

The slide features several light purple circles of varying sizes and opacities. Some are solid, while others are hollow outlines. They are arranged in a pattern that frames the text.

CS 4450: Principles of Programming Languages

10/26/2009

Language Processing:
Interpreting Errors



Today

- We'll consider how to write an interpreter for a small arithmetic language
 - This is a slight excursion from the Ramsey & Kamin text
- Like ImpCore, it also has **side effects**

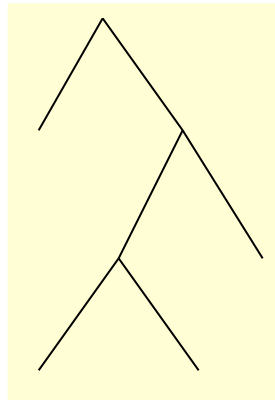


Today: Language Processing

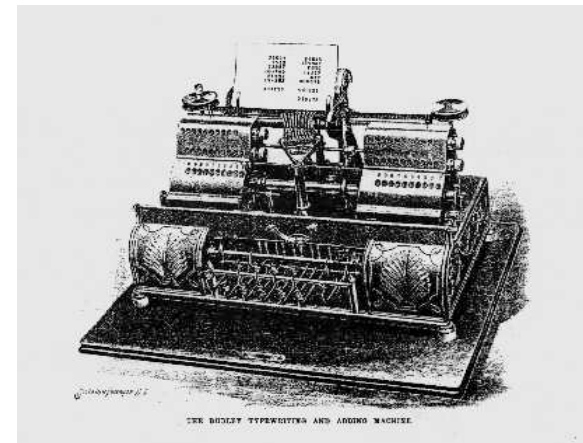
- Representing languages: Abstract Syntax
 - ...and how to represent AS in Haskell
 - Defining new types with `data` declarations
 - Abstract Syntax Trees
 - Introduction to Backus-Naur Form (BNF)
 - Wikipedia entry may be helpful
 - http://en.wikipedia.org/wiki/Backus-Naur_form
- Defining Languages with Interpreters
 - Simple example: arithmetic language
- Key consideration: languages with “effects”
 - Ex: how do you interpret “1 / 0”?

An interpreter is a function

Representation
of a program



Each expression/statement
defined in terms of Haskell



`interp :: AbstractSyntax` \longrightarrow "Computation of Values"

The Language We'll Interpret Today

```
data Op      = Plus | Minus | Times | Div
data Exp     = Const Int | Aexp Op Exp Exp
```

N.b., simpler than Scheme;
all its operators
are binary, and only Int values

```
ex1 = Aexp Plus (Const 1) (Const 2)  -- (+ 1 2)
ex2 = Aexp Div (Const 1) (Const 0)  -- (/ 1 0)
```

Interpreter, v1.0

```
interp :: Exp -> Int
interp (Const v)           = v
interp (Aexp Plus e1 e2)   = interp e1 + interp e2
interp (Aexp Minus e1 e2) = interp e1 - interp e2
interp (Aexp Times e1 e2) = interp e1 * interp e2
interp (Aexp Div e1 e2)    = interp e1 `div` interp e2
```

```
Arith> interp ex1
```

```
3
```

```
Arith> interp ex2
```

```
Program error: divide by zero
```



An error is a “side effect”

The value of an `Exp` is an `Int`, but the error “happens on the side” - i.e., it’s not a proper value

```
Arith> interp ex2
```

```
Program error: divide by zero
```

Question: how do we handle this error side effect?

Answer: Add a value for errors

Haskell has a tool for just this sort of thing:

```
data Maybe a = Just a | Nothing
```



Handling an error

Haskell has a tool for just this sort of thing:

```
data Maybe a = Just a | Nothing
```

- If (interp e) doesn't have any errors, then return (Just v) where v is the value of e
- Otherwise, signify that something is wrong within (interp e) by returning Nothing

This is what we want

```
Arith> interp ex1
Just 3
Arith> interp ex2
Nothing
```


Interpreter, v2.0

```
interp2 :: Exp -> Maybe Int
```

```
interp2 (Const v) = Just v
```

```
interp2 (Aexp Plus e1 e2)
= case (interp2 e1,interp2 e2) of
  (Just v1,Just v2) -> Just (v1 + v2)
  (_,_)             -> Nothing
```

```
interp2 (Aexp Div e1 e2)
= case (interp2 e1,interp2 e2) of
  (Just v1,Just 0) -> Nothing
  (Just v1,Just v2) -> Just (v1 `div` v2)
  (_,_)             -> Nothing
```

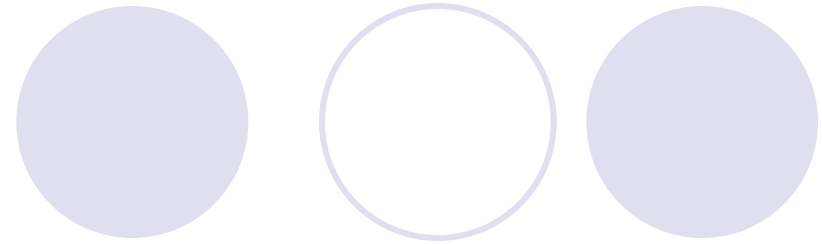


*notice how
the fact that
errors may happen
is now reflected
in the type signature*

This works, but...

- The definition is a little bit “hairy”
 - i.e., there’s a lot of exposed “plumbing”
- We want to have our cake and eat it, too
 - accurate error handling
 - plus easy-to-read code
- Haskell has tools for this
 - called “do notation”
 - we’ll see another version later

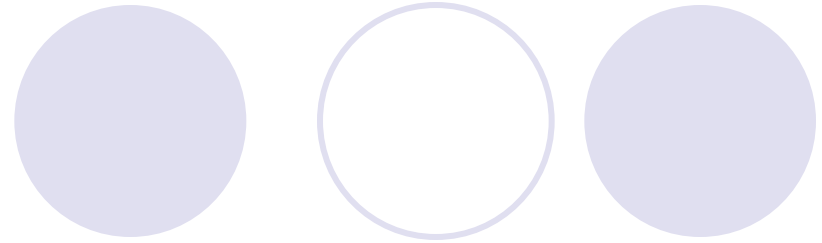
do Notation



```
interp3 (Aexp Plus e1 e2) = do v1 <- interp3 e1
                               v2 <- interp3 e2
                               Just (v1+v2)
```

- evaluate (interp3 e1) first; if it is:
 - (Just v1), then strip off the Just,
 - Nothing, then the whole do expression is Nothing
- evaluate (interp3 e2) next; if it is:
 - (Just v2), then strip off the Just,
 - Nothing, then the whole do expression is Nothing
- If you've got both v1 and v2, Just (v1+v2)

Interpreter, v3.0



```
interp3 :: Exp -> Maybe Int
interp3 (Const v)           = Just v
interp3 (Aexp Plus e1 e2) = do v1 <- interp3 e1
                               v2 <- interp3 e2
                               Just (v1+v2)
interp3 (Aexp Div e1 e2)  = do v1 <- interp3 e1
                               v2 <- interp3 e2
                               if v2==0
                               then
                                   Nothing
                               else
                                   Just (v1+v2)
```

Alternative formulation of “do”

```
do v <- x
    e
```

can also be written

```
x >>= \ v -> e
```

“bind”

Ex:

```
do v1 <- interp3 e1
   v2 <- interp3 e2
   Just (v1+v2)
```

becomes

```
interp3 e1 >>= \ v1 ->
interp3 e2 >>= \ v2 ->
  Just (v1+v2)
```

Another equivalence: can write “return” for “Just” as:

```
Just (v1+v2) == return (v1+v2)
```


Throwing an error

```
interp4 (Aexp Div e1 e2) = interp4 e1 >>= \ v1 ->
                             interp4 e2 >>= \ v2 ->
                               if v2==0
                                 then Nothing
                                 else Just (v1 `div` v2)
```

Pattern

```
if condition
  then throw_error
  else good_value
```

generalizes to:

```
throw :: ?
throw condition goodval = if condition
                           then Nothing
                           else Just goodval
```

Throwing an error

```
interp4 (Aexp Div e1 e2) = interp4 e1 >>= \ v1 ->  
                           interp4 e2 >>= \ v2 ->  
                             throw (v2==0) (v1 `div` v2)
```

Pattern

```
if condition  
  then throw_error  
  else good_value
```

generalizes to:

```
throw :: ?  
throw condition goodval = if condition  
                           then Nothing  
                           else Just goodval
```

Throwing an error

```
interp4 (Aexp Div e1 e2) = interp4 e1 >>= \ v1 ->  
                             interp4 e2 >>= \ v2 ->  
                             throw (v2==0) (v1 `div` v2)
```

Pattern

```
if condition  
  then throw_error  
  else good_value
```

generalizes to:

```
throw :: Bool -> a -> Maybe a  
throw condition goodval = if condition  
                             then Nothing  
                             else Just goodval
```

Interpreter, v5.0

```
interp5 :: Exp -> Maybe Int
interp5 (Const v)          = return v
interp5 (Aexp Plus e1 e2) = interp5 e1 >>= \ v1 ->
                             interp5 e2 >>= \ v2 ->
                             return (v1+v2)
interp5 (Aexp Minus e1 e2) = interp5 e1 >>= \ v1 ->
                             interp5 e2 >>= \ v2 ->
                             return (v1-v2)
interp5 (Aexp Times e1 e2) = interp5 e1 >>= \ v1 ->
                             interp5 e2 >>= \ v2 ->
                             return (v1*v2)
interp5 (Aexp Div e1 e2)   = interp5 e1 >>= \ v1 ->
                             interp5 e2 >>= \ v2 ->
                             throw (v2==0) (v1 `div` v2)
```

Observe: Just and Nothing don't occur in the above text

Interpreter, v6.0

```
interp6 :: ??????  
interp6 (Const v)           = return v  
interp6 (Aexp Plus e1 e2)   = interp6 e1 >>= \ v1 ->  
                             interp6 e2 >>= \ v2 ->  
                             return (v1+v2)  
interp6 (Aexp Minus e1 e2) = interp6 e1 >>= \ v1 ->  
                             interp6 e2 >>= \ v2 ->  
                             return (v1-v2)  
interp6 (Aexp Times e1 e2) = interp6 e1 >>= \ v1 ->  
                             interp6 e2 >>= \ v2 ->  
                             return (v1*v2)  
-- interp6 (Aexp Div e1 e2) = interp6 e1 >>= \ v1 ->  
--                             interp6 e2 >>= \ v2 ->  
--                             throw (v2==0) (v1 `div` v2)
```

Question: what's the type of interp6?

interp6 :: (Monad m) => Exp -> m Int

Monad is a class with >>=, return

Monad Class in Haskell

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b

instance Monad Maybe where
  return          = Just
  (Just v) >>= f = f v
  Nothing >>= f  = Nothing
  x >> y         = x >>= \ _ -> y
```

Monad Laws

For any monad:

```
-- Left Unit
(return v) >>= f = f v
-- Right Unit
x >>= return = x
-- Associative
x >>= (\ v -> y >>= \ w -> z)
      = (x >>= \ v -> y) >>= \ w -> z
```

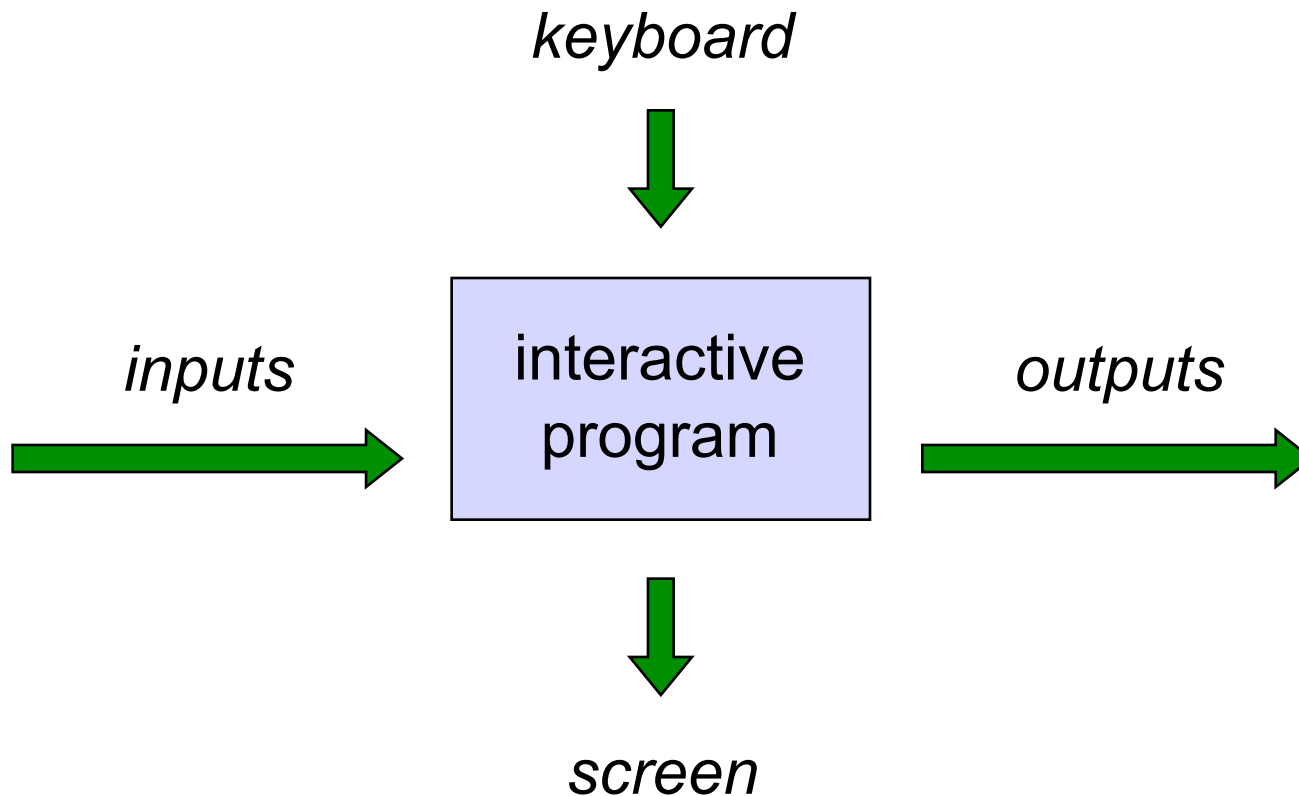
N.b., every instance of the Monad class is not a monad

Input/Output in Haskell

To date, we have seen how Haskell can be used to write batch programs that take all their inputs at the start and give all their outputs at the end.



However, we would also like to use Haskell to write interactive programs that read from the keyboard and write to the screen, as they are running.



Read-eval-print loop

- Many interpreters are interactive
 - type in a term, and it tries to evaluate it
 - e.g., the Hugs interpreter we use
- Interactive interpreters do the following
 - **read**: scan/parse/type-check what you type
 - **eval**: interpret the result AST
 - **print**: give you the answer (including errors)
- ...our interpreter will have a REP loop:

```
Hugs> repl
TigerScheme> (+ x 1)
      val it = 2
TigerScheme>
```

Read Eval Print loop

```
repl = top initEnv
```

```
top :: Env -> IO ()
```

```
top env = do
```

```
    putStr "TigerScheme> "
```

```
    iLine <- getLine
```

```
    process env iLine
```

provided to you

```
process :: Env -> String -> IO ()
```

```
process env "quit" = return ()
```

```
process env iLine =
```

```
    case (parse iLine) of
```

```
        Just term ->
```

```
            putStr (" val = " ++ show v ++ "\n") >> top env
```

```
                where v = evaluate term
```

```
        Nothing -> putStr "Syntax Error!\n" >> top env
```

what you write



The TigerScheme Interpreter

- This is the function that we will spend the rest of the semester writing
- It is our vehicle for defining Scheme
 - i.e., it's our language design for Scheme
- Defining the Interpreter
 - ...means defining the **values** produced by Scheme programs
 - ...and defining an **evaluation function** from AST to these values

The Problem



Haskell programs are pure mathematical functions:

- Haskell programs have no side effects.

However, reading from the keyboard and writing to the screen are side effects:

- Interactive programs have side effects.

The Solution

Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure actions that may involve side effects.

`IO a`

The type of actions that return a value of type `a`.

For example:

IO Char

The type of actions that return a character.

IO ()

The type of purely side effecting actions that return no result value.

Note:

- () is the type of tuples with no components.

Basic Actions



The standard library provides a number of actions, including the following three primitives:

- The action getChar reads a character from the keyboard, echoes it to the screen, and returns the character as its result value:

```
getChar :: IO Char
```

- The action putChar c writes the character c to the screen, and returns no result value:

```
putChar :: Char → IO ()
```

- The action return v simply returns the value v, without performing any interaction:

```
return :: a → IO a
```

Sequencing



A sequence of actions can be combined as a single composite action using the keyword do.

For example:

```
a :: IO (Char,Char)
a = do x ← getChar
      getChar
      y ← getChar
      return (x,y)
```

Derived Primitives

- Reading a string from the keyboard:

```
getLine :: IO String
getLine = do x ← getChar
            if x == '\n' then
                return []
            else
                do xs ← getLine
                   return (x:xs)
```

- Writing a string to the screen:

```
putStr      :: String → IO ()
putStr []   = return ()
putStr (x:xs) = do putChar x
                  putStr xs
```

- Writing a string and moving to a new line:

```
putStrLn   :: String → IO ()
putStrLn xs = do putStr xs
                putChar '\n'
```

Example



We can now define an action that prompts for a string to be entered and displays its length:

```
strlen :: IO ()
strlen = do putStrLn "Enter a string: "
           xs ← getLine
           putStrLn "The string has "
           putStrLn (show (length xs))
           putStrLnLn " characters"
```

For example:



```
> strlen
```

```
Enter a string: abcde
```

```
The string has 5 characters
```

Note:

- Evaluating an action executes its side effects, with the final result value being discarded.