

# 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

```
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]
```

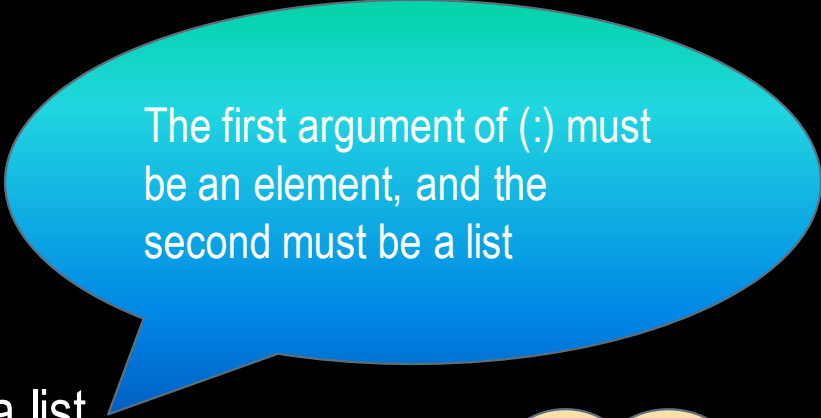
- “cons” operator (:):
- Short form for construct
- Adds an element to the front of a list

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

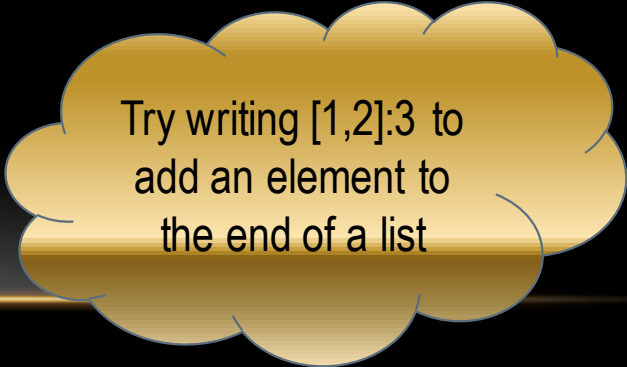
```
[1,2,3]
```

```
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 OF 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 the 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'`

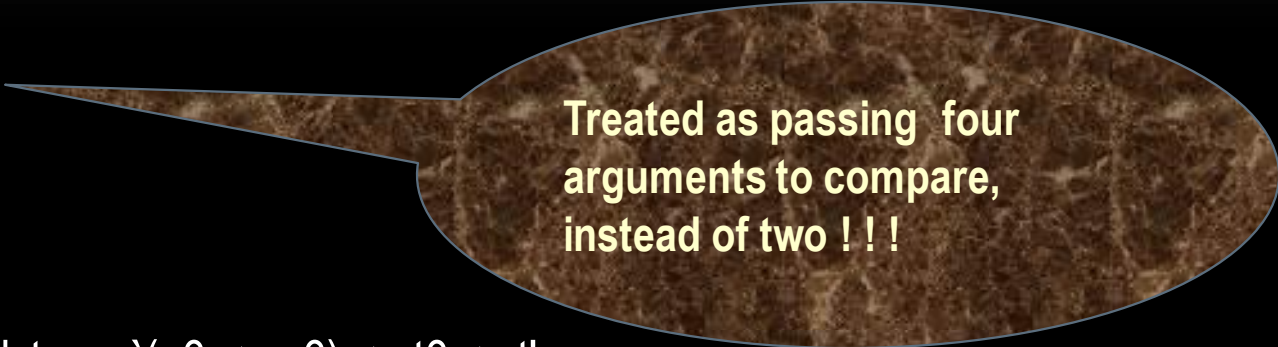
with actual type `'Ordering'`

- \* 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.
- Example:

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

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

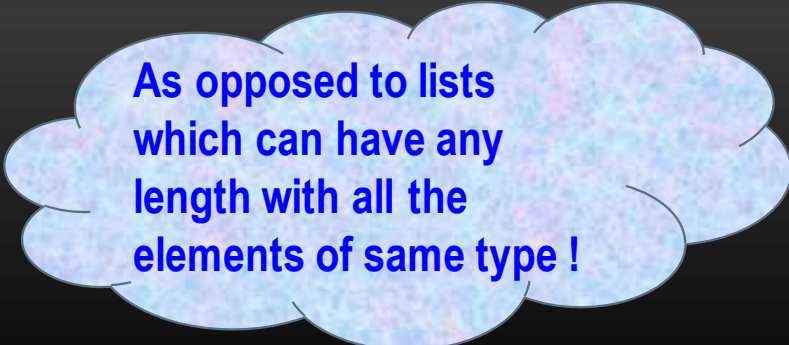
- The same above notation is used for writing a tuple's type:

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

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

```
ghci> ()
```

```
()
```



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

# 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'
```

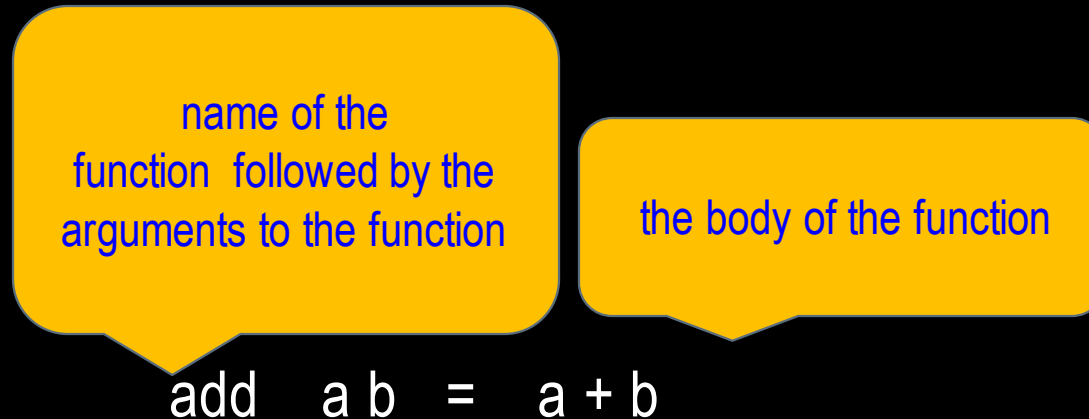
```
In the expression: head drop 4 "azerty"
```

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

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

# FUNCTION TYPES AND PURITY

- 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.

# FUNCTION TYPES:

With arguments and  
with return type

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

Click to add text

With arguments and  
without return type

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

Without arguments  
and with return  
type

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

Without  
arguments and  
without return ty  
pe

```
Click to aadd = do
  let a= 5
  let b=5
  print(a+b)
main = do
  add
dd text
```