Testing Recursion

CSC345: Programming Languages and Paradigms

WCU
WEST CHESTER UNIVERSITY

Testing and Properties

Test Driven Development (TDD):

- Software development process
- Relies on the repetition of a very short development cycle
- Automatic grading!

One typical process:

- Add a test
- Run all tests and see if the new one fails
- Write some code
- Run tests
- Refactor code
- Repeat

How to write tests for Haskell?

```
-- Testing functions
test_exor1, test_exor2, test_exor3, test_exor4 :: Bool
test_exor1 = (exOr True True) == False
test_exor2 = (exOr True False) == True
test_exor3 = (exOr False True) == True
test_exor4 = (exOr False False) == False
```

We want all 4 of these tests to evaluate to **`True`**.

Demo

Prompt: Write tests for `threeMax` function

- (It's easily possible to make a mistake in coding not only your **`threeMax`** function but also your test functions.)

Recursion

(A very important programming mechanism)

- Recursion: definition in which a function or object refers to itself

- *Prompt*: can you give a template for “primitive recursion”
  - Primitive recursion: pattern for most (but not all) recursive problems

```
func n
  | n == 0     = …                    -- base case
  | n > 0      = … func (n-1)         -- recursive step
```

- Going from “goal” (what we want to compute) backwards to the base case (using the recursive step)
- Base case is typically the simplest case
- Recursive step: a way of going from the value at **`n-1`** to the next value at **`n`**

Example: the *factorial* fn

6!

6! = 1 * 2 * 3 * 4 * 5 * 6

6! = 5! * 6 Recursive Step

- What is the base case?

0! = 1 Defined by mathematicians

```
func n
  | n == 0     = …                    -- base case
  | n > 0      = … func (n-1)         -- recursive step
```

```
fac :: Integer -> Integer
fac n
  | n == 0 = 1
  | n > 0  = fac (n-1) * n
```

Demo
Trace of `fac 4`

Try **`fac (-2)`**

How to fix?

Example: a function that sums the factorials

$$0! + 1! + \ldots + n!$$

Example: a function `rangeProduct` that given natural numbers *m* and *n*, returns the product of those integers between them, inclusive

$$m * (m + 1) * \ldots * (n - 1) * n$$

```
> rangeProduct 3 6
3 * 4 * 5 * 6
360
```

Trace of `rangeProduct`

How is **`fac`** related to **`rangeProduct`**