CSc 372

Comparative Programming Languages

3 : Haskell — Introduction

Christian Collberg

collberg+372@gmail.com

Department of Computer Science

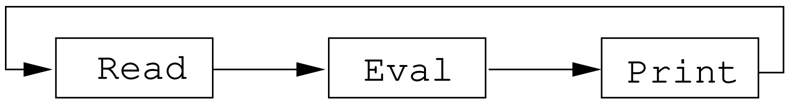
University of Arizona

What is Haskell?

- Haskell is a functional programming language.
- We study Haskell because, compared to Scheme
  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).

What is Haskell?...

- 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
```

What is Haskell?…

```
> hugs
Prelude> :load /usr/lib/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?...

```
eliza = interact (writeStr hi $ session initial [])
 where hi = "\n\
             \Hi! I’m Eliza. I am your personal therapy computer.\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
         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
  1. higher-order functions, functions which take functions as arguments.
  2. function composition, which is a way to combine simple functions into more powerful ones.
  3. 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.
- We will now look at one complex function called `commaint`.

`commaint` – A Haskell Program…

- 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, $\cdots$

`commaint` – A Haskell Program…

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)
```

`commaint` – A Haskell Program. . .

```
"1234567"
    |        reverse
    v
"7654321"
    |        iterate (drop 3)                       g
    v                                               r
["7654321","4321","1","","", ...]                   o
    |        map (take 3)                           u
    v                                               p
["765","432","1","","",...]
    |        takeWhile (not.null)                   3
    v
["765", "432", "1"]
    |        foldr1 (\x y->x++","++y)
    v
"765,432,1"
    |        reverse
    v
"1,234,567"
```

`commaint` – A Haskell Program...

`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)
```

`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."

`commaint` – A Haskell Program...

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

- `group n` 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 n xs` returns the string that remains when the first `n` characters of `xs` are removed.

`commaint` – A Haskell Program...

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

- `iterate (drop 3) s` returns the infinite (!) list of strings

  ```
  [s, drop 3 s, drop 3 (drop 3 s),
     drop 3 (drop 3 (drop 3 s)), ···]
  ```

- `take n s` returns the first `n` characters of `s`.

`commaint` – A Haskell Program. . .

```
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.

commaint – A Haskell Program...

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

- foldr1 (\x y->x++","++y) s takes a list of strings s as input. It appends the strings together, inserting a comma inbetween each pair of strings.

`commaint` – A Haskell Program...

- 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!}
```