Lecture 5: I/O!

CSC 131
Spring, 2019

Kim Bruce

Homework

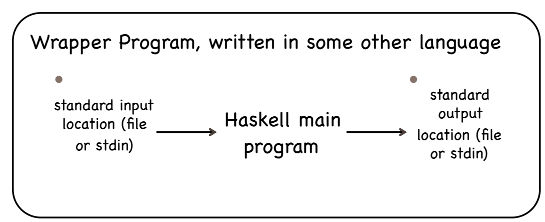
- First line:
  - module Hmwk2 where
  - Next line should be name as comment
  - Name of program file should be Hmwk2.hs

History of I/O

- Big embarrassment to lazy functional programming community
  - ML, Scheme/LISP/Racket didn't care about being purely functional
- Alternatives:
  - Streams ⟵ *Haskell 1.0 adopted, essentially lazy lists*
  - Continuations
    - pure functions passed to IO routines to process input
  - Pass state of world as parameter
    - Hard to make single-threaded

Stream Model

- Move side effects outside of functional program
  - main:: String -> String



  - But what if more than one file or socket or ...?

Stream Model

- Enrich argument and return type of main to include all input and output events.

```
main :: [Response] -> [Request]
data Request   =  ReadFile Filename
          |  WriteFile FileName String
          | …
data Response  =  RequestFailed
          |  ReadOK String
          |  WriteOk
          |  Success  | …
```

- Wrapper program interprets requests and adds responses to input.

Stream Model is Awkward!

- Hard to extend
  - New I/O operations require adding new constructors to Request and Response types, modifying wrapper
- Does not associate Request with Response
  - easy to get "out-of-step," which can lead to deadlock
- Not composable
  - no easy way to combine two "main" programs
- … and other problems!!!

Defining Monads

part of Standard Prelude

- class Monad m where
  (>>=) :: m a → (a → m b) → m b
  return :: a → m a
  fail:: string → m a
  - >>= allows a kind of composition of wrapped values or computations -- called *bind*
  - return wraps an unwrapped value.
  - fail takes error string & aborts program

Maybe Monad

- instance Monad Maybe where *part of Standard Prelude*
  (>>=) Nothing f = Nothing
  (>>=) (Just x) f = f x
  return x = Just x
  fail s = Nothing
- >>= preserves "Nothing",
- >>= unwraps argument to compute w/ a Just'ed value
- Second arg of >>= is function applied to unwrapped value
- *Abbreviate* compu >>= \x → exp *as*
  do x <- compu
  exp

Back to Example

- Expression
  - getPFN name rooms phones =
    do rm <- getDormFor name rooms
    num <- getPhoneForRoom rm phones
    return num
  - *abbreviates*
  - getPFN name rooms phones =
    getDormFor name rooms >>=
    (\rm -> getPhoneForRoom rm phones)

Monads to Rescue!

- Value of type (IO a) is an action
  - that may perform some input/output
  - and deliver result of type a

I/O

- main :: IO( ) -- "IO action"
- main = putStrLn "Hello World!"
- where putStrLn:: String → IO( )
- getLine :: IO String -- "IO action" returning string

- Want echo = putStrLn getLine
  - Types don't match
  - Need >> = for IO monad!!
  - echo = do str <- getLine
    putStrLn str

See monad.hs

Connecting Actions

getLine IO String

String *putStrLn* IO ( )

Glued together with >>=

More IO

```
ask :: String -> String -> IO()
ask prompt ansPrefix =
        do putStr (prompt++" ")
           response <- getLine
           putStrLn (ansPrefix ++ " " ++ response)

getInteger :: IO Integer
getInteger = do putStr "Enter an integer: "
                line <- getLine
                return (read line)
        -- converts string to Integer then to IO Integer
```

IO & Ref Transparency

- Main program is IO action w/type IO( )
- Perform IO in IO actions & call pure functions from inside there
- Can never escape from IO! *Unlike Maybe.*
  - *No constructors for IO, so can't pattern match to escape!!!*
- IO impure in that successive calls of getLine return different values.

Using IO in Haskell

- Can build language at IO monad level:

```
ifIO :: IO Bool -> IO a -> IO a -> IO a
ifIO b tv fv = do { bv <- b;
                    if bv then tv else fv}

whileIO :: IO Bool -> IO( ) -> IO( )
whileIO b m = ifIO b
                (do {m; whileIO b m})
                (return( ))
```

See Chapter 7 of Learn You a Haskell

More Info on Monads

- See "documentation" page of class web page
- For comprehensive list of tutorials, see
  - Monad Tutorials Timeline

Evaluating Functional Languages

Program Correctness

- Verification easier if language satisfies declarative language test -- evaluate once and reuse as necessary.
- let val I = E in E' end is equivalent to [E/I]E'
- Not true if imperative features.
- Only true if lazy evaluation.
- Let E be a functional expression (no side effects). If E converges to a value v with eager evaluation then it converges to the same value with lazy eval.

Why not be lazy?

- Eager languages easier to implement w/ conventional techniques
- If side-effects then when will they occur?
- If computing in parallel, want to start as soon as feasible.
  - Even if result may be wasted!
- Can simulate in eager languages by adding extra dummy parameter to delay evaluation.

Haskell Later ...

- Type inference algorithm
- Support for ADT's and modules
- Support for exception handling
- Garbage collection

Implementation Issues

- Slower than imperative
  - Lists instead of arrays
  - Passing around functions is expensive
    - See why later!
  - Recursion can use more space than iteration
  - Lack of destructive updating (but sharing helps!)
  - Listful style
  - Lazy has its own extra overhead

Summary of Haskell

- Successful language for designing large systems
- Lots of experimentation with language design
  - Type classes, software transactional memory, parser combinators
- See "Tackling the Awkward Squad"

Summary of Functional Langs

- Use requires alternative approaches
- Declarative languages support reasoning
- Higher-order functions powerful
- Some loss of efficiency balanced by programmer efficiency
- Implicit Polymorphism
- Strongly influence modern imperative languages

Building an Interpreter or compiler

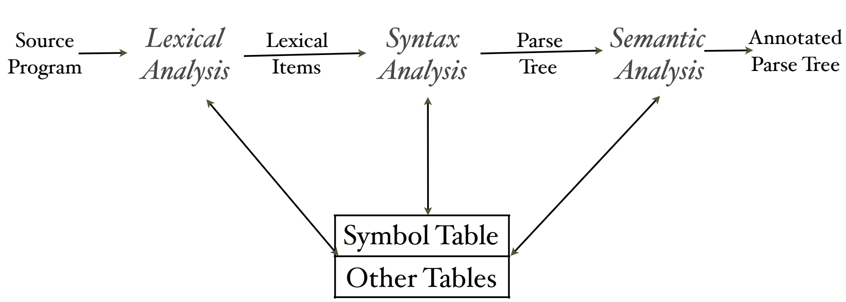
Compiler Structure

- Analysis:
  - Break into lexical items, build parse tree, annotate parse tree (e.g. via type checking)
- Synthesis:
  - generate simple intermediate code, optimization (look at instructions in context), code generation, linking and loading.

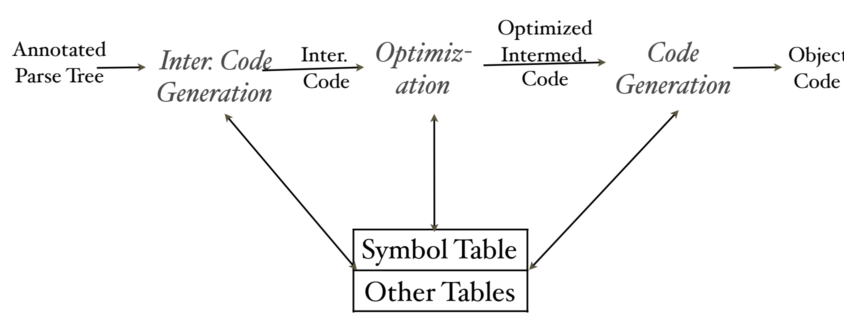
Symbol Table

- Symbol table:
  - Contains all id names,
  - kind of id (vble, array name, proc name, formal parameter),
  - type of value,
  - where visible, etc.

Analysis



Synthesis



Portable Compilers

- Separate front and back-ends that share same intermediate code (GNU compilers)
  - Write single front end for each language
  - Write single back end for each operating system - architecture combination.
  - Mix and match to build complete compilers
- JVM starting to play that role too

Step 1: Lexical Analysis

Lexing

- Lexer returns a list of all tokens from the input stream.
- Build from either regular expressions or (equivalently) finite automaton recognizing the tokens.
- See program LexArith.hs in class examples.
    - Haskell program uses modules to hide info

Explaining LexArith

- module LexArith(...) where
  - lists funcs and types exported (includes constructors)
- code details follow in file
  - getid :: [Char] -> [Char] -> ([Char], [Char])
    - takes string and prefix of id to first full id and rest of string to be processed
  - getnum :: [Char] -> Int -> (Int, [Char])
    - similar
  - getToken: [Char] → (Token, [Char])
    - takes string to pair of first recognized token and rest of list to be processed