Leibniz Universität Hannover

Programmieren 1

Compound Data and Variant Data

Human-Computer Interaction Group

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

Leibniz Universität Hannover

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

- Operators
  - [p_1 p_2 ...] and/or, cond, cond-fun, stack, types
- Recipe for enumerations
  - Fixed number of categories
- Recipe for intervals
  - One or more ranges of numbers
- Array operations
- Characters and Strings
- Loops

Review: Local Variables and Dictionaries

```
1 a!

f: {
    a print
} fun       # a mit Wert 1 wird in lokales Dictionary kopiert

a 1 + a!

g: {
    a print
} fun       # a mit Wert 2 wird in lokales Dictionary kopiert

f g         # Ausgabe: 12
```

Preview

- Key-value arrays
- Data definitions
- Recipe for Compound Data (Product Types)
- Recipe for Variant Data (Sum Types)
- Recursion
- Recipe for Self-Referential Data (Recursive Types)

KEY-VALUE ARRAYS

Key-Value Arrays: Symbols as Keys

- Storing pairs of keys and values allows accessing values by key
  - The key has to precede the value: `key: value`
- Example

```
>> [x: 10 y: 20] point!
>> point :x get               # get x-coordinate by key
10
>> point .:x                  # equivalent abbreviation
10
>> point :y get               # get y-coordinate by key
20
>> point .:y                  # equivalent abbreviation
20
>> point .:z                  # try non-existent key
nil
```

Key-Value Arrays: Strings as Keys

```
>> planet-diameters: ["Mercury" 4878, "Venus" 12104,
        "Earth" 12756, "Mars" 6780, "Jupiter" 139822] !
>> planet-diameters "Earth" get    # use string as key
12756
>> planet-diameters "Pluto" get    # try non-existing key
nil
>> planet-diameters "Earth" 0 key-get   # 0 if key not present
12756
>> planet-diameters "Pluto" 0 key-get   # 0 if key not present
0
```

RECIPE FOR COMPOUND DATA (PRODUCT TYPES)

Design Recipes

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

Compound Data (Product Types)

- Group a fixed number of (potentially) different kinds of data
  - Product types: Cartesian product of components
  - Components may be atomic or structured
- Examples
  - Person with components name and age
  - Postal address with components street name, house number, zip code
  - 2D point with x- and y-coordinates as components
  - Linear function $f(x) = mx + b$ with slope m and intercept b as components
- Data definition
  - In PostFix, arrays are used to represent compound data

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

```
#<
Define a function that evaluates a linear function
of the form f(x) = mx + b.
Define a function that computes the intersection of two
linear functions of the form f(x) = mx + b, if an intersection exists.
>#
```

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 (and constructor function)

```
# A linear function f(x) = mx + b
# has components slope m and intercept b.

# Constructor function.
line: (m :Num, b :Num -> :Arr) {
    [m: m, b: b] # key-value array
} fun
```

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

```
value: (f :Arr, x :Num -> :Num)
intersect: (f :Arr, g :Arr -> :Obj)
```

4a. Function Stub

- Function stub returns an arbitrary value from the function's range
- The function stub can be executed

```
value: (f :Arr, x :Num -> :Num) {
    0
} fun

intersect: (f :Arr, g :Arr -> :Obj) {
    0
} fun
```

4b. Purpose Statement

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

```
# Evaluates linear function f at position x.
value: (f :Arr, x :Num -> :Num) {
    0
} fun
# Intersects two linear functions f and g.
# Returns :none if no intersection exists.
# Returns :all if f and g are identical.
intersect: (f :Arr, g :Arr -> :Obj) {
    0
} fun
```

5. Examples and Expected Results (Test Function)

```
value-test: {
    2 3 line f!          # f(x) = 2x + 3
    f 0 value 3 test=    # f(0) = 3
    f 1 value 5 test=    # f(1) = 5
    f 2 value 7 test=    # f(2) = 7
    test-stats
} fun
```

5. Examples and Expected Results (Test Function)

```
intersect-test: {
    # identical lines
    3 4 line  3 4 line  intersect  :all  test=
    # parallel, but not identical lines
    3 4 line  3 5 line  intersect  :none  test=
    1e-10 eps! # tolerance
    # f(x) = x + 2, g(x) = -x + 2
    1 2 line -1 2 line intersect 0 eps test~=
    # f(x) = x + 2, g(x) = -x + 4
    1 2 line -1 4 line intersect 1 eps test~=
    # f(x) = x - 1, g(x) = -3x + 3
    1 -1 line -3 3 line intersect 1 eps test~=
    test-stats
} fun
```

6. Function Body (value)

```
# Evaluates linear function f at position x.
value: (f :Arr, x :Num -> :Num) {
    f .:m  x *  f .:b  +           # x .:m  means  x :m get
} fun


# Could also be written as:
value: (f :Arr, x :Num -> :Num) {
    f :m get   # get slope m
    x *        # compute m * x
    f :b get   # get intersect b
    +          # compute m * x + b
} fun
```

6. Function Body (intersect)

```
intersect: (f :Arr, g :Arr -> :Obj) {
    f .:m m1!, f .:b b1! # f(x) = m1 x + b1
    g .:m m2!, g .:b b2! # g(x) = m2 x + b2
    {
        { m1 m2 =  b1 b2 =  and } { :all } # f and g are identical
        { m1 m2 =  b1 b2 !=  and } { :none } # not identical, but parallel
        { true } { # all other cases
            # f(x) = g(x)
            # <=> m1 x + b1 = m2 x + b2
            # <=> x = (b2 - b1) / (m1 - m2)
            b2 b1 -  m1 m2 -  /
        }
    } cond
} fun
```

7. Testing

- Call test function

```
value-test
intersect-test
```

- Test results

```
line.pf, line 13: Check passed.
line.pf, line 14: Check passed.
line.pf, line 15: Check passed.
All 3 tests passed!
line.pf, line 43: Check passed.
line.pf, line 44: Check passed.
line.pf, line 46: Check passed.
line.pf, line 47: Check passed.
line.pf, line 48: Check passed.
All 5 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
  - Abstract components with reusable helper functions
  - Allows changing the representation without breaking code

Accessor Functions to Abstract from Representation

- Without accessor functions for components

```
value: (f :Arr, x :Num -> :Num) {
    f .:m x * f .:b +
} fun
```

- With accessor functions for components

```
line-m: (f :Arr -> :Num) { f .:m } fun
line-b: (f :Arr -> :Num) { f .:b } fun

value: (f :Arr, x :Num -> :Num) {
    f line-m x * f line-b +
} fun
```

... or simply:

```
line-m: { :m get } !
line-b: { :b get } !
```

Changing the Representation (to array without keys)

```
# constructor function
line: (m :Num, b :Num -> :Arr) {
    [m b]
} fun

# accessor functions
line-m: (f :Arr -> :Num) { f 0 get } fun
line-b: (f :Arr -> :Num) { f 1 get } fun

value: (f :Arr, x :Num -> :Num) {
    f line-m x * f line-b +
} fun
```

Note: Changed representation

… or simply:

```
line-m: { 0 get } !
line-b: { 1 get } !
```

Note: No change in value function!

DATA DEFINITIONS FOR COMPOUND DATA

Data Definitions for Compound Data (datadef)

- Use datadef to automatically generate constructor, accessor, and type test (detector) functions
- Creates a new type name
- You write:

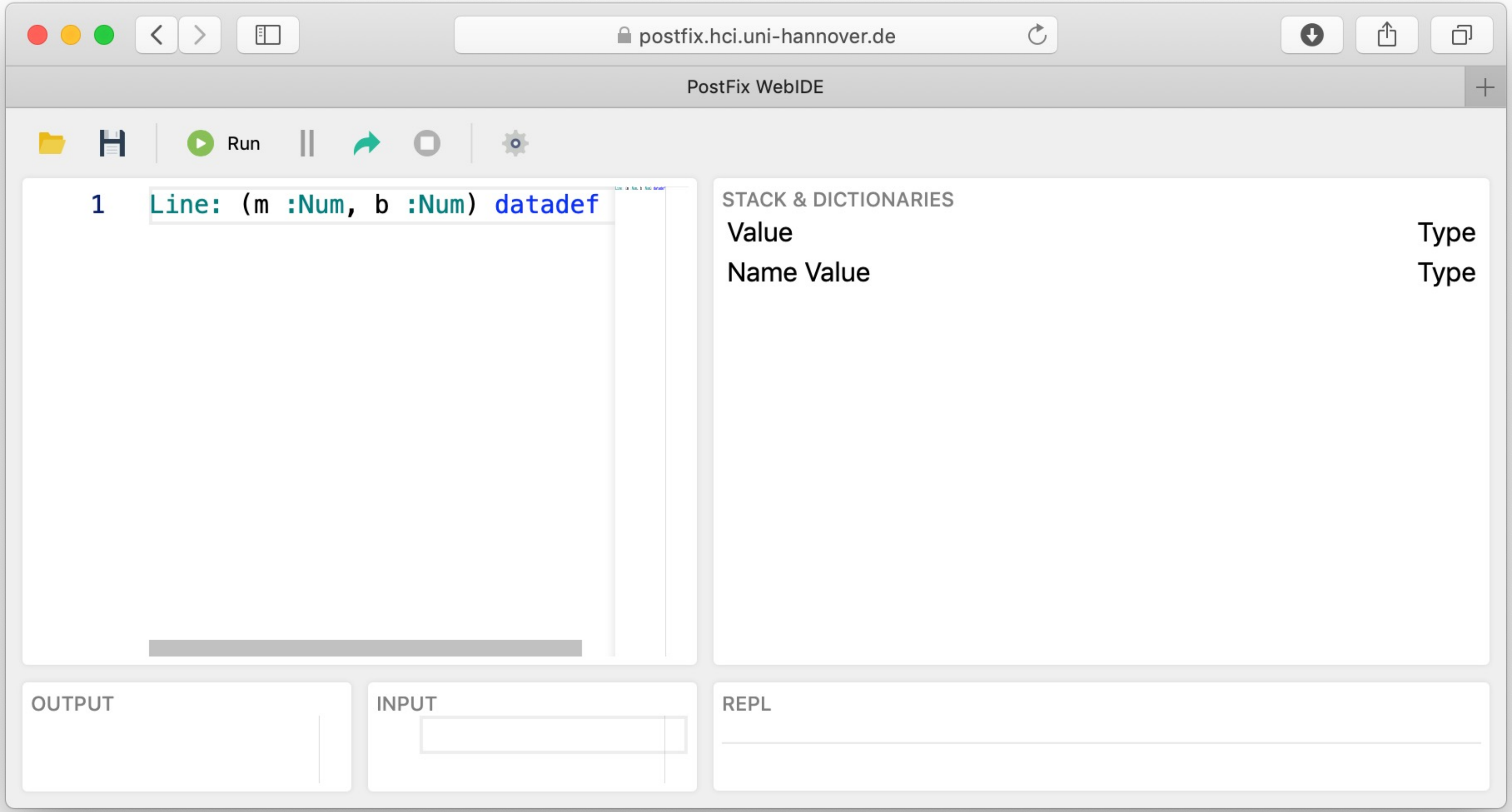
```
Line: (m :Num, b :Num) datadef
```

- PostFix generates:

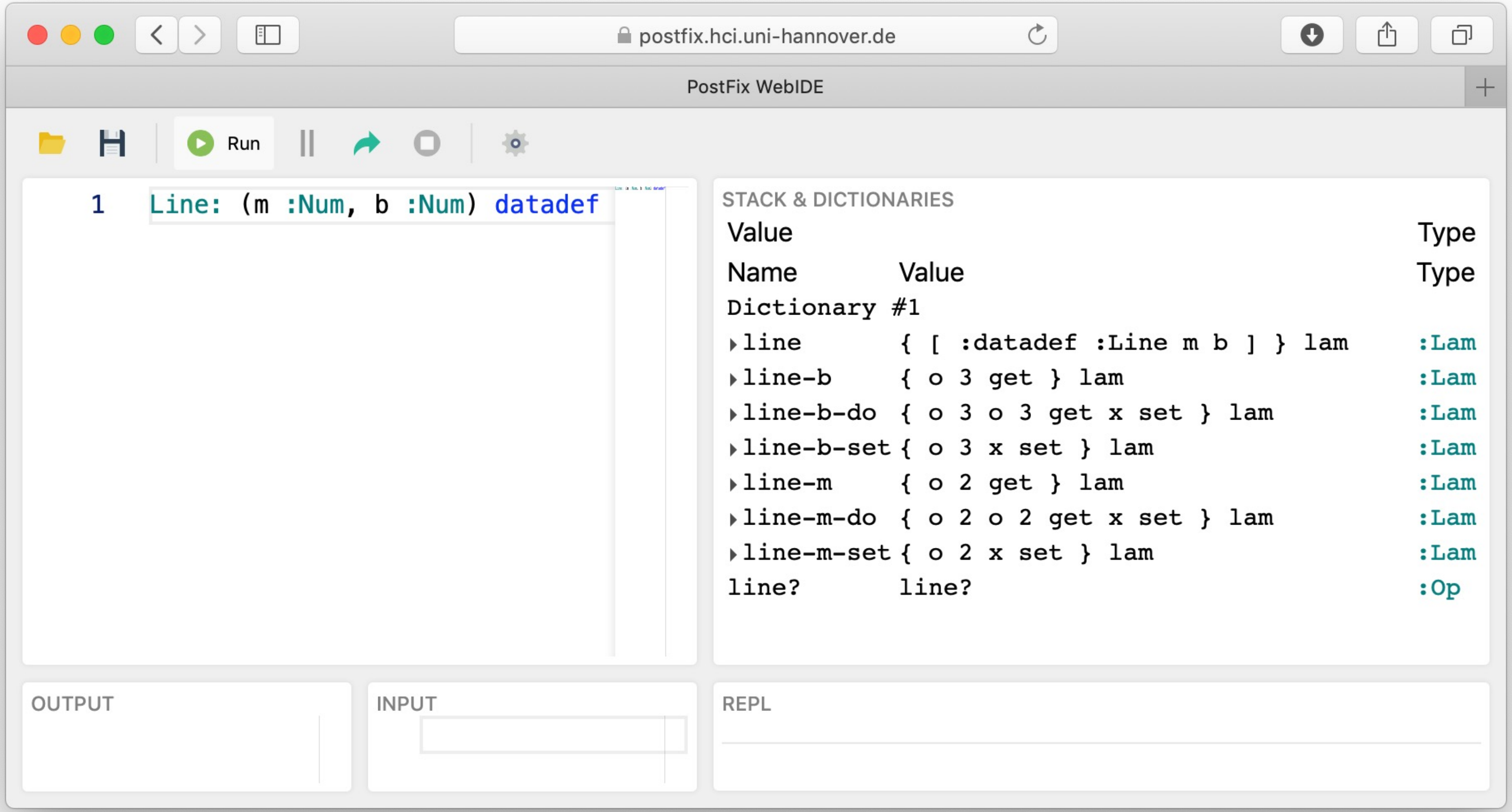
```
line: (m :Num, b :Num -> :Line) { … } fun
line-m: (f :Line -> :Num) { … } fun
line-b: (f :Line -> :Num) { … } fun
line?: (o :Obj -> :Bool) { … } fun
```

Note: New type name `Line`

Data Definitions (datadef) for Compound Data



Data Definitions (datadef) for Compound Data



Generated Constructor, Accessor, and Type Test Functions

```
# constructor function
line: (m :Num, b :Num -> :Line) {
    [:datadef :Line m b]
} fun

# accessor function
line-m: (o :Line -> :Num) {
    o 2 get
} fun

# accessor function
line-b: (o :Line -> :Num) {
    o 3 get
} fun
```

```
# type test function (detector)
line?: (o :Obj -> :Bool) {
   { { o arr? }
     { o length 2 >= }
     { o 0 get :datadef = }
     { o 1 get :Line = } } and
} fun
```

Use of Generated Functions

```
# Evaluates linear function f at position x.
value: (f :Line, x :Num -> :Num) {
    f line-m x *  f line-b  +
} fun
```

Note: Use of new type `Line`

Setting a Component

```
Point: (x :Int, y :Int) datadef
1 2 point p!
p 5 point-x-set q!
p str println # [ :datadef :Point 1 2 ]
q str println # [ :datadef :Point 5 2 ]
p {10 +} point-x-do r!
r str println # [ :datadef :Point 11 2 ]
r 3 5 set s!  # or just use array set operator
s str println # [ :datadef :Point 11 5 ]
```

RECIPE FOR VARIANT DATA (SUM TYPES)

Design Recipes

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

Variant Data (Sum Types)

- Represent data that can take on one of different variants/forms
  - Such data types are called variant, (tagged) union, or sum type
- Example
  - A program needs to handle different variants of 2D shapes
  - The shape variants are:
    - Rectangle: width and height
    - Circle: radius
- Data definition
  - PostFix uses arrays to represent variant data

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

```
#<
Define a function that computes the area of a shape.
A shape can be one of a rectangle or a circle.
 >#
```

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 (and constructor function)

```
# enumeration of shape variants:
# :rect, :circle

rect: (width :Num, height :Num -> :Arr) { # constructor
    [rect:  width: width, height: height]
} fun

circle: (radius :Num -> :Arr) { # constructor
    [circle:  radius: radius]
} fun
```

2. Data Definition (Accessor and Detector Functions)

```
rect-width: (r :Arr -> :Num) { # accessor function
    r .:width
} fun

rect-height: (r :Arr -> :Num) { # accessor function
    r .:height
} fun

rect?: (o :Obj -> :Bool) { # detector function
  [ { o arr? }
    { o length 5 = }
    { o 0 get :rect = } ] and
} fun
```

2. Data Definition (Accessor and Detector Functions)

```
circle-radius: (p :Arr -> :Num) { # accessor function
    p .:radius
} fun

circle?: (o :Obj -> :Bool) { # detector function
  [ { o arr? }
    { o length 3 = }
    { o 0 get :circle = } ] and
} fun
```

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

```
area: (shape :Arr -> :Num)
```

4. Function Stub and Purpose Statement

- Briefly describes what the function does. Ideally as a single sentence. Multiple sentences may be necessary.
- Function stub returns an arbitrary value from the function's range

```
# Computes the area of a shape.
area: (shape :Arr -> :Num) {
    0
} fun
```

5. Examples and Expected Results (Test Function)

```
area-test: {
    1 2 rect area 2 test=
    3 4 rect area 12 test=
    1e-10 eps!
    2 circle area, PI 2 * 2 *, eps test~=
    9.2 circle area, PI 9.2 * 9.2 *, eps test~=
    test-stats
} fun
```

6. Function Body (value)

```
# Computes the area of a shape.
area: (shape :Arr -> :Num) {
    {shape rect?} { # rectangle variant
        shape rect-width
        shape rect-height *
    }
    {shape circle?} { # circle variant
        shape circle-radius
        shape circle-radius *
        PI *
    }
} cond-fun
```

7. Testing

- Call test function

  `area-test`

- Test results

  ```
  shapes.pf, line 48: Check passed.
  shapes.pf, line 49: Check passed.
  shapes.pf, line 51: Check passed.
  shapes.pf, line 52: Check passed.
  All 4 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
  - Automatically create constructor, detector, and accessor functions using datadef

DATA DEFINITIONS FOR VARIANT DATA

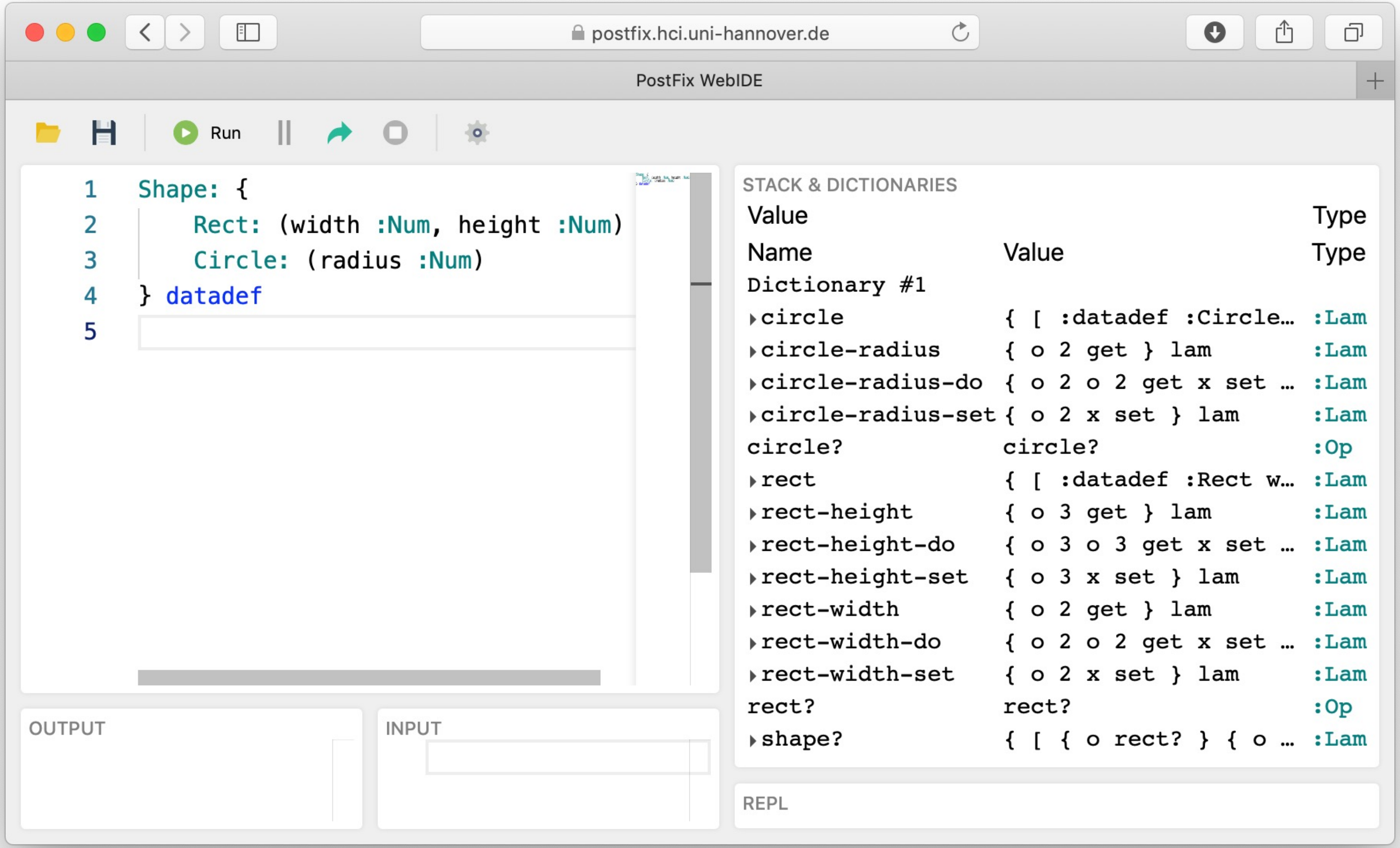
Data Definitions for Variant Data (datadef)

- Use datadef to automatically generate constructor, accessor, and type test (detector) functions
- Example: You write:

```
Shape: {
    Rect: (width :Num, height :Num)
    Circle: (radius :Num)
} datadef
```

- PostFix generates functions for each variant:
  - Constructors: rect, circle
  - Detectors: shape?, rect? circle?
  - Accessors: rect-width, rect-height, circle-radius

Data Definitions (datadef) for Compound Data



Use of Generated Functions

```
# Compute the area of a shape.
area: (shape :Shape -> :Num) {
    {shape rect?} {
        shape rect-width
        shape rect-height *
    }
    {shape circle?} {
        shape circle-radius
        shape circle-radius *
        PI *
    }
} cond-fun
```

Note: Use of new type **Shape**

RECURSION

Recursion

- Solve a problem by solving simpler problems of the same kind and combining the results
- Recursive definition
  - The definition contains what is defined
  - A natural number is either zero or the successor of a natural number
- Recursive data
  - A list is either empty or a value followed by a list
- Recursive algorithm
  - To compute the factorial of n, compute the factorial of n-1 and multiply the result by n: f(1) = 1, f(n) = n * f(n-1), for n = 2, 3, ...
  - A non-recursive base case: factorial of 1
  - A way to reduce problem towards base case: f(n) = n * f(n-1)

Found in a book index:
Recursion, see Recursion.

List:
real world: shopping list
computer science: a way to organize a collection of data items

Recursive Definition of Factorial

- Pseudocode
  - fac(0) = 1
  - fac(n) = n * fac(n - 1)
- PostFix

```
fac: (n :Int -> :Int) {
    { n 1 <= } { 1 }
    { true } { n 1 - fac  n  * }
} cond-fun
```

base case

recursion ends for $x \leq 1$

recursive call

recursive call on a smaller problem, one step closer towards recursion end

- Examples

0 fac	→	1	3 fac	→	6
1 fac	→	1	4 fac	→	24
2 fac	→	2	5 fac	→	120

Iterative Definition of Factorial

```
fac: (n :Int -> :Int) {
    1 # initial value
    1 n 1 + {*} for    # 1..n
} fun
```

Recursive Definition of Lists

- Data Definition

```
List: {                                    # A list is either
  End: ()                                  # empty or
  Pair: (value :Obj, rest :List)           # a value and a list.
} datadef
```

base case

recursive definition

self-referential data definition

List in computer science: a way to organize a collection of values

- Examples

```
end ➔ [:End]
10 end pair ➔ [Pair: 10 [:End]]
10 20 end pair pair ➔ [Pair: 10 [Pair: 20 [:End]]]
```

Length of a List

- The empty list has length 0. A non-empty list pair(value, rest) is one larger than the rest of the list.

```
list-length: (a :List -> :Int) {
    { a end? } { 0 }                 base case
    { a pair? } { a pair-rest  list-length  1 + }
} cond-fun                                recursive call
```

- Examples

```
end list-length ➔ 0
5 end pair list-length ➔ 1
5 6 end pair pair list-length ➔ 2
```

Structure of Data Mirrors Structure of Algorithm

- Recursive data

```
List: {                                # A list is either
  End: ()                              # empty or
  Pair: (value :Obj, rest :List) # a value and a list.
} datadef
```

- Recursive algorithm

```
list-length: (a :List -> :Int) {
  { a end? }  { 0 }                     # len(empty list) = 0
  { a pair? } { a pair-rest  list-length  1 + }
} cond-fun                              # len(pair(value, rest)) = 1 + len(rest)
```

RECIPE FOR SELF-REFERENTIAL DATA (RECURSIVE TYPES)

Design Recipes

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

Recipe for Self-Referential Data (Recursive Types)

- Represent data that can take on one of different variants at least one of which is self-referential
  - A special case of variant data: At least one branch is self-referential and at least one branch is not self-referential
  - Recursive types: The type to be defined is mentioned in its definition
- Self-referential data can represent information of arbitrary size
- Example
  - A list is either empty (variant 1) or a value followed by a list (variant 2)
- Data definition
  - PostFix uses arrays to represent self-referential data
  - datadef allows to conveniently create such types

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

```
#<
Compute the sum of the values of a list of integer numbers.
 >#
```

2a. Data Definition

- How should domain information be represented as data in the program? How to interpret the data as real-world information?
- Data definition
  - Determine and name the variants (here: `End` and `Pair`)
  - Identify self-references
  - Determine the types in each variant
    - `End`: empty parameter list, not self-referential
    - `Pair`: value is an integer number, rest is the self-reference to `List`

```
List: {
  End: ()                              # empty list
  Pair: (value :Int, rest :List) # has self-reference
} datadef
```

2b. Example Values for Data Definition

- Create at least one example value per variant in the data definition
- Create examples that use the self-referential variant(s) more than once (i.e., create examples of different lengths)

```
List: {
  End:  ()                            # variant 1
  Pair: (value :Int, rest :List) # variant 2
} datadef
```

- Examples
  - end # variant 1
  - 10 end pair # variant 2 and then variant 1
  - 10 20 end pair pair # variant 2, then variant 2, and finally variant 1
  - 10 20 30 end pair pair pair # variants 2, 2, 2, and finally 1

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

```
sum: (a :List -> :Int)
```

4a. Function Stub

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

```
sum: (a :List -> :Int) {
    0
} fun
```

4b. Purpose Statement

- Briefly describes what the function does (not how!). Ideally as a single sentence. Multiple sentences may be necessary.

```
# Computes the sum of the values of the list.
sum: (a :List -> :Int) {
    0
} fun
```

5b. Examples and Expected Results (Test Function)

- Write several examples with expected results, at least one per variant in the data definition
  - Use the example values created before (in 2b)

```
sum-test: {
    end sum  0  test=
    10 end pair sum  10  test=
    10 20 end pair pair sum  30  test=
    10 20 30 end pair pair pair sum  60  test=
    test-stats
} fun
```

6. Template

- Translate the data definition into a template
- Use the `cond` or `cond-fun` operators with condition-action pairs
  - Conditions: Write one condition per variant using the detector for the respective variant
  - Actions: Add the accessors relevant for the respective variant
  - Actions: Add one recursive call per self-reference

6. Template

- Data definition

```
List: {
  End: ()
  Pair: (value :Int, rest :List)
} datadef
```

self-reference

- Translate the data definition into a template

```
sum: (a :List -> :Int) {
  { a end? } { … }
  { a pair? } {
    … a pair-value … a pair-rest sum …
  }
} cond-fun
```

self-reference

recursive call on self-reference

6. Function Body

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
# Computes the sum of the values of the list.
sum: (a :List -> :Int) {
  { a end? } { … }
  { a pair? } {
    … a pair-value …
    … a pair-rest sum …
  }
} cond-fun
```

6. Function Body

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
# Computes the sum of the values of the list.
sum: (a :List -> :Int) {
  { a end? } { 0 }
  { a pair? } {
    … a pair-value …
    … a pair-rest sum …
  }
} cond-fun
```

purpose statement

(non-recursive) base case

assume that sum already does what the purpose statement says

induction step ("leap of faith")

6. Function Body

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
# Computes the sum of the values of the list.
sum: (a :List -> :Int) {
  { a end? } { 0 }
  { a pair? } {
    a pair-value
    a pair-rest sum  +
  }
} cond-fun
```

sum of empty list is 0 (base case)

if sum(rest) = r (induction hypothesis), then sum(pair(v, rest)) = v + r (induction step)

7. Testing

- Call test function

```
sum-test
```

- Test results

```
list.pf, line 18: Check passed.
list.pf, line 19: Check passed.
list.pf, line 21: Check passed.
list.pf, line 22: Check passed.
All 4 tests passed!
```

SELF-REFERENTIAL DATA, EXAMPLE 2

1. Problem Statement (Example 2)

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

- Example

```
#<
Write a function that determines whether
a value is present in a list of integer numbers.
>#
```

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

```
List: {
  End: ()                       # empty list
  Pair: (value :Int, rest :List) # has self-reference
} datadef
```

3. Function Name and Parameter List (Example 2)

- 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

```
list-contains: (a :List, x :Int -> :Bool)
```

4a. Function Stub (Example 2)

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

```
list-contains: (a :List, x :Int -> :Bool) {
    false
} fun
```

4b. Purpose Statement (Example 2)

- Briefly describes what the function does (not how!). Ideally as a single sentence. Multiple sentences may be necessary.

```
# Returns true if (and only if) a contains x.
list-contains: (a :List, x :Int -> :Bool) {
    false
} fun
```

5. Examples and Expected Results (Example 2)

- Write several examples with expected results, at least one per variant in the data definition
  - Use the example values created before (in 2b)

```
list-contains-test: {
  end 0 list-contains false test=
  1 end pair 2 list-contains false test=
  1 end pair 1 list-contains true test=
  1 2 end pair pair 3 list-contains false test=
  1 2 end pair pair 2 list-contains true test=
  1 2 3 end pair pair pair 3 list-contains true test=
  test-stats
} fun
```

6. Template (Example 2)

- Data definition

```
List: {
  End: ()
  Pair: (value :Int, rest :List)
} datadef
```

self-reference

- Translate the data definition into a template

```
list-contains: (a :List, x :Int -> :Bool) {
  { a end? } { … }
  { a pair? } {
    … a pair-value … a pair-rest list-contains …
  }
} cond-fun
```

self-reference

recursive call on self-reference

6. Function Body (Example 2)

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
# Returns true iff a contains x.
list-contains: (a :List, x :Int -> :Bool) {
  { a end? } { … }
  { a pair? } {
    … a pair-value …
    … a pair-rest list-contains …
  }
} cond-fun
```

6. Function Body (Example 2)

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
# Returns true iff a contains x.
list-contains: (a :List, x :Int -> :Bool) {
  { a end? } { false }
  { a pair? } {
    … a pair-value …
    … a pair-rest list-contains …
  }
} cond-fun
```

purpose statement

(non-recursive) base case

assume that list-contains already does what the purpose statement says

6. Function Body (Example 2)

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
# Returns true iff a contains x.
list-contains: (a :List, x :Int -> :Bool) {
  { a end? } { false }
  { a pair? } {
    … a pair-value …
    … a pair-rest x list-contains …
  }
} cond-fun
```

empty list does not contain anything (base case)

check whether x is first element or whether rest contains x

6. Function Body (Example 2)

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)
- Example

```
# Returns true iff a contains x.
list-contains: (a :List, x :Int -> :Bool) {
  { a end? } { false }
  { a pair? } {
    a pair-value x = { true }
    { a pair-rest x list-contains } if
  }
} cond-fun
```

empty list does not contain anything (base case)

check whether x is first element or whether rest contains x

7. Testing (Example 2)

- Call test function

  ```
  list-contains-test
  ```

- Test results

  list-contains.pf, line 16: Check passed.
  list-contains.pf, line 17: Check passed.
  list-contains.pf, line 18: Check passed.
  list-contains.pf, line 19: Check passed.
  list-contains.pf, line 20: Check passed.
  list-contains.pf, line 21: Check passed.
  All 6 tests passed!

8. Review and Revise (Example 2)

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

8. Review and Revise (Simplify Conditions)

```
list-contains: (a :List, x :Int -> :Bool) {
  { a end? } { false }
  { a pair? } {
    a pair-value x = { true }
    { a pair-rest x list-contains } if
  }
} cond-fun
```

simplify conditions

```
list-contains: (a :List, x :Int -> :Bool) {
  { a end? } { false }
  { a pair-value x = } { true }
  { true } { a pair-rest x list-contains }
} cond-fun
```

Summary

- Array processing operations
- Key-value arrays
- Data definitions
- Recipe for Compound Data (Product Types)
- Recipe for Variant Data (Sum Types)
- Recursion
- Recipe for Self-Referential Data (Recursive Types)