Plan for Today (>30% of TCEs are in)

Discussion about Final

- 2 sides of 8.5x11 sheet of paper
- See posted notes for tomorrow's recitation for what could be on final.
- Friday will have review and can do examples.

PA5 Suggestions

Register allocation for expressions

What are IO monads?

Expression Evaluation

Sethi-Ullman Register allocation

```
label(node)
    if node is leaf then node.label = 1
    else if node is binary
        if node.left.label == node.right.label
        then node.label = node.left.label + 1
    else
        node.label = max( node.left.label,
        nodel.right.label )
    else if node is unary
        node.label = node.child.label
```

Questions to understand how to answer

- How many registers are needed if not using memory (aka push/pop)?
- Order of evaluation to make this register count work?
- What if an operator has more than 2 operands?

THE IO MONAD

Kathleen Fisher for cs242 at Tufts

Lightly edited with permission, Michelle Strout 4/13/15

Reading: "Tackling the Awkward Squad," Sections 1-2
"Real World Haskell," Chapter 7: I/O

Thanks to Simon Peyton Jones for many of these slides.

Why Monads?

- Predictive parser
  - Passed around a list of tokens while processing.
- PA3 MeggyJava compiler
  - Passed around a number to create unique labels for code generation.
- PA4 MeggyJava compiler
  - Passing around a symbol table with parameter and method type and code generation information.
- Monads will help us abstract away some of that passing around.

Monadic Input and Output

The Problem

A functional program defines a pure function, with no side effects.

Tension

The whole point of running a program is to have some side effect.

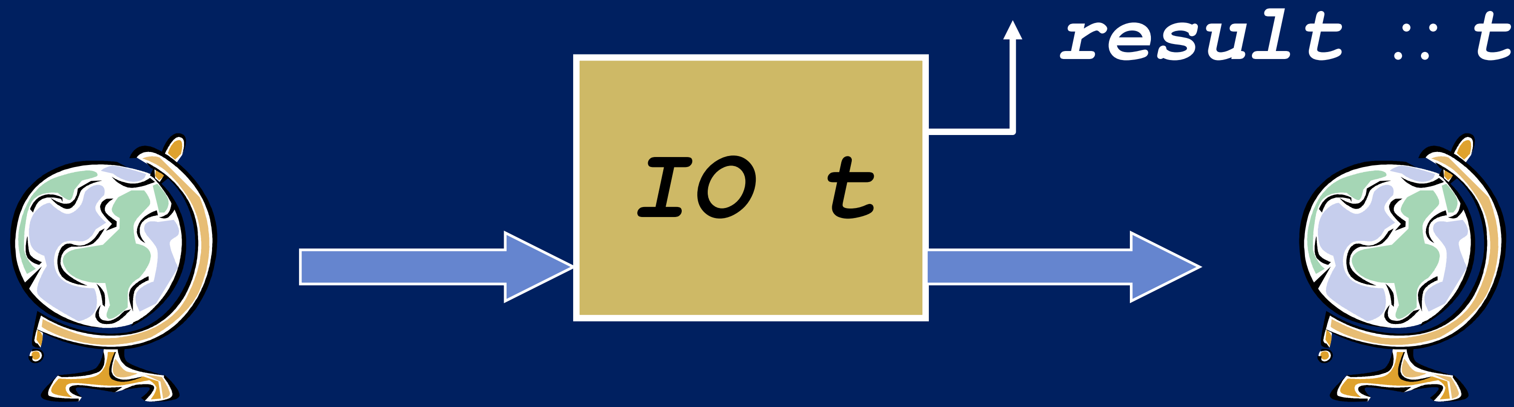
Monadic I/O: The Key Idea

A value of type (`IO t`) is an "action." When performed, it may do some input/output before delivering a result of type `t`.

A Helpful Picture

A value of type (`IO t`) is an "action." When performed, it may do some input/output before delivering a result of type `t`.

```
type IO t = World -> (t, World)
```



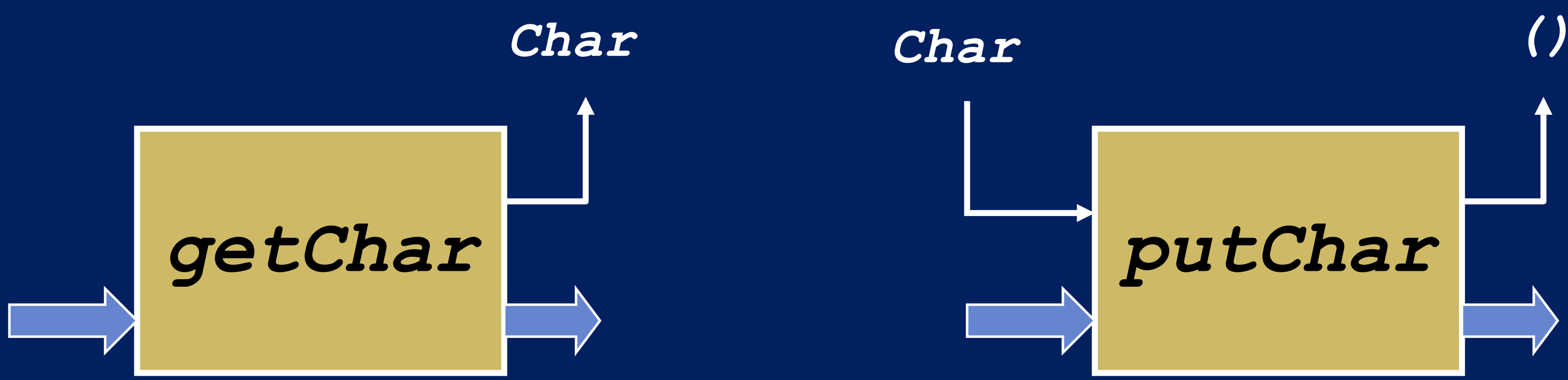
Actions are First Class

A value of type (`IO t`) is an "action." When performed, it may do some input/output before delivering a result of type `t`.

```
type IO t = World -> (t, World)
```

- "Actions" are sometimes called "computations."
- An action is a first-class value.
- Evaluating an action has no effect; performing the action has the effect.

Simple I/O



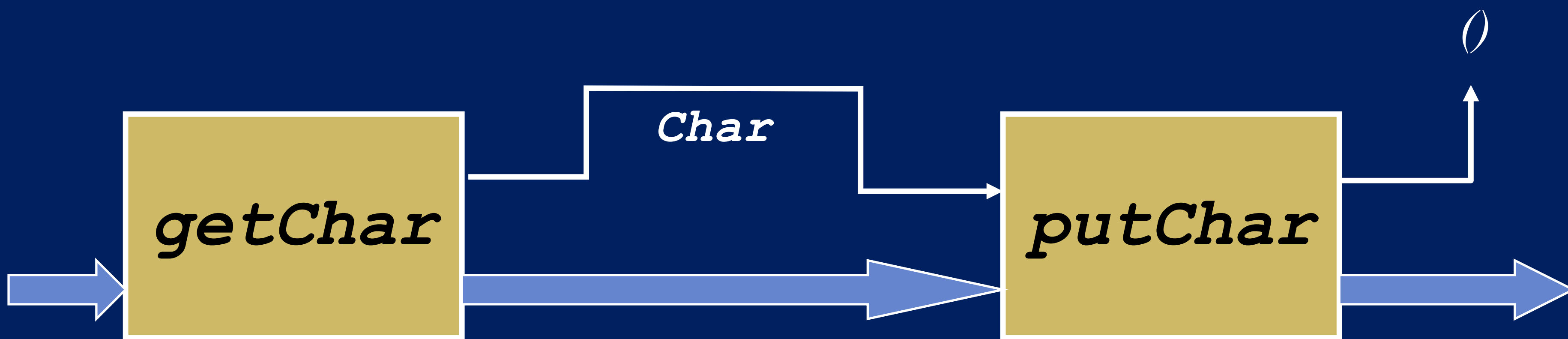

```
getChar :: IO Char
putChar :: Char -> IO ()

main :: IO ()
main = putChar 'x'
```

Main program is an action of type IO ()

Connection Actions

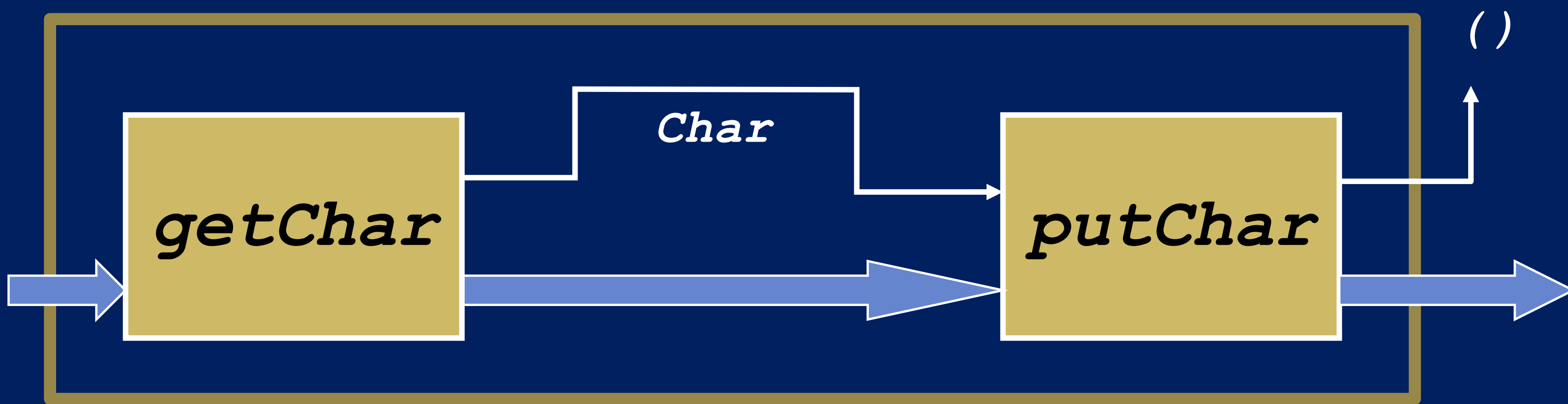
To read a character and then write it back out, we need to connect two actions.



The "bind" combinator lets us make these connections.

The Bind Combinator (>>=)

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

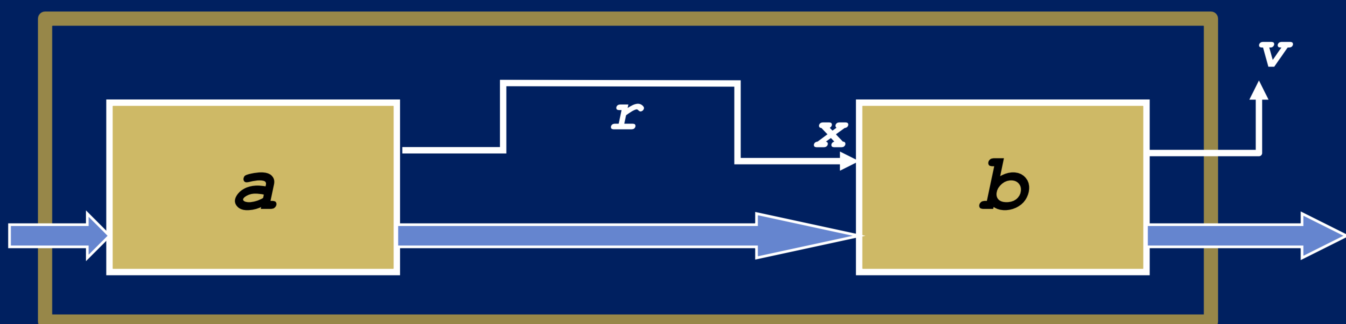


- We have connected two actions to make a new, bigger action.

```
echo :: IO ()
echo = getChar >>= putChar
```

The (>>=) Combinator

- Operator is called bind because it *binds* the result of the left-hand action in the action on the right.
- Performing compound action `a >>= \x->b`:
  - performs action `a`, to yield value `r`
  - applies function `\x->b` to `r`
  - performs the resulting action `b{x <- r}`
  - returns the resulting value `v`



Printing a Character Twice

```
echoDup :: IO ()
echoDup = getChar    >>= (\c  ->
          putChar c  >>= (\() ->
          putChar c  ))
```

- The parentheses are optional because lambda abstractions extend "as far to the right as possible."
- The `putChar` function returns unit, so there is no interesting value to pass on.

The (>>) Combinator

- The "then" combinator (>>) does sequencing when there is no value to pass:

```
(>>) :: IO a -> IO b -> IO b
m >> n  =  m >>= (\_ -> n)
```

```
echoDup :: IO ()
echoDup = getChar     >>= \c  ->
          putChar c  >>
          putChar c
```

```
echoTwice :: IO ()
echoTwice = echo >> echo
```

Getting Two Characters

```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar   >>= \c1 ->
              getChar   >>= \c2 ->
              ????
```

- We want to return `(c1,c2)`.
  - But, `(c1,c2) :: (Char, Char)`
  - And we need to return something of type `IO(Char, Char)`
- We need to have some way to convert values of “plain” type into the I/O Monad.

The `return` Combinator

- The action (`return v`) does no IO and immediately returns `v`:

```
return :: a -> IO a
```


```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar   >>= \c1 ->
              getChar   >>= \c2 ->
              return (c1,c2)
```

The "do" Notation

- The "do" notation adds syntactic sugar to make monadic code easier to read.

```
-- Plain Syntax
getTwoChars :: IO (Char,Char)
getTwoChars = getChar   >>= \c1 ->
              getChar   >>= \c2 ->
              return (c1,c2)
```

```
-- Do Notation
getTwoCharsDo :: IO(Char,Char)
getTwoCharsDo = do { c1 <- getChar ;
                     c2 <- getChar ;
                     return (c1,c2) }
```

- Do syntax designed to look imperative.

Desugaring "do" Notation

- The "do" notation *only* adds syntactic sugar:

```
do { x<-e; es }      =   e >>= \x -> do { es }
do { e; es }         =   e >> do { es }
do { e }             =   e
do {let ds; es}      =   let ds in do {es}
```

The scope of variables bound in a generator is the rest of the "do" expression.

The last item in a "do" expression must be an expression.

Syntactic Variations

- The following are equivalent:

```
do { x1 <- p1; ...; xn <- pn; q }
```

```
do   x1 <- p1; ...; xn <- pn; q
```

```
do x1 <- p1
   ...
   xn <- pn
   q
```

If the semicolons are omitted, then the generators must line up. The indentation replaces the punctuation.

Bigger Example

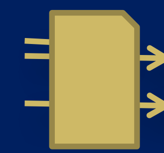
- The **getLine** function reads a line of input:

```
getLine :: IO [Char]
getLine = do { c <- getChar ;
               if c == '\n' then
                    return []
               else
                    do { cs <- getLine;
                         return (c:cs) }}
```

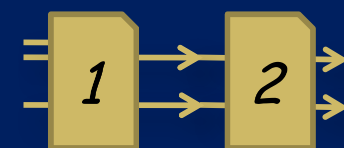
Note the "regular" code mixed with the monadic operations and the nested "do" expression.

An Analogy: Monad as Assembly Line

- Each action in the IO monad is a possible stage in an assembly line.



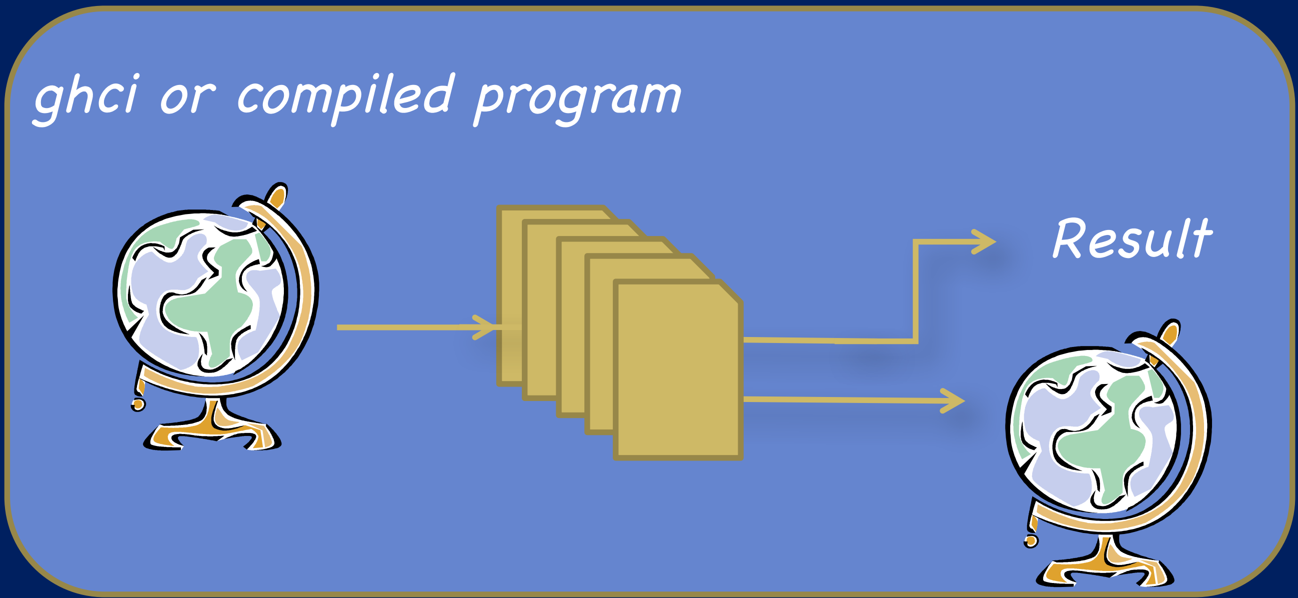
- For an action with type `IO a`, the type
  - tags the action as suitable for the IO assembly line via the `IO` type constructor.
  - indicates that the kind of thing being passed to the next stage in the assembly line has type `a`.



- The bind operator "snaps" two stages s1 and s2 together to build a compound stage.
- The return operator converts a pure value into a stage in the assembly line.
- The assembly line *does nothing* until it is turned on.
- The only safe way to "run" an IO assembly is to execute the program, either using ghci or running an executable.

Powering the Assembly Line

- Running the program turns on the IO assembly line.
- The assembly line gets "the world" as its input and delivers a result and a modified world.
- The types guarantee that the world flows in a single thread through the assembly line.



Implementation

- GHC uses world-passing semantics for the IO monad:

```
type IO t = World -> (t, World)
```

- It represents the "world" by an un-forgeable token of type **`World`**, and implements **`bind`** and **`return`** as:

```
return :: a -> IO a
return a = \w -> (a,w)
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>=) m k = \w -> case m w of (r,w') -> k r w'
```

- Using this form, the compiler can do its normal optimizations. The dependence on the world ensures the resulting code will still be single-threaded.

- The code generator then converts the code to modify the world "in-place."

But what does this mean?

Reference: http://stackoverflow.com/questions/3117583/is-haskell-truly-pure-is-any-language-that-deals-with-input-and-output-outside

Summary

– IO Monad value for Haskell main is like a program AST

– The IO Monad value is evaluated in the sense that IO actions are bound together sequentially including some IO actions that contain lambda function values based on input.

– The ghc compiler then converts these IO Monad values into C code and executes the C code at runtime.

→ **Show usage of monad for compiler example.**