Programmieren 1

Einführung

Human-Computer Interaction Group

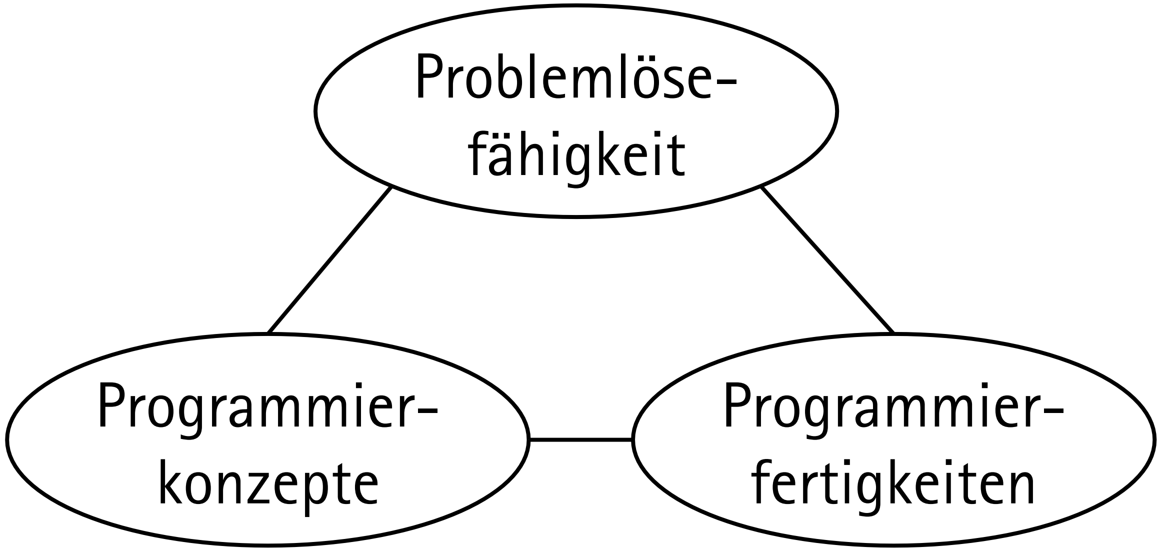
Prof. Dr. Michael Rohs
michael.rohs@hci.uni-hannover.de

Programmierausbildung

Studiengang	1. Semester	2. Semester	3./4. Semester
Informatik	Programmieren I (PostFix, C), 5 LP Rohs	Programmieren II (Java) Becker	Programmier- praktikum von Voigt
Techn. Inf.	Programmieren I (PostFix, C), 5 LP Rohs	Programmieren II (Java) Becker	Programmier- praktikum TI Olbrich

Lernziele

- grundlegende Programmierkonzepte und Methoden kennen und verstanden haben
- algorithmisches Denken einüben
- Programmierkompetenz und Programmierfertigkeiten erlangen
- Fähigkeit des systematischen Entwurfs von einfachen Programmen
- Fähigkeit des Strukturierens von einfachen Programmierproblemen



Personen

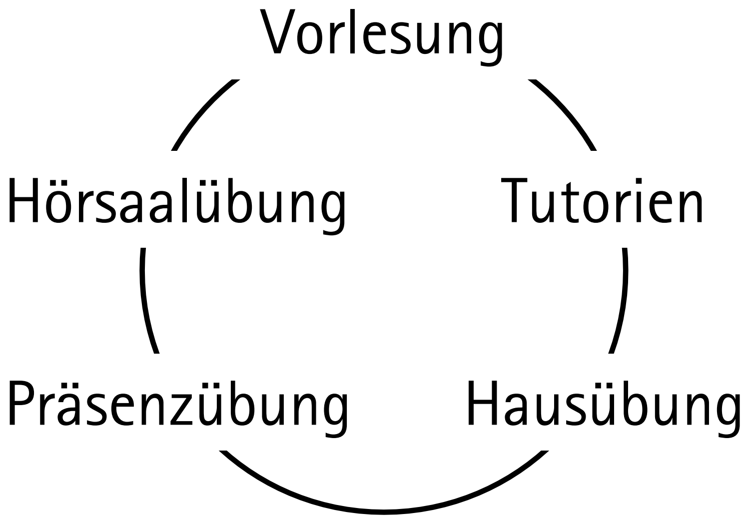
- Vorlesung

  Prof. Dr. Michael Rohs

- Übung

  Tim Dünte, M.Sc.

  Jan Feuchter, M.Sc.



- Tutoren

Yazan Alkhatib	Efe Erdal	Felix Plamper	Kevin Schumann
Patrick Bastek	Jan Habe	Niklas Rabe	Benjamin Simon
Viktor Boos	Julian Helmsen	Finn Reeger	Alexandro Steinert
Jan Dukart	Sebastian Knackstedt	Bircan Sahin	Leo Thern
	Lukas Nolting	Bastian Schmidt	

Emails, Sprechstunde

- Emailanfragen
  - programmieren1@hci.uni-hannover.de

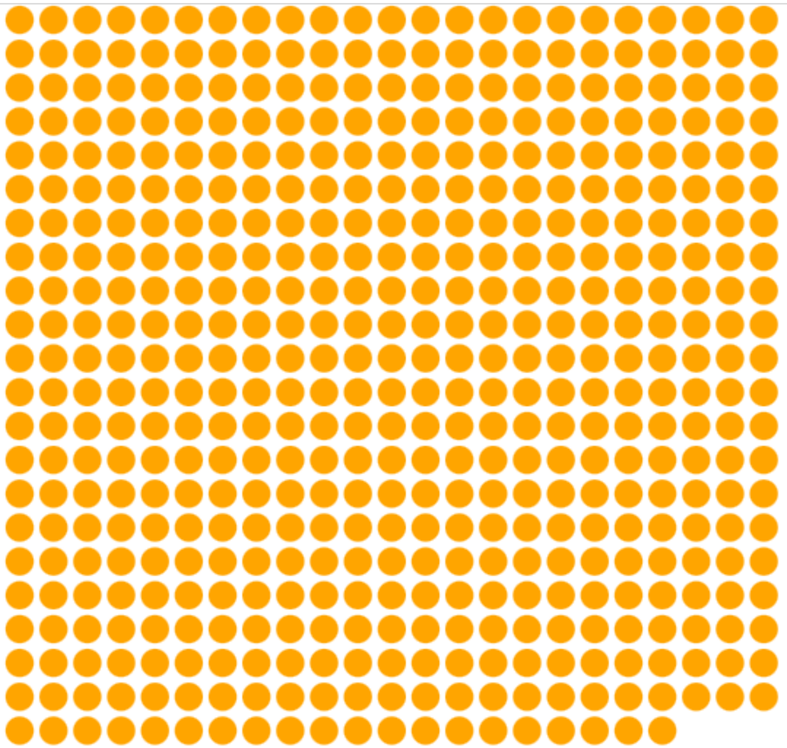
- Sprechstunde
  - was sich nicht per Email klären lässt
  - montags 9-11 Uhr
  - Tel.: 0511 / 762 2435 (Michael Rohs)

Teilnehmerinnen und Teilnehmer

- 585 Teilnehmerinnen und Teilnehmer (Stud.IP, 13.10., 15:00 Uhr)
- B.Sc. Informatik 225 (313), B.Sc. Technische Informatik 34 (57)

  Stand: 6.10.2022 (Vergleichszahlen: 2021)

- Informatik, Bachelor
- Technische Informatik, Bachelor
- Juniorstudium Elektrotechnik
- Juniorstudium Informatik
- Gasthörendenstudium
- Nebenfächler



```
"Participants" 800 800 654 [
    on-draw: (participants) {
        dot: [overlay: [[circle: 20 "orange"] [circle: 24 "white"]]] !
        cols: participants sqrt ceil int !
        row: [beside: cols { pop dot } array] !
        full-rows: participants cols div !
        cols-last-row: participants full-rows cols * - !
        field: [above: full-rows { pop row } array] !
        cols-last-row 0 > {
            row: [beside: cols-last-row { pop dot } array] !
            field: [above: [field row] ] !
        } if
        field
    } lam
] show
```

Vorlesung und Hörsaalübung

- Vorlesung, Pause, Hörsaalübung im Kino


- Folien auf Stud.IP
- Vorlesungsaufzeichnung auf Stud.IP (ELSA)

Vorlesung

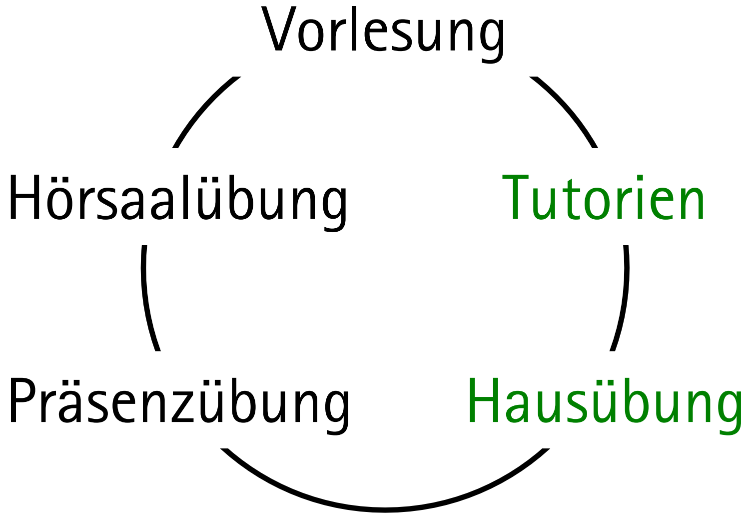
Hörsaalübung Tutorien

Präsenzübung Hausübung

Hausübungen und Tutorien

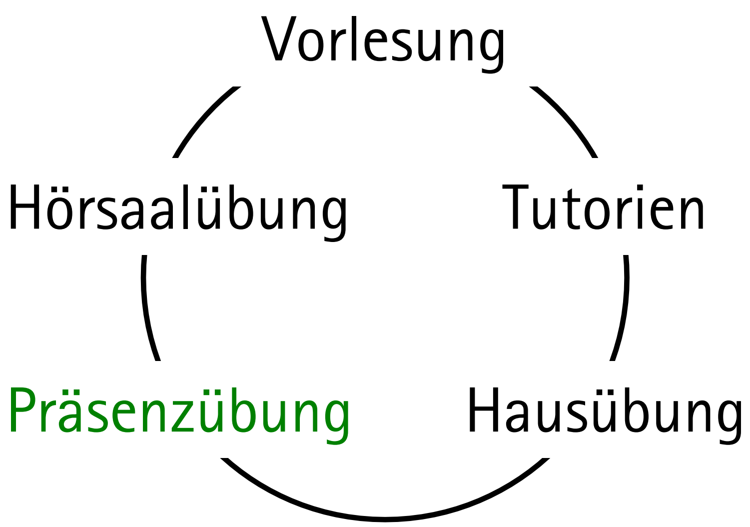
- Bearbeitung Hausübungen im Zweierteam
  - Ausgabe Freitag (ab 14.10.)
  - Abgabe bis folgenden Donnerstag 23:59 Uhr
  - https://assignments.hci.uni-hannover.de

- Tutorien: Besprechung der Abgabe mit Tutor
  - im Rechnerraum HG F411
  - Woche nach der Abgabe, ab 24.10.
- Zweierteams
  - Tutorien 17.-21.10.: Tipps und Partnersuche
  - Wechsel des Tutoriums nur aus triftigem Grund möglich



Präsenzübungen

- Appelstr. 11, Raum A145
- ca. 4 kurze Programmieraufgaben werden auf Papier (oder eigenem Laptop) bearbeitet
- mit Tutor*in durchgesprochen
- ab 14.10. (heute)
- Zeitplanung
  - 1h für Kleingruppe
  - 4 Aufgaben (15 Min. je Aufgabe)
  - 10 Min. Bearbeitung + 5 Min. Besprechung



Lectures

#	Date	Topic	HÜ→	HÜ←
1	14.10.	Organization, computers, programming, algorithms, PostFix introduction (execution model, IDE, basic operators, booleans, naming)	1	20.10. 23:59
2	21.10.	PostFix (primitive types, functions, parameters, local variables, tests), recipe for atomic data	2	27.10. 23:59
3	28.10.	PostFix (operators, array operations, string operations), recipes for enumerations, intervals, and itemizations	3	3.11. 23:59
4	4.11.	Recipes for compound and variant data, iteration and recursion, PostFix (loops, association arrays, data definitions)	4	10.11. 23:59
5	11.11.	C introduction (if, variables, functions, loops), Programming I C library	5	17.11. 23:59
6	18.11.	Data types, infix expressions, C language (enum, switch)	6	24.11. 23:59
7	25.11.	Compound and variant data, C language (formatted output, struct, union)	7	1.12. 23:59
8	2.12.	C language (arrays, pointers) arrays: fixed-size collections, linear and binary search	8	8.12. 23:59
9	9.12.	Dynamic memory (malloc, free), recursion (recursive data, recursive algorithms)	9	15.12. 23:59
10	16.12.	Linked lists, binary trees, search trees	10	22.12. 23:59
online → 11	23.12.	C language (program structure, scope, lifetime, linkage), function pointers, pointer lists	11	12.1. 23:59
12	13.1.	List and tree operations (filter, map, reduce), objects, object lists	12	19.1. 23:59
13	20.1.	Dynamic data structures (stacks, queues, maps, sets), iterators, documentation tools	(13)	
14	27.1.	C language (remaining C keywords), finite state machines, quicksort	(14)	

Ressourcen

- Webseite
  - https://www.hci.uni-hannover.de/de/lehre/lehrveranstaltungen/winter-2022/programmieren-1
  - PostFix Enführung, PostFix Entwicklungsumgebung
  - Design Recipes, Programming I C Library
- Stud.IP
  - https://studip.uni-hannover.de
  - zu Vorlesung und Übung in Stud.IP eintragen
  - Folien, Übungsblätter, Diskussionsforum
- Abgabe der Hausübungen
  - https://assignments.hci.uni-hannover.de

erfolgreiche Teilnahme an Programmieren 1

- erfolgreiche Teilnahme an der Übung → Studienleistung
  - jede Übung muss im Zweierteam mit Tutor besprochen worden sein
  - jede Übung muss mit mindestens einem Punkt bewertet worden sein
  - jedes Teammitglied darf sich einmal vom Partner vertreten lassen
- erfolgreiche Teilnahme an der Klausur → Prüfungsleistung
  - unbenotet, 90 Minuten
  - Programmier- und Wissensaufgaben
  - Bonus: eine Aufgabe darf weggelassen werden
  - Teilnahme an Klausur auch für Nebenfächler notwendig,
    ggf. als Testat zur Erlangung der Studienleistung

INTRODUCTION

Computer Science

"~~computer science equals programming~~"

"the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application"

Denning, Comer, Gries et al.: Report on the ACM Task Force on the Core of Computer Science. New York: ACM Press, 1988.

Programming

- Describing a problem so precisely that a computer can solve it
- Developing programs
  - Programs consist of data (operands) and instructions (operators)
- Fundamental skill of computer scientists
  - Engineering
  - Creativity
- Learning to program requires practice
- Learning to program sharpens your analytical skills
  - Analyzing a problem, exploring possible solutions, evaluating a solution
  - Carries over to other domains

Programming

Windows,
macOS,
Linux

computer

real world,
application
domain

programming
language

C, Java, Python

how to design
programs

programmer

compiler,
tools

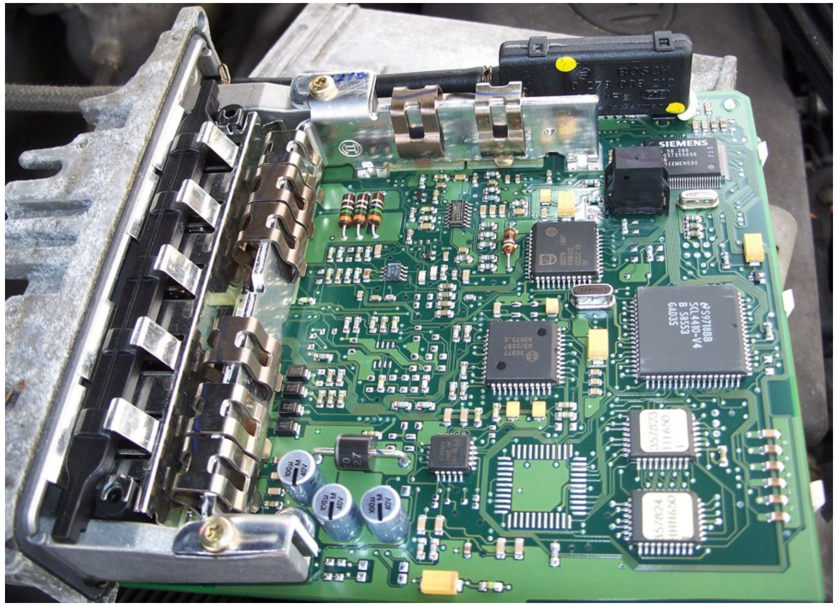
gcc, make,
text editor,
command
line

Nachbardiskussion (2 Minuten)

- Wann wurde das erste Programm geschrieben?

Programmed/Programmable Systems are Ubiquitous

- Banking transaction systems
- Airline booking and baggage handling systems
- Automotive control systems, engine control units
- Medical devices
- Mobile phones and smartwatches
- The Internet
- Ticket vending machines
- Washing machines
- Elevators
- Traffic lights
- etc.



https://de.wikipedia.org/wiki/Motorsteuerung#/media/File:Motorsteuerung_VW_Golf_TDI_innen.jpg

Our Society Relies on Programmed/Programmable Systems

Responsibility of Computer Scientists:

- Correctness
- Reliability
- Efficiency
- Security
- Safety
- Usability
- Accessibility
- Maintainability
- etc.

POSTFIX

A BEGINNER'S PROGRAMMING LANGUAGE & WORKBENCH

The PostFix Programming Language

- A simple programming language
  - Small number of concepts
  - Simple syntax, general rules
  - Simple execution model
- Web-based development environment
  - Programs can be executed step-by-step
  - Execution state is always fully visible
  - Individual instructions can be tested immediately
- Development environment
  - https://postfix.hci.uni-hannover.de
- Language tutorial
  - https://postfix.hci.uni-hannover.de/postfix-lang.html
- Questions, suggestions, bug reports
  - postfix@hci.uni-hannover.de
- Helps to understand fundamental programming concepts

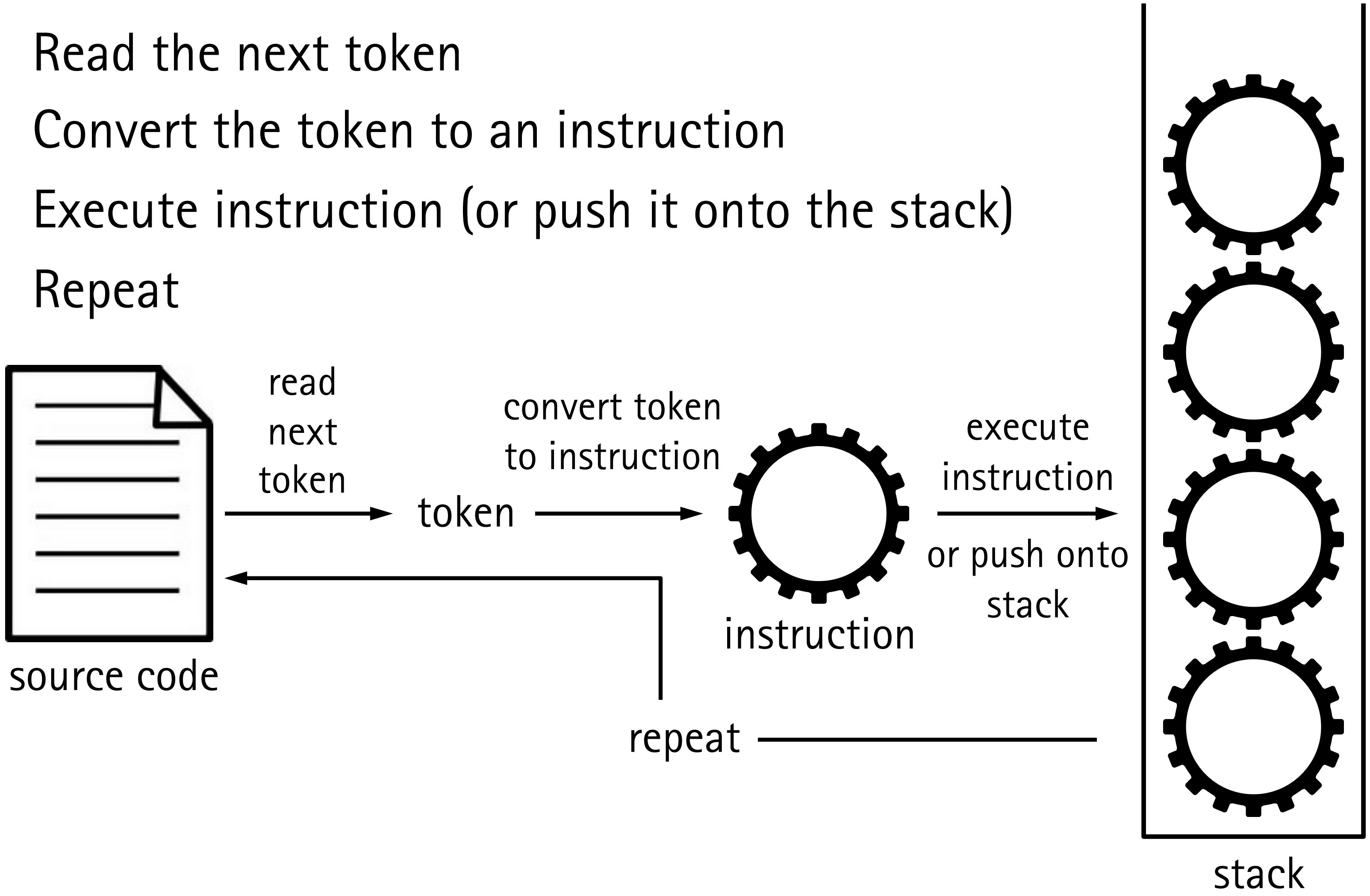
PostFix: Program Text to Sequence of Instructions

- PostFix programs are text (source code)
- PostFix interpreter reads PostFix source code and executes the instructions
- Characters $\xrightarrow{\text{grouped to}}$ Tokens $\xrightarrow{\text{converted to}}$ Instructions $\xrightarrow{\text{executed}}$ Effect
  - Program text consists of individual characters
  - Characters are grouped into words ("tokens")
  - Tokens that the interpreter knows are converted to instructions
    - Example: "12" is converted to an number instruction
    - Example: "mod" is converted to a modulo instruction
  - Executing instructions has an effect on the state of the program

`12␣34␣mod` (9 characters)
`12 34 mod` (3 tokens)
(3 instructions)

The PostFix Execution Model

1. Read the next token
2. Convert the token to an instruction
3. Execute instruction (or push it onto the stack)
4. Repeat

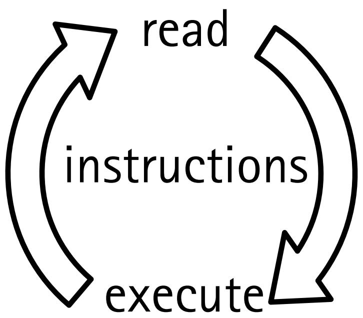


stack stores instructions
or intermediate results

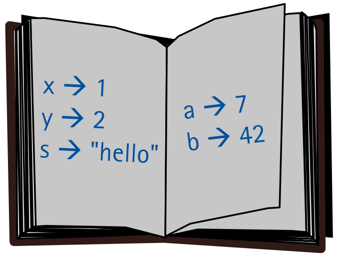
executing an instruction
has an effect on the stack

PostFix Execution: Program Steps

- PostFix executes programs in small steps
  - Step: read instruction, execute instruction
  - Each step changes the state of the program
- Instructions process data
  - Instruction: operator
  - Data: operand
- The program state consists of
  - the next instruction to execute
  - the operands on the runtime stack
  - the entries in the current dictionary
- Programs may process input
  and may produce output



stack of operands

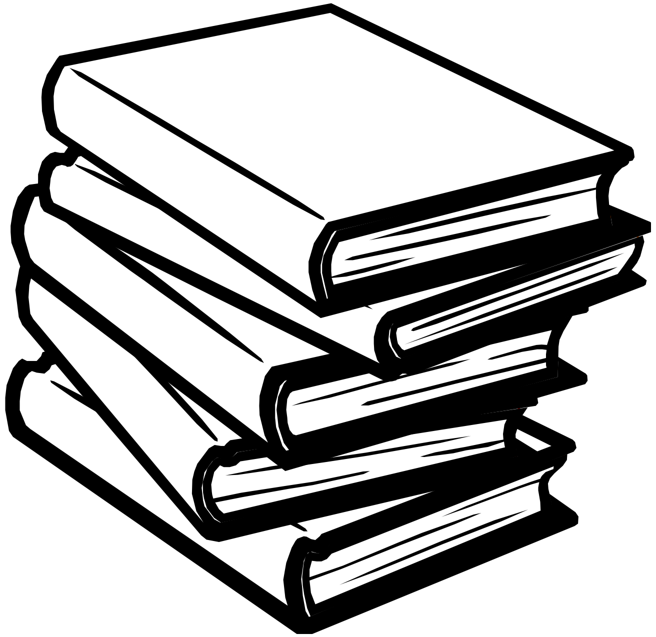


named values

PostFix Execution: Stack

- Operand stack
  - Intermediate results are stored on a stack
  - Push: Place another item on top
  - Pop: Remove topmost item and return it
- Stack writing convention
  - Rotated 90° clockwise, top of stack right
  - Example: (bottom) 1 2 3 (top)
    (stack with three items, 3 is topmost item)
- Operators get operands from stack and push result
  - Initial stack: 1 2 3
  - Steps of "+" operator: pop 3, pop 2, add, push 5
  - Final stack: 1 5

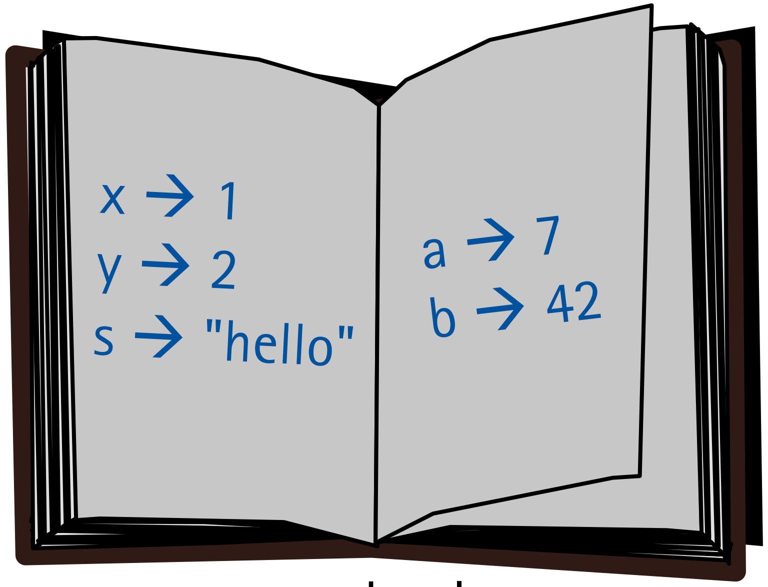
3
2 → 1 2 3
1



stack of operands

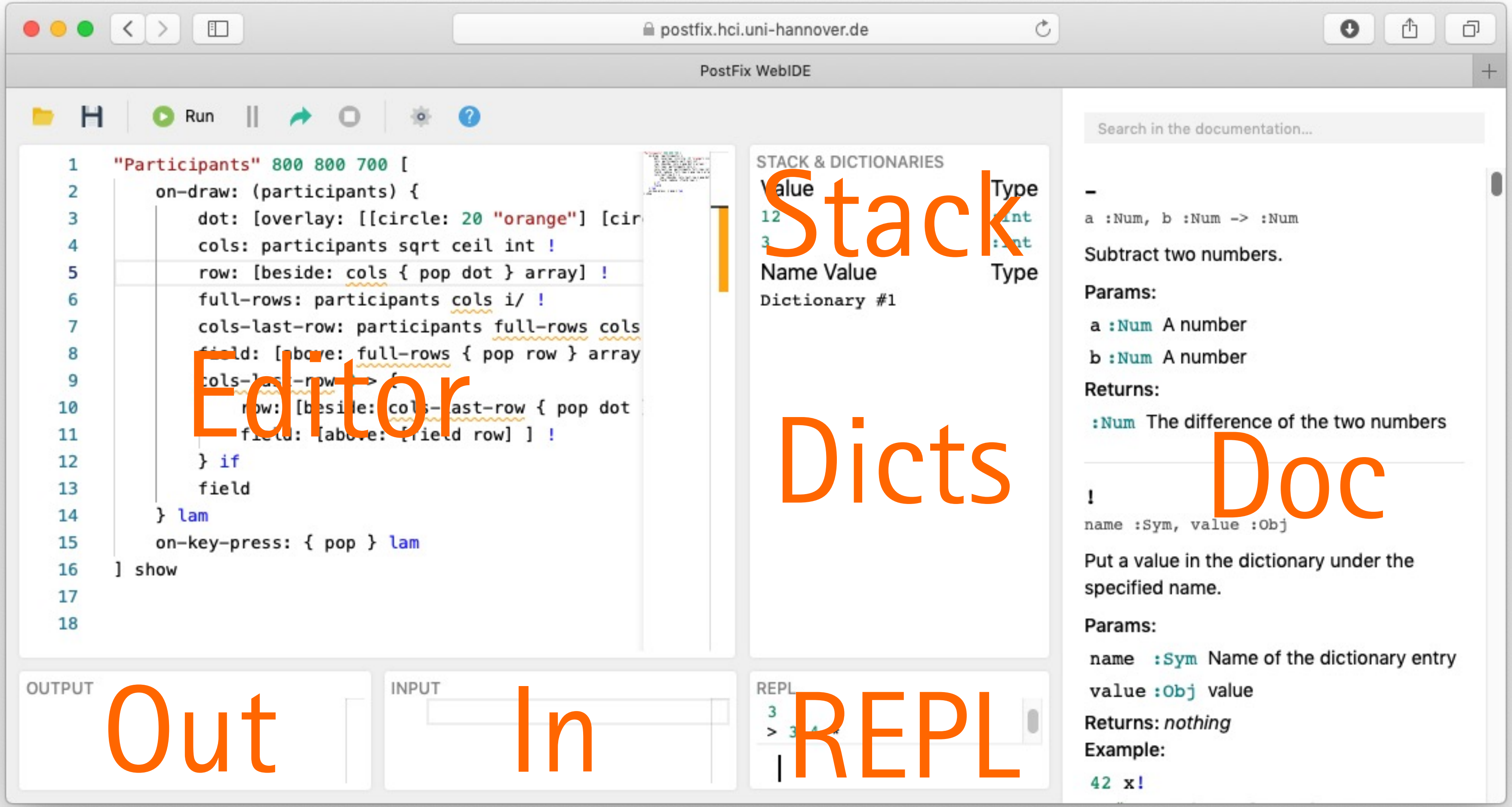
PostFix Execution: Dictionary

- Dictionaries store name-value pairs
  - Allows giving names to values and looking up values by name
  - Put: Write a new name-value pair into the dictionary (e.g., x → 1)
  - Get: Look up a value by its name

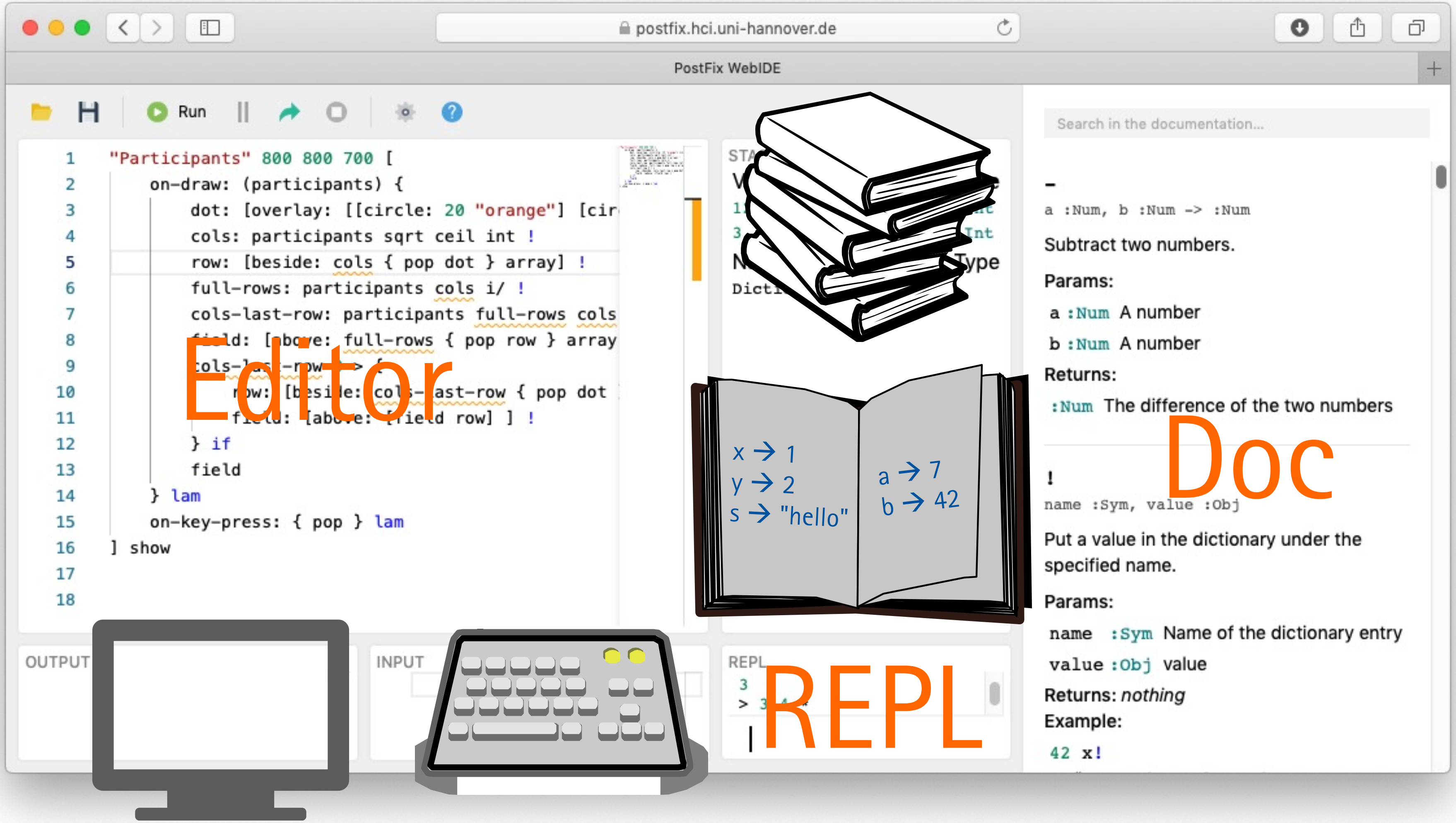


named values

PostFix – Integrated Development Environment (IDE)
https://postfix.hci.uni-hannover.de



PostFix – Integrated Development Environment (IDE)
https://postfix.hci.uni-hannover.de



Arithmetic Operators

```
> 100 200 +   # addition
300
> 20 3 *      # multiplication
60
> 3 2 /       # division
1.5
> 8 5 div     # integer division
1
> 8 5 mod     # modulo function (rest of division)
3
```

Program Elements

- Arithmetic operators
- Logical operators
- Relational and equality operators
- Input and output operators

- Assignment (of a value to a name)
- Selection (conditional execution, if)
- Iteration (repeated execution, loop)
- Sequence (block of code to be executed as a unit)

Boolean Values and Logical Operators

Boolean values: true, false

Logical operators: and, or, not

```
> true true or
true
> true false or
true
> false true or
true
> false false or
false
```

```
> true true and
true
> true false and
false
> false true and
false
> false false and
false
> true not
false
> false not
true
```

Relational Operators and Equality Operators

```
1 1 =    # true
1 2 =    # false
1 1 !=   # false
1 2 !=   # true
1 2 <    # true
2 1 >    # true
1 1 <=   # true
1 1 >=   # true

1 2 + 3 =        # true
0.1 0.2 + 0.3 =  # false!
```

```
"hello world" "hello world" =    # true
"hello" "world" !=   # true
"Alice" "Bob" <      # true
```

Input and Output Operators

- `read-int` read an integer value from the keyboard and put it on the stack
- `read-flt` read a floating-point value from the keyboard and put it on the stack
- `100 print` pop an object from the stack and show it in the output area
- `1.2 println` pop an object from the stack and show it in the output area followed by a line break (ln = line)

Naming (Assignment)

! is the assignment operator

- ! operator assigns an object to a name
- Enters name-value pair in the dictionary

```
# define variables
3 x!    # give the name x to the value 3
4 y!    # give the name y to the value 4
# use variables
x       # lookup value in dict, push onto stack
y       # lookup value in dict, push onto stack
+       # add the two topmost objects on the stack
x y *   # 3 4 * => 12
```

Sequence of Instructions: Swapping the Contents of Two Variables

```
 ↓
3 x!
2 y!
 ↓
x h!
y x!
h y!
 ↓
x println
y println
```

Paper and pencil test:

x	y	h
~~3~~	~~2~~	
2	3	3

```
# without helper
# variable, just
# using the stack
x y x! y!
x println
y println
```

if: Conditional Execution (Selection)

- The execution of program parts may depend on a condition
  - Syntax: test {then-part} [{else-part}] if
    - The else-part is optional (may be omitted)
  - Semantics: If test is true then execute then-part, otherwise execute else-part
- Example: If user enters a value above 10
  then output “large”, otherwise output “small”.

```
read-int z!
z 10 > {"large"} {"small"} if
```

condition {then part}   {else part}

{...} is a block, which may or may not be executed

Check condition. If true execute {then part}, otherwise execute {else part}.

if: Conditional Execution (Selection)

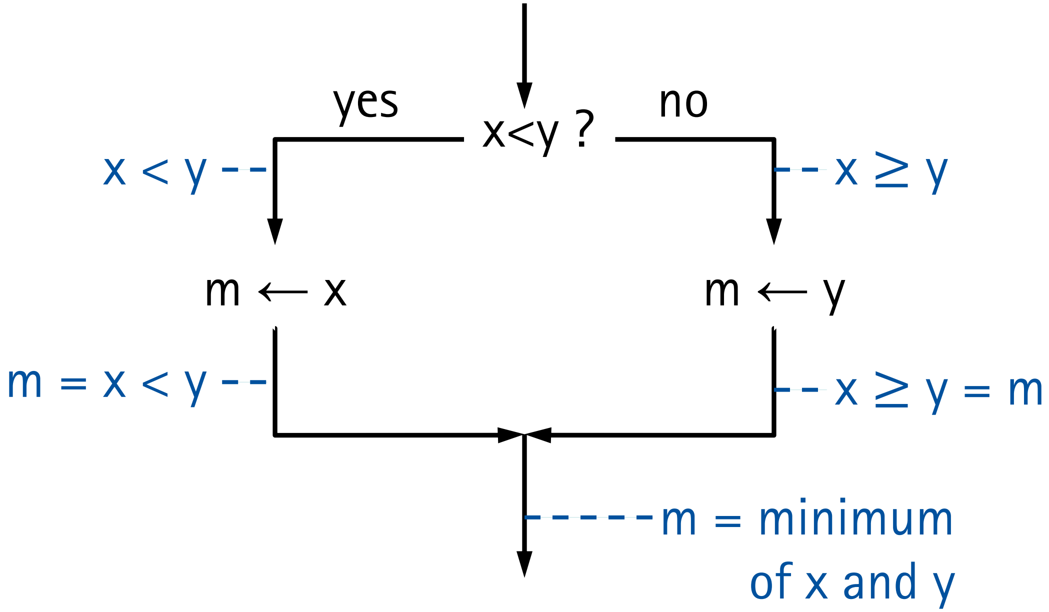
```
read-int z!
z 10 > {
  "large"
} {
  "small"
} if
```

```
read-int z!
z 100 > {
  "very large indeed"
} if
```

Conditional Execution: Minimum of Two Numbers

Finding the minimum of two numbers:

> "If x < y then assign the value of x to m,
> else assign the value of y to m."



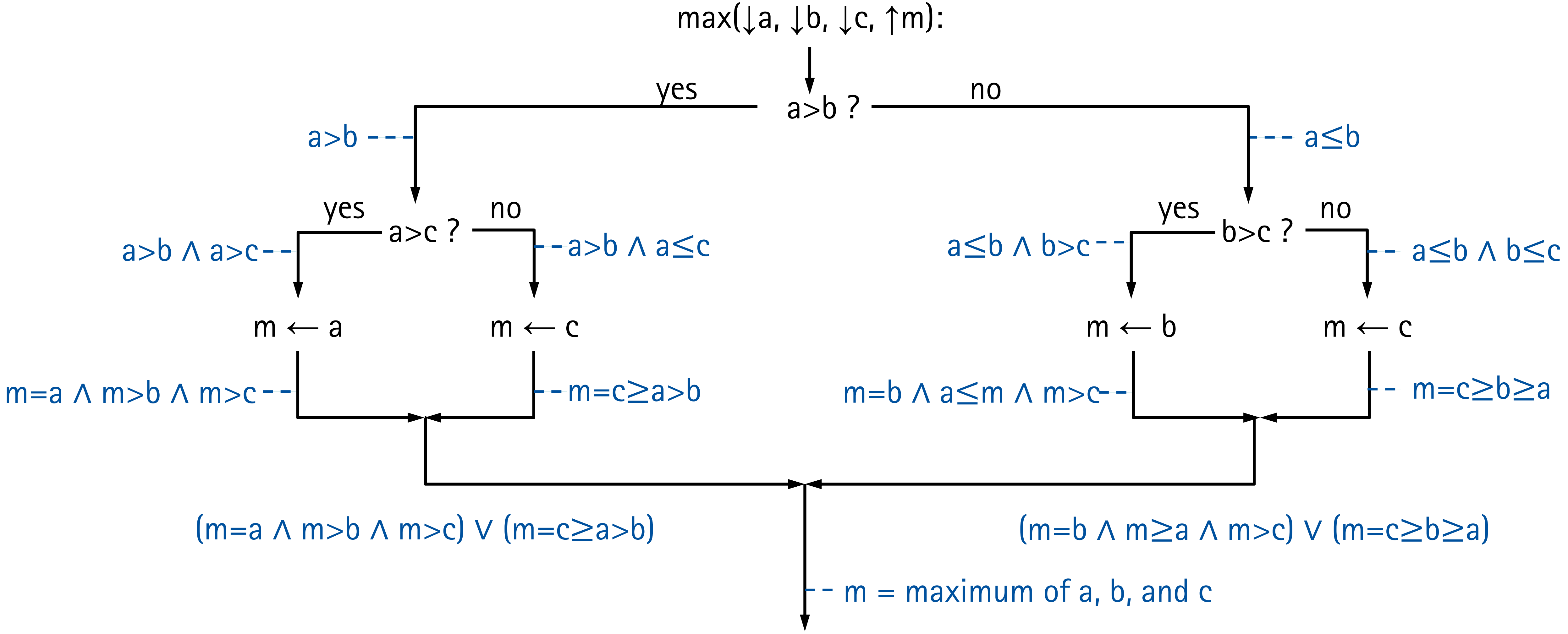

```
read-int x!
read-int y!
x y < { x m! } { y m! } if
m println
```

Conditional Execution: Maximum of Three Numbers

- Finding the maximum of three numbers (a, b, c), but only have pairwise comparison (if)

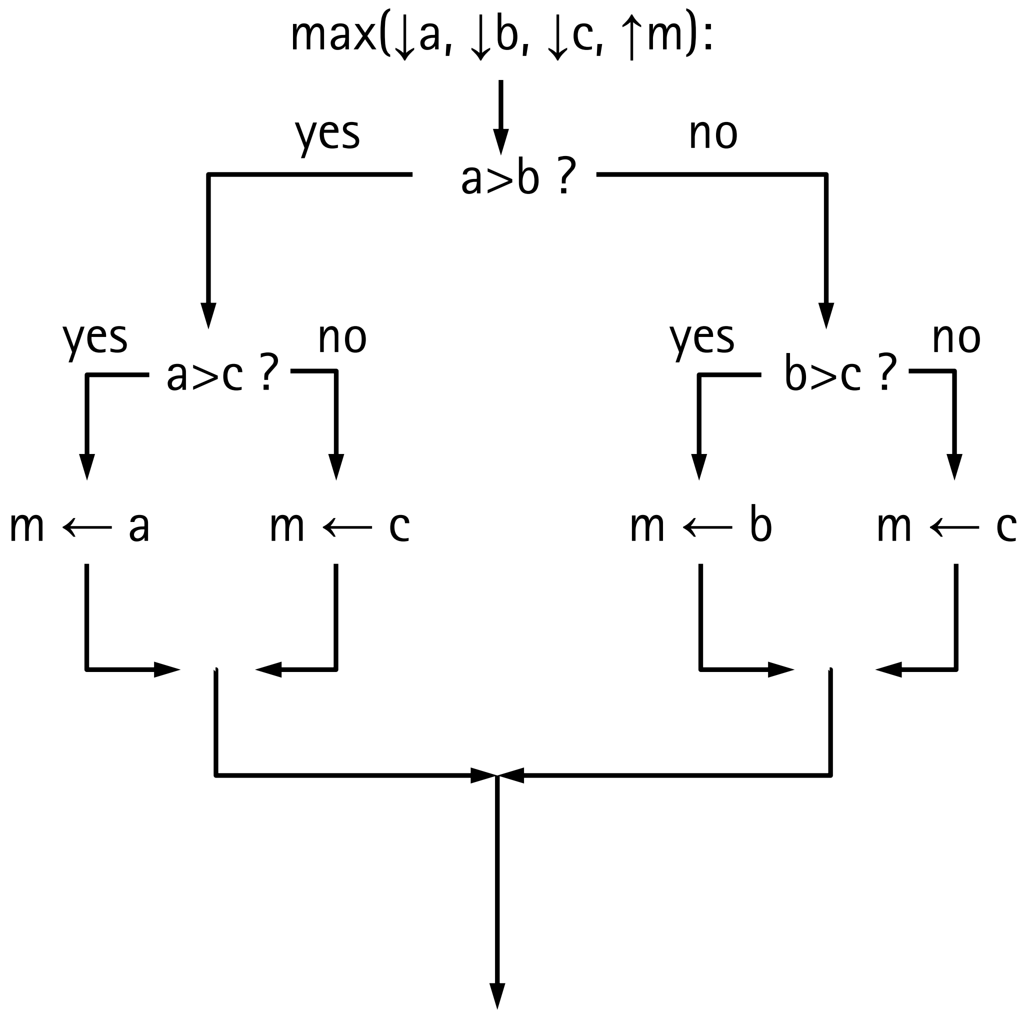
- Idea: Two steps of pairwise comparisons
  - Step 1: Test if $a > b$
    - if so, then b cannot be the maximum, do step 2a
    - otherwise $a \leq b$, do step 2b
  - Step 2a: Decide between a and c
  - Step 2b: Decide between b and c

Conditional Execution: Maximum of Three Numbers



Conditional Execution: Maximum of Three Numbers

```
"a: " println    read-int a!
"b: " println    read-int b!
"c: " println    read-int c!
a b > {
    a c > { a m! } { c m! } if
} {
    b c > { b m! } { c m! } if
} if
"The maximum is: " print
m println
```



Conditional Execution: Maximum of Three Numbers

- Finding the maximum of three numbers (a, b, c), but only have pairwise comparison (if)

- Idea 2: use conjunction of tests (logical and)
  - Test if $a \geq b \land a \geq c$. If so, then a is the maximum.
  - Otherwise, test if $b \geq a \land b \geq c$. If so, then b is the maximum.
  - If neither a nor b is the maximum, then c must be the maximum.

```
a b >=  a c >=  and {
    a m!
} {
    b a >=  b c >=  and {
        b m!
    } {
        c m!
    } if
} if
```

Conditional Execution: Maximum of Three Numbers

- Finding the maximum of three numbers (a, b, c), but only have pairwise comparison (if)

- Idea 1: Two steps of pairwise comparisons
- Idea 2: Conjunction of tests (logical and)

- Often multiple valid solutions to a programming problem
- How to be sure that a program is correct?

m = max(a, b, c) correct?

```
a b >  a c >  and {
    a m!
} {
    b a >  b c >  and {
        b m!
    } {
        c m!
    } if
} if

m = max(a, b, c)?
```

Counter example:
if a = 2, b = 2, c = 1,
then m = 1 ↯

m = max(a, b, c)   correct?

```
a b >  a c >  and {
    a m!
} {
    b a >  b c >  and {
        b m!
    } {
      😀  c m!
    } if
} if
```

😀

$\Leftrightarrow \neg(a>b \land a>c) \land \neg(b>a \land b>c)$

$\Leftrightarrow (a\leq b \lor a\leq c) \land (b\leq a \lor b\leq c)$ De Morgan

$\Leftrightarrow ab\land ba \lor ab\land bc \lor ac\land ba \lor ac\land bc$ Distributivgesetz

$\Leftrightarrow a=b \lor abc \lor bac \lor ac\land bc$

m = max(a, b, c)   correct?

```
a b >=  a c >=  and {
    a m!
} {
    b a >=  b c >=  and {
        b m!
    } {
😀      c m!
    } if
} if
```

😀

$\Leftrightarrow \neg(a \geq b \wedge a \geq c) \wedge \neg(b \geq a \wedge b \geq c)$

$\Leftrightarrow (a<b \vee a<c) \wedge (b<a \vee b<c)$ De Morgan

$\Leftrightarrow (ab \vee ac) \wedge (ba \vee bc)$ xy meint x<y

$\Leftrightarrow ab \wedge ba \vee ab \wedge bc \vee ac \wedge ba \vee ac \wedge bc$ Distributivgesetz

$\Leftrightarrow \text{false} \vee a<b<c \vee b<a<c \vee a<c \wedge b<c$

loop: Repeated Execution (Iteration)

- Repeatedly execute a piece of code until `break` or `breakif` is executed
  - `{...} loop`
- Example: Output the multiples of 5

```
5 x!
1 i!
{ i 10 > { break } if # exit from the loop if
                      # condition is true
    i x * println     # output i * x
    i 1 + i!          # increase i by 1
} loop
```

Iteration: Repetition of a block of code, here implemented with a loop instruction.

Nachbardiskussion (2 Minuten): Was ist hier falsch?

```
5 n!
1 i!
{ i n > breakif
    i print          i 1 + i!
} loop
```

We have seen: Program Elements

- Arithmetic operators `i 100 +`
- Logical operators `valid positive and`
- Relational and equality operators `a b >`
- Input and output operators `read-int, 1.2 println`

- Assignment (of a value to a name) `5 x!`
- Selection (conditional execution, if) `z 10 > {"large"} {"small"} if`
- Iteration (repeated execution, loop) `{... break ...} loop`
- Sequence (block of instructions) `{ m 2 * 1 + x! }`

Algorithm

- Finite, stepwise, precise procedure for solving a problem
- Analogy
  - Recipe
- Example

  name | ↓ parameter in | ↑ parameter out, result

  add_up_numbers_from_1_to_n (↓n, ↑s):

  1. $s \leftarrow 0$
  2. $i \leftarrow 1$
  3. repeat, while $i \leq n$
     - 3.1 $s \leftarrow s + i$
     - 3.2 $i \leftarrow i + 1$

  sequence of steps

  repetition, loop

- Program: Description of an algorithm in a programming language

Mössenböck: Sprechen Sie Java? dpunkt.verlag, 2011

Algorithm

- Parts of an algorithm
  - Name
  - Input parameters and output parameters (results)
  - Definition of steps

- Example above
  - Name: add_up_numbers_from_1_to_n
  - Input parameters: ↓n
  - Output parameters (results): ↑s
  - Definition of steps: 1., 2., 3., 3.1, 3.2, ...

Algorithm vs. Program

Algorithm (Pseudo Code):

add_up_numbers_from_1_to_n (↓n, ↑s):

1. $s \leftarrow 0$
2. $i \leftarrow 1$
3. repeat, while $i \leq n$
   - 3.1 $s \leftarrow s + i$
   - 3.2 $i \leftarrow i + 1$

Program (PostFix Syntax):

```
read-int n!
0 s!
1 i!
{ i n > breakif
    s i + s!
    i 1 + i!
} loop
s
```

Syntax vs. Semantics

- Syntax
  - Rules that determine how to create sentences
  - Example: Assignment is written as: name "!"

"how you write something"

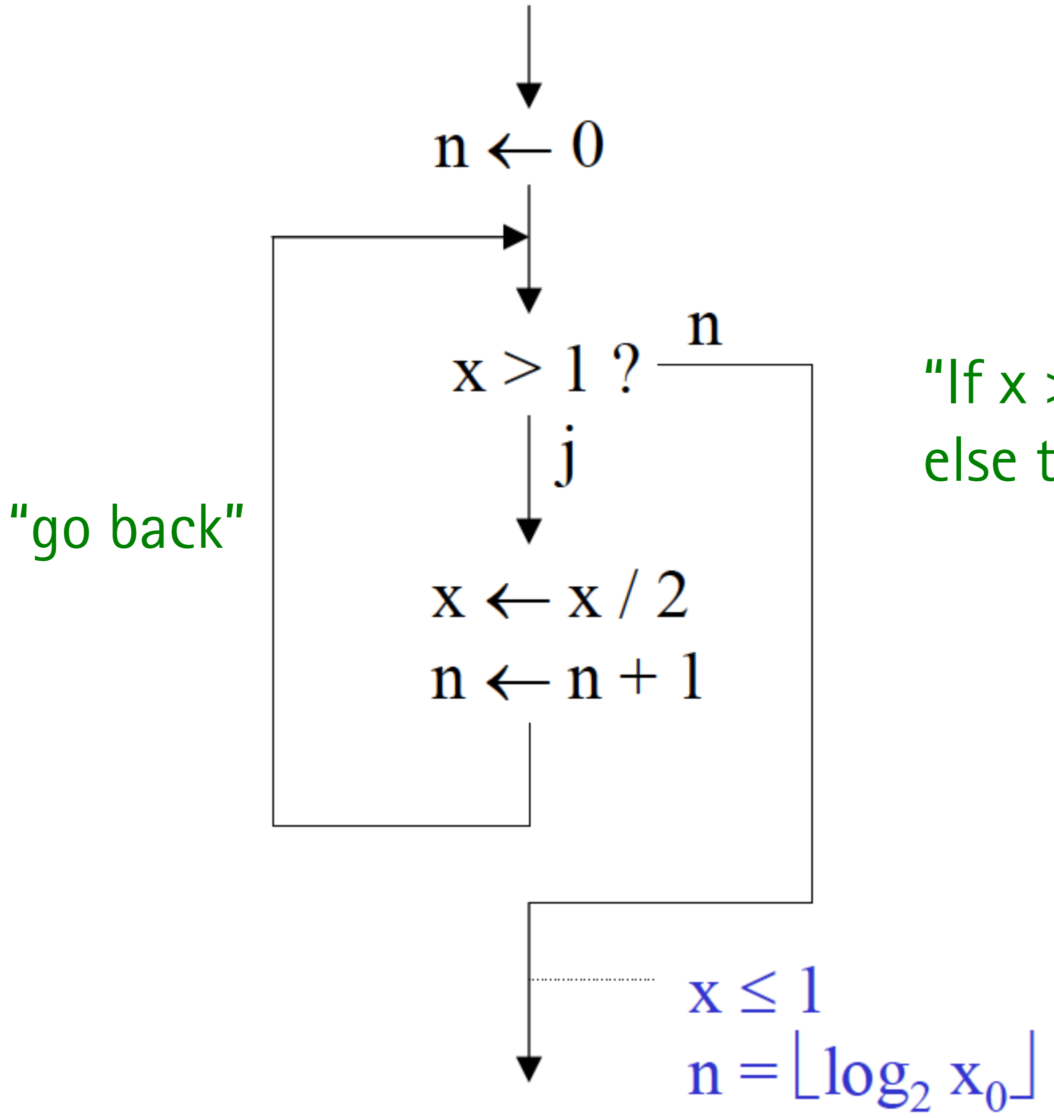
- Semantics
  - The meaning of a sentence
  - Example: Take topmost value from stack and store it in dictionary under name

"what that something means"

Syntax vs. Semantics

Syntax	Semantics
`read-int n!`	read integer number, put on stack, store in dictionary as `n`
`0 s!`	put integer number `0` on stack, pop from stack, store as `s`
`1 i!`	put integer number `1` on stack, pop from stack, store as `i`
`{ i n > breakif`	exit loop if value stored as `i` is larger than value stored as `n`
`    s i + s!`	look up values of s and `i`, add, store result as `s`
`    i 1 + i!`	look up value of `i`, add `1`, update `i`
`} loop`	
`s`	return the value stored in s as the final result

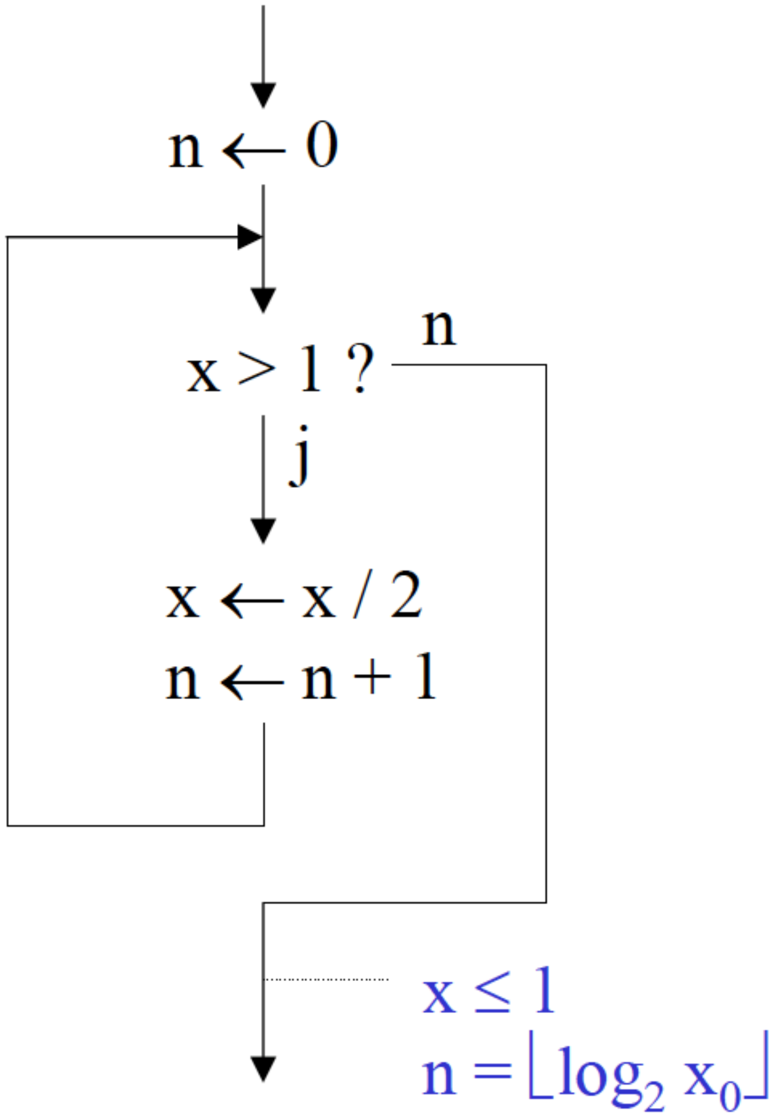
Iteration Example: Computing $\lfloor \log_2(x) \rfloor$



"If x > 1 then take the "j" branch, else take the "n" branch."

Mössenböck: Sprechen Sie Java? dpunkt.verlag, 2011

Iteration Example: Computing $\lfloor \log_2(x) \rfloor$

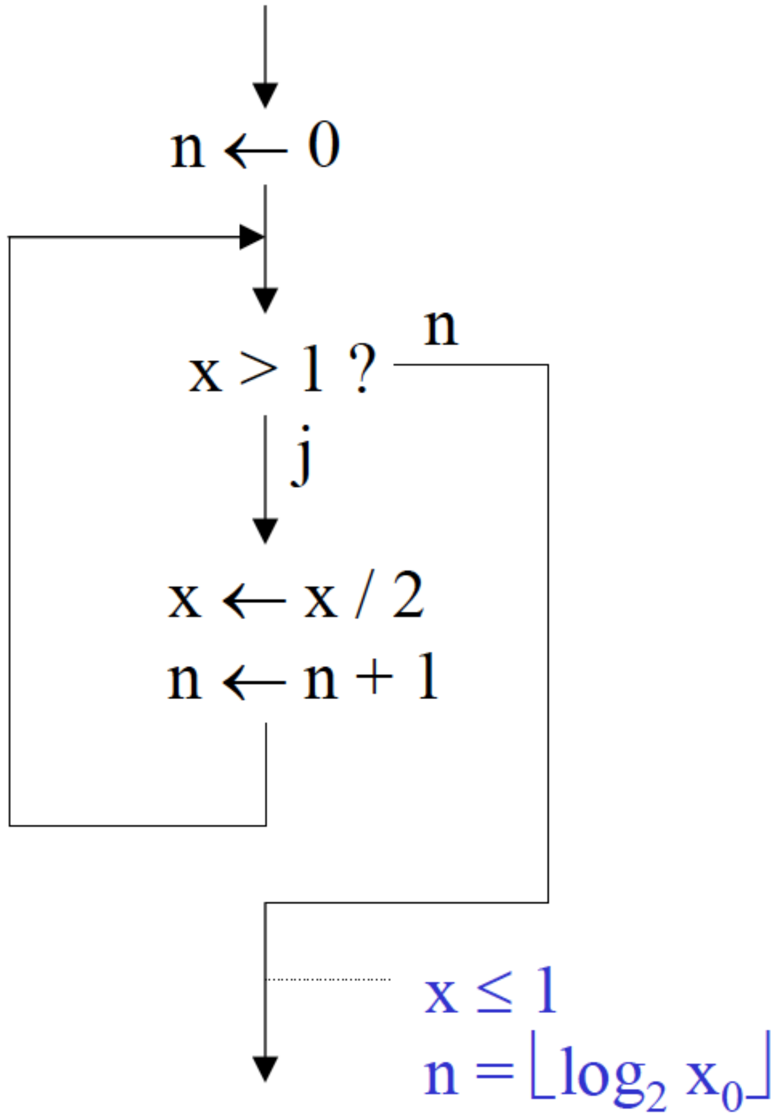


Paper and pencil test:

x	n
~~4~~	~~0~~
~~2~~	~~1~~
1	2

Mössenböck: Sprechen Sie Java? dpunkt.verlag, 2011

Iteration Example: Computing $\lfloor \log_2(x) \rfloor$

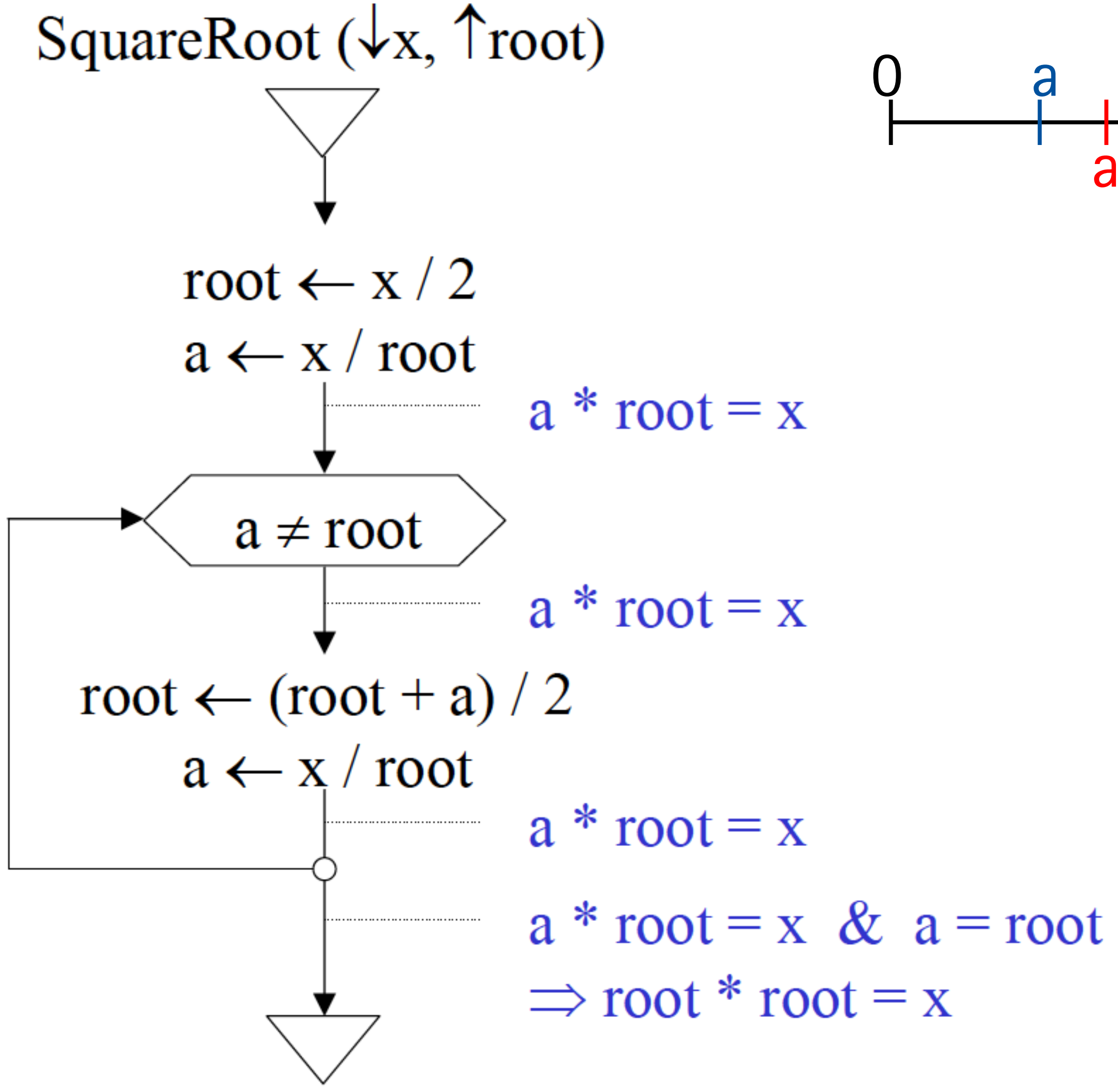



```
read-int x!

0 n!
{ x 1 <= breakif
    x 2 div x!
    n 1 + n!
} loop
n println
```

Mössenböck: Sprechen Sie Java? dpunkt.verlag, 2011

Iteration Example: Computing $\sqrt{x}$



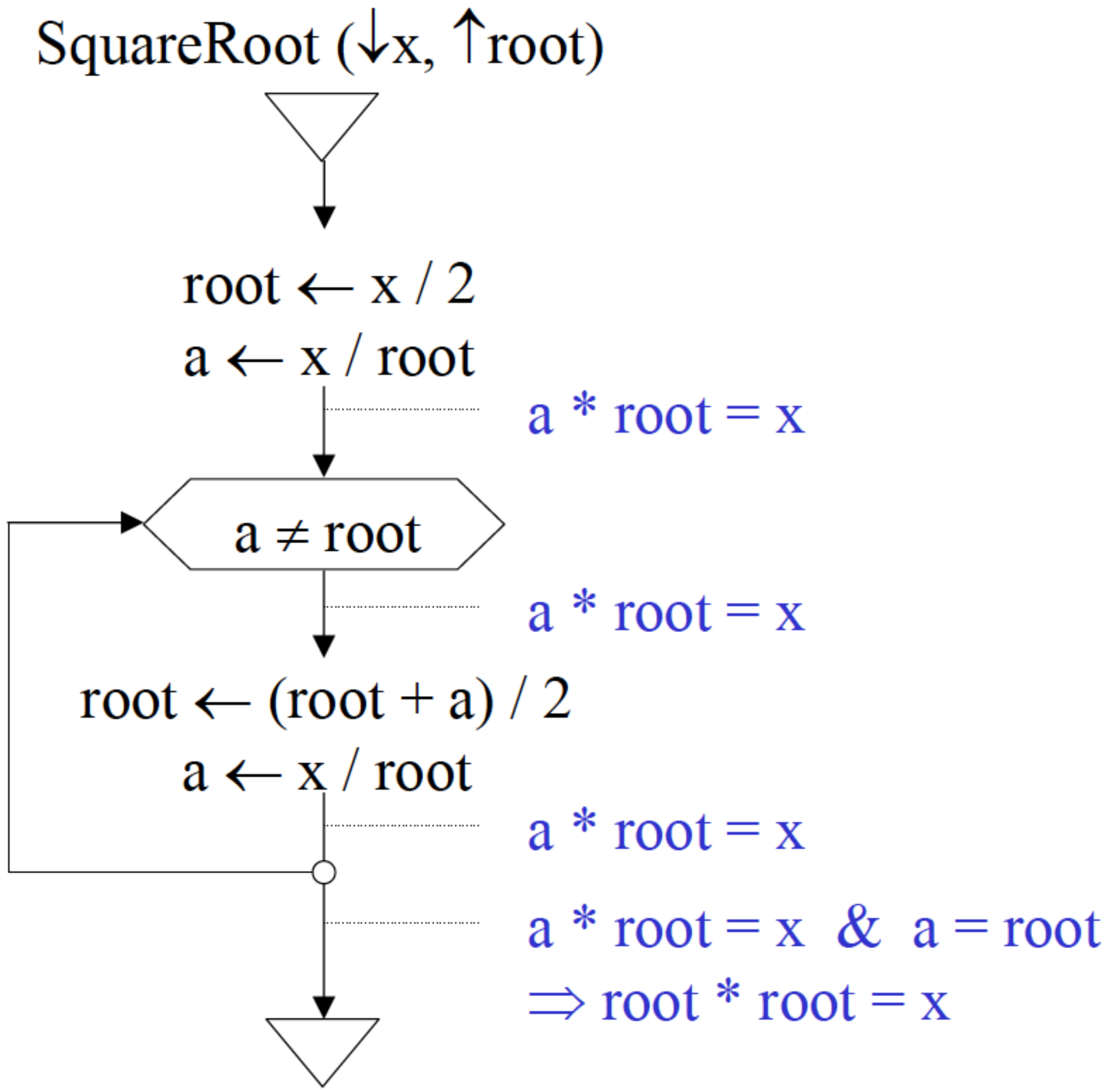
Paper and pencil test:

x	root	a
10	5	2
	3.5	2.857
	3.179	3.146
	3.162	3.162

Instead of a≠root use
|a-root|>0.001

Mössenböck: Sprechen Sie Java? dpunkt.verlag, 2011

Iteration Example: Computing $\sqrt{x}$




```
read-flt x!
x 2 / root!
x root / a!
{ a root - abs 0.001 < breakif
    root a + 2 / root!
    x root / a!
} loop
root println
```

Mössenböck: Sprechen Sie Java? dpunkt.verlag, 2011

Summary

- Organization
  - Vorlesung + Hörsaalübung (Prüfungsleistung, Klausur)
  - Hausübung (Studienleistung, Übungsabgaben)
  - Tutorien (Abnahme der Abgaben, Besprechung der Aufgaben)
  - Kleingruppenübung (Präsenzform, optional)
- Programs = data (operands) and instructions (operators)
- PostFix execution model and environment
- PostFix operators: arithmetic, logical, relational/equality
- Assignment (of a value to a name)
- Selection (conditional execution, if)
- Iteration (repeated execution, loop)
- Algorithms

Leisure Reading

Peter Norvig: Teach Yourself Programming in Ten Years

http://norvig.com/21-days.html