COMP105 Lecture 7

More Complex Recursion and Guards

Multiple base cases

Recursive functions are allowed to have **multiple base cases**

- Each base case represents a different stopping condition

```
is_even 0 = True
is_even 1 = False
is_even n = is_even (n-2)
```

```
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

Multiple recursive rules

More complex recursive functions may have **more than one** recursive rule

```
even_sum 0 = 0
even_sum 1 = 0
even_sum x = if x `mod` 2 == 0
                 then x + even_sum (x-1)
                 else     even_sum (x-1)
```

It's important to make sure that the rules are **comprehensive**

- We have to make sure that there is always some recursive rule that can be applied

Nicer syntax for multiple rules

We can use **guards** to make our rules prettier

```
even_sum x
     | (x == 0) = 0
     | (x == 1) = 0
     | (x `mod` 2 == 0) = x + even_sum (x-1)
     | otherwise        =     even_sum (x-1)
```

Guard syntax

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

The guards are listed after the function name and arguments

- Each guard has the format `| <test> = <expression>`
- `<test>` is an expression that evaluates to True or False
- `<expression>` can be anything
- The special `otherwise` test is a catch-all

Guards are a good alternative to a load of nested `if`s

Guards vs. pattern matching

We've now seen two ways to write our functions

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

-- Guards
factorial n
    | n == 1    = 1
    | otherwise = n * factorial (n-1)
```

Pattern matching works best for things that can be pattern matched (eg. tuples, lists)

Guards work best when you have conditionals to test

Advice on recursion

Recursion can look easy when someone else does it

But writing your own recursive functions can be hard at first

Formulating problems in a recursive way is a **skill** that we will develop on the course

- Seeing lots of examples will help
- Trying lots of exercises will also help

Advice on recursion

When trying to design a recursive function think about

- When do you want to stop?
  - These a probably the base cases

- How do you make progress towards a base case?
  - How do you make the problem smaller?
  - There might be multiple cases to consider here
  - These will be the recursive rules

- What do you need to do to get to the smaller case?
  - These will be the operations that you need to carry out within each rule

Advice on recursion

It can be helpful to imagine that the function **already works** for all smaller cases

Suppose that you already have a function that computes `factorial (n-1)`

- What do we have to do to compute `factorial n`?
- Just multiply `factorial (n-1)` by `n`

Suppose that you have a function that computes `even_sum (x-1)`

- If `x` is even then add `x` to `even_sum (x-1)`
- If `x` is odd then add nothing to `even_sum (x-1)`

Exercises

1. Use guards to write a function `sign` that takes an integer `x` and returns `"negative"` if $x < 0$, `"zero"` if $x = 0$ and `"positive"` if $x > 0$

2. Write a function `odd_product` that takes an integer `x` and multiplies all of the odd numbers less than `x` together. So `odd_product 7 = 1 * 3 * 5 * 7`