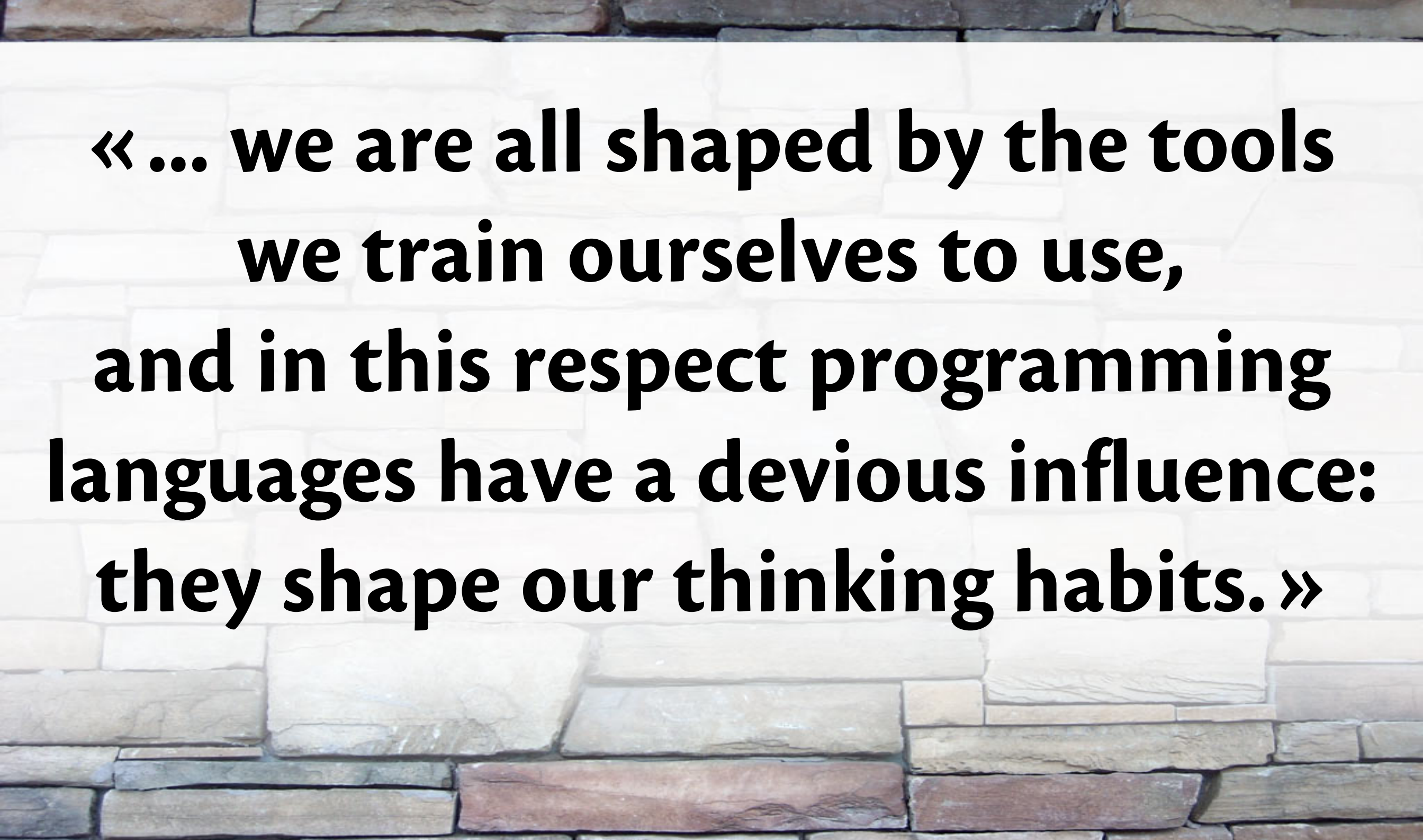


**« It is not only the violinist who is
shaped by his violin ... »**

(wrote Edsger W. Dijkstra in 2001)



**« ... we are all shaped by the tools
we train ourselves to use,
and in this respect programming
languages have a devious influence:
they shape our thinking habits. »**



A Superficial Exploration of Haskell

The background of the slide is a close-up photograph of a stone wall. The stones are of various sizes and shapes, with colors ranging from light beige to dark brown and reddish tones. The texture is rough and natural.

Haskell

is a general-purpose programming language

**is especially well-suited for writing
parsers / compilers / DSLs**

**is especially well-suited for prototyping
complex software in quick iterations**

The background of the slide is a close-up photograph of a stone wall. The stones are of various sizes and shapes, with colors ranging from light beige to dark brown and reddish tones. The texture is rough and natural.

Haskell

has been popular in academia for some time

is becoming increasingly popular in the software industry

Key Features of Haskell

functional and declarative

highly composable core

**encourages the use of concise code to
describe the problem, not the solution**

your list is your iteration

Key Features of Haskell

**lazy by default
(separates equation from execution)**

clean separation of concerns:

- **pure functional core
(no side-effects here)**
- **imperative shell for I/O
(no algorithms here)**

Key Features of Haskell

well-typed-ness and type inference

(a skilled Haskeller encodes problem domain invariants in the type system, so that entire classes of potential bugs will be compile-time errors instead of runtime exceptions that slipped through the cracks in your test suite)

**Cultural
shock
in
3...
2...
1...**

Here's a trivial snippet of code in Python:

```
>>> x = 1  
>>> x = x + 1  
>>> x  
2  
>>>
```

Nothing could be simpler!

Naively ... expect sunshine and flying colors ...
port
it
to
Haskell ...

```
λ: let x = 1  
λ: let x = x + 1  
λ: x  
^CInterrupted.  
λ:
```

Huh? What just happened?!?!

**To
emulate
mutation
in
Haskell ...**

λ : let x = 1

λ : x <- return \$ x + 1

λ : x

2

λ :

Laziness

Quick recap of the "map" function

```
λ: let list = [20..30]
```

```
λ: list
```

```
[20,21,22,23,24,25,26,27,28,  
  29,30]
```

```
λ: map (+1) list
```

```
[21,22,23,24,25,26,27,28,29,  
  30,31]
```

```
λ:
```

```
λ: let list1 = [20..70]
λ: let list2 = map (+1) list1
λ: :sprint list1
list1 =
λ: :sprint list2
list2 = _
λ:
λ: head list2
21
λ: :sprint list1
list1 = 20 : _
λ: :sprint list2
list2 = 21 : _
```



```
λ: take 5 list2
```

```
[21,22,23,24,25]
```

```
λ: :sprint list1
```

```
list1 = 20 : 21 : 22 : 23 : 24 : _
```

```
λ: :sprint list2
```

```
list2 = 21 : 22 : 23 : 24 : 25 : _
```



```
λ: list2 !! 17
```

```
38
```

```
λ: :sprint list1
```

```
list1 = 20 : 21 : 22 : 23 : 24 : 25 : 26 : 27 : 28 : 29 :  
        30 : 31 : 32 : 33 : 34 : 35 : 36 : 37 : _
```

```
λ: :sprint list2
```

```
list2 = 21 : 22 : 23 : 24 : 25 : _ : _ : _ : _ : _ :  
        _ : _ : _ : _ : _ : _ : 38 : _
```

```
λ: length list2
```

```
51
```

```
λ: :sprint list1
```

```
list1 = [20,21,22,23,24,25,26,27,28,29,  
        30,31,32,33,34,35,36,37,38,39,  
        40,41,42,43,44,45,46,47,48,49,  
        50,51,52,53,54,55,56,57,58,59,  
        60,61,62,63,64,65,66,67,68,69,70]
```

```
λ: :sprint list2
```

```
list2 = [21,22,23,24,25,_,_,_,_,_,  
        _,_,_,_,_,_,_,38,_,_,  
        _,_,_,_,_,_,_,_,_,_,  
        _,_,_,_,_,_,_,_,_,_,  
        _,_,_,_,_,_,_,_,_,_,_]
```

```
λ: sum list2
```

```
2346
```

```
λ: :sprint list2
```

```
list2 = [21,22,23,24,25,26,27,28,29,30,  
         31,32,33,34,35,36,37,38,39,40,  
         41,42,43,44,45,46,47,48,49,50,  
         51,52,53,54,55,56,57,58,59,60,  
         61,62,63,64,65,66,67,68,69,70,71]
```


How to write basic Haskell

Encapsulate:

- **sequencing**
- **decision**
- **iteration**

in functional abstractions.

Sequencing statement execution in Python

```
>>> x = 1
>>> y = x * 5
>>> z = y + 3
>>> z
8
>>>
```

Sequencing expression evaluation in Haskell

```
λ: let x = 1
λ: let f = (+3) . (*5)
λ:
λ: f x
8
λ:
```

Deciding between statement executions in Python

```
>>> x = 1
>>> y = 2
>>> if x == y:
...     print 'equal'
... else:
...     print 'not equal'
...
not equal
>>>
```

Haskell says:

**What's wrong with this picture
(from a separation-of-concerns
perspective)?**

Deciding between expression evaluations in Python

```
>>> x = 1
>>> y = 2
>>> z = 'equal' if x == y else 'not equal'
>>> z
'not equal'
>>>
```

Deciding between expression evaluations in Haskell

```
λ: let x = 1
```

```
λ: let y = 2
```

```
λ: let z = if x == y then "equal" else "not equal"
```

```
λ: z
```

```
"not equal"
```

```
λ:
```

Flavors of iteration you'll encounter in Haskell

**arranged according to
what kind of input is available and
what kind of output is desired**

Finite loop flavor #1

Available input:

- **a list of scalar values**
- **a function of one argument**

Desired output:

- **a list of scalar values**

Use "map"

(Also, your list is your iteration.)

Finite loop flavor #2

Available input:

- a list of scalar values
- a predicate

Desired output:

- a list of scalar values

Use "filter"

(Also, your list is your iteration.)

Finite loop flavor #3

Available input:

- **exactly one scalar value**
- **a list of functions
of one argument each**

Desired output:

- **a list of scalar values**

Use "<*>"

(An example will follow shortly.)

(Also, your list is your iteration.)

Finite loop flavor #4

Available input:

- a list of scalar values
- a function of two arguments
- the initial value of an accumulator (a scalar value)

Desired output:

- the final value of the accumulator (a scalar value that absorbs the entire contents of the input list)

Finite loop flavor #4

Use "foldl"

(An example will follow shortly.)

(Also, your list is your iteration.)

Finite loop flavor #5

Available input:

- a list of scalar values
- a function of two arguments
- the initial value of an accumulator (a scalar value)

Desired output:

- a list containing a "progress report" (what did the accumulator look like each step along the way?)

Finite loop flavor #5

Use "scanl"

(An example will follow shortly.)

(Also, your list is your iteration.)

(Potentially) endless loop flavor #1

Available input:

- **the initial value
of an accumulator
(a scalar value)**
- **a successor function**
- **a predicate
(a.k.a. exit condition)**

Desired output:

- **the final value
of the accumulator
after however many
successive function
applications**

(Potentially) endless loop flavor #1

Use "until"

(An example will follow shortly.)

(OK, list \neq iteration. Just this once.)

(Potentially) endless loop flavor #2

Available input:

- **the initial value of an accumulator (a scalar value)**
- **a successor function**
- **a predicate (a.k.a. exit condition)**

Desired output:

- **a list containing a "progress report" (what did the accumulator look like after each successive function application?)**

(Potentially) endless loop flavor #2

Use "unfoldr"

(An example will follow shortly.)

(Also, your list is your iteration.)



Flavors of iteration you'll encounter in Haskell

here come the promised examples

Using "<*>"

(finite loop flavor #3)

```
λ: let function_list_1 = [(+2), (*3), (^2)]
```

```
λ: let function_list_2 = [(*5), (+7), (*4), (subtract 10)]
```

```
λ: let i = 12
```

```
λ: import Control.Applicative
```

```
λ: function_list_1 <*> [i]
```

```
[14,36,144]
```

```
λ: function_list_2 <*> [i]
```

```
[60,19,48,2]
```

```
λ:
```

Using "foldl" (finite loop flavor #4)

```
λ: foldl (+) 5 [50..70]  
1265
```

In Python we'd write:

```
>>> input_list = range(50,71)  
>>> accumulator = 5  
>>> for i in input_list:  
...     accumulator += i  
...  
>>> accumulator  
1265
```


Using "scanl" (finite loop flavor #5)

```
λ: scanl (+) 5 [50..70]  
[5,55,106,158,211,265,  
 320,376,433,491,550,610,  
 671,733,796,860,925,991,  
 1058,1126,1195,1265]
```

In Python we'd write:

```
>>> input_list = range(50,71)  
>>> accumulator = 5  
>>> ol = [accumulator]  
>>> for i in input_list:  
...     accumulator += i  
...     ol.append(accumulator)  
...  
>>> ol  
[5, 55, 106, 158, 211, 265,  
 320, 376, 433, 491, 550, 610,  
 671, 733, 796, 860, 925, 991,  
 1058, 1126, 1195, 1265]
```


Using "until" (endless loop flavor #1)

```
λ: until (>100) (*2) 1  
128
```

In Python we'd write:

```
>>> def succf(x):  
...     return x * 2  
...  
>>> acc = 1  
>>> while True:  
...     acc = succf(acc)  
...     if acc > 100:  
...         break  
...  
>>> acc  
128
```

Using "unfoldr" (endless loop flavor #2)

```
λ: :{  
  let operate_on x =  
    if x > 100  
    then  
      Nothing  
    else  
      Just (x, x * 2)  
  :}  
λ: unfoldr operate_on 1  
[1,2,4,8,16,32,64]
```

In Python we'd write:

```
>>> def succf(x):  
...     return x * 2  
...  
>>> acc = 1  
>>> ol = [accumulator]  
>>> while True:  
...     acc = succf(acc)  
...     if acc > 100:  
...         break  
...     ol.append(acc)  
...  
>>> ol  
[1, 2, 4, 8, 16, 32, 64]
```

Flavors of iteration you'll encounter in Haskell

**map, filter, $\langle * \rangle$, foldl, scanl,
until, unfoldr**

The background of the image is a stone wall. The top and bottom edges are composed of dark, reddish-brown stones. The central portion of the image is a large, white rectangular area. Centered within this white area is the text "Follow me on GitHub" in a large, bold, black font. Directly below this, the URL "github.com/dserban" is displayed in a slightly smaller, bold, black font.

Follow me on GitHub
github.com/dserban



lambda.rosedu.org