Adapted from Graham's Lecture slides.

第四章: 函数定义

利用现有函数定义新函数 条件表达式,模式匹配 Lambda表达式,Section记号



1

利用已有函数定义新函数

- 问题:判定一个整数是不是偶数。
 even::Int→Bool
 - even n = ...
- 问题:求一个浮点数的倒数
 recip::Float → Float
 recipx=...
- 问题:将一个序列在位置n分开 splitAt::Int→[a]→([a],[a]) splitAtnxs=...



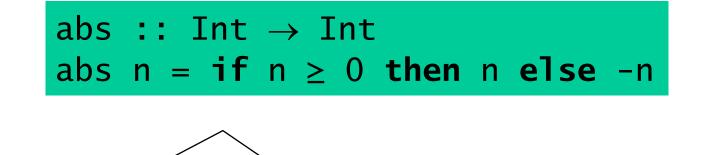
利用已有函数定义新函数

- 问题:判定一个整数是不是偶数。
 even :: Int → Bool
 even n = n 'mod' 2 == 0
- 问题:求一个浮点数的倒数
 recip::Float → Float
 recip x = 1/x
- 问题: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)





As in most programming languages, functions can be defined using <u>conditional expressions</u>.



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



Conditional expressions can be nested:

signum :: Int \rightarrow Int signum n = if n < 0 then -1 else if n == 0 then 0 else 1

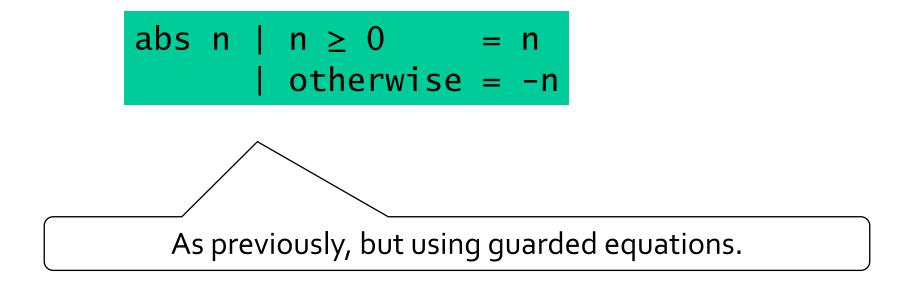
Note:

In Haskell, conditional expressions must <u>always</u> 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 <u>guarded equations</u>.





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

Note:

The catch all condition <u>otherwise</u> is defined in the prelude by otherwise = True.



Pattern Matching

Many functions have a particularly clear definition using <u>pattern</u> <u>matching</u> on their arguments.

not	:: Bool	\rightarrow 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

(&&)	:: E	$3001 \rightarrow$	B	$ool \rightarrow Bool$
True	&&	True	=	True
True	&&	False	=	False
False	&&	True	=	False
False	&&	False	=	False

can be defined more compactly by

True && True = True _ && _ = False



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

Note:

The underscore symbol _ is a <u>wildcard</u> pattern that matches any argument value.



Patterns are matched <u>in order</u>. For example, the following definition always returns False:

_	&&	_	=	False
True	&&	True	=	True

Patterns may not <u>repeat</u> variables. For example, the following definition gives an error:



List Patterns

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



Functions on lists can be defined using <u>x:xs</u> patterns.

head ::
$$[a] \rightarrow a$$

head $(x:_) = x$
tail :: $[a] \rightarrow [a]$
tail $(_:xs) = xs$

head and tail map any non-empty list to its first and remaining elements.



Note:

x:xs patterns only match <u>non-empty</u> lists:

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

head
$$x: = x$$



Tuple Patterns

A tuple of patterns is itself a pattern, which matches any tuple of the same arity whose components all match the corresponding patterns in order.





Functions can be constructed without naming the functions by using <u>lambda expressions</u>.

 $\lambda x \rightarrow x + x$

the nameless function that takes a number x and returns the result x + x.



- The symbol λ is the Greek letter <u>lambda</u>, and is typed at the keyboard as a backslash \.
- In mathematics, nameless functions are usually denoted using the \mapsto symbol, as in $x \mapsto x + x$.
- In Haskell, the use of the λ symbol for nameless functions comes from the lambda calculus, the theory of functions on which Haskell is based.



Why Are Lambda's Useful?

Lambda expressions can be used to give a formal meaning to functions defined using <u>currying</u>.

For example:

add x
$$y = x + y$$

means

add =
$$\lambda x \rightarrow (\lambda y \rightarrow x + y)$$



Lambda expressions can be used to avoid naming functions that are only <u>referenced once</u>.

For example:

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

can be simplified to

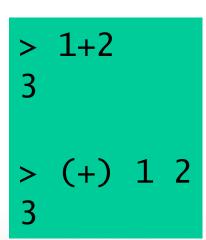
odds n = map ($\lambda x \rightarrow x^{*2} + 1$) [0...1]





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

For example:





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

For example:



In general, if \oplus is an operator then functions of the form (\oplus), (x \oplus) and (\oplus y) are called <u>sections</u>.

$$(\oplus) = \langle x - \rangle (\langle y - \rangle x \oplus y)$$
$$(x \oplus) = \langle y - \rangle x \oplus y$$
$$(\oplus y) = \langle x - \rangle x \oplus y$$

• 对于函数f::a→b→c, `f` 可以作为 operator 来用:

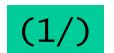
$$f x y = x f y$$



Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections. For example:





- reciprocation function



- doubling function



- halving function



作业

- 4-1 Consider a function <u>safetail</u> 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] \rightarrow Bool can be used to test if a list is empty.



- 4⁻² The Luhn algorithm is used to check bank card numbers for simple errors such as mistyping a digit, and proceeds as follows:
 - consider each digit as a separate number;
 - moving left, double every other number from the second last (从右向 左,偶数位的数字乘2)
 - subtract 9 from each number that is now greater than 9; add all the resulting numbers together;
 - 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 fourdigit bank card number is valid. For example:

```
> luhn 1 7 8 4
True
```

```
> luhn 4 7 8 3
False"
```

