# Haskell and Domain-Specific Languages

## Haskell nejen pro informatiky

Otakar Smrž

Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics
Charles University in Prague
otakar.smrz@mff.cuni.cz

https://wiki.ufal.ms.mff.cuni.cz/courses:pfl080

Part I

# Higher-Order Functions

# Function Composition

One of the most fancy higher-order functions is function composition:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
(f . g) x = f (g x)
```

# Function Composition

One of the most fancy higher-order functions is function composition:

```
(.) :: (a -> b) -> (c -> a) -> c -> b
(.) f g x = f (g x)
```

# Function Composition

One of the most fancy higher-order functions is function composition:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
(f . g) x = f (g x)
```

. . . which is unlike function application:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

# Function Composition

One of the most fancy higher-order functions is function composition:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
(f . g) x = f (g x)
```

...which is unlike function application:

```
($) :: (a -> b) -> (a -> b)
($) f = f
```

# Function Composition

One of the most fancy higher-order functions is function composition:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
(f . g) x = f (g x)
```

. . . which is unlike function application:

```
($) :: (a -> b) -> (a -> b)
($) f = f
```

Recall fixity declarations, sections and the `` and () notations.

# Function Composition

One of the most fancy higher-order functions is function composition:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
(f . g) x = f (g x)
```

. . . which is unlike function application:

```
($) :: (a -> b) -> (a -> b)
($) f = f
```

Recall fixity declarations, sections and the `` and () notations.

Visit Wikipedia for the explanation of $\eta$-conversion, $\beta$-reduction, and $\alpha$-conversion in the lambda calculus.

# Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

# Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- `(.)`, `($)`, **`flip`**, **`const`**, **`id`**

# Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- `(.)`, `($)`, **`flip`**, **`const`**, **`id`**
- **`curry`**, **`uncurry`**, **`iterate`**, **`until`**

# Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- `(.)`, `($)`, `flip`, `const`, `id`
- `curry`, `uncurry`, `iterate`, `until`
- `map`, `filter`, `zipWith`

# Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- `(.)`, `($)`, **`flip`**, **`const`**, **`id`**
- **`curry`**, **`uncurry`**, **`iterate`**, **`until`**
- **`map`**, **`filter`**, **`zipWith`**
- **`foldr`**, **`foldl`**, **`foldr1`**, **`foldl1`**, **`scanr`**, **`scanl`**, **`unfoldr`**, **`unzip`**

# Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- `(.)`, `($)`, **`flip`**, **`const`**, **`id`**
- **`curry`**, **`uncurry`**, **`iterate`**, **`until`**
- **`map`**, **`filter`**, **`zipWith`**
- **`foldr`**, **`foldl`**, **`foldr1`**, **`foldl1`**, **`scanr`**, **`scanl`**, **`unfoldr`**, **`unzip`**
- **`takeWhile`**, **`dropWhile`**, **`span`**, **`break`**, **`groupBy`**, **`nubBy`**

# Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- `(.)`, `($)`, **`flip`**, **`const`**, **`id`**
- **`curry`**, **`uncurry`**, **`iterate`**, **`until`**
- **`map`**, **`filter`**, **`zipWith`**
- **`foldr`**, **`foldl`**, **`foldr1`**, **`foldl1`**, **`scanr`**, **`scanl`**, **`unfoldr`**, **`unzip`**
- **`takeWhile`**, **`dropWhile`**, **`span`**, **`break`**, **`groupBy`**, **`nubBy`**
- **`and`**, **`or`**, **`all`**, **`any`**, **`maximum`**, **`minimum`**

# Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- `(.)`, `($)`, `flip`, `const`, `id`
- `curry`, `uncurry`, `iterate`, `until`
- `map`, `filter`, `zipWith`
- `foldr`, `foldl`, `foldr1`, `foldl1`, `scanr`, `scanl`, `unfoldr`, `unzip`
- `takeWhile`, `dropWhile`, `span`, `break`, `groupBy`, `nubBy`
- `and`, `or`, `all`, `any`, `maximum`, `minimum`
- `concat`, `length`, `elem`, `notElem`, `reverse`

# Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- `(.)`, `($)`, **`flip`**, **`const`**, **`id`**
- **`curry`**, **`uncurry`**, **`iterate`**, **`until`**
- **`map`**, **`filter`**, **`zipWith`**
- **`foldr`**, **`foldl`**, **`foldr1`**, **`foldl1`**, **`scanr`**, **`scanl`**, **`unfoldr`**, **`unzip`**
- **`takeWhile`**, **`dropWhile`**, **`span`**, **`break`**, **`groupBy`**, **`nubBy`**
- **`and`**, **`or`**, **`all`**, **`any`**, **`maximum`**, **`minimum`**
- **`concat`**, **`length`**, **`elem`**, **`notElem`**, **`reverse`**
- **`product`**, **`sum`**, **`foldl'`**, `($!)`

# Correctness

We can use equational reasoning to prove functions' properties.

# Correctness

We can use equational reasoning to prove functions' properties.

```
flip . flip  ≅  id
```

# Correctness

We can use equational reasoning to prove functions' properties.

```
flip . flip ≅ id
```

```
map f . map g ≡ map (f . g)
```

# Correctness

We can use equational reasoning to prove functions' properties.

```
flip . flip  ≅  id

map f . map g  ≡  map (f . g)

notElem  ≡  (.) not . elem
```

# Correctness

We can use equational reasoning to prove functions' properties.

```
flip . flip ≅ id

map f . map g ≡ map (f . g)

notElem ≡ (.) not . elem

reverse ≡ foldl (flip (:)) []
```

# Correctness

We can use equational reasoning to prove functions' properties.

```
flip . flip  ≅  id

map f . map g  ≡  map (f . g)

notElem  ≡  (.) not . elem

reverse  ≡  foldl (flip (:)) []
```

Why did we write $\cong$ somewhere, and not $\equiv$ everywhere?

# Correctness

We can use equational reasoning to prove functions' properties.

$$\texttt{flip . flip} \cong \texttt{id} \qquad\qquad (\texttt{flip . flip})\ \texttt{f} \equiv \texttt{f}$$

$$\texttt{map f . map g} \equiv \texttt{map (f . g)}$$

$$\texttt{notElem} \equiv \texttt{(.) not . elem}$$

$$\texttt{reverse} \equiv \texttt{foldl (flip (:)) []}$$

Why did we write $\cong$ somewhere, and not $\equiv$ everywhere?

Part II

# Tree Structures

# Trees

Write functions for folding and linearizing trees of these types:

# Trees

Write functions for folding and linearizing trees of these types:

```
data Tree a = Node a [Tree a]
```

# Trees

Write functions for folding and linearizing trees of these types:

```
data Tree a = Node a [Tree a]
```

```
data BinTree a b = Fork a (BinTree a b) (BinTree a b) |
                   Leaf b
```

# Trees

Write functions for folding and linearizing trees of these types:

```
data Tree a = Node a [Tree a]
```

```
data BinTree a b = Fork a (BinTree a b) (BinTree a b) |
                   Leaf b
```

Colored red-black trees can implement sets and finite maps.

# Trees

Write functions for folding and linearizing trees of these types:

```
data Tree a = Node a [Tree a]
```

```
data BinTree a b = Fork a (BinTree a b) (BinTree a b) |
                   Leaf b
```

Colored red-black trees can implement sets and finite maps.

Study the zipper representation of trees by Huet (1).

# HaXml

HaXml is a library for processing XML and DTD that provides interesting DSL for document transformations.

# HaXml

HaXml is a library for processing XML and DTD that provides interesting DSL for document transformations.

```haskell
data Element = Elem Name [Attribute] [Content]
data Content = CElem Element
             | CText String
```

# HaXml

HaXml is a library for processing XML and DTD that provides interesting DSL for document transformations.

```
data Element = Elem Name [Attribute] [Content]
data Content = CElem Element
             | CText String

type CFilter = Content -> [Content]
```

# HaXml

HaXml is a library for processing XML and DTD that provides interesting DSL for document transformations.

```
data Element = Elem Name [Attribute] [Content]
data Content = CElem Element
             | CText String
```

```
type CFilter = Content -> [Content]
```

Read the paper by Wallace and Runciman (2) and try out `Text.XML.HaXml`.

# References


Gérard Huet.
Functional Pearl. The Zipper.
*Journal of Functional Programming*, 5(7):549–554, 1997.

Malcolm Wallace and Colin Runciman.
Haskell and XML: Generic combinators or type-based translation?
In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 148–159. ACM Press, 1999.