Writing a Compiler in Haskell

Today

- Some Haskell History
- Haskell main for keeping side-effects contained
- Writing functions in Haskell
- Debugging Haskell
- User-defined datatypes
- Lexicographical analysis for punctuation and keywords in Haskell

This week

- PA1: Start PA3whiledots.java.s. It is due in 11 days along with demo!
- HW2: Due Tuesday September 6th
  - Reading assignment posted on class schedule for today
  - Friday will be using Haskell in discussion section

Some Haskell History

1990 Haskell 1.0

- Designed by committee starting with 1987 meeting at a conference.
- See "A History of Haskell Being Lazy with Class", by Hudak, Hughes, Jones and Wadler.

1999 The Haskell 98 Report

2013 Sabbatical in Australia

- Philip Wadler
- Gabriele Keller
- Manuel Chakravarty

2016 Haskell has momentum

- https://wiki.haskell.org/Haskell_in_industry

September 2016, CS 453

- Focus is on a small subset of Haskell that enables writing a compiler.

Some Haskell Features

Purely functional

- Any function with same input returns same output.
- Wait isn't that true in C, Java, ... everything?
- No side effects and no state, just values.

Strongly typed

- It will infer all of the types based on how values are used.
- You can also declare some of the types to make code more readable.

Lazy

- Expressions are not evaluated unless they are needed.
- More on this later in the semester.

Functions are first class objects

- Functions are values too!
- An expression can evaluate to a function.

Haskell keeps side-effects contained

Pure means no side effects

- I/O is a side effect
- Storing state is a side effect
- How on earth is this a useful language?

Main module and main function

- We will be recommending using it in the main function only
- main function essentially builds the AST for a program that does I/O
- The AST returned by main is always the same, referential transparency
- Then the Haskell system interprets that AST.

The do block syntax

- Results in code that looks imperative.
- Is syntactic sugar for stuff we will cover in more depth in November.

Interacting with a user

```
module Main where

main :: IO ()
main = do
    putStrLn "Pick a number: "
    n1 <- getLine
    putStrLn ("Number is " ++ (show n1))
    putStrLn ("Another number: ")
    n2 <- getLine
    let n3 = (read n1) + (read n2)
    putStrLn ("Sum of numbers = " ++ (show n3))
```

Writing Functions with Pattern Matching

```
f :: a -> b
f x = case x of
   ... -> blah
   ... -> foo
   ...
   ... -> dah
```

```
-- Equivalent
f :: a -> b
f ... = blah
f ... = foo
   ...
f ... = dah
```

```
-- Examples

-- Duplicating a string list
f :: [String]->[String]
f (x:xs) = x:(f xs)
f []     = []

-- Sum values in a list.

-- Concat list of strings.

-- Second Int in a 3-tuple.
```

Writing a Compiler in Haskell

Today

- Some Haskell History
- Haskell main for keeping side-effects contained
- Writing functions in Haskell
- Debugging Haskell
- User-defined datatypes
- Lexicographical analysis for punctuation and keywords in Haskell

This week

- PA1: Start PA3whiledots.java.s. It is due in 11 days along with demo!
- HW2: Due Tuesday September 6th
  - Reading assignment posted on class schedule for today
  - Friday will be using Haskell in discussion section

Debugging Haskell

Try evaluating the expressions one step at a time

Examples

- <Step through this with some example functions we just wrote as class>

User-defined Datatypes in Haskell

Kindof like enumerate types but can have fields

```
data Bool = False | True
data Shape = Point | Rect Int Int Int Int | Circle Int
```

Can derive handy properties

```
data Color = Blue | Red | Yellow deriving (Show)
main = print Yellow
data Color = Blue | Red | Yellow deriving (Show,Eq)
if (Yellow==Blue) then ... else ...
```

Constructors can be used in pattern matching

```
foo :: Shape -> String
foo Point = "Point"
foo Rect p1 p2 p3 p4 = "Rect " ++ (show p1) ++ ...
```

Some Lexical Analysis with Haskell

```
module Lexer where

import Data.Char  -- needed for isSpace function

data Token
  = TokenIfKW
  | TokenComma
  -- TODO: constructors for all other tokens
  deriving (Show,Eq)

lexer :: String -> [Token]
lexer [] = []
lexer ('i':'f':rest) = TokenIfKW : lexer rest
-- TODO: patterns for other keyword and punctuation tokens
lexer (c:rest) = if (isSpace c) then lexer rest
```

Before Next Time

PA1: Start PA3whiledots.java.s. It is due in 11 days along with demo!

HW2: Due Tuesday September 6th

- Reading assignment posted on class schedule for today
- Friday will be Haskell in discussion section