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

第4章：函数的定义
Function Definition

主要知识点：利用已有函数定义新函数、条件表达式、模式匹配、Lambda表达式、Section

利用已有函数定义新函数

问题1	判断一个整数是不是偶数

```
even :: Int -> Bool
even n = mod n 2 == 0
```

问题2	求一个浮点数的倒数

```
recip :: Double -> Double
recip x = 1 / x
```

问题2	将一个序列在位置n分开

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)
```

Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions.

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

abs takes an integer n and returns n if it is non-negative and -n otherwise.

Conditional Expressions

Conditional expressions can be nested

```
signum :: Int -> Int
signum n = if n < 0 then -1 else
              if n == 0 then 0 else 1
```

- ✳ In Haskell, conditional expressions must always have an else branch, which avoids any possible ambiguity problems with nested conditionals.

Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

```haskell
abs :: Int -> Int
abs n | n >= 0 = n
      | otherwise = -n
```

Guarded Equations

- Guarded equations can be used to make definitions involving multiple conditions easier to read .

```haskell
signum :: Int -> Int
signum n | n < 0       = -1
         | n == 0      =  0
         | otherwise = -1
```

✳ The catch all condition otherwise is defined in Prelude by otherwise = True

Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments.

```
not :: Bool -> Bool
not False = True
not True  = False
```

not maps False to True, and True to False

- Functions can often be defined in many different ways using pattern matching. For example:

```
(&&) :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_    && _    = False
```

- However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False

```
(&&) :: Bool -> Bool -> Bool
True  && b = b
False && _ = False
```

- The underscore _ is a ***wildcard*** pattern that matches any argument value.

- Patterns are matched ***in order***.
- For example, the following definition always returns False:

```
(&&) :: Bool -> Bool -> Bool
_    && _    = False
True && True = True
```

- Patterns may not repeat variables.
- For example, the following definition gives an error:

```
(&&) :: Bool -> Bool -> Bool
b && b = b
_ && _ = False
```

List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (:) called "cons" that adds an element to the start of a list.

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

=

```
1:(2:(3:(4:[])))
```

List Patterns

Functions on lists can be defined using x:xs patterns

```
head :: [a] -> a
head (x:_) = x
```

- head map any non-empty list to its first element.

```
tail :: [a] -> [a]
tail (_:xs) = xs
```

- tail map any non-empty list to its tail list.

List Patterns

- x:xs patterns only match non-empty lists.

```
program — ghc-9.4.2 -B/Users/nrutas/.ghcup/ghc/9.4.2/li...
ghci>
ghci> head [1,2,3]
1
ghci>
ghci> head []
*** Exception: Prelude.head: empty list
```

- x:xs patterns must be parenthesised, because application has priority over (:).
- For example, the following definition gives an error:

```
head x:_ = x
```

Tuple Patterns

```
-- Extract the first component of a pair.
fst :: (a, b) -> a
fst    (x, _) =  x

-- Extract the second component of a pair.
snd :: (a, b) -> b
snd    (_, y) =  y
```

Lambda Expressions

Functions can be constructed without naming the functions by using lambda expressions.

```
\x -> x + x
```

- the nameless function that takes a value x and returns the result x + x

Why Lambda Expressions

- Lambda expressions can be used to give a formal meaning to functions defined using currying.

```
add x y = x + y
```

=

```
add = \x -> (\y -> x + y)
```

Why Lambda Expressions

- Lambda expressions can be used to avoid naming functions that are only referenced once.

```
odds n = map f [0..n-1]
    where
      f x = x * 2 + 1
```

```
-- defined in Prelude
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

can be simplified to

```
odds n = map (\x -> x * 2 + 1) [0..n-1]
```

Operator Sections

An operator written between its two arguments can be converted into a curried function written before its two arguments by using parentheses.

nrutas — ghc-9.4.2 -B/Users/nrut...

```
ghci>
ghci> 1 + 2
3
ghci> (+) 1 2
3
ghci> :type (+)
(+) :: Num a => a -> a -> a
ghci>
```

Operator Sections

- This convention also allows one of the arguments of the operator to be included in the parentheses.

nrutas — ghc-9.4.2 -B/Users/...

```
ghci>
ghci> (+1) 2
3
ghci> :type (+1)
(+1) :: Num a => a -> a
ghci>
ghci> (1+) 2
3
ghci> :type (1+)
(1+) :: Num a => a -> a
ghci>
ghci> (1-) 2
-1
ghci> :type (1-)
(1-) :: Num a => a -> a
ghci>
```

nrutas — ghc-9.4.2 -B/Users/nruta...

```
ghci>
ghci> :type (1-)
(1-) :: Num a => a -> a
ghci>
ghci> :type (-1)
(-1) :: Num a => a
ghci> (-1) 2

<interactive>:25:1: error:
```

Operator Sections

In general, if is an operator ⊕ then functions of the form (⊕), (x ⊕) and (⊕ y) are called sections.

```
(⊕)   = \x -> (\y -> x ⊕ y)

(x ⊕) = \y -> x ⊕ y

(⊕ y) = \x -> x ⊕ y
```

Why Operator Sections

- Useful functions can sometimes be constructed in a simple way using sections.

`(+ 1)`	successor function
`(1 /)`	reciprocation function
`(* 2)`	doubling function
`(/ 2)`	halving function

作业

作业

4-1 Consider a function safetail that behaves in the same way as tail, except that safetail maps the empty list to the empty list, whereas tail gives an error in this case. Define safetail using:

(a) a conditional expression;
(b) guarded equations;
(c) pattern matching.

✳Hint: the library function `null :: [a] -> Bool` can be used to test if a list is empty.

作业

4-2 The Luhn algorithm is used to check bank card numbers for simple errors such as mistyping a digit, and proceeds as follows:

(1) consider each digit as a separate number;

(2) moving left, double every other number from the second last;（从右向左，偶数位的数字乘2）

(3) subtract 9 from each number that is now greater than 9; add all the resulting numbers together;

(4) if the total is divisible by 10, the card number is valid.

Define a function luhn :: Int -> Int -> Int -> Int -> Bool that decides if a four-digit bank card number is valid. For example:

```
> luhn  1  7  8  4
True

> luhn  4  7  8  3
False"
```

第4章： 函数的定义
Function Definition

就到这里吧