What is Haskell?

Haskell is a functional programming language.

We study Haskell because, compared to other functional languages:

1. Haskell is statically typed (the signature of all functions and the types of all variables are known prior to execution);
2. Haskell uses lazy rather than eager evaluation (expressions are only evaluated when needed);
3. Haskell uses type inference to assign types to expressions, freeing the programmer from having to give explicit types;
4. Haskell is pure (it has no side-effects).

Haskell implementations are also interactive which means that the user interface is like a calculator; you enter expressions, the Haskell interpreter checks them, evaluates them, and prints the result. This is called the “read-eval-print” loop:

```
> hugs
Prelude> (2*5)+3
13
```

> hugs
Prelude> :load /usr/lib64/hugs/demos/Eliza.hs
Eliza> eliza
Hi! I’m Eliza. I am your personal therapy computer. Please tell me your problem.

> hello
How do you...please state your problem.

> i’m bored!
Did you come to me because you are bored?
What is Haskell?

```haskell
eliza = interact (writeStr hi $ session initial [])
  where hi = "\n
  \Hi! I’m Eliza. I am your personal ....\n  \Please tell me your problem.\n  \n"

session rs prev
  = readLine "> " (\l ->
    let ws = words (trim l)
    (response,rs') = if prev==ws then
      repeated rs else answer rs ws
      repeated rs = repeat rs
    in writeStr (response ++ "\n\n") $
      session rs' ws)
```

commaint – A Haskell Program

- Real functional programs are, naturally, a bit more complex. They make heavy use of
  - higher-order functions, functions which take functions as arguments.
  - function composition, which is a way to combine simple functions into more powerful ones.
  - function libraries, collections of functions that have proven useful. The standard.prelude that you've seen that the Haskell interpreter loads on start-up, is one such collection.

So what does a “real” functional Haskell program look like? Let’s have a quick look at one simple (?) function, commaint.

- commaint works on strings, which are simply lists of characters.
- You are not supposed to understand this! Yet...

From the commaint documentation:

```
[commaint] takes a single string argument containing a sequence of digits, and outputs the same sequence with commas inserted after every group of three digits, · · ·
```

--- Sample interaction: -------

```
? commaint "1234567"
1,234,567
```

--- commaint in Haskell: -------

```
commaint = reverse . foldr1 (\x y->x++","++y) .
group 3 . reverse
  where group n = takeWhile (not.null) .
    map (take n).iterate (drop n)
```
Haskell Program:

```haskell
commaint = reverse . foldr1 (\x y->x++","++y) .
group 3 . reverse
  where group n = takeWhile (not.null) .
    map (take n).iterate (drop n)
```

**commaint** in English:

“First reverse the input string. Take the resulting string and separate into chunks of length 3. Then append the chunks together, inserting a comma between chunks. Reverse the resulting string.”

- **group** is a “local function.” It takes a string and an integer as arguments. It divides the string up in chunks of length n.
- **reverse** reverses the order of the characters in a string.
- **drop** returns the string that remains when the first n characters of xs are removed.
- **iterate (drop 3)** returns the infinite (!) list of strings
  ```haskell
  [s, drop 3 s, drop 3 (drop 3 s),
   drop 3 (drop 3 (drop 3 s)), ...]
  ```
commaint = reverse . foldr1 (\x y->x++","++y) .
  group 3 . reverse
  where group n = takeWhile (not.null) .
    map (take n).iterate (drop n)

- map (take n) s takes a list of strings as input. It returns
  another list of strings, where each string has been shortened
  to n characters. (take n) is a function argument to map.
- takeWhile (not.null) removes all empty strings from a list
  of strings.

Since Haskell is an interactive language, we can always try out
(parts of) functions that we don’t understand.

? reverse "1234567"
7654321

? take 3 "dasdasdasd"
das

? map (take 3) ["1234","23423","45324","" ]
["123", "234", "453", []]

? iterate (drop 3) "7654321"
["7654321", "4321", "1", [], [], · · · {interrupt!}

There are several implementations of Haskell. They are mostly
the same, but differ in which libraries they support.
- In these slides the examples use the hugs Haskell interpreter.
- A better choice these days is the Haskell platform, which you
can download from here: http://hackage.haskell.org/platform.
- The Haskell platform comes with the ghci Haskell interpreter.
To get some of the examples in these slides to work you may need to import some libraries that ghci needs but that hugs loads automatically.

Here’s a list of ghci libraries:


In particular, you may need these libraries:
- Data.Char (for character operations such as toUpper)
- Data.List (for list operations such as sort)

To load these libraries in your programs say

```
import Data.Char
import Data.List
```

To load these libraries interactively when running ghci, type

```
:m Data.Char Data.List
```