CENG 3549 – Functional Programming

Modules & Lists and Strings & Recursive Functions

Burak Ekici

October 6, 2022

Outline

Structuring Code into Modules

- **note:** separate namespaces for functions and types

Structuring Code into Modules

- **note:** separate namespaces for functions and types
- split source code into several files

Structuring Code into Modules

- **note:** separate namespaces for functions and types
- split source code into several files
- for each module `Module` create file `Module.hs`

Structuring Code into Modules

- **note:** separate namespaces for functions and types
- split source code into several files
- for each module `Module` create file `Module.hs`
- module names always start with uppercase letters

Structuring Code into Modules

- **note:** separate namespaces for functions and types
- split source code into several files
- for each module `Module` create file `Module.hs`
- module names always start with uppercase letters
- start module by module header with optional export list

```
module Module (...) where
```

Structuring Code into Modules

- **note:** separate namespaces for functions and types
- split source code into several files
- for each module `Module` create file `Module.hs`
- module names always start with uppercase letters
- start module by module header with optional export list
  `module Module (...) where`
- export list is list of functions and types visible outside

Structuring Code into Modules

- **note:** separate namespaces for functions and types
- split source code into several files
- for each module `Module` create file `Module.hs`
- module names always start with uppercase letters
- start module by module header with optional export list

```
module Module (...) where
```

- export list is list of functions and types visible outside
- without export list all functions and types visible

Structuring Code into Modules

- **note:** separate namespaces for functions and types
- split source code into several files
- for each module `Module` create file `Module.hs`
- module names always start with uppercase letters
- start module by module header with optional export list
  `module Module (...) where`
- export list is list of functions and types visible outside
- without export list all functions and types visible

Example

```
module Stack where
type Stack a = [a]
empty = []
push  = (:)
pop s = (head s, tail s)
```

Type Synonyms

- **type** Stack a = [a] is a type synonym

Type Synonyms

- **type** Stack a = [a] is a type synonym
- gives alternative name for [a] (no new types involved)

Type Synonyms

- `type Stack a = [a]` is a type synonym
- gives alternative name for `[a]` (no new types involved)
- afterwards, both names may be used interchangeably

Type Synonyms

- `type Stack a = [a]` is a type synonym
- gives alternative name for `[a]` (no new types involved)
- afterwards, both names may be used interchangeably

Type Signatures

- every function `f` may be preceded by a type signature `f :: T`, stating that `f` is of type `T`

Type Synonyms

- `type Stack a = [a]` is a type synonym
- gives alternative name for `[a]` (no new types involved)
- afterwards, both names may be used interchangeably

Type Signatures

- every function `f` may be preceded by a type signature `f :: T`, stating that `f` is of type `T`
- good for documentation purposes

Type Synonyms

- `type Stack a = [a]` is a type synonym
- gives alternative name for `[a]` (no new types involved)
- afterwards, both names may be used interchangeably

Type Signatures

- every function `f` may be preceded by a type signature `f :: T`, stating that `f` is of type `T`
- good for documentation purposes

Example

```
push :: a -> Stack a -> Stack a
push = (:)
```

- note the partial application of `(:)`
- this is equivalent to `push x s = x : s`

Remark (imports)

- `import M` imports all functions and types exported by module `M`

Remark (imports)

- `import M` imports all functions and types exported by module `M`
- we may restrict to `f1`, ..., `fN`, writing `import M (f1, ..., fN)`

Remark (imports)

- `import M` imports all functions and types exported by module `M`
- we may restrict to `f1, ..., fN`, writing `import M (f1, ..., fN)`
- by `import M hiding (f1, ..., fN)` we import everything except for functions `f1` to `fN`

Remark (imports)

- `import M` imports all functions and types exported by module `M`
- we may restrict to `f1, ..., fN`, writing `import M (f1, ..., fN)`
- by `import M hiding (f1, ..., fN)` we import everything except for functions `f1` to `fN`
- `import qualified M` allows us to access all functions exported by `M` using prefix "`M.`"

Remark (imports)

- `import M` imports all functions and types exported by module `M`
- we may restrict to `f1, ..., fN`, writing `import M (f1, ..., fN)`
- by `import M hiding (f1, ..., fN)` we import everything except for functions `f1` to `fN`
- `import qualified M` allows us to access all functions exported by `M` using prefix "`M.`"
- `import qualified M as N`, similar to `import qualified M` but additionally rename `M` to `N`

Remark (imports)

- `import M` imports all functions and types exported by module `M`
- we may restrict to `f1, ..., fN`, writing `import M (f1, ..., fN)`
- by `import M hiding (f1, ..., fN)` we import everything except for functions `f1` to `fN`
- `import qualified M` allows us to access all functions exported by `M` using prefix "`M.`"
- `import qualified M as N`, similar to `import qualified M` but additionally rename `M` to `N`

Examples

- `import Stack`
- `import Stack (push, pop)`
- `import Stack hiding (pop)`
- `import qualified Stack`
- `import qualified Stack as S`

Outline

Definition (strings)

- strings are lists

Definition (strings)

- strings are lists
- type `String` is type synonym for `[Char]`

Definition (strings)

- strings are lists
- type `String` is type synonym for `[Char]`
- that is, strings are lists of characters

Definition (strings)

- strings are lists
- type `String` is type synonym for `[Char]`
- that is, strings are lists of characters
- consequently, all list functions apply also to `String`s

Definition (strings)

- strings are lists
- type `String` is type synonym for `[Char]`
- that is, strings are lists of characters
- consequently, all list functions apply also to `String`s

Remark (some implications)

- `[]` is same as `""` for strings

Definition (strings)

- strings are lists
- type `String` is type synonym for `[Char]`
- that is, strings are lists of characters
- consequently, all list functions apply also to `String`s

Remark (some implications)

- `[]` is same as `""` for strings
- `['h','e','l','l','o']` is same as `"hello"` for strings

Definition (strings)

- strings are lists
- type `String` is type synonym for `[Char]`
- that is, strings are lists of characters
- consequently, all list functions apply also to `String`s

Remark (some implications)

- `[]` is same as `""` for strings
- `['h','e','l','l','o']` is same as `"hello"` for strings

Example (some useful functions on strings)

- `lines :: String -> [String]` – breaks string at newlines

Definition (strings)

- strings are lists
- type `String` is type synonym for `[Char]`
- that is, strings are lists of characters
- consequently, all list functions apply also to `String`s

Remark (some implications)

- `[]` is same as `""` for strings
- `['h','e','l','l','o']` is same as `"hello"` for strings

Example (some useful functions on strings)

- `lines :: String -> [String]` – breaks string at newlines
- `unlines :: [String] -> String` – concatenates strings, inserting newlines

Definition (strings)

- strings are lists
- type `String` is type synonym for `[Char]`
- that is, strings are lists of characters
- consequently, all list functions apply also to `String`s

Remark (some implications)

- `[]` is same as `""` for strings
- `['h','e','l','l','o']` is same as `"hello"` for strings

Example (some useful functions on strings)

- `lines :: String -> [String]` – breaks string at newlines
- `unlines :: [String] -> String` – concatenates strings, inserting newlines
- `words :: String -> [String]` – breaks string at white space

Definition (strings)

- strings are lists
- type `String` is type synonym for `[Char]`
- that is, strings are lists of characters
- consequently, all list functions apply also to `String`s

Remark (some implications)

- `[]` is same as `""` for strings
- `['h','e','l','l','o']` is same as `"hello"` for strings

Example (some useful functions on strings)

- `lines :: String -> [String]` – breaks string at newlines
- `unlines :: [String] -> String` – concatenates strings, inserting newlines
- `words :: String -> [String]` – breaks string at white space
- `unwords :: [String] -> String` – concatenates strings, inserting spaces

Interlude (function composition)

- in mathematics $f \circ g$ usually denotes applying f after g

Interlude (function composition)

- in mathematics $f \circ g$ usually denotes applying f after g
- more precisely, $(f \circ g)(x) = f(g(x))$

Interlude (function composition)

- in mathematics $f \circ g$ usually denotes applying f after g
- more precisely, $(f \circ g)(x) = f(g(x))$
- only possible if output of g compatible with input of f, that is, $f : B \to C$ and $g : A \to B$

Interlude (function composition)

- in mathematics $f \circ g$ usually denotes applying f after g
- more precisely, $(f \circ g)(x) = f(g(x))$
- only possible if output of g compatible with input of f, that is, $f : B \to C$ and $g : A \to B$
- in Haskell: `(.) :: (b -> c) -> (a -> b) -> (a -> c)`

Interlude (function composition)

- in mathematics $f \circ g$ usually denotes applying f after g
- more precisely, $(f \circ g)(x) = f(g(x))$
- only possible if output of g compatible with input of f, that is, $f : B \to C$ and $g : A \to B$
- in Haskell: `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- try "`:info (.)`" in GHCi

Interlude (function composition)

- in mathematics $f \circ g$ usually denotes applying f after g
- more precisely, $(f \circ g)(x) = f(g(x))$
- only possible if output of g compatible with input of f, that is, $f : B \to C$ and $g : A \to B$
- in Haskell: `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- try "`:info (.)`" in GHCi

Examples

- `map (f . g) xs` – to every element of `xs`, first apply `g` and then `f`

Interlude (function composition)

- in mathematics $f \circ g$ usually denotes applying f after g
- more precisely, $(f \circ g)(x) = f(g(x))$
- only possible if output of g compatible with input of f, that is, $f : B \to C$ and $g : A \to B$
- in Haskell: `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- try "`:info (.)`" in GHCi

Examples

- `map (f . g) xs` – to every element of `xs`, first apply `g` and then `f`
- equivalent to `map f (map g xs)`

Interlude (function composition)

- in mathematics $f \circ g$ usually denotes applying f after g
- more precisely, $(f \circ g)(x) = f(g(x))$
- only possible if output of g compatible with input of f, that is, $f : B \to C$ and $g : A \to B$
- in Haskell: `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- try "`:info (.)`" in GHCi

Examples

- `map (f . g) xs` – to every element of `xs`, first apply `g` and then `f`
- equivalent to `map f (map g xs)`
- what are the results of `unwords . words` and `words . unwords`?

List Comprehensions – Generators

- in mathematics set comprehensions can be used to construct new sets from existing sets

List Comprehensions – Generators

- in mathematics set comprehensions can be used to construct new sets from existing sets
- e.g., $\{x^2 \mid x \in \{1, \ldots, 5\}\}$ produces $\{1, 4, 9, 16, 25\}$

List Comprehensions – Generators

- in mathematics set comprehensions can be used to construct new sets from existing sets
- e.g., $\{x^2 \mid x \in \{1, \ldots, 5\}\}$ produces $\{1, 4, 9, 16, 25\}$
- in Haskell: `[x^2 | x <- [1..5]]`

List Comprehensions – Generators

- in mathematics set comprehensions can be used to construct new sets from existing sets
- e.g., $\{x^2 \mid x \in \{1, \ldots, 5\}\}$ produces $\{1, 4, 9, 16, 25\}$
- in Haskell: `[x^2 | x <- [1..5]]`
- here, `x <- [1..5]` is called generator

List Comprehensions – Generators

- in mathematics set comprehensions can be used to construct new sets from existing sets
- e.g., $\{x^2 \mid x \in \{1, \ldots, 5\}\}$ produces $\{1, 4, 9, 16, 25\}$
- in Haskell: `[x^2 | x <- [1..5]]`
- here, `x <- [1..5]` is called generator
- there may be more than one generator, e.g., `[(x, y) | x <- xs, y <- xs]` (all pairs of elements from `xs`)

List Comprehensions – Generators

- in mathematics set comprehensions can be used to construct new sets from existing sets
- e.g., $\{x^2 \mid x \in \{1, \ldots, 5\}\}$ produces $\{1, 4, 9, 16, 25\}$
- in Haskell: `[x^2 | x <- [1..5]]`
- here, `x <- [1..5]` is called generator
- there may be more than one generator, e.g., `[(x, y) | x <- xs, y <- xs]` (all pairs of elements from `xs`)
- order is important: `[(x,y) | x <- [1,2], y <- ["a","b"]] = [(1,"a"),(1,"b"),(2,"a"),(2,"b")]`

List Comprehensions – Generators

- in mathematics set comprehensions can be used to construct new sets from existing sets
- e.g., $\{x^2 \mid x \in \{1, \ldots, 5\}\}$ produces $\{1, 4, 9, 16, 25\}$
- in Haskell: `[x^2 | x <- [1..5]]`
- here, `x <- [1..5]` is called generator
- there may be more than one generator, e.g., `[(x, y) | x <- xs, y <- xs]` (all pairs of elements from `xs`)
- order is important: `[(x,y) | x <- [1,2], y <- ["a","b"]] = [(1,"a"),(1,"b"),(2,"a"),(2,"b")]`

Examples

- `length xs = sum [1 | _ <- xs]`

List Comprehensions – Generators

- in mathematics set comprehensions can be used to construct new sets from existing sets
- e.g., $\{x^2 \mid x \in \{1, \ldots, 5\}\}$ produces $\{1, 4, 9, 16, 25\}$
- in Haskell: `[x^2 | x <- [1..5]]`
- here, `x <- [1..5]` is called generator
- there may be more than one generator, e.g., `[(x, y) | x <- xs, y <- xs]` (all pairs of elements from `xs`)
- order is important: `[(x,y) | x <- [1,2], y <- ["a","b"]] = [(1,"a"),(1,"b"),(2,"a"),(2,"b")]`

Examples

- `length xs = sum [1 | _ <- xs]`
- `firsts ps = [x | (x, _) <- ps]`

List Comprehensions – Generators

- in mathematics set comprehensions can be used to construct new sets from existing sets
- e.g., $\{x^2 \mid x \in \{1, \ldots, 5\}\}$ produces $\{1, 4, 9, 16, 25\}$
- in Haskell: `[x^2 | x <- [1..5]]`
- here, `x <- [1..5]` is called generator
- there may be more than one generator, e.g., `[(x, y) | x <- xs, y <- xs]` (all pairs of elements from `xs`)
- order is important: `[(x,y) | x <- [1,2], y <- ["a","b"]] = [(1,"a"),(1,"b"),(2,"a"),(2,"b")]`

Examples

- `length xs = sum [1 | _ <- xs]`
- `firsts ps = [x | (x, _) <- ps]`
- `concat xss = [x | xs <- xss, x <- xs]`

List Comprehensions – Guards

- filter values before generating result

List Comprehensions – Guards

- filter values before generating result
- e.g., $\{x^2 \mid x \in \mathbb{N}, x > 5\}$

List Comprehensions – Guards

- filter values before generating result
- e.g., $\{x^2 \mid x \in \mathbb{N}, x > 5\}$
- in Haskell: `[x^2 | x <- xs, x > 5]`; square every number in `xs` that is greater than 5

List Comprehensions – Guards

- filter values before generating result
- e.g., $\{x^2 \mid x \in \mathbb{N}, x > 5\}$
- in Haskell: `[x^2 | x <- xs, x > 5]`; square every number in `xs` that is greater than 5

Examples

- `[x | x <- [1..10], even x]`

List Comprehensions – Guards

- filter values before generating result
- e.g., $\{x^2 \mid x \in \mathbb{N}, x > 5\}$
- in Haskell: `[x^2 | x <- xs, x > 5]`; square every number in `xs` that is greater than 5

Examples

- `[x | x <- [1..10], even x]`
- `findAll k t = [v | (k', v) <- t, k == k']`

List Comprehensions – Guards

- filter values before generating result
- e.g., $\{x^2 \mid x \in \mathbb{N}, x > 5\}$
- in Haskell: `[x^2 | x <- xs, x > 5]`; square every number in `xs` that is greater than 5

Examples

- `[x | x <- [1..10], even x]`
- `findAll k t = [v | (k', v) <- t, k == k']`
- `factors n = [x | x <- [1..n], n`mod`x == 0]`

List Comprehensions – Guards

- filter values before generating result
- e.g., $\{x^2 \mid x \in \mathbb{N}, x > 5\}$
- in Haskell: `[x^2 | x <- xs, x > 5]`; square every number in `xs` that is greater than 5

Examples

- `[x | x <- [1..10], even x]`
- `findAll k t = [v | (k', v) <- t, k == k']`
- `factors n = [x | x <- [1..n], n`mod`x == 0]`
- `primes = [n | n <- [1..], factors n == [1,n]]`

Outline

Basic Concepts

- functions may be defined in terms of other functions

```
factorial :: Int -> Int
factorial n = product [1..n]
```

Basic Concepts

- functions may be defined in terms of other functions

```
factorial :: Int -> Int
factorial n = product [1..n]
```

- or in terms of themselves (that is, recursively)

```
factorial n
  | n <= 1    = 1
  | otherwise = n * factorial (n - 1)
```

Basic Concepts

- functions may be defined in terms of other functions

```
factorial :: Int -> Int
factorial n = product [1..n]
```

- or in terms of themselves (that is, recursively)

```
factorial n
  | n <= 1    = 1
  | otherwise = n * factorial (n - 1)
```

- **note:** `factorial` does not loop forever, since at some point its argument will be 1 or smaller (its termination condition)

Basic Concepts

- functions may be defined in terms of other functions

```
factorial :: Int -> Int
factorial n = product [1..n]
```

- or in terms of themselves (that is, recursively)

```
factorial n
  | n <= 1    = 1
  | otherwise = n * factorial (n - 1)
```

- **note:** `factorial` does not loop forever, since at some point its argument will be `1` or smaller (its termination condition)
- recipe for defining recursive functions

Basic Concepts

- functions may be defined in terms of other functions

```
factorial :: Int -> Int
factorial n = product [1..n]
```

- or in terms of themselves (that is, recursively)

```
factorial n
  | n <= 1    = 1
  | otherwise = n * factorial (n - 1)
```

- **note:** `factorial` does not loop forever, since at some point its argument will be 1 or smaller (its termination condition)
- recipe for defining recursive functions
  1. define type (e.g., `product :: [Int] -> Int`)

Basic Concepts

- functions may be defined in terms of other functions

```
factorial :: Int -> Int
factorial n = product [1..n]
```

- or in terms of themselves (that is, recursively)

```
factorial n
  | n <= 1    = 1
  | otherwise = n * factorial (n - 1)
```

- **note:** `factorial` does not loop forever, since at some point its argument will be `1` or smaller (its termination condition)
- recipe for defining recursive functions
  1. define type (e.g., `product :: [Int] -> Int`)
  2. enumerate cases (e.g., `[]` and `x:xs`)

Basic Concepts

- functions may be defined in terms of other functions

```
factorial :: Int -> Int
factorial n = product [1..n]
```

- or in terms of themselves (that is, recursively)

```
factorial n
  | n <= 1    = 1
  | otherwise = n * factorial (n - 1)
```

- **note:** `factorial` does not loop forever, since at some point its argument will be `1` or smaller (its termination condition)
- recipe for defining recursive functions
  1. define type (e.g., `product :: [Int] -> Int`)
  2. enumerate cases (e.g., `[]` and `x:xs`)
  3. define simple cases (e.g., `product [] = 1`)

Basic Concepts

- functions may be defined in terms of other functions

```
factorial :: Int -> Int
factorial n = product [1..n]
```

- or in terms of themselves (that is, recursively)

```
factorial n
  | n <= 1    = 1
  | otherwise = n * factorial (n - 1)
```

- **note:** `factorial` does not loop forever, since at some point its argument will be `1` or smaller (its termination condition)
- recipe for defining recursive functions
  1. define type (e.g., `product :: [Int] -> Int`)
  2. enumerate cases (e.g., `[]` and `x:xs`)
  3. define simple cases (e.g., `product [] = 1`)
  4. define other cases (e.g., `product (x:xs) = x * product xs`)

Basic Concepts

- functions may be defined in terms of other functions

```
factorial :: Int -> Int
factorial n = product [1..n]
```

- or in terms of themselves (that is, recursively)

```
factorial n
  | n <= 1    = 1
  | otherwise = n * factorial (n - 1)
```

- **note:** `factorial` does not loop forever, since at some point its argument will be `1` or smaller (its termination condition)
- recipe for defining recursive functions
  1. define type (e.g., `product :: [Int] -> Int`)
  2. enumerate cases (e.g., `[]` and `x:xs`)
  3. define simple cases (e.g., `product [] = 1`)
  4. define other cases (e.g., `product (x:xs) = x * product xs`)
  5. generalize and simplify (e.g., `product :: Num a => [a] -> a` and `product = foldr (*) 1`)

Example (drop)

- define type: `drop :: Int -> [a] -> [a]`

Example (drop)

- define type: `drop :: Int -> [a] -> [a]`
- enumerate cases:

```
drop 0 []     =
drop 0 (x:xs) =
drop n []     =
drop n (x:xs) =
```

Example (drop)

- define type: `drop :: Int -> [a] -> [a]`
- enumerate cases:

```
drop 0 []     =
drop 0 (x:xs) =
drop n []     =
drop n (x:xs) =
```

- define simple cases:

```
drop 0 []     =
```

Example (drop)

- define type: `drop :: Int -> [a] -> [a]`
- enumerate cases:

```
drop 0 []     =
drop 0 (x:xs) =
drop n []     =
drop n (x:xs) =
```

- define simple cases:

```
drop 0 []     = []
```

Example (drop)

- define type: `drop :: Int -> [a] -> [a]`
- enumerate cases:

```
drop 0 []     =
drop 0 (x:xs) =
drop n []     =
drop n (x:xs) =
```

- define simple cases:

```
drop 0 []     = []
drop 0 (x:xs) =
```

Example (drop)

- define type: `drop :: Int -> [a] -> [a]`
- enumerate cases:

```
drop 0 []     =
drop 0 (x:xs) =
drop n []     =
drop n (x:xs) =
```

- define simple cases:

```
drop 0 []     = []
drop 0 (x:xs) = x : xs
```

Example (drop)

- define type: `drop :: Int -> [a] -> [a]`
- enumerate cases:

```
drop 0 []     =
drop 0 (x:xs) =
drop n []     =
drop n (x:xs) =
```

- define simple cases:

```
drop 0 []     = []
drop 0 (x:xs) = x : xs
drop n []     =
```

Example (drop)

- define type: `drop :: Int -> [a] -> [a]`
- enumerate cases:

```
drop 0 []     =
drop 0 (x:xs) =
drop n []     =
drop n (x:xs) =
```

- define simple cases:

```
drop 0 []     = []
drop 0 (x:xs) = x : xs
drop n []     = []
```

Example (drop)

- define type: `drop :: Int -> [a] -> [a]`
- enumerate cases:

```
drop 0 []     =
drop 0 (x:xs) =
drop n []     =
drop n (x:xs) =
```

- define simple cases:

```
drop 0 []     = []
drop 0 (x:xs) = x : xs
drop n []     = []
```

- define other cases:

```
drop n (x:xs) =
```

Example (drop)

- define type: `drop :: Int -> [a] -> [a]`
- enumerate cases:
```
drop 0 []     =
drop 0 (x:xs) =
drop n []     =
drop n (x:xs) =
```
- define simple cases:
```
drop 0 []     = []
drop 0 (x:xs) = x : xs
drop n []     = []
```
- define other cases:
```
drop n (x:xs) = drop (n - 1) xs
```

Example (drop)

- define type: `drop :: Int -> [a] -> [a]`
- enumerate cases:

```
drop 0 []     =
drop 0 (x:xs) =
drop n []     =
drop n (x:xs) =
```

- define simple cases:

```
drop 0 []     = []
drop 0 (x:xs) = x : xs
drop n []     = []
```

- define other cases:

```
drop n (x:xs) = drop (n - 1) xs
```

- generalize and simplify:

```
drop :: Integer -> [a] -> [a]
drop n xs | n <= 0  =  xs
drop _ []           =  []
drop n (_:xs)       =  drop (n - 1) xs
```

Example (init)

- define type: `init :: [a] -> [a]`

Example (init)

- define type: `init :: [a] -> [a]`
- enumerate cases:
  `init (x:xs) =`

Example (init)

- define type: `init :: [a] -> [a]`
- enumerate cases:
  `init (x:xs) =`
- define simple cases:
  `init (x:xs) | null xs =`

Example (init)

- define type: `init :: [a] -> [a]`
- enumerate cases:
  `init (x:xs) =`
- define simple cases:
  `init (x:xs) | null xs = []`

Example (init)

- define type: `init :: [a] -> [a]`
- enumerate cases:
  `init (x:xs) =`
- define simple cases:
  `init (x:xs) | null xs = []`
- define other cases:
  `| otherwise =`

Example (init)

- define type: `init :: [a] -> [a]`
- enumerate cases:
  `init (x:xs) =`
- define simple cases:
  `init (x:xs) | null xs = []`
- define other cases:
  `| otherwise = x : init xs`

Example (init)

- define type: `init :: [a] -> [a]`
- enumerate cases:
  ```
  init (x:xs) =
  ```
- define simple cases:
  ```
  init (x:xs) | null xs = []
  ```
- define other cases:
  ```
              | otherwise = x : init xs
  ```
- generalize and simplify:
  ```
  init :: [a] -> [a]
  init [_]    = []
  init (x:xs) = x : init xs
  ```

Outline

Printing a Calendar

- given a month and a year, print the corresponding calendar
- separate construction phase (computation of days, leap year, ... in file `Calendar.hs`) from printing
- we concentrate on printing, assuming machinery for construction

Printing a Calendar

- given a month and a year, print the corresponding calendar
- separate construction phase (computation of days, leap year, ... in file `Calendar.hs`) from printing
- we concentrate on printing, assuming machinery for construction

Example (October 2022)

```
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

Helper Functions: Combining two Lists via a Function

- `myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
- `myZipWith f` $[x_1,\ldots,x_m]$ $[y_1,\ldots,y_n] = [x_1 \grave{} f \grave{} y_1,\ldots,x_{\min\{m,n\}} \grave{} f \grave{} y_{\min\{m,n\}}]$
- specialization `myZip :: [a] -> [b] -> [(a, b)]`

  `myZip = myZipWith (,)`

Helper Functions: Combining two Lists via a Function

- `myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
- `myZipWith f` $[x_1,\ldots,x_m]$ $[y_1,\ldots,y_n] = [x_1 `f` y_1,\ldots,x_{\min\{m,n\}} `f` y_{\min\{m,n\}}]$
- specialization `myZip :: [a] -> [b] -> [(a, b)]`

  `myZip = myZipWith (,)`

Example

- `myZip [1,2,3] ['a','b'] = [(1,'a'),(2,'b')]`
- `myZipWith (*) [1,2] [3,4,5] = [1*3,2*4] = [3,8]`
- `myZipWith drop [1,0] ["a","b"] = [drop 1 "a",drop 0 "b"] = ["","b"]`

Helper Functions: Combining two Lists via a Function

- `myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
- `myZipWith f` $[x_1,\ldots,x_m]$ $[y_1,\ldots,y_n] = [x_1 `f` y_1,\ldots,x_{\min\{m,n\}} `f` y_{\min\{m,n\}}]$
- specialization `myZip :: [a] -> [b] -> [(a, b)]`

  `myZip = myZipWith (,)`

Example

- `myZip [1,2,3] ['a','b'] = [(1,'a'),(2,'b')]`
- `myZipWith (*) [1,2] [3,4,5] = [1*3,2*4] = [3,8]`
- `myZipWith drop [1,0] ["a","b"] = [drop 1 "a",drop 0 "b"] = ["","b"]`

Helper Functions (cont'd): Right Folding without a base

- `myFoldr1` – special version of `myFoldr`, without base value (does not work on empty list)

```
myFoldr1 :: (a -> a -> a) -> [a] -> a
myFoldr1 f l =
    case l of
        [x]  -> x
        x:xs -> f x (myFoldr1 f xs)
```

Helper Functions (cont'd): Right Justifying a String & Grouping a List

- `myRjustify` – right-justify given text inside box of given width

```
myRjustify :: Int -> String -> String
myRjustify n l =
    if length l < n then replicate (n-(length l)) ' ' ++ l
    else error ("list of length " ++ show (length l) ++ "does not fit in a box of width " ++ show n)
```

Helper Functions (cont'd): Right Justifying a String & Grouping a List

- `myRjustify` – right-justify given text inside box of given width

```
myRjustify :: Int -> String -> String
myRjustify n l =
    if length l < n then replicate (n-(length l)) ' ' ++ l
    else error ("list of length " ++ show (length l) ++ "does not fit in a box of width " ++ show n)
```

- `myGroupsOfSize` – split list into sublists of given length

```
myGroupsOfSize :: Int -> [a] -> [[a]]
myGroupsOfSize n l =
    let (xs, ys) = mySplitAt n l
    in if null xs then [] else xs : myGroupsOfSize n ys
```

Helper Functions (cont'd): Right Justifying a String & Grouping a List

- `myRjustify` – right-justify given text inside box of given width

```
myRjustify :: Int -> String -> String
myRjustify n l =
    if length l < n then replicate (n-(length l)) ' ' ++ l
    else error ("list of length " ++ show (length l) ++ "does not fit in a box of width " ++ show n)
```

- `myGroupsOfSize` – split list into sublists of given length

```
myGroupsOfSize :: Int -> [a] -> [[a]]
myGroupsOfSize n l =
    let (xs, ys) = mySplitAt n l
    in if null xs then [] else xs : myGroupsOfSize n ys
```

- `mySplitAt :: Int -> [a] -> ([a], [a])` – split list at given position

The Picture Analogon

pictures:

- atomic part: pixel
- height and width
- white pixel

strings:

- atomic part: character
- number of rows and columns
- blank character

The Picture Analogon

pictures:

- atomic part: pixel
- height and width
- white pixel

strings:

- atomic part: character
- number of rows and columns
- blank character

The Picture Analogon

pictures:

- atomic part: pixel
- height and width
- white pixel

strings:

- atomic part: character
- number of rows and columns
- blank character

The Picture Analogon

pictures:

- atomic part: pixel
- height and width
- white pixel

strings:

- atomic part: character
- number of rows and columns
- blank character

The Picture Analogon

pictures:

- atomic part: pixel
- height and width
- white pixel

strings:

- atomic part: character
- number of rows and columns
- blank character

Auxiliary Types

```
type Height  = Int
type Width   = Int
type Picture = (Height, Width, [[Char]])
```

The Picture Analogon

pictures:

- atomic part: pixel
- height and width
- white pixel

strings:

- atomic part: character
- number of rows and columns
- blank character

Auxiliary Types

```
type Height  = Int
type Width   = Int
type Picture = (Height, Width, [[Char]])
```

- consider `(h, w, rs)`

The Picture Analogon

pictures:

- atomic part: pixel
- height and width
- white pixel

strings:

- atomic part: character
- number of rows and columns
- blank character

Auxiliary Types

```
type Height  = Int
type Width   = Int
type Picture = (Height, Width, [[Char]])
```

- consider `(h, w, rs)`
- `rs :: [[Char]]` – “list of rows”

The Picture Analogon

pictures:

- atomic part: pixel
- height and width
- white pixel

strings:

- atomic part: character
- number of rows and columns
- blank character

Auxiliary Types

```
type Height  = Int
type Width   = Int
type Picture = (Height, Width, [[Char]])
```

- consider (h, w, rs)
- rs :: [[Char]] – “list of rows”
- invariant 1: length of rs is height h

The Picture Analogon

pictures:

- atomic part: pixel
- height and width
- white pixel

strings:

- atomic part: character
- number of rows and columns
- blank character

Auxiliary Types

```
type Height  = Int
type Width   = Int
type Picture = (Height, Width, [[Char]])
```

- consider (h, w, rs)
- rs :: [[Char]] – “list of rows”
- invariant 1: length of rs is height h
- invariant 2: all rows (that is, lists in rs) have length w

The Picture Analogon

pictures:

- atomic part: pixel
- height and width
- white pixel

strings:

- atomic part: character
- number of rows and columns
- blank character

Auxiliary Types

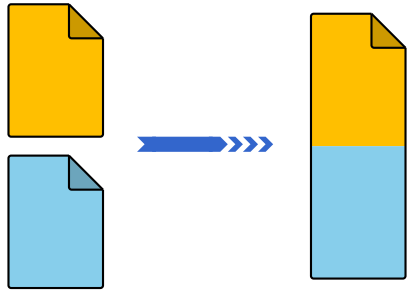
```
type Height  = Int
type Width   = Int
type Picture = (Height, Width, [[Char]])
```

- consider `(h, w, rs)`
- `rs :: [[Char]]` – "list of rows"
- invariant 1: length of `rs` is height `h`
- invariant 2: all rows (that is, lists in `rs`) have length `w`

Showing Pictures

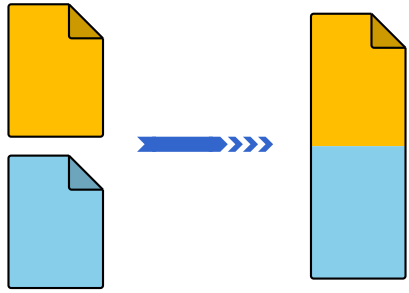
transform a `Picture` into a `String` and print it out on the standard IO considering escape characters

```
printPic (h, w, css) = putStr (unlines css)
```



above

```
above :: Picture -> Picture -> Picture
above p q =
    case p of
        (h1,w1,l1) -> case q of
                        (h2,w2,l2) -> if w1 == w2 then (h1+h2, w1, l1 ++ l2) else error "different widths"
```



above

```
above :: Picture -> Picture -> Picture
above p q =
    case p of
        (h1,w1,l1) -> case q of
                          (h2,w2,l2) -> if w1 == w2 then (h1+h2, w1, l1 ++ l2) else error "different widths"
```

Stacking Several Pictures Above Each Other

```
stack :: [Picture] -> Picture
stack = myFoldr1 above
```

beside

```
beside :: Picture -> Picture -> Picture
beside p q =
    case p of
        (h1,w1,l1) -> case q of
                        (h2,w2,l2) -> if h1 == h2 then (h1, w1+w2, myZipWith (++) l1 l2)
                                      else error "different heights"
```

beside

```
beside :: Picture -> Picture -> Picture
beside p q =
    case p of
        (h1,w1,l1) -> case q of
                          (h2,w2,l2) -> if h1 == h2 then (h1, w1+w2, myZipWith (++) l1 l2)
                                        else error "different heights"
```

Spreading Several Pictures Beside Each Other

```
spread :: [Picture] -> Picture
spread = myFoldr1 beside
```

beside

```
beside :: Picture -> Picture -> Picture
beside p q =
    case p of
        (h1,w1,l1) -> case q of
                        (h2,w2,l2) -> if h1 == h2 then (h1, w1+w2, myZipWith (++) l1 l2)
                                      else error "different heights"
```

Spreading Several Pictures Beside Each Other

```
spread :: [Picture] -> Picture
spread = myFoldr1 beside
```

Tiling Several Pictures

```
tile :: [[Picture]] -> Picture
tile l = stack (myMap spread l)
```

Creating Pictures

- single‘pixels’

```
pixel :: Char -> Picture
pixel c = (1, 1, [[c]])
```

Creating Pictures

- single‘pixels’

```
pixel :: Char -> Picture
pixel c = (1, 1, [[c]])
```

- rows

```
row :: String -> Picture
row r = (1, length r, [r])
```

Creating Pictures

- single‘pixels’

```
pixel :: Char -> Picture
pixel c = (1, 1, [[c]])
```

- rows

```
row :: String -> Picture
row r = (1, length r, [r])
```

- blank

```
blank :: Int -> Int -> Picture
blank h w = (h, w, replicate h (replicate w ' '))
```

Creating Pictures

- single‘pixels’

```
pixel :: Char -> Picture
pixel c = (1, 1, [[c]])
```

- rows

```
row :: String -> Picture
row r = (1, length r, [r])
```

- blank

```
blank :: Int -> Int -> Picture
blank h w = (h, w, replicate h (replicate w ' '))
```

Remark

`replicate :: Int -> a -> [a]` – replicates single element given number of times

Constructing a Month

- assume function `monthInfo :: Int -> Int -> (Int, Int)`, returning the first weekday of the month together with the number of days for the month
- where days are 0 (Sunday), 1 (Monday), . . .
- e.g., `monthInfo 10 2022` = `(6, 31)`, meaning that the first weekday of October 2022 is a Saturday and the month has 31 days

```
daysOfMonth :: Month -> Year -> [Picture]
daysOfMonth m y =
    let (d, t) = monthInfo m y
        pic n  = if 1 <= n && n <= t then show n else ""
    in myMap row (myMap (myRjustify 3) (myMap pic [1-d..42-d]))

month :: Month -> Year -> Picture
month m y = tile (myGroupsOfSize 7 (daysOfMonth m y))
```

Printing a Month

- print result of `month m y`

```
printMonth :: Month -> Year -> IO()
printMonth m y =
    let weekdays = row " Su Mo Tu We Th Fr Sa"
    in printPic (above weekdays (month m y))
```

Thanks! & Questions?