Week 2

Ch 1: Starting Out
Ch 2: Believe the Type

University of the Fraser Valley

Dr. Russell Campbell

Russell.Campbell@ufv.ca

COMP 481: Functional and Logic Programming

1

Terminology

- referential transparency
- lazy
- statically typed
- type inference

2

— Math Operations —

3

Operators

4

Inline vs Prefix

Take a moment to read the error message when attempting to sum two values of different types, e.g.: `5 + "llama"`.

- the operations written between two values (such as `*`):
- are considered functions with two parameters
- are described as "inline" when written in between its arguments
- can be written in prefix style, i.e.: `(*) 2 3` evaluates to `6`

5

More Math Functions

6

— Lists —

7

List
Functions

8

List Creation with Ranges

Examples:

- `[1..20]`
- `[2,4..20]`
- `[20,19..1]`
- `take 24 [13,26..]`
- `take 10 (cycle [1,2,3])`
- `take 12 (cycle "LOL ")`
- `take 10 (repeat 5)`
- `replicate 3 10`
- watch out with using ranges and floating-point accuracy
- `[0.1, 0.3 .. 1]`

9

List Comprehensions

10

List Access

- use `!!` to access one element using an index
- `[1, 2, 3] !! 0`
- again, strings are also lists of characters:
- `"hello" !! 4`

We can also have lists inside of lists:

- `[ [1,2,3], [4,5,6], [7,8,9] ]`

11

— Tuples —

12

Tuple Types

- different numbers of elements are treated as distinct types
  - so, a 2-tuple is considered different type from a 3-tuple
- for tuples with different types in corresponding elements are altogether different types as well
  - e.g.: `(1, 'a')` different type from `('a', 1)`
- can compare elements of the same type
- cannot compare tuples of different lengths

13

Tuple Functions

- `fst`
- `snd`

- `zip`

14

— Chapter 2 —

15

GHCI Multiline

```
:{   :}
:set +m
```

16

Tuple Types

Every thing in Haskell has a type determined at compile time:

- you do not need to explicitly declare types
  if there is enough other information for Haskell
- `::` reads as "has type of"
  - e.g.: `(True, 'a') :: (Bool, Char)`
- `:t` lets us check the type of whatever follows the command

17

Common Types

`Int`

`Integer`

`Float`

`Double`

`Bool` `True` `False`

`Char`

`[Char]` `String`

`()`

18

Type Variables

The type of a function could involve lists, but lists themselves could contain some arbitrary type:

- so, the declaration will involve a placeholder for the element type, such as `a`
  - e.g.: `:t head;` `head :: [a] -> a`
  - e.g.: `:t fst;` `fst :: (a, b) -> a`

Note that the type of `fst` function describes the first element in the pair `a` to have the same type as the return value of the `fst` function.

19

Intro to Type Classes

```
:t (==)
                    =>          class constraint
                a               Eq

                Eq
```

20

Eq & Ord Type Classes

The `Eq` type class supports equality testing.

- so if a function has an `Eq` class constraint for one of its type variables, then that function must implement BOTH `==` and `/=` within its definition
- "definition" means the function's statements of execution

The `Ord` type class is used by types that need arrange their values in some order

- try `:t (>)`
- the `compare` function takes two input values both with type an instance of `Ord`
  - the return type is `Ordering`
  - `Ordering` is a type with values `GT`, `LT`, `EQ`

21

Show and Ord Type Class

Show

show

Ord

:t (>)

22

Read Type Class

All types we have seen so far except functions are instances of the `Read` type class as well.

- the `read` function (inverse of the `show` function)
- `read` takes a String type value as input and returns what value would be expected when used in context

For example: `` `read "True" || False` ``

- the above context would expect a value of type `` `Bool` `` in place of the `` `read "True"` `` expression
- `` `read "4"` `` will result in an error, because the expression is not used in any context, so it does not know what type to expect

23

Type Annotations

type annotations

```
::
read "5" :: Int
(read "5" :: Float) * 4


[read "True", False, True]
```

24

Enum Type Class

The `Enum` type class describes any type that has values which are totally ordered:

- the advantage is to be able to specify list ranges with `..`
- its `pred` function will return the value that directly precedes its input value in the total order
- its `succ` function will return the next value directly after it input value in the total order

Examples of types in this type class:

- `(), Bool, Char, Ordering, Int, Integer, Float, Double`
- try the above in creating a few lists

25

The Bounded Type Class

Bounded

maxBound

- :t maxBound

maxBound

Bounded

Bounded

26

The Num Type Class

Look at `:t 20`

- similar type as `maxBound`, but with type class `Num` instead
- so a value such as `20` is also a polymorphic constant
  - "can act like any type that's an instance of the `Num` type class"
  - `Int, Integer, Float, Double`

Try `t: 20 :: Double`

- consider the multiplication operator `:t (*)`
  - accepts two numbers and returns a number of the same type
  - e.g.: `(5 :: Int) * (6 :: Integer)` will cause a type error
  - e.g.: `5 * (6 :: Integer)` has no such error
  - `5` can act like either an `Int` or an `Integer`, but not both at once
- to be an instance of `Num`, a type must also be an instance of `Show` and `Eq`

27

The Floating Type Class

Float Double

:t

sin

cos

sqrt

28

The `Integral` Type Class

- envelopes the `` `Int` `` and `` `Integer` `` types
- only whole numbers

An example with more than one class constraint:

- ```
  :t fromIntegral
  fromIntegral :: (Integral a, Num b) => a -> b
  ```
- returns a more general type for the same value
- you can use `` `fromIntegral` `` to smoothly combine expressions that use mixed numeric types
  - e.g.: `` `fromIntegral (length [1,2,3,4]) + 3.2` ``

29

Converting Float Values to Int

30
  ```

— Finishing Up —

31

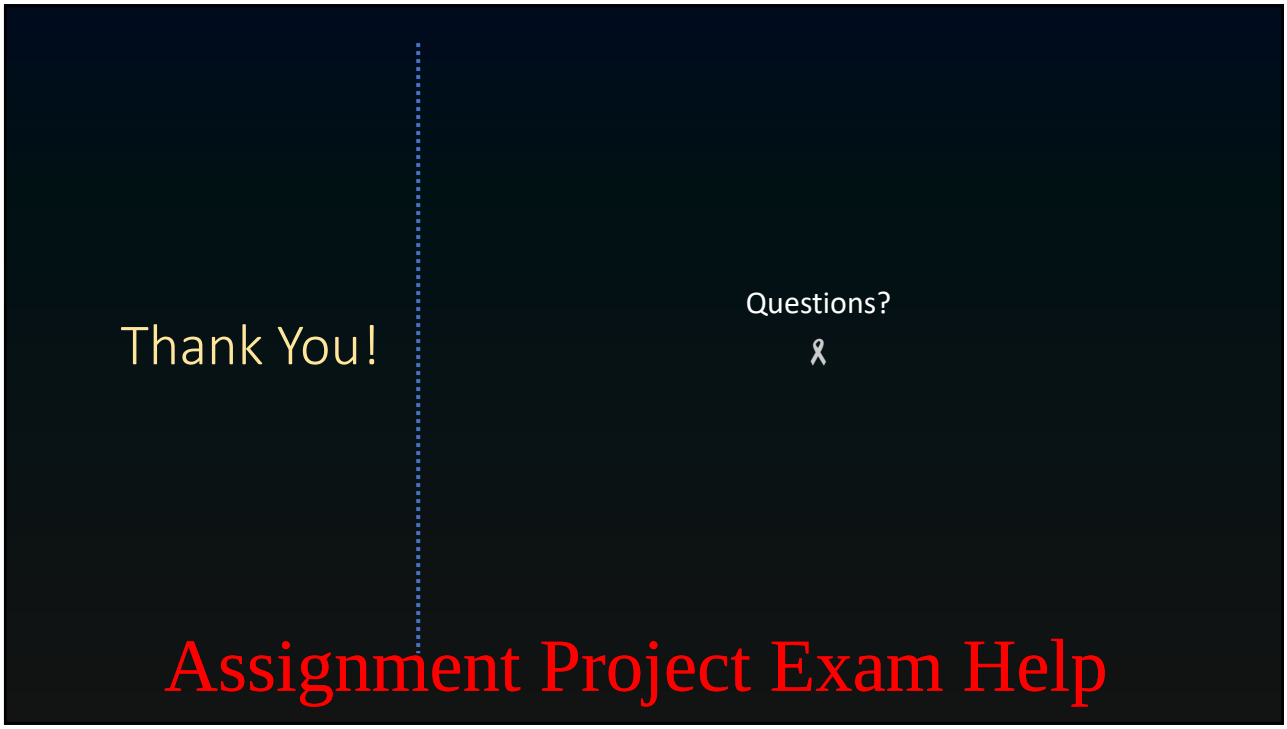
Final Remarks

Ord

Eq

prerequisite

32



33