Lists

Tjark Weber
tjark.weber@it.uu.se

Today and Tomorrow

- Various Forms of Recursion
- Lists: Definition, Examples
- Pattern Matching and Recursion for Lists
- Documenting variants
- List Operations in the Haskell Prelude
- Lists: Remarks and Syntactic Sugar

Lists:

Definition, Examples

Tuples Have a Fixed Number of Components

When you use tuples in your program, you have to choose the number of components *at the time you write your code*.

For instance, you can write a function that computes the maximum of two integers

```
max2 :: (Integer,Integer)->Integer
max2 (a, b) = if a > b then a else b
```

and another function that computes the maximum of three integers

```
max3 :: (Integer,Integer,Integer)->Integer
max3 (a, b, c) = ...
```

and so on, but with tuples you cannot write *one* function that computes the maximum of an arbitrary number of integers.

Lists

The **list type** is a data type that implements an ordered collection of values, where the same value may occur more than once.

A **list** is a computer representation of the mathematical concept of a (finite or infinite) sequence.

Lists: Real Life Examples

We use lists in many situations. For instance:

- Shopping lists
- Checklists
- To-do lists
- Wish lists
- Lists of lists
- ...

Lists in Haskell: Examples

```
[18, 12, -5, 12, 10]

[2.0, 5.3 - 1.2, 3.7, -1E5]

["Bread", "Butter", "Milk"]

[(1,"A"), (2,"B")]

[abs, \x -> x+1]

[[1], [2, 3]]

[]
```

Lists in Haskell

If `e1`, `e2`, `...`, `eN` are expressions of type `T`, then

```
[e1, e2, ..., eN]
```

is an expression of type `[T]`.

Note that lists in Haskell must be **homogeneous**, i.e., all elements must have the same type. (Other programming languages may or may not have this requirement.)

For instance, this is *not* a valid Haskell expression:

```
[1, "not homogeneous"]
```

Lists: Properties

Lists contain elements. Therefore, the list type is sometimes called a **container** data type.

Properties:

- *Variability*: the number of elements in a list is arbitrary and only determined at run time.
- *Multiplicity*: an element may appear several times in a list.
- *Linearity*: the internal structure of a list is linear.
- *Extensionality*: two lists are equal only if they contain equal elements (in the same order).

(There are other container data types, e.g., sets, queues, . . . , that have different properties.)

List Expressions

Lists (just like tuples) can be constructed from expressions that need to be evaluated.

```
[18, 3+9, 5-7, length "foo", 10]
  ⟶ [18, 12, 5-7, length "foo", 10]
  ⟶ [18, 12, -2, length "foo", 10]
  ⟶ [18, 12, -2, 3, 10]
```

Lists in Haskell: Inductive Definition

In Haskell, lists are constructed according to the following inductive definition.

- `[]` is a list, called the **empty list**.
- If `x` is an element (called the **head**) and `xs` is a list (called the **tail**), then `x : xs` is a list.

Nothing else is a list, i.e., all lists are constructed according to these two rules.

Lists in Haskell: Examples

```
Prelude> []
[]

Prelude> 1 : []
[1]

Prelude> 2 : 1 : []
[2,1]

Prelude> 3 : 2 : 1 : []
[3,2,1]
```

Lists: Aggregated vs. Constructed Form

The **aggregated form**

```
[e1, ..., eN]
```

is syntactic sugar for the **constructed form**

```
e1 : (... : (eN : []))
```

Both represent the same expression. For instance,

```
Prelude> [1,2,3] == 1:2:3:[]
True
```

The Type of []

What is the type of the empty list, `[]`?

Depending on the context, `[]` could be a list of integers

```
Prelude> 1:[]
[1]
```

or a list of strings

```
Prelude> ["foo"] == []
False
```

or a list with elements of any other type.

We say that `[]` is a **polymorphic** value of type `[a]`. Here, `a` is a **type variable**. Type variables stand for arbitrary types.

About :

The identifier : behaves like a binary operator that is

- right-associative: e.g., `1 : 2 : []` is `1 : (2 : [])`
- of the type `a -> [a] -> [a]`
- has precedence 5 (lower than +, but higher than ==).

```
Prelude> :i (:)
data [] a = ... | a : [a]  -- Defined in `GHC.Types'
infixr 5 :

Prelude> :t (:)
(:) :: a -> [a] -> [a]
```

Pattern Matching and Recursion for Lists

Pattern Matching for Lists: Aggregated Form

Remember that data type "skeletons" can be used as patterns. Suppose `p1`, ..., `pN` are patterns for the same type `T`. Then

```
[p1, ..., pN]
```

is a pattern for type `[T]`. It matches a list of length N if `p1`, ..., `pN` match the corresponding list elements.

Pattern Matching for Lists: Examples

```
Prelude> let [x,y,z] = [1,2,3]
Prelude> x
1
Prelude> y
2
Prelude> z
3

Prelude> let [_,y,3] = [1,2,3]
Prelude> y
2

Prelude> let [x,y,z] = [1,2]
Prelude> x
*** Exception: <interactive>:24:5-19: Irrefutable pattern
    failed for pattern [x, y, z]
```

Pattern Matching for Lists: Constructed Form

Also the constructed form can be used for list patterns. Suppose `p` is a pattern for type `T`, and `ps` is a pattern for type `[T]`. Then

```
p : ps
```

is a pattern for type `[T]`. It matches a list if `p` matches the head and `ps` matches the tail of the list. (Note that it never matches the empty list.)

Note: `:` is not really a function in Haskell (function calls are not allowed in patterns!), but a (data type) **constructor**. The list data type has two constructors, namely `[]` and `:`. These *can* be used in patterns.

Pattern Matching for Lists: Examples

```
Prelude> let x:xs = [1,2,3]
Prelude> x
1
Prelude> xs
[2,3]

Prelude> let _:x:xs = [1,2,3]
Prelude> x
2
Prelude> xs
[3]

Prelude> let x:xs = []
Prelude> x
*** Exception: <interactive>:33:5-13: Irrefutable pattern
    failed for pattern x : xs
```

Lists vs. Tuples

Tuples: example `(3, 8.0, 5>8)`

- Fixed number of components
- Heterogeneous (components may be of different types)
- Direct access to all components (via pattern matching)

Lists: example `[3, 8, 5]`

- Arbitrary length
- Homogenous (elements of the same type)
- Sequential access to the elements (pattern matching can only access the head element directly)

Data Abstraction Once Again

By now, we have seen *all* primitive operations for lists.

- Lists are constructed using `[]` and `:`
- Lists can be taken apart using `[]` and `:` patterns.

All other functions that construct or operate on lists are implemented in terms of these primitives.

This is (data) abstraction once again: we don't need to know how lists are actually represented in memory.

List Functions: null, head, tail

Using pattern matching, we can write simple functions for lists. For instance,

```
null []  = True
null _   = False

head (x:_) = x
-- head [] = error "empty list"

tail (_:xs) = xs
-- tail [] = error "empty list"
```

Recursive List Functions: `length`

Using pattern matching and recursion, we can write further functions for lists. For instance,

```
length []     = 0
length (_:xs) = 1 + length xs
```

Evaluation of `length`

Evaluation of `length` follows the usual evaluation rules for function application (using pattern matching for lists). For instance,

```
length ["hello", "world" ++ "!"]
  ⟶ 1 + length xs  (where xs = ["world" ++ "!"])
  ⟶ 1 + length ["world" ++ "!"]  (where xs = ...)
  ⟶ [1 + (1 + length xs') (where xs' = [])]  (where xs = ...)
  ⟶ [1 + (1 + length []) (where xs' = [])]  (where xs = ...)
  ⟶ 1 + (1 + 0)  (where ...)
  ⟶ 1 + 1  (where ...)
  ⟶ 2
```

Note that since `length` is using an underscore pattern to match individual list elements, these are *not* evaluated.

Structural Recursion

Many recursive functions for lists follow the same recursion scheme:

```
-- f xs  (variant: length xs)
-- base case
f [] = ...
-- recursion over the tail
f (x:xs) = ... (f xs) ...
```

This is known as **structural recursion** (i.e., recursion over the structure of lists).

Note the similarity to simple recursion for integers.

Finding the Last Element

`last xs` returns the last element of a non-empty list `xs`: e.g.,
`last [1,2,3]` returns 3

Finding the Last Element

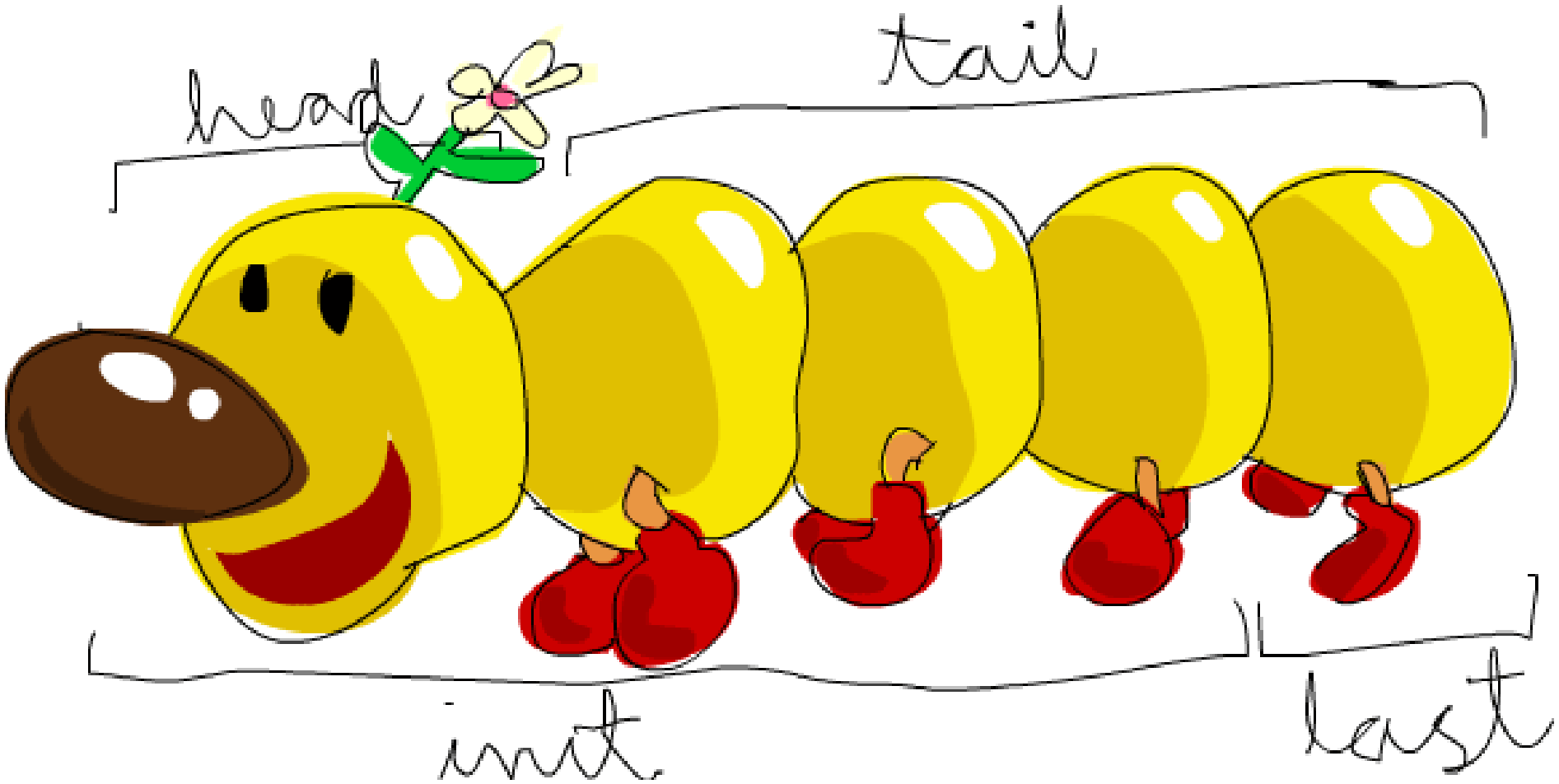
`last xs` returns the last element of a non-empty list `xs`: e.g., `last [1,2,3]` returns 3

```
last [x]    = x
last (_:xs) = last xs
-- last [] = error "empty list"
```

Note that the number of operations required to evaluate `last xs` depends on the length of `xs`. (More precisely, if the list `xs` has length $n > 0$, evaluating `last xs` performs $n - 1$ recursive calls.)

Only the *head element* of a list can be accessed easily!

head, tail, init, last



Source: http://learnyouahaskell.com/starting-out

Concatenating Two Lists

`xs ++ ys` returns the concatenation of `xs` and `ys`: e.g.,
`[1,2] ++ [3,4,5]` returns `[1,2,3,4,5]`

Concatenating Two Lists

`xs ++ ys` returns the concatenation of `xs` and `ys`: e.g.,
`[1,2] ++ [3,4,5]` returns `[1,2,3,4,5]`

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Note that if the list `xs` has length n, evaluating `xs ++ ys` performs n recursive calls.

List Reversal

`reverse xs` returns a list consisting of `xs`'s elements in reverse order: e.g., `reverse [1,2,3]` returns `[3,2,1]`

List Reversal

`reverse xs` returns a list consisting of `xs`'s elements in reverse order: e.g., `reverse [1,2,3]` returns `[3,2,1]`

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Note that evaluation of `reverse xs` requires a number of recursive calls to ++ (exactly how many calls depends on the length of `xs`). Later, we will see a better way to implement `reverse`.

List Membership

`elem e xs` returns `True` if (and only if) `e` is an element of `xs`: e.g., `elem 2 [1,3,2]` returns `True`, `elem 2 [1,3,0]` returns `False`

List Membership

`elem e xs` returns `True` if (and only if) `e` is an element of `xs`: e.g., `elem 2 [1,3,2]` returns `True`, `elem 2 [1,3,0]` returns `False`

```
elem e []       = False
elem e (x:xs) = e==x || elem e xs
```

Note that we can apply `elem` only to lists whose elements can be compared with ==. Applying `elem` to, e.g., a list of functions results in a type error.

Specification of Functions: Variants

Variants are a formal explanation why recursive functions terminate. For recursive functions, we require that a variant is given as part of the function specification.

```
{- pairwiseSum xs
   RETURNS: a list consisting of the sums of adjacent elements of
        xs
   EXAMPLE: pairwiseSum [1,-2,3] = [-1,1]
-}
pairwiseSum :: Num a => [a] -> [a]
-- VARIANT: length xs
pairwiseSum (x:y:xs) = (x+y) : pairwiseSum (y:xs)
pairwiseSum _ = []
```

Specification of Functions: No Variants

For functions defined entirely in terms of structural recursion, variants may be omitted.

```
{- rev xs
   RETURNS: a list consisting of xs's elements in reverse order
   EXAMPLE: rev [1,2,3] = [3,2,1]
-}
rev :: [a] -> [a]
rev [] = []
rev (x : xs) = rev xs ++ [x]
```

Lists Operations in the Haskell Prelude

Lists Operations in the Haskell Prelude

The Haskell Prelude already provides many of the basic list operations that we have seen in this lecture. You don't need to declare them yourself!

Function	Type	Function	Type
`null`	`[a]->Bool`	`reverse`	`[a]->[a]`
`head`	`[a]->a`	`++`	`[a]->[a]->[a]`
`tail`	`[a]->[a]`	`elem`	`a->[a]->Bool` [1]
`init`	`[a]->[a]`	`take`	`Int->[a]->[a]`
`last`	`[a]->a`	`drop`	`Int->[a]->[a]`
`length`	`[a]->Int`	`replicate`	`Int->a->[a]`

`null`, `head` and `tail` are actually seldomly used. Pattern matching usually results in more readable and concise code.

[1] Where a is any type that admits equality (in particular, not a function type).

Equality on Lists

In Haskell, equality on lists is structural, i.e., two lists are equal only if they contain equal elements (in the same order).

```
Prelude> [1] == [1]
```

Equality on Lists

In Haskell, equality on lists is structural, i.e., two lists are equal only if they contain equal elements (in the same order).

```
Prelude> [1] == [1]
True

Prelude> [1,2] == [2,1]
```

Equality on Lists

In Haskell, equality on lists is structural, i.e., two lists are equal only if they contain equal elements (in the same order).

```
Prelude> [1] == [1]
True

Prelude> [1,2] == [2,1]
False

Prelude> [1] == [1,1]
```

Equality on Lists

In Haskell, equality on lists is structural, i.e., two lists are equal only if they contain equal elements (in the same order).

```
Prelude> [1] == [1]
True

Prelude> [1,2] == [2,1]
False

Prelude> [1] == [1,1]
False
```

Equality on Lists (cont.)

Lists can be tested for equality only if their elements can be tested for equality. For instance, lists of functions cannot be tested for equality:

```
Prelude> [(&&)] == [(||)]

<interactive>:2:8:
    No instance for (Eq (Bool -> Bool -> Bool))
      arising from a use of `=='
  Possible fix:
    add an instance declaration for (Eq (Bool -> Bool -> Bool))
  In the expression: [(&&)] == [(||)]
  ...
```

Lists:

Remarks, Syntactic Sugar

Aggregated vs. Constructed Form: Usage

Code should be readable and concise.

The aggregated form of lists `[e1, ..., eN]` is mostly used for ...

- lists given explicitly,
- results (when displayed by GHCi).

The constructed form `x:xs` is mostly used for ...

- decomposition of a list by pattern matching,
- construction of a list from its head and tail.

Lists of Lists

Elements of a list may themselves be lists. For instance, here is a list whose elements are of type `[Integer]`:

```
[[42], [], [1,2,3]]
```

Thus, the entire list is of type `[[Integer]]`

We can apply list operations to lists of lists. For instance, let's put two lists of lists together:

```
Prelude> [[42], [], [1,2,3]] ++ [[1], [1,2,3]]
[[42],[],[1,2,3],[1],[1,2,3]]
```

Lists of Lists (cont.)

This is an empty list:

```
[]
```

This is a list that contains one empty list:

```
[[]]
```

This is a list that contains three empty lists:

```
[[],[],[]]
```

Strings are Lists of Characters

In Haskell, a string is just syntactic sugar for a list of characters.

```
Prelude> :t "hello"
"hello" :: [Char]

Prelude> "hello" == ['h','e','l','l','o']
True
```

Thus, all functions that can be applied to lists (e.g., `head`, `tail`, ...) can also be applied to strings.

Ranges

Ranges are syntactic sugar to write lists whose elements can be enumerated:

```
Prelude> [10..20]
[10,11,12,13,14,15,16,17,18,19,20]

Prelude> ['a'..'f']
"abcdef"
```

A range may also specify the second list element:

```
Prelude> [10,12..20]
[10,12,14,16,18,20]
```

Ranges (cont.)

Watch out!

```
Prelude> [20..10]
[]

Prelude> [20,19..10]
[20,19,18,17,16,15,14,13,12,11,10]
```

List Comprehension

You're probably familiar with set comprehension: e.g.,

$$\{x^2 \mid x \in \mathbb{N}, x \leq 10\}$$

Haskell has **list comprehension**: syntactic sugar to

1. select all elements from a given list that satisfy some *condition*, and
2. apply some *function* to these elements.

List Comprehension: Examples

For instance,

```
Prelude> [ x*x | x <- [1..100], x <= 10 ]
[1,4,9,16,25,36,49,64,81,100]

Prelude> [ length x | x <- ["Program","Design","and","Data",
     "Structures"], head x == 'D' ]
[6,4]

Prelude> [ c | c <- "IdontLIKEFROGS", c `elem` ['A'..'Z'] ]
"ILIKEFROGS"
```

It is even possible to select elements from several lists:

```
Prelude> [ (x,y) | x <- [1..3], y <- [2..4], x<y ]
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```