

Functional Programming

Recursion

H. Turgut Uyar

2013-2016

1 / 24

License



© 2013-2016 H. Turgut Uyar

You are free to:

- Share – copy and redistribute the material in any medium or format
- Adapt – remix, transform, and build upon the material

Under the following terms:

- Attribution – You must give appropriate credit, provide a link to the license, and indicate if changes were made.
- NonCommercial – You may not use the material for commercial purposes.
- ShareAlike – If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

For more information:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Read the full license:

<https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode>

2 / 24

Topics

1 Recursion

- Primitive Recursion
- Tail Recursion
- Tree Recursion

2 Examples

- Exponentiation
- Counting Change
- Square Roots

3 / 24

Recursion Examples

greatest common divisor

```
gcd :: Integer -> Integer -> Integer
gcd x y = if y == 0 then x else gcd y (x `mod` y)
```

factorial

```
fac :: Integer -> Integer
fac n
  | n < 0    = error "negative parameter"
  | n == 0   = 1
  | otherwise = n * fac (n - 1)
```

4 / 24

Stack Frame Example

```
gcd x y = if y == 0 then x else gcd y (x 'mod' y)
```

```
gcd 9702 945
~> gcd 945 252
  ~> gcd 252 189
    ~> gcd 189 63
      ~> gcd 63 0
        ~> 63
      ~> 63
    ~> 63
  ~> 63
~> 63
```

5 / 24

Stack Frame Example

```
fac n
| n < 0    = error "negative parameter"
| n == 0    = 1
| otherwise = n * fac (n - 1)
```

```
fac 4
~> 4 * fac 3
  ~> 3 * fac 2
    ~> 2 * fac 1
      ~> 1 * fac 0
        ~> 1
      ~> 1
    ~> 2
  ~> 6
~> 24
```

6 / 24

Tail Recursion

- **tail recursive**: result of recursive call is also result of caller
- recursive call is last action, nothing left for caller to do
- no need to keep the stack frame, reuse frame of caller
- increased performance

7 / 24

Stack Frame Example

```
gcd x y = if y == 0 then x else gcd y (x 'mod' y)
```

```
gcd 9702 945
~> gcd 945 252
~> gcd 252 189
~> gcd 189 63
~> gcd 63 0
~> 63
```

8 / 24

Tail Recursion

- rearranging a function to be tail recursive:
- define a helper function that takes an accumulator
- base case: return accumulator
- recursive case: make recursive call with new accumulator value

9 / 24

Tail Recursion Example

tail recursive factorial

```
facIter :: Integer -> Integer -> Integer
facIter acc n
  | n < 0    = error "negative parameter"
  | n == 0   = acc
  | otherwise = facIter (acc * n) (n - 1)

fac :: Integer -> Integer
fac n = facIter 1 n
```

10 / 24

Stack Frame Example

```
facIter acc n
  | n < 0    = error "negative parameter"
  | n == 0   = acc
  | otherwise = facIter (acc * n) (n - 1)
```

```
fac 4
~> facIter 1 4
  ~> facIter 4 3
  ~> facIter 12 2
  ~> facIter 24 1
  ~> facIter 24 0
  ~> 24
~> 24
```

11 / 24

Tail Recursion Example

- helper function can be local
- negativity check only once

```
fac :: Integer -> Integer
fac n
  | n < 0    = error "negative parameter"
  | otherwise = facIter 1 n
  where
    facIter :: Integer -> Integer -> Integer
    facIter acc n'
      | n' == 0   = acc
      | otherwise = facIter (acc * n') (n' - 1)
```

12 / 24

Tree Recursion

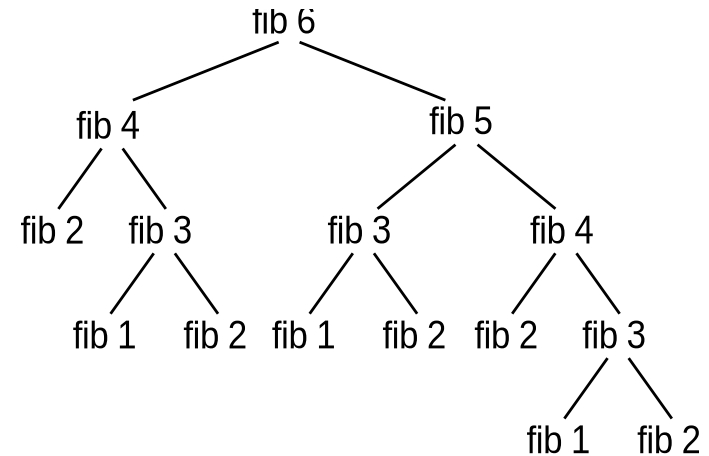
Fibonacci sequence

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib_{n-2} + fib_{n-1} & \text{if } n > 2 \end{cases}$$

```
fib :: Integer -> Integer
fib n
  | n == 1    = 1
  | n == 2    = 1
  | otherwise = fib (n - 2) + fib (n - 1)
```

13 / 24

Stack Frame Example



14 / 24

Exponentiation

```
pow :: Integer -> Integer -> Integer
pow x y
  | y == 0    = 1
  | otherwise = x * pow x (y - 1)
```

- exercise: write a tail recursive version
- to get faster, use the following definition:

$$x^y = \begin{cases} 1 & \text{if } y = 0 \\ (x^{y/2})^2 & \text{if } y \text{ is even} \\ x \cdot x^{y-1} & \text{if } y \text{ is odd} \end{cases}$$

15 / 24

Fast Exponentiation

```
pow :: Integer -> Integer -> Integer
pow x y
  | y == 0    = 1
  | even y    = sqr (pow x (y `div` 2))
  | otherwise = x * pow x (y - 1)
where
  sqr :: Integer -> Integer
  sqr n = n * n
```

16 / 24

Counting Change

- how many different ways to change given amount of money?
- into units of 1, 5, 10, 25, 50

```
countChange :: Integer -> Integer
countChange amount = cc amount 50
  where
    cc :: Integer -> Integer -> Integer
    cc a n
      | a == 0      = 1
      | a < 0 || n == 0 = 0
      | otherwise   = cc (a - n) n + cc a (next n)
```

17 / 24

Counting Change

```
next :: Integer -> Integer
next n
  | n == 50 = 25
  | n == 25 = 10
  | n == 10 = 5
  | n == 5  = 1
  | n == 1  = 0
```

18 / 24

Square Roots with Newton's Method

- start with an initial guess y (say $y = 1$)
- repeatedly improve the guess by taking the mean of y and x/y
- until the guess is good enough ($\sqrt{x} \cdot \sqrt{x} = x$)

example: $\sqrt{2}$

y	x/y	next guess
1	$2 / 1 = 2$	1.5
1.5	$2 / 1.5 = 1.333$	1.4167
1.4167	$2 / 1.4167 = 1.4118$	1.4142
1.4142

19 / 24

Square Roots with Newton's Method

```
newton :: Float -> Float -> Float
newton guess x
  | isGoodEnough guess x = guess
  | otherwise             = newton (improve guess x) x
```

```
isGoodEnough :: Float -> Float -> Bool
isGoodEnough guess x = abs (guess*guess - x) < 0.001
```

```
improve :: Float -> Float -> Float
improve guess x = (guess + x/guess) / 2.0
```

```
sqrt :: Float -> Float
sqrt x = newton 1.0 x
```

20 / 24

Square Roots with Newton's Method

```
sqrt :: Float -> Float
sqrt x = newton 1.0 x
  where
    newton :: Float -> Float -> Float
    newton guess x'
      | isGoodEnough guess x' = guess
      | otherwise              = newton (improve guess x')

    isGoodEnough :: Float -> Float -> Bool
    isGoodEnough guess x' =
      abs (guess*guess - x') < 0.001

    improve :: Float -> Float -> Float
    improve guess x' = (guess + x'/guess) / 2.0
```

21 / 24

Square Roots with Newton's Method

- doesn't work with too small and too large numbers (why?)

```
isGoodEnough guess x' =
  (abs (guess*guess - x')) / x' < 0.001
```

22 / 24

Square Roots with Newton's Method

- no need to pass x around, it's already in scope

```
sqrt x = newton 1.0
  where
    newton :: Float -> Float
    newton guess
      | isGoodEnough guess = guess
      | otherwise          = newton (improve guess)

    isGoodEnough :: Float -> Bool
    isGoodEnough guess =
      (abs (guess*guess - x)) / x < 0.001

    improve :: Float -> Float
    improve guess = (guess + x/guess) / 2.0
```

23 / 24

References

Required Reading: Thompson

- Chapter 3: **Basic types and definitions**
- Chapter 4: **Designing and writing programs**

24 / 24