

# 19CSE313 – PRINCIPLES OF PROGRAMMING LANGUAGES

Haskell Environment & ghci

---

# HASKELL INTERPRETERS AND COMPILERS

- Language with many implementations - two are widely used
  - Hugs interpreter - for teaching
  - Glasgow Haskell Compiler (GHC) - for real applications
    - ✓ Compiles to native code,
    - ✓ Supports parallel execution, and
    - ✓ Provides useful performance analysis and debugging tools.
  - ❖ GHC Has three main components:
    - ghc - An optimizing compiler that generates fast native code
    - ghci - An interactive interpreter and debugger
    - runghc - A program for running Haskell programs as scripts, without needing to compile them first

# DOWNLOAD HASKELL

- <https://www.haskell.org/downloads/>

# ghci - THE INTERACTIVE INTERPRETER

- Allows to
  - Enter and evaluate Haskell expressions,
  - Explore modules and
  - Debug code
- Includes a standard library of useful functions - loaded and ready to use.
- To use definitions from other modules, we must load them into ghci, using the `:module` command:  
  
ghci> **:module + Data.Ratio**
  - We can now use the functionality of the `Data.Ratio` module, which lets us work with rational numbers (fractions).
- When we load other modules or source files, they will also show up in the prompt.
- Getting help
  - Enter `:?` at the ghci prompt, it will print a long help message.

# ghci IN ACTION

```
C:\> Command Prompt - ghci
Microsoft Windows [Version 10.0.19043.1415]
(c) Microsoft Corporation. All rights reserved.

C:\Users\admin>d:

D:\>dir
Volume in drive D is Media
Volume Serial Number is AE41-898B

Directory of D:\

06-03-2017  09:22 PM    <DIR>          Bills paid online
30-06-2020  08:13 PM         1,297,301 Inner Engineering - A Yogi's Guide to Joy ( PDFDrive.com ).pdf
30-06-2020  08:16 PM         3,311,434 Life and Death in One Breath ( PDFDrive.com ).pdf
26-07-2017  12:41 PM    <DIR>          Matlab
17-12-2021  09:36 AM    <DIR>          My Healing Sound
17-12-2021  09:27 AM       99,774,152 My Healing Sound-20211217T035548Z-001.zip
16-05-2018  09:22 PM    <DIR>          Office2003
22-12-2019  11:02 AM    <DIR>          Photos
20-12-2021  01:45 PM    <DIR>          PPL
14-07-2021  03:19 PM    <DIR>          Songs
30-06-2020  08:19 PM         812,623 Taste of Well Being ( PDFDrive.com ).pdf
01-03-2019  02:14 PM       1,471,659,710 www.TamilRockerss.bz - Petta (2019)[Tamil 720p HDRip - x264 - 5.1 AC3 - 1.4G.mkv
06-02-2019  10:24 PM       760,855,224 www.TamilRockerss.ch - K.G.F Chapter 1 (2018)[Tamil Proper - HDRip - XviD - .avi
        6 File(s)  2,337,710,444 bytes
        7 Dir(s)  94,049,103,872 bytes free

D:\>>cd PPL

D:\PPL>ghci
GHCi, version 9.2.1: https://www.haskell.org/ghc/  :? for help
ghci>
```

# BASIC INTERACTION WITH `ghci` as a Calculator

- Basic arithmetic works similarly to languages such as C and Python
- Expressions are written in *infix* form, where an operator appears between its operands:

```
ghci> 2+2
```

```
4
```

```
ghci> 12*12
```

```
144
```

```
ghci> 9.0/4.0
```

```
2.25
```

```
ghci>
```

---

# SOME EXAMPLES OF BASIC ARITHMETIC IN ghci

- Expressions can be written in prefix form also:

```
ghci> (+) 2 2
```

```
4
```

Note: In Prefix form, the operator must be enclosed in parenthesis

- Integer Exponentiation:
  - (^) provides integer exponentiation

Example:

```
ghci> 2^2
```

```
4
```

# NEGATIVE NUMBERS

- Unary Minus

- Example:

```
ghci> -3
```

```
-3
```

- Using Negative Number in Infix Expression:

```
ghci> 2 + -3
```

```
<interactive>:9:1: error:
```

Precedence parsing error

cannot mix '+' [infixl 6] and prefix '-' [infixl 6] in the same  
infix expression

```
ghci>
```



# NEGATIVE NUMBER IN INFIX EXPRESSION

- Negative Numbers must be enclosed in parenthesis in an infix expression to avoid ambiguity

```
ghci> 2 + (-3)
```

```
-1
```

```
ghci> 3 + (-(13 * 37))
```

```
-478
```

# USAGE OF SPACE AND NEGATIVE NUMBERS

- Most of the time white spaces (Blank, Tab) can be omitted in expressions

- Example 1

```
ghci> 2*3
```

```
6
```

- Example 2

```
ghci> 2*-3
```

```
<interactive>:13:2: error:
```

```
* Variable not in scope: (*-) :: t0 -> t1 -> t
```

```
* Perhaps you meant one of these:
```

```
  `*>' (imported from Prelude), `**' (imported from Prelude),
```

```
  `*' (imported from Prelude)
```

Haskell reads `*-` as a single operator. Could be used if defined.

Solution:

```
ghci> 2*(-3)
```

```
-6
```

# LOGICAL OPERATORS AND BOOLEAN VALUES

- The values of Boolean logic in Haskell are
  - True
  - False
- Capitalization of these names is important.
- Logical Operators:
  - (&&) is logical “and”
  - (||) is logical “or”
  - not - a function is used for negation
  - Example:

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

```
False
```

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

```
True
```

```
ghci> not True
```

```
False
```

# USAGE OF 0 AS FALSE AND NON ZERO VALUE AS TRUE

- Example:

```
ghci> True && 1
```

```
<interactive>:19:9: error:
```

- \* No instance for (Num Bool) arising from the literal `1'

- \* In the second argument of `(&&)', namely `1'

In the expression: True && 1

In an equation for `it': it = True && 1

Haskell does not, nor does it consider a nonzero value to be True

# RELATIONAL OPERATORS IN HASKELL

- Most of Haskell's comparison operators are similar to those used in C and similar languages
- Examples:

ghci> 1==1        (Equality operator)

True

ghci> 2<3        (Less than operator)

True

ghci> 4>=3.99    (Greater than or Equal to operator)

True

ghci> 2/=3        (Not equal to Operator)

True

# OPERATOR PRECEDENCE AND ASSOCIATIVITY

- Haskell assigns numeric precedence values to operators
- lowest precedence is 1 and Highest precedence is 9
- A higher-precedence operator is applied before a lower precedence operator in an expression.
- ghci's :info command can be used to check the precedence level of individual operators
- Example:

```
ghci> :info (+)
```

```
type Num :: * -> Constraint
```

```
class Num a where
```

```
  (+) :: a -> a -> a
```

```
  ...
```

```
    -- Defined in `GHC.Num'
```

```
infixl 6 +
```

```
ghci> :info (*)
```

```
type Num :: * -> Constraint
```

```
class Num a where
```

```
  ...
```

```
  (*) :: a -> a -> a
```

```
  ...
```

```
    -- Defined in `GHC.Num'
```

```
infixl 7 *
```

# OVERRIDING PRECEDENCE USING ( )

- We can use parentheses to explicitly group parts of an expression, and
- Precedence allows us to omit a few parentheses.
- Example:

```
ghci> 1 + (4 * 4)
```

```
17
```

```
ghci> 1 + 4 * 4
```

```
17
```

Both Expressions are  
Equivalent

Combination of precedence  
and associativity rules are  
usually referred to as *fixity*  
rules

- Since (\*) has a higher precedence than (+),  $1 + 4 * 4$  is evaluated as  $1 + (4 * 4)$ , and not  $(1 + 4) * 4$ .
- Associativity of operators
  - Determines whether an expression containing multiple uses of an operator is evaluated from left to right or right to left
  - The (+) and (\*) operators are left associative, which is represented as infixl in :info
  - A right associative operator is displayed with infixr

Example:

```
ghci> :info (^)
```

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

```
infixr 8 ^
```

# CONSTANTS AND VARIABLES

- Haskell's Prelude, the standard library, defines at least one well known mathematical constant for us:

```
ghci> pi
```

```
3.141592653589793
```

- Haskell's coverage of mathematical constants is not comprehensive

```
ghci> e      [where e is the Euler Number]
```

```
<interactive>:4:1: error: Variable not in scope: e
```

- We may have to define it ourselves



# DEFINING e USING LET CONSTRUCT AND ITS USAGE

## Example 1: Defining e using let

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

```
ghci> e
```

```
2.718281828459045
```

- exp is the in-built exponential function
- Unlike other languages haskell does not require a parenthesis around the arguments to a function

## Example 2: Using the defined e in an arithmetic expression

```
ghci> (e ** pi) - pi
```

```
19.999099979189467
```

# STRINGS AND CHARACTERS

- A text string is surrounded by double quotes  
Sequences

```
ghci> "This is a string."
```

```
"This is a string."
```

- Example for `\n`

```
ghci> putStrLn "Here's a newline -->\n<-- See?"
```

```
Here's a newline -->
```

```
<-- See?
```

- The `putStrLn` function prints a string.

- Single Character

```
ghci> 'a'
```

```
'a'
```

## Escape

Escape	Unicode	Character
<code>\0</code>	U+0000	Null character
<code>\a</code>	U+0007	Alert
<code>\b</code>	U+0008	Backspace
<code>\f</code>	U+000C	Form feed
<code>\n</code>	U+000A	Newline (linefeed)
<code>\r</code>	U+000D	Carriage return
<code>\t</code>	U+0009	Horizontal tab
<code>\v</code>	U+000B	Vertical tab
<code>\"</code>	U+0022	Double-quote
<code>\&amp;</code>	<i>n/a</i>	Empty string
<code>\'</code>	U+0027	Single quote
<code>\\</code>	U+005C	Backslash

# LIST OF CHARACTERS AND LISTS

```
ghci> let a = ['l', 'o', 't', 's', ' ', 'o', 'f', ' ', 'w', 'o', 'r', 'k']
```

```
ghci> a
```

```
"lots of work"
```

String is actually a list of characters

```
ghci> a=="lots of work"
```

```
True
```

The Empty String "" or []

```
ghci> let b = ""
```

```
ghci> b==[]
```

```
True
```

# LISTS

- A list is surrounded by square brackets; the elements are separated by commas

```
ghci> [1,2,3]
```

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

- Commas are separators, not terminators

```
ghci> [1,2,]
```

```
<interactive>:28:6: error: parse error on input `]'
```

- A list can be of any length. An empty list is written []:

```
ghci> []
```

```
[]
```

# LISTS

- All elements of a list must be of the same type.

- Example 1:

```
ghci> ["foo", "bar", "baz", "quux", "fnord", "xyzzzy"]  
["foo","bar","baz","quux","fnord","xyzzzy"]
```

- Example 2:

```
ghci> [True, False, "testing"]
```

```
<interactive>:32:15: error:
```

```
* Couldn't match type `[Char]' with `Bool'
```

```
Expected: Bool
```

```
Actual: String
```

```
* In the expression: "testing"
```

```
In the expression: [True, False, "testing"]
```

```
In an equation for `it': it = [True, False, "testing"]
```

# THE .. ENUMERATION NOTATION FOR LISTS

- Haskell fills in the rest of the items in the list when enumeration notation `..` is used.
- Example:

```
ghci> [1..10]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

- Can be used only for types whose elements can be enumerated.
- Makes no sense for text strings
- Example:

```
ghci> ["foo".. "quux"]
```

```
<interactive>:35:1: error:
```

```
* No instance for (Enum String)
```

```
arising from the arithmetic sequence `\"foo\" .. \"quux\"`
```

```
* In the expression: [\"foo\" .. \"quux\"]
```

```
In an equation for `it`: it = [\"foo\" .. \"quux\"]
```

use of range notation gives  
a *closed interval* containing  
both end points

# SPECIFYING THE SIZE OF THE STEP (OPTIONAL)

- Provide the first two elements, followed by the value at which to stop
- Example:

```
ghci> [1.0,1.25..2.0]
```

```
[1.0,1.25,1.5,1.75,2.0]
```

```
ghci> [1,4..15]
```

```
[1,4,7,10,13]
```

```
ghci> [10,9..1]
```

```
[10,9,8,7,6,5,4,3,2,1]
```

# LIST OPERATIONS

- Concatenation operator (++):

```
ghci> [3,1,3] ++ [3,7]
```

```
[3,1,3,3,7]
```

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

```
[False,True,True]
```

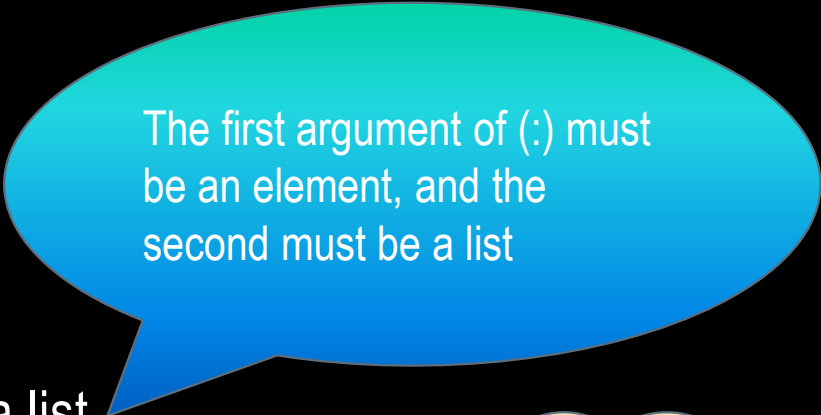
- “cons” operator (:):
- Short form for construct
- Adds an element to the front of a list

```
ghci> 1 : [2,3]
```

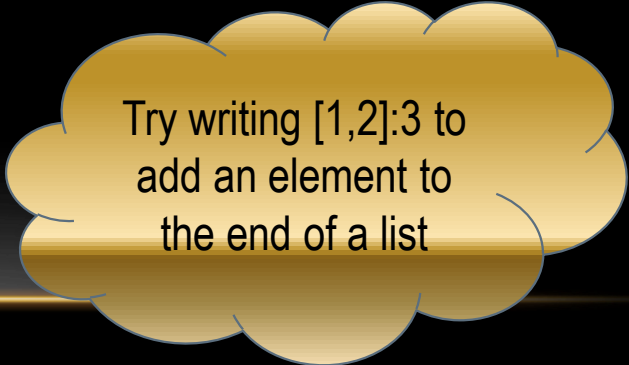
```
[1,2,3]
```

```
ghci> 1 : []
```

```
[1]
```



The first argument of (:) must be an element, and the second must be a list



Try writing [1,2]:3 to add an element to the end of a list



# TYPES

- Haskell type names must start with an uppercase letter, and variable names must start with a lowercase letter
- Setting ghci to print the type of an expression using +t:

- Example 1:

```
ghci> 'c'
```

```
'c'
```

Before giving :set +t

- Example 2:

```
ghci> :set +t
```

```
ghci> 'c'
```

```
'c'
```

```
it :: Char
```

After giving :set +t

# THE CRYPTIC WORD “*it*”

- The word “*it*” is the name of a special variable, in which `ghci` (not Haskell ! ) stores the result of the last expression we evaluated.

- Another Example:

```
ghci> "Hello"
```

```
"Hello"
```

```
it :: String
```

- Decoding the above Example:
- It tells about the special variable `it`
- The text of the form `x :: y` is read as “the expression `x` has the type `y`.”
- In the above expression “`it`” has the type of `String`

# OTHER HANDY USES OF “*it*”

- Example 1:

```
ghci> "Hello"
```

```
"Hello"
```

```
it :: String
```

```
ghci> it ++ "Haskell"
```

```
"HelloHaskell"
```

```
it :: [Char]
```

The result of the expression we just evaluated can be used in a new expression

# OTHER HANDY USES OF “*it*”

- Example 2:

```
ghci> it++3
```

```
<interactive>:9:5: error:
```

- \* No instance for (Num [Char]) arising from the literal `3'
- \* In the second argument of `(++)', namely `3'

In the expression: `it ++ 3`

In an equation for ``it'`: `it = it ++ 3`

```
ghci> it
```

```
"HelloHaskell"
```

```
it :: [Char]
```

```
ghci> it++"World"
```

```
"HelloHaskellWorld"
```

```
it :: [Char]
```

When evaluating an expression, ghci won't change the value of `it` if the evaluation fails.

# SOME MORE EXAMPLES

```
ghci> 5
```

```
5
```

```
it :: Num a => a
```

```
ghci> 5.5
```

```
5.5
```

```
it :: Fractional a => a
```

```
ghci> 7^80
```

```
4053621559714438683206586610901667380087522225101208374619245  
4448001
```

```
it :: Num a => a
```

Haskell's integer type is named Num. The size of an Integer value is bounded only by your system's memory capacity.

# RATIONAL NUMBERS

- Don't look quite the same as integers
- To construct a rational number, we use the (%) operator.

```
ghci> :m +Data.Ratio      (:m is the short form for :module command)
```

```
ghci> 11 % 29
```

```
11 % 29
```

```
it :: Integral a => Ratio a
```

```
ghci> it
```

```
11 % 29
```

```
it :: Integral a => Ratio a
```



Read as Ratio of Integer

# RATIOS OF NON-INTEGRAL TYPE

```
ghci> 3.14 % 8
```

```
<interactive>:19:1: error:
```

\* Ambiguous type variable ``a0'` arising from a use of ``print'`  
prevents the constraint ``(Show a0)'` from being solved.

Probable fix: use a type annotation to specify what ``a0'` should be.

These potential instances exist:

instance Show a => Show (Ratio a) -- Defined in ``GHC.Real'`

instance Show Ordering -- Defined in ``GHC.Show'`

instance Show a => Show (Maybe a) -- Defined in ``GHC.Show'`

...plus 24 others

...plus 14 instances involving out-of-scope types

(use `-fprint-potential-instances` to see them all)

\* In a stmt of an interactive GHCi command: print it

# RATIOS OF NON-INTEGRAL TYPE

```
ghci> 1.2 % 3.4
```

```
<interactive>:20:1: error:
```

\* Ambiguous type variable ``a0'` arising from a use of ``print'`

prevents the constraint ``(Show a0)'` from being solved.

Probable fix: use a type annotation to specify what ``a0'` should be.

These potential instances exist:

```
instance Show a => Show (Ratio a) -- Defined in `GHC.Real'
```

```
instance Show Ordering -- Defined in `GHC.Show'
```

```
instance Show a => Show (Maybe a) -- Defined in `GHC.Show'
```

...plus 24 others

...plus 14 instances involving out-of-scope types

(use `-fprint-potential-instances` to see them all)

\* In a stmt of an interactive GHCi command: print it



# TURNING OFF THE TYPE INFORMATION

- The extra type information can be turned off at any time, using the `:unset` command

- Example

```
ghci> :unset +t
```

```
ghci> 2
```

```
2
```

- Type command:

```
ghci> :type 'a'
```

```
'a' :: Char
```

```
ghci> "Hi"
```

```
"Hi"
```

```
ghci> :type it
```

```
it :: String
```

The `:type` command prints the type information for any expression including it

Type doesn't actually evaluate the expression; it checks only its type and prints it

# TYPE SYSTEMS

- Every expression and function in Haskell has a *type*
- For example, the value `True` has the type `Bool`, while the value `"foo"` has the type `String`
- The type of a value indicates that it shares certain properties with other values of the same type
- For example, we can add numbers and concatenate lists; these are properties of those types
- An expression has type `X`, or is of type `X`
- Aspects of Haskell's Type System
  - There are three interesting aspects to types in Haskell
    - ✓ They are strong,
    - ✓ They are static and
    - ✓ They can be automatically inferred

# STRONG TYPES

- The type system guarantees that a program cannot contain certain kinds of errors which doesn't make sense such as
  - ❖ Using an integer as a function or
  - ❖ If a function expects to work with integers and if a string is passed, it will be rejected by the Haskell compiler
- Haskell will not automatically coerce values from one type to another and will raise a compilation error in such a situation
- Explicit type conversion must be performed using coercion functions
- The benefit of strong typing is that it catches real bugs in our code before they can cause problems

# TYPE INFERENCE

- The ability of a Haskell compiler to automatically deduce the types of almost all expressions in a program is known as *Type Inference*

# STATIC TYPES

- A *static* type system means that the compiler knows the type of every value and expression at compile time

```
ghci> True && "false"
```

```
<interactive>:27:10: error:
```

```
* Couldn't match type `[Char]' with `Bool'
```

```
Expected: Bool
```

```
Actual: String
```

```
* In the second argument of `(&&)', namely `"false"'
```

```
In the expression: True && "false"
```

```
In an equation for `it': it = True && "false"
```

A Haskell compiler or interpreter will detect when we try to use expressions whose types don't match, and reject our code with an error message before we run it ! ! !

- Haskell's combination of strong and static typing makes it impossible for type errors to occur at runtime.

# SOME COMMON BASIC TYPES

Type	Description
Char	Unicode character
Bool	Boolean logic. Possible values: True and False
Int	Signed, fixed-width integers. Exact range of values. depends on the system's longest "native" integer: 32-bit machine: 32 bits wide, 64-bit machine: 64 bits wide. Haskell standard guarantees only that an Int is wider than 28 bits.
Integer	Signed integer of unbounded size. Not used as often as Ints as more expensive in performance and space. Advantage: No overflow, so reliable results
Double	Floating-point numbers. Typically 64 bits wide. Uses the system's native floating-point representation

# TYPE INFERENCE EXAMPLE REVISITED

ghci> :type 'a'

'a' :: Char

**Automatic Type Inference**

ghci> 'a' :: Char

'a'

**Explicit type declaration**

ghci> [1,2,3] :: Int

<interactive>:30:1: error.

\* Couldn't match expected type `Int` with actual type `[a0]`

\* In the expression: `[1, 2, 3] :: Int`

In an equation for ``it'`: `it = [1, 2, 3] :: Int`

**Type Signature...  
But Wrong !!!**

# FUNCTION APPLICATION (PRE-DEFINED FUNCTIONS)

- To apply a function in Haskell, write the name of the function followed by its arguments
- Examples:

```
ghci> odd 3
```

```
True
```

```
ghci> compare 2 3
```

```
LT
```

- Function application has higher precedence than operator usage
- Example:

```
ghci> (compare 2 3) == LT
```

```
True
```

```
ghci> compare 2 3 == LT
```

```
True
```

# USAGE OF PARENTHESIS IN COMPOUND EXPRESSIONS

```
ghci> compare (sqrt 3) (sqrt 6)
```

```
LT
```

```
ghci> compare sqrt 3 sqrt 6
```

```
<interactive>:36:1: error:
```

- \* Couldn't match expected type `(a0 -> a0) -> t0 -> t'`

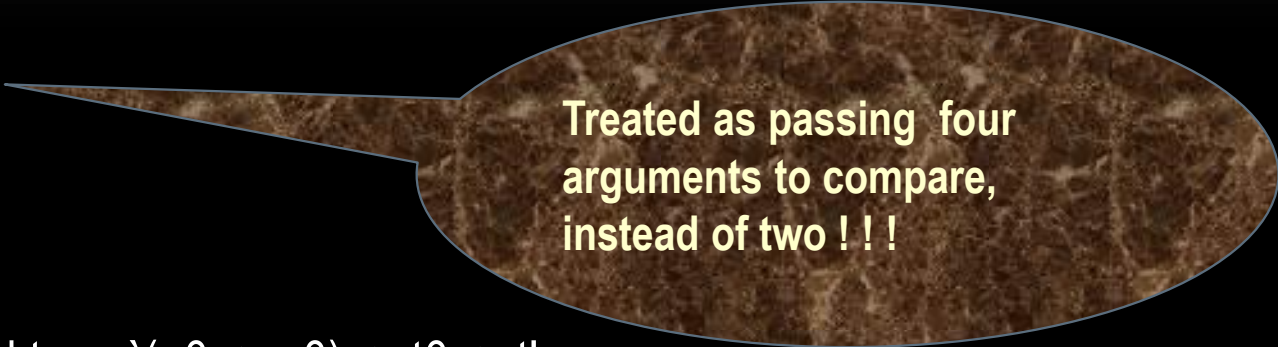
with actual type `'Ordering'`

- \* The function `'compare'` is applied to four value arguments,  
but its type `'(a1 -> a1) -> (a1 -> a1) -> Ordering'` has only two

In the expression: `compare sqrt 3 sqrt 6`

In an equation for `'it'`: `it = compare sqrt 3 sqrt 6`

- \* Relevant bindings include `it :: t` (bound at `<interactive>:36:1`)



Treated as passing four  
arguments to compare,  
instead of two !!!



# USEFUL COMPOSITE DATA TYPES: LISTS AND TUPLES

- The head function returns the first element of a list:

```
ghci> head [1,2,3,4]
```

```
1
```

- Its counterpart, tail, returns all but the head of a list:

```
ghci> tail [1,2,3,4]
```

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

```
ghci> tail [2,3,4]
```

```
[3,4]
```

```
ghci> tail [True,False]
```

```
[False]
```

```
ghci> tail "list"
```

```
"ist"
```

```
ghci> tail []
```

```
*** Exception: Prelude.tail: empty list
```

Because the values in a list can have any type, we call the list type *polymorphic* !!!

# TUPLE

- Fixed-size collection of values • • •
- Each value can have a different type
- A tuple is specified by enclosing its elements in parentheses, separated by commas.

As opposed to lists  
which can have any  
length with all the  
elements of same type !

- Example:

```
ghci> (1964, "Labyrinths")
```

```
(1964, "Labyrinths")
```

- The same above notation is used for writing a tuple's type:

```
ghci> :type (True, "hello")
```

```
(True, "hello") :: (Bool, String)
```

```
ghci> ()
```

```
()
```

# FUNCTIONS OVER LISTS AND TUPLES

- take - Given a number  $n$  and a list, take returns the first  $n$  elements of the list
- drop - returns all *but* the first  $n$  elements of the list

```
ghci> take 2 [1,2,3,4,5]
```

```
[1,2]
```

```
ghci> drop 3 [1,2,3,4,5]
```

```
[4,5]
```

- For tuples, the fst and snd functions return the first and second element of a pair, respectively:

```
ghci> fst (1,'a')
```

```
1
```

```
ghci> snd (1,'a')
```

```
'a'
```

# PASSING AN EXPRESSION TO A FUNCTION

- In Haskell, function application is left-associative
- For example: The expression `a b c d` is equivalent to `((a b) c) d`.
- To use one expression as an argument to another, use explicit parentheses for correct execution

Example:

```
ghci> head (drop 4 "azerty")
```

```
't'
```

```
ghci> head drop 4 "azerty"
```

```
<interactive>:57:6: error:
```

```
* Couldn't match expected type: [t0 -> String -> t]
```

```
with actual type: Int -> [a0] -> [a0]
```

```
* Probable cause: `drop' is applied to too few arguments
```

```
In the first argument of `head', namely `drop'
```

```
In the expression: head drop 4 "azerty"
```

```
In an equation for `it': it = head drop 4 "azerty"
```

```
* Relevant bindings include it :: t (bound at <interactive>:57:1)
```

# SEQUENCE OF FUNCTION CALLS - EXAMPLE

```
main = do
  print(odd 3)
  print(compare 2 3)
  print(head [1,2,3,4])
  print(tail [1,2,3,4])
  print(tail "list")
  take 2 [1,2,3,4,5]
  drop 3 [1,2,3,4,5]
```

# FUNCTION TYPES AND PURITY

- Example: The pre-defined function *lines* splits a string on line boundaries

```
ghci> lines "the quick\nbrown fox\njumps"

["the quick","brown fox","jumps"]
```
- Type Signature of a Function: gives a hint as to what the function might actually do
- Example:

```
ghci> :type lines

lines :: String -> [String]
```
- Lines takes one String, and returns many (a list of strings).
- The signature is read as “lines has the type String to list-of-String”
- `->` is read as “to,” which loosely translates to “returns.”

# EXAMPLES OF FUNCTION SIGNATURES

$f :: X \rightarrow Y$

- $f$  is a function taking arguments of type  $X$  and returning results of type  $Y$ .
- $\sin :: \text{Float} \rightarrow \text{Float}$
- $\text{age} :: \text{Person} \rightarrow \text{Int}$
- $\text{add} :: (\text{Integer}, \text{Integer}) \rightarrow \text{Integer}$
- $\text{logBase} :: \text{Float} \rightarrow (\text{Float} \rightarrow \text{Float})$
- Function Call:  
 $\sin 3.14$  or  $\sin (3.14)$  or  $\sin(3.14)$

# FUNCTIONAL COMPOSITION

- Suppose  $f :: Y \rightarrow Z$  and  $g :: X \rightarrow Y$  are two given functions. We can combine them into a new function

$$f . g :: X \rightarrow Z$$

- The above applies  $g$  to an argument of type  $X$ , giving a result of type  $Y$ , and then applies  $f$  to this result, giving a final result of type  $Z$ .
- In other words:

$$(f . g) x = f (g x)$$

- The order of composition is from right to left because we write functions to the left of the arguments to which they are applied.
- Example:

```
ghci> odd(sqr 3)
```

```
False
```

```
ghci> odd(mod 22 3)
```

```
True
```

```
ghci> odd(mod 20 3)
```

```
False
```



# SIDE EFFECTS

- Side Effect: Introduces a dependency between the global state of the system and the behavior of a function.
- Consider a function that reads and returns the value of a global variable.
- If some other code can modify that global variable, then the result of a particular application of our function depends on the current value of the global variable.
- The function has a side effect, even though it never modifies the variable itself.
- Side effects are essentially invisible inputs to, or outputs from, functions.

# PURE AND IMPURE FUNCTIONS

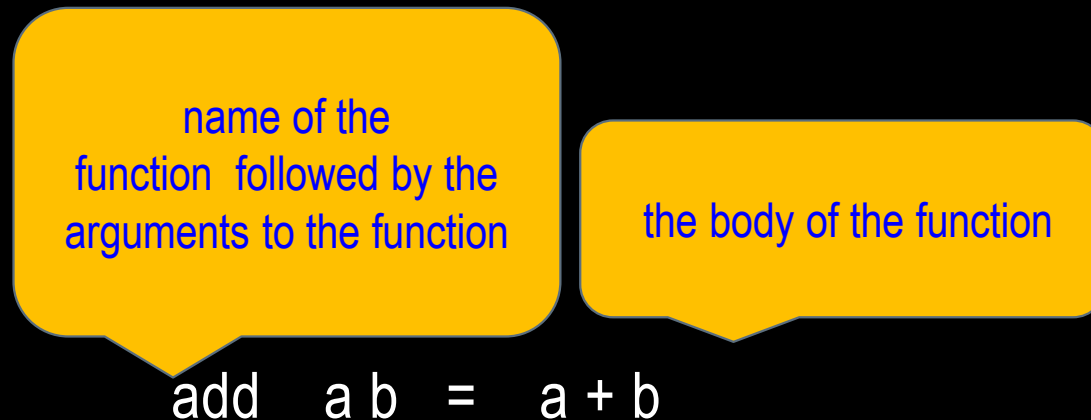
- In Haskell, the default is for functions to *not* have side effects: the result of a function depends only on the inputs that we explicitly provide.
- Pure Functions: Functions without Side effects
- Impure Functions: Functions with Side effects
- Identifying an Impure Function:
  - By reading the type signature
  - The type of the function's result will begin with IO
  - Example:

```
ghci> :type readFile
```

```
readFile :: FilePath -> IO String
```
- Haskell's type system prevents us from accidentally mixing pure and impure code

# HASKELL SOURCE FILES, AND WRITING SIMPLE FUNCTIONS

- Simple functions in Haskell
  - Haskell source files are usually identified with a suffix of .hs
  - A simple function named add.hs is defined as follows



- Haskell doesn't have a return keyword, because a function is a single expression, not a sequence of statements.
- The value of the expression is the result of the function.
- `=` symbol in Haskell code, it represents "meaning"—the name on the left is defined to be the expression on the right.

# RUNNING THE FUNCTION

- Save the source file
- Load it into ghci
- Use the add function straightaway
- Sample run of add.hs

```
ghci> :load add.hs
```

```
[1 of 1] Compiling Main           ( add.hs, interpreted )
```

```
Ok, one module loaded.
```

```
ghci> add 1 2
```

```
3
```

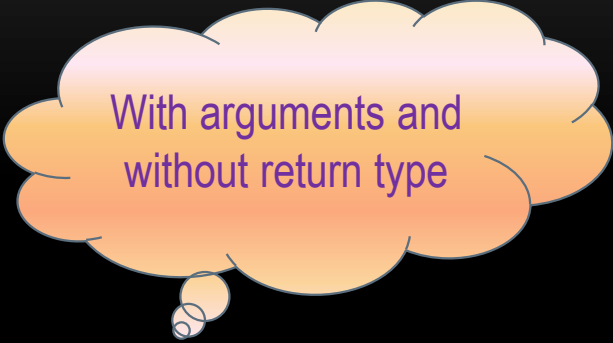
When we apply add to the values 1 and 2, the variables a and b on the lefthand side of our definition are given (or “bound to”) the values 1 and 2, so the result is the expression  $1 + 2$ .

# TYPES OF FUNCTIONS IN HASKELL



With arguments and  
with return type

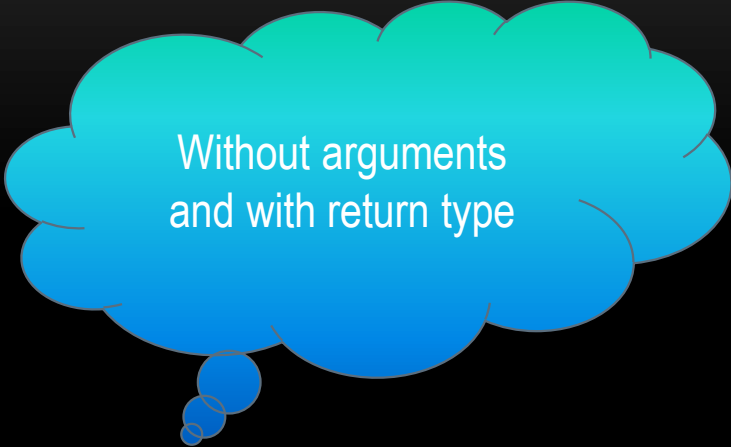
```
add :: Int -> Int -> Int
add x y = x + y
main = do
    print(add 4 5)
```



With arguments and  
without return type

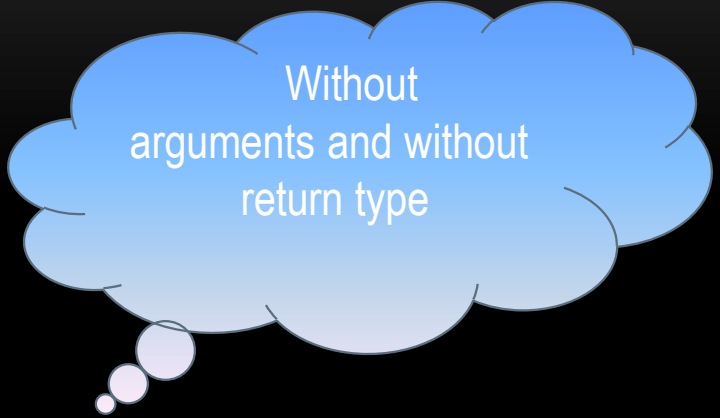
```
add :: Int -> Int -> IO()
add x y = print(x + y)
main = do
    add 3 4
```

# TYPES OF FUNCTIONS IN HASKELL



Without arguments  
and with return type

```
add :: ()->Int
add () = do
  let b = 5
  b+0
main = do
  print(add())
```



Without  
arguments and without  
return type

```
add = do
  let a= 5
  let b=5
  print(a+b)
main = do
  add
```

# VARIABLES IN HASKELL

- In Haskell, a variable provides a way to give a name to an expression.
- Once a variable is bound to (i.e., associated with) a particular expression, its value does not change.
- We can always use the name of the variable instead of writing out the expression, and we will get the same result either way
- For example, if we run the following tiny Python script, it will print the number 11:

```
x = 10
```

```
x = 11
```

```
# value of x is now 11
```

```
print x
```

# VARIABLES IN HASKELL

- In contrast, trying the equivalent in Haskell results in an error:

```
-- file: ch02/Assign.hs
```

```
x = 10
```

```
x = 11
```

```
ghci> :load assign.hs
```

```
[1 of 1] Compiling Main          ( assign.hs, interpreted )
```

```
assign.hs:3:1: error:
```

```
    Multiple declarations of `x'
```

```
    Declared at: assign.hs:2:1
```

```
              assign.hs:3:1
```

```
|
```

```
3 | x = 11
```

```
| ^
```

```
Failed, no modules loaded.
```



# CONDITIONAL EVALUATION

- Haskell has an if expression: `if test then expr1 else expr2`
- An expression of type `Bool`, immediately following the `if`. We refer to this as a predicate.
- A `then` keyword, followed by another expression. This expression will be used as the value of the if expression if the predicate evaluates to `True`.
- An `else` keyword, followed by another expression. This expression will be used as the value of the if expression if the predicate evaluates to `False`.
- The expressions that follow the `then` and `else` keywords as “branches.”
- The branches must have the same types; the if expression will also have this type. An expression such as `if True then 1 else "foo"` has different types for its branches, so it is ill typed and a compiler or interpreter will reject

# CONDITIONAL EVALUATION

- Example:

`doubleSmallNumber x = if x > 100`

`then x`

`else x*2`

- `if even x then y else (x + y)`

- `ghci> :type if 1==0 then 'a' else "a"`

`<interactive>:1:23:`

`Couldn't match expected type `Char' with actual type `[Char]'`

`In the expression: "a"`

`In the expression: if 1 == 0 then 'a' else "a"`

- GHCi expects the types of `expr1` and `expr2` in a conditional expression `if test then expr1 else expr2` to be the same.

# FUNCTION EVALUATION

- Example:

isOdd  $n = \text{mod } n \ 2 == 1$  [mod - standard modulo function]

- Consider the evaluation of  $(1+2)$

- In usual languages:

- First, evaluate the subexpression  $1 + 2$ , to give 3
    - Apply the odd function with  $n$  bound to 3
    - Finally, evaluate  $\text{mod } 3 \ 2$  to give 1, and  $1 == 1$  to give True

- Strict and Non-Strict Evaluation:

- In languages that uses *strict* evaluation, the arguments to a function are evaluated before the function is applied.
  - Haskell uses *non-strict* evaluation

# FUNCTION EVALUATION

- Consider the evaluation of the function `sqr(3+4)` where `sqr` is defined as:

`sqr :: Integer -> Integer`

`sqr x = x*x`

- There are basically two ways to reduce the expression `sqr (3+4)` to its simplest possible form, namely 49.
- Either evaluate `3+4` first, or apply the definition of `sqr` first:

```
sqr (3+4)
= sqr 7
= let x = 7 in x*x
= 7*7
= 49
```

```
sqr (3+4)
= let x = 3+4 in x*x
= let x = 7 in x*x
= 7*7
= 49
```

- The number of reductions is same in both cases, but order is different
- Method on left is strict / eager evaluation and method on the right is lazy evaluation

# HASKELL'S NON-STRICT OR LAZY EVALUATION

- In Haskell, the subexpression  $1 + 2$  is *not* reduced to the value 3
- Instead, a “promise” is created that when the value of the expression `isOdd (1 + 2)` is needed, it will be able to compute it
- Thunk: The record used to track an unevaluated expression
- A thunk is created and the actual evaluation is deferred until it is really needed
- If the result of this expression is never subsequently used, we will not compute its value at all.

# LAZY EVALUATION - EXAMPLE

```
fst (sqr 1,sqr 2)
= fst (1*1,sqr 2)
= fst (1,sqr 2)
= fst (1,2*2)
= fst (1,4)
= 1
```

```
fst (sqr 1,sqr 2)
= let p = (sqr 1,sqr 2)
  in fst p
= sqr 1
= 1*1
= 1
```

$\text{fst } (x,y) = x$

- Under eager evaluation the value `sqr 2` is computed, while under lazy evaluation that value is not needed and is not computed.

# LAZY EVALUATION - EXAMPLE

Consider the definitions:

`infinity :: Integer`

`infinity = 1 + infinity`

`three :: Integer -> Integer`

`three x = 3`

Evaluating `infinity` will cause GHCi to go into a long, silent think trying to compute  $1 + (1 + (1 + (1 + \dots$  until eventually it runs out of space and returns an error message. The value of `infinity` is  $\perp$ .

There are two ways to evaluate `three infinity`

```
three infinity
= three (1+infinity)
= three (1+(1+infinity))
= ...
```

```
three infinity
= let x = infinity in 3
= 3
```

- Here eager evaluation gets stuck in a loop trying to evaluate `infinity`, while lazy evaluation returns the answer 3 at once.
- We don't need to evaluate the argument of `three` in order to return 3.

# RECURSION

`factorial :: Integer -> Integer`

`factorial n = fact (n,1)`

`fact :: (Integer,Integer) -> Integer`

`fact (x,y) = if x==0 then y else fact (x-1,x*y)`

```
factorial 3
= fact (3,1)
= fact (3-1,3*1)
= fact (2,3)
= fact (2-1,2*3)
= fact (1,6)
= fact (1-1,1*6)
= fact (0,6)
= 6
```

```
factorial 3
= fact (3,1)
= fact (3-1,3*1)
= fact (2-1,2*(3*1))
= fact (1-1,1*(2*(3*1)))
= 1*(2*(3*1))
= 1*(2*3)
= 1*6
= 6
```

The number of reduction steps is basically the same but lazy evaluation requires much more space to achieve the answer ! The expression  $1*(2*(3*1))$  is built up in memory before being evaluated.

The expression  $1*(2*(3*1))$  is built up in memory before being evaluated.



# PROS AND CONS OF LAZY EVALUATION

- On the plus side, Lazy evaluation terminates whenever *any* reduction order terminates; it never takes more steps than eager evaluation, and sometimes infinitely fewer.
- On the minus side, it can require a lot more space and it is more difficult to understand the precise order in which things happen.
- While Haskell uses lazy evaluation, ML (another popular functional language) uses eager evaluation.
- The function `three` is non-strict, while `(+)` is strict in both arguments.
- Because Haskell uses lazy evaluation we can define non-strict functions.
- Hence Haskell is referred to as a *non-strict* functional language.

# CONDITIONAL EVALUATION – AN EXAMPLE WITH DROP

```
ghci> drop 2 "foobar"
```

```
"obar"
```

```
ghci> drop 4 "foobar"
```

```
"ar"
```

```
ghci> drop 4 [1,2]
```

```
[]
```

```
ghci> drop 0 [1,2]
```

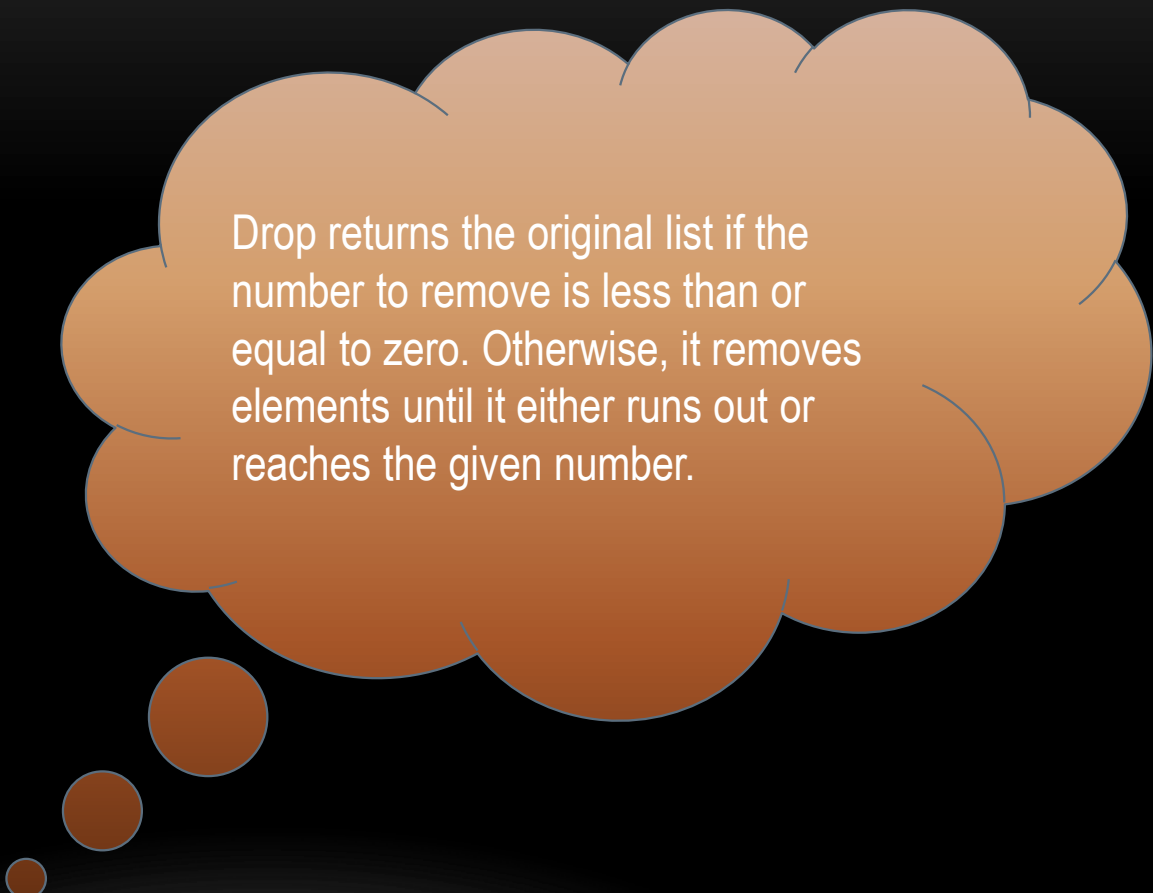
```
[1,2]
```

```
ghci> drop 7 []
```

```
[]
```

```
ghci> drop (-2) "foo"
```

```
"foo"
```



Drop returns the original list if the number to remove is less than or equal to zero. Otherwise, it removes elements until it either runs out or reaches the given number.

# MYDROP FUNCTION

- Uses Haskell's if expression to decide what to do

-- file: ch02/myDrop.hs

```
myDrop n xs = if n <= 0 || null xs
```

```
  then xs
```

```
  else myDrop (n - 1) (tail xs)
```

null function below checks  
whether a list is empty !

xs - common naming pattern  
for lists !!!

Indentation is important: it  
*continues* an existing  
definition, instead of starting  
a new one.  
**Don't omit the indentation !!!**

# EXECUTING MYDROP

- Save the Haskell function in a file named *myDrop.hs*, then load it into ghci:

```
ghci> :load myDrop.hs
```

```
[1 of 1] Compiling Main ( myDrop.hs, interpreted )
```

```
Ok, modules loaded: Main.
```

```
ghci> myDrop 2 "foobar"
```

```
"obar"
```

```
ghci> myDrop 4 "foobar"
```

```
"ar"
```

```
ghci> myDrop 4 [1,2]
```

```
[]
```

```
ghci> myDrop 0 [1,2]
```

```
[1,2]
```

```
ghci> myDrop 7 []
```

```
[]
```

```
ghci> myDrop (-2) "foo"
```

```
"foo"
```

# I/O IN HASKELL – A SIMPLE EXAMPLE

```
main = do
```

```
  putStrLn "Hello, what's your name?"
```

```
  name <- getLine
```

```
  putStrLn ("Hey " ++ name ++ ", you rock!")
```

putStrLn writes out a String, followed by an end-of-line character

getLine reads a line from standard input and binds via <- to the name inpStr

```
ghci> :l nameio.hs
```

```
[1 of 1] Compiling Main
```

```
Ok, one module loaded.
```

```
ghci> main
```

```
Hello, what's your name?
```

```
John
```

```
Hey John, you rock!
```

( nameio.hs, interpreted

list concatenation operator ++ can join the input string with our own text

# TYPES OF PUTSTRLN AND GETLINE

```
ghci> :type putStrLn
```

```
putStrLn :: String -> IO ()
```

```
ghci> :type getLine
```

```
getLine :: IO String
```

- Both of these types have IO in their return value
- They may have side effects, or they may return different values even when called with the same arguments, or both.
- The () is an empty tuple (pronounced “unit”), indicating that there is no return value from putStrLn.

# A SIDE-EFFECT EXAMPLE WITH PUTSTRLN:

```
ghci> let writefoo = putStrLn "foo"
```

```
ghci> writefoo
```

```
foo
```

- The output foo is not a return value from putStrLn.
- It's the side effect of putStrLn actually writing foo to the terminal.
- ghci actually executed writefoo.
- This means that, when given an I/O action, ghci will perform it for you on the spot.

# AN EXAMPLE OF CALLING PURE CODE FROM WITHIN AN I/O ACTION

```
name2reply :: String -> String
```

```
name2reply name =
```

```
"Pleased to meet you, " ++ name ++ ".\n" ++
```

```
"Your name contains " ++ show (length name) ++ " characters."
```

A Regular Haskell function, obeys all the rules : always returns the same result when given the same input, has no side effects, operates lazily. It uses other Haskell functions: (++) , show, and length.

```
main :: IO ()
```

```
main = do
```

```
  putStrLn "Greetings once again. What is your name?"
```

```
  inpStr <- getLine
```

```
  let outStr = name2reply inpStr
```

```
  putStrLn outStr
```

```
ghci> :l io1.hs
[1 of 1] Compiling Main           ( io1.hs, interpreted )
Ok, one module loaded.
ghci> main
Greetings once again. What is your name?
John
Pleased to meet you, John.
Your name contains 4 characters.
```



# READING A NUMBER IN HASKELL

```
main = do
  putStrLn "Enter a number:"
  x <- getLine
  let num = read x :: Int
  print(num)
```

```
ghci> main
Enter a number:
4
4
```

# CHECKING DATA CONVERSION

```
import Data.Typeable(typeOf)
```

```
main = do
```

```
  x <- getLine
```

```
  print(typeOf(x))  -- display the type before conversion
```

```
  let num = read x:: Int
```

```
  print(typeOf(num)) --display the type after conversion
```

```
ghci> :l numreadtype.hs
```

```
[1 of 1] Compiling Main
```

```
( numreadtype.hs, interpreted )
```

```
Ok, one module loaded.
```

```
ghci> main
```

```
5
```

```
[Char]
```

```
Int
```

# CUSTOMISED STRING TO DATATYPE FUNCTION

```
str2Int::String->Int    --custconvint.hs
```

```
str2Int x = read x :: Int
```

```
main = do
```

```
  putStrLn ("Enter number1:")
```

```
  input1 <- getLine
```

```
  let num1 = str2Int input1
```

```
  print(num1)
```

```
  putStrLn ("Enter number2:")
```

```
  input2 <- getLine
```

```
  let num2 = str2Int input2
```

```
  print(num2)
```

Enter number1:

23

23

Enter number2:

42

42

ghci>

# MENU DRIVEN CALCULATOR

```
--addprint.hs
```

```
str2Int::String->Int
```

```
str2Int x = read x :: Int
```

```
add::(Int,Int)->Int
```

```
add(x,y) =
```

```
(x+y)
```

```
sub::(Int,Int)->Int
```

```
sub(x,y) =
```

```
(x-y)
```

```
main = do
```

```
  putStrLn "Enter number1:"
```

```
  input1 <- getLine
```

```
  let num1 = str2Int input1
```

```
  putStrLn "Enter number2:"
```

```
  input2 <- getLine
```

```
  let num2 = str2Int input2
```

```
  putStrLn "Enter option: 1.add, 2. sub"
```

```
  input3 <- getLine
```

```
  let num3 = str2Int input3
```

```
  putStrLn "The result is:"
```

```
  if num3 == 1
```

```
    then print(add(num1,num2))
```

```
    else print(sub(num1,num2))
```

# MENU DRIVEN CALCULATOR - OUTPUT

```
ghci> :l addprint.hs
```

```
[1 of 1] Compiling Main          ( addprint.hs,  
interpreted )
```

```
Ok, one module loaded.
```

```
ghci> main
```

```
Enter number1:
```

```
34
```

```
Enter number2:
```

```
23
```

```
Enter option: 1.add, 2. sub
```

```
1
```

```
The result is:
```

```
57
```

```
ghci> main
```

```
Enter number1:
```

```
34
```

```
Enter number2:
```

```
23
```

```
Enter option: 1.add, 2. sub
```

```
2
```

```
The result is:
```

```
11
```