Introduction to Programming: Lecture 13

K Narayan Kumar

Chennai Mathematical Institute

http://www.cmi.ac.in/~kumar

19 Sep 2013

List Comprehension

List Comprehension

- Pick all the even numbers from a list of integers:

List Comprehension

- Pick all the even numbers from a list of integers:

```
evenonly = filter iseven
```

List Comprehension

- Pick all the even numbers from a list of integers:

  ```
  evenonly = filter iseven
  ```

- This is far more obscure than

  $\{x \mid x \in L, \mathit{iseven}(x)\}$

List Comprehension

- Pick all the even numbers from a list of integers:

  ```
  evenonly = filter iseven
  ```

- This is far more obscure than

  $\{x \mid x \in L, \mathit{iseven}(x)\}$

- Haskell allows you write this almost verbatim.

  ```
  evenonly l = [x | x <- l, iseven x]
  ```

List Comprehension

- Pick all the even numbers from a list of integers:

```
evenonly = filter iseven
```

- This is far more obscure than

  $\{x \mid x \in L, \mathit{iseven}(x)\}$

- Haskell allows you write this almost verbatim.

```
evenonly l = [x | x <- l, iseven x]

sqeven l =  [x*x | x <- l, iseven x]
```

List Comprehension

- Pick all the even numbers from a list of integers:

```
evenonly = filter iseven
```

- This is far more obscure than

$\{x \mid x \in L, iseven(x)\}$

- Haskell allows you write this almost verbatim.

```
evenonly l = [x | x <- l, iseven x]

sqeven l =  [x*x | x <- l, iseven x]
```

- This notation for constructing new lists from existing lists through filtering and mapping is called list comprehension.

Examples

Examples

- List of pairs of integers below 10

Examples

- List of pairs of integers below 10

  `[(x,y) | x<-[1..10], y<-[1..10]]`

  Multiple lists can be used.

Examples

- List of pairs of integers below 10

  `[(x,y) | x<-[1..10], y<-[1..10]]`

  Multiple lists can be used.

- We can write `concat` as follows:

  `concat l = [x | y <- l, x <- y]`

Examples

- List of pairs of integers below 10

  ```
  [(x,y) | x<-[1..10], y<-[1..10]]
  ```

  Multiple lists can be used.

- We can write `concat` as follows:

  ```
  concat l = [x | y <- l, x <- y]
  ```

- The set of Pythogorean triples below 100

  ```
  [(x,y,z) | x<-[1..100], y<-[1..100], z<-[1..100],
                   x*x + y*y == z*z]
  ```

Examples

- List of pairs of integers below 10

  ```
  [(x,y) | x<-[1..10], y<-[1..10]]
  ```

  Multiple lists can be used.

- We can write `concat` as follows:

  ```
  concat l = [x | y <- l, x <- y]
  ```

- The set of Pythogorean triples below 100

  ```
  [(x,y,z) | x<-[1..100], y<-[1..100], z<-[1..100],
             x*x + y*y == z*z]
  ```

  Oops, that produces duplicates.

Examples

- List of pairs of integers below 10

  `[(x,y) | x<-[1..10], y<-[1..10]]`

  Multiple lists can be used.
- We can write `concat` as follows:

  `concat l = [x | y <- l, x <- y]`
- The set of Pythogorean triples below 100

  ```
  [(x,y,z) | x<-[1..100], y<-[1..100], z<-[1..100],
             x*x + y*y == z*z]
  ```

  Oops, that produces duplicates.

  ```
  [(x,y,z) | x<-[1..100], y<-[(x+1)..100],
             z<-[(y+1)..100], x*x + y*y == z*z]
  ```

Examples ...

- Divisors of `n`

Examples ...

- Divisors of `n`

```
divisors n = [x | x <- [1..n], (mod n x) == 0]
```

Examples ...

- Divisors of `n`

  ```
  divisors n = [x | x <- [1..n], (mod n x) == 0]
  ```

- Primes below `n`

Examples ...

- Divisors of `n`

```
divisors n = [x | x <- [1..n], (mod n x) == 0]
```

- Primes below `n`

```
primes n = [x | x <- [1..n], (divisors x == [1,x])]
```

Examples ...

- Divisors of `n`

```
divisors n = [x | x <- [1..n], (mod n x) == 0]
```

- Primes below `n`

```
primes n = [x | x <- [1..n], (divisors x == [1,x])]
```

- Quicksort

Examples ...

- Divisors of `n`

```
divisors n = [x | x <- [1..n], (mod n x) == 0]
```

- Primes below `n`

```
primes n = [x | x <- [1..n], (divisors x == [1,x])]
```

- Quicksort

```
quicksort [] = []
quicksort (x:xs) = (quicksort l)++[x]++(quicksort u)
   where
   l = [y | y <- xs, y < x]
   u = [y | y <- xs, y >= x]
```

List comprehension ...

```
evenlist l  = [ (x:xs) | (x:xs) <- l,
                        (mod (length (x:xs)) 2) == 0 ]
```

List comprehension ...

```
evenlist l  = [ (x:xs) | (x:xs) <- l,
                       (mod (length (x:xs)) 2) == 0 ]
```

- Extract all even length non-empty lists from a given list of lists.

List comprehension ...

```
evenlist l  = [ (x:xs) | (x:xs) <- l,
                         (mod (length (x:xs)) 2) == 0 ]
```

- Extract all even length non-empty lists from a given list of lists.

```
headOfeven  = [ x | (x:xs) <- l,
                         (mod (length (x:xs)) 2) == 0 ]
```

- Extract the head of all the even length lists in a given list of lists.

List comprehension ...

List comprehension ...

- List comprehension does not look like a functional definition.

List comprehension ...

- List comprehension does not look like a functional definition.
- How is a list comprehension reduced?

List comprehension ...

- List comprehension does not look like a functional definition.
- How is a list comprehension reduced?
  - In what order are the expressions evaluated?

List comprehension ...

- List comprehension does not look like a functional definition.
- How is a list comprehension reduced?
  - In what order are the expressions evaluated?
  - What is the complexity of a program written using list comprehension?

List comprehension ...

- List comprehension does not look like a functional definition.
- How is a list comprehension reduced?
  - In what order are the expressions evaluated?
  - What is the complexity of a program written using list comprehension?
- List comprehension is actually a defined construct and can be translated using `map`, `filter` and `concat`

Translating list comprehension

- A list comprehension has the form

  ```
  [e | q1, q2, ..., qN]
  ```

Translating list comprehension

- A list comprehension has the form

  `[e | q1, q2, ..., qN]`

  where each `qi` is a *qualifier*.

Translating list comprehension

- A list comprehension has the form

  `[e | q1, q2, ..., qN]`

  where each `qi` is a qualifier.

- A qualifier is either
  - a boolean expression `b` or
  - of the form `p <- l` where `p` is a pattern and `l` is a list valued expression.

Translating list comprehension

- A list comprehension has the form

  `[e | q1, q2, ..., qN]`

  where each `qi` is a qualifier.

- A qualifier is either
  - a boolean expression `b` or
  - of the form `p <- l` where `p` is a pattern and `l` is a list valued expression.

```
[(x,y,z) | x<-[1..100], y<-[(x+1)..100],
           z<-[(y+1)..100], x*x + y*y == z*z]
```

Translation ...

- A boolean condition acts as a filter.

  ```
  [e | b,Q] = if b then [e | Q] else []
  ```

  It depends only on qualifiers that appeared to its left.

Translation ...

- A boolean condition acts as a filter.

  ```
  [e | b,Q] = if b then [e | Q] else []
  ```

  It depends only on qualifiers that appeared to its left.

- Every matching pattern `p` of `l` generates a possible set of candidates for the answer.

  ```
  [e | p <- l, Q] = concat (map f l)
                    where
                    f p = [e | Q]
                    f _ = []
  ```

Translation ...

- A boolean condition acts as a filter.

  ```
  [e | b,Q] = if b then [e | Q] else []
  ```

  It depends only on qualifiers that appeared to its left.

- Every matching pattern `p` of `l` generates a possible set of candidates for the answer.

  ```
  [e | p <- l, Q] = concat (map f l)
                    where
                    f p = [e | Q]
                    f _ = []
  ```

- Finally, the base case.

  ```
  [e|] = [e]
  ```

Example

- Consider

```
[ x | (x:xs) <- l, (mod (length (x:xs)) 2) == 0 ]
```

Example

- Consider

  ```
  [ x | (x:xs) <- l, (mod (length (x:xs)) 2) == 0 ]
  ```

- Using the rule for `e <- l`

  ```
  concat (map f l)
  where
   f (x:xs)  = [x  |  (mod (length (x:xs)) 2) == 0 ]
   f _       = []
  ```

Example

- Consider

```
[ x | (x:xs) <- l, (mod (length (x:xs)) 2) == 0 ]
```

- Using the rule for `e <- l`

```
concat (map f l)
where
 f (x:xs)   = [x   |   (mod (length (x:xs)) 2) == 0 ]
 f _        = []
```

- Using the rule for `b`

```
concat (map f l)
 where
 f (x:xs)   =  if ((mod (length (x:xs)) 2) == 0)
                     then [x|] else []
 f _        = []
```

Example

```
concat (map f l)
 where
 f (x:xs)  =  if ((mod (length (x:xs)) 2) == 0)
                    then [x|] else []
 f _       = []
```

- Finally

```
concat (map f l)
 where
 f (x:xs)  =  if ((mod (length (x:xs)) 2) == 0)
                    then [x] else []
 f _       = []
```

Example

- Consider

```
[(x,y) | x <- [1..100], y<-[(x+1)..100], x*x = y]
```

Example

- Consider

  `[(x,y) | x <- [1..100], y<-[(x+1)..100], x*x = y]`

- Using the rule for `p <- l`

```
concat (map f [1..100])
 where
 f x = [(x,y) | y <- [(x+1)..100], x*x = y]
```

Example

- Consider

```
[(x,y) | x <- [1..100], y<-[(x+1)..100], x*x = y]
```

- Using the rule for `p <- l`

```
concat (map f [1..100])
 where
 f x = [(x,y) | y <- [(x+1)..100], x*x = y]
```

- Once again expanding using the rule for `p <- l`

```
concat (map f [1..100])
 where
 f x = concat (map g [(x+1)..100])
        where
        g y =  [(x,y) | x*x = y]
        g _ =  []
```

Example ...

- Applying the rule for boolean condition we get

```
concat (map f [1..100])
 where
 f x = concat (map g [(x+1)..100])
         where
         g y =  if (x*x == y) then [(x,y)|] else []
         g _ =  []
```

Example ...

- Applying the rule for boolean condition we get

```
concat (map f [1..100])
 where
 f x = concat (map g [(x+1)..100])
        where
        g y =  if (x*x == y) then [(x,y)|] else []
        g _ =  []
```

- Finally, applying the base condition we get

```
concat (map f [1..100])
 where
 f x = concat (map g [(x+1)..100])
        where
        g y =  if (x*x == y) then [(x,y)] else []
        g _ =  []
```

Example: Sieve

- The famous Sieve algorithm to find primes works as follows:
- Consider the numbers

  $\{2, 3, 4, \ldots\}$
- Repeatedly pick the left most element and
  - output it as prime
  - delete all its multiples from the list

Example: Sieve

- The famous Sieve algorithm to find primes works as follows:
- Consider the numbers

  $\{2, 3, 4, \ldots\}$
- Repeatedly pick the left most element and
  - output it as prime
  - delete all its multiples from the list
- Precisely the list of primes are output.

Example: Sieve

- The famous Sieve algorithm to find primes works as follows:
- Consider the numbers

  $\{2, 3, 4, \ldots\}$
- Repeatedly pick the left most element and
  - output it as prime
  - delete all its multiples from the list
- Precisely the list of primes are output.
- In Haskell,

```
primes = sieve [2..]
 where
 sieve (x:xs) =
   x : (sieve [ y | y <- xs, mod y x > 0 ])
```

- But this is an infinite list (?)

Example: Sieve

- The famous Sieve algorithm to find primes works as follows:
- Consider the numbers

  $\{2, 3, 4, \ldots\}$
- Repeatedly pick the left most element and
  - output it as prime
  - delete all its multiples from the list
- Precisely the list of primes are output.
- In Haskell,

```
primes = sieve [2..]
 where
 sieve (x:xs) =
   x : (sieve [ y | y <- xs, mod y x > 0 ])
```

- But this is an infinite list (?)
- Lazy evaluation to the rescue!