EECS 368
Programming Language Paradigms

Dr. Andy Gill

Department of Electrical Engineering & Computer Science
University of Kansas

April 11, 2014

Strings

A **String** is a sequence of characters enclosed in double quotes. Internally and explicitly, however, strings are represented as lists of characters.

```
"Hello" :: String
```

This means the same as `['H','e','l','l','o']`.

```
> "Hello" == ['H','e','l','l','o']
True
```

Because strings are just special kinds of lists, any polymorphic function that operates on lists can also be applied to strings. For example:

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

```
> length "Hello"
5
> drop 2 "Hello"
"llo"
> head "Hello"
'H'
```

The Zip Function

A useful library function is zip, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] -> [b] -> [(a,b)]
```

```
> zip [a,b,c] [1,2,3,4]
[(a,1),(b,2),(c,3)]
```

Using zip we can define a function returns the list of all pairs of adjacent elements from a list:

```
pairs    :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

```
> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```

Factorial

```haskell
factorial  :: Int -> Int
factorial n = product [1..n]
```

factorial maps any integer n to the product of the integers between 1 and n.

Expressions are evaluated by a stepwise process of applying functions to their arguments.

```
factorial 4
```

=

Expressions are evaluated by a stepwise process of applying functions to their arguments.

```
factorial 4
```

=

```
product [1..4]
```

=

Expressions are evaluated by a stepwise process of applying functions to their arguments.

```
factorial 4
```

$=$

```
product [1..4]
```

$=$

```
product [1,2,3,4]
```

$=$

Expressions are evaluated by a stepwise process of applying functions to their arguments.

```
factorial 4
```

=

```
product [1..4]
```

=

```
product [1,2,3,4]
```

=

```
1*2*3*4
```

=

Expressions are evaluated by a stepwise process of applying functions to their arguments.

```
factorial 4
```

=

```
product [1..4]
```

=

```
product [1,2,3,4]
```

=

```
1*2*3*4
```

=

```
24
```

KU

Recursive Functions

In Haskell, functions can also be defined in terms of themselves. Such functions are called recursive.

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

factorial maps 0 to 1, and any other positive integer to the product of itself and the factorial of its predecessor.

=

```
factorial 3
factorial 3
```

=

```
3 * factorial 2
```

=

```
factorial 3
```

$=$

```
3 * factorial 2
```

$=$

```
3 * (2 * factorial 1)
```

$=$

```
factorial 3
```

=

```
3 * factorial 2
```

=

```
3 * (2 * factorial 1)
```

=

```
3 * (2 * (1 * factorial 0))
```

=

```
factorial 3
```

=

```
3 * factorial 2
```

=

```
3 * (2 * factorial 1)
```

=

```
3 * (2 * (1 * factorial 0))
```

=

```
3 * (2 * (1 * 1))
```

=

```
factorial 3
```

=

```
3 * factorial 2
```

=

```
3 * (2 * factorial 1)
```

=

```
3 * (2 * (1 * factorial 0))
```

=

```
3 * (2 * (1 * 1))
```

=

```
3 * (2 * 1)
```

=

```
factorial 3
```
=
```
3 * factorial 2
```
=
```
3 * (2 * factorial 1)
```
=
```
3 * (2 * (1 * factorial 0))
```
=
```
3 * (2 * (1 * 1))
```
=
```
3 * (2 * 1)
```
=
```
3 * 2
```
=

```
factorial 3
```
$=$
```
3 * factorial 2
```
$=$
```
3 * (2 * factorial 1)
```
$=$
```
3 * (2 * (1 * factorial 0))
```
$=$
```
3 * (2 * (1 * 1))
```
$=$
```
3 * (2 * 1)
```
$=$
```
3 * 2
```
$=$
```
6
```

- `factorial 0 = 1` is appropriate because 1 is the identity for multiplication:

$$1 * x = x = x * 1$$

- The recursive definition diverges on integers < 0 because the base case is never reached:

```
> factorial (-1)
Error: Control stack overflow
```

Why is Recursion Useful?

- Some functions, such as factorial, are simpler to define in terms of other functions.
- As we shall see, however, many functions can naturally be defined in terms of themselves.
- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

Recursion on Lists

Recursion is not restricted to numbers, but can also be used to define functions on lists.

```
product          :: [Int] -> Int
product []        = 1
product (n:ns)    = n * product ns
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

```
product [2,3,4]
```

=

```
product [2,3,4]
```

=

```
2 * product [3,4]
```

=

```
product [2,3,4]
```

$=$

```
2 * product [3,4]
```

$=$

```
2 * (3 * product [4])
```

$=$

```
product [2,3,4]
```

=

```
2 * product [3,4]
```

=

```
2 * (3 * product [4])
```

=

```
2 * (3 * (4 * product []))
```

=

```
product [2,3,4]
```
=
```
2 * product [3,4]
```
=
```
2 * (3 * product [4])
```
=
```
2 * (3 * (4 * product []))
```
=
```
2 * (3 * (4 * 1))
```
=

```
product [2,3,4]
```
=
```
2 * product [3,4]
```
=
```
2 * (3 * product [4])
```
=
```
2 * (3 * (4 * product []))
```
=
```
2 * (3 * (4 * 1))
```
=
```
24
```

Using the same pattern of recursion as in product we can define the length function on lists.

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

length maps the empty list to 0, and any non-empty list to the successor of the length of its tail.

```
length [2,3,4]
```

=

```
1 + length [2,3]
```

=

```
1 + (1 + length [3])
```

=

```
1 + (1 + (1 + length []))
```

=

```
1 + (1 + (1 + 0))
```

=

```
3
```

Using a similar pattern of recursion we can define the reverse function on lists.

```
reverse          :: [a] -> [a]
reverse []        = []
reverse (x:xs) = reverse xs ++ [x]
```

reverse maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head.

```
reverse [1,2,3]
```

=

```
reverse [2,3] ++ [1]
```

=

```
(reverse [3] ++ [2]) ++ [1]
```

=

```
((reverse [] ++ [3]) ++ [2]) ++ [1]
```

=

```
(([] ++ [3]) ++ [2]) ++ [1]
```

=

```
[3,2,1]
```

Multiple Arguments

Functions with more than one argument can also be defined using recursion. Consider `zip`.

```
zip                    :: [a] -> [b] -> [(a,b)]
zip []      _           =
```

Multiple Arguments

Functions with more than one argument can also be defined using recursion. Consider zip.

```
zip                 :: [a] -> [b] -> [(a,b)]
zip []      _        = []
zip _       []       =
```

Multiple Arguments

Functions with more than one argument can also be defined using recursion. Consider `zip`.

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

Multiple Arguments

Functions with more than one argument can also be defined using recursion. Consider `zip`.

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

Remove the first n elements from a list:

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

Appending two lists:

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

Quicksort

The quicksort algorithm for sorting a list of integers can be specified by the following two rules:

- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail values $\leq$ the head, sorting the tail values $>$ the head, and then appending the resulting lists on either side of the head value.

Using recursion, this specification can be translated directly into an implementation:

```
qsort         :: [Int] -> [Int]
qsort []       = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a <- xs, a <= x]
      larger  = [b | b <- xs, b > x]
```

This is probably the simplest implementation of quicksort in any programming language!

KU