Modules

- Modules
- Local definition
- Operators

Notes Chapter 21, 24

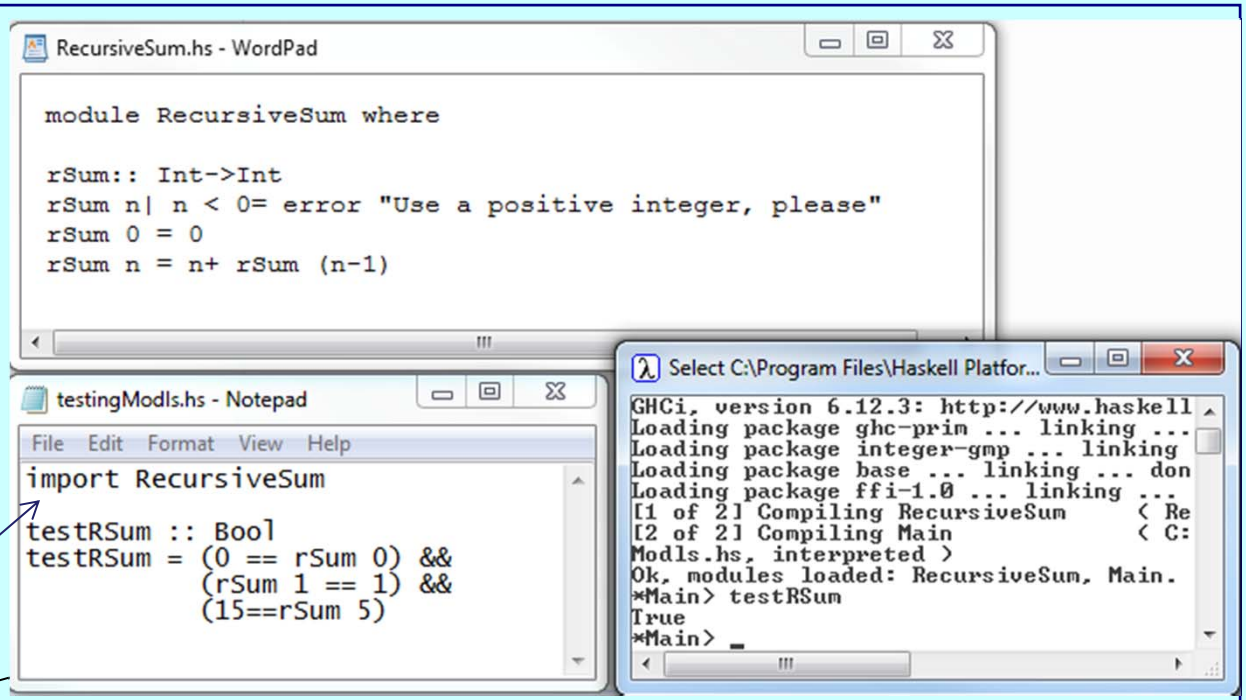
Modules

A module defines a collection of values, functions, classes, etc. and *exports* some of these resources, making them available to other modules.

A Haskell *program* is a collection of modules.

Each module is contained in its own file and the name of the module should match the name of the file (without the ".hs" extension, of course).

Modules



enables us to use any definitions in the module "RecursiveSum".

Restricting access

Example:

1 MyModule.hs

```
1   module MyModule (copyIt, comp) where
2
3   comp::Float->Float
4   comp x = x + add4 x
5
6   copyIt::String->String
7   copyIt text = text ++ text
8
9   add4::Float->Float
10  add4 x = x + 4.0
11
```

1 MyTest.hs

```
1   module MyTest where
2   import MyModule
3
4   func:: Float -> Float
5   func x = x+ comp x
6   --funct x = x + add4 x
7
```

```
Prelude> :load "MyTest.hs"
[1 of 2] Compiling MyModule         ( MyModule.hs, interpreted )
[2 of 2] Compiling MyTest           ( MyTest.hs, interpreted )
Ok, modules loaded: MyTest, MyModule.
*MyTest> func 10
34.0
*MyTest> add4 23

<interactive>:11:1: Not in scope: ‘add4’
*MyTest> copyIt "It's Sunday"
"It's SundayIt's Sunday"
*MyTest>
```

Hiding

```
import Prelude hiding (max)
max::Int->Int->Int->Int
max x y z
   | x>=y && x>= z =x
   | y>=x && y>= z =y
   | otherwise =z
```

Packages

Modules can be grouped in packages. A package is a unit consisting of Haskell modules, C source code and header files, documentation, test cases, and auxiliary tools.
For example:

- **base package** -contains the Prelude and a large collection of useful libraries including debugging utilities.
- **QuickCheck package** – contains tools for testing Haskell programs automatically. (http://www.math.chalmers.se/~rjmh/QuickCheck/)
- **OpenGL package** contains a Haskell binding for the OpenGL graphics system (http://www.opengl.org/)

Local definitions

Consider this:

```
fc:: Float -> Float -> Float -> Float
fc a b c = (a + b + c)/3 +  sqrt ((a + b + c)/3)
```

Local definitions

```
fc:: Float -> Float -> Float -> Float
fc a b c = avr3+ sqrt (avr3)
   where avr3 = (a + b +c)/3
```

Local definitions are definitions within a function that make the function easier to read and understand.

Local and non-local definitions

```
fc1:: Float -> Float -> Float -> Float
fc1 a b c = (a + b + c)/3 +  sqrt ((a + b + c)/3)


fc3 a b c = val a b c + sqrt (val a b c)

fc:: Float -> Float -> Float -> Float
fc a b c = avr a b c + sqrt (avr a b c)
      where avr x y z = (x + y +z)/3

fc2:: Float -> Float -> Float -> Float
fc2 a b c = avr a b c + sqrt (avr a b c)
      where avr::Float->Float->Float->Float
            avr x y z = (x + y +z)/3

val a b c= (a+ b+ c)/3
```

Fractions example

1/1 , 0/1, 4/5, 123/234, -12/34

1/0 NOT -defined

5/3 - reduce form

20/18= 10/9

4/5 + 3/2= (4*2+ 3*5)/2*5= 23/10

4/8 + 3/2= 1/2+ 3/2 = 4/2 = 2/1

4/5 - 3/2= (4*2- 3*5)/2*5= (-7)/10

4/8 - 3/2= 1/2 - 3/2 = (-2)/2 = (-1)/1

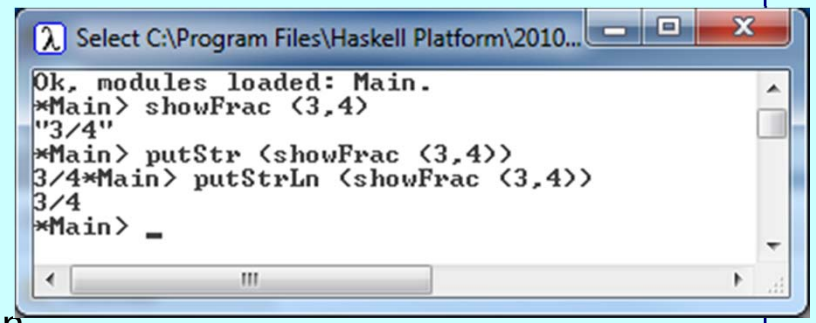
2/3 < 5/3 (T), 5/2 < 9/4(F)

Fractions example

```
--assume that all numbers are non-negative integers
-- for the Fraction example

type Fraction = (Int, Int)
------display as  "a/b"--------
showFrac :: Fraction ->String
--make a fraction in reduced form
makeFrac :: Int -> Int ->Fraction

--Operation on fractions----------
--------addition----------------
addFrac:: Fraction ->Fraction->Fraction
--subtraction-------------------
subFrac:: Fraction->Fraction->Fraction
--comparison for fractions-----------
compFrac ::Fraction ->Fraction->Bool
```



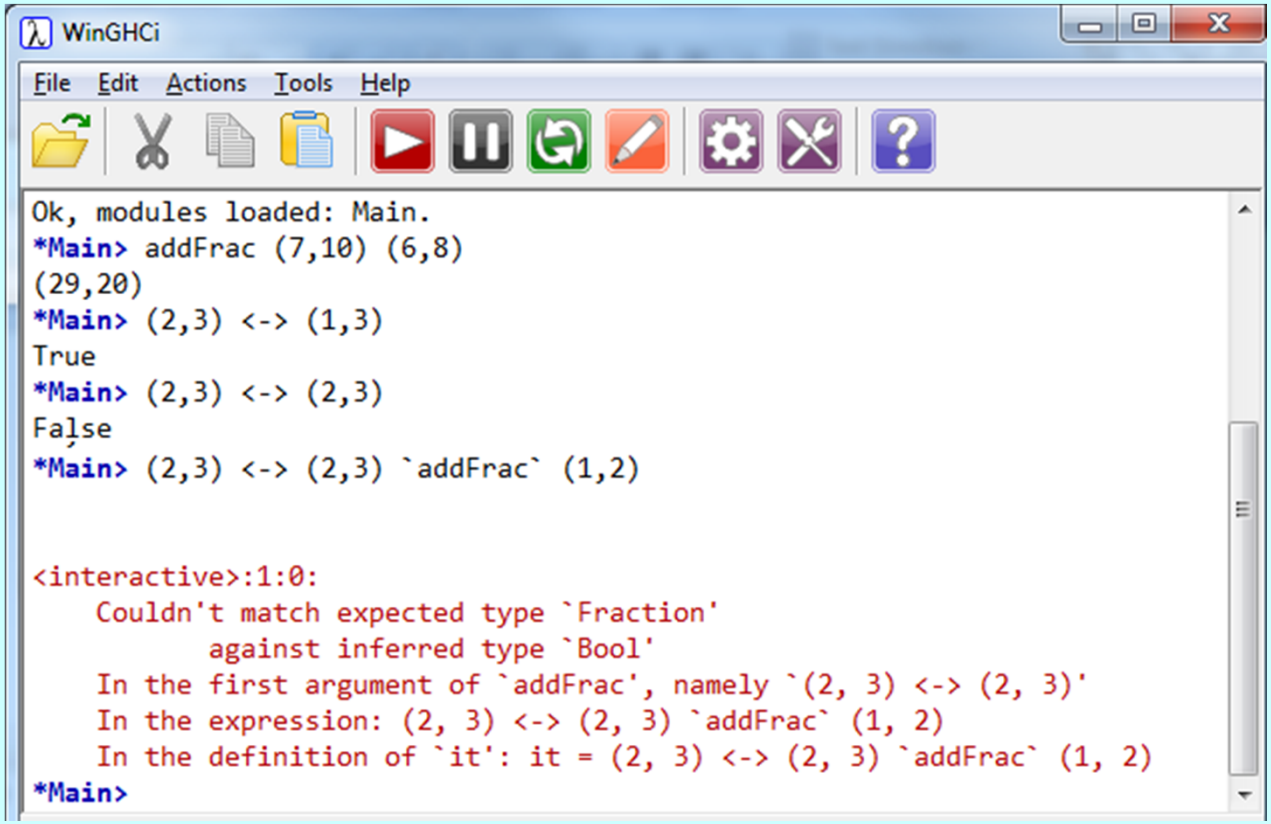
Fractions example

We can define an operator .

Example:

```
(<->) :: Fraction -> Fraction -> Bool
a  <->  b = addFrac a b == (1,1)
Main> (1,3) <-> (2,3)
True
Main> (1,4) <-> 2,3)
False
```

Fractions example



Prefix and infix

In mathematics:
add (3, 4) -- prefix – function notation
3 + 4 -- infix -operator notation

In Haskell:

(+) 3 4 -- prefix - function

3 + 4 -- infix -operator

```
(*) 3   4    =   12
3 * 4        =   12

mod  9 4     =   1
9 `mod` 4    =   1
```

Fixity

Setting the associativity and/or binding power of an operator:

```
infixr  binding-power  operator
infixl  binding-power  operator
infix   binding-power  operator
```

Examples:

```
infixr  7   <->
infixl  5   <->
```

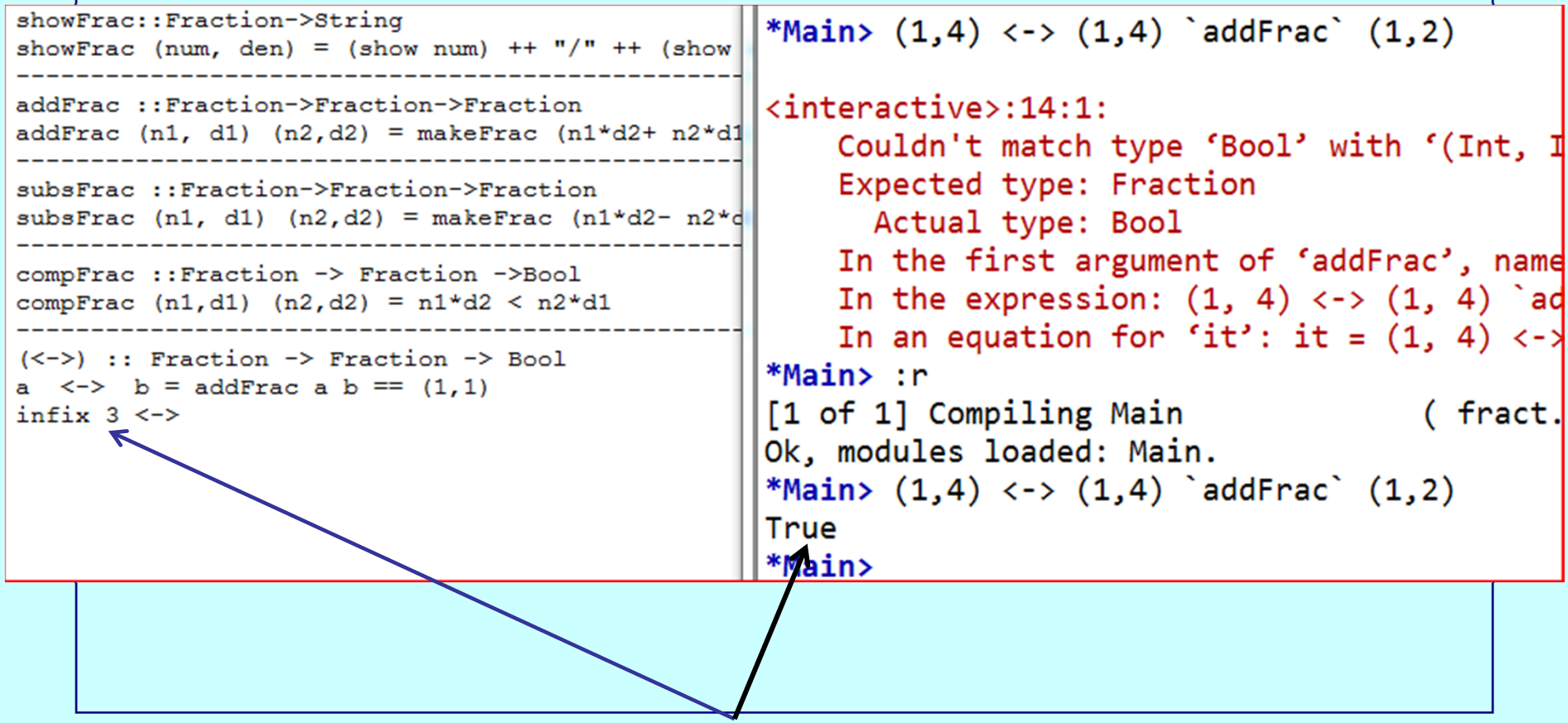
```
------159.202 demos of Fraction type (from Notes)------
type Fraction = (Int, Int)
-------------------------------------------------------
makeFrac :: Int -> Int -> Fraction
makeFrac _ 0 = error "Denominator is zero"
makeFrac num den = ( num `div` d, den `div` d)
   where d= gcd num den

showFrac::Fraction->String
showFrac (num, den) = (show num) ++ "/" ++ (show den)++"\n"
-------------------------------------------------------
addFrac ::Fraction->Fraction->Fraction
addFrac (n1, d1) (n2,d2) = makeFrac (n1*d2+ n2*d1) (d2*d1)
-------------------------------------------------------
subsFrac ::Fraction->Fraction->Fraction
subsFrac (n1, d1) (n2,d2) = makeFrac (n1*d2- n2*d1) (d2*d1)
-------------------------------------------------
compFrac ::Fraction -> Fraction ->Bool
compFrac (n1,d1) (n2,d2) = n1*d2 < n2*d1
-------------------------------------------------
(<->) :: Fraction -> Fraction -> Bool
a  <->  b = addFrac a b == (1,1)
--infix 3 <->
```

```
GHCi, version 7.10.2: http://www.haskell.org/ghc/  :? for help
Prelude> :cd C:\Users\ecalude\Desktop
Prelude> :load "fract.hs"
[1 of 1] Compiling Main             ( fract.hs, interpreted )
Ok, modules loaded: Main.
*Main> putStr (showFrac (addFrac (15,30) (3,6)))
1/1
*Main> (15,30) <-> (1,2)
True
*Main> (1,4) <-> addFrac (1,8) (1,8)
False
*Main> (1,4) <->addFrac (1,4) (1,2)
True
*Main> (1,4) <-> (1,4) `addFrac` (1,2)

<interactive>:14:1:
    Couldn't match type ‘Bool’ with ‘(Int, Int)’
    Expected type: Fraction
      Actual type: Bool
    In the first argument of ‘addFrac’, namely ‘(1, 4) <-> (1, 4)’
    In the expression: (1, 4) <-> (1, 4) `addFrac` (1, 2)
    In an equation for ‘it’: it = (1, 4) <-> (1, 4) `addFrac` (1, 2)
*Main>
```

Problem solved



```
showFrac::Fraction->String
showFrac (num, den) = (show num) ++ "/" ++ (show
-------------------------------------------------
addFrac ::Fraction->Fraction->Fraction
addFrac (n1, d1) (n2,d2) = makeFrac (n1*d2+ n2*d1
-------------------------------------------------
subsFrac ::Fraction->Fraction->Fraction
subsFrac (n1, d1) (n2,d2) = makeFrac (n1*d2- n2*c
-------------------------------------------------
compFrac ::Fraction -> Fraction ->Bool
compFrac (n1,d1) (n2,d2) = n1*d2 < n2*d1
-------------------------------------------------
(<->) :: Fraction -> Fraction -> Bool
a  <->  b = addFrac a b == (1,1)
infix 3 <->
```

```
*Main> (1,4) <-> (1,4) `addFrac` (1,2)

<interactive>:14:1:
    Couldn't match type ‘Bool’ with ‘(Int, I
    Expected type: Fraction
      Actual type: Bool
    In the first argument of ‘addFrac’, name
    In the expression: (1, 4) <-> (1, 4) `ad
    In an equation for ‘it’: it = (1, 4) <->
*Main> :r
[1 of 1] Compiling Main             ( fract.
Ok, modules loaded: Main.
*Main> (1,4) <-> (1,4) `addFrac` (1,2)
True
*Main>
```

Operators

Operators are functions with a special name.

Their name may consist of the following symbols

```
! # $ % & * + . / < = > ? @ \ ^ | - ~
```

When you write down the type of an operator you have to put the name between parentheses:

```
(+++) :: (Int, Int) ->(Int, Int)->(Int, Int, Int)
(x, y) +++ (y, z) = (x, y, z)
```

Using parentheses allows us to use it as a prefix function:

```
(+) 3 4
```

Syntax diversion

Quotes
Here is an overview of all the uses of quotes in Haskell and their meaning:

'a'	a value of type *Char*
"yada"	a value of type *String*
`` `mod` ``	function mod used as an operator, e.g. 12 `` `mod` `` 5
fact'	a single quote as part of the name of a function

Questions

1) Create an infix operator # which returns the average of the two Float arguments. Note / performs floating point division.

2) Consider the function:

```
foo a 1 = a
foo a b = a * foo a (b-1)
```

(i) Find out the value of foo 3 2
(ii) Re-write the function using recursion and guards
(iii) Explain in one sentence what the function does.
(iv) Write the type of the function

Next Type classes