Programmieren 1

Enumerations and Intervals

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 and intervals	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

- PostFix
  - Types, data arrays `[…]`, executable arrays `{…}`, functions `f: (…) {…} fun`, parameters, local variables, tests
- Functions
  - Functions are named, reusable pieces of computation
  - Some parts are fixed, others (parameters) are filled with actual values (arguments) when calling the function
- How to Design Programs
  - From problem statement to well-organized solution
- Recipe for Atomic Data
  - Primitive data that cannot be subdivided further

Preview

- Operators
- Recipe for enumerations
- Recipe for intervals
- Array operations
- Characters and Strings
- Loops

OPERATORS

Operators in PostFix

	Operators
arithmetic	`+` `-` `*` `/` `div` (integer division) `mod`
relational, equality	`<` `<=` `>` `>=` `=` (equal) `!=` (not equal)
logical	`and` `or` `not`
conditional	`if` `cond`
assignment	`x!` `!` `vref`
stack	`pop` `dup` `swap` `copy` `clear`
loops	`loop` `for` `fori` `break` `breakif`
arrays	`[` `]` `{` `}` `array` `length` `exec`
array access	`set` `get` `.3` `.:x` `.i`
function definition	`lam` `fun` `cond-fun`
parameters	`(` `)` `->`
data definition	`datadef`
type test	`int?` `flt?` `num?` `bool?` `str?` `sym?` `...`
type cast	`int` `flt` `str` `sym` `arr` `exearr`
tests	`test=` `test~=` `test!=` `test!~=` `test-stats`

Short-Circuit Logical Operators

- `and` and `or` either take two Boolean operands or one array
  - Array of predicates: May not need to evaluate all of them
  - `{p₁ p₂ … pₙ} and`
  - `{p₁ p₂ … pₙ} or`
- **Short-circuit and:** Test elements one-by-one, stop on first false and return false, if no false found return true

```
{{1 positive} {0 positive} {2 positive}} and
```

not evaluated

- **Short-circuit or:** Test elements one-by-one, stop on first true and return true, if no true found return false

```
{{0 positive} {1 positive} {2 positive} } or
```

not evaluated

Two Boolean operands:

```
p₁ p₂ and
p₁ p₂ or
```

cond: Conditional Operator

- Multiple cases are inconvenient with nested ifs
- cond operator processes conditions one-by-one
- executes first action for which condition is true
- Example:

```
10 age!
{
    { age 3 <= } { "Toddler" }      # condition 1, action 1
    { age 13 <= } { "Child" }       # condition 2, action 2
    { age 18 < } { "Teenager" }     # condition 3, action 3
    { true } { "Adult" }            # default case, action 4
} cond
```

cond-fun: Conditional Function Operator

Function with cond as its body may be abbreviated to cond-fun

```
f: (age :Num -> :Str) {
  {
    { age 3 <= } { "Toddler" }
    { age 13 <= } { "Child" }
    { age 18 < } { "Teenager" }
    { true } { "Adult" }
  } cond
} fun
```

equivalent to

```
f: (age :Num -> :Str) {
  { age 3 <= } { "Toddler" }
  { age 13 <= } { "Child" }
  { age 18 < } { "Teenager" }
  { true } { "Adult" }
} cond-fun
```

Stack Operators

- Stack operations allow writing functions without variables
  - But can be difficult to read

- clear: Remove all elements from stack
- pop: Remove top stack element
- dup: Duplicate top stack element
- swap: Exchange top two stack elements

Stack Operators

```
dist: { dup * swap dup * + sqrt } ! 3 4 dist
```

Tokens	Stack	Dictionary	Comment
…! 3 4	3 4	dist → {…}	Store array in dictionary, push 3 and 4
dist	3 4	dist → {…}	Reference (execute) dist
dup	3 4 4	dist → {…}	Duplicate top stack element
*	3 16	dist → {…}	Multiply two top stack elements
swap	16 3	dist → {…}	Swap two top stack elements
dup	16 3 3	dist → {…}	Duplicate top stack element
*	16 9	dist → {…}	Multiply two top stack elements
+	25	dist → {…}	Add two top stack elements
sqrt	5.0	dist → {…}	Take square root of top stack element

Type Test Operators

- type: Provide the type of an object (:Obj -> :Sym)
  - `123 type → :Int`

- Test whether an object as a specific type (:Obj -> :Bool)
- int? flt? num? bool? str? sym? arr? exearr?
  - `123 int? → true`
  - `123.456 num? → true`
  - `"hello" bool? → false`

Type Cast Operators

- Convert the type of an object (return nil if not possible)
- int, flt, str, sym, arr, exearr
  - `67.89 int → 67`
  - `67.89 str → "67.89"`
  - `"67.89" flt → 67.89`
  - `"67.89" int → 67`
  - `"abc" int → nil`
  - `"hello" arr → ["hello"]`
  - `[1 2 3] str → "[ 1 2 3 ]"`

Convert to string before printing:

```
["hello"] str println
```

Output:

```
[ "hello" ]
```

- round: Round floating-point to integer number
  - 67.89 round → 68 (type :Int)

RECIPE FOR ENUMERATIONS

Design Recipes

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

Enumerations

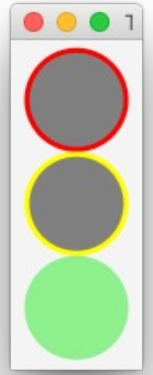
- An enumeration type can represent one of a fixed number of distinct values
- Each enumerated value names a category
- Examples
  - Dog breeds (Poodle, Golden Retriever, Saint Bernard)
  - Suit of cards (diamonds ♦, clubs ♣, hearts ♥, and spades ♠)
  - Continents (Europe, Asia, Africa, etc.)
- In PostFix, such types can be represented as symbols
  - Symbols are names

1. Problem Statement

- Write down the problem statement as a comment.
  - What is the relevant information?
  - What should the function do with the data?

- Example

```
#<
Design a function that returns the duration of a traffic light phase.
Durations:   red: 3 s, red-yellow: 1 s, green: 2 s, yellow: 1 s.
>#
```



2. Data Definition

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

- Write the data definition as a list of required symbols

- Data definition

```
# enumeration of TrafficLight states:
# :red, :red-yellow, :green, :yellow

# :Int represents the phase duration in milliseconds
```

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 each parameter means
- Write parameter list
  - Parameter names and types left of the arrow
  - Result type right of the arrow
- Example

```
traffic-light-duration: (color :Sym -> :Int)
```

4a. Function Stub

- Function stub returns an arbitrary value from the function's range
- The function stub is syntactically complete (can be executed)

- Example

```
traffic-light-duration: (color :Sym -> :Int) {
    0
} fun
```

4b. Purpose Statement

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

- Example

```
# Duration of the given traffic light phase in milliseconds.
```

5. Examples with Expected Results

- Examples
  - Duration of red is 3 s
  - Duration of red-yellow is 1 s
  - Duration of green is 2 s
  - Duration of yellow is 1 s
- Test function

```
traffic-light-duration-test: {
  :red traffic-light-duration 3000 test=
  :red-yellow traffic-light-duration 1000 test=
  :green traffic-light-duration 2000 test=
  :yellow traffic-light-duration 1000 test=
  test-stats
} fun
```

6. Function Body

```
traffic-light-duration: (color :Sym -> :Int) {
  color :red = {
    3000 # red
  } {
    color :red-yellow = {
        1000 # red-yellow
      } {
        color :green = {
        2000 # green
      } {
        1000 # yellow
      } if
    } if
  } if
} fun
```

7a. Testing

- Calling the test function

```
traffic-light-duration-test
```

- Test results

```
:red traffic-light-duration 3000 test= ✓
:red-yellow traffic-light-duration 1000 test= ✓
:green traffic-light-duration 2000 test= ✓
:yellow traffic-light-duration 1000 test= ✓
✓ All 4 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
- Improve / generalize the function
  - cond operator
  - cond-fun operator

cond Operator to Handle Multiple Cases

```
traffic-light-duration: (color :Sym -> :Int) {
  {
    { color :red = } { 3000 }
    { color :red-yellow = } { 1000 }
    { color :green = } { 2000 }
    { color :yellow = } { 1000 }
  } cond
} fun
```

```
traffic-light-duration: (color :Sym -> :Int) {
  color :red = {
    3000 # red
  } {
    color :red-yellow = {
        1000 # red-yellow
      } {
        color :green = {
        2000 # green
      } {
        1000 # yellow
      } if
    } if
  } if
} fun
```

cond-fun Operator to Handle Multiple Cases

```
traffic-light-duration: (color :Sym -> :Int) {
  { color :red = } { 3000 }
  { color :red-yellow = } { 1000 }
  { color :green = } { 2000 }
  { color :yellow = } { 1000 }
} cond-fun
```

cond-fun Operator to Handle Multiple Cases

```
traffic-light-duration: (color :Sym -> :Int) {
  { color :red = } { 3000 }
  { color :red-yellow = } { 1000 }
  { color :green = } { 2000 }
  { color :yellow = } { 1000 }
} cond-fun

:blue traffic-light-duration
```

cond-fun Operator to Handle Multiple Cases

```
traffic-light-duration: (color :Sym -> :Int) {
  { color :red = } { 3000 }
  { color :red-yellow = } { 1000 }
  { color :green = } { 2000 }
  { color :yellow = } { 1000 }
} cond-fun

:blue traffic-light-duration
```

```
255    traffic-light-duration: (color :Sym -> :Int) {
```

Error: Expected fun to return 1 values but it returned 0 values

cond-fun Operator to Handle Multiple Cases

```
traffic-light-duration: (color :Sym -> :Int) {
  { color :red = } { 3000 }
  { color :red-yellow = } { 1000 }
  { color :green = } { 2000 }
  { color :yellow = } { 1000 }
  { true } { color str " is not a valid color" + err }
} cond-fun

:blue traffic-light-duration
```

cond-fun Operator to Handle Multiple Cases

```
traffic-light-duration: (color :Sym -> :Int) {
  { color :red = } { 3000 }
  { color :red-yellow = } { 1000 }
  { color :green = } { 2000 }
  { color :yellow = } { 1000 }
  { true } { color str " is not a valid color" + err }
} cond-fun

:blue traffic-light-duration
```

```
 { true } { color str " is not a valid color" + err }
```

Error: :blue is not a valid color

cond-fun Operator to Handle Multiple Cases

```
#<
Duration of the given traffic light phase in milliseconds.
@param color traffic light phase, allowed values are
            :red, :red-yellow, :green, and :yellow
@return the duration of the phase in milliseconds
>#
traffic-light-duration: (color :Sym -> :Int) {
  { color :red = } { 3000 }
  { color :red-yellow = } { 1000 }
  { color :green = } { 2000 }
  { color :yellow = } { 1000 }
  { true } { color str " is not a valid color" + err }
} cond-fun
```

RECIPE FOR INTERVALS

Design Recipes

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

Intervals

- Intervals represent one or more ranges of numbers
- Example: Taxation scheme with
  - No tax for goods below 1000 €
  - Moderate tax for goods between 1 k€ - 10 k€
  - High tax for goods of 10 k€ or more
- Pay attention to boundary cases
  - Interval notation is precise: [0, 1000), [1000, 10000), [10000, inf)
  - Brackets [and] denote inclusive boundaries
  - Parentheses (and) denote exclusive boundaries
- Data definitions
  - Symbols to name each interval
  - Constants to define boundaries

1. Problem Statement

- Write down the problem statement as a comment.
  - What is the relevant information?
  - What should the function do with the data?

- Example

```
#<
A fictitious country has decided to introduce a three-stage sales
tax. Cheap items below 1 k€ are not taxed. Luxury goods of
10 k€ or more are taxed at 10%. Items in between are taxed at
a rate of 5%. Give a data definition and define a function that
computes the amount of tax for a given item price.
>#
```

2. Data Definition

- How should domain information be represented as data in the program? How to interpret the data as real-world information?
- Data definition
  - Name each interval as a symbol
  - Give interpretations of data types
  - Capture the interval boundaries as constants

```
# enumeration of tax stages:
# :no-tax :low-tax :high-tax

# :Int represents Euro

1000 LOW-TAX-BOUNDARY!  # interpretation: price in Euro
10000 HIGH-TAX-BOUNDARY! # interpretation: price in Euro
```

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 each parameter means
- Write parameter list
  - Parameter names and types left of the arrow
  - Result type right of the arrow
- Example

```
sales-tax: (price :Int -> :Int)
```

4a. Function Stub

- Function stub returns an arbitrary value from the function's range
- The function stub is syntactically complete (can be executed)

```
sales-tax: (price :Int -> :Int) {
    0
} fun
```

4b. Purpose Statement

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

- Example

```
# Returns the amount of tax for the given price.
```

5a. Examples with Expected Results

- For a price of 0 € expect a sales tax of 0 €.
- For a price of 537 € expect a sales tax of 0 €.
- For a price of 1000 € expect a sales tax of 50 € (5 %).
- For a price of 1282 € expect a sales tax of 64 € (5 %).
- For a price of 10000 € expect a sales tax of 1000 € (10 %).
- For a price of 12017 € expect a sales tax of 1202 € (10 %).

5b. Examples and Expected Results (Test Function)

```
sales-tax-test: {
    0 sales-tax, 0, test=
    537 sales-tax, 0, test=
    1000 sales-tax, 1000 0.05 * round, test=
    1282 sales-tax, 1282 0.05 * round, test=
    10000 sales-tax, 10000 0.10 * round, test=
    12017 sales-tax, 12017 0.10 * round, test=
    test-stats
} fun
sales-tax-test
```

6. Function Body

```
# Returns the amount of tax for the given price.
sales-tax: (price :Int -> :Int) { # :Int represents Euro
  { 0 price <=  price 1000 <  and } { # :no-tax interval
      0
  }
  { 1000 price <=  price 10000 <  and } { # :low-tax interval
      price 0.05 * round
  }
  { price 10000 >= } { # :high-tax interval
      price 0.10 * round
  }
  { true } { # error: if this line is reached then price < 0
      "sales-tax, error: negative price" err
  }
} cond-fun
```

7. Testing

- Main function call test function

  `sales-tax-test`

- Test results

  tax.pf, line 36: Check passed.
  tax.pf, line 37: Check passed.
  tax.pf, line 38: Check passed.
  tax.pf, line 39: Check passed.
  tax.pf, line 40: Check passed.
  tax.pf, line 41: Check passed.
  All 6 tests passed!

8. Review and Revise

- Review the products of the steps
  - Improve function name
  - Improve parameter names
  - Improve purpose statement
  - Improve and extend tests

- Improve / generalize the function
  - Constants
  - Simplify conditions (because, checked from top)
  - Templates for future functions on this data

Constants and Simplified Conditions

```
0.05 LOW-TAX-RATE!
0.10 HIGH-TAX-RATE!
sales-tax: (price :Int -> :Int) { # :Int represents Euro
    { price 0 < } { # error
        "sales-tax, error: negative price" err
    }
    { price LOW-TAX-BOUNDARY < } { # :no-tax interval
        0
    }
    { price HIGH-TAX-BOUNDARY < } { # :low-tax interval
        price LOW-TAX-RATE * round
    }
    { true } { # :high-tax interval
        price HIGH-TAX-RATE * round
    }
} cond-fun
```

ARRAY OPERATIONS

Arrays Operations

```
>> [1 2 3] 0 "x" set
[ "x" 2 3 ]
>> [1 2 3] 4 append
[ 1 2 3 4 ]
>> ["hop" "skip"] ["jump" "now"] +
[ "hop" "skip" "jump" "now" ]
>> ["hop" "skip" "jump"] "fall" contains
false
```

- Arrays cannot be modified (in PostFix)
  - cannot be modified once created → are immutable
- Array operations may create a new array

Strings and Arrays are Immutable

		stack contents:
`"hello" s!`		
`s 0 "H" set`	`# new array on stack`	`"Hello"`
`s`	`# old one still exists`	`"Hello" "hello"`
`pop`	`# remove top value`	`"Hello"`
`s!`	`# store new value`	
`s`	`# now s has new value`	`"Hello"`

Array Operations

Create an n-element array and initialize it with the given object or executable array, call executable array for each index

- `n {…} array`
- Examples:
- `>> 3 "hello" array`

  `["hello" "hello" "hello" ]`
- `>> 5 {} array`

  `[ 0 1 2 3 4 ]`
- `>> 5 { 1 + } array`

  `[ 1 2 3 4 5 ]`

Array Operations

```
>> [10 20 30] 20 remove      # remove element (first occurrence)
[ 10 30 ]

>> [10 20 30] 2 remove-at    # remove at index
[ 10 20 ]

>> [10 20 30] 0 9 insert     # insert at index
[ 9 10 20 30 ]

>> [10 20 30] 3 9 insert     # insert at index
[ 10 20 30 9 ]
```

Array Operations

```
>> [10 20 30] 30 find        # get index of element
2

>> [10 20 30] 40 find        # returns nil if not found
nil
    0 1 2 3 4 5
>> [7 8 9 7 8 9] 7 1 find-from    # start at index 1
3
```

Arrays Operations

```
     0 1 2 3 4
>> [1 2 3 4 5] 1 3 slice   # part of an array
[ 2 3 ]                    # from (inclusive), to (exclusive)

>> [1 2 3] reverse
[ 3 2 1 ]

>> [1 2 3 4 5 6] shuffle
[ 4 2 5 3 1 6 ]

>> sort
[ 1 2 3 4 5 6 ]
```

CHARACTERS AND STRINGS

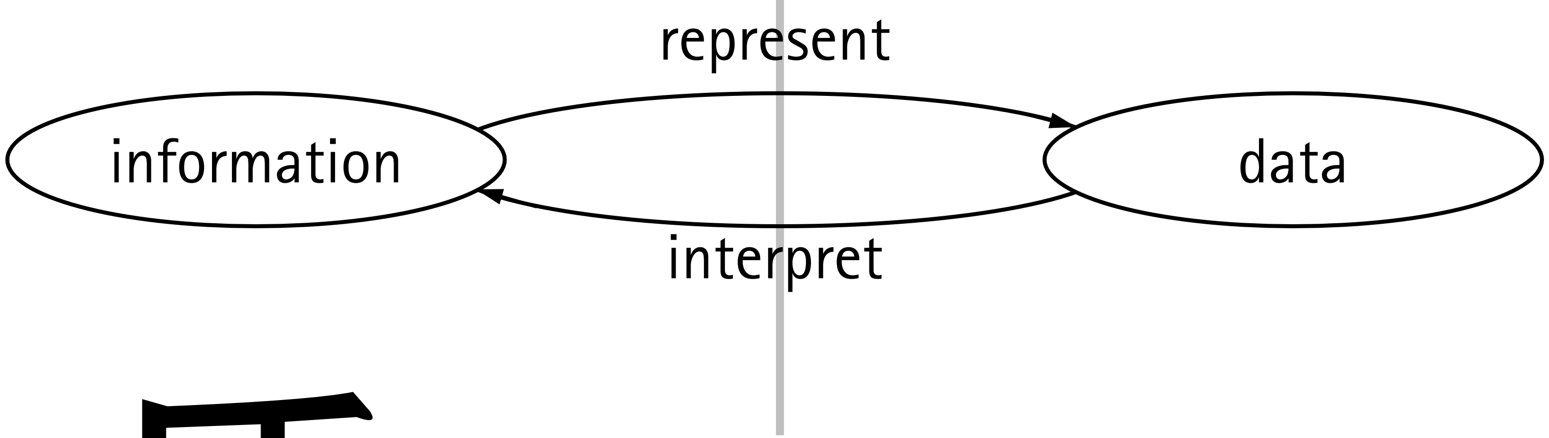
Concatenation and Length of Strings

```
>> "what a" " lovely" + " day" +
"what a lovely day"

>> "abc" length  # number of characters
3
```

Representing Characters

World/Domain: | Computer/Program:



乕

1. tiger
2. brave, fierce
3. a surname

entry #20053 in Unicode table

Accessing Characters, Strings <-> Characters


```
>> "abc" 0 get            # get first character
"a"

>> "abc" str->chars
[97 98 99]                # 'a' has ASCII/Unicode value 97

>> "abc" str->chars 0 get
97                        # extract from array

>> 97 char->str           # convert ASCII/Unicode value
"a"                       # to string

>> 20053 char->str        # convert Unicode value
"乕"
```

The ASCII Table

- ASCII = American Standard Code for Information Interchange
- Representing characters as numbers
- Interpreting numbers as characters
- Example:
  - character: A (information)
  - represented as number: 65 (data)
- Problem with ASCII: only 7 bits, not enough to encode all characters → Unicode

Decimal	Character	Meaning
0	'\0'	null character
9	'\t'	horizontal tab
10	'\n'	newline
13	'\r'	carriage return
32	' '	blank
33	'!'	
…	…	
47	'/'	
48	'0'	digits
…	…	
57	'9'	
58	':'	
…	…	
65	'A'	uppercase letters
…	…	
90	'Z'	
…	…	
97	'a'	lowercase letters
…	…	
122	'z'	
…	…	

String Operations

- `"HELLO" lower-case → "hello"`
- `"hello" upper-case → "HELLO"`
- `"  hello "  trim → "hello"`
- `"hello" "l" "x" replace-first → "hexlo"`
- `"hello" "l" "x" replace-all → "hexxo"`
- `"hello" 0 remove-at → "ello"`
- `"ello" 0 "h" insert → "hello"`

String Operations

- "hello" "ll" contains → true
- "hello" "lo" find → 3
- "hello" "h" 1 find-from → nil
- "he<u>ll</u>o" 2 4 slice → "ll"
- "this is nice" " " split → ["this" "is" "nice"]
- "this,is,nice" "," split → ["this" "is" "nice"]

LOOPS

for: Loop over Ranges

- General loop

```
5 x!
1 i!
{   i 10 > breakif
    i x * println
    i 1 + i!
} loop
```

- for loop over range

```
5 x!

1 11 {
    x * println
} for
```

```
lower upper {...} for
```

- lower index inclusive
- upper index exclusive

for: Loop over Ranges

- with indices on the stack

```
1 11 {
    x * println
} for
```

- with index variable i

```
5 x!
1 11 { i!
    i x * println
} for
```

- with index variable i and lam

```
1 11 { i!
    i x * println
} lam for
```

- with lam and parameters

```
1 11 (i) {
    i x * println
} lam for
```

Nested Loops

```
1 5 { row!
    1 10 { column!
        row print  "," print  column print  " " print
    } for
    "" println
} for
```

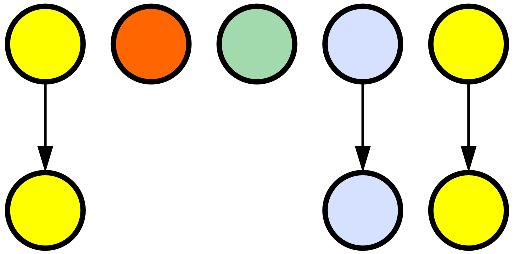
- Output:

```
1,1 1,2 1,3 1,4 1,5 1,6 1,7 1,8 1,9
2,1 2,2 2,3 2,4 2,5 2,6 2,7 2,8 2,9
3,1 3,2 3,3 3,4 3,5 3,6 3,7 3,8 3,9
4,1 4,2 4,3 4,4 4,5 4,6 4,7 4,8 4,9
```

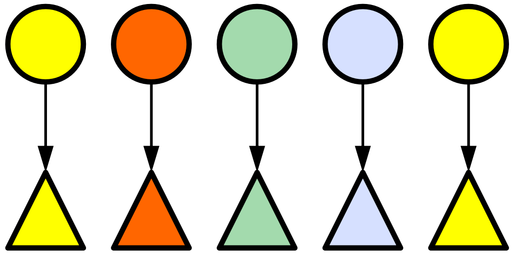
ARRAY-PROCESSING OPERATIONS
(OPTIONAL TOPIC)

Array-Processing Operations: filter, map, fold/reduce

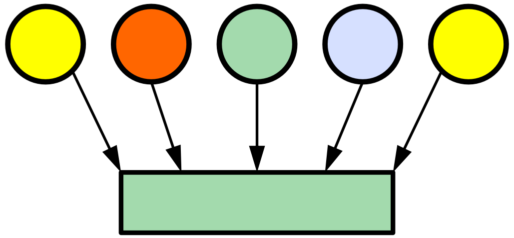
filter:



map:



fold / reduce:



- Operations on each element of an array
- Treat the array as a whole
- Powerful form of abstraction

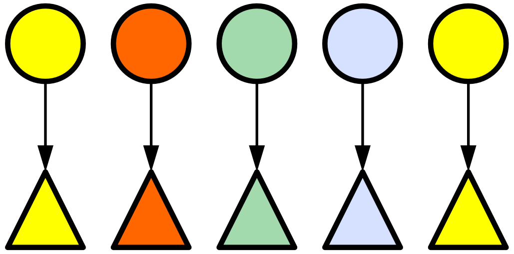
for: Loop over Arrays (map)

```
[10 20 30] a!
>> a { print } for # print each element
102030

>> [ a { 1 + } for ] # add 1 to each element, new array
[ 11 21 31 ]

>> ["peanut sauce" "rice" "vegetables"] a!
>> [a { "!" + } for]
[ "peanut sauce!" "rice!" "vegetables!" ]
```

map:



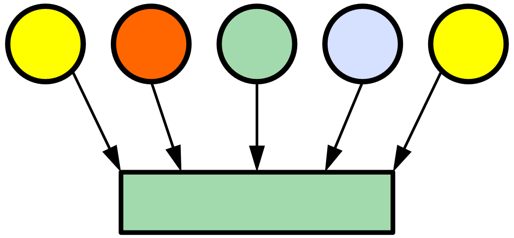
for: Loop over Arrays (fold/reduce)

```
>> [10 20 30] a!
>> 0 a {+} for # compute the sum
60

>> ["vegetables" "rice" "pizza" "burger"] a!
>> 0 a {length +} for # total number of characters
25

>> [1 "hello" -3.14 "world" :yummy] a!
>> 0 a { str? { 1 + } if } for # count strings only
2
```

fold / reduce:



for: Loop over Arrays (forall)

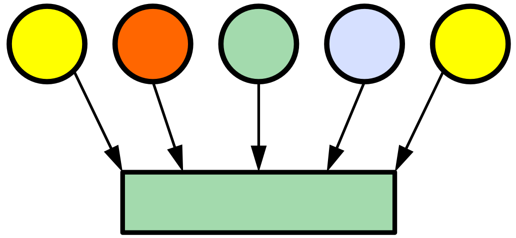
```
forall: (a :Arr, predicate :ExeArr -> :Bool) {
    true
    a {
        predicate not {
            pop false break
        } if
    } for
} fun

≫ [1 2 3 4] {3 <} forall
false
≫ [1 2 3 4] {5 <} forall
true
```

true iff, the predicate holds for all array elements

iff = if an only if

fold / reduce:



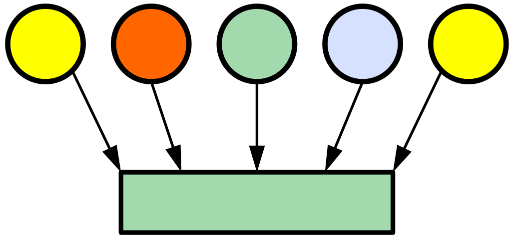
for: Loop over Arrays (exists)

```
exists: (a :Arr, predicate :ExeArr -> :Bool) {
    false
    a {
        predicate { pop true break } if
    } for
} fun

>> [1 2 3 4] {1 <} exists
false
>> [1 2 3 4] {2 <} exists
true
```

true, iff the predicate holds for at least one array element

fold / reduce:

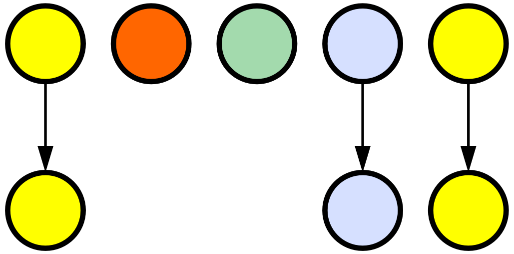


for: Loop over Arrays (filter)

```
>> [1 "hello" -3.14 "world" :yummy] a!
>> [ a (x) { x str? { x } if } lam for ]
[ "hello" "world" ]


>>[ a { dup str? not { pop } if } for ]
[ "hello" "world" ]
```

filter:



for: Loop over Arrays (filter)

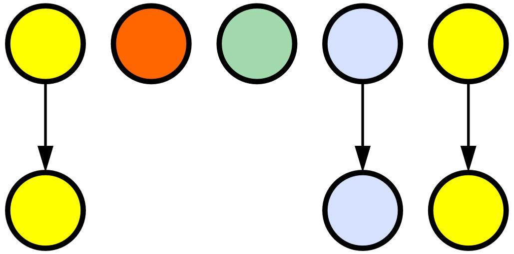
```
filter: (a :Arr, predicate :ExeArr) {
    [a { dup predicate not { pop } if } for]
} fun

>> [1 0 6 -3] { 0 > } filter
[ 1 6 ]

>> ["a" "b" 6 -3] { int? } filter
[ 6 -3 ]

>> ["a" "b" 6 -3] { str? } filter
[ "a" "b" ]
```

filter:



only keep the elements for which the predicate holds

Summary

- Operators
  - $\{p_1\ p_2 \ldots\}$ and/or, cond, cond-fun, stack, types
- Enumerations
  - Fixed number of categories (:cat, :dog, :mouse)
- Intervals
  - One or more ranges of numbers
  - (:no-tax :low-tax :high-tax + interval boundaries)
- Array operations
- Characters and Strings
- Loops