Symbolic & Declarative Computing / Artificial Intelligence (COS5012-B)

October 06, 2016 Week 2 Lab3

LAB MANUAL: INTRODUCTION TO LISTS

In this tutorial you will learn

- Ranges
- Infinite Lists
- Lazy Evaluation
- Tuples

Ranges

- Sugar Coating

```
Hugs> [1..20]
Hugs> ['a'..'z']
```

- Cool! Actually its more than just sugar coating

```
Hugs> [2,4..20]
Hugs> ['a','d'..'z']
```

- Ranges with steps aren't as smart as you might expect them to be

```
Hugs> [1,2,4,8,16..100]  - - Problem?
```

- You can only specify one step
- Some sequences that aren't arithmetic are ambiguous if given only by a few of their first terms.

```
Hugs> [20..1]
Hugs> [20,19..1]
```

- Watch out when using floating point numbers in ranges! It is advised not to use them in ranges.

Infinite Lists

- In Haskell, lists can be infinite.
- Ranges can be used to build infinite lists

```
Hugs> [1,2..]
```

- How an infinite list can be stored?

Lazy Evaluation

- No value is ever computed until it is needed
- Lazy evaluation allows infinite lists

- Arithmetic over infinite lists is supported

```
Hugs> take 24 [2,4..]
```

- Because Haskell is lazy, it won't try to evaluate the infinite list immediately because it would never finish. It'll wait to see what you want to get out of that infinite lists. And here it sees you just want the first 24 elements and it gladly obliges.

Functions producing infinite lists

- `cycle` takes a list and cycles it into an infinite list. If you just try to display the result, it will go on forever so you have to slice it off somewhere.

```
Hugs> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
Hugs> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

- `repeat` takes an element and produces an infinite list of just that element. It's like cycling a list with only one element.

```
Hugs> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

List Comprehension

- You are probably familiar with set comprehensions. They're normally used for building more specific sets out of general sets. A basic comprehension for a set that contains the first ten even natural numbers is set notation. The part before the pipe is called the output function, x is the variable, N is the input set and x <= 10 is the predicate. That means that the set contains the doubles of all natural numbers that satisfy the predicate.

Exercise

- Write in Haskell a statement that gives us the doubles of first 10 natural numbers.

```
Hugs> take 10 [2,4..]
Hugs> [x*2 | x <- [1..10]]
```

- Let's add a condition (or a predicate) to that comprehension
- Predicates go after the binding parts and are separated from them by a comma

```
Hugs> [x*2 | x <- [1..10], x*2 >= 12]
```

- How about if we wanted all numbers from 50 to 100 whose remainder when divided with the number 7 is 3?

```
Hugs> [ x | x <- [50..100], x `mod` 7 == 3]
```

- This phenomenon using predicates is also called *filtering*.

- Filtering can be done using more than one predicates. All the predicates need to be separated by commas
- To get all numbers from 10 to 20 that are not 13, 15 or 19

```
Hugs> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
```

- Not only more than one predicates are permissible but also resultant list can also be constructed using more than one input lists
- Make a list by multiplying numbers of two different lists

```
Hugs> [ x*y | x <- [2,5,10], y <- [8,10,11]]
```

Exercise

What will be the output of the following expression

- `[ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]`
- `sum [1 | _ <- [2,4..10]]`
- `take 5 [x*x | x <- [1..]]`
- `take 5 [x*x | x <- [1..], even x]`

Tuples

Tuples offer another way of storing multiple values in a single value. Tuples and lists have two key differences:

- Tuples have a fixed number of elements (immutable); you can't cons to a tuple. Therefore, it makes sense to use tuples when you know in advance how many values are to be stored. For example, we might want a type for storing 2D coordinates of a point. We know exactly how many values we need for each point (two – the x and y coordinates), so tuples are applicable.
- The elements of a tuple do not need to be all of the same type. For instance, in a phonebook application we might want to handle the entries by crunching three values into one: the name, phone number, and the number of times we made calls. In such a case the three values won't have the same type, since the name and the phone number are strings, but contact counter will be a number, so lists wouldn't work.
- Tuples are marked by parentheses with elements delimited by commas. Let's look at some sample tuples:

```
(True, 1)
("Hello world", False)
(4, 5, "Six", True, 'b')
```

- In general we use n-tuple to denote a tuple of size n

Exercise

- Write down the 3-tuple whose first element is 4, second element is "hello" and third element is True.
- Which of the following are valid tuples?
  - `(4, 4)`
  - `(4, "hello")`
  - `(True, "Blah", "foo")`
  - `()`

Tuples within tuples (and other combinations)

We can apply the same reasoning to tuples about storing lists within lists. Tuples are things too, so you can store tuples within tuples (within tuples up to any arbitrary level of complexity). Likewise, you could also have lists of tuples, tuples of lists, and all sorts of related combinations.

```
((2,3), True)
((2,3), [2,3])
[(1,2), (3,4), (5,6)]
```

The type of a tuple is defined not only by its size, but, like lists, by the types of objects it contains. For example, the tuples `("Hello",32)` and `(47,"World")` are fundamentally different. One is of type `(String,Int)`, whereas the other is `(Int,String)`

Exercise

Which of these are valid Haskell, and why?

- `1:(2,3)`
- `(2,4):(2,3)`
- `(2,4):[]`
- `[(2,4),(5,5),('a','b')]`
- `([2,4],[2,2])`

References

- http://learnyouahaskell.com/starting-out#ready-set-go
- https://wiki.haskell.org/How_to_work_on_lists