Session 3

List Comprehensions

Recursion

Universiteit Antwerpen

Set Comprehensions

In mathematics, the comprehension 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}.

Universiteit Antwerpen

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..3], y ← [1..2]]

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

Universiteit Antwerpen

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

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

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

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

Universiteit Antwerpen

Dependant Generators

Later generators can depend on the variables that are introduced by earlier generators.

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

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 $x \leq y$.

Universiteit Antwerpen

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

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

For example:

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

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

Universiteit Antwerpen

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.

Universiteit Antwerpen

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

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

For example:

```
> factors 15

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

Universiteit Antwerpen

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 = factors n == [1,n]
```

For example:

```
> prime 15
False

> prime 7
True
```

Universiteit Antwerpen

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 = [x | x ← [1..n], prime x]
```

For example:

```
> primes 40

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

Universiteit Antwerpen

Exercises

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

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

that maps a number n to the list of all triads with components in the range [1..n].

Universiteit Antwerpen

(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]
```

Session 3

List Comprehensions

Recursion

Universiteit Antwerpen

Introduction

As we have seen, many functions can naturally be defined in terms of other functions.

```
factorial   :: Int → Int
factorial n = product [1..n]
```

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

Universiteit Antwerpen

Recursive Functions

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

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

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

Universiteit Antwerpen

For example:

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

Universiteit Antwerpen

Why is Recursion Useful?

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

- In practice, however, most 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.

Universiteit Antwerpen

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 (x:xs)   = x * product xs
```

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 [1,2,3]
=
  product (1:(2:(3:[])))
=
  1 * product (2:(3:[]))
=
  1 * (2 * product (3:[]))
=
  1 * (2 * (3 * product []))
=
  1 * (2 * (3 * 1))
=
  6
```

Universiteit Antwerpen

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.

Universiteit Antwerpen

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

```
qsort        :: [Int] → [Int]
qsort []     = []
qsort (x:xs) = qsort [a | a ← xs, a ≤ x]
               ++ [x] ++
               qsort [b | b ← xs, b > x]
```

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

Universiteit Antwerpen

For example (abbreviating qsort as q):

```
q [3,2,4,1,5]
        ↓
q [2,1] ++ [3] ++ q [4,5]
   ↓                 ↓
q [1] ++ [2] ++ q []      q [] ++ [4] ++ q [5]
 ↓               ↓         ↓               ↓
[1]             []        []              [5]
```

Exercises

(1) Define a recursive function

```
insert :: Int → [Int] → [Int]
```

that inserts an integer into the correct position in a sorted list of integers. For example:

```
> insert 3 [1,2,4,5]

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

Universiteit Antwerpen

(2) Define a recursive function

```
isort :: [Int] → [Int]
```

that implements <u>insertion sort</u>, which can be specified by the following two rules:

- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail and inserting the head into the result.

(3) Define a recursive function

```
merge :: [Int] → [Int] → [Int]
```

that merges two sorted lists of integers to give a single sorted list. For example:

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

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

Universiteit Antwerpen

(4) Define a recursive function

```
msort :: [Int] → [Int]
```

that implements merge sort, which can be specified 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.

(5) Test both sorting functions using Hugs to see how they compare. For example:

```
> :set +s

> isort (reverse [1..500])

> msort (reverse [1..500])
```

The command :set +s tells Hugs to give some useful statistics after each evaluation.

Universiteit Antwerpen