Adapted from Graham's Lecture slides.

第八章：类型和类族的定义

类型定义，数据定义

递归类型，类族和例化

命题真伪判断问题，抽象机及编译

Type Declarations

In Haskell, a new name for an existing type can be defined using a type declaration.

```
type String = [Char]
```

String is a synonym for the type [Char].

Type declarations can be used to make other types easier to read. For example, given

```
type Pos = (Int,Int)
```

we can define:

```
origin :: Pos
origin = (0,0)

left :: Pos → Pos
left (x,y) = (x-1,y)
```

Like function definitions, type declarations can also have parameters. For example, given

```
type Pair a = (a,a)
```

we can define:

```
mult :: Pair Int → Int
mult (m,n) = m*n

copy :: a → Pair a
copy x = (x,x)
```

Type declarations can be nested:

```
type Pos = (Int,Int)

type Trans = Pos → Pos
```

✓

However, they cannot be recursive:

```
type Tree = (Int,[Tree])
```

✗

Data Declarations

A completely new type can be defined by specifying its values using a <u>data declaration</u>.

```
data Bool = False | True
```

Bool is a new type, with two new values False and True.

Note:

- The two values False and True are called the constructors for the type Bool.

- Type and constructor names must always begin with an upper-case letter.

- Data declarations are similar to context free grammars. The former specifies the values of a type, the latter the sentences of a language.

Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers :: [Answer]
answers = [Yes,No,Unknown]

flip :: Answer → Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
square :: Float → Shape
square n = Rect n n

area :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Note:

- Shape has values of the form Circle r where r is a float, and Rect x y where x and y are floats.

- Circle and Rect can be viewed as functions that construct values of type Shape:

```
circle :: Float → Shape

rect :: Float → Float → Shape
```

Not surprisingly, data declarations themselves can also have parameters. For example, given

```
data Maybe a = Nothing | Just a
```

we can define:

```
safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, types can be recursive.

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat → Nat.

Note:

- A value of type Nat is either Zero, or of the form Succ n where n :: Nat. That is, Nat contains the following infinite sequence of values:

```
Zero

Succ Zero

Succ (Succ Zero)
```

⋮

- We can think of values of type Nat as natural numbers, where Zero represents 0, and Succ represents the successor function `1+`.

- For example, the value

```
Succ (Succ (Succ Zero))
```

represents the natural number

```
1 + (1 + (1 + 0))
```

=

```
3
```

Using recursion, it is easy to define functions that convert between values of type Nat and Int:

```
nat2int :: Nat → Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int → Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Two naturals can be added by converting them to integers, adding, and then converting back:

```
add :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

However, using recursion the function add can be defined without the need for conversions:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

For example:

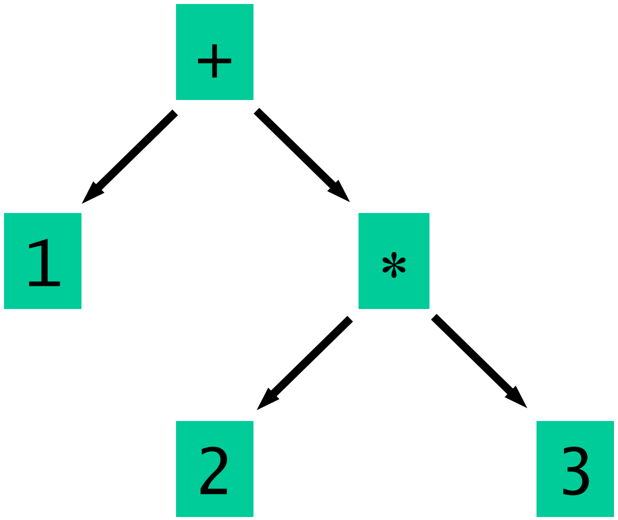
```
    add (Succ (Succ Zero)) (Succ Zero)
=
    Succ (add (Succ Zero) (Succ Zero))
=
    Succ (Succ (add Zero (Succ Zero))
=
    Succ (Succ (Succ Zero))
```

Note:

- The recursive definition for add corresponds to the laws 0+n = n and (1+m)+n = 1+(m+n).

Arithmetic Expressions

Consider a simple form of expressions built up from integers using addition and multiplication.



Using recursion, a suitable new type to represent such expressions can be declared by:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

For example, the expression on the previous slide would be represented as follows:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Using recursion, it is now easy to define functions that process expressions. For example:

```
size :: Expr → Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval :: Expr → Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Note:

- The three constructors have types:

```
Val :: Int → Expr
Add :: Expr → Expr → Expr
Mul :: Expr → Expr → Expr
```

- Many functions on expressions can be defined by replacing the constructors by other functions using a suitable <u>fold</u> function. For example:

```
eval = folde id (+) (*)
```

Newtype Declarations

If a new type has a single constructor with a single argument, then it can also be declared using the newtype mechanism.

```
newtype Nat = N Int
```

Comparison:

data Nat = N Int — Less efficient

type Nat = Int — Less safe

Using newtype helps improve type safety, without affecting performance.

Class and instance declarations

We now turn our attention from types to classes. In Haskell, a new class can be declared using the class mechanism.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

For a type a to be an instance of the class Eq, it must support equality and inequality operators of the specified types.

Class and instance declarations

The type Bool can be made into an equality type as follows:

```
instance Eq Bool where
  False == False = True
  True == True   = True
  _ == _         = False
```

Note:

- Only types that are declared using the **data** and **newtype** mechanisms can be made into instances of classes.
- Default definitions can be overridden in instance declarations if desired.

Class and instance declarations

Classes can also be extended to form new classes.

```
class Eq a => Ord a where
   (<), (<=), (>), (>=) :: a -> a -> Bool
   min, max             :: a -> a -> a

   min x y | x <= y    = x
           | otherwise = y

   max x y | x <= y    = y
           | otherwise = x
```

```
instance Ord Bool where
    False < True = True
    _     < _    = False

    b <= c = (b < c) || (b == c)
    b > c  = c < b
    b >= c = c <= b
```

Derived instances

When new types are declared, it is usually appropriate to make them into instances of a number of built-in classes.

```
data Bool = False | True
     deriving (Eq, Ord, Show, Read)
```

```
> False == False
True

> False < True
True
```

应用1：Tautology Checker

问题：Develop a function that decides if simple logical propositions are always true.

$$A \wedge \neg A$$

$$(A \wedge B) \Rightarrow A$$

$$A \Rightarrow (A \wedge B)$$

$$(A \wedge (A \Rightarrow B)) \Rightarrow B$$

应用1：Tautology Checker

解法：求各个命题的真值表，判断结果是否都是真。

A	$A \wedge \neg A$
F	F
T	F

A	B	$(A \wedge B) \Rightarrow A$
F	F	T
F	T	T
T	F	T
T	T	T

A	B	$A \Rightarrow (A \wedge B)$
F	F	T
F	T	T
T	F	F
T	T	T

A	B	$(A \wedge (A \Rightarrow B)) \Rightarrow B$
F	F	T
F	T	T
T	F	T
T	T	T

应用1：Tautology Checker

命题表示

```
data Prop = Const Bool
          | Var Char
          | Not Prop
          | And Prop Prop
          | Imply Prop Prop
```

p1 :: Prop
p1 = And (Var ’A’) (Not (Var ’A’))

p2 :: Prop
p2 = Imply (And (Var ’A’) (Var ’B’)) (Var ’A’)

练习： 定义函数 vars :: Prop -> [Char], 求出一个命题表达式中的变量。

应用1：Tautology Checker

置换表

```
type Subst = Assoc  Char  Bool
```

```
subst :: Subst
subst = [ ('A' ,True),  ('B', False)]
```

练习：

(1) 给定一个Char的序列（如，['A', 'B']) 定义函数substs求出所有可能的置换表。

varSubsts :: [Chair] -> [Subst]

(2) 给定一个置换表和一个命题表达式，定义函数eval求出命题的值。

eval :: Subst -> Prop -> Bool

应用1：Tautology Checker

最终程序

```
isTaut :: Prop -> Bool
isTaut p = and [eval s p  | s <- varSubsts vs ]
     where  vs = rmdups (vars p)
```

```
> isTaut p1
True

> isTaut p2
True

> isTaut p3
False

> isTaut p4
True
```

应用2: 抽象机

- 表达式计算

```
data Expr = Val Int | Add Expr Expr

value :: Expr -> Int
value (Val n)      = n
value (Add x y)  = value x + value y
```

这没有描述计算的顺序。如何描述这样的控制?

北京大学 PEKING UNIVERSITY

应用2: 抽象机

- 引进控制堆栈，描述当前计算结束后需要“继续”计算的部分

type Cont = [Op]

data Op = EVAL Expr | ADD Int

```
eval :: Expr -> Cont -> Int
eval (Val n)   c = exec c n
eval (Add x y) c = eval x (EVAL y : c)
```

应用2: 抽象机

- 计算"控制"堆栈

```
type Cont = [Op]

data Op = EVAL Expr | ADD Int
```

后续计算

当前值

```
exec :: Cont -> Int -> Int
exec []               n = n
exec (EVAL y : c) n = eval y (ADD n : c)
exec (ADD n : c)  m = exec c (n+m)
```

应用2: 抽象机

- 主函数

```
value :: Expr -> Int
value e = eval e []
```

练习： 给出

value (Add (Add (Val 2) (Val 3)) (Val 4))

的运算过程。

作业

8-1 Using recursion and the function add, define a function that multiplies two natural numbers.

8-2 Define a suitable function folde for expressions and give a few examples of its use.

8-3 Define a type Tree a of binary trees built from Leaf values of type a using a Node constructor that takes two binary trees as parameters.