result[0] = 1; result[1] = 1

  for i in 0 .. int sqrt(float n):
    if result[i] == 0:
      for j in countup(i*i, n, i):
        result[j] = 1

discard eratosthenes(100_000_000)
```

C

```c
#include <stdlib.h>
#include <math.h>

char* eratosthenes(int n)
{
  char* sieve = calloc(n+1,sizeof(char));
  sieve[0] = 1; sieve[1] = 1;
  int m = (int) sqrt((double) n);

  for(int i = 0; i <= m; i++) {
    if(!sieve[i]) {
      for (int j = i*i; j <= n; j += i)
        sieve[j] = 1;
    }
  }
  return sieve;
}

int main() {
  eratosthenes(100000000);
}
```

Micro-Benchmark

Python - 35.1s

```python
def eratosthenes(n):
  sieve = [1] * 2 + [0] * (n - 1)
  for i in range(int(n**0.5)):
    if not sieve[i]:
      for j in range(i*i, n+1, i):
        sieve[j] = 1
  return sieve

eratosthenes(100000000)
```

Nimrod

```nimrod
import math
proc eratosthenes(n): auto =
  result = newSeq[int8](n+1)
  result[0] = 1; result[1] = 1

  for i in 0 .. int sqrt(float n):
    if result[i] == 0:
      for j in countup(i*i, n, i):
        result[j] = 1

discard eratosthenes(100_000_000)
```

C - 2.6s

```c
#include <stdlib.h>
#include <math.h>

char* eratosthenes(int n)
{
  char* sieve = calloc(n+1,sizeof(char));
  sieve[0] = 1; sieve[1] = 1;
  int m = (int) sqrt((double) n);

  for(int i = 0; i <= m; i++) {
    if(!sieve[i]) {
      for (int j = i*i; j <= n; j += i)
        sieve[j] = 1;
    }
  }
  return sieve;
}

int main() {
  eratosthenes(100000000);
}
```

Micro-Benchmark

Python - 35.1s

```python
def eratosthenes(n):
  sieve = [1] * 2 + [0] * (n - 1)
  for i in range(int(n**0.5)):
    if not sieve[i]:
      for j in range(i*i, n+1, i):
        sieve[j] = 1
  return sieve

eratosthenes(100000000)
```

Nimrod - 2.6s

```nim
import math
proc eratosthenes(n): auto =
  result = newSeq[int8](n+1)
  result[0] = 1; result[1] = 1

  for i in 0 .. int sqrt(float n):
    if result[i] == 0:
      for j in countup(i*i, n, i):
        result[j] = 1

discard eratosthenes(100_000_000)
```

C - 2.6s

```c
#include <stdlib.h>
#include <math.h>

char* eratosthenes(int n)
{
  char* sieve = calloc(n+1,sizeof(char));
  sieve[0] = 1; sieve[1] = 1;
  int m = (int) sqrt((double) n);

  for(int i = 0; i <= m; i++) {
    if(!sieve[i]) {
      for (int j = i*i; j <= n; j += i)
        sieve[j] = 1;
    }
  }
  return sieve;
}

int main() {
  eratosthenes(100000000);
}
```

Funktionen?

- Funktion = Prozedur ohne Seiteneffekte
- Compiler muss verifizieren können

```
proc sum(x, y: int): int {.noSideEffect.} =
  x + y

proc minus(x, y: int): int {.noSideEffect.} =
  echo x # error: 'minus' can have side effects
  x - y
```

Mehr Compileranalyse

- Effects geben Info was Prozedur machen kann
- Nur bestimmte Seiteneffekte erlauben:

```
proc readLine(): string {.tags: [FReadIO].}
```

- Analog auch für Exceptions:

```
proc doRaise() {.raises: [EIO, EOverflow].}
```

- Automatisch inferierte Effects ausgeben:

```
proc lastChar(): char =
  let line = stdin.readLine()
  # Hint: FReadIO [User]
  if line.len == 0:
    raise newException(EIO, "IO")
    # Hint: ref EIO [User]
  return line[line.high]
  {.effects.}
```

Prozeduren erster Klasse

- Prozeduren als Rückgabewerte, Parameter und Variablen

```
proc printer(x: int): proc =
  proc y() =
    echo "hello " & $x
  return y

proc callMe(p) =
  for i in 1..10:
    p()

var printer10 = printer(10)
printer10()
callMe(printer10)
```

Generics

- Allgemeine Generics:

```
proc sum[T](x, y: T): T =
  x + y

echo sum(12.5, 13.5)
echo sum("foo", "bar") # error: type mismatch for ‘+‘
```

Generics

- Allgemeine Generics:

```
proc sum[T: int|int64|float](x, y: T): T =
  x + y

echo sum(12.5, 13.5)
echo sum("foo", "bar") # error: type mismatch for sum
```

Generics

- Allgemeine Generics:

```
proc sum[T: int|int64|float](x, y: T): T =
  x + y

echo sum(12.5, 13.5)
echo sum("foo", "bar") # error: type mismatch for sum
```

- Type Constraints für kompakteren Code:

```
import strutils

proc sum(xs: seq[int|string]): int =
  for x in xs:
    let y =
      when x is int: x
      else: parseInt(x)
    result += y

echo sum(@["12", "14", "9"])
```

Parameter

- Parameter sind standardmäßig unveränderlich (wie `let`)
- Können aber als `var` deklariert werden:

```
proc `++`(x: var int; y = 1; z = 0) =
  x = x + y + z

var a = 10

++a
a ++ 2
a.`++`(3, 4)
```

Eigenheiten von Nimrod

- Identifier sind case insensitive
  $\Rightarrow$ Man kann Lieblingskonvention verwenden:

```
proc SetPosition*(window: PWindow; x, y: cint) # SDL2
proc set_default_dpi*(dpi: cdouble) # libRSVG
window.setPosition(640, 480)
setDefaultDPI(90.0)
```

Eigenheiten von Nimrod

- Identifier sind case insensitive
  $\Rightarrow$ Man kann Lieblingskonvention verwenden:

```
proc SetPosition*(window: PWindow; x, y: cint) # SDL2
proc set_default_dpi*(dpi: cdouble) # libRSVG
window.setPosition(640, 480)
setDefaultDPI(90.0)
```

- Nur Leerzeichen zum einrücken, Tabs sind verboten
  $\Rightarrow$ Keine Missverständnisse bei Einrückungen

Eigenheiten von Nimrod

- Identifier sind case insensitive
  $\Rightarrow$ Man kann Lieblingskonvention verwenden:

```
proc SetPosition*(window: PWindow; x, y: cint) # SDL2
proc set_default_dpi*(dpi: cdouble) # libRSVG
window.setPosition(640, 480)
setDefaultDPI(90.0)
```

- Nur Leerzeichen zum einrücken, Tabs sind verboten
  $\Rightarrow$ Keine Missverständnisse bei Einrückungen
- Mehrere Zeilen auskommentieren, wird nicht geparst:

```
when false:
  this code is broken()
```

Eigenheiten von Nimrod

- Identifier sind case insensitive
  ⇒ Man kann Lieblingskonvention verwenden:

```
proc SetPosition*(window: PWindow; x, y: cint) # SDL2
proc set_default_dpi*(dpi: cdouble) # libRSVG
window.setPosition(640, 480)
setDefaultDPI(90.0)
```

- Nur Leerzeichen zum einrücken, Tabs sind verboten
  ⇒ Keine Missverständnisse bei Einrückungen
- Mehrere Zeilen auskommentieren, wird nicht geparst:

```
when false:
  this code is broken()
```

- Identifier müssen vor Benutzung deklariert sein
  ⇒ Effizientere Kompilierung

Metaprogrammierung: Templates (1)

- Templates fügen Code zur Compiletime ein
- Werden genau wie normale Prozeduren aufgerufen

```
import os
const debug = false

proc log*(msg: string) =
  if debug:
    echo msg

proc expensive(): string =
  sleep(milsecs = 1000)
  result = "That was difficult"

10.times:
  log(expensive())
```

Metaprogrammierung: Templates (1)

- Templates fügen Code zur Compiletime ein
- Werden genau wie normale Prozeduren aufgerufen

```
import os
const debug = false

template log*(msg: string) =
  if debug:
    echo msg

proc expensive(): string =
  sleep(milsecs = 1000)
  result = "That was difficult"

10.times:
  log(expensive())
```

Metaprogrammierung: Templates (2)

- Beispiel vom Anfang:

```
for i in 1..10:
  echo str
```

- Templates können mit allgemeinen Typen arbeiten:
  `expr`, `stmt`, `typedesc`
- Spezielle Syntax für Templates mit Statement-Parametern:

```
template times(x: expr, y: stmt): stmt =
  for i in 1..x:
    y

10.times:
  echo "hi"
```

Metaprogrammierung: Term Rewriting (1)

```
var x: int
for i in 1..1_000_000_000:
  x += 2 * i
echo x
```

Metaprogrammierung: Term Rewriting (1)

- Schreiben wir unsere eigenen Compiler-Optimierungen

```
template optMul{`*`(a,2)}(a: int): int =
  let x = a
  x + x

template canonMul{`*`(a,b)}(a: int{lit}, b: int): int =
  b * a

var x: int
for i in 1..1_000_000_000:
  x += 2 * i
echo x
```

Metaprogrammierung: Term Rewriting (2)

- Auch komplexere Patterns

```
template optLog1{a and a}(a): auto = a
template optLog2{a and (b or (not b))}(a,b): auto = a
template optLog3{a and not a}(a: int): auto = 0

var
  x = 12
  s = x and x
  # Hint: optLog1(x) --> 'x' [Pattern]
```

Metaprogrammierung: Term Rewriting (2)

- Auch komplexere Patterns

```
template optLog1{a and a}(a): auto = a
template optLog2{a and (b or (not b))}(a,b): auto = a
template optLog3{a and not a}(a: int): auto = 0

var
  x = 12
  s = x and x
  # Hint: optLog1(x) --> 'x' [Pattern]

  r = (x and x) and ((s or s) or (not (s or s)))
  # Hint: optLog2(x and x, s or s) --> 'x and x' [Pattern]
  # Hint: optLog1(x) --> 'x' [Pattern]
```

Metaprogrammierung: Term Rewriting (2)

- Auch komplexere Patterns

```
template optLog1{a and a}(a): auto = a
template optLog2{a and (b or (not b))}(a,b): auto = a
template optLog3{a and not a}(a: int): auto = 0

var
  x = 12
  s = x and x
  # Hint: optLog1(x) --> 'x' [Pattern]

  r = (x and x) and ((s or s) or (not (s or s)))
  # Hint: optLog2(x and x, s or s) --> 'x and x' [Pattern]
  # Hint: optLog1(x) --> 'x' [Pattern]

  q = (s and not x) and not (s and not x)
  # Hint: optLog3(s and not x) --> '0' [Pattern]
```

Metaprogrammierung: Macros

```
proc `^`*(base, exp: int): int =
  var (base, exp) = (base, exp)
  result = 1

  while exp != 0:
    if (exp and 1) != 0:
      result *= base
    exp = exp shr 1
    base *= base
```

Metaprogrammierung: Macros

- Macros werden zur Compiletime ausgeführt
- Können AST von Parametern lesen, geben AST zurück
- Lassen sich aber ganz normal schreiben
- Können normale Prozeduren aufrufen

```
proc `^`*(base, exp: int): int

import macros

macro potSum(n: int): expr =
  var sum = 0
  for i in 1..int(n.intVal):
    sum += 2^i
  result = parseExpr($sum) # parse string to AST

echo potSum(10)
```

Metaprogrammierung: Macros

- Macros werden zur Compiletime ausgeführt
- Können AST von Parametern lesen, geben AST zurück
- Lassen sich aber ganz normal schreiben
- Können normale Prozeduren aufrufen

```
proc `^`*(base, exp: int): int

import macros

macro potSum(n: int): expr =
  var sum = 0
  for i in 1..int(n.intVal):
    sum += 2^i
  result = newIntLitNode(sum) # directly build AST

echo potSum(10)
```

Immer mehr Metaprogrammierung

- `units`: Kleine Library für physikalische Einheiten
- Nur physikalisch sinnvolle Berechnungen möglich
- Kein Runtime-Overhead

```
quantity(Time, second, "s")
quantity(Velocity, meterPerSecond, "m/s")
quantity(Acceleration, meterPerSecondSquared, "m/s²")

Velocity := Length / Time
Acceleration := Velocity / Time

var t: Time = 4.seconds
var v = 2.meters / t
var a: Acceleration = v / millisecond
ac *= 3.0
echo ac # 1500 m/s²
```

Macros: AST von Parametern

```
macro `:=`(assign, data): stmt =
  assert assign.kind == nnkIdent
  let to = $assign.ident

  assert data.len == 3
  assert data.kind == nnkInfix

  let m1 = data[0]
  assert m1.kind == nnkIdent
  assert $m1.ident == "/"

  let from1 = $data[1].ident
  let from2 = $data[2].ident

  result = parseStmt("isMult(" &
    from1 & ", " & to & ", " &
    from2 & ")")
```

Beispiel-AST

```
Velocity := Length / Time
echo treeRepr(assign)
echo treeRepr(data)
```

```
assign =
  Ident !"Velocity"
data =
  Infix
    Ident !"/"
    Ident !"Length"
    Ident !"Time"
```

Garbage Collector

- Vieles auf dem Stack, ohne Garbage Collector
- Nur bei Memory Allocation auf Heap kann GC laufen
- Soft realtime möglich, indem man den GC selbst steuert:

```
gcDisable()
while true:
  gameLogic()
  renderFrame()
  gcStep(us = leftTime)
  sleep(restTime)
```

Oh my C

- Code in C ...

```
void chi(char* name) {
  printf("awesome %s\n", name);
}
```

- ... lässt sich direkt in Nimrod integrieren:

```
{.compile: "hi.c".}
proc hi*(name: cstring) {.importc: "chi".}

hi("GPN14")
```

- Automatisch generierte Definitionen für Libraries mit `c2nim`:

```
proc set_default_dpi*(dpi: cdouble) {.cdecl,
  importc: "rsvg_set_default_dpi",
  dynlib: "librsvg-2.so".}
```

Objektorientiert

```
type
  Position = tuple[x, y: float]

  Graphic = object of TObject
    pos: Position

  Circle = object of Graphic
    radius: float

  Rectangle = object of Graphic
    size: tuple[w, h: float]

var c = Circle(pos: (20.5,30.1), radius: 10.9)
```

- `method` statt `proc`: Dynamic dispatch

Object Variants

```
type
  Position = tuple[x, y: float]

  GraphicKind = enum Circle, Rectangle

  Graphic = object
    pos: Position
    case kind: GraphicKind
    of Circle:
      radius: float
    of Rectangle:
      size: tuple[w, h: float]

var c = Graphic(kind: Circle, pos: (20.5,30.1),
                radius: 10.9)
```

Pointer

```
type
  List*[T] = ref TList[T] # garbage collected pointer
  TList[T] = object
    data: T
    next: List[T]

proc newList*[T](data: T): List[T] =
  new(result)
  result.data = data
# procs don't belong to object
proc insert*[T](x: var List[T], y: List[T]) =
  let tmp = x.next
  x.next = y
  y.next = tmp

var ls = newList("foo")
ls.insert(newList("bar"))
```

Funktional

```
import sequtils

let
  a = @[1, 2, 3, 4]
  b = a.concat(@[5, 6, 7]).map(proc(x: int): string =
    "number " & $x)

let
  colors = @["red", "yellow", "black"]
  f1 = filter(colors, proc(x): bool = x.len < 6)
  f2 = colors.filter do (x) -> bool : x.len > 5
```

Tools

- Dokumentation: `nimrod doc`
- Formatter / Pretty Printer: `nimrod pretty`
- Search and Replace: `nimgrep` unterstützt case insensitivity
- Packetmanager: `babel`

Debugger

- Debug-Builds liefern Stacktraces
- `gdb` nutzbar
- Embedded Nimrod Debugger

Profiler

- `valgrind` und ähnliche nutzbar
- Embedded Stack Trace Profiler

Nimrod

- Junge Programmiersprache mit tollen Features
- Viele Libraries und Bindings:

Collections: hash tables, bit sets, intsets, critbits, ...
Strings: substrings, parseutils, encodings, matchers, ...
Spiele: SDL2, OpenGL, SFML, Allegro, Chipmunk, ...
Internet: asyncnet, http, irc, ftp, smtp, jester, ...
...

- Was mir fehlt: Math: bignums, vectors, matrices
- Kleine Community mit viel Potential für neue Leute

http://nimrod-lang.org