Programming in Haskell: Lecture 4

S P Suresh

August 19, 2019

Recursive definitions

- **Base case**: $f(0)$ is given

Recursive definitions

- **Base case**: $f(0)$ is given
- **Inductive step**: For $n > 0$, $f(n)$ is defined in terms of f on smaller values: $f(n-1), f(n-2), \ldots, f(0)$

Recursive definitions

- **Base case**: $f(0)$ is given
- **Inductive step**: For $n > 0$, $f(n)$ is defined in terms of f on smaller values: $f(n-1), f(n-2), \ldots, f(0)$
- Factorial

Recursive definitions

- **Base case**: $f(0)$ is given
- **Inductive step**: For $n > 0$, $f(n)$ is defined in terms of f on smaller values: $f(n-1), f(n-2), \ldots, f(0)$
- Factorial
  - $0! = 1$

Recursive definitions

- **Base case**: $f(0)$ is given
- **Inductive step**: For $n > 0$, $f(n)$ is defined in terms of f on smaller values: $f(n-1), f(n-2), \ldots, f(0)$
- Factorial
  - $0! = 1$
  - For $n > 0$, $n! = n \times (n-1)!$

Example program: Factorial

- In Haskell:

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

Example program: Factorial

- In Haskell:

```haskell
factorial :: Integer -> Integer
factorial 0     = 1
factorial n     = n * factorial (n-1)
```

- `Integer` represents integers of arbitrarily large magnitude

```
fac 40
815915283247897734345611269596115894272000000000
```

Example program: Factorial

- In Haskell:

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

- `Integer` represents integers of arbitrarily large magnitude

```
fac 40
815915283247897734345611269596115894272000000000
```

  - `Int` has lower and upper bounds

Example program: Factorial

- In Haskell:

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

- `Integer` represents integers of arbitrarily large magnitude

```
fac 40
815915283247897734345611269596115894272000000000
```

  - `Int` has lower and upper bounds
  - -2^{63} and $2^{63} - 1$ on my machine

Example program: Factorial

- In Haskell:

```haskell
factorial :: Integer -> Integer
factorial 0    = 1
factorial n    = n * factorial (n-1)
```

Example program: Factorial

- In Haskell:

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

- Note the parentheses around `n-1`

Example program: Factorial

- In Haskell:

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

- Note the parentheses around `n-1`
  - `factorial n-1` is interpreted as `(factorial n) - 1`

Example program: Factorial

- In Haskell:

```haskell
factorial :: Integer -> Integer
factorial 0    = 1
factorial n    = n * factorial (n-1)
```

- Note the parentheses around `n-1`
  - `factorial n-1` is interpreted as `(factorial n) - 1`
- No guarantee of termination

Example program: Factorial

- In Haskell:

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

- Note the parentheses around `n-1`
  - `factorial n-1` is interpreted as `(factorial n) - 1`
- No guarantee of termination
- What is `factorial (-1)`?

Example program: Factorial

- In Haskell:

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

- Note the parentheses around `n-1`
  - `factorial n-1` is interpreted as `(factorial n) - 1`
- No guarantee of termination
- What is `factorial (-1)`?
  - Note the parentheses again!

Example program: Factorial

- In Haskell:

```haskell
factorial :: Integer -> Integer
factorial 0    = 1
factorial n    = n * factorial (n-1)
```

- Note the parentheses around `n-1`
  - `factorial n-1` is interpreted as `(factorial n) - 1`
- No guarantee of termination
- What is `factorial (-1)`?
  - Note the parentheses again!
  - `factorial -1` is `1` subtracted from the function `factorial`

Example program: Factorial

- In Haskell:

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

- Note the parentheses around `n-1`
  - `factorial n-1` is interpreted as `(factorial n) - 1`
- No guarantee of termination
- What is `factorial (-1)`?
  - Note the parentheses again!
  - `factorial -1` is `1` subtracted from the function `factorial`
  - Type error!

Example program: Factorial

- Fix the factorial function to work on negative values

```
        factorial :: Integer -> Integer
        factorial 0    = 1
        factorial n
            | n < 0    = n * factorial (n+1)
            | n > 0    = n * factorial (n-1)
fac (-20)
    2432902008176640000
fac (-19)
    -121645100408832000
```

Example program: gcd

- Euclid's algorithm for computing the greatest common divisor

Example program: gcd

- Euclid's algorithm for computing the greatest common divisor
- Assume a and b are not both zero

Example program: gcd

- Euclid's algorithm for computing the greatest common divisor
- Assume a and b are not both zero
- Handle negative integers appropriately

Example program: gcd

- Euclid's algorithm for computing the greatest common divisor
- Assume a and b are not both zero
- Handle negative integers appropriately
- Haskell function `gcd`

```
gcd :: Integer -> Integer -> Integer
gcd a b
    | a < 0 || b < 0     = gcd (abs a) (abs b)
    | b == 0             = a
    | otherwise          = gcd b (a `mod` b)
```

Example program: gcd

- The built-in function `abs :: Integer -> Integer` returns the absolute value of an integer

Example program: gcd

- The built-in function `abs :: Integer -> Integer` returns the absolute value of an integer
- The built-in function `mod :: Integer -> Integer -> Integer` returns the mod value

Example program: gcd

- The built-in function `abs :: Integer -> Integer` returns the absolute value of an integer
- The built-in function `mod :: Integer -> Integer -> Integer` returns the mod value
- Binary functions can be used as infix operators by enclosing them inside **backticks** – like `` `mod` ``

Example program: gcd

- The built-in function `abs :: Integer -> Integer` returns the absolute value of an integer
- The built-in function `mod :: Integer -> Integer -> Integer` returns the mod value
- Binary functions can be used as infix operators by enclosing them inside **backticks** – like `` `mod` ``
- On the other hand, infix operators can be used in prefix form by enclosing in parentheses – like `(+) 5 3`

Aside: sectioning

- `(+)` works on two numbers

Aside: sectioning

- `(+)` works on two numbers
- What is `(+) n`?

Aside: sectioning

- `(+)` works on two numbers
- What is `(+) n`?
- It is a function that adds `n` to any input it receives

Aside: sectioning

- `(+)` works on two numbers
- What is `(+) n`?
- It is a function that adds `n` to any input it receives
- Special syntax in Haskell: `(n+)`

Aside: sectioning

- `(+)` works on two numbers
- What is `(+) n`?
- It is a function that adds `n` to any input it receives
- Special syntax in Haskell: `(n+)`
- Fixes the first argument: `(5+) 3 = 8`

Aside: sectioning

- `(+)` works on two numbers
- What is `(+) n`?
- It is a function that adds `n` to any input it receives
- Special syntax in Haskell: `(n+)`
- Fixes the first argument: `(5+) 3 = 8`
- `(+n)` fixes the second argument: `(+5) 8 = 13`

Aside: sectioning

- `(+)` works on two numbers
- What is `(+) n`?
- It is a function that adds `n` to any input it receives
- Special syntax in Haskell: `(n+)`
- Fixes the first argument: `(5+) 3 = 8`
- `(+n)` fixes the second argument: `(+5) 8 = 13`
- Expressions like `(+5)` and `(3+)` are called **sections**

Aside: sectioning

- Can be applied to other binary operators too

Aside: sectioning

- Can be applied to other binary operators too
- `(*8) 3 = 24`

Aside: sectioning

- Can be applied to other binary operators too
- `(*8) 3 = 24`
- `(8/) 3 = 2.6666666666666665`

Aside: sectioning

- Can be applied to other binary operators too
- `(*8) 3 = 24`
- `(8/) 3 = 2.6666666666666665`
- `(/8) 3 = 0.375`

Aside: sectioning

- Can be applied to other binary operators too
- `(*8) 3 = 24`
- `(8/) 3 = 2.6666666666666665`
- `(/8) 3 = 0.375`
- `(8-) 3 = 5`

Aside: sectioning

- Can be applied to other binary operators too
- `(*8) 3 = 24`
- `(8/) 3 = 2.6666666666666665`
- `(/8) 3 = 0.375`
- `(8-) 3 = 5`
- `(- 8) 3` does not work, though! Interpreted as a negative number, not a section

Aside: sectioning

- Can be applied to other binary operators too
- `(*8) 3 = 24`
- `(8/) 3 = 2.6666666666666665`
- `(/8) 3 = 0.375`
- `(8-) 3 = 5`
- `(- 8) 3` does not work, though! Interpreted as a negative number, not a section
- Use `subtract` instead

Aside: sectioning

- Can be applied to other binary operators too
- `(*8) 3 = 24`
- `(8/) 3 = 2.6666666666666665`
- `(/8) 3 = 0.375`
- `(8-) 3 = 5`
- `(- 8) 3` does not work, though! Interpreted as a negative number, not a section
- Use `subtract` instead
- `(subtract 8) 3 = -5`

Example program: largest divisor

- Find the largest divisor of n, other than n itself

Example program: largest divisor

- Find the largest divisor of n, other than n itself
- **Strategy**: try $n-1, n-2, \ldots$

Example program: largest divisor

- Find the largest divisor of n, other than n itself
- **Strategy**: try $n-1, n-2, \ldots$
- In the worst case, stop at 1

Example program: largest divisor

- Find the largest divisor of n, other than n itself
- **Strategy**: try $n-1, n-2, \ldots$
- In the worst case, stop at 1
- Haskell function `largestDiv`

```
largestDiv :: Integer -> Integer
largestDiv n = divSearch n (n-1)

divSearch :: Integer -> Integer -> Integer
divSearch m i
    | m `mod` i == 0    = i
    | otherwise         = divSearch m (i-1)
```

Local definitions

- `divSearch` is a helper function

Local definitions

- `divSearch` is a helper function
- No need to invoke it independently

Local definitions

- `divSearch` is a helper function
- No need to invoke it independently
- We can make the definition **local**

```
largestDiv :: Integer -> Integer
largestDiv n = divSearch n (n-1)
    where
        divSearch :: Integer -> Integer -> Integer
        divSearch m i
            | m `mod` i == 0    = i
            | otherwise         = divSearch m (i-1)
```

Local definitions

- Local functions can use names defined in the surrounding context

Local definitions

- Local functions can use names defined in the surrounding context
- The first argument of `divSearch`, `m`, never changes

Local definitions

- Local functions can use names defined in the surrounding context
- The first argument of `divSearch`, `m`, never changes
- It is in fact the argument of `largestDiv`

Local definitions

- Local functions can use names defined in the surrounding context
- The first argument of `divSearch`, `m`, never changes
- It is in fact the argument of `largestDiv`
- Simplified `divSearch`:

```
largestDiv :: Integer -> Integer
largestDiv n = divSearch (n-1)
    where
        divSearch :: Integer -> Integer
        divSearch i
            | n `mod` i == 0    = i
            | otherwise         = divSearch (i-1)
```

Local definitions

- Can also use `let` to define local functions

```
largestDiv :: Integer -> Integer
largestDiv n = let divSearch i
                     | n `mod` i == 0  = i
                     | otherwise       = divSearch (i-1)
               in
                   divSearch (n-1)
```

Local definitions

- Reduce the search space:

```
largestDiv :: Integer -> Integer
largestDiv n = let divSearch i
                     | n `mod` i == 0  = i
                     | otherwise       = divSearch (i-1)
               in
                  divSearch $ n `div` 2
```

Local definitions

- Reduce the search space:

```
largestDiv :: Integer -> Integer
largestDiv n = let divSearch i
                     | n `mod` i == 0  = i
                     | otherwise       = divSearch (i-1)
               in
                   divSearch $ n `div` 2
```

- `divSearch $ n `div` 2` is equivalent to `divSearch (n `div` 2)`

Local definitions

- Reduce the search space:

```
largestDiv :: Integer -> Integer
largestDiv n = let divSearch i
                     | n `mod` i == 0  = i
                     | otherwise       = divSearch (i-1)
               in
                   divSearch $ n `div` 2
```

- `divSearch $ n `div` 2` is equivalent to `divSearch (n `div` 2)`
- `$` helps reduce clutter involving nested parentheses:

Local definitions

- Reduce the search space:

```
largestDiv :: Integer -> Integer
largestDiv n = let divSearch i
                     | n `mod` i == 0  = i
                     | otherwise       = divSearch (i-1)
               in
                   divSearch $ n `div` 2
```

- `divSearch $ n `div` 2` is equivalent to `divSearch (n `div` 2)`
- `$` helps reduce clutter involving nested parentheses:
- `f $ g $ h $ x+1` instead of `f (g (h (x+1)))`

Example: length of an integer

- The number of digits in a non-negative integer n

Example: length of an integer

- The number of digits in a non-negative integer n
- If $n < 10$, there is just one digit

Example: length of an integer

- The number of digits in a non-negative integer n
- If $n < 10$, there is just one digit
- Otherwise, determine the number of digits in n div 10 and add 1

Example: length of an integer

- The number of digits in a non-negative integer n
- If $n < 10$, there is just one digit
- Otherwise, determine the number of digits in $n \text{ div } 10$ and add 1
- Haskell function `intLength`

```
intLength :: Integer -> Integer
intLength n
    | n < 0         = 0
    | n < 10        = 1
    | otherwise     = 1 + intLength (n `div` 10)
```

Example: reverse a number

- `intReverse n` reverses the digits of `n` (non-negative)

Example: reverse a number

- `intReverse n` reverses the digits of `n` (non-negative)
- `intReverse 13276` should give `67231`

Example: reverse a number

- `intReverse n` reverses the digits of `n` (non-negative)
- `intReverse 13276` should give `67231`
- **Strategy**:

Example: reverse a number

- `intReverse n` reverses the digits of `n` (non-negative)
- `intReverse 13276` should give `67231`
- Strategy:
  - Split `13276` as `1327` and `6` using **`div`** and **`mod`**

Example: reverse a number

- `intReverse n` reverses the digits of `n` (non-negative)
- `intReverse 13276` should give `67231`
- Strategy:
  - Split `13276` as `1327` and `6` using **`div`** and **`mod`**
  - Recursively reverse `1327` to get `7231`

Example: reverse a number

- `intReverse n` reverses the digits of `n` (non-negative)
- `intReverse 13276` should give `67231`
- Strategy:
  - Split `13276` as `1327` and `6` using **`div`** and **`mod`**
  - Recursively reverse `1327` to get `7231`
  - Multiply `6` by a suitable power of `10` and add: `60000 + 7231 = 67231`

Example: reverse a number

- `intReverse n` reverses the digits of `n` (non-negative)
- `intReverse 13276` should give `67231`
- Strategy:
  - Split `13276` as `1327` and `6` using **`div`** and **`mod`**
  - Recursively reverse `1327` to get `7231`
  - Multiply `6` by a suitable power of `10` and add: `60000 + 7231 = 67231`
  - Use `intLength` to determine the power of `10`

Example: reverse a number

```
intReverse :: Integer -> Integer
intReverse n
    | n < 10        = n
    | otherwise     = intReverse (n `div` 10) +
                      (n `mod` 10) *
                      power 10 (intLength n - 1)

power :: Integer -> Integer -> Integer
power m 0  = 1
power m n  = m * power m (n-1)
```