



PROGRAMMIERSPRACHE NIM

AGENDA

- I. EINFÜHRUNG
- II. VARIABLENDEKLARATION
- III. BEDINGUNGEN
- IV. SCHLEIFEN
- V. DATENTYPEN
- VI. PROZEDUREN
- VII. OOP
- VIII. EXCEPTIONS
- IX. GENERICS
- X. TEMPLATES
- XI. MACRO

GESCHICHTE ZU NIM

Entwicklung: Begonnen 2005 von **Andreas Rumpf**, ursprünglich **Nimrod** genannt, veröffentlicht 2008.

Erster Compiler: Zunächst in Pascal (Free Pascal Compiler) geschrieben, ab 2008 in Nim selbst implementiert.

Umbenennung: Von Nimrod zu **Nim** mit Version 0.10.2 (Dezember 2014).

Version 1.0: Veröffentlicht am **23. September 2019**, markiert die Stabilität der Sprache und ihrer Entwicklungswerkzeuge (Compiler, Paketmanager).

Version 2.0: Veröffentlicht am **1. August 2023**, mit Fokus auf das ARC/ORC-Speichermodell für effizientes und vorhersehbares Speichermanagement.





ALLGEMEINES

Einflüsse

Nim wurde von vielen Programmiersprachen inspiriert, darunter **Python**, **Ada**, **Lisp** und **C++**

Ziel: Effizient, ausdrucksstark und elegant.

Plattform

Nim kann nach **C**, **C++**, **JavaScript**, **Objective-C** und **LLVM** kompilieren und funktioniert auf verschiedenen Systemen.

Über Nim

Statisch typisiert: Nim prüft den Datentyp von Variablen schon beim Kompilieren. Es unterstützt auch Makros, um den Code zur Kompilierzeit anzupassen.

Unterstützte Paradigmen: Metaprogrammierung, funktionale Programmierung, prozedurale und objektorientierte Programmierung.

INSTALLATIONSMÖGLICHKEITEN

Homebrew (macOS, Linux, WSL)

1 **brew** install nim

Alternativ: [Download – Nim Programming language](#)

Online-Compiler: [Nim Online Editor](#)

VARIABLENDEKLARATION

Nim unterstützt sowohl veränderbare als auch unveränderliche Variablen

Syntax:

`var <name>: <type>`

`var <name>: <type> = <value>`

- Deklaration mit ``var``:

`var a: int`

`var b = 7 #Typ automatisch als int erkannt`

`var b = 9`

`var b = "HELLO" #error`

``var` Block:`

`var`

`c = -11`

`d = "Hello"`

Unveränderliche Zuweisungen:

- ``const``: Wert muss zur Compile- Zeit bekannt sein

`const g = 9`

`const g = -27 #error`

- ``let``: Wert kann zur Laufzeit gesetzt werden, bleibt aber unveränderlich

`let j = 25`

`let j = -44 #error`

BEDINGUNGEN

Vergleichsoperatoren:

Wertvergleich:

`<=, >=, <, >, ==, !=`

Objekt Referenz/Type:

is, is not

```
var p1: Person      var p2: Person
```

```
p1 is Person == true
```

```
p1 is p2 == false
```

Element in Sequenz:

in, not in

```
2 in [1,2,3] == true
```

```
"ell" in "Hello" == true
```

Logische Operatoren:

and, or, ()

and ist höhergestellt als or

Nim ist strongly-typed, hat also keine impliziten true, false Werte, wie z.B. in JS und Python. Elemente müssen mit `bool()` konvertiert werden

Zahlen: `bool(0) = false` , `bool(-1) = true` , `bool(1) = true`

String, Sequences, Arrays, Dictionaries: wenn leer `== false`

References, Objects: bei `= nil` zu false ausgewertet

BEDINGUNGEN

If:

```
if <Bedingung>:  
    <Block>
```

Elif:

```
if <Bedingung>:  
    <Block>  
elif <Bedingung2>:  
    <Block2>
```

Else:

```
if <Bedingung>:  
    <Block>  
else:  
    <Block2>
```

Beispiel:

```
let x = -7  
  
if x == 5:  
    echo "x ist Fünf"  
elif bool(x):  
    echo "x ist nicht Null"  
else:  
    echo "x ist Null"
```

When:

läuft

zu Compile-Time

```
when system.hostOS == "windows":  
    echo "running on Windows!"  
elif system.hostOS == "linux":  
    echo "running on Linux!"  
elif system.hostOS == "macosx":  
    echo "running on Mac OS X!"  
else:  
    echo "unknown operating system"
```


BEDINGUNGEN

Case-Statement:

```
case x:
of <case1>:
    <Block1>
of <case2>:
    <Block2>
of <case3>:
    <Block3>
else:
    <Block4>
```

Case-Statement Beispiel:

```
case x:
of 1:
    echo "x ist Eins"
of 2..5:
    echo "x ist zwischen Zwei bis Fünf"
of 0,6:
    echo "x ist entweder Null oder Sechs"
of 7..9,13..20:
    echo "x ist etwas"
else:
    echo "was weiß ich"
```

Besonderheiten:

Alle Fälle müssen abgedeckt werden aber dürfen nicht doppelt vorkommen. Ansonsten wird ein Kompilierfehler geworfen. Bei einem Treffer wird der Rest nicht mehr ausgeführt

Mehrere Werte können in einem `of` kombiniert werden

z.B. 0..7 für 0 bis 7 oder 8,10 für 8 oder 10

SCHLEIFEN

While-Loop:

```
while <Bedingung>:  
    <Block>
```

```
var num = 0  
while num < 100:  
    echo num  
    num += 1
```

For-Loop:

```
for <Element> in <Collection>:  
    <Block>  
  
# <Block> = echo i  
for i in countdown(10,1): # 10 9 8 7 ...  
  
for i in countup(1,10): # 1 2 3 4 ...  
  
for i in 0..10: # 0 bis 10  
for i in 0..<10: # 0 bis 9  
  
# Quasi For-Each Loops  
  
for i in [4,2,3,4]: # 4 2 3 4  
  
for idx, c in s[0 .. ^1]: # s Inhalt als Slice mit Index  
    echo idx, ": ", c  
  
for idx, c in s: # s Inhalt mit Index  
    echo idx, ": ", c
```

Block, Break, Continue

Block:

Definiert einen neuen Block mit Scope

```
block myblock:  
    var x = "hi"  
echo x # geht nicht
```

Break:

Beendet frühzeitig einen Block

```
while true:  
    echo "in der Loop"  
    break  
    echo "immer noch in Loop"  
echo "raus"  
  
# in der Loop  
# raus
```

Continue:

springt zur nächsten Iteration der Schleife

```
for i in 1 .. 5:  
    if i <= 3: continue  
    echo i # 4 5
```

PRIMITIVE DATENTYPEN IN NIM

Datentyp	Beschreibung	Beispielwerte
Integer	Ganze Zahlen ohne Dezimalstellen.	32, -174, 0, 10_000_000
Float	Gleitkommazahlen, die reelle Zahlen approximieren.	2.73, -3.14, 4e7
Char	Einzelne ASCII-Zeichen, dargestellt zwischen zwei einfachen Anführungszeichen.	'a', '+', '2' 'ab' -> error
String	Eine Folge von Zeichen, dargestellt zwischen zwei doppelten Anführungszeichen.	"Hallo Welt", "", "32"
Boolean	Wahrheitswerte, die nur true oder false annehmen können.	true, false

PRIMITIVE DATENTYPEN IN NIM

bool



```
if zahl > 3 and zahl < 10:
  #....

if input = "Yes" or input = "yes"
  #....
```

int / float



```
var
  x1 = 0
  y1 = 0'i8
  z1 = 0'i32
  u = 0'u

var
  x2 = 0.0
  y2 = 0.0'f32
  z2 = 0.0'f64
```

div / mod [int]

String / char



```
var
  str1 = "Hello"
  str2 = " World"
  str = str1 & str2

str.add($'!')

echo len(str)
echo str[11]
```

AUSGABE
12
!

Type Conversion



```
var
  x: int = 2.5.int
  y: int = int(2.5)
```

ADVANCED DATATYPES

ARRAYS



```
var my_array: array[5, int] = [3, 8, 4, 0, 1]  
my_array[1] = 5
```

```
echo my_array
```

```
var ndim_array: array[2, array[3, int]] = [[2, 4, 6], [1, 3, 5]]
```

```
echo ndim_array
```

```
echo ndim_array[0][1]
```

```
var a = [3, 8, 4, 0, 1]
```

AUSGABE

```
[3, 5, 4, 0, 1]
```

```
[[2, 4, 6], [1, 3, 5]]
```

```
4
```

ADVANCED DATATYPES

SLICES



```
var my_array = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
echo my_array[0 .. 6]
```

```
echo my_array[2 ..< 5]
```

```
echo my_array[3 .. ^2]
```

$\triangleq$ `echo my_array[3 .. my_array.len()-2]`

AUSGABE

```
@[1, 2, 3, 4, 5, 6, 7]
```

```
@[3 ,4 ,5]
```

```
@[4, 5, 6, 7, 8]
```

ADVANCED DATATYPES

SEQUENCES



```
var my_seq: seq[string] = @["Hello", "Nim"]
my_seq.add("!")
my_seq.del(1)
my_seq.insert("World", 1)

echo my_seq
echo my_seq[1]
```

var a = @["Hello", "Nim"]

AUSGABE

```
@["Hello", "World", "!"]
World
```



```
import sequtils

var
  a = @[1, 2, 3, 4]
  b = @[4, 5, 6, 7]
  c = concat(a, b)

echo c.deduplicate()
echo c.filter(proc(x: int): bool = x < 5)
c.keepIf(proc(x: int): bool = x < 5)
echo c
echo c.map(proc(x: int): int = x * 2)
```

AUSGABE

```
@[1, 2, 3, 4, 5, 6, 7]
@[1, 2, 3, 4, 4]
@[1, 2, 3, 4, 4]
@[2, 4, 6, 8, 8]
```


ADVANCED DATATYPES

TUPLES



```
var
  my_tuple: (string, float) = ("Nim", 2.0)
  person1: tuple[name: String, age: int] = ("Daniel", 19)

echo my_tuple
echo my_tuple[0]

echo person1
echo person.name
```

```
var a = ("Nim", 2.0)
```

AUSGABE

("Nim", 2.0)

Nim

(name: "Daniel", age: 19)

Daniel

ADVANCED DATATYPES

ENUMS



```
type
    Color = enum
        RED, BLUE, GREEN
```

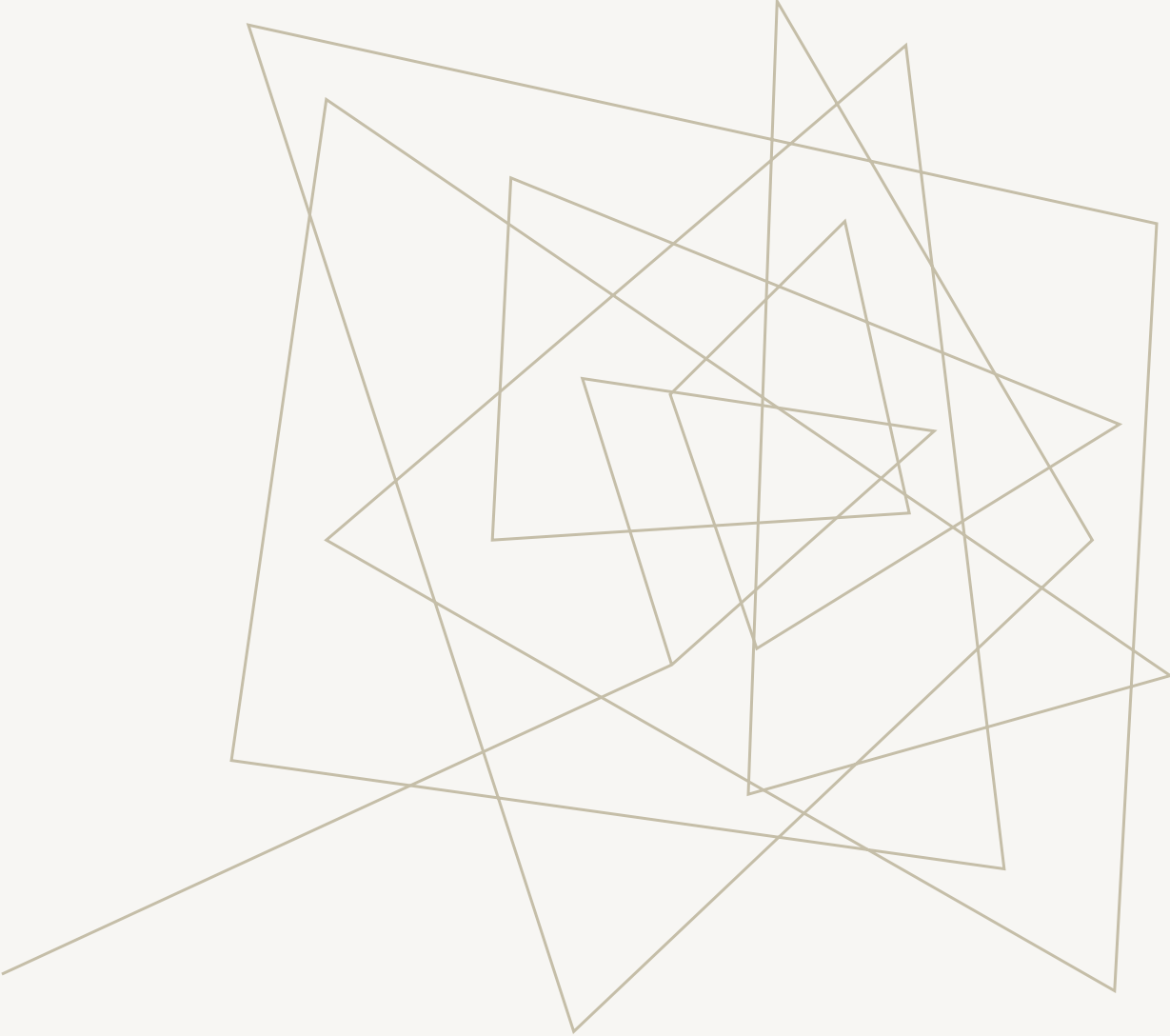
```
var textColor = Color.RED
echo Color
echo Color.RED
```

```
var textColor: Color = Color.RED
    oder
var textColor = RED
```

AUSGABE

Color

RED



LIVE AUFGABE

ADVANCED
DATATYPES

PROZEDUREN

Syntax:

```
proc <name>(<param1>:<typ1>, <param2>:<typ2>, ...): <returnTyp> =  
    <prozedur_block>
```

Beispiel:

```
proc findeMax(x: int, y: int): int =  
    if x > y:  
        return x  
    else:  
        return y  
#in der Prozedur  
#außerhalb der Prozedur
```

Void-Prozeduren:

```
proc echoModulBewertung(modul: string) =  
    case modul  
    of "pr3", "Pr3", "PR3":  
        echo modul, "ist cool!"  
    else:  
        echo modul, "ist ganz in Ordnung."
```

PROZEDUREN

Veränderlichkeit von Parametern:

- Normalerweise können Parameter innerhalb einer Prozedur nicht verändert werden

```
proc ändereParam(param: int) =  
  param += 5  
  
var param = 10  
echo(param)  
ändereParam(param)  
echo(param)
```

Fehlermeldung:

```
expression 'param' is immutable, not 'var'
```

➔ Wenn eine Änderung erforderlich ist, muss eine neue Variable mit `var` im Prozedurkörper deklariert werden

Beispiel:

```
proc divmod(a, b: int; ergebnis, rest: var int) =  
  ergebnis = a div b # Ganzzahl-Division  
  rest = a mod b     # Modulo-Operation  
  
var x, y: int  
divmod(8, 5, x, y) # ändert x und y  
echo x # Ausgabe: 1  
echo y # Ausgabe: 3
```

PROZEDUREN

In Nim wird die **Uniform Function Call Syntax (UFCS)** unterstützt. Diese Syntax erlaubt verschiedene flexible Möglichkeiten Prozeduren aufzurufen.

Beispiel:

```
proc plus(x, y: int): int = x + y
proc multi(x, y: int): int = x * y

let a = 2
let b = 3
let c = 4

echo a.plus(b) == plus(a, b)      # True
echo c.multi(a) == multi(c, a)    # True

# Verketteter Aufruf
echo a.plus(b).multi(c)  # (2 + 3) * 4 = 20

# Alternativ
echo multi((a.plus(b)), c)  # (2 + 3) * 4 = 20
```

Standard-Aufruf:

procName(arg1, arg2)

UFCS-Stil:

1. arg1.procName(arg2)
2. procName arg1, arg2 (häufig für echo oder len)
3. arg1.procName arg2, arg3 (weniger verbreitet)

PROZEDUREN

Jede Prozedur, die einen Wert zurückgibt, hat eine implizite result-Variable. Diese:

- Wird automatisch zu Beginn der Prozedur deklariert und mit dem Standardwert des Rückgabetyps initialisiert (z.B. 0 für int, false für bool, "" für string, @[] für seq)
- Enthält den Rückgabewert der Prozedur

```
proc findBiggest(a: seq[int]): int =  
  # result = 0  
  for number in a:  
    if number > result:  
      result = number  
  # Ende der proc  
  
let d = @[3, -5, 11, 33, 7, -15]  
echo findBiggest(d)
```

Achtung proc nicht 100% korrekt:

Wenn die Sequenz nur aus negativen Zahlen besteht, würde die Prozedur 0 zurückgeben

Two thin orange lines intersect on the left side of the slide. One line is horizontal, and the other is diagonal, crossing it.

ZUSAMMENFASSUNG RÜCKGABE VON PROZEDUREN

1. result ist implizit: Jede Prozedur mit einem Rückgabewert hat eine automatische result-Variable.

2. Rückgabe ohne return: Wenn man keinen return verwendet, wird der Wert von result automatisch zurückgegeben.

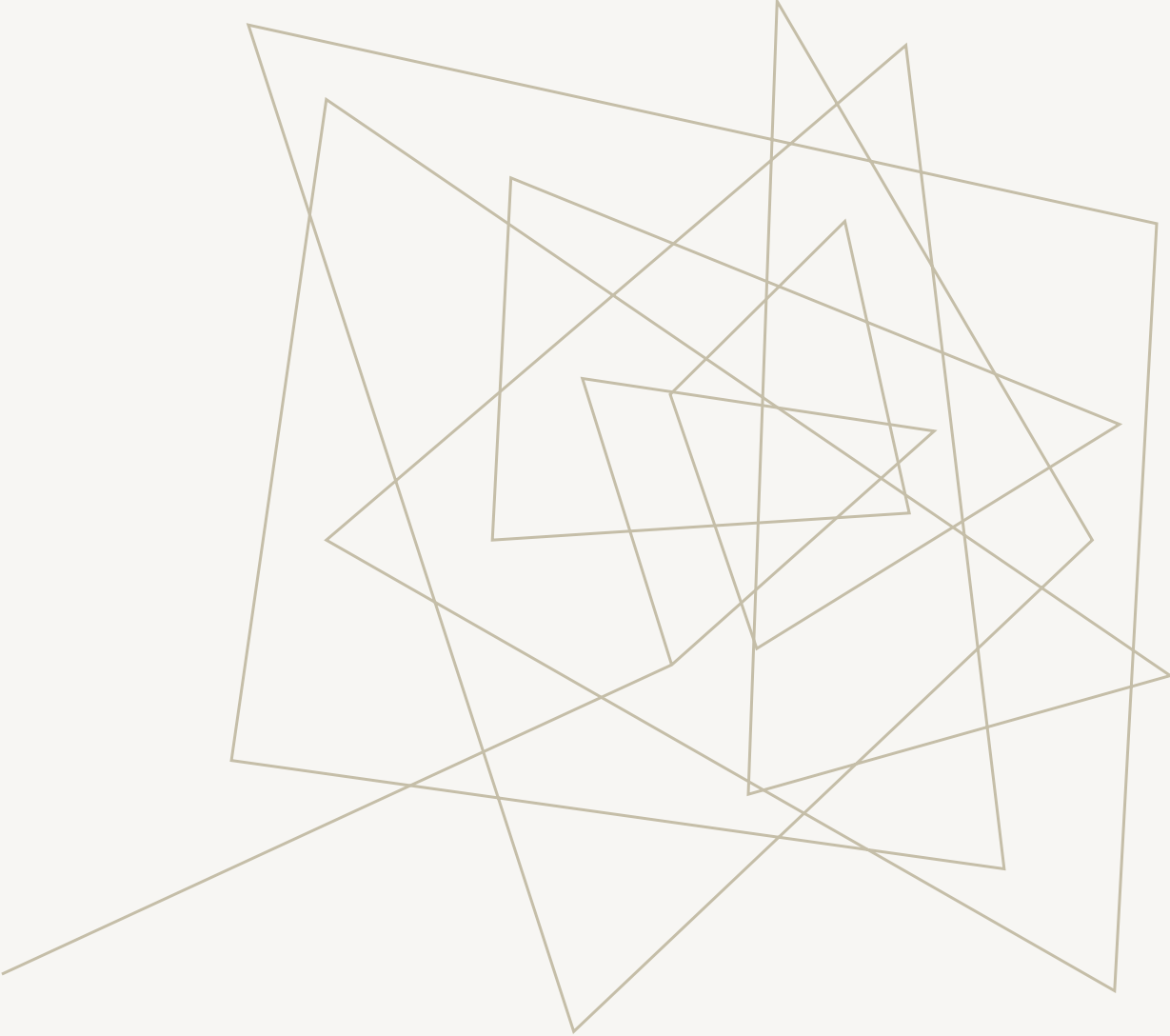
3. Letzter Ausdruck als Rückgabewert: Falls result nicht explizit gesetzt wird, wird der Wert des letzten Ausdrucks zurückgegeben.

4. Manuelles return: Ist optional, aber sinnvoll für vorzeitige Rückgaben.

ÜBERLADEN VON FUNKTIONEN

```
proc toString(x: int): string =  
  result = if x < 0: "negativ"  
           elif x > 0: "positiv"  
           else: "Null"  
  
proc toString(x: bool): string =  
  result = if x: "ja" else: "ein"  
  
echo toString(13)    # "positiv"  
echo toString(true) # "ja"
```

-> Operatoren wie + , - , * sind in Nim auch überladbare Funktionen



LIVE AUFGABE

PROZEDUREN

Abstract geometric lines in the top-left corner of the slide, consisting of several overlapping, thin, light-brown lines forming various polygons and shapes.

OBJEKTORIENTIERTES PROGRAMMIEREN IN NIM

VERERBUNG

- Vererbung in Nim ist optional und wird durch das RootObj aktiviert.
- Vererbung wird oft mit Ref-Objekten verwendet.
- Felder oder Methoden mit * sind Public und damit von anderen Klassen zugänglich.
- Keine mehrfache Vererbung
- Keine Vererbung über Klassen sondern Objekte

```
1  type
2    Person = ref object of RootObj
3      name*: string  # Öffentlich zugänglich
4      age: int       # Privat (ohne *)
5
6    Student = ref object of Person
7      id*: int
8
```

```
1  type
2    Person = object
3      name*: string  # Öffentlich zugänglich
4      age: int       # Privat (ohne *)
5
6    Student = ref object of Person
7      id*: int
8
```

Da Person nicht von RootObj erbt kann Student auch nicht von Person Erben

ÜBERBLICK ZU POINTERN UND REFERENZEN

- Pointer und Referenzen ermöglichen indirekten Zugriff auf Speicher.
- Pointer sind "Low-Level" und erfordern manuelles Dereferenzieren.
- Referenzen (ref) werden automatisch verwaltet und sind sicherer.
- Anwendungsfall: Pointer für Systemprogrammierung
Speicheroptimierungen.
- Referenzen für Objekthierarchien und dynamische Speicherzuweisung.

BEISPIEL CODE ZU POINTERN

```
1  type
2  |   IntPtr = ptr int
3  var p: IntPtr
4  new(p)      # Speicher für Pointer allokieren
5  p[] = 42    # Dereferenzierung
6  echo p[]    # Ausgabe: 42
7
```

BEISPIEL ZU REFERENZEN

```
1  type
2    Person = ref object
3      name: string
4      age: int
5
6  var p: Person
7  new(p)      # Speicher wird allokiert
8  p.name = "Anna"
9  p.age = 25
10 echo p.name # Ausgabe: Anna
```

WANN VERWENDET MAN POINTER UND REFERENZEN?

•Pointer:

- Systemprogrammierung (z.B. Zugriff auf Hardware).
- Direkte Kontrolle über Speicher.
- Performance-optimierte Anwendungen.

•Referenzen:

- Dynamische Speicherzuweisung (Heap).
- Erstellen von Objekthierarchien mit `ref object`.
- Sichere und flexible Handhabung von Objekten.

ERSTELLEN VON OBJEKTEN

- Objektkonstruktion erfolgt mit TypeName(fields...)
- Upcasting erlaubt Zuweisung von Unterklasse zu Basisklasse.

```
1  type
2    Animal = ref object of RootObj
3      species*: string
4
5    Dog = ref object of Animal
6      breed*: string
7
8  var
9    myDog: Dog
10   myAnimal: Animal
11
12  # Objekt erstellen
13  myDog = Dog(species: "Canine", breed: "Labrador")
14
15  # Upcasting
16  myAnimal = myDog
17  echo myAnimal.species # Zugriff auf Felder der Basisklasse
18
19  # Typprüfung
20  if myAnimal of Dog:
21    echo "It's a dog!"
22
```

GEGENSEITIG REKURSIVE TYPEN

- Dies sind Typen, die aufeinander verweisen und voneinander abhängen.
- Ein Beispiel wäre ein Baum (Tree), bei dem jeder Knoten (Node) auf andere Knoten verweist, die wiederum auf weitere Knoten verweisen können.
- Erforderliche Deklaration:
in Nim müssen gegenseitig rekursive Typen in einem einzigen type-Block deklariert werden.

Warum ist das nötig?

Der Compiler muss die Typen vollständig sehen können, um ihre gegenseitige Beziehung korrekt zu verstehen.

- Effiziente Kompilierung:
Indem rekursive Typen in einem einzigen Block deklariert werden, kann der Compiler diese Typen schneller und ohne zusätzliche "Vorausblicke" verarbeiten.
Dies trägt dazu bei, die Kompilation zu optimieren und zu verhindern, dass der Compiler nach Symbolen suchen muss, die nicht sofort verfügbar sind.

BEISPIEL FÜR GEGENSEITIG REKURSIVE TYPEN

```
1  type
2      Node = ref object          # Knoten im Baum
3          le, ri: Node          # Linker und rechter Teilbaum
4          sym: ref Sym          # Blätter enthalten einen Verweis auf ein Symbol
5
6      Sym = object              # Symbol
7          name: string          # Name des Symbols
8          line: int             # Zeile, in der das Symbol deklariert wurde
9          code: Node            # Verweis auf den abstrakten Syntaxbaum
10
```

TYPKONVERTIERUNGEN VS. TYPCASTS

- In Nim gibt es zwei Arten von Typumwandlungen:

1. Typkonvertierungen:

- Wandeln Werte in einen anderen Typ um, bewahren den abstrakten Wert.
- Sicherer, der Compiler prüft die Gültigkeit der Konvertierung.
- Syntax: `Zieltyp(Ausdruck)`
- Fehlgeschlagene Konvertierungen lösen Fehler zur Laufzeit aus.

2. Typcasts:

- Zwingen den Compiler, das Bitmuster anders zu interpretieren.
- Syntax: `cast[Zieltyp](Ausdruck)`

TYPKONVERTIERUNGEN – EINFACHES BEISPIEL

```
1  type
2  |   Person = ref object of RootObj
3  |       name: string
4  |
5  |   Student = ref object of Person
6  |       id: int
7  |
8  |   proc getID(x: Person): int =
9  |       Student(x).id # Konvertierung von Person zu Student
10 |
11 |   var
12 |       s = Student(name: "Anna", id: 123)
13 |       p: Person = s # Upcasting möglich
14 |
15 |   echo getID(p)      # OK: p ist ein Student
16 |
17 |   # Fehler zur Laufzeit:
18 |   var p2 = Person(name: "Tom")
19 |   echo getID(p2)     # InvalidObjectConversionDefect wird ausgelöst!
20 |
```

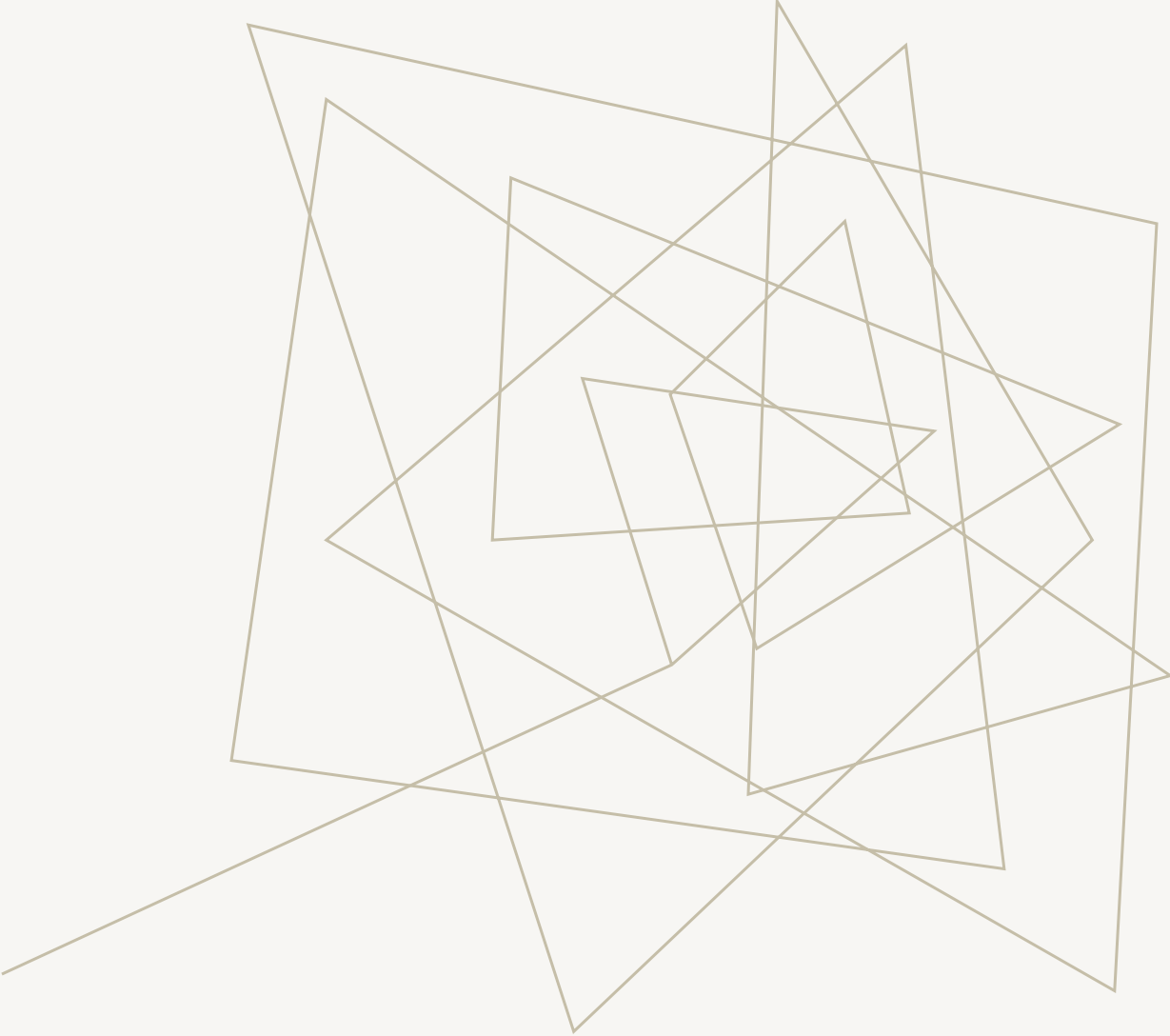
TYPCASTS – BEISPIEL

```
1  var x: int = 65
2  var y: char = cast[char](x)  # Cast von int zu char
3
4  echo y  # Ausgabe: A
5
```

PROPERTIES IN NIM

```
1  type
2  |   Person = object
3  |   |   name: string
4
5  # Setter für den Namen
6  proc `name=`*(p: var Person, newName: string) {.inline.} =
7  |   p.name = newName
8
9  # Getter für den Namen
10 proc name*(p: Person): string {.inline.} =
11 |   result = p.name
12
```

- In Nim gibt es keine speziellen Property-Kywords wie in anderen Programmiersprachen.
- Stattdessen werden Getter und Setter wie gewöhnliche Prozeduren behandelt.
- Getter-Prozeduren können mit der normalen Methodenaufruf-Syntax verwendet werden.
- Setter-Prozeduren benötigen jedoch eine spezielle Syntax mit einem = Operator.



LIVE AUFGABE

OBJEKT ORIENTIERT
PROGRAMMIEREN

EXCEPTIONES

- Exeptions sind Objekte
- Namenskonvention: Exceptiones-Typen enden mit "Error".
- System Module geben Exeptionhierarchie vor

- Exceptiones müssen auf dem Heap allokiert werden (unbekannte Lebenszeit) -> Datentyp „ref“ legt es auf den Heap
- Compiler verhindert, dass Exceptiones vom Stack ausgelöst werden
- Msg-Feld für Begründung sollte ausgefüllt werden

```
var
  e: ref OSError
new(e)
e.msg = "the request to the OS failed"
raise e
```

EXCEPTIONS-RAISE STATEMENT

- Raise ruft eine Exception auf
- Nach dem Raise Statement bedarf es ein Exception-objekt, sonst wird die letzte verwendete Exception re-raised

```
var  
    e: ref OSError  
new(e)  
e.msg = "the request to the OS failed"  
raise e
```

```
raise newException(OSError, "the request to the OS failed")
```

EXCEPTIONES-TRY STATEMENT

- Try Statement bestehen aus mindestens 2 Blöcken
- Try -> Code wird ausgeführt falls kein Fehler „geraised“ wird
- Except -> Code des jeweiligen except Blocks wird ausgeführt
- Finally(Optional) -> wird immer ausgeführt
- `getCurrentException()` -> Methode um das Exception-objekt zu bekommen

```
var
  f: File
if open(f, "numbers.txt"):
  try:
    let a = readLine(f)
    let b = readLine(f)
    echo "sum: ", parseInt(a) + parseInt(b)
  except OverflowDefect:
    echo "overflow!"
  except ValueError:
    echo "could not convert string to integer"
  except IOError:
    echo "IO error!"
  except CatchableError:
    echo "Unknown exception!"
    # reraise the unknown exception:
    raise
  finally:
    close(f)
```

EXCEPTIONS- ANNOTATING PROCEDURES WITH RAISED EXCEPTIONS

- Nim unterstützt eine explizite Fehlerbehandlung
- Explizite Dokumentation von Exceptions, die eine Prozedur oder Funktion auslösen kann
- Falls eine andere Exception geraised wird, stoppt der Compiler an der Stelle des Fehlers
- Vorteile:
 - Bessere Dokumentation
 - Fehlerbehandlung

```
proc complexProc() {.raises: [IOError, ArithmeticDefect].} =  
  ...  
  
proc simpleProc() {.raises: [].} =  
  ...
```

`.raises:[].` -> darf keine Exceptions werfen



GENERIC

Nutzen um allgemeinen und wiederverwendbaren Code zu schreiben, der mit unterschiedlichen Typen arbeiten kann.

Hauptmerkmale:

- **Typsicherheit:** Der Compiler stellt sicher, dass die verwendeten Typen korrekt sind.
- **Flexibilität:** Der Code ist auf unterschiedliche Datentypen anwendbar.
- **Effizienz:** Nim löst Generics zur Kompilierzeit auf, wodurch der generierte Code typenspezifisch ist.

```
proc addInt(a, b: int): int =  
  return a + b  
  
proc addFloat(a, b: float): float =  
  return a + b
```

```
proc add[T](a, b: T): T =  
  return a + b
```

Generics in Nim werden durch Typparameter in eckigen Klammern [T] definiert. Beim Aufruf wird sie durch einen konkreten Typ ersetzt

```
echo add(3, 5)           # Für int  
echo add(3.5, 4.2)       # Für float  
echo add("Hello", " Nim") # Für string (wenn + unterstützt wird)
```

GENERIC CODE

BEISPIEL

Einfache Ausgabe der Eingabe

```
proc identity[T](value: T): T =  
  return value  
  
echo identity(42)           # Ausgabe: 42 (int)  
echo identity("Nim")       # Ausgabe: Nim (string)
```

Vergleich der beiden Parameter

```
proc isEqual[T](a, b: T): bool =  
  return a == b  
  
echo isEqual(5, 5)          # Ausgabe: true  
echo isEqual("Nim", "nim") # Ausgabe: false
```

Mehrere Typparameter

```
proc swap[T, U](a: T, b: U): (U, T) =  
  return (b, a)  
  
echo swap(42, "Nim")       # Ausgabe: ("Nim", 42)
```

DATENSTRUKTUREN UND EINSCHRÄNKUNGEN (CONSTRAINTS)

In Nim kannst du auch generische Typen erstellen, wie z. B. Listen, Arrays oder **eigene Strukturen**.

Generische Objekte

Beispiel eigene Struktur

```
type
  Box[T] = object
    value: T
```

Verwendung

```
var intBox: Box[int] = Box[int](value: 42)
var strBox: Box[string] = Box[string](value: "Hallo")
echo intBox.value    # Ausgabe: 42
echo strBox.value    # Ausgabe: Hallo
```

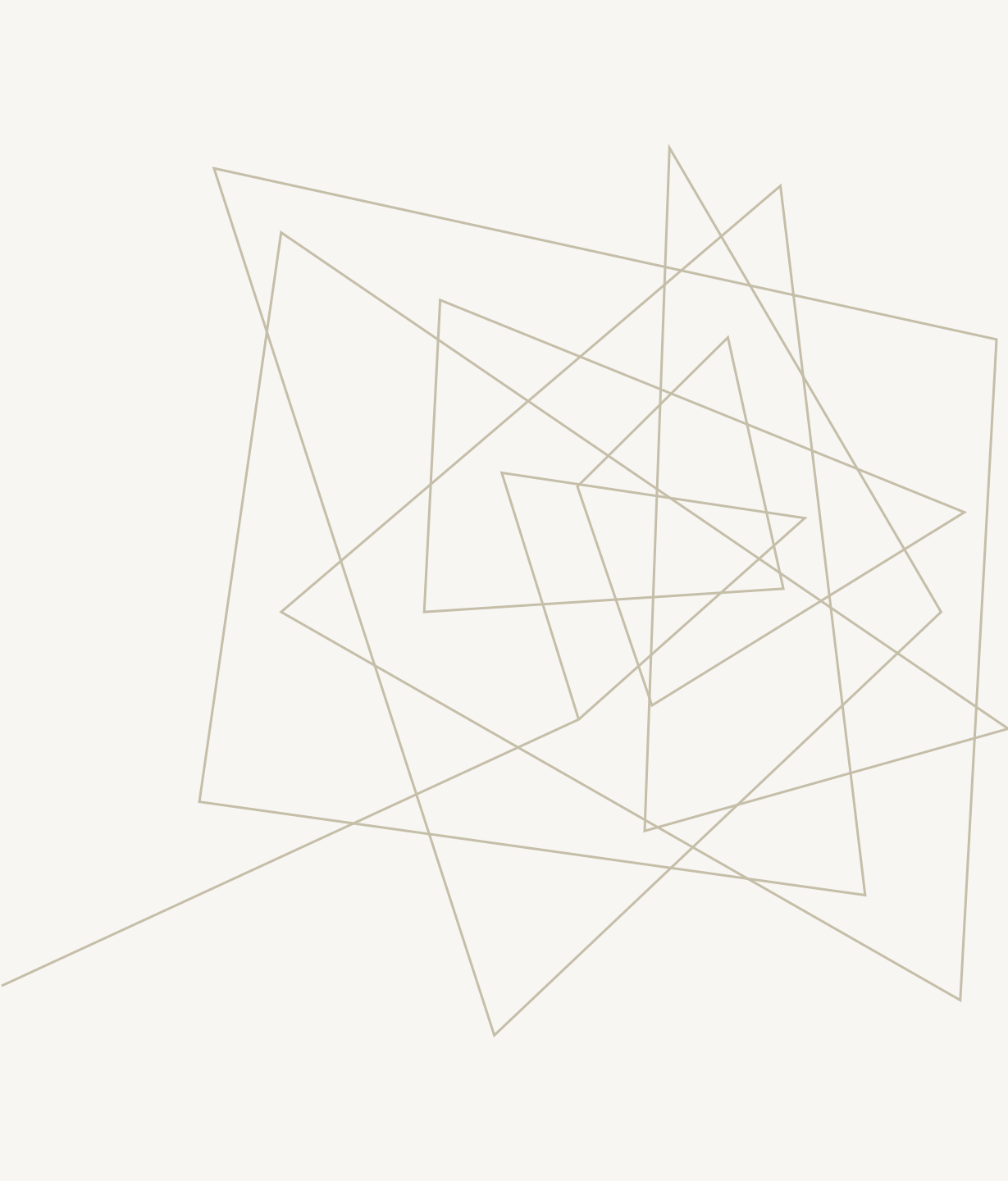
Einschränkungen von Typen (Constraints)

Beispiel eigene Struktur

```
proc addNumbers[T: SomeNumber](a, b: T): T =
  return a + b
```

Verwendung

```
echo addNumbers(3, 5)      # Gültig
# echo addNumbers("3", "5") # Fehler
```


An abstract geometric drawing on the left side of the slide. It consists of several overlapping, irregular polygons and lines drawn in a light brown or tan color. The shapes are nested and intersecting, creating a complex, layered effect. The lines vary in length and orientation, some forming closed shapes while others are open segments.

TEMPLATES

Sind ein Bestandteil der Metaprogrammierung in Nim. Sie ermöglichen es, wiederverwendbaren Code zu schreiben, der während der Kompilierzeit in den aufrufenden Code eingebettet wird.

In Nim werden Templates zur Kompilierzeit expandiert, während normale Funktionen zur Laufzeit ausgeführt werden. Der zentrale Unterschied liegt also in Zeitpunkt und Art der Verarbeitung. Sind auch Typunabhängig.

Templates:

- Der Compiler ersetzt bei der Kompilierung die Template-Verwendung durch den tatsächlichen Code des Templates, angepasst an den Kontext.
- Es handelt sich um eine Art **Code-Generator**, der den finalen Code direkt in das Programm einfügt.
- Nach der Kompilierung gibt es keine Spur mehr

Normale Funktionen:

- Der Code bleibt als separater Baustein (z. B. eine maschinenlesbare Funktion) erhalten und wird erst zur Laufzeit ausgeführt.
- Sie benötigen zusätzlichen Overhead, wie z. B. die Arbeit des Funktionsaufrufs, Verwalten von Parametern. Einfacherer wäre direkt $a*2$

```
template TemplateName(parameters): ReturnType =  
  # Code des Templates
```

ReturnType ist optional

Einfaches Template

```
template printMessage(message: string) =  
  echo "Nachricht: ", message  
  
# Verwendung:  
printMessage("Hallo, Nim!") # Ausgabe: Nachricht: Hallo, Nim!
```

Einfache Berechnungen

```
template square(x) =  
  (x * x)  
  
# Verwendung:  
let result = square(5)  
echo result # Ausgabe: 25
```

Hier wird der Code `(x * x)` direkt eingebettet, sodass kein Funktionsaufruf erfolgt.

Templates mit Typinferenz

Templates können generisch sein und automatisch den Typ der Parameter ableiten

```
template printTwice(x) =  
  echo x  
  echo x  
  
# Verwendung:  
printTwice(42)      # Ausgabe: 42 42  
printTwice("Nim")   # Ausgabe: Nim Nim
```



MACROS

Makros in Nim sind Werkzeuge, die zur Compile-Zeit laufen und Code generieren oder transformieren, indem sie direkt auf der Abstract Syntax Tree (AST) des Programms arbeiten

Makro Argumente

Untyped Arguments

- Syntaxbaum wird unverändert an das Makro übergeben
- Vorteile:
 - Syntaxbaum ist vorhersehbar und weniger komplex
- Nachteile:
 - Funktioniert nicht gut mit Nim's Overloading

Typed Arguments

- Syntaxbaum wird semantisch geprüft:
 - Identifier werden zu Symbolen aufgelöst
 - Typinformationen sind sichtbar
- Vorteil:
 - Liefert genauere Syntaxbäume

```
macro foo(arg: int): untyped =  
  echo arg.type # Typinformation verfügbar
```

Makro Argumente

Static Arguments

- Übergibt Werte als Werte, nicht als Syntaxbäume

```
macro myMacro(arg: static[int]): untyped =  
  echo arg # Ergebnis: ein int-Wert (z. B. 7)
```

Code-Blöcke als Argumente

- Code-Blöcke können als letzte Argumente in einem Funktionsaufruf genutzt werden

```
echo "Hello ":  
  let a = "Wor"  
  let b = "ld!"  
  a & b
```

Der Syntaxbaum (AST)

Syntaxbäume sind zentrale Elemente für Makros
Analyse-Tools:

- **treeRepr**: Gibt Syntaxbäume als String aus
- **dumpTree**: Debugging von Syntaxbäumen

```
dumpTree:  
  var mt: MyType = MyType(a:123.456, b:"abcdef")  
  
# output:  
#   StmtList  
#     VarSection  
#       IdentDefs  
#         Ident "mt"  
#         Ident "MyType"  
#       ObjConstr  
#         Ident "MyType"  
#         ExprColonExpr  
#           Ident "a"  
#           FloatLit 123.456  
#         ExprColonExpr  
#           Ident "b"  
#           StrLit "abcdef"
```

Eigene Makros schreiben

Custom Semantic Checking

- Eingabe validieren:
 - **expectKind**: Überprüft den Knotentyp
 - **expectLen**: Überprüft die Anzahl der Knoten

```
macro myAssert(arg: untyped): untyped =  
  arg.expectKind nnkInfix
```


Eigene Makros schreiben

Generierung von Code

- Zwei Methoden:
 1. Manuell mit **newTree** und **newLit**:
 - Kontrolle über Syntaxbaum
 2. **quote do::**
 - Wörtliches Schreiben des generierten Codes

```
macro myMacro(arg: untyped): untyped =  
  result = quote do:  
    echo `arg`
```

Beispiel Codegenerierung mit Makros

```
macro myAssert(arg: untyped): untyped =  
  let op = newLit(" " & arg[0].repr & " ")  
  let lhs = arg[1]  
  let rhs = arg[2]  
  result = quote do:  
    if not `arg`:  
      raise newException(AssertionDefect, `$lhs` & `op` & `$rhs`)
```

Generierter Code:

```
if not (a != b):  
  raise newException(AssertionDefect, $a & " != " & $b)
```

A series of thin, light brown lines forming an abstract geometric pattern on the left side of the slide. The lines intersect to create various polygons and shapes, extending from the top left towards the bottom left.

VIELEN DANK