Input / Output

Functions

There are several important functions when dealing with interactive Haskell programs:

interact

`interact` takes a function of type `String -> String`, and the entire input from stdin is passed to this function as its input, and the resulting string is output on stdout. Because of the nature of Haskell, the output will seem interactive as it gets evaluated bit by bit; but the function provided is written as if the whole input is specified at once. Remember to think of functional programming as not having a concept of time.

unlines

`unlines` takes a list, and returns it interleaved with newlines, such as:

```
Input: unlines ["aa","bb","cc","dd","ee"]
Output: "aa\nbb\ncc\ndd\nee\n"
```

lines

`lines` is the reverse of unlines, essentially:

```
Input: lines "aa\nbb\nbb"
Output: ["aa","bb","bb"]
```

unwords

`unwords` takes a list, and returns it interleaved with spaces, such as:

```
Input: unwords ["aa","bb","cc","dd","ee"]
Output: "aa bb cc dd ee"
```

words

`words` is the reverse of unwords, essentially:

```
Input: words "aa bb bb"
```

```
Output: ["aa","bb","bb"]
```

show

`show` takes an element of a type instance of the `Show` typeclass and formats it as a string:

```
Input: show 3.14
Output: "3.14"
```

read

`read` is the reverse of `show`, essentially:

```
Input: read "3.14" ::Float
Output: 3.14
```

Writing interactive programs

Generally, interactive programs seem to be of the form:

```
some_fun :: Show argument => [argument] -> String -> String
some_fun xs = unlines . map doSomething . lines
```

Though there are more complicated variations (the 2009 paper uses a mapState function which is a state machine).

Why does this work?

Haskell uses 'Lazy Evaluation' which is a strategy for program reduction. It works by not evaluating arguments to functions until they are needed, and even then only evaluating as much as is needed by the function. This is why infinite data structures can be used in Haskell. The following would be impossible with strict evaluation...

```
numbers = numsFrom 0
        where
        numsFrom n = n : numsFrom (n+1)
head numbers
```

...because the programming language would try to evaluate the entire `numbers` list (which is infinite) just to return the first element. In Haskell, due to lazy evaluation, this will work.

An interactive program can therefore be formulated as a lazily evaluated function from a list of the user's inputs to a list of program's outputs. The computation of the program's output list always proceeds as far as possible when evaluating the user's input list, suspending reduction only when the next item in the list is strictly needed.

Arbori (binari) de căutare

Modulul Arbore

Scrieţi un modul Haskell numit `Arbore` care defineşte o colecţie conţinând un tip algebric de date `Arbore a` pentru reprezentarea arborilor binari de căutare cu date de tip `a`.

Modulul trebuie să exporte tipul de date `Arbore` si următoarele funcţii:

- `adauga` - care dat fiind un element x si un arbore t adaugă elementul x în arborele t
- `cauta` - care dat fiind un element x si un arbore t spune dacă acesta se află în arbore.
- `init` - care dată fiind o listă de elemente creează un arbore binar de căutare care le conţine
- `parcurge` - care dat fiind un arbore produce lista obţinută prin parcurgerea în inordine (SRD) a arborelui.

Programul principal

Scrieţi un modul `Principal`, executabil, care foloseşte funcţiile din modulul Arbore pentru a citi de la

tastatură o listă de numere separate prin spaţiu, aflate pe aceeași line, si afișează numerele citite, în ordine crescătoare, separate prin spaţiu.

Compilaţi și executaţi programul