计算概论A—实验班

函数式程序设计

Functional Programming

胡振江，张 伟

北京大学 计算机学院

2023年09～12月

Adapted from Graham’s Lecture slides

第8章：类型和类簇的声明/定义
Declaring Type and Type Class

Type Declaration

- 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 Declaration

- 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)
```

Type Declaration

- 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 Declaration

- Type declarations can be nested:

```
type Pos = (Int, Int)
type Trans = Pos -> Pos
```

- However, they cannot be recursive:

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

• error: Cycle in type synonym declarations

Data Declaration

- A completely new type can be defined by specifying its values using a data declaration.

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

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

- Bool is a type constructor, and False/True is a data constructor
- Type/Data 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.

Data Declaration

- 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
```

Data Declaration

- The data constructors 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
```

Data Declaration

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

- Shape has values of the form Circle r where r is a Float, and Rect x y where x and y are Float.
- Circle and Rect can be viewed as ***functions*** that construct values of type Shape:

```
Circle :: Float -> Shape
  Rect :: Float -> Float -> Shape
```

Data Declaration

- The type constructors 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 $ div m n

safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just $ head xs
```

Recursive Type

- 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 two data constructors
  - Zero :: Nat
  - Succ :: Nat -> Nat

Recursive Type

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

- 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`
  - `Succ $ Succ $ Succ Zero`
  - `...`

Recursive Type

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

- We can think of values of type Nat as natural numbers, where Zero represents 0, and Succ represents the function (1+).
- For example, the value

```
Succ $ Succ $ Succ Zero
```

represents the natural number

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

Recursive Type

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

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

```haskell
nat2int :: Nat -> Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ $ int2nat $ n - 1
```

Recursive Type

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

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

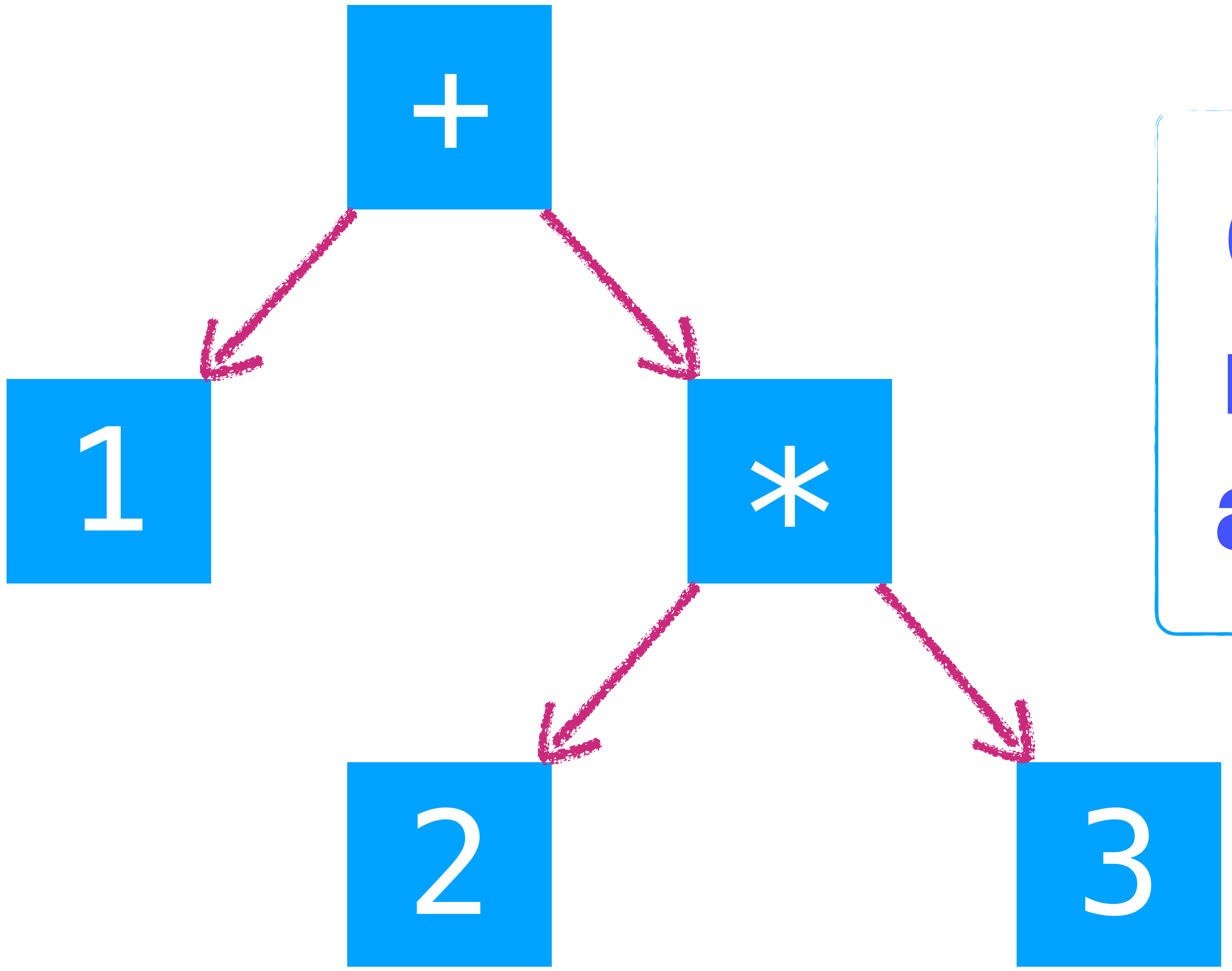
```
add :: Nat -> Nat -> Nat
add m n = int2nat $ nat2int m + nat2int n
```

- 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
```

Example: A Type for Arithmetic Expressions

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

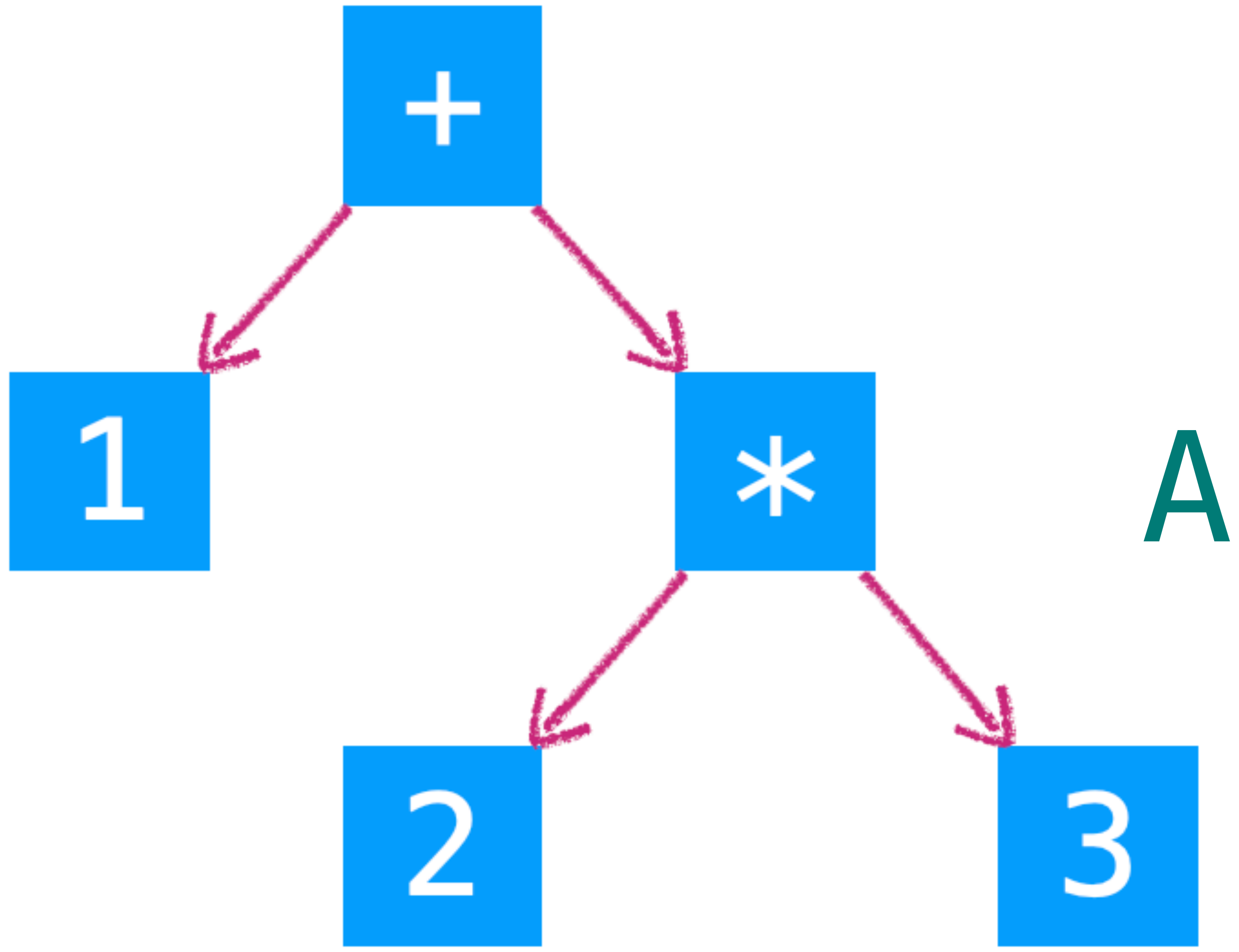


Can we define a type to represent this kinds of arithmetic expressions

Example: A Type for Arithmetic Expressions

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

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




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

Example: A Type for Arithmetic Expressions

- 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
```

Example: A Type for Arithmetic Expressions

- The three constructors have types:
  - `Val :: Int -> Expr`
  - `Add :: Expr -> Expr -> Expr`
  - `Mul :: Expr -> Expr -> Expr`

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

对于类型 Expr
是否存在一个对应的fold函数呢

如果你真正理解了Natural和List上的fold函数
这就是一件非常简单的事情
把这三个data constructors替换为恰当的三个函数

Example: A Type for Arithmetic Expressions

```haskell
folde :: (Int -> a) -> (a -> a -> a) -> (a -> a -> a) -> Expr -> a
```

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

```haskell
size :: Expr -> Int
size = folde (\x -> 1) (+) (+)
```

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

```haskell
eval :: Expr -> Int
eval = folde id (+) (*)
```

Newtype Declaration

- 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

Type class and instance declaration

- Declare a type class

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
    x == y = not (x /= y)
    {-# MINIMAL (==) | (/=) #-}
```

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

Type class and instance declaration

- Declare that a type is an instance of a type class

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

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

Type class and instance declaration

- Type classes can also be extended to form new type classes.

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

    compare x y = if x == y then EQ
                  -- NB: must be '<=' not '<' to validate the
                  -- above claim about the minimal things that
                  -- can be defined for an instance of Ord:
                  else if x <= y then LT
                  else GT

    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }

        -- These two default methods use '<=' rather than 'compare'
        -- because the latter is often more expensive
    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
    {-# MINIMAL compare | (<=) #-}
```

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

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)
```

```
ghci> False < True
True
ghci> False == True
False
```

Example: Tautology Checker / 重言检查器

- The Problem: Develop a function that decides if a simple propositional formula is always true.

1. $A \wedge \neg A$
2. $(A \wedge B) \Rightarrow A$
3. $A \Rightarrow (A \wedge B)$
4. $(A \wedge (A \Rightarrow B)) \Rightarrow B$

Example: Tautology Checker / 重言检查器

✤ 求解方法：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

Example: Tautology Checker / 重言检查器

✤ 定义一个用于表示命题公式的类型

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

1. $A \wedge \neg A$
2. $(A \wedge B) \Rightarrow A$
3. $A \Rightarrow (A \wedge B)$
4. $(A \wedge (A \Rightarrow B)) \Rightarrow B$

```
p1 = And (Var 'A') (Not (Var 'A'))
p2 = Imply (And (Var 'A') (Var 'B')) (Var 'A')
p3 = Imply (Var 'A') (And (Var 'A') (Var 'B'))
p4 = Imply (And (Var 'A') (Imply (Var 'A') (Var 'B'))) (Var 'B')
```

Example: Tautology Checker / 重言检查器

✤ 定义函数 vars :: Prop -> [Char], 求出一个命题公式中的变量

```
vars :: Prop -> [Char]
vars (Const _)    = []
vars (Var x)      = [x]
vars (Not p)      = vars p
vars (And p q)    = vars p ++ vars q
vars (Imply p q) = vars p ++ vars q
```

```
ghci> vars p4
"AABB"
```

```
p4 = Imply (And (Var 'A') (Imply (Var 'A') (Var 'B'))) (Var 'B')
```

Example: Tautology Checker / 重言检查器

✤定义一个类型，用于表达变量与值之间的绑定/置换关系

置换表

```
type Subst = Assoc Char Bool
type Assoc k v = [(k, v)]
```

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

Example: Tautology Checker / 重言检查器

- 定义函数 bools :: Int -> [[Bool]]，用于生成n个bool类型值所有可能的排列

```haskell
bools :: Int -> [[Bool]]
bools 0 = [[]]
bools n = map (False:) bss ++ map (True:) bss
    where bss = bools $ n - 1
```

```
ghci> bools 2
[[False,False],[False,True],[True,False],[True,True]]
```

Example: Tautology Checker / 重言检查器

- 定义函数 varSubsts :: [Char] -> [Subst]：接收一组bool变量，生成对这些变量所有可能的赋值/置换方式

```
varSubsts :: [Char] -> [Subst]
varSubsts vs = map (zip vs) (bools $ length vs)
```

```
ghci> varSubsts "AB"
[[('A',False),('B',False)],[('A',False),('B',True)],
[('A',True),('B',False)],[('A',True),('B',True)]]
```

Example: Tautology Checker / 重言检查器

✤ 定义函数 eval :: Subst -> Prop -> Bool：给定一个命题公式和一个置换表，评估这个命题公式的值

```
eval :: Subst -> Prop -> Bool
eval  _ (Const b)   = b
eval s (Var x)      = find x s
eval s (Not p)      = not (eval s p)
eval s (And p q)    = eval s p  &&  eval s q
eval s (Imply p q) = eval s p  <=  eval s q
```

Example: Tautology Checker / 重言检查器

✤ 定义函数 isTaut :: Prop -> Bool：判断一个命题公式是否重言

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

```
ghci> isTaut p1
False
ghci> isTaut p2
True
ghci> isTaut p3
False
ghci> isTaut p4
True
```

Example: Abstract Machine

✤ 计算表达式的值

For example, the expression $(2 + 3) + 4$ is evaluated as follows:

```
  value (Add (Add (Val 2) (Val 3)) (Val 4))
=     { applying value }
  value (Add (Val 2) (Val 3)) + value (Val 4)
=     { applying the first value }
  (value (Val 2) + value (Val 3)) + value (Val 4)
=     { applying the first value }
  (2 + value (Val 3)) + value (Val 4)
=     { applying the first value }
  (2 + 3) + value (Val 4)
=     { applying the first + }
  5 + value (Val 4)
=     { applying value }
  5 + 4
=     { applying + }
  9
```

- 在类型声明中，未指定表达求值的详细步骤
- Haskell语言在背后帮我们做了很多事情

可以自定义
表达式的求值步骤吗

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

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

Example: Abstract Machine

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

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

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

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
```

作业

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.

第8章：类型和类簇的声明/定义
Declaring Type and Type Class

就到这里吧