Haskell: Data continued

CS 331 Programming Languages
Lecture Slides
Wednesday, March 5, 2025

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Unit Overview
The Haskell Programming Language

Topics

- ✓ PL feature: type system
- ✓ PL category: functional PLs
- ✓ Introduction to Haskell
- ✓ Haskell: functions
- ✓ Haskell: lists
- ✓ Haskell: flow of control
- ✓ Haskell: I/O
- (part) Haskell: data

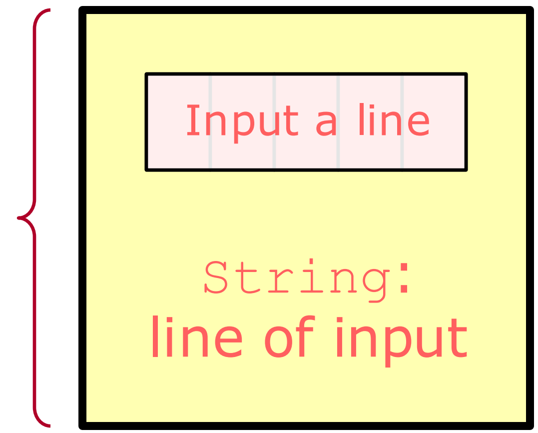
Review

Review
Haskell: I/O [1/3]

An **I/O action** is a value that holds a description of a sequence of zero or more side effects, plus a wrapped (potential) value.

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

I/O Action
returned by `getLine`



I/O is performed by returning an I/O action from a program.

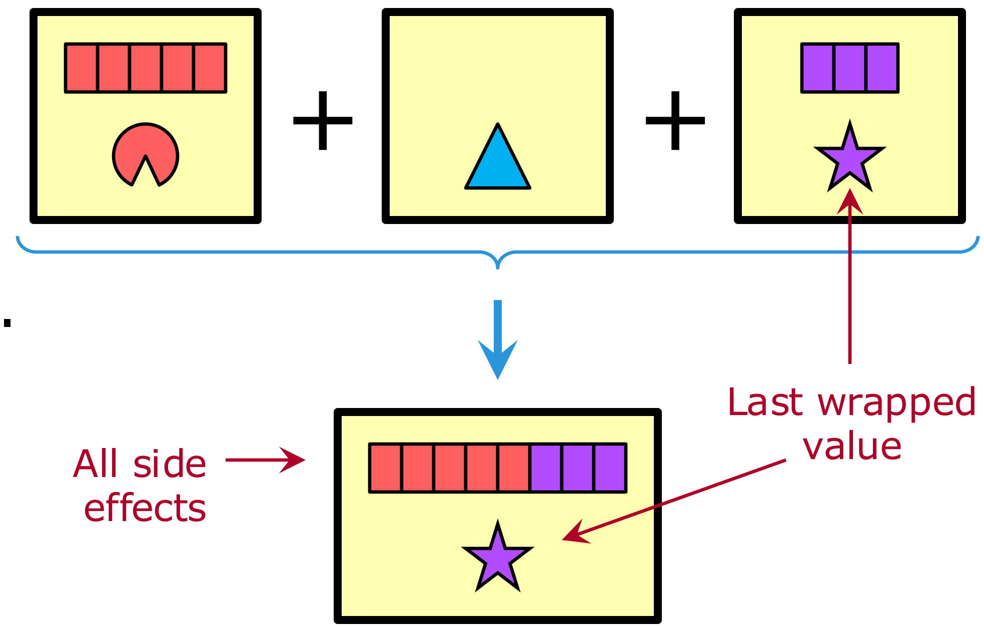
```
> main = putStrLn "Hello, world!"
> main
Hello, world!
```

For code from this topic, see `io.hs`.

Haskell: I/O [2/3]

We can combine I/O actions using the >> and >>= operators.

```
> putStr "Type: " >> getLine >>=
    (\ line -> (putStr "You typed: " >> putStrLn line))
Type: Yo!
You typed: Yo!
```



Do-expression: syntactic sugar around the operators.

```
do
    putStr "Type: "
    line <- getLine
    putStr "You typed: "
    putStrLn line
```

Review
Haskell: I/O [3/3]

Inside an I/O do-expression:

- "*NAME* `<-` *EXPR*" binds an identifier to an I/O-wrapped value—useful when doing input.
- "`let` *NAME* `=` *EXPR*" binds an identifier to a non-wrapped value.
- "`return` *EXPR*" creates a do-nothing (no side effects) I/O action wrapping a given value.

`return` does not return! However, we only use it as the last thing in a do-expression (otherwise it is pointless). So it *feels* like it returns.

DONE

- Write a program of the kind that might be assigned early in Computer Science I.

See `io.hs` *or* `squarenums.hs`*.*
Also see `bestnum.hs`*.*

Haskell: Data — data Declaration

Haskell **data declaration**:

```
data Product = Pr String String
  -- product name, manufacturer name
```

`Product` is a new type. `Pr` is a **constructor** for `Product`. Literals of type `Product` are marked by the fact that they begin with “`Pr`”.

Pattern matching works with constructors. We can use this to retrieve names from a `Product` object.

```
-- pName - Get product name from a Product
pName :: Product -> String
pName (Pr pn _) = pn
```

For code from this topic, see `data.hs`*.*

Haskell: Data — Overloading & Type Classes

Overload `==` for `Product`; make `Product` an **instance** of class `Eq`:

```
instance Eq Product  where
    Pr pn1 mn1 == Pr pn2 mn2 =
        (pn1 == pn2) && (mn1 == mn2)
```

Now we can use operator `==` *and* operator `/=` with `Product`.

Similarly overload function `show`, using typeclass `Show`:

```
instance Show Product  where
    show (Pr pn mn) = pn ++ " [made by " ++ mn ++ "]"
```

Haskell: Data

continued

Haskell: Data
Options & Parametrization [1/6]

A Haskell data declaration allows for multiple options on the right-hand side. These are separated by vertical bars ("`|`").

For example, if Haskell did not have `Bool`, then we could write it ourselves.

```
data Bool = True | False
```

The above illustrates why `True` and `False` are capitalized in Haskell: like `Pr` on the previous slides, they are *constructors*.

Haskell: Data
Options & Parametrization [2/6]

Haskell data declarations can also be **parametrized**. A parametrized type is similar to a C++ class template.

We have seen one Haskell example: `IO`. Much like `std::vector` in C++, this takes an argument indicating the type of the value it wraps. Haskell calls `IO` a **type constructor**.

```
// C++
vector         // Incomplete; not a type
vector<int>    // A type: vector of int

-- Haskell
IO             -- Incomplete; not a type
IO String      -- A type: an IO action that wraps a String
```

Haskell: Data
Options & Parametrization [3/6]

Another standard type constructor is `Maybe`. This allows us to make a value of an existing type that can also have a null value.

`Maybe` would be declared as follows.

```
data Maybe t = Just t | Nothing
```

One way to use `Maybe` is to make `Nothing` indicate an error, while `Just` indicates a valid result.

```
lookInd2 :: Integer -> [t] -> Maybe t
lookInd2 0 (x:_) = Just x
lookInd2 n (_:xs) = lookInd2 (n-1) xs
lookInd2 _ [] = Nothing
```

Haskell: Data
Options & Parametrization [4/6]

Yet another standard type constructor: `Either`. This allows us to make a value that holds one of two specified types.

`Either` would be declared as follows.

```
data Either a b = Left a | Right b
```

The following function uses pattern matching to determine which type is held.

```
checkEither :: Either a b -> String
checkEither (Left _) = "First type"
checkEither (Right _) = "Second type"
```

Haskell: Data
Options & Parametrization [5/6]

Let's make a Binary Tree type, with a data item in each node.

Such a Binary Tree either has no nodes (it is **empty**) or it has a **root** node, which contains a data item and has **left** and **right** **subtrees**, each of which is a Binary Tree.

I will call the type `BT`. It will have two constructors.

- `BTEmpty` gives an empty Binary Tree.
- `BTNode`, followed by an item of the value type, the left subtree, and the right subtree, constructs a nonempty tree.

```
data BT vt = BTEmpty | BTNode vt (BT vt) (BT vt)
```

The **value type**

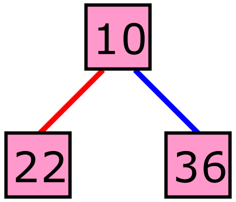
For example, here is a `BT` with one node, containing `42`:

```
BTNode 42 BTEmpty BTEmpty
```

Haskell: Data
Options & Parametrization [6/6]

Let’s construct the Binary Tree pictured. The left subtree has a root containing `22` and no other nodes. The right subtree is similar.

```
leftT = BTNode 22 BTEmpty BTEmpty
rightT = BTNode 36 BTEmpty BTEmpty
```



Construct our tree from a new root and the above two.

```
theTree = BTNode 10 leftT rightT
```

We could write out the whole tree—but there is no need to do this.

```
BTNode 10 (BTNode 22 BTEmpty BTEmpty)
          (BTNode 36 BTEmpty BTEmpty)
```

Haskell: Data
Treesort [1/6]

With the tools we have covered, we can write the *Treesort* algorithm in Haskell.

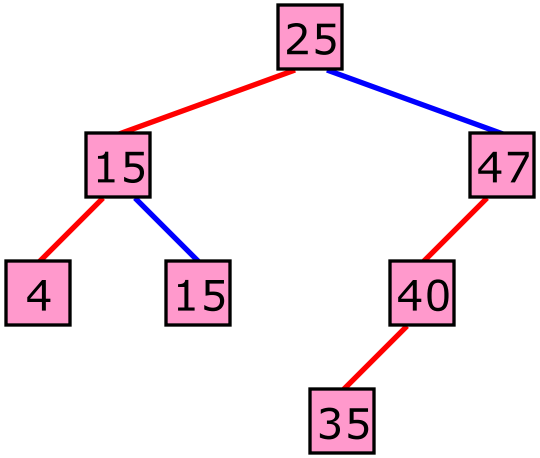
Treesort is a comparison sort. It operates as follows. Given a list:

- Create an empty Binary Search Tree.
- Insert each list item into tree.
- Do an inorder traversal of the tree to generate the final sorted list.

Haskell: Data
Treesort [2/6]

Recall. A **Binary Search Tree** is a Binary Tree in which each node contains a single key, and these have an order relationship:

- Every key in a node's right subtree is ≥ the node's key.
- Every key in a node's left subtree is ≤ the node's key.

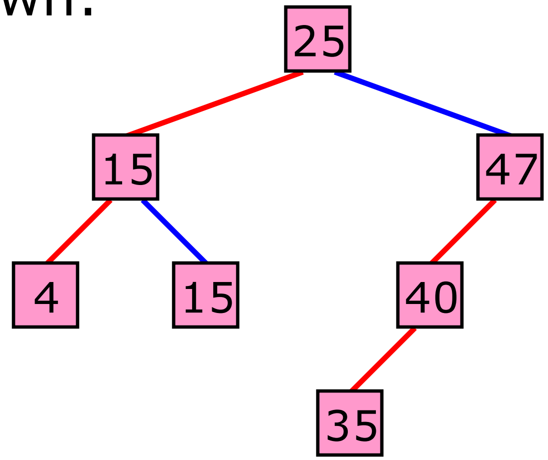


Haskell: Data
Treesort [3/6]

Recall. The **inorder traversal** of a Binary Tree visits a node's left subtree, then the node itself, then the right subtree. The subtrees are visited using recursive inorder traversals.

Inorder traversal of the Binary Tree shown:

- 4 15 15 25 35 40 47



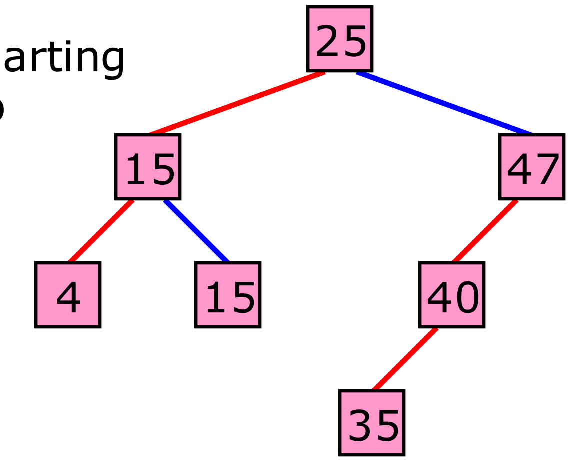
An inorder traversal of a Binary Search Tree visits the keys in sorted order.

Haskell: Data
Treesort [4/6]

Example. Treesort this list.

25	47	15	40	4	15	35

- Create an empty Binary Search Tree. Go through the list, inserting each item into the tree.
  - Insert into a Binary Search Tree by starting at the root and comparing the item to be inserted with the item in the node, passing to the left or right subtree, as appropriate, and continuing. When the bottom is reached, insert in this spot.



- Do an inorder traversal of the tree, appending each item to a list.

In a PL like C++, we usually prefer to copy the data back to the original storage. But we cannot do that in Haskell.

4	15	15	25	35	40	47

Haskell: Data
Treesort [5/6]

Treesort is not an efficient sorting algorithm. And a naïve Haskell implementation is likely to be even slower than it could be.

However, I think Treesort makes for a good example.

- It is not difficult to understand.
- Implementation requires nontrivial data structures.
- It is easy to tell whether an implementation works.

Let's write Treesort in Haskell, using `BT` for the Binary Search Tree.

Haskell: Data
Treesort [6/6]

Plan

- Function `bstInsert`. Takes a `BT` holding a Binary Search Tree and an item to insert. Returns the resulting Binary Search Tree.
  Operation: recursively navigates down through subtrees.
- Function `inorderTraverse`. Takes a `BT`. Returns a list of the items in the tree in the order given by an inorder traversal.
  Operation: makes recursive calls on the two subtrees and concatenates the results along with the root item.
- Function `treesort`. Takes a list and returns a sorted list.
  Operation: creates an empty `BT`, calls `bstInsert` with each item in the given list, then calls `inorderTraverse` and returns its result.

TO DO

- Implement Treesort as above.

Done. See `data.hs`*.*