Leibniz Universität Hannover

Programmieren 1

Atomic Data and Functions

Human-Computer Interaction Group

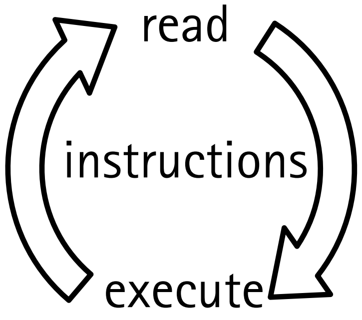
Prof. Dr. Michael Rohs
michael.rohs@hci.uni-hannover.de

Lectures

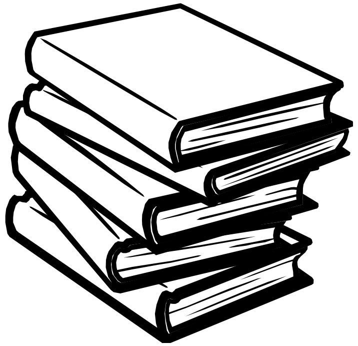
#	Date	Topic	HÜ→	HÜ←
1	14.10.	Organization, computers, programming, algorithms, PostFix introduction (execution model, IDE, basic operators, booleans, naming)	1	20.10. 23:59
2	21.10.	PostFix (primitive types, functions, parameters, local variables, tests), recipe for atomic data	2	27.10. 23:59
3	28.10.	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)	

Review

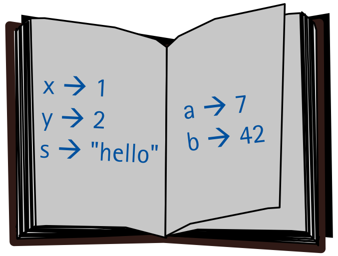
- Programs = data (operands) and instructions (operators)
- Operators: arithmetic, logical, relational/equality
- PostFix execution model and environment



- Assignment (of a value to a name)
- Selection (conditional execution, if)
- Iteration (repeated execution, loop)
- Algorithms



stack of operands



named values

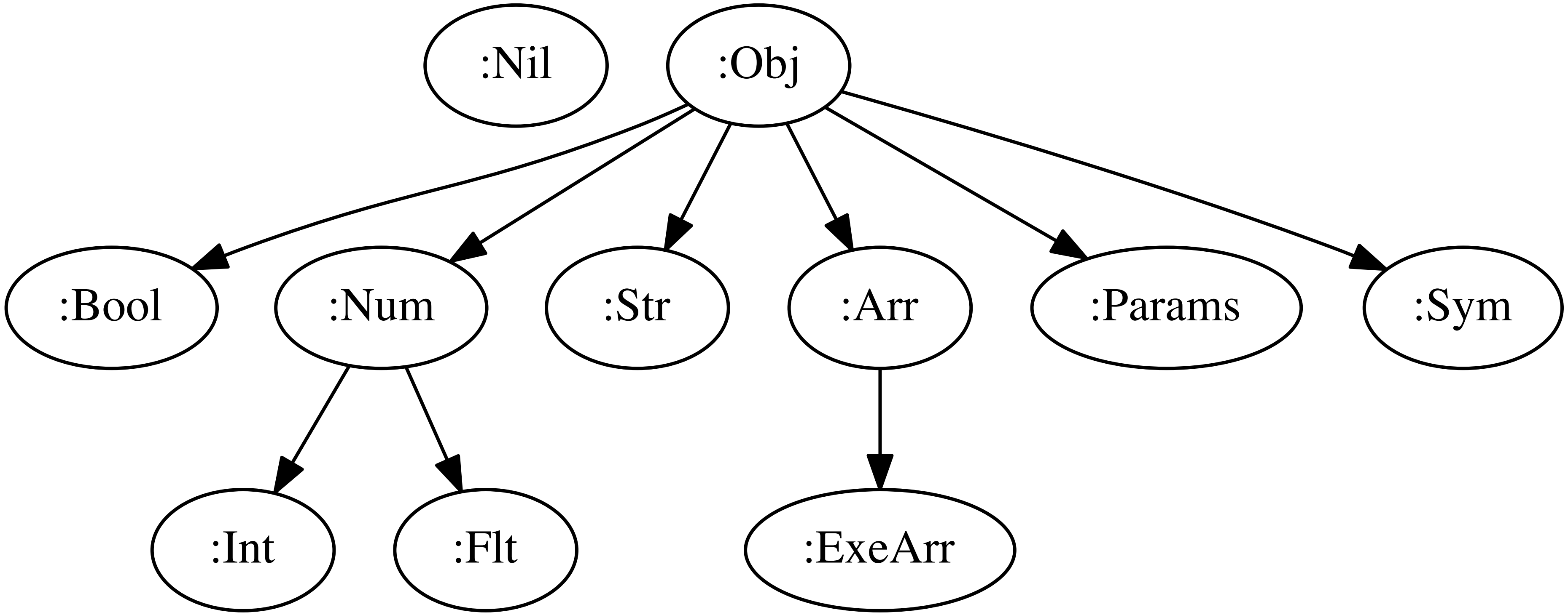
Preview

- PostFix
  - Types, data arrays, executable arrays
- Functions
  - Functions are named, reusable pieces of computation
  - Functions have parameters
  - Functions have local variables
- How to Design Programs
  - From problem statement to well-organized solution
- Simple Atomic (Nondivisible) Data

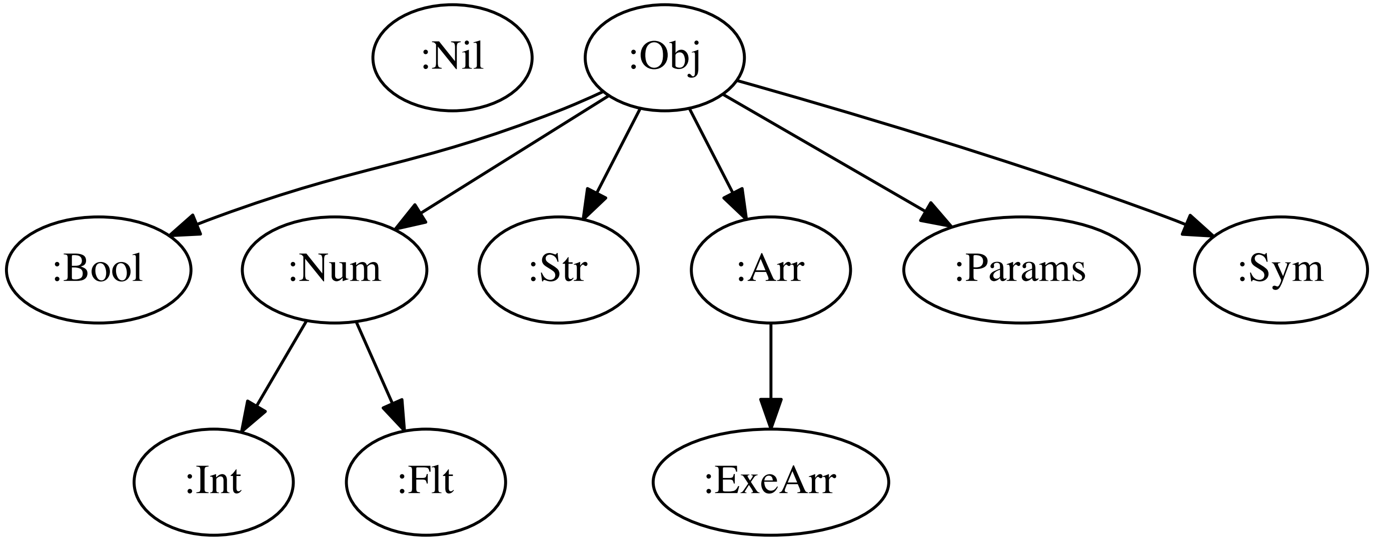
POSTFIX

Types

Data in PostFix is categorized into different types



Types



:Nil	value nil, “nothing”
:Bool	values true and false
:Int	signed integer values `123`
:Flt	floating-point values, `0.123`
:Num	numbers, either :Int or :Flt
:Str	strings, i.e., sequences of characters, enclosed in "…"
:Arr	arrays of arbitrary objects, written as […]
:ExeArr	executable arrays of arbitrary objects, written as {…}
:Params	parameter lists, written as (…)
:Sym	symbols (names), written with a preceding or trailing colon, i.e., :x and x: denote the same symbol

Arrays

- A collection of any number of elements
  - Array elements may be of different types
  - Array elements are accessed by their index (zero-based)
- Data arrays
  - `[ ]`
  - `[1 "two" false 3.14]`
  - `[1 [2 3]]`
- Executable arrays
  - `{ }`
  - `{1 "two" false 3.14}`
  - `{1 {2 3}}`
- Array operators
  - `[10 20 30] length` → 3
  - `[10 20 30] 0 get` → 10
  - `[10 20 30] 1 get` → 20
  - `[10 20 30] .1` → 20 # abbreviation for `[…] 1 get`

Data Arrays vs. Executable Arrays

- Parsing data arrays [...]: Elements executed immediately
- Parsing executable arrays {...}: Execution of elements is deferred (done later)
  - When executable array is being parsed, interpreter is in "deferred mode"
- Example program source code: `[1 2 +]` `{1 2 +}`

next token	stack
[	ArrOpen
1	ArrOpen Int(1)
2	ArrOpen Int(1) Int(2)
+	ArrOpen Int(3)
]	Arr(Int(3))

next token	stack
{	ExeArrOpen
1	ExeArrOpen Int(1)
2	ExeArrOpen Int(1) Int(2)
+	ExeArrOpen Int(1) Int(2) Plus
}	ExeArr(Int(1) Int(2) Plus)

- Program result: `[3]` `{1 2 +}`

a data array with one element — an executable array with three elements

Executable Arrays

- Explicit execution of an executable array
  ```
  {1 2 +} exec            # result: 3
  ```
- Conditional execution of an executable array
  ```
  true {1 2 +} if         # result: 3
  ```
- Executable arrays are executed when referenced in dictionary
  ```
  {1 2 +} f!              # store f → {…} in dictionary
  f                       # reference and execute, result: 3
  ```
- Referencing the array without executing
  ```
  :f vref                 # {1 2 +}    (vref = value reference)
  exec                    # explicit execution, result: 3
  ```

Referenced Values are Automatically Executed

- Referencing: Looking up in dictionary and executing

```
f          # name without colon
```

- Value-referencing: Referencing without executing, pushing

```
:f         # with colon, just push name on stack
vref       # lookup value, but don't execute
exec       # explicitly execute
```

Two Ways of Naming Values

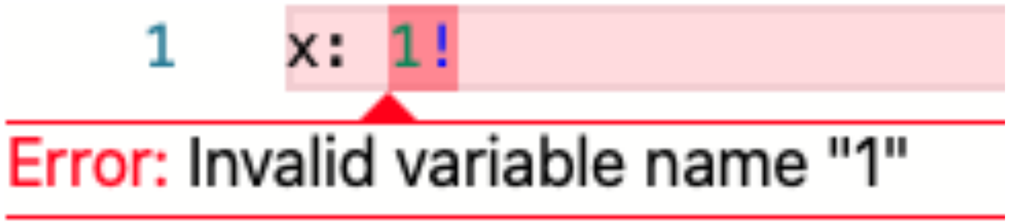
```
value name!
3 x!
4 y!
```

is equivalent to: `name: value !`

```
x: 3 !
y: 4 !
```

note the space

missing space



Choosing Names for Variables and Constants

- Short but meaningful names
  - Examples: sum, greeting
- Very short names for short-lived variables
  - Only used in small scope, e.g., for counting, indexing
  - Examples: i, j, x
- Longer name for long-lived variables
  - More descriptive
  - Need to be understood in larger scope
  - Examples: input-text, input_text, inputText
- Don't make the type name part of the variable name

FROM EXECUTABLE ARRAYS TO FUNCTIONS

From Executable Arrays to Functions

- We may give a name to a piece of code:

```
>> { 1 + } f!   # Executable array { 1 + } is given the name f.
                # The stack is now empty.
>> 10 f         # The executable array is referenced...
11              # processes 10 and produces 11
>> f            # The executable array is referenced again...
12              # processes 11 and produces 12
```

From Executable Arrays to Functions

Program: `{ 1 + } f! 10 f f`

Tokens	Stack	Dictionary	Comment
`{ 1 + }`	`{ 1 + }`		Push executable array on the stack
`f!`		`f → { 1 + }`	Enter executable array into dictionary
`10`	`10`	`f → { 1 + }`	Push 10
`f`	`10`	`f → { 1 + }`	Lookup f, execute array
`1`	`10 1`	`f → { 1 + }`	Push 1
`+`	`11`	`f → { 1 + }`	Execute +
`f`	`11`	`f → { 1 + }`	Lookup f, execute array
`1`	`11 1`	`f → { 1 + }`	Push 1
`+`	`12`	`f → { 1 + }`	Execute +

From Executable Arrays to Functions

- Executable array to compute $f(x) = x^2$:

```
>> { dup * } f!   # Associate name f with this executable array.
                  # The stack is now empty.
>> 3 f            # Duplicate the topmost value on the stack
                  # and multiply it with itself.
9                 # The result.
```

From Executable Arrays to Functions

Program: `{ dup * } f! 3 f`

Tokens	Stack	Dictionary	Comment
`{ dup * }`	`{ dup * }`		Push executable array on stack
`f!`		`f → { dup * }`	Enter in dictionary
`3`	`3`	`f → { dup * }`	Push 3
`f`	`3`	`f → { dup * }`	Lookup f, execute array
`dup`	`3 3`	`f → { dup * }`	Duplicate top stack element
`*`	`9`	`f → { dup * }`	Execute *

Defining Variables in an Executable Array

Program: `{ x! x x * } f! 3 f`

Tokens	Stack	Dictionary	Comment
`{ x! x x * }`			Push exec. array on the stack
`f!`		`f → { x! x x * }`	Enter exec. array into dict.
`3`	`3`	`f → { x! x x * }`	Push 3
`f`	`3`	`f → { x! x x * }`	Lookup f, execute array
`x!`		`f → { x! x x * }, x → 3`	Store argument in dict.
`x`	`3`	`f → { x! x x * }, x → 3`	Lookup x
`x`	`3 3`	`f → { x! x x * }, x → 3`	Lookup x, again
`*`	`9`	`f → { x! x x * }, x → 3`	Execute *

Defining Variables in an Executable Array

- With just one dictionary:

```
100 x!                  # set x to 100
{ x! x x * } f!         # store executable array in dictionary
3 f                     # executing f changes x to 3
x println               # output: 3, not 100
```

Defining Variables in an Array with a Local Dictionary

- Executable array with its own dictionary

```
100 x!                      # set x to 100
{ x! x x * } lam f!         # add new dict. to {…}, then store {…} in outer dict.
3 f                         # x in {…} is different from outer x
x println                   # output: 100
```

note the lam

Defining Variables in an Array without a Local Dictionary

Program: `100 x! { x! x x * } f! 3 f`

Tokens	Stack	Dictionary	Comment
`100`	`100`		Push 100
`x!`		`x → 100`	Store 100 in dictionary
`{…}`	`{…}`	`x → 100`	Push exec. array on the stack
`f!`		`x → 100, f → {…}`	Store exec. array in dictionary
`3`	`3`	`x → 100, f → {…}`	Push 3
`f`	`3`	`x → 100, f → {…}`	Lookup f, execute array
`x!`		`x → 3, f → {…}`	Store argument in dict.
…	…	…	…

Defining Variables in an Array with a Local Dictionary

Program: `100 x! { x! x x * } lam f! 3 f`

Tokens	Stack	Dictionary	Comment
`100`	`100`		Push 100
`x!`		`x → 100`	Store 100 in dictionary
`{…}`	`{…}`	`x → 100`	Push exec. array on the stack
`lam`	`{…} lam`	`x → 100`	Add copy of dict. to exec. array
`f!`		`x → 100, f → {…} lam`	Store exec. array in dictionary
`3`	`3`	`x → 100, f → {…} lam`	Push 3
`f`	`3`	`x → 100`	Lookup f, exec. array, switch dicts.
…	…	…	…

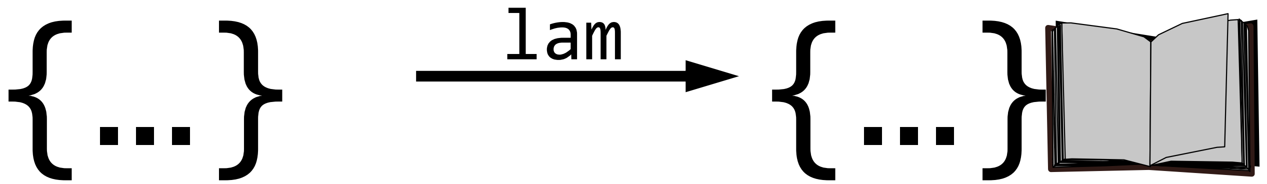
Defining Variables in an Array with a Local Dictionary

Program: `100 x! { x! x x * } lam f! 3 f`

Tokens	Stack	Dictionary	Comment
…	…	…	…
f	3	x → 100	Lookup f, execute array, switch dicts.
x!		x → 3	Store argument in current dict.
x x	3 3	x → 3	Reference (execute) x twice
*	9	x → 3	Perform multiplication
	9	x → 100, f → {…} lam	Return from f, the outer dictionary is restored as the current dictionary

Local Variables

- Executable arrays may have their own local dictionary
- Variables set in such arrays do not affect variables outside
- `lam` operator:
    - Copies the current dictionary ("snapshot" of the current environment)
    - Associates the copy with the executable array
    - Changes to the local dictionary are not reflected in the original dictionary

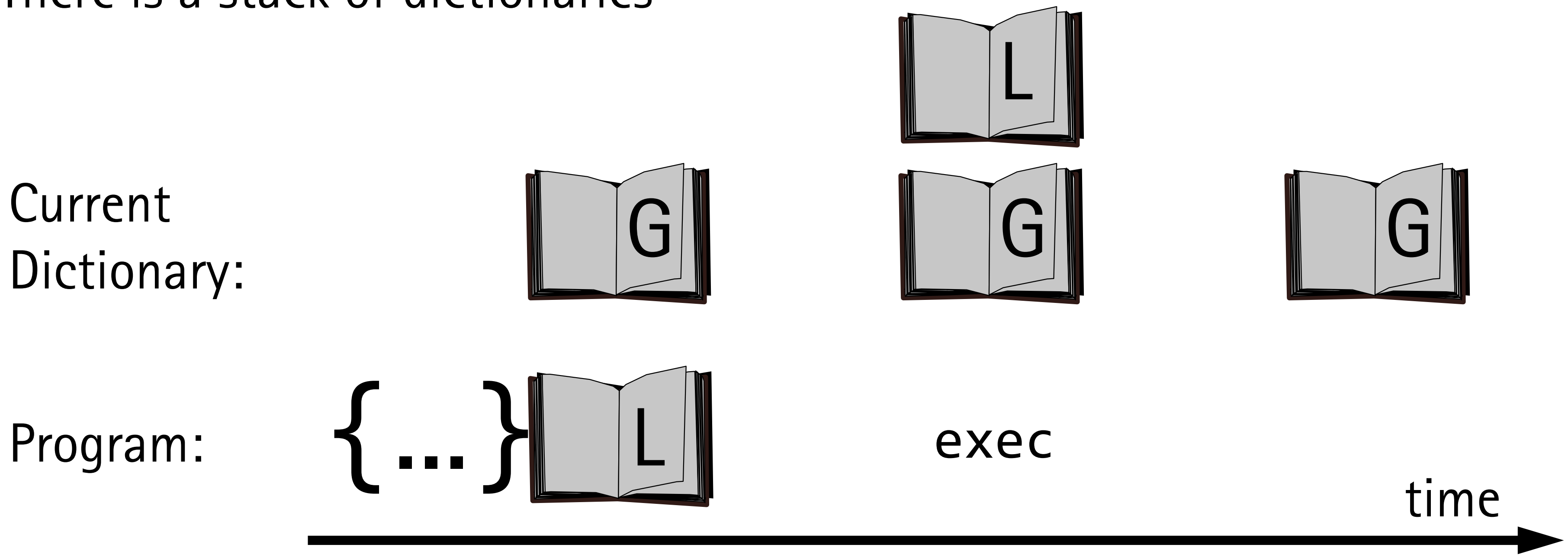


executable array with associated dictionary

Local Variables

- Executing an executable array with a local dictionary
  - Make the local dictionary the current dictionary
  - Execute the contents of the array
  - Restore the previous dictionary

⇒ There is a stack of dictionaries



Defining Functions in PostFix

- Functions in PostFix are executable arrays with a local dictionary
- How to define functions:

```
1. {…} lam f!
2. f: {…} lam !   # equivalent to 1.
3. f: {…} fun     # equivalent to 2. (except puts f in f's dictionary)
```

`fun` stands for "function"
(not for fun, obviously)

Visibility of Variables

- `lam` and `fun` create independent copies of the current dictionary
- So, everything that is in the current dictionary when lam is executed is also present (visible) in the copied dictionary

Visibility of Variables

Program: 7 `x! f: { y! x y * } fun 3 f`

Tokens	Stack	Dictionary	Comment
7 x!		`x → 7`	
`f: {…}`	`f: {…}`	`x → 7`	
`fun`		`x → 7, f → {…} lam`	Copy current dict., associate copy with {…}, enter `f → {…} lam`
3	3	`x → 7, f → {…} lam`	
`f`	3	`x → 7`	Reference (execute) f: f's local dict. becomes the current dictionary
`y!`		`x → 7, y → 3`	Store y ⟶ 3 in the current dict.
`x y *`	21	`x → 7, f → {…} lam`	Return from f, restore the outer dict.

Functions

- Function: A reusable piece of code with a well-defined interface
  - Don't repeat yourself (DRY) principle
- An important form of abstraction
  - Functions encapsulate some computation
- Functions provide a well-defined interface
  - Describe what is done, ignore how
- Functions have an implementation
  - Defines how to compute the result from the parameters
- Not identical to functions in mathematics
  - Function may produce output or read input in addition to returning a result (side effects)
  - Some languages use the term “procedure”

Function: Difference between Two Numbers

```
minus: { x! y!   # defining function minus
    x y -
} fun

1 2 minus        # calling function minus with arguments 1 and 2
```

Result: 1

Why?

Function: Difference between Two Numbers

Program: `diff: { x! y! x y - } fun    1 2 diff`

Tokens	Stack	Dictionary	Comment
`1`	`1`	`diff → {…} lam`	Push 1
`2`	`1 2`	`diff → {…} lam`	Push 2
`diff`	`1 2`		Reference (execute) diff: diff's local dict. becomes the current dict.
`x!`	`1`	`x → 2`	Store y → 2 in current dict.
`y!`		`x → 2, y → 1`	Store y → 1 in current dict.
`x y`	`2 1`	`x → 2, y → 1`	Reference (execute) x and y
`-`	`1`	`x → 2, y → 1`	Compute the difference
	`1`	`diff → {…} lam`	Return from diff, restore the outer dictionary

Parameter Lists

- Reverse variable assignments:

```
minus: { y! x!
      x y -
} fun               1 2 minus # returns -1
```

- or use parameter lists:

- Parameter lists specify the parameters of a function

```
minus: (x y) {
      x y -
} fun              1 2 minus # returns -1
```

Parameter Lists with Types

- Parameter lists with types specify the inputs (arguments) and outputs (return values) of a function:

```
minus: (x :Num, y :Num -> :Num) {
    x y -
} fun
```

- Read parameter list as:
  "The function takes two numbers as input and returns a single number. Within the function, the first input is named x and the second input is named y."

A Function on Strings

- Program:

```
emphasize: (s :Str -> :Str) {
    s "!" +
} fun
"hello" emphasize
```

- Result:

```
"hello!"
```

HOW TO DESIGN PROGRAMS

Steps in Programming

From a problem statement to a well-organized solution

1. Problem statement
2. Data definition
3. Function name and parameter list
4. Function stub and purpose statement
5. Examples with expected results
6. Implementation
7. Test and revision

based on: http://www.ccs.neu.edu/home/matthias/HtDP2e/Draft/part_one.html

1. Problem statement (given)

"Design a function that converts degrees Celsius to degrees Fahrenheit."

The problem statement should answer these questions:

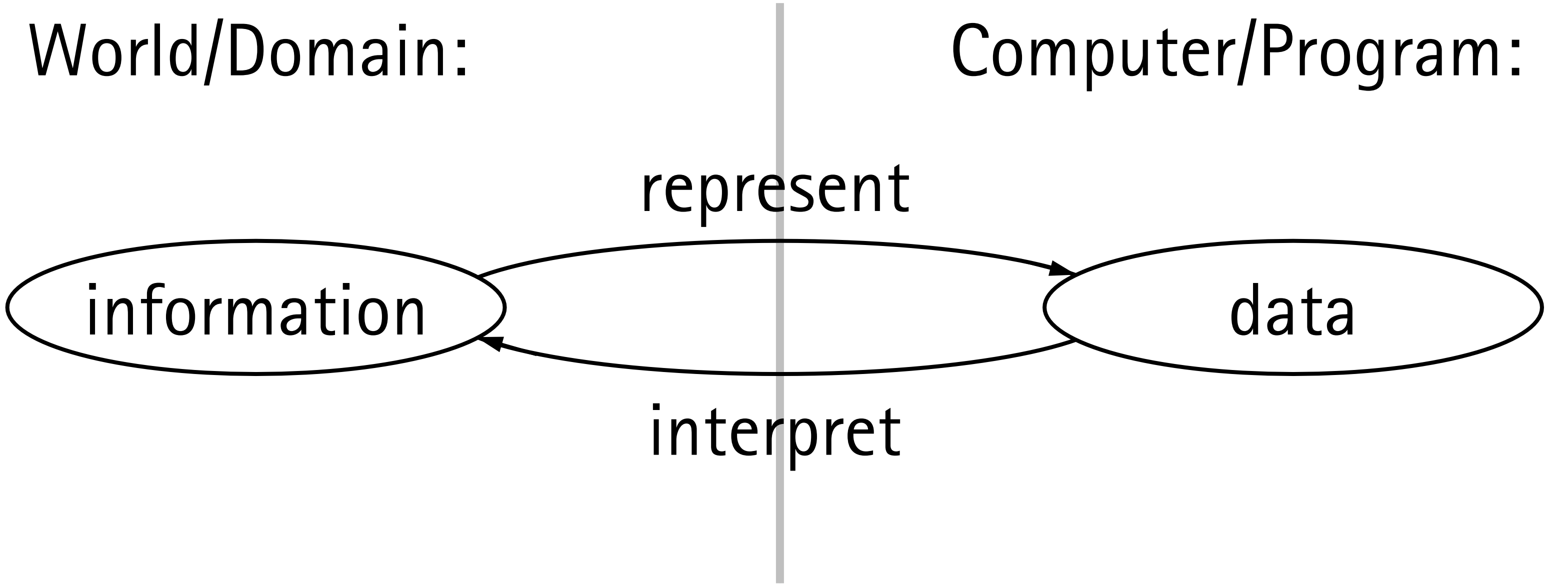
- What information needs to be represented?
- What should the program do with the data?
- What cases need to be considered?

2. Write a data definition

Express how you wish to represent information as data.

“temperature is a number”

“interpretation of the number is degrees Celsius”



based on: http://www.ccs.neu.edu/home/matthias/HtDP2e/Draft/part_one.html

3. Write a function name and parameter list

```
celsius-to-fahrenheit: (celsius :Int -> :Int)
```

function name

parameter name and type (input)

result type (output)

Function consumes arguments and produces results

degrees Celsius
type integer

function
celsius-to-fahrenheit

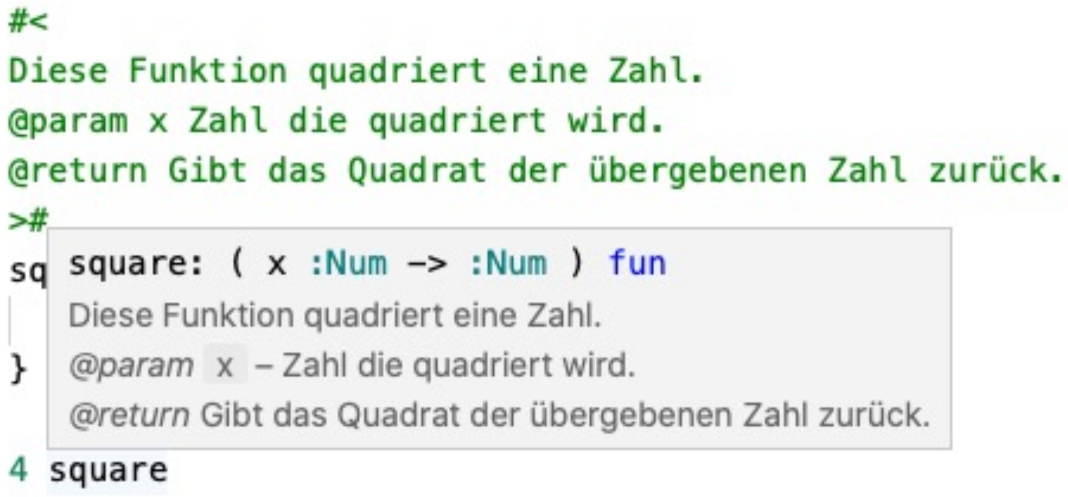
degrees Fahrenheit
type integer

4. Write a function stub and purpose statement (documentation)

```
# Takes a temperature value in degrees Celsius and
# returns the corresponding value in degrees Fahrenheit.
celsius-to-fahrenheit: (celsius :Int -> :Int) {
    0
} fun
```

stub returns arbitrary integer value

- The function stub returns an arbitrary value from the function's domain.
- Runs, but does not provide the correct result yet.



5. Give examples with expected results

- Formula: °F = °C * 9 / 5 + 32

- Given: 0, expect: 32
- Given: 10, expect: 50
- Given: -5, expect: 23
- Given: 100, expect: 212

6. Implement the function body

```
celsius-to-fahrenheit: (celsius :Int -> :Int) {
    celsius 9 * 5 div 32 +
} fun
```

div is the integer division operator

The implementation defines how to compute the result from the parameters.

Implementing: Comments First

- When implementing it is often helpful to start with comments that describe the steps of algorithm and then implementing them.
- Example:

```
// Determine number of characters of result
// Allocate required memory
// Run through input and copy required characters
// Terminate the result string
```

- This already shows you the skeleton of your solution.

7a. Test the function

Apply the function to the test examples and check whether it yields the expected results:

```
  0 celsius-to-fahrenheit # expect:  32
 10 celsius-to-fahrenheit # expect:  50
 -5 celsius-to-fahrenheit # expect:  23
100 celsius-to-fahrenheit # expect: 212
```

7b. Review and revise the function

Review and revise the function name, the parameter names, and the purpose statement. Improve them if necessary. The purpose statement should describe what the function computes (not how) and should mention the given inputs and produced result.

```
# Takes a temperature value in degrees Celsius
# and returns the corresponding value in
# degrees Fahrenheit.
celsius-to-fahrenheit: (celsius :Int -> :Int) {
    celsius 9 * 5 div 32 +
} fun
```

Complete Program (in celsius-to-fahrenheit.pf)

```
# Takes a temperature value in degrees Celsius
# and returns the corresponding value in
# degrees Fahrenheit.
celsius-to-fahrenheit: (celsius :Int -> :Int) {
    celsius 9 * 5 div 32 +
} fun

  0 celsius-to-fahrenheit # expect:  32
 10 celsius-to-fahrenheit # expect:  50
 -5 celsius-to-fahrenheit # expect:  23
100 celsius-to-fahrenheit # expect: 212
```

Test Cases

- A test case compares the actual result and expected result
  - `actual expected test=`
- Example
  - If function celsius-to-fahrenheit is given 0, we expect 32 as the result
  - Test case: `0 celsius-to-fahrenheit  32  test=`
- If actual is equal to expected result, then the output is:
  Check passed.
- Otherwise the output is:
  Actual value 99 differs from expected value 32.

Complete Program with Test Cases

```
# Takes a temperature value in degrees Celsius
# and returns the corresponding value in
# degrees Fahrenheit.
celsius-to-fahrenheit: (celsius :Int -> :Int) {
    celsius 9 * 5 div 32 +
} fun

  0 celsius-to-fahrenheit  32  test=
 10 celsius-to-fahrenheit  50  test=
 -5 celsius-to-fahrenheit  23  test=
100 celsius-to-fahrenheit 212  test=
test-stats   # print test statistics
```

Complete Program with Test Cases

```
  0 celsius-to-fahrenheit  32  test=✓
 10 celsius-to-fahrenheit  50  test=✓
 -5 celsius-to-fahrenheit  23  test=✓
100 celsius-to-fahrenheit 212  test=✓
test-stats   # print test statistics
```

```
✓ All 4 tests passed
```

RECIPE FOR ATOMIC DATA

Design Recipes

- Recipe for Atomic Data
- Recipe for Enumerations
- Recipe for Intervals
- Recipe for Itemizations
- Recipe for Compound Data (Product Types)
- Recipe for Variant Data (Sum Types)

Atomic Data

- Smallest conceptual unit of data
- Cannot be broken down further
- Example data types
  - integer numbers: -123 (:Int)
  - floating-point numbers: 3.14 (:Flt)
  - Boolean values: true, false (:Bool)
  - Strings: "hello" (:Str)

Strings are treated as atomic here, even though strings consist of characters

Steps in Programming

From a problem statement to a well-organized solution

1. Problem statement
2. Data definition
3. Function name and parameter list
4. Function stub and purpose statement
5. Examples with expected results
6. Implementation
7. Test and revision

based on: http://www.ccs.neu.edu/home/matthias/HtDP2e/Draft/part_one.html

1. Problem Statement

- Write down the problem statement as a comment.
  - What information needs to be represented?
  - What should the function (to be implemented) do with the data?
  - What cases need to be considered?
- Example

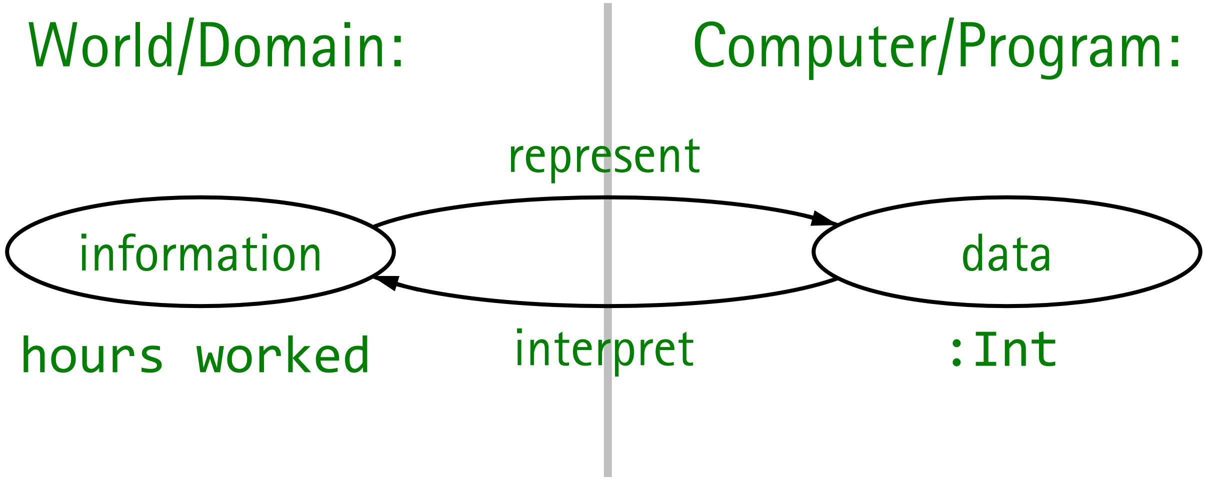
```
#<
Design a function that computes weekly wages with overtime from
hours worked. The hourly rate is 10 €/hour. Regular working time is
40 hours/week. Overtime is paid 150% of the normal rate of pay.
>#
```

2. Data Definition

- How should domain information be represented as data in the program?
- How to interpret the data as real-world information?

- Informal data definition (comment)

```
# :Int represents hours worked
# :Int represents wage in cents
```



3. Function Name and Parameter List

- Find a good function name
  - Short, non-abbreviated, descriptive name that describes what the function does
- Find good parameter names
  - Short, non-abbreviated, descriptive name that describes what the parameter means
- Write parameter list
  - Parameter names and types left of the arrow
  - Result type right of the arrow
- Example

```
hours-to-wages: (hours :Int -> :Int)
```

4a. Function Stub

- Function stub returns an arbitrary value from the function's range
- The function stub compiles

- Example

```
hours-to-wages: (hours :Int -> :Int) {
    0
} fun
```

4b. Purpose Statement

- Briefly describes what the function does. Ideally as a single sentence. Multiple sentences may be necessary.

- Example

```
# Compute the wage in cents given the number of hours worked.
```

5. Examples with Expected Results

- Examples (wage is 10 € per hour, 15 € for overtime)
  - For 0 hours worked, expect 0 cents.
  - For 20 hours worked, expect 20 * 1000 cents.
  - For 39 hours worked, expect 39 * 1000 cents.
  - For 40 hours worked, expect 40 * 1000 cents.
  - For 41 hours worked, expect 40 * 1000 + 1 * 1500 cents.
  - For 45 hours worked, expect 40 * 1000 + 5 * 1500 cents.

5. Examples with Expected Results (Test Function)

- Comparison of actual and expected result
- Examples (wage is 10 € per hour, 15 € for overtime)

```
hours-to-wages-test: {
    0 hours-to-wages  0  test=
    20 hours-to-wages  20 1000 *  test=
    39 hours-to-wages  39 1000 *  test=
    40 hours-to-wages  40 1000 *  test=
    41 hours-to-wages  40 1000 * 1 1500 * +  test=
    45 hours-to-wages  40 1000 * 5 1500 * +  test=
    test-stats
} fun
```

6. Function Body

- Implementation of the function

- Example

```
hours-to-wages: (hours :Int -> :Int) {
    hours 40 <= {
        hours 1000 *
    } {
        40 1000 *  hours 40 - 1500 *  +
    } if
} fun
```

7a. Testing

- Call test function

  ```
  hours-to-wages-test
  ```

- Test results

  wages.pf, line 27: Check passed.
  wages.pf, line 28: Check passed.
  wages.pf, line 29: Check passed.
  wages.pf, line 30: Check passed.
  wages.pf, line 31: Check passed.
  wages.pf, line 32: Check passed.
  All 6 tests passed!

7b. Review and Revise

- Review the products of the steps
  - Improve function name
  - Improve parameter names
  - Improve purpose statement
  - Improve and extend tests
- Generalize the function
  - Constants
  - Templates for future functions on this data

Constants

- Don't hardcode literal values, don't repeat them
  - Don't repeat yourself (DRY) principle
- Instead: Define constants
- Constants give a name to a value
- Example

```
40 WEEKLY_HOURS! # regular work hours per week
1000 HOURLY_RATE_REGULAR! # in cents
1500 HOURLY_RATE_OVERTIME! # in cents
```

Generalizing the Function with Constants

```
hours-to-wages: (hours :Int -> :Int) {
  hours 40 <= {
    hours 1000 *
  } {
    40 1000 *
    hours 40 - 1500 * +
  } if
} fun
```

Generalizing the Function with Constants

```
40 WEEKLY_HOURS! # regular work hours per week
1000 HOURLY_RATE_REGULAR! # in cents
1500 HOURLY_RATE_OVERTIME! # in cents

hours-to-wages: (hours :Int -> :Int) {
  hours WEEKLY_HOURS <= {
    hours HOURLY_RATE_REGULAR *
  } {
    WEEKLY_HOURS HOURLY_RATE_REGULAR *
    hours WEEKLY_HOURS - HOURLY_RATE_OVERTIME * +
  } if
} fun
```

Generalizing the Function with Additional Parameters

```
hours-to-wages: (
  weekly-hours :Int,
  hourly-rate-regular :Int,
  hourly-rate-overtime :Int,
  hours-worked :Int
  -> :Int)
{
  hours-worked weekly-hours <= {
    hours-worked hourly-rate-regular *
  } {
    weekly-hours hourly-rate-regular *
    hours-worked weekly-hours - hourly-rate-overtime * +
  } if
} fun
```

Literal values, constants, or parameters?

- Is it a good idea to
  - generalize a function with constants?
  - generalize a function with additional parameters?

- It depends...!
  - Functions should be flexible pieces of computation
  - Functions should be easy to use
  - More parameters → more flexible, but more effort to use
  - Compromise: Constants for rarely changing data, parameters for frequently changing data

Use Constants and Parameters

- Constants for rarely changing data
- Parameters for frequently changing data
- Example (assuming WEEKLY_HOURS and OVERTIME_FACTOR change infrequently):

```
hours-to-wages: (hours :Int, rate :Int -> :Int) {
  hours WEEKLY_HOURS <= {
    hours rate *
  } {
    WEEKLY_HOURS rate *
    hours WEEKLY_HOURS - rate OVERTIME_FACTOR * *
    + round int
  } if
} fun
```

Helper Functions on a Wish List

- Designing functions requires thought
- A different concern should be outsourced in a helper function
  - A function should perform one well-defined task
- A reusable subtask should be outsourced in a helper function
  - Don't Repeat Yourself (DRY)
- When implementing a function you may find parts that (a) are different concerns or (b) are reusable subtasks
  - Put required helper functions on a wish list
  - Implement a stub for the helper functions

Summary

- PostFix
  - Types, data arrays, executable arrays
- Functions
  - Functions are named, reusable pieces of computation
  - Functions have parameters
  - Functions have local variables
- How to Design Programs
  - From problem statement to well-organized solution
- Simple Atomic (Nondivisible) Data