

# CSc 372 — Comparative Programming Languages

## 8 : Haskell — Function Examples

Christian Collberg  
Department of Computer Science  
University of Arizona  
collberg+372@gmail.com

Copyright © 2005 Christian Collberg

September 7, 2005

# Functions over Lists

## 1 Breaking Lists — `ctake`

- `ctake n xs` (from the standard prelude) takes a number  $n$  and a list of characters, and returns the first  $n$  elements of the list.

Examples:

```
ctake 3 ['a','b','c','d','e'] ⇒ ['a','b','c']
ctake 3 ['a','b'] ⇒ ['a','b']
```

Haskell:

```
ctake :: Int -> [Char] -> [Char]
ctake 0 _          = [ ]
ctake _ [ ]       = [ ]
ctake (n+1) (x:xs) = x : ctake n xs
```

## 2 Don't Get Confused!

- What do the two arrows in the signature of `ctake` mean?

```
ctake :: Int -> [Char] -> [Char]
```

- This is something called *Currying*, which we will talk about in the next lecture.
- For now, think “two arrows in the function signature means the function takes two arguments.”
- This is a lie, but I'll be more truthful later.
- `ctake` takes an `Int` and a list of `Chars` as input, and returns a list of `Chars`.

## 3 Breaking Lists — `drop`

- `drop n xs` (from the standard prelude) takes a number  $n$  and a list, and returns the remaining elements when the first  $n$  have been removed.

Examples:

```
drop 3 ['a','b','c','d','e'] ⇒ ['d','e']
drop 3 ['a','b'] ⇒ [ ]
drop 3 [1,2,3,4,5] ⇒ [4,5]
```

Haskell:

```
drop :: Int -> [a] -> [a]
drop 0 xs          = xs
drop _ [ ]       = [ ]
drop (n+1) (x:xs) = drop n xs
```

## 4 Don't Get Confused (take 2)!

- What do the two `[a]`s in the signature of `drop` mean?

```
drop :: Int -> [a] -> [a]
```

- `drop` is what's called a *Polymorphic Function*, which we will talk more about soon.
- The idea is that `a` is a *type variable*, that can take on any type we want.
- So, `drop` can work on lists of `Ints`, lists of `Chars`, etc.

## 5 List Element Selection

- The operator `!!` in the standard prelude returns an element of a list. Lists are indexed starting at 0.

Examples:

```
[2,5,8,3,9,5,7] !!3    => 3
[2,5] !!3              => ERROR
[[1],[2,3],[4]] !!1!!0 => 2
```

- We can write our own list element selector function:

```
elmt :: [Int] -> Int -> Int
elmt (x:_) 0      = x
elmt (_:xs) (n+1) = elmt xs n
```

## 6 Don't Get Confused (take 3)!

- We can actually define `elmt` to be an operator, just like in the standard prelude:

```
infixl 9 !!
```

```
(!!) :: [a] -> Int -> a
(x:_) !! 0      = x
(_:xs) !! (n+1) = xs !! n
```

- `infixl 9` declares `!!` to be a left-associative operator with precedence 9.
- We'll talk more about this later...

## 7 The zip Function

- `zip` takes two lists `xs` and `ys` and returns a list `zs` of pairs drawn from `xs` and `ys`. `xs` and `ys` are combined like the two parts of a zipper.
- Extra elements from different length lists are discarded.

Examples:

```
zip [1,2] ['a','b'] => [(1,'a'),(2,'b')]
zip [1,2,3] ['a','b'] => [(1,'a'),(2,'b')]
```

Haskell:

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ = [ ]
```

## 8 The remdups Function

- Define a function to remove duplicate adjacent elements from a list:

```
remdups [1]           => [1]
remdups [1,2,1]       => [1,2,1]
remdups [1,2,1,1,2]   => [1,2,2]
remdups [1,1,1,2]     => [1,2,1]
```

- We have to consider three cases:
  1. The first two elements of the list are identical. Remove one of them, then remove duplicates from the rest of the list.
  2. The first two elements are different. Keep them and remove duplicates from the rest of the list.
  3. There are fewer than two elements in the list. Keep them.

## 9 The remdups Function...

Algorithm in English:

**Case 1:** Let the first two elements of the list be  $x$  and  $y$ . Let  $x==y$ . Example:  $L=[1,1,2,3]$ ,  $x==y==1$ . Discard  $x$ . Recursively remove duplicates from the remaining list  $L=[1,2,3]$ .

**Case 2:** The first two elements of the list  $(x,y)$  are different ( $x/=y$ ). Example:  $L=[1,2,2,3]$ ,  $x==1$ ,  $y==2$ . Append  $x$  onto the result of removing duplicates from the list  $L'=[2,2,3]$  from which  $x$  has been removed.

**Case 3:** The list has 0 or 1 element. Return it.

## 10 The remdups Function...

Simulation:

```
remdups [1,2,2] =>
  1:(remdups 2:[2]) ≡   <- case 2,x=1,y=2,xs=[2]
  1:(remdups [2,2]) =>
    1:(remdups 2:[ ]) ≡ <- case 1,x=y=2,xs=[ ]
    1:(remdups [2]) =>
      1:[2] =>         <- case 3,xs=[2]
      [1,2]
```

## 11 The remdups Function...

Algorithm in Haskell:

```
remdups :: [Int] -> [Int]
remdups x:y:xs =
```

```

if x == y then
  remdups y:xs      ← case 1
else
  x : remdups y:xs ← case 2
remdups xs = xs     ← case 3

```

**case 1:** First two elements identical.

**case 2:** First two elements different.

**case 3:** Less than 2 elements left.

- `x:y:xs` matches any list with 2 or more elements.

## 12 Haskell Guards

- Remember the *guard* syntax in Haskell:

```

fun_name fun_args
  | guard1      = expr1
  | guard2      = expr2
  | ...
  | otherwise    = exprn

```

- This is equivalent to:

```

fun_name fun_args
  if guard1 then
    expr1
  else if guard2 then
    expr2
  else if ...
  else exprn

```

## 13 Haskell Guards...

- Many functions become more succinct using guards:

fact with guards:

```

fact :: Int -> Int
fact n
  | n==0      = 1
  | otherwise = n * fact (n-1)

```

remdups with guards:

```

remdups :: Eq [a] => [a] -> [a]
remdups (x:y:xs)
  | x==y      = remdups (y:xs)
  | x /= y    = x : remdups (y:xs)
remdups xs = xs

```

## 14 Don't Get Confused (take 4)!

- What does the `Eq [a] =>` mean in the signature of `remdups`?

```
remdups :: Eq [a] => [a] -> [a]
```

- Again, `remdups` is defined as a polymorphic function, and should therefore work on lists of any element type.
- However, it will only work on elements for which `==` is defined, because, without an equality test available we can't test if two adjacent elements are the same!
- `Eq [a] =>` means that `remdups` can only be applied to elements that can be compared with `==`.
- We'll talk more about this later...

## 15 The append Function

- We want to define a function that appends two lists together:

```
append [1,2] [3,4] => [1,2,3,4]
append [ ] [1,2]  => [1,2]
```

Only use `cons` ("`:`") and recursion.

- Remember that `cons` creates a new list from an element `x` and a list `xs`, such that `x` is the first element and `xs` the last elements of the list:

```
5 : [1,2]      => [5,1,2]
```

## 16 The append Function...

"Algorithm" for append xs ys:

1. Take `xs` apart and use `cons` to put the elements together to make a new list.
2. Again use `cons` to make `ys` the tail of this new list.

## 17 The append Function...

Simulation:

```
append [1,2,3] [4,5] =>
  1: (append [2,3] [4,5]) =>
    1: (2: (append [3] [4,5])) =>
      1: (2: (3: (append [ ] [4,5]))) =>
        1: (2: (3: [4,5])) =>
          1: (2: [3,4,5]) =>
            1: [2,3,4,5] =>
              [1,2,3,4,5]
```

## 18 The append Function...

- Note how we take the first argument apart when going into the recursion, and how it is put together when returning back up.
- Notice also that the second argument to `append` is never traversed. It is simply “tacked on” (using `cons`) to the end of the new list when the bottom of the recursion has been reached.

## 19 The append Function...

Algorithm in Haskell:

```
append :: [a] -> [a] -> [a]
append [ ] xs = xs
append (x:xs) ys = x : append xs ys
```

++ as append:

- It is more convenient to define `append` as an infix operator. `++` is predefined in the standard prelude.

```
infixr 5 ++
(++ ) :: [a] -> [a] -> [a]
[ ] ++ xs = xs
(x:xs) ++ ys = x : (xs ++ ys)
```

# Local Definitions

## 20 The where Clause

- In some languages we can **nest** declarations, i.e. declarations can be made **local** to a particular procedure:

```
function P (...) : ...
  function X (...) : ...
  ....
begin ... X(...) ... end.
```

The local function **X** can only be accessed from within **P**. This is an important way to break a complicated routine into manageable chunks. We also **hide** the definition of **X** from routines other than **P**.

- Haskell has a **where**-clause that works much the same way as a local function or variable.

## 21 The where Clause...

- The **where**-clause follows **after** a function body:

```
fun_name fun_args =
  <fun_body>
  where
    decl1
    decl2
    ...
    decln
```

- A declaration **decl<sub>i</sub>** is like any global function definition.
- Note that a constant declaration **id = expr** is allowed since it is seen as a constant 0-argument function.

## 22 The where Clause...

```
deriv f x =
  (f(x+dx) - f x)/dx
  where dx = 0.0001
```

```
sqrt x = newton f x
  where f y = y2 - x
```

- Note that the **scope** (area of visibility) of a **where**-clause is the entire right-hand side of the function definition.



## 23 The where Clause...

```
g :: Int -> Int
g n | (n `mod` 3) == x = x
    | (n `mod` 3) == y = y
    | (n `mod` 3) == z = z
    where x = 0
          y = 1
          z = 2
```

## 24 The let Clause

- An other, less flexible way, of introducing a local definition, is the **let**-clause.
- The syntax of a **let**-clause:

```
let
  <local_definitions>
in
  <expression>
```

- Note that the scope of the **let**-clause is only one expression, whereas the **where** clause can span over several.

## 25 The let Clause...

```
f :: [Int] -> [Int]
f [] = []
f xs =
  let
    square a = a * a
    one = 1
    (y:ys) = xs
  in
    (square y + one) : f ys
```

## 26 The let Clause...

```
f [1,2] ⇒
(square 1 + one) : f [2] ⇒
2 : f [2] ⇒
2 : ((square 2 + one) : f []) ⇒
2 : (5 : f []) ⇒
  2 : (5 : []) ⇒
2 : [5] ⇒
[2,5]
```

# Rational Arithmetic Package

## 27 Rational Arithmetic

- Build a package implementing rational arithmetic.

### Arithmetic Laws:

$$\begin{array}{l|l} \frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd} & \frac{a}{b} - \frac{c}{d} = \frac{ad-bc}{bd} \\ \frac{a}{b} * \frac{c}{d} = \frac{ac}{bd} & \frac{a}{b} / \frac{c}{d} = \frac{ad}{bc} \end{array}$$

$$\begin{array}{l} \frac{5}{4} + \frac{6}{7} = \frac{5*7+4*6}{4*7} = \frac{59}{28} \\ \frac{5}{4} * \frac{6}{7} = \frac{5*6}{4*7} = \frac{15}{14} \end{array}$$

## 28 Rational Arithmetic...

- There is more than one way to represent the same rational number:

$$\frac{1}{7} = \frac{-1}{-7} = \frac{3}{21} = \frac{168}{1176}$$

We would like to represent each rational number  $\frac{a}{b}$  in the simplest way, called the **normal** form, such that  $a$  and  $b$  are **relatively prime**. Hence,  $\frac{168}{1176}$  would always be represented as  $\frac{1}{7}$ .

- Two numbers  $a$  and  $b$  are relatively prime if  $a$  and  $b$  have no common divisor. 9 and 16 are relatively prime, but 9 and 15 aren't (they both have the common divisor 3).
- 0 is always represented by  $\frac{0}{1}$ .

## 29 Rational Arithmetic...

- We represent a rational number as a tuple of the numerator and the denominator:

`type Rat = (Int, Int)`

- We normalize a **Rat** by dividing the numerator and denominator by their **greatest common divisor**.

```
normRat :: Rat -> Rat
normRat (_,0) = error("Invalid!\n")
normRat (0,_) = (0,1)
normRat (x,y) = (a `div` d,b `div` d)
  where   a = (signum y) * x
         d = abs y
         b = gcd a b
normRat (-168,1176) => (-1,7)
```

## 30 Rational Arithmetic...

### The signum Function:

- `signum x` (from the standard prelude) returns -1 if `x` is negative, 0 if `x` is 0, and 1 if `x` is positive.

```
signum :: (Num a, Ord a) => a -> Int
signum n | n == 0   = 0
         | n > 0    = 1
         | n < 0    = -1
```

### The gcd Function:

```
gcd :: Int -> Int -> Int
gcd x y = gcd' (abs x) (abs y)
  where gcd' x 0 = x
        gcd' x y = gcd' y (rem x y)
```

```
gcd 78 42    => 6
```

## 31 Rational Arithmetic...

### Arithmetic:

```
negRat :: Rat -> Rat
negRat (a,b) = normRat (-a,b)
```

```
addRat,subRat,mulRat,divRat ::Rat -> Rat -> Rat
addRat (a,b) (c,d) = normRat (a*d + c*b, b*d)
subRat (a,b) (c,d) = normRat (a*d - c*b, b*d)
mulRat (a,b) (c,d) = normRat (a*c, b*d)
divRat (a,b) (c,d) = normRat (a*d, b*c)
```

### Examples:

```
> addRat (4,5) (5,6)
(49,30)
```

## 32 Rational Arithmetic...

### Relational Comparison:

```
eqRat :: Rat -> Rat -> Bool
eqRat (a,0) (c,d) = err
eqRat (a,b) (c,0) = err
eqRat (a,b) (c,d) = (a*d == b*c)
  where err = error "Invalid!"
```

### Examples:

```
> eqRat (4, 0) (4, 1)
Invalid!
> eqRat (4, 0) (4, 0)
Invalid!
> eqRat (4, 5) (4, 5)
```

```
True  
> eqRat (4, 5) (4, 6)  
False
```

# Exercises

## 33 Homework

- Define a function `split xs` that takes a list of pairs, makes two lists, one from the first elements of the pair and the other from the second pair elements, and returns the two lists as a pair.

Examples:

```
split [(1,"a"),(2,"b"),(3,"c")]  
⇒ ([1,2,3],[ "a", "b", "c"])
```

```
split [(1,True),(2,False),(3,False)]  
⇒ ([1,2,3],[True,False,False])
```

## 34 Homework

- We model vectors as triples of floating point numbers:

```
type Vector = (Float, Float, Float)
```

Define functions `add'v`, `scale'v`, `dot'v` (dot product), and `cross'v` (cross product) according to the definitions below:

$$\begin{aligned}(a_1, a_2, a_3) + (b_1, b_2, b_3) &= (a_1 + b_1, a_2 + b_2, a_3 + b_3) \\ k(a_1, a_2, a_3) &= (ka_1, ka_2, ka_3) \\ (a_1, a_2, a_3) \cdot (b_1, b_2, b_3) &= a_1b_1 + a_2b_2 + a_3b_3 \\ (a_1, a_2, a_3) \times (b_1, b_2, b_3) &= (a_2b_3 - b_2a_3, a_3b_1 - a_1b_2, a_1b_2 - a_2b_1)\end{aligned}$$

## 35 Homework...

1. Now model  $3 \times 3$  matrices as triples of `Vector`.
2. Define a function `scale'm` that scales a matrix `m` by a float `s`, i.e. multiplies all elements by `s`.
3. Define a function `add'm` that adds two matrices `a` and `b` together to form a new matrix `c`, i.e.  $c_{i,j} = a_{i,j} + b_{i,j}$ .
4. Define a function `transpose'm m` that turns the rows of a matrix `m` into columns, and vice versa, i.e.  $t_{i,j} = m_{j,i}$ .