Computer Science 312

Making choices

Tail Recursion

Local contexts with where clauses

Dealing with Errors

Modules, Imports, Exports

Making Choices in Haskell

Use a set of function clauses, where pattern matching selects the option (**`factorial`**, **`fibonacci`**, etc.)

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

fibonacci:: Integer -> Integer
fibonacci 1 = 1
fibonnaci 2 = 1
fibonacci n = fibonacci (n – 1) + fibonacci (n – 2)
```

Making Choices in Haskell

Simple function clauses will not work in the cases of functions like **`max`**, **`min`**, and **`sum`**, where the option depends on an explicit condition

```
Prelude> max 3 4
4

Prelude> sum 1 4
10
```

Use a Function Guard

```
<function name> <parameters>
    | <Boolean expression-1> = <expression-1>
    …
    | <Boolean expression-n> = <expression-n>
    | otherwise = <default expression>
```

```
myMax :: Ord a => a -> a -> Bool
myMax x y
    | x > y = x
    | otherwise y

sum :: Integer -> Integer -> Integer
sum lower upper
    | lower == upper = lower
    | otherwise = lower + sum (lower + 1) upper
```

Indentation is significant!

Or Use an **`if-else`** Expression

```
<function name> <parameters> =
    if <Boolean expression-1> then
        <expression-1>
    …
    else if <Boolean expression-n> then
        <expression-n>
    else
        <default expression>
```

```
myMax :: Ord a => a -> a -> Bool
myMax x y =
    if x > y then
        x
    else
        y
```

Syntactic sugar for function guards

Which Form to Prefer?

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

```
factorial :: Integer -> Integer
factorial n
    | n == 1 = 1
    | otherwise = n * factorial (n – 1)
```

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

How Costly Is Recursion?

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

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

Factorial is O(n) in running time and memory

Fibonacci is O(k^n) in running time and O(log$n$) in memory

The growth of memory is due to cells for each function call pushed onto the runtime stack

Tail Recursion

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

```
tailFactorial :: Integer -> Integer -> Integer
tailFactorial 1 result = result
tailFactorial n result = tailFactorial (n – 1) (n * result)
```

In tail recursion, there is no work remaining to be done after a recursive call

The compiler can translate this to a loop with a single record for state variables on the stack

Tracing the Two Versions

```
factorial 3 ->
    3 * factorial 2 ->
        2 * factorial 1 ->
            <- 1
        <- 2
    <- 6
```

```
tailFactorial 3 1 ->
    tailFactorial 2 3 ->
        tailFactorial 1 6 ->
            <- 6
        <- 6
    <- 6
```

Maintain the Interface with a Helper

```
factorial :: Integer -> Integer
factorial n = tailFactorial n 1

tailFactorial :: Integer -> Integer -> Integer
tailFactorial 1 result = result
tailFactorial n result = tailFactorial (n – 1) (result * n)
```

Nest the Helper in a **`where`** Clause

```
factorial :: Integer -> Integer
factorial n = tailFactorial n 1
    where
    tailFactorial :: Integer -> Integer -> Integer
    tailFactorial 1 result = result
    tailFactorial n result = tailFactorial (n – 1) (result * n)
```

If a function is just a helper for another function, you can define it locally within a **`where`** clause

The scope of the **`where`** is determined by indentation

Nest Function in a **`where`** Clause

```
factorial :: Integer -> Integer
factorial n = tailFactorial n 1
    where
    tailFactorial :: Integer -> Integer -> Integer
    tailFactorial 1 result = result
    tailFactorial n result = tailFactorial (n – 1) (result * n)
```

```
fib :: Integer -> Integer
fib n = tailFib 1 0 n
    where
    tailFib:: Integer -> Integer -> Integer
    tailFib a b 1 = a
    tailFib a b n = tailFib (a + b) a (n – 1)
```

Both functions are now linear in running time and constant in memory usage

For next time

Introduction to lists, strings and tuples