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:[]))
  ```

Lists can also be written in a convenient bracket notation.

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

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

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

  ```
  1:2:[] ≡ 1:(2:[])
  ```

  so

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

  can be written without brackets as

  ```
  3:2:
  ```
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!

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

Head Tail

2:3:[] or [2, 3]

1:2:3:[] or [1, 2, 3]

Standard Operations on Lists
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 $(x:xs) \equiv x$
2. tail $(x:xs) \equiv xs$

The cons operator ":" constructs new lists, head and tail take them apart.

length and ++

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

Examples:

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>length [1,2,3]</td>
<td>3</td>
</tr>
<tr>
<td>length [ ]</td>
<td>0</td>
</tr>
<tr>
<td>[1,2] ++ [3,4]</td>
<td>[1,2,3,4]</td>
</tr>
<tr>
<td>[1,2] ++ [ ]</td>
<td>[1,2]</td>
</tr>
<tr>
<td>[1] ++ [2,3] ++ [4]</td>
<td>[1,2,3,4]</td>
</tr>
<tr>
<td>length ([1]++[2,3])</td>
<td>3</td>
</tr>
<tr>
<td>[1] ++ [length [2,3]]</td>
<td>[1,2]</td>
</tr>
</tbody>
</table>

concat

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

Examples:

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>concat [[1],[4,5],[6]]</td>
<td>[1,4,5,6]</td>
</tr>
</tbody>
</table>

Note that concat takes a list of lists as argument.
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.

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']
concat ["Have ","a ","cow ","man!"]
⇔ "Have a cow, man!"
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, ...` denote list elements.
  - `xs, ys, zs, ...` denote lists of elements.
  - `xss, yss, zss, ...` 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)
```

>` abslist []
[]
> abslist [1]
[1]
> abslist [1,-2]
[1,2]
Recursion Over Two Lists

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

```haskell
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...

```haskell
> listeq [1] [2]
False
> listeq [1] [1]
True
> listeq [1,1] [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:

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

Append...

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

\[ [m, k..n] \Rightarrow [m, m+(k-m)*1, m+(k-m)*2, \ldots, n] \]

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:

\[ [3..] \Rightarrow [3,4,5,6,7,\ldots] \]
\[ [4,3..] \Rightarrow [4,3,2,1,0,-1,-2,\ldots] \]

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.

\(\ldots, ++, concat, and list comprehensions create lists.\)

\(head\) and \(tail\) take lists apart.
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. \([0..2]\)
2. \([0..2] : []\)
4. \([0..2] : [] : [2]\)

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 `gete1mt xs n` which returns the \(n\):th element of a list of integers.
2. Write a function `evenelmts xs` which returns a new list consisting of the 0:th, 2:nd, 4:th, \ldots elements of an integer list \(xs\).
For each of the function signatures on the next slide, describe in words what type of function they represent. For example, for f1 you’d say “this is a function which takes one Int argument and returns an Int result.”

Also, for each signature, give an example of a function that would have this signature. For example, “f1 could be the abs function which takes an Int as argument and returns its absolute value.”

f1 :: Int -> Int
f2 :: Int -> Bool
f3 :: (Int,Int) -> Int
f4 :: [Int] -> Int
f5 :: [Int] -> Bool
f6 :: [Int] -> Int -> Bool
f7 :: [Int] -> [Int] -> [Int]
f8 :: [[Int]] -> [Int]
f9 :: [Int] -> [Int]
f10 :: [Int] -> [Bool]