Haskell Scripts

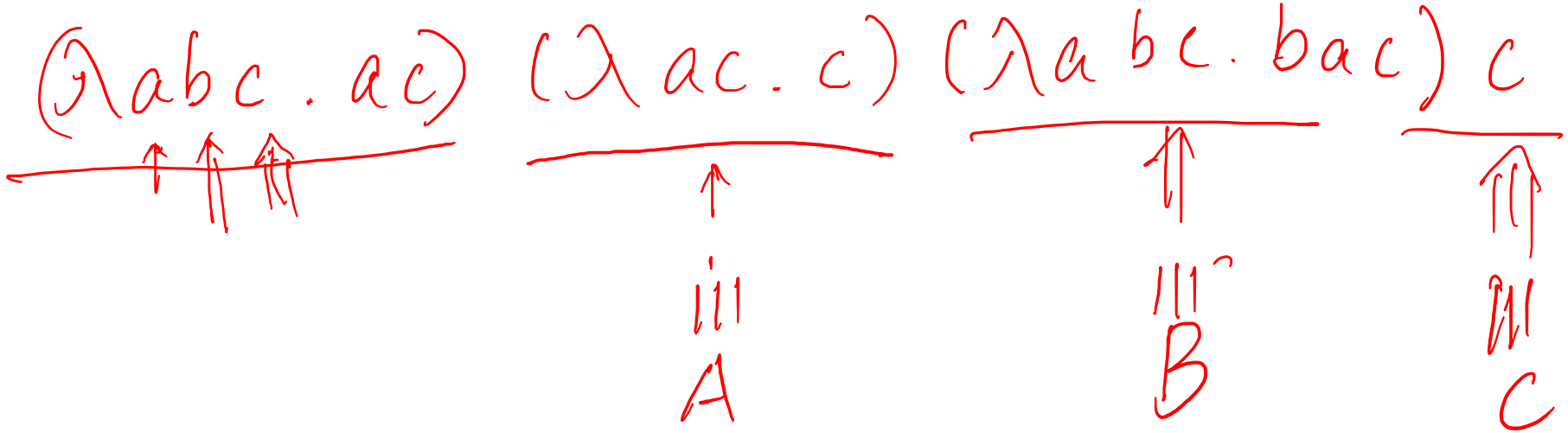
Yan Huang

yh33@indiana.edu

β-reduction $(\lambda a.M)\ x \to_\beta M[a \to x]$

Last Quiz

$(\lambda abc.ac)\ (\lambda ac.c)\ (\lambda abc.bac)\ c$



$\to_\beta AC \equiv (\lambda ac.c)\ c \to_\beta \lambda c.c$

$(\lambda ac.c)\ x \to (\lambda c.c)[a \to x] \equiv \lambda c.c$

Objectives

- Writing Haskell programs in "`.hs`" files
- Note some differences between programs typed into GHCi and programs written in script files
- Operator Precedence
- Operator Associativity

Haskell Scripts

- As well as the functions in the standard library, you can also define your own functions;
- New functions are defined within a script, a text file comprising a sequence of definitions;
- By convention, Haskell scripts usually have a .hs suffix on their filename.

My First Script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running GHCi.

Start an editor, type in the following two function definitions, and save the script as test.hs:

```
double x     = x + x

quadruple x = double (double x)
```

:load

Leaving the editor open, in another window start up GHCi with the new script:

```
$ ghci test.hs
```

Now both the standard library and the file test.hs are loaded, and functions from both can be used:

```
> quadruple 10
40
> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

Leaving GHCi open, return to the editor, add the following two definitions, and save:

```
factorial n = product [1..n]

average ns = sum ns `div` length ns
```

Note:

$\equiv$ div (sum ns) (length ns)

- `div` is enclosed in back quotes, not forward;
- `x `f` y` is just syntactic sugar for `f x y`.

GHCi does not automatically detect that the script has been changed, so a reload command must be executed before the new definitions can be used:

```
> :reload
Reading file "test.hs"
> factorial 10
3628800
> average [1,2,3,4,5]
3
```

Useful GHCi Commands

Command	Meaning
:load *name*	load script *name*
:reload	reload current script
:type *expr*	show type of *expr*
:?	show all commands
:quit	quit GHCi
:set editor *name*	set editor to *name*
:edit *name*	edit script *name*
:edit	edit current script

:info

:help

Naming Requirements and Conventions

- Function and argument names must begin with a lower-case letter. For example:

`myFun` `fun1` `arg_2` `x'`

By convention, list arguments usually have an s suffix on their name. For example:

`xs` `ns` `nss`

The Layout Rule

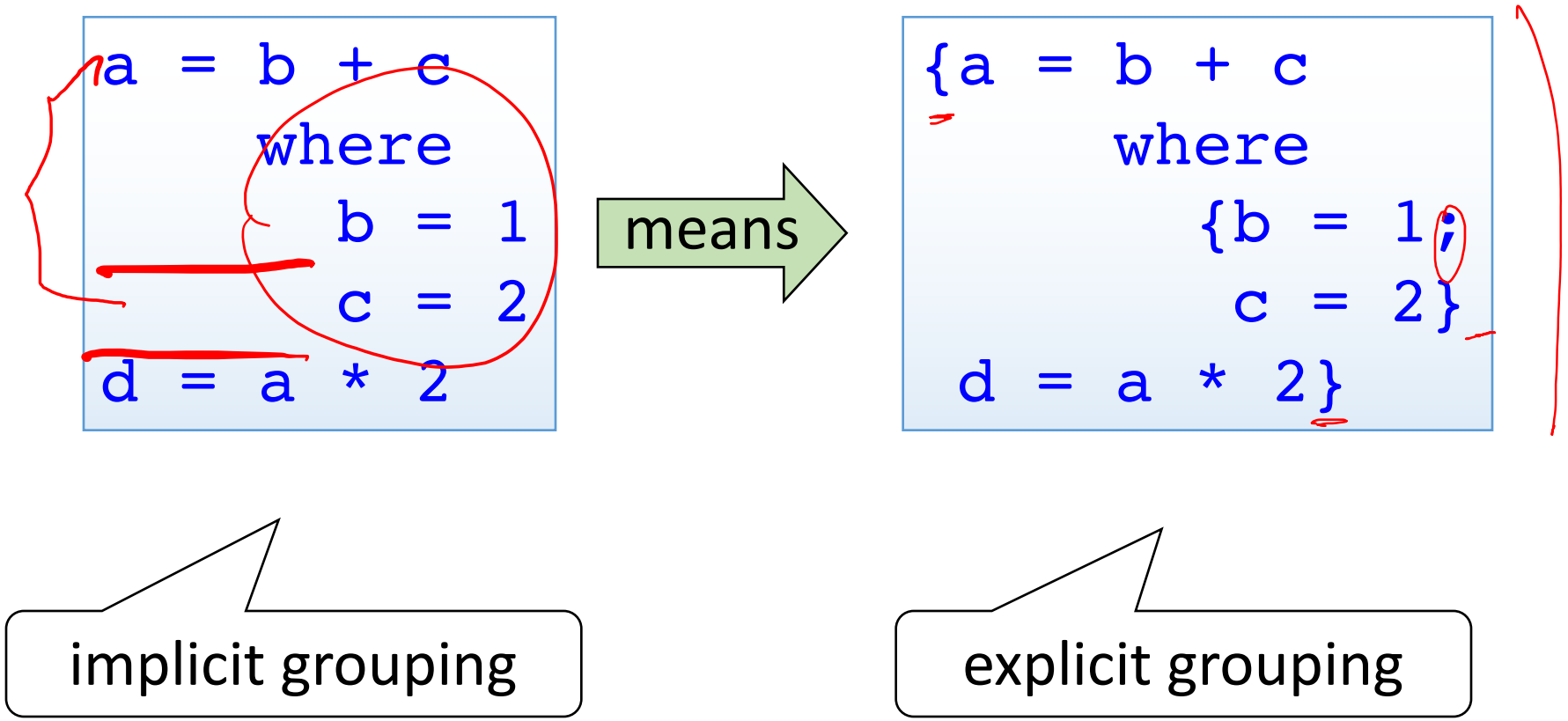
In a sequence of definitions, each definition must begin in precisely the same column:

```
a = 10
b = 20
c = 30
```

```
a = 10
 b = 20
c = 30
```

```
 a = 10
b = 20
 c = 30
```

The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.



```
In script:

a = 1
```

```
In GHCi:

let a = 1
```

Operators

```
Prelude λ: 3+5
8
Prelude λ: (+) 3 5
8
Prelude λ: 3 / 5
0.6
Prelude λ: (/) 3 5
0.6
```

```
Prelude λ: mod 3 5
3
Prelude λ: 3 `mod` 5
3
Prelude λ: 2 `elem` [1,2,3]
True
Prelude λ: elem 2 [1,2,3]
True
```

- Operators are in essence functions.
- Operators whose identifiers consist of symbols are by default ***infix***. Surround them with ***parentheses*** to use as prefix operators.
- Operators with letters in their identifiers are ***prefix*** by default. Place them in back-quotes to use as infix operators.

Operator Precedence

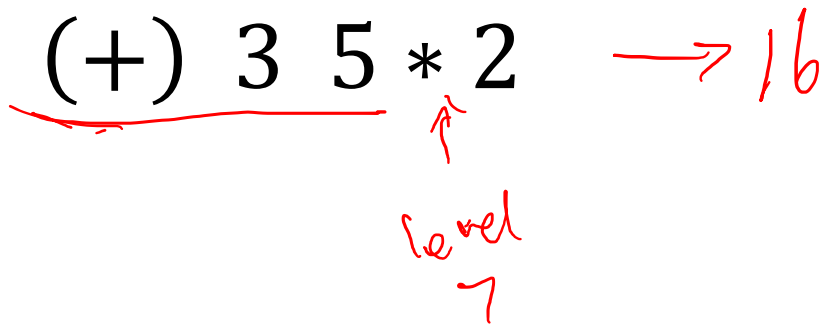
$$3 + 5 * 2$$

→ 13

“*” has higher precedence than “+”

```
Prelude λ: :info +
class Num a where
  (+) :: a -> a -> a
  ...
  -- Defined in ‘GHC.Num’
infixl 6 +
Prelude λ: :info *
class Num a where
  ...
  (*) :: a -> a -> a
  ...
  -- Defined in ‘GHC.Num’
infixl 7 *
```

Operator Precedence



- Prefix form (+) is treated as normal function (precedence escalated to that of function applications, which is higher than '*')
- Infix + is processed as normal binary operators

Precedence and Associativity of Selected Operators

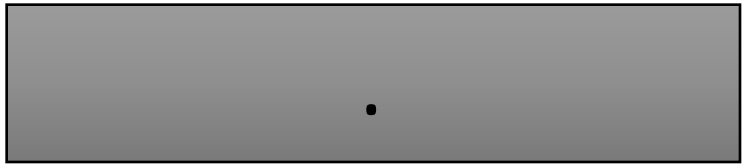
Predefined in Prelude

	Left-associative	Non-associative	Right-associative
9	!!		.
8			^, ^^, **
7	*, /, `div`		
6	+, -		
5			:, ++
4		==, /=, >, >=, <, <= `elem`, `notElem`	
3			&&
2			||
1	>>, >>=		
0			$, $!, `seq`

- Function applications are of level 10 precedence.
- Any operator lacking a fixity declaration is assumed to be `infixl 9`.

[3, 4, 100] !! 2 → 100

```
*Main λ: :type (!!)
(!!) :: [a] -> Int -> a
```




```
*Main λ: :type (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

f1 → f2 → f3

^, ^^, **

```
*Main λ: :type (^)
(^) :: (Integral b, Num a) => a -> b -> a

*Main λ: :type (^^)
(^^) :: (Fractional a, Integral b) => a -> b -> a

*Main λ: :type (**)
(**) :: Floating a => a -> a -> a
```

*, /, `div`

```
*Main λ: :type (*)
(*) :: Num a => a -> a -> a

*Main λ: :type (/)
(/) :: Fractional a => a -> a -> a

*Main λ: :type div
div :: Integral a => a -> a -> a
```

+, -

:, ++

5 : [1,2,3] ⇒ eval [5,1,2,3]

```
*Main λ: :type (:)
(:) :: a -> [a] -> [a]

*Main λ: :type (++)
(++) :: [a] -> [a] -> [a]
```

[1,3] ++ [0,2] → eval [1,3,0,2]

==, /=, >, >=, <, <=
`elem`, `notElem`

```
*Main λ: :type elem
elem :: (Eq a, Foldable t) => a -> t a -> Bool

*Main λ: :type notElem
notElem :: (Eq a, Foldable t) => a -> t a -> Bool
```

&&

```
*Main λ: :type (&&)
(&&) :: Bool -> Bool -> Bool
```

||

>>, >>=

```
*Main λ: :t (>>)
(>>) :: Monad m => m a -> m b -> m b

*Main λ: :t (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

$, $!, `seq`

```
*Main λ: :info ($)
($) :: (a -> b) -> a -> b        -- Defined in ‘GHC.Base’
infixr 0 $

*Main λ: :info ($!)
($!) :: (a -> b) -> a -> b       -- Defined in ‘GHC.Base’
infixr 0 $!

*Main λ: :info seq
seq :: a -> b -> b   -- Defined in ‘GHC.Prim’
infixr 0 `seq`
```

Non-Associative Operators

==, /=, >, >=, <, <=
`elem`, `notElem`

```
Prelude λ: True == False
False
Prelude λ: True == False == False

<interactive>:551:1:
    Precedence parsing error
        cannot mix ‘==’ [infix 4] and ‘==’ [infix 4] in the same infix expression
```

Operator Associativity

- Associate to the *left*

1+2+3 ≡ (1+2)+3 → 3+3 → 6

5-4-3 ≡ (5-4)-3 → 1-3 → -2

18 / 9 / 1 ≡ (18/9)/1 → 2/1 → 2

Operator Associativity

- Associate to the *right*

5^3^2

≡ 5^(3^2)

→ 5^9 → 1953125

```
Prelude λ: :info ^

(^) :: (Num a, Integral b) => a -> b -> a  -- Defined in ‘GHC.Real’

infixr 8 ^
```

1:2:3:[]

→ 1:2:3:[]

→ 1:2:[3]

→ 1:[2,3] → [1,2,3]

```
Prelude λ: :info :

data [] a = ... | a : [a]          -- Defined in ‘GHC.Types’

infixr 5 :
```

Operator Precedence and Associativity

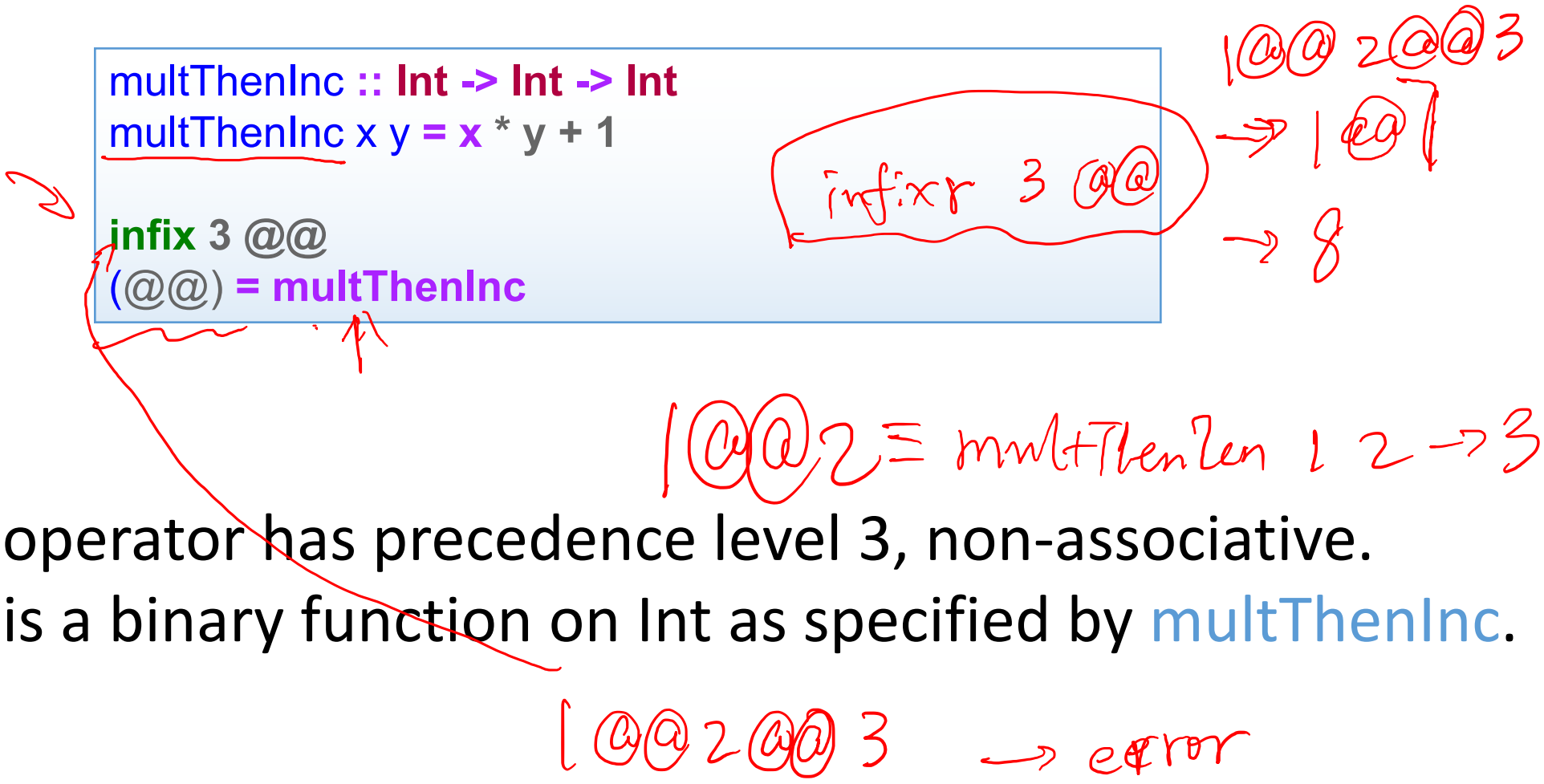
$$f\ (g\ x) \not\equiv f\ g\ x$$

$$f\ (g\ x) \equiv f\ \$\ g\ x$$

Customizing Precedence and Associativity

```
multThenInc :: Int -> Int -> Int
multThenInc x y = x * y + 1

infix 3 @@
(@@) = multThenInc
```



@@ operator has precedence level 3, non-associative.

@@ is a binary function on Int as specified by multThenInc.

Operator Precedence and Associativity

[124, 12, 8, 10]

(5^3-1^2):(2^3*3`div`2):(3+5):(3+2^3`div`2+3):[]

124 : 12 : 8 : 10 : []

5^3-1^2 → 125 - 1 → 124

(2^3)*3`div`2 → 8 * 3 `div` 2 → 24 `div` 2 → 12

3+5 → 8

3+2^3`div`2+3 → 3 + 8 `div` 2 + 3 → 3 + 4 + 3 → 10

Exercises

(1) Fix the syntax errors in the program below, and test your solution using GHCi.

```
N = a 'div' length xs
    where
      a = 10
      xs = [1,2,3,4,5]
```

```
mylast xs = xs !! (length xs - 1)
```

(2) Show how the library function `last` that selects the last element of a list can be defined using the functions introduced in this lecture.

length.
!!
:
head,
tail.

(3) Can you think of another possible definition?

See this next week.

(4) Similarly, show how the library function `init` that removes the last element from a list can be defined (possibly in two different ways).

1 @@ 2 @@ 3 + 4 → 1 @@ 2 @@ 7 → 3 @@ 7 → 22

(5) What does `1 @@ 2 @@ 3 + 4` evaluate to under the following three definitions? Explain.

```
multThenInc :: Int -> Int -> Int
multThenInc x y = x * y + 1

infixl 3 @@
(@@) = multThenInc
```

```
multThenInc :: Int -> Int -> Int
multThenInc x y = x * y + 1

infixl 7 @@
(@@) = multThenInc
```

```
multThenInc :: Int -> Int -> Int
multThenInc x y = x * y + 1

infix 5 @@
(@@) = multThenInc
```

(6) What do `1 @@ 2 @@ 3`,
`1 == 2 == 2`,
`3 @@ 5 == 1` evaluate to, respectively? Explain.