Adapted from Graham's Lecture slides.

第三章： 类型和类族

基本概念， 基本类型，
组合类型 (List, Tuple, Function),
多态类型， 基本类族

What is a Type?

A type is a collection of related values. For example, in Haskell the basic type

```
Bool
```

contains the two logical values:

```
False        True
```

Type Errors

Applying a function to one or more arguments of the wrong type is called a <u>type error</u>.

```
> 1 + False
error ...
```

1 is a number and False is a logical value, but + requires two numbers.

Types in Haskell

- If evaluating an expression e would produce a value of type t, then e has type t, written

```
e :: t
```

- Every well formed expression has a type, which can be automatically calculated at compile time using a process called type inference.

$$\frac{f :: A \to B \quad e :: A}{f\ e :: B}$$

- All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.

- In GHCi, the :type command calculates the type of an expression, without evaluating it:

```
> not False
True

> :type not False
not False :: Bool
```

Basic Types

Haskell has a number of basic types, including:

`Bool` - logical values

`Char` - single characters

`String` - strings of characters

Basic Types

Haskell has a number of basic types, including:

`Int` - *fix-precision* integer numbers. GHC: [-2^63, 2^63-1]

Evaluating `2^63 :: Int` gives a negtive number

`Integer` - *arbitrary-precision* integer numbers.

Evaluating `2^63 :: Integer` gives the correct result

`Word`
- *fix-precision unsigned* integer numbers.
- the same size with `Int`

`Natural`
- *arbitrary-precision unsigned* integer numbers.
- defined in the module Numeric.Natural (located in base package)

Basic Types

Haskell has a number of basic types, including:

`Float` - *single-precision* floating-point numbers

Evaluating `sqrt 2 :: Float` gives `1.4142135`

`Double` - *double-precision* floating-point numbers

Evaluating `sqrt 2 :: Double` gives `1.4142135623730951`

List Types

A list is a sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a','b','c','d'] :: [Char]
```

In general:

[t] is the type of lists with elements of type t.

Note:

- The type of a list says nothing about its length:

```
[False,True] :: [Bool]

[False,True,False] :: [Bool]
```

- The type of the elements is unrestricted. For example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```

Tuple Types

A tuple is a sequence of values of possibly-different types:

```
(False,True) :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)
```

In general:

(t1,t2,…,tn) is the type of n-tuples whose ith components have type ti for any i in 1…n.

Note:

- The type of a tuple encodes its size:

```
(False,True) :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)
```

- The type of the components is unrestricted:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b']) :: (Bool,[Char])
```

Function Types

A function is a mapping from values of one type to values of another type:

```
not  :: Bool → Bool

even :: Int → Bool
```

In general:

t1 → t2 is the type of functions that map values of type t1 to values to type t2.

Note:

- The arrow → is typed at the keyboard as ->.

- The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

```
add :: (Int,Int) → Int
add (x,y) = x+y

zeroto :: Int → [Int]
zeroto n = [0..n]
```

Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

```
add’ :: Int → (Int → Int)
add’ x y = x+y
```

add’ takes an integer x and returns a function add’ x. In turn, this function takes an integer y and returns the result x+y.

Note:

- add and add’ produce the same final result, but add takes its two arguments at the same time, whereas add’ takes them one at a time:

```
add  :: (Int,Int) → Int

add’  :: Int → (Int → Int)
```

- Functions that take their arguments one at a time are called <u>curried</u> functions, celebrating the work of Haskell Curry on such functions.

- Functions with more than two arguments can be curried by returning nested functions:

```
mult :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult takes an integer x and returns a function <u>mult x</u>, which in turn takes an integer y and returns a function <u>mult x y</u>, which finally takes an integer z and returns the result x*y*z.

Why is Currying Useful?

Curried functions are more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function.

For example:

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

Currying Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

- The arrow $\rightarrow$ associates to the right.

```
Int → Int → Int → Int
```

Means Int $\rightarrow$ (Int $\rightarrow$ (Int $\rightarrow$ Int)).

- As a consequence, it is then natural for function application to associate to the left.

```
mult x y z
```

Means ((mult x) y) z.

Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables.

```
length :: [a] → Int
```

For any type a, length takes a list of values of type a and returns an integer.

Note:

- Type variables can be instantiated to different types in different circumstances:

```
> length [False,True]
2

> length [1,2,3,4]
4
```

a = Bool

a = Int

- Type variables must begin with a lower-case letter, and are usually named a, b, c, etc.

- Many of the functions defined in the standard prelude are polymorphic. For example:

```
fst :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip :: [a] → [b] → [(a,b)]

id :: a → a
```

Overloaded Functions

A polymorphic function is called <u>overloaded</u> if its type contains one or more class constraints.

```
(+) :: Num a ⇒ a -> a -> a
```

For any numeric type a, (+) takes two values of type a and returns a value of type a.

Note:

- Constrained type variables can be instantiated to any types that satisfy the constraints:

```
> 1 + 2
3

> 1.0 + 2.0
3.0

> 'a' + 'b'
ERROR
```

a = Int

a = Float

Char is not a numeric type

- Haskell has a number of type classes, including:

`Num` - Numeric types

`Eq` - Equality types

`Ord` - Ordered types

- For example:

```
(+)  :: Num a ⇒ a → a → a
(==) :: Eq a  ⇒ a → a → Bool
(<)  :: Ord a ⇒ a → a → Bool
```

Eq： equality types

(以下文字来自Prelude官方说明文档)

https://hackage.haskell.org/package/base-4.15.0.0/docs/Prelude.html#t:Eq

```
class Eq a where
```

Source

The Eq class defines equality (==) and inequality (/=). All the basic datatypes exported by the Prelude are instances of Eq, and Eq may be derived for any datatype whose constituents are also instances of Eq.

The Haskell Report defines no laws for Eq. However, == is customarily expected to implement an equivalence relationship where two values comparing equal are indistinguishable by "public" functions, with a "public" function being one not allowing to see implementation details. For example, for a type representing non-normalised natural numbers modulo 100, a "public" function doesn't make the difference between 1 and 201. It is expected to have the following properties:

Reflexivity

`x == x` = `True`

Symmetry

`x == y` = `y == x`

Transitivity

if `x == y && y == z` = `True`, then `x == z` = `True`

Substitutivity

if `x == y` = `True` and `f` is a "public" function whose return type is an instance of `Eq`, then `f x == f y` = `True`

Negation

`x /= y` = `not (x == y)`

Eq： equality types

(以下文字来自Prelude官方说明文档)

https://hackage.haskell.org/package/base-4.15.0.0/docs/Prelude.html#t:Eq

Minimal complete definition

```
(==) | (/=)
```

Methods

```
(==) :: a -> a -> Bool    infix 4
```

```
(/=) :: a -> a -> Bool    infix 4
```

Ord

https://hackage.haskell.org/package/base-4.15.0.0/docs/Prelude.html#t:Ord

```
class Eq a => Ord a where
```

Source

The `Ord` class is used for totally ordered datatypes.

Instances of `Ord` can be derived for any user-defined datatype whose constituent types are in `Ord`. The declared order of the constructors in the data declaration determines the ordering in derived `Ord` instances. The `Ordering` datatype allows a single comparison to determine the precise ordering of two objects.

The Haskell Report defines no laws for `Ord`. However, `<=` is customarily expected to implement a non-strict partial order and have the following properties:

Transitivity
: if `x <= y && y <= z` = `True`, then `x <= z` = `True`

Reflexivity
: `x <= x` = `True`

Antisymmetry
: if `x <= y && y <= x` = `True`, then `x == y` = `True`

Note that the following operator interactions are expected to hold:

1. `x >= y` = `y <= x`
2. `x < y` = `x <= y && x /= y`
3. `x > y` = `y < x`
4. `x < y` = `compare x y == LT`
5. `x > y` = `compare x y == GT`
6. `x == y` = `compare x y == EQ`
7. `min x y == if x <= y then x else y` = `True`
8. `max x y == if x >= y then x else y` = `True`

Note that (7.) and (8.) do *not* require `min` and `max` to return either of their arguments. The result is merely required to *equal* one of the arguments in terms of `(==)`.

Ord

https://hackage.haskell.org/package/base-4.15.0.0/docs/Prelude.html#t:Ord

Minimal complete definition: either `compare` or `<=`. Using `compare` can be more efficient for complex types.

Minimal complete definition

```
compare | (<=)
```

Methods

```
compare :: a -> a -> Ordering                # Source
```

```
(<) :: a -> a -> Bool    | infix 4 |         # Source
```

```
(<=) :: a -> a -> Bool   | infix 4 |         # Source
```

```
(>) :: a -> a -> Bool    | infix 4 |         # Source
```

```
(>=) :: a -> a -> Bool   | infix 4 |         # Source
```

```
max :: a -> a -> a                           # Source
```

```
min :: a -> a -> a                           # Source
```

```
data Ordering
```

Constructors

- **LT**
- **EQ**
- **GT**

Num： basic numeric class

(以下文字来自Prelude官方说明文档)

https://hackage.haskell.org/package/base-4.15.0.0/docs/Prelude.html#t:Num

```
class Num a where
```

Source

Basic numeric class.

The Haskell Report defines no laws for `Num`. However, `(+)` and `(*)` are customarily expected to define a ring and have the following properties:

Associativity of `(+)`

`(x + y) + z = x + (y + z)`

Commutativity of `(+)`

`x + y = y + x`

`fromInteger 0` is the additive identity

`x + fromInteger 0 = x`

`negate` gives the additive inverse

`x + negate x = fromInteger 0`

Associativity of `(*)`

`(x * y) * z = x * (y * z)`

`fromInteger 1` is the multiplicative identity

`x * fromInteger 1 = x` and `fromInteger 1 * x = x`

Distributivity of `(*)` with respect to `(+)`

`a * (b + c) = (a * b) + (a * c)` and `(b + c) * a = (b * a) + (c * a)`

Note that it *isn't* customarily expected that a type instance of both `Num` and `Ord` implement an ordered ring. Indeed, in `base` only `Integer` and `Rational` do.

Methods

```
(+) :: a -> a -> a    infixl 6
```
\# Source

```
(-) :: a -> a -> a    infixl 6
```
\# Source

```
(*) :: a -> a -> a    infixl 7
```
\# Source

```
negate :: a -> a
```
\# Source

Unary negation.

Minimal complete definition

```
(+), (*), abs, signum, fromInteger, (negate | (-))
```

```
abs :: a -> a
```

Absolute value.

```
signum :: a -> a
```

Sign of a number. The functions `abs` and `signum` should satisfy the law:

```
abs x * signum x == x
```

For real numbers, the `signum` is either `-1` (negative), `0` (zero) or `1` (positive).

```
fromInteger :: Integer -> a
```
\# Source

Conversion from an `Integer`. An integer literal represents the application of the function `fromInteger` to the appropriate value of type `Integer`, so such literals have type `(Num a) => a`.

Hints and Tips

- When defining a new function in Haskell, it is useful to begin by writing down its type;
- Within a script, it is good practice to state the type of every new function defined;
- When stating the types of polymorphic functions that use numbers, equality or orderings, take care to include the necessary class constraints.

作业

3-1 What are the types of the following values?

```
['a','b','c']

('a','b','c')

[(False,'0'),(True,'1')]

([False,True],['0','1'])

[tail,init,reverse]
```

3-2 What are the types of the following functions?

```
second xs = head (tail xs)

swap (x,y) = (y,x)

pair x y = (x,y)

double x = x*2

palindrome xs = reverse xs == xs

twice f x = f (f x)
```

and check your answers using GHCi.

3-3 阅读教科书，用例子（在ghci上运行）展示Int与Integer的区别以及show和read的用法。

3-4 阅读教科书以及Prelude模块的官方文档，理解Integral 和 Fractional 两个 Type Class中定义的函数/操作符，用例子（在ghci上运行）展示每一个函数/操作符的用法。