Getting Start

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Glasgow Haskell Compiler (GHC)

- Main Components of GHC:
  1. `ghci`: Interpreter and debugger
  2. `ghc`: Compiler that generates native code
  3. `runghc`: Run Haskell programs as a script without compiling
- Most of the time, we will use `ghci` during implementation
- `ghc` can be used to generate object files and executable programs
- Native object files can be linked to other programming language such as C
- Examples are based on `GHC` under the UNIX environment

GHCi

- To start the GHCi interpreter, simply execute the ghci command:

```
$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude>
```

- The prompt Prelude> indicates the Prelude standard library is loaded
- Other libraries such as Data.Char can be loaded using the import command:

```
Prelude> import Data.Char
Prelude Data.Char>
```

- The more library we import, the longer the prompt
- To fix the prompt, we used the :set prompt command:

```
Prelude Data.Char> :set prompt "ghci> "
ghci>
```

- GHCi prompt will be shown in **Boldface texts**

- The standard `Prelude` comes with a number of **infix** arithmetic operators:

```
ghci> 23 + 49          -- An inline comment starts with -- symbols
72
```

```
ghci> 32 - 67
-35
```

```
ghci> 2 * (-26)        -- 2 * -26 will cause an error
-52
```

```
ghci> 44 / (-(2 * 4))  -- 44 / -(2 * 4) will cause an error
-5.5
```

```
ghci> 2 ^ 6
64
```

- To mix infix and unary (such as −) operators, we must wrap the expression in parentheses
- The / operator is the floating-point division
- Haskell does not have an integer division

Exponent Operators

- The ^ operator is x^y where
  - x is an integer or a floating-point number
  - y is a non-negative integer

```
ghci> 2 ^ 4
16
ghci> (-1.2) ^ 3
-1.728
ghci> 2 ^ (-4)
*** Exception: Negative exponent
```

- The ^^ operator is x^y where
  - x is a floating-point number
  - y is an integer

```
ghci> 1.5 ^^ (-2)
0.4444444444444444
ghci> 5 ^^ 2
25.0
ghci> (5::Int) ^^ 2

<interactive>:11:1: error: ...
ghci> 1.5 ^^ 1.2

<interactive>:14:1: error: ...
```

Exponent Operators

- The $**$ operator is x^y where
  - x is a floating-point number
  - y is a floating-point number

```
ghci> 1.2 ** 2.3
1.5209567545525315
ghci> 1.5 ** (-1.7)
0.501931971314158
```

- We can turn an infix operator into a postfix operator by enclosing the operator in parentheses:

```
ghci> (+) 23 49
72
ghci> (-) 32 67
-35
ghci> (^) 2 100
1267650600228229401496703205376
```

Boolean in Haskell

- Boolean literals in Haskell are `True` and `False`
- Logical "and" `(&&)` and logical "or" `(||)` are provided

```
ghci> True
True
ghci> True && False
False
ghci> False || True
True
```

- Comparison operators, ==, /=, <, <=, >, and >= are provided
- /= is the "not equal" operator in Haskell
- As usual, the result of a comparison is a Boolean:

```
ghci> 32 == 32
True
ghci> 45 /= 9
True
ghci> 12 <= 33
True
ghci> 12 <= 11
False
```

Operator Precedence and Associativity

- Haskell has the same operator precedence as in C and Java

```
ghci> 1 + 2 * 3
7
ghci> (1 + 2) * 3
9
```

- Use the `:info` command to view an operator precedence and its associativity

```
ghci> :info (+)
class Num a where
  (+) :: a -> a -> a
  ...
    -- Defined in ‘GHC.Num’
infixl 6 +
ghci> :info (*)
class Num a where
  ...
  (*) :: a -> a -> a
  ...
    -- Defined in ‘GHC.Num’
infixl 7 *
ghci> :info (^)
(^) :: (Num a, Integral b) => a -> b -> a
    -- Defined in ‘GHC.Real’
infixr 8 ^
```

- Note: `infixl` (left) vs `infixr` (right) associativity

Constant and Variable

- `pi` is provided but the Euler's number `e` is not:

```
ghci> pi
3.141592653589793
ghci> e

<interactive>:3:1: error: Variable not in scope: e
```

- The function `exp x` returns e^x

```
ghci> exp 1
2.718281828459045
```

- Use `let` construct in `GHCi` to define the value of `e`:

```
ghci> let e = exp 1
ghci> e
2.718281828459045
```

- For now, if we need to define a variable to be available under the `GHCi` environment, we have to use `let`
- `let` will only be used in a source file in a slightly different circumstances

Lists

- A list in Haskell is surrounded by square brackets

```
ghci> [12, 4, 1, 99]
[12,4,1,99]
ghci> [True]
[True,False,False,True]
ghci> [True, False, False, True, False]
[True,False,False,True,False]
ghci> []
[]
```

- Elements must have the same type and separated by commas
- [ ] is the empty list
- Haskell supports **enumeration notation** on list:

```
ghci> [4..20]
[4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> [4,8..20]
[4,8,12,16,20]
ghci> [22,18..3]
[22,18,14,10,6]
ghci> [1.0, 1.5..5.0]
[1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0]
```

Operators on Lists

- Use (++) to concatenate two lists:

```
ghci> [5, 9] ++ [1, 23, 4]
[5,9,1,23,4]
ghci> [True, False] ++ [True, True, False]
[True,False,True,True,False]
```

- Use the cons operator (:) to add an element in front of a list:

```
ghci> 12 : [44, 29]
[12,44,29]
ghci> True : []
[True]
```

- Make sure to check the type of lists:

```
ghci> [1, 2] ++ [False, True]

<interactive>:28:2: error:
    • No instance for (Num Bool) arising from the literal ‘1’
    • In the expression: 1
      In the first argument of ‘(++)’, namely ‘[1, 2]’
      In the expression: [1, 2] ++ [False, True]
ghci> True : [12]

<interactive>:27:9: error:
    • No instance for (Num Bool) arising from the literal ‘12’
    • In the expression: 12
      In the second argument of ‘(:)’, namely ‘[12]’
      In the expression: True : [12]
```

Strings in Haskell

- Haskell use the same string literals as in C and Java

```
ghci> "Haskell is fun"
"Haskell is fun"
```

- It supports escape characters such as `'\n'` and `'\t'`

```
ghci> "Hello\nHaskell"
"Hello\nHaskell"
ghci> putStrLn "Hello\nHaskell"
Hello
Haskell
```

- The function `putStrLn` print a string on a console screen just like `System.out.println()`
- Haskell use the same character literals as in C and Java

```
ghci> 'H'
'H'
ghci> '\n'
'\n'
```

- A string in Haskell is simply a list of characters

```
ghci> ['H', 'a', 's', 'k', 'e', 'l', 'l']
"Haskell"
```

String in Haskell

- A single character is not the same as a string containing one character:

```
ghci> 'a' == "a"

<interactive>:36:8: error:
    • Couldn't match expected type ‘Char’ with actual type ‘[Char]’
    • In the second argument of ‘(==)’, namely ‘"a"’
      In the expression: 'a' == "a"
      In an equation for ‘it’: it = 'a' == "a"
```

- `'a'` is a character but `"a"` is a string
- Since a string is a list, list operators can be used as expected:

```
ghci> "I love " ++ "Haskell"
"I love Haskell"
ghci> 'H':"askell"
"Haskell"
```

- `""` is the empty string

```
ghci> ""
""
```

Types

- To view the type of a Haskell expression, use `:t` command:

```
ghci> :t 123
123 :: Num p => p
ghci> :t True
True :: Bool
ghci> :t "Haskell"
"Haskell" :: [Char]
```

- The `x :: y` means the expression `x` has the type `y`
- `x :: [y]` means the expression `x` has the type list where each element has the type `y`
- `:t` can also be used to check the type of a function as well

```
ghci> :t (^)
(^) :: (Integral b, Num a) => a -> b -> a
ghci> :t sqrt
sqrt :: Floating a => a -> a
```

- `Integral b`, `Num a`, and `Floating a` are similar to Java `Interface` and will be dicussed in detail later

The `it` Variable

- The `it` variable is the expression that we just evalueated

```
ghci> 1 + 2
3
ghci> it
3
ghci> "Hello" ++ " World!!!"
"Hello World!!!"
ghci> it
"Hello World!!!"
ghci> it ++ " from Haskell"
"Hello World!!! from Haskell"
```

- Be careful, the type of `it` depends on the last evaluated expression

```
ghci> 5 * 12
60
ghci> :t it
it :: Num a => a
ghci> "Hello"
"Hello"
ghci> :t it
it :: [Char]
ghci> [1,2] ++ [3,4]
[1,2,3,4]
ghci> :t it
it :: Num a => [a]
```

GHCi Commands

- Use `:?` to show all available commands and options
- In a UNIX like environment, to repeat the last **expression**, use the up arrow
- The command `:` only repeats the last command (not expression)
- Use `:set prompt str` to set the GHCi prompt to the string `str`
- Use `:q` or `:quit` to exit GHCi
- Before next lecture:
  - Make sure to get familiar with GHCi
  - Unix environment is preferred but not necessary