List Comprehensions and ADTs (slides, where noted, due to Graham Hutton)

1

Outline

- List comprehensions
- String comprehensions
- Type declarations
- Algebraic data types (ADTs)
- Pattern matching
- Case expressions

- Countdown: putting these together

2

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}.

3

Lists Comprehensions

In Haskell, a similar comprehension notation can be used to construct new <u>lists</u> 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].

4

Note:

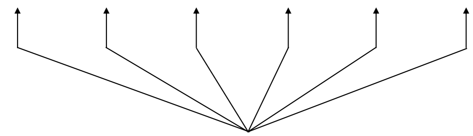
- The expression x ← [1..5] is called a generator, as it states how to generate values for x
- Comprehensions can have multiple generators, separated by commas. For example:

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

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

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

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


```
> [(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 depend on the variables that are introduced by earlier generators.

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


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

For example:

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

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

```
concat :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

Guards

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

```
[x | x ← [1..10], even x]
```

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.

Can you define a function that maps a positive integer to its list of factors, e.g.,

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

```
factors :: Int → [Int]
factors n = [x | x ← [1..n], n `mod` x == 0]
```

9

9

Using factors, we can define a function that decides if a number is prime? E.g.,

prime n = length (factors n) == 2

```
> prime 15
False

> prime 7
True
```

```
prime :: Int → Bool
prime n = factors n == [1,n]
```

Programming in Haskell, modified from a slide due to G. Hutton

10

10

And using prime as guard, we can now define a function that returns the list of all primes up to a given limit. E.g.,:

```
> primes 40

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

```
primes :: Int → [Int]
primes n = [x | x <- [2..n], prime x]
```

Programming in Haskell, modified from a slide due to G. Hutton

11

11

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)]
```

For example:

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

[('a',1),('b',2),('c',3)]
```

Programming in Haskell, modified from a slide due to G. Hutton

12

12

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

For example:

[1,2,3,4]
[2,3,4]

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

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

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


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

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

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

```
sorted :: Ord a ⇒ [a] → Bool
sorted xs = and [x ≤ y | (x,y) ← pairs xs]
```

[True, False, True]

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

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

```
positions :: Eq a ⇒ a → [a] → [Int]
positions x xs =
   [i | (x',i) ← zip xs [0..], x == x']
```


String Comprehensions

A string 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.

For example:

```
> count 's' "Mississippi"
4
```

```
count :: Char → String → Int
count x xs = length [x' | x' ← xs, x == x']
```


Exercises

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

```
pyths :: Int → [(Int,Int,Int)]
```

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 numbers up to a given limit. For example:

```
> perfects 500

[6,28,496]
```

[1, 2, 3] [4, 5, 6]
= 32

(3) The inner product of two lists of integers xs and ys of length n is given by the sum of the products of the corresponding integers:

$$\sum_{i=0}^{n-1} (xs_i * ys_i)$$

zipWith

Hoogle

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

Outline

- List comprehensions
- String comprehensions
- Type declarations
- Algebraic datatypes (ADTs)
- Pattern matching
- Case expressions
- Countdown: putting these together

Type Declarations

In Haskell, a new name for an existing type can be defined using a type declaration.

```
type String = [Char]
```

String is a synonym, also called *type alias*, for the type [Char].

Type declarations can be used to make other types easier to read. For example, given

```
type Pos = (Int,Int)
```

we can define:

```
origin :: Pos
origin = (0,0)

left :: Pos → Pos
left (x,y) = (x-1,y)
```

Like function definitions, type declarations can also have parameters. For example, given

```
type Pair a = (a,a)
```

we can define:

```
mult :: Pair Int → Int
mult (m,n) = m * n

copy :: a → Pair a
copy x = (x,x)
```

Programming in Haskell, slide due to G. Hutton 25

25

Type declarations can be nested:

```
type Pos = (Int,Int)

type Trans = Pos → Pos
```

✔

However, they cannot be recursive:

```
type Tree = (Int,[Tree])
```

✘

Programming in Haskell, slide due to G. Hutton 26

26

Algebraic Datatypes (ADTs)

A completely new type, an algebraic data type, can be defined by specifying its values using a data declaration.

```
data Bool = False | True
```

Bool is a new type, with two new values False and True.

Programming in Haskell, modified from a slide due to G. Hutton 27

27

Note:

- The two values False and True are called the constructors for the datatype Bool
- Datatype and constructor names must always begin with an upper-case letter
- The Bool datatype is an enumeration type – this is Haskell's way of defining them. Other examples:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
data Month = Jan | Feb | … | Nov | Dec
```

Programming in Haskell, modified from a slide due to G. Hutton 28

28

Values of those new types and datatypes can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers :: [Answer]
answers = [Yes,No,Unknown]

flip :: Answer → Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

Programming in Haskell, slide due to G. Hutton 29

29

The constructors in a datatype declaration can also have parameters. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
square :: Float → Shape
square n = Rect n n

area :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Programming in Haskell, slide due to G. Hutton 30

30

Note:

- Shape has values of the form Circle r where r is a float, and Rect x y where x and y are floats.
- Circle and Rect can be viewed as <u>functions</u> that construct values of type Shape:

```
Circle :: Float → Shape

Rect :: Float → Float → Shape
```

Programming in Haskell, slide due to G. Hutton 31

31

Algebraic data types are tagged unions (aka sums) of products (aka records)

```
data Shape = Circle Float
           | Rect Float Float
```

union

Haskell keyword

the new type

new constructors (a.k.a. tags, disjuncts, summands) ~~Line~~ is a ~~binary~~ constructor, ~~Triangle~~ is a ~~ternary~~ …

Circle unary Rect binary

Programming in Haskell, A Milanova 32

32

Not surprisingly, data declarations themselves can also have parameters. For example, given

```
data Maybe a = Nothing | Just a
```

we can define: safediv

10 `div` 5 = 2
10 `div` 0 = ...

safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv n d = Just (n `div` d)

Recursive Types

In Haskell, algebraic datatypes can be declared in terms of themselves. That is, they can be recursive.

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat → Nat.

0 : Zero
1 : Succ Zero
2 : Succ (Succ Zero)
...

Note:

- A value of type Nat is either Zero, or of the form Succ n where n :: Nat. That is, Nat contains the following infinite sequence of values:

```
Zero

Succ Zero

Succ (Succ Zero)

⋮
```


- We can think of values of type Nat as natural numbers, where Zero represents 0, and Succ represents the successor function 1+.

- For example, the value

```
Succ (Succ (Succ Zero))
```

represents the natural number

```
1 + (1 + (1 + 0)) = 3
```

Using recursion, it is easy to define functions that convert between values of type Nat and Int:

```
nat2int :: Nat → Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int → Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```


data Nat = Zero | Succ Nat

Two naturals can be added by converting them to integers, adding, and then converting back:

```
add :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

However, using recursion the function add can be defined without the need for conversions:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```


data Lexp = Var Name
| App Lexp Lexp
| Lambda Name Lexp

Using recursion, a suitable new algebraic datatype to represent arithmetic expressions can be declared by:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

For example, a certain expression is represented as follows:

1 2*3

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

1 + 2*3

Using recursion, it is now easy to define functions that process expressions. For example:

```
size :: Expr → Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval :: Expr → Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Add / Val 1 / Mul / Val 2 / Val 3

A data constructor constructs a new data object. For example:

```
Val 1
Val 3
Mul (Val 1) (Val 3)
Add (Val 1) (Mul (Val 2) (Val 3))
```

A type constructor constructs a new type object. For example:

```
safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)
```


Exercises

(1) Using recursion and the function add, define a function that multiplies two natural numbers.

(2) Define a type Tree a of binary trees built from Leaf values of type a using a Node constructor that takes two binary trees as parameters.

data Tree a = Leaf a | Node (Tree a) (Tree a)

data Tree a = Leaf | Node (Tree a) a (Tree a)

Outline

- List comprehensions
- String comprehensions
- Type declarations
- Algebraic datatypes (ADTs)
- Pattern matching
- Case expressions
- Countdown: putting these together

Pattern Matching with Case Expressions

Examine values of an algebraic data type. For example:

```
area :: Shape -> Float
area s = case s of
         Circle r -> pi * r^2
         Rect x y -> x * y
```

Two important points on case expressions:

- Test: does the given value match this pattern?
- Binding: if value matches, bind corresponding values of **s** and pattern

Notes:

An underscore _ is the wildcard pattern. It matches anything.

As-pattern

A pattern x@pat can be used to match a pattern while retaining the variable referring the object being matched:

```
foo::Expr -> String
foo e@(Add _ _) = e ++ "is Add expression"
```

Patterns can be nested:

```
bar::Expr -> String
bar e@(Add _ r@(Add _ _)) = e ++ "is …"
```


45

Notes:

Patterns are matched in turn until

```
foo::Expr -> String
foo e@(Add _ _) = e ++ "is Add expression"
```

Patterns can be nested:

```
bar::Expr -> String
bar e@(Add _ r@(Add _ _)) = e ++ "is …"
```


46

Exercises

data Nat = Zero | Succ Nat

(1) Write the ~~safediv~~ int2Nat function using a case expression.

```
int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))

int2nat :: Int -> Maybe Nat
int2nat 0 = Just Zero
int2nat n = if n < 0
            then Nothing
            else Just (Succ (int2nat (n-1)))
```

Almost!

47

Outline

- List comprehensions
- String comprehensions
- Type declarations
- Algebraic datatypes (ADTs)
- Pattern matching
- Case expressions
- Countdown: putting these together

48

What is Countdown? Example

Using the numbers

1 3 7 10 25 50

and the arithmetic operators

\+ - * ÷

construct an expression whose value is 765

Rules

- All the numbers, including intermediate results, must be positive naturals (1,2,3,…).
- Each of the source numbers can be used at most once when constructing the expression.
- We abstract from other rules that are adopted on television for pragmatic reasons.

For our example, one possible solution is

```
(25-10) * (50+1)   =   765
```

Notes:

- There are 780 solutions for this example.
- Changing the target number to 831 gives an example that has no solutions.

Evaluating Expressions

Operators:

```
data Op = Add | Sub | Mul | Div
```

Apply an operator:

```
apply :: Op → Int → Int → Int
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

Decide if the result of applying an operator to two positive natural numbers is another positive natural number:

```
valid :: Op → Int → Int → Bool
valid Add _ _ = True
valid Sub x y = x > y
valid Mul _ _ = True
valid Div x y = x `mod` y == 0
```

Expressions:

```
data Expr = Val Int | App Op Expr Expr
```

Programming in Haskell, slide due to G. Hutton

53

Return the overall value of an expression, provided that it is a positive natural number:

```
eval :: Expr → [Int]
eval (Val n)     = [n | n > 0]
eval (App o l r) = [apply o x y | x ← eval l
                                , y ← eval r
                                , valid o x y]
```

Either succeeds and returns a singleton list or fails and returns the empty list.

Programming in Haskell, slide due to G. Hutton

54

Formalising The Problem

Return a list of all possible ways of choosing zero or more elements from a list:

```
choices :: [a] → [[a]]
```

For example:

```
> choices [1,2]

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

Programming in Haskell, slide due to G. Hutton

55

Return a list of all the values in an expression:

```
values :: Expr → [Int]
values (Val n)     = [n]
values (App _ l r) = values l ++ values r
```

Decide if an expression is a solution for a given list of source numbers and a target number:

```
solution :: Expr → [Int] → Int → Bool
solution e ns n = elem (values e) (choices ns)
                  && eval e == [n]
```

Programming in Haskell, slide due to G. Hutton

56

Brute Force Solution

Return a list of all possible ways of splitting a list into two non-empty parts:

```
split :: [a] → [([a],[a])]
```

For example:

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

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

Programming in Haskell, slide due to G. Hutton

57

Return a list of all possible expressions whose values are precisely a given list of numbers:

```
exprs :: [Int] → [Expr]
exprs []  = []
exprs [n] = [Val n]
exprs ns  = [e | (ls,rs) ← split ns
               , l       ← exprs ls
               , r       ← exprs rs
               , e       ← combine l r]
```

The key function in this lecture.

Programming in Haskell, slide due to G. Hutton

58

Combine two expressions using each operator:

```
combine :: Expr → Expr → [Expr]
combine l r =
     [App o l r | o ← [Add,Sub,Mul,Div]]
```

Return a list of all possible expressions that solve an instance of the countdown problem:

```
solutions :: [Int] → Int → [Expr]
solutions ns n = [e | ns' ← choices ns
                    , e   ← exprs ns'
                    , eval e == [n]]
```

Programming in Haskell, slide due to G. Hutton

59

Can We Do Better?

- Many of the expressions that are considered will typically be invalid - fail to evaluate.
- For our example, only around 5 million of the 33 million possible expressions are valid.
- Combining generation with evaluation would allow earlier rejection of invalid expressions.

Programming in Haskell, slide due to G. Hutton

60

Fusing Two Functions

Valid expressions and their values:

```
type Result = (Expr,Int)
```

We seek to define a function that fuses together the generation and evaluation of expressions:

```
results :: [Int] → [Result]
results ns = [(e,n) | e ← exprs ns
                    , n ← eval e]
```

Programming in Haskell, slide due to G. Hutton

61

This behaviour is achieved by defining

```
results []  = []
results [n] = [(Val n,n) | n > 0]
results ns  =
   [res | (ls,rs) ← split ns
        , lx      ← results ls
        , ry      ← results rs
        , res     ← combine' lx ry]
```

where

```
combine' :: Result → Result → [Result]
```

Programming in Haskell, slide due to G. Hutton

62

Combining results:

```
combine' (l,x) (r,y) =
   [(App o l r, apply o x y)
      | o ← [Add,Sub,Mul,Div]
      , valid o x y]
```

New function that solves countdown problems:

```
solutions' :: [Int] → Int → [Expr]
solutions' ns n =
   [e | ns'   ← choices ns
      , (e,m) ← results ns'
      , m == n]
```

Programming in Haskell, slide due to G. Hutton

63

Can We Do Better?

- Many expressions will be essentially the same using simple arithmetic properties, such as:

```
x * y  =  y * x

x * 1  =  x
```

- Exploiting such properties would considerably reduce the search and solution spaces.

Programming in Haskell, slide due to G. Hutton

64

Exploiting Properties

Strengthening the valid predicate to take account of commutativity and identity properties:

```
valid :: Op → Int → Int → Bool
valid Add x y  = x ≤ y
valid Sub x y  = x > y
valid Mul x y  = x ≤ y && x ≠ 1 && y ≠ 1
valid Div x y  = x `mod` y == 0 && y ≠ 1
```