University of Auckland

415.360

Functional Programming

Christian Collberg

March 1, 1996

Gofer Lists

The List Datatype I

- 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 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:[ ]`.

Slide 3–1

The List Datatype II

- 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.
- More cons examples:

```
1:[2,3]         ⇒ [1,2,3]
[1]:[[2],[3]]   ⇒ [[1],[2],[3]]
```

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

```
[1,[1],1]       ⇒ Illegal!
[[1],[2],[[3]]]⇒ Illegal!
[1,True]        ⇒ Illegal!
```

Slide 3–2

Operations on Lists I

- 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`

```
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]
```

Slide 3–3

Operations on Lists II

Gofer has a number of built-in operations on Lists:

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

Examples:

```
length [1,2,3]          ⇒ 3
length [ ]              ⇒ 0
[1,2] ++ [3,4]          ⇒ [1,2,3,4]
[1,2] ++ [ ]            ⇒ [1,2]
[1] ++ [2,3] ++ [4]     ⇒ [1,2,3,4]
length ([1]++[2,3])     ⇒ 3
[1] ++ [length [2,3]]⇒ [1,2]
```

Slide 3–4

Operations on Lists III

- `concat xss` – all of the lists in `xss` appended together.
- `map f xs` – list of values obtained by applying the function `f` to the values in `xs`.

```
concat [[1],[4,5],[6]]  ⇒ [1,4,5,6]
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**.
- Note that `concat` takes a **list of lists** as argument.

Slide 3–5

Operations on Lists IV

Variable Naming Conventions:

- `x, y, z,` ··· denote list elements.
- `xs, ys, zs,` ··· denote lists of elements.
- `xss, yss, zss,` ··· denote lists of lists of elements.

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]
```

Slide 3–6

Basic Types IV – String

- A Gofer string is a list of characters:

  `type String = [Char]`
- All list manipulation functions can be applied to strings.
- Note that `"" == []`.

Examples:

```
"Chris"      ⇔ ['C','h','r','i','s']
head "Chris"    ⇔ 'C'
tail "Chris"    ⇔ ['h','r','i','s']
"Chris" ++ "tian" ⇔
   ['C','h','r','i',···,'i','a','n']
map ord "Hello" ⇔
   [72,101,108,108,111]
concat ["Have ","a ","cow, ","man!"]
   ⇔ "Have a cow, man!"
```

Slide 3–7

Arithmetic Sequences I

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

```
[1..3] ⇒ [1,2,3]
[5..1] ⇒ []
```

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

```
[1,3..9]    ⇒ [1,3,5,7,9]
[9,8..5]    ⇒ [9,8,7,6,5]
[9,8..11]   ⇒ []
```

Or, in general:

```
[m,k..n] ⇒
    [m,m+(k-m)*1,m+(k-m)*2,···,n]
```

Slide 3–8

Arithmetic Sequences II

- 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**. Gofer has the capability to create infinite arithmetic sequences:

```
[3..]     ⇒ [3,4,5,6,7,···]
[4,3..]   ⇒ [4,3,2,1,0,-1,-2,···]
```

Slide 3–9

List Comprehensions I

- Gofer has a notation called **list comprehension** (adapted from mathematics where it is used to construct sets) that is very convenient to describe certain kinds of lists. Syntax:

  [**expr** | **qualifier**, **qualifier**, ···]

  In English, this reads:

  "Generate a list where the elements are of the form **expr**, such that the elements fulfill the conditions in the **qualifier**s."

- Note that in B&W semicolon (";") is used instead of comma (",").
- The **expression** can be any valid Gofer expression.

Slide 3–10

List Comprehensions II

- The **qualifier**s can have three different forms: Generators, Filters, and Local Definitions.

Generator Qualifiers:

- Generate a number of elements that can be used in the **expression** part of the list comprehension. Syntax:

  `pattern <- list_expr`

- The `pattern` is often a simple variable. The `list_expr` is often an arithmetic sequence.

```
[n | n<-[1..5]] ⇒ [1,2,3,4,5]
[n*n | n<-[1..5]] ⇒ [1,4,9,16,25]

[(n,n*n) | n<-[1..3]] ⇒
      [(1,1),(2,4),(3,9)]
```

Slide 3–11

List Comprehensions III

Filter Qualifiers:

- A filter is a boolean expression that removes elements that would otherwise have been included in the list comprehension. We often use a generator to produce a sequence of elements, and a filter to remove elements which are not needed.

```
[n*n | n<-[1..9],even n] ⇒
      [4,16,36,64]
[(n,n*n) | n<-[1..3],n<n*n] ⇒
      [(2,4),(3,9)]
```

Local Definitions:

- We can define a local variable within the list comprehension. Example:

```
[n*n | n = 2] ⇒ [4]
```

Slide 3–12

List Comprehensions IV

- Earlier generators (those to the left) vary more slowly than later ones. Compare Pascal's nested **for**-loops, where earlier (outer) loop indecies vary more slowly than later (inner) ones.

Pascal:

```
for i := 1 to 9 do
   for j := 1 to 3 do
      print (i, j)
```

Gofer:

```
[(i,j) | i<-[1..9], j<-[1..3]] ⇒
      [  (1,1),(1,2),(1,3),
         (2,1),(2,2),(2,3),
            ...
         (9,1),(9,2),(9,3)]
```

Slide 3–13

List Comprehensions V

- Qualifiers to the right may use values generated by qualifiers to the left. Compare Pascal where inner loops may use index values generated by outer loops.

Pascal:

```
for i := 1 to 3 do
   for j := i to 4 do
      print (i, j)
```

Gofer:

```
[(i,j) | i<-[1..3], j<-[i..4]] ⇒
      [  (1,1),(1,2),(1,3),(1,4)
         (2,2),(2,3),(2,4),
         (3,3),(3,4)]
[n*n | n<-[1..10], even n] ⇒
      [4,16,36,64,100]
```

Slide 3–14

Comprehension Example I

- Define a function `doublePos xs` that doubles the positive elements in a list of integers.
- `[Int]` is a type specifier meaning "list of integers".

In English:

"Generate a list of elements of the form `2*x`, where the `x`:s are the positive elements from the list `xs`.

In Gofer:

```
doublePos ::  [Int] -> [Int]
doublePos xs = [2*x | x<-xs, x>0]


?  doublePos [-1,-2,1,2,3]
   [2,4,6]
```

Slide 3–15

Comprehension Example II

- Define a function `spaces n` which returns a string of `n` spaces.

Example:

```
?  spaces 10
   "          "
```

Gofer:

```
spaces ::  Int -> String
spaces n = [' ' | i <- [1..n]]
```

- Note that the expression part of the comprehension is of type `Char`.
- Note that the generated values of `i` are never used.

Slide 3–16

Comprehension Example III

- Define a function `factors n` which returns a list of the integers that divide `n`. Omit the trivial factors `1` and `n`.

Examples:

```
factors 5 ⇒ []
factors 100 ⇒ [2,4,5,10,20,25,50]
```

In Gofer:

```
factors ::  Int -> [Int]
factors n = [i | i<-[2..n-1],
                 mod n i == 0]
```

- `mod` is an ordinary prefix function. It can be used as an infix operator by enclosing it in single quotes: `` n `mod` i == 0 ``.

Slide 3–17

Comprehension Example IV

Pythagorean Triads:

- Generate a list of triples (x, y, z) such that $x^2 + y^2 = z^2$ and $x, y, z \leq n$.

```
triads n = [(x,y,z)|
   x<-[1..n], y<-[1..n], z<-[1..n],
   x^2 + y^2 == z^2]

triads 5 ⇒ [(3,4,5),(4,3,5)]
```

- We can easily avoid generating duplicates:

```
triads' n = [(x,y,z)|
   x<-[1..n], y<-[x..n], z<-[y..n],
   x^2 + y^2 == z^2]

triads' 11 ⇒ [(3,4,5), (6,8,10)]
```

Slide 3–18

Summary I

- B&W reading list:
  1. List notation: 48–50
  2. List operations: 53–56
  3. Strings: 30, 43
  4. List comprehensions: 50–53
- The bracketed list notation `[1,2,3]` is just an abbreviation for the list contructor 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.

Slide 3–19

Summary II

- 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 appart as `m` and `k`.
- A list comprehension `[e|q]` generates a list where all the elements have the form **e**, and fulfill the requirements of the qualifier **q**. **q** can be a generator `x<-list` in which case **x** takes on the values in **list** one at a time. Or, **q** can be a a boolean expression that **filters** out unwanted values.

Slide 3–20

Homework I

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

Homework I (a):

```
1 : []
```

Homework I (b):

```
1 : [] : []
```

Homework I (c):

```
1 : [1]
```

Homework I (d):

```
[] : [1]
```

Homework I (e):

```
[1] : [1] : []
```

Slide 3–21

Homework II

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

Homework II (a):

```
[7..11]
```

Homework II (b):

```
[11..7]
```

Homework II (c):

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

Homework II (d):

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

Homework II (e):

```
[n*n | n<-[1..10],even n]
```

Slide 3–22

Homework III

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

Homework III (a):

```
[7 | n<-[1..4]]
```

Homework III (b):

```
[ (x,y) | x<-[1..3], y<-[4..7]]
```

Homework III (c):

```
[ (m,n) | m<-[1..3], n<-[1..m]]
```

Homework III (d):

```
[j | i<-[1,-1,2,-2], i>0, j<-[1..i]]
```

Homework III (e):

```
[a+b | (a,b)<-[(1,2),(3,4),(5,6)]]
```

Slide 3–23

Homework IV

- Use a list comprehension to define a function `neglist xs` that computes the number of negative elements in a list `xs`.

Template:

```
neglist ::  [Int] -> Int
neglist n = ···
```

Examples:

```
?  neglist [1,2,3,4,5]
   0
?  neglist [1,-3,-4,3,4,-5]
   3
```

Slide 3–24

Homework V

- Use a list comprehension to define a function `gensquares low high` that generates a list of squares of all the even numbers from a given lower limit `low` to an upper limit `high`.

Template:

```
gensquares ::  Int -> Int -> [Int]
gensquares low high =
   [ ··· | ··· ]
```

Examples:

```
?  gensquares 2 5
   [4, 16]
?  gensquares 3 10
   [16, 36, 64, 100]
```

Slide 3–25