COMP105 Lecture 7

Recursion

Recursion

A **recursive** function is one that calls itself

```
factorial n = if n > 1
              then n * factorial (n-1)
              else 1
```

A recursive function has

- A **base case**
- One or more recursive **rules** that move us closer to the base case

Recursion in action

```
factorial n = if n > 1
                then n * factorial (n-1)
                else 1
```

```
factorial 4
→ 4 * factorial 3
→ 4 * 3 * factorial 2
→ 4 * 3 * 2 * factorial 1
→ 4 * 3 * 2 * 1
→ 24
```

Nicer syntax for recursive functions

We can use **pattern matching** to remove the `if`

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

Haskell processes this from **top to bottom**

- First check if the argument matches 1
- If not, fall through to the next case

Nicer syntax for recursive functions

```
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

The “catch-all” case must come last

Base cases

Every recursive function must have a **base case**

- It gives a stopping condition for the recursion
- It is usually the simplest case
- You can have more than one base case

Recursion with no base case will **never terminate**

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

```
factorial 2  →  2 * factorial 1
→  2 * 1 * factorial 0
→  2 * 1 * 0 * factorial (-1) ...
```

Recursive rules

Each recursive rule **makes progress** towards a base case

- Usually means making an argument smaller
- There can be more than one recursive rule

If no progress is made then the recursion will **never terminate**

```
factorial 1 = 1
factorial n = n * factorial n
```

```
factorial 2  →  2 * factorial 2
→  2 * 2 * factorial 2  →  ...
```

Comparison to imperative languages

Recursion is the only way to do **looping** in functional programming

```
while (condition)
{
    <lots of computation>
}
```

Base cases are like the **stopping condition** of a loop

Recursive rules do the **computation**

Anything that you can do with a loop can be done by recursion

- But there is not a simple way to translate between the two

Some more examples

Compute 16^x:

```
pow16 0 = 1
pow16 x = 16 * pow16 (x-1)
```

Multiply two numbers together:

```
multiply x 1 = x
multiply x y = x + multiply x (y-1)
```

Exercises

1. Use pattern matching to write a function `smallPrime` that takes one integer `x` and returns `True` if `x` is 2, 3, 5, or 7, and `False` otherwise

2. Write a function `sumUpTo` that takes one parameter n and computes $1 + 2 + \cdots + n$