
CSc 372

Comparative Programming Languages

6 : Haskell — Lists

Christian Collberg

collberg+372@gmail.com

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg

The List Datatype

- All functional programming languages have the **ConsList** ADT built-in. It is called so because lists are constructed by “consing” (adding) an element on to the beginning of the list.
- Lists are defined recursively:
 1. The empty list `[]` is a list.
 2. An element `x` followed by a list `L` (`x:L`), is a list.
- Examples:

```
[ ]  
2 : [ ]  
3 : ( 2 : [ ] )  
4 : ( 3 : ( 2 : [ ] ) )
```

The List Datatype...

- The cons operator " : " is right associative (it binds to the right, i.e.

$$1 : 2 : [] \equiv 1 : (2 : [])$$

so

$$3 : (2 : [])$$

can be written without brackets as

$$3 : 2 : []$$

The List Datatype...

- Lists can also be written in a convenient bracket notation.

$2 : [\] \quad \Rightarrow \quad [2]$

$3 : (2 : [\]) \quad \Rightarrow \quad [3 , 2]$

$4 : (3 : (2 : [\])) \quad \Rightarrow \quad [4 , 3 , 2]$

- You can make lists-of-lists ($[[1] , [5]]$),
lists-of-lists-of-lists ($[[[1 , 2]] , [[3]]]$), etc.

The List Datatype...

- More cons examples:

`1 : [2 , 3]` \Rightarrow `[1 , 2 , 3]`

`[1] : [[2] , [3]]` \Rightarrow `[[1] , [2] , [3]]`

- Note that the elements of a list must be of the same type!

`[1 , [1] , 1]` \Rightarrow `Illegal!`

`[[1] , [2] , [[3]]]` \Rightarrow `Illegal!`

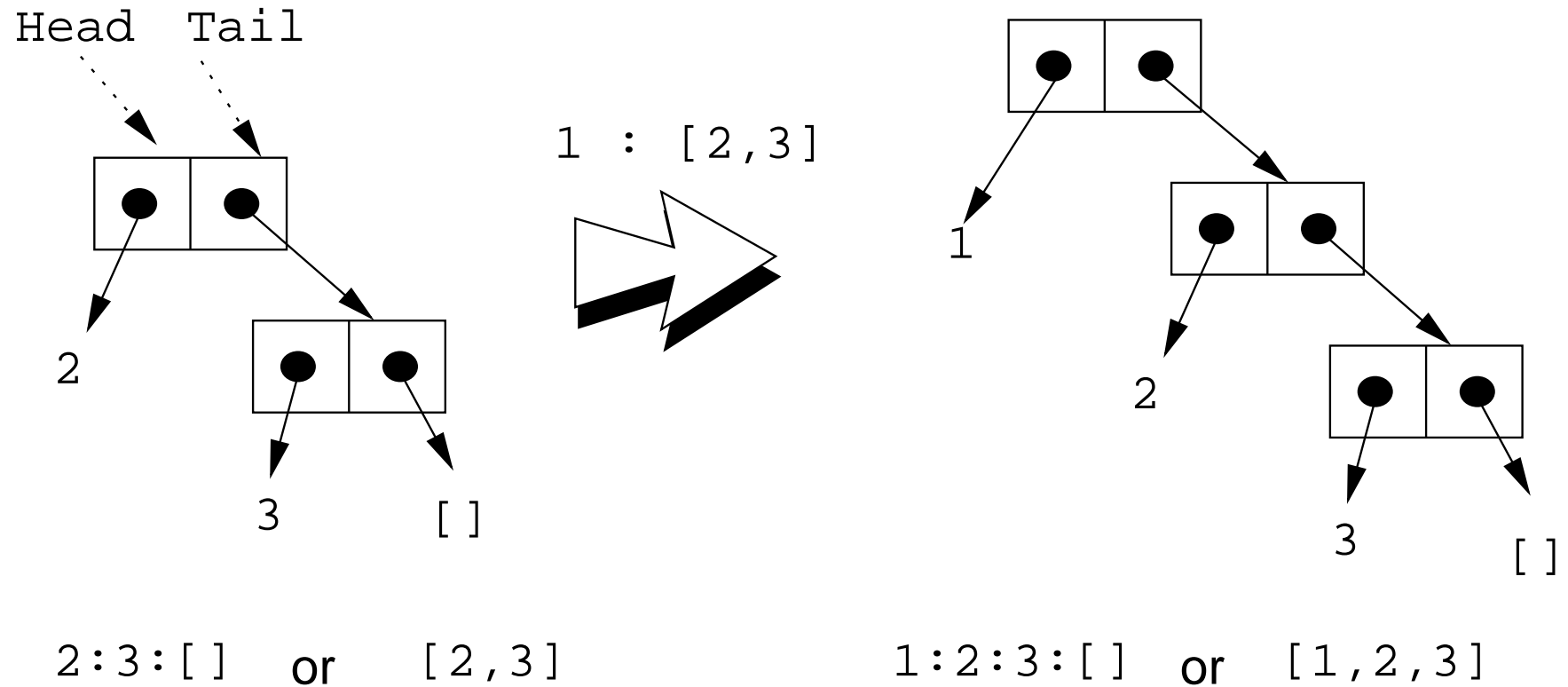
`[1 , True]` \Rightarrow `Illegal!`

Internal Representation

- Internally, Haskell lists are represented as linked `cons-cells`.
- A cons-cell is like a C `struct` with two pointer fields `head` and `tail`.
- The `head` field points to the first element of the list, the `tail` field to the rest of the list.
- The `:`-operator creates a new cons-cell (using `malloc`) and fills in the `head` and `tail` fields to point to the first element of the new list, and the rest of the list, respectively.

Internal Representation...

● Example:



Standard Operations on Lists

head and tail

- The Standard Prelude has many built-in operations on lists.
- Two principal operators are used to take lists apart:
 1. `head L` – returns the first element of `L`.
 2. `tail L` – returns `L` without the first element.
- The cons operator `" : "` is closely related to `head` and `tail`:
 1. `head (x:xs) ≡ x`
 2. `tail (x:xs) ≡ xs`
- The cons operator `" : "` constructs new lists, `head` and `tail` take them apart.

head and tail...

```
head [1,2,3]      ⇒ 1
tail [1,2,3]      ⇒ [2,3]
tail [1]          ⇒ [ ]    ([1] == 1:[ ] )
head [ ]          ⇒ ERROR
tail [ ]          ⇒ ERROR
head (1:[2,3])    ⇒ 1
tail (1:[2,3])    ⇒ [2,3]
head (tail [1,2,3]) ⇒ 2
head (tail [[1], [2], [3,3]]) ⇒ 2
```

length and ++

- `length xs` – Number of elements in the list `xs`.
- `xs ++ ys` – The elements of `xs` followed by the elements of `ys`.

Examples:

<code>length [1,2,3]</code>	\Rightarrow 3
<code>length []</code>	\Rightarrow 0
<code>[1,2] ++ [3,4]</code>	\Rightarrow <code>[1,2,3,4]</code>
<code>[1,2] ++ []</code>	\Rightarrow <code>[1,2]</code>
<code>[1] ++ [2,3] ++ [4]</code>	\Rightarrow <code>[1,2,3,4]</code>
<code>length ([1]++[2,3])</code>	\Rightarrow 3
<code>[1] ++ [length [2,3]]</code>	\Rightarrow <code>[1,2]</code>

concat

- `concat xss` – all of the lists in `xss` appended together.

`concat [[1],[4,5],[6]] ⇒ [1,4,5,6]`

- Note that `concat` takes a **list of lists** as argument.

map

- `map f xs` – list of values obtained by applying the function `f` to the values in `xs`.

`map even [1,2,3] ⇒ [False,True,False]`

`map square [1,2,3] ⇒ [1,4,9]`

- Note that `map` takes a function as its first argument. A function which takes a function as an argument or delivers one as its result, is called a **higher-order function**.
- We will talk more about higher-order functions in future lectures.

More list operation examples

`head ([1,2] ++ [3,4]) ⇒`

`head [1,2,3,4] ⇒ 1`

`tail (concat [[1],[3,4],[5]]) ⇒`

`tail [1,3,4,5] ⇒ [3,4,5]`

`tail (map double (concat [[1],[3],[4]])) ⇒`

`tail (map double [1,3,4]) ⇒`

`tail [2,6,8] ⇒ [6,8]`

The String Type

- A Haskell string is a list of characters:
`type String = [Char]`
- All list manipulation functions can be applied to strings.
- Note that `" " == []`.

```
"Chris"      ⇔ ['C','h','r','i','s']
head "Chris" ⇔ 'C'
tail "Chris" ⇔ ['h','r','i','s']
"Chris" ++ "tian" ⇔
    ['C','h','r','i','s','t','i','a','n']
map ord "Hello" ⇔
    [72,101,108,108,111]
concat ["Have ", "a ", "cow, ", "man! "]
    ⇔ "Have a cow, man!"
```

Recursion Over Lists

Recursion on the Tail

- Compute the length of a list.
- This is called **recursion on the tail**.

```
len :: [Int] -> Int
len xs = if xs == [] then
           0
        else
           1 + len (tail xs)
```

Variable Naming Conventions

- When we write functions over lists it's convenient to use a consistent variable naming convention. We let
 - x, y, z, \dots denote list elements.
 - xs, ys, zs, \dots denote lists of elements.
 - xss, yss, zss, \dots denote lists of lists of elements.

Map Function

- Map a list of numbers to a new list of their absolute values.
- In the previous examples we returned an `Int` — here we're mapping a list to a new list.
- This is called a **map function**.

```
abslist :: [Int] -> [Int]
abslist xs = if xs == [] then
               []
             else
               abs (head xs) : abslist (tail xs)
```

Map Function...

```
> abslist []  
[]  
> abslist [1]  
[1]  
  abslist [1,-2]  
[1,2]
```

Recursion Over Two Lists

- `listeq xs ys` returns `True` if two lists are equal.

```
listeq :: [Int] -> [Int] -> Bool
listeq xs ys = if xs==[] && ys==[] then
    True
    else if xs==[] || ys==[] then
    False
    else if head xs /= head ys then
    False
    else
        listeq (tail xs) (tail ys)
```

Recursion Over Two Lists...

```
> listeq [1] [2]
```

```
False
```

```
> listeq [1] [1]
```

```
True
```

```
> listeq [1] [1,2]
```

```
False
```

```
> listeq [1,2] [1,2]
```

```
True
```

Append

- `append xs ys` takes two lists as arguments and returns a new list, consisting of the elements of `xs` followed by the elements of `ys`.
- To do this recursively, we take `xs` apart on the way down into the recursion, and “attach” them to `ys` on the way up:

```
append :: [Int] -> [Int] -> [Int]
append xs ys = if xs==[] then
                ys
            else
                (head xs) : (append (tail xs) ys)
```

Append...

```
> append [] []  
[]  
> append [1] []  
[1]  
> append [1] [2]  
[1, 2]  
> append [1, 2, 3] [4, 5, 6]  
[1, 2, 3, 4, 5, 6]
```

Arithmetic Sequences

Arithmetic Sequences

- Haskell provides a convenient notation for lists of numbers where the difference between consecutive numbers is constant.

$[1..3] \Rightarrow [1, 2, 3]$

$[5..1] \Rightarrow []$

- A similar notation is used when the difference between consecutive elements is $\neq 1$: Examples:

$[1, 3..9] \Rightarrow [1, 3, 5, 7, 9]$

$[9, 8..5] \Rightarrow [9, 8, 7, 6, 5]$

$[9, 8..11] \Rightarrow []$

Or, in general:

$[m, k..n] \Rightarrow$

$[m, m + (k - m) * 1, m + (k - m) * 2, \dots, n]$

Arithmetic Sequences...

- Or, in English

“ m and k are the first two elements of the sequence. All consecutive pairs of elements have the same difference as m and k . No element is greater than n .”
- Or, in some other words,

“ m and k form a **prototype** for consecutive element pairs in the list.”
- Later in the course we will talk about **infinite lists**. Haskell has the capability to create infinite arithmetic sequences:
 $[3..] \Rightarrow [3, 4, 5, 6, 7, \dots]$
 $[4, 3..] \Rightarrow [4, 3, 2, 1, 0, -1, -2, \dots]$

Summary

- The bracketed list notation `[1 , 2 , 3]` is just an abbreviation for the list constructor notation `1 : 2 : 3 : []`.
- Lists can contain anything: integers, characters, tuples, other lists, but every list must contain elements of the same type only.
- `:`, `++`, `concat`, and list comprehensions create lists.
- `head` and `tail` take lists apart.

Summary...

- The notation $[m..n]$ generates lists of integers from m to n .
- If the difference between consecutive integers is $\neq 1$, we use the slightly different notation $[m, k..n]$. The first two elements of the generated list are m and k . The remaining elements are as far apart as m and k .

Homework

- Which of the following are legal list constructions? First work out the answer in your head, then try it out with the hugs interpreter.

1. 1 : []

2. 1 : [] : []

3. 1 : [1]

4. [] : [1]

5. [1] : [1] : []

Homework

- Show the lists generated by the following Haskell list expressions.

1. `[7..11]`

2. `[11..7]`

3. `[3,6..12]`

4. `[12,9..2]`

Homework

1. Write a function `getElement xs n` which returns the n :th element of a list of integers.
2. Write a function `evenElements xs` which returns a new list consisting of the 0:th, 2:nd, 4:th, ... elements of an integer list `xs`.