PROGRAMMING IN HASKELL

Chapter 6 - Recursive Functions

Introduction

Many functions can naturally be defined in terms of other functions.


```
fac  :: Int
fac n = product [1..n]
```


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

Expressions are <u>evaluated</u> by a stepwise process of applying functions to their arguments.

For example:


```
  fac 4
=
  product [1..4]
=
  product [1,2,3,4]
=
  1*2*3*4
=
  24
```

Recursive Functions

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


```
fac 0 = 1
fac n = n * fac (n-1)
```


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

Using the definition of fac, we can reason about the result:

```
  fac 3
=
  3 * fac 2
=
  3 * 2 * fac 1
=
  3 * 2 * 1
=
  3 * 2 * 1 * 1
=
  6
```

Note:

- fac 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:


```
> fac (-1)

Exception: stack overflow
```

Why is Recursion Useful?

- Some functions, such as factorial, are simpler to define in terms of other functions.

- As we shall s[illegible]unctions can naturally be defined in ter[illegible]mselves.

- 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 <u>lists</u>.


```
product
product []     = 1
product (n:ns) = n * produ
```


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

For example:

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


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

For example:

```
   length [1,2,3]
=
   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 maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head.

For example:

```
  reverse [1,2,3]
=
  reverse [2,3] ++ [1]
=
  (revers
=
  ((reverse [] ++ [3]) ++
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

Multiple Arguments

Functions with more than one argument can also be defined using recursion. For example:

- Zipping the element


```
zip          :: [a] → [b] → [(a,b)]
zip xs ys      = ?
```

Remove the first n elements from a list:

```
drop    :: Int → [a] → [a]
drop    = ?
```


Appending two lists:

```
(++)    :: [a] → [a] → [a]
(++)    = ?
```

Quicksort

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

- The empty list is al

- Non-empty lists can be sorted by sorti es ≤ the head, sorting the tail values > the head, and then appending the resulting lists on either side of the head value.

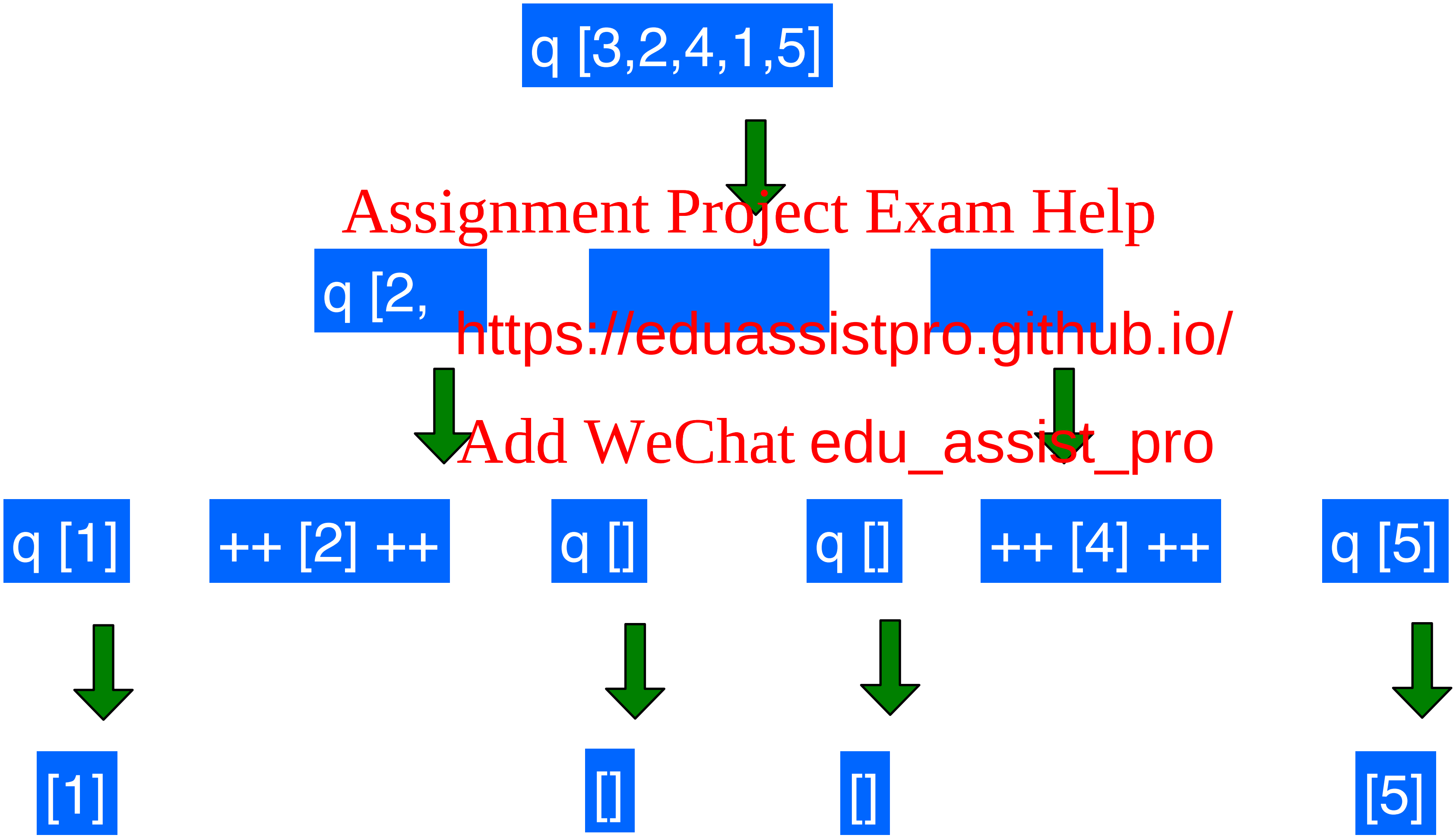
Using recursion, this specification can be translated directly into an implementation:

```
qsort        :: Ord a ⇒ [a] → [a]
qsort []      = []
qsort (x:xs) =
   qsort small
   where
      smaller = [a | a ← xs, a ≤
      larger  = [b | b ← xs, b >
```


Note:

- This is probably the <u>simplest</u> implementation of quicksort in any programming language!

For example (abbreviating qsort as q):



Exercises

(1) Without looking at the standard prelude, define the following library functions using recursion:

- Decide if all lo


```
and :: [Bool] → Bool
```

- Concatenate a list of lists:

```
concat :: [[a]] → [a]
```

- Produce a list with n identical elements:

```
replicate :: Int → a → [a]
```

- Select the nth element of a list:

```
(!!) :: [a] → Int → a
```

- Decide if a value is an element of a list:

```
elem :: Eq a ⇒ a → [a] → Bool
```

(2) Define a recursive function

```
merge :: Ord a ⇒ [a] → [a] → [a]
```


that merges two ingle sorted list. For example:


```
> merge [2,5,6] [1,3,4]

[1,2,3,4,5,6]
```

(3) Define a recursive function

msort :: Ord a ⇒ [a] → [a]

that implements ________ ied by the following two rules:

- Lists of length ≤ 1 are already sorted;

- Other lists can be sorted by sorting the two halves and merging the resulting lists.

PROGRAMMING IN HASKELL

Chapter 5 - List Comprehensions

Set Comprehensions

In mathematics, the <u>comprehension</u> notation can be used to construct new sets from old sets.

$\{x^2 \mid x \in \{1...5\}\}$

The set $\{1,4,9,16,25\}$ of all numbers x^2 such that x is an element of the set $\{1...5\}$.

Lists Comprehensions

In Haskell, a similar comprehension notation can be used to construct new lists from old lists.

[x^2 | x ← [1..5]]

The list [1,4,9,16,25] of all numbers x^2 such that x is an element of the list [1..5].

Note:

- The expression x ← [1..5] is called a generator, as it states how to generate values for x.

- Comprehensions can have multiple generators, separated by commas. For example:


```
> [(x,y) | x ← [1,2,3], y ← [4,5]]

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

- Changing the order of the generators changes the order of the elements in the final list:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(
```


- Multiple generators are like nested loops, with later generators as more deeply nested loops whose variables change value more frequently.

For example:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

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


x ← [1,2,3] is the last generator, so the value of the x component of each pair changes most frequently.

Dependant Generators

Later generators can <u>depend</u> on the variables that are introduced by earlier generators.

[(x,y) | x

The list [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] of all pairs of numbers (x,y) such that x,y are elements of the list [1..3] and y ≥ x.

Using a dependant generator we can define the library function that concatenates a list of lists:

```
concat     :: [[a]] → [a]
concat xss = ?
```


For example:

```
> concat [[1,2,3],[4,5],[6]]

[1,2,3,4,5,6]
```

Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

[x | x ← [1

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

Using a guard we can define a function that maps a positive integer to its list of factors:

```
factors   :: Int → [Int]
factors n =
```


For example:

```
> factors 15

[1,3,5,15]
```

Hint: Using n `mod` x == 0 checks whether the remainder of integer division is 0.

A positive integer is prime if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

```
prime  :: Int → Bool
prime n = ?
```


For example:

```
> prime 15
False

> prime 7
True
```

Using a guard we can now define a function that returns the list of all primes up to a given limit:

```
primes   :: Int → [Int]
primes n =
```


For example:

```
> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]
```

The Zip Function

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


```
zip :: [a] →
```


For example:

```
> 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 <u>pairs</u> of adjacent elements from a list:

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


For example:

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

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

Using pairs we can define a function that decides if the elements in a list are sorted:

```
sorted   :: Ord a ⇒ [a] → Bool
sorted xs =
   and [x ≤ y
```


For example:

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

Using zip we can define a function that returns the list of all positions of a value in a list:

```
positions :: Eq a ⇒ a → [a] → [Int]
positions x
```


For example:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

String Comprehensions

A <u>string</u> is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.


```
"abc" :: String
```

Means ['a', 'b', 'c'] :: [Char].

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

```
> length "abcde"
5

> take 3 "abcde"
"abc"

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

Similarly, list comprehensions can also be used to define functions on strings, such counting how many times a character occurs in a string:

```
count    :: Char -> String -> Int
count x xs
   length [x’
```


For example:

```
> count ’s’ "Mississippi"
4
```

Exercises

(1)

A triple (x,y,z) of positive integers is called <u>pythagorean</u> if $x^2 + y^2 = z^2$. Using a list comprehension, define a function


```
pyths ::
```


that maps an integer n to all such triples with components in [1..n]. For example:

```
> pyths 5
[(3,4,5),(4,3,5)]
```

(2) A positive integer is perfect if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension, define a function

```
perfects :: Int → [Int]
```


that returns the list of all perfect nu iven limit. For example:


```
> perfects 500

[6,28,496]
```

(3) The <u>scalar product</u> of two lists of integers xs and ys of length n is give by the sum of the products of the corresponding integers:

$$\sum_{i=0}^{n-1}$$

Using a list comprehension, define a function that returns the scalar product of two lists.