Haskell Lists

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

Learning Group Activity

With your learning group, share answers from the learning group activity.

Blake will go around and check participation. If you have questions on the LGA (and your group members cannot answer), ask Blake. If he cannot answer, feel free to Email me or the list.

List Notations

- `[1, 2, 3]` denotes a list containing the integers 1, 2 and 3
- `[1..10]` denotes a list containing the integers 1 through 10 (inclusively)
- `[1..]` denotes an *infinite* list containing all positive integers (works in Haskell because of lazy evaluation)

List Operations

Concatenating Lists

To join two lists together, use the ++ infix operator:

```
GHCi> [1, 2, 3, 4] ++ [-1, -4, -2, -3]
[1,2,3,4,-1,-4,-2,-3]
```

Cons ("Construct") Operator

The construct (`:`) operator adds an element to the front of a list:

```
GHCi> 10 : [1, 2]
[10,1,2]
```

The cons operator is typically used to construct lists in a recursive function.

head and last

The head and last functions return the first and last elements of a list:

```
GHCi> head [1,2,3]
1
GHCi> last [1,2,3]
3
```

init and tail

The `init` and `tail` functions return all but the last and first (respectively) elements in a list:

```
GHCi> init [1,2,3]
[1,2]
GHCi> tail [1,2,3]
[2,3]
```

take and drop

The take function returns the first n elements in a list:

```
GHCi> take 3 [1..5]
[1,2,3]
```

The drop function returns the list without the first n elements:

```
GHCi> drop 3 [1..5]
[4,5]
```

cycle and repeat

`cycle` takes a *list* and returns the infinite list that continually cycles through the given list:

```
GHCi> take 15 (cycle [1..6])
[1,2,3,4,5,6,1,2,3,4,5,6,1,2]
```

`repeat` takes *anything* and returns the list of that element infinitely repeated:

```
GHCi> take 5 (repeat 4)
[4,4,4,4,4]
```

cycle and repeat

`cycle` takes a *list* and returns the infinite list that continually cycles through the given list:

```
GHCi> take 15 (cycle [1..6])
[1,2,3,4,5,6,1,2,3,4,5,6,1,2]
```

`repeat` takes *anything* and returns the list of that element infinitely repeated:

```
GHCi> take 5 (repeat 4)
[4,4,4,4,4]
```

How does the result of these two operations differ?

- `take 9 (cycle [1..3])`
- `take 3 (repeat [1..3])`

elem: Is it an elem-ent?

The `elem` function takes anything and determines if it is an element in the list:

```
GHCi> elem 10 [1..9]
False
GHCi> elem 10 [1..10]
True
```

elem: Is it an elem-ent?

The `elem` function takes anything and determines if it is an element in the list:

```
GHCi> elem 10 [1..9]
False
GHCi> elem 10 [1..10]
True
```

With your learning group...

`elem 10 [1..]` returns `True`, however, `elem -1 [1..]` never returns. What does this tell us about the implementation or algorithmic complexity of `elem`? What is the take-away from this?

map: Apply a function to each element

`map` takes a function and a list and applies the function to each element of the list:

```
GHCi> double x = x * 2
GHCi> map double [1..3]
[2,4,6]
```

More List Operations

- `length` returns the length
- `reverse` reverses a list
- `sum` returns the sum
- `product` returns the product
- `minimum` and `maximum` return the min/max element

List Comprehensions

Set Builder Notation (most likely review)

In mathematics, we can use the **set builder notation** to quickly specify sets:

$$\{x \times 2 | x \in \{10, \ldots, 20\}\}$$

This reads "the set of all x times 2 such that x is in the set of integers from 10 to 20".

Translating Set Builder to Haskell

Haskell's **list comprehension** notation looks strikingly similar to the set builder notation used in maths:

```
GHCi> [x * 2 | x <- [10..20]]
[20,22,24,26,28,30,32,34,36,38,40]
```

Translating Set Builder to Haskell

Haskell's **list comprehension** notation looks strikingly similar to the set builder notation used in maths:

```
GHCi> [x * 2 | x <- [10..20]]
[20,22,24,26,28,30,32,34,36,38,40]
```

- Elements are "drawn" from the list `[1..10]`
- `x` takes the value of `10` first, then 11, ..
- The list is built by computing `x * 2`

Translating Set Builder to Haskell

Haskell's **list comprehension** notation looks strikingly similar to the set builder notation used in maths:

```
GHCi> [x * 2 | x <- [10..20]]
[20,22,24,26,28,30,32,34,36,38,40]
```

- Elements are "drawn" from the list `[1..10]`
- `x` takes the value of `10` first, then 11, ..
- The list is built by computing `x * 2`

Make sense how this works? Questions?

Predicates

Predicates can be created on a list comprehension by adding them with a comma:

```
GHCi> [x + 1 | x <- [10..20], x `mod` 2 == 0]
[11,13,15,17,19,21]
```

In this case, `x `mod` 2 == 0` must evaluate to `True` for the number to be used in the comprehension.

Predicates

Predicates can be created on a list comprehension by adding them with a comma:

```
GHCi> [x + 1 | x <- [10..20], x `mod` 2 == 0]
[11,13,15,17,19,21]
```

In this case, `x `mod` 2 == 0` must evaluate to `True` for the number to be used in the comprehension.

You can use multiple predicates as well, simply add more commas:

```
GHCi> [x + 1 | x <- [10..20], x `mod` 2 == 0, x /= 14]
[11,13,17,19,21]
```

Drawing from Multiple Lists

Haskell has a syntax to draw from multiple lists in a comprehension. For example:

```
GHCi> [x + y | x <- [1..5], y <- [1..3]]
[2,3,4,3,4,5,4,5,6,5,6,7,6,7,8]
```

Notice that Haskell draws all possible combinations of `x` and `y`, first `x = 1` for all `y`, then `x = 2` for all `y`, ...

Nested Comprehensions

Notice we can nest comprehensions:

```
GHCi> [[x + 1 | x <- [1..y]] | y <- [1..3]]
[[2],[2,3],[2,3,4]]
```

Tuples in Haskell

Pronunciation

One of the hardest things about tuples is how to pronounce it.

- You could pronounce it TWO-pull, as in "quadruple"
- You could pronounce it TUH-pel, as in "quintuple"

Both are correct. So don't get angry at anyone, please!

Tuples

- **Tuples** are used to store multiple heterogeneous (of different type) values as a single value
- Tuples come in different sizes, each of different type
- A tuple of same size but different member types is also a different type
- Written in parentheses: `(1, "hello")`

Tuple Operations

- `fst`: get the first element from a tuple
- snd: get the second element from a tuple
- `zip`: merge the elements from two lists into a single list of tuples:

```
GHCi> zip [1..3] [4..6]
[(1,4),(2,5),(3,6)]
GHCi> zip [1..] ["alpha", "beta", "gamma"]
[(1, "alpha"), (2, "beta"), (3, "gamma")]
```

Examples

List Comprehensions

- Filter out uppercase letters (works since strings are lists too):

```
noUppers st = [c | c <- st, not (elem c ['A'..'Z'])]
```

- Generating tuples in a comprehension:

```
[(a, b, c) | a <- [1..5], b <- [1..5], c <- [1..5]]
```

End of Lecture: Roadmap

Roadmap

- Questions?
- Jack is back next time!
- Thursday's LGA does not need to be split up ahead of time.