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

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

The Problem
Monadic I/O: The Key Idea

A value of type \((\text{IO } t)\) is an “action.” When performed, it may do some input/output before delivering a result of type \(t\).
A value of type \( \text{IO} \ t \) is an “action.” When performed, it may do some input/output before delivering a result of type \( t \).

\[
\text{type IO } t = \text{World } \rightarrow (t, \text{World})
\]
Actions are First Class

A value of type \((\text{IO } t)\) 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 ()
To read a character and then write it back out, we need to connect two actions.

The “bind” combinator lets us make these connections.
We have connected two actions to make a new, bigger action.

```
echo :: IO ()
echo = getChar >>= putChar
```
The (\(\gg\gg=\)) Combinator

- Operator is called \textit{bind} because it \textit{binds} the result of the left-hand action in the action on the right.

- Performing compound action \(a \gg\gg= \lambda x\to b\):
  - performs action \(a\), to yield value \(r\)
  - applies function \(\lambda x\to b\) to \(r\)
  - performs the resulting action \(b\{x \leftarrow r\}\)
  - returns the resulting value \(v\)
Printing a Character Twice

\[
\text{echoDup} :: \text{IO} \ ()
\]
\[
\text{echoDup} = \text{getChar} \gg= (\backslash c \mapsto \text{putChar} c \gg= (\backslash () \mapsto \text{putChar} c))
\]

- The parentheses are optional because lambda abstractions extend “as far to the right as possible.”
- The \text{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:

\[
(\ggg) \;::\; \text{IO} \;a \;\rightarrow\; \text{IO} \;b \;\rightarrow\; \text{IO} \;b
\]

\[
m \ggg n = m \gggg (\_ \rightarrow n)
\]

echoDup :: IO ()
echoDup = getChar >>= \_ \rightarrow putChar c >> putChar c

echoTwice :: IO ()
echoTwice = echo >>= echo
We want to return \((c_1, c_2)\).

But, \((c_1, c_2) :: (\text{Char}, \text{Char})\)

And we need to return something of type

\(\text{IO}(\text{Char}, \text{Char})\)

We need to have some way to convert values of “plain” type into the I/O Monad.

```haskell
getTwoChars :: IO (Char, Char)
getTwoChars = getChar >>= \c1 ->
               getChar >>= \c2 ->
               ????
```
The **return** Combinator

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

  \[ \text{return} :: a \rightarrow \text{IO} \ a \]

- Haskell Monad Functions:

  ```haskell
  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

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

--- Do Notation

```haskell
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:

\[
\begin{align*}
\text{do } \{ x<-e; es \} & \quad = \quad e \gg= \ \lambda x \rightarrow \text{do } \{ es \} \\
\text{do } \{ e; es \} & \quad = \quad e \gg \text{do } \{ es \} \\
\text{do } \{ e \} & \quad = \quad e \\
\text{do } \{ \text{let } ds; es \} & \quad = \quad \text{let } ds \text{ in do } \{ es \}
\end{align*}
\]

- 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.
The following are equivalent:

\[
\begin{align*}
\text{do } \{ & x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n; q \}\ \\
\text{do } & x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n; q \\
\text{do } & x_1 \leftarrow p_1 \\
& \quad \ldots \\
& \quad x_n \leftarrow p_n \\
& \quad q
\end{align*}
\]

If the semicolons are omitted, then the generators must line up. The indentation replaces the punctuation.
The `getLine` function reads a line of input:

```haskell
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.
GHC uses world-passing semantics for the IO monad:

\[
\text{type } \text{IO } t = \text{World } \to (t, \text{World})
\]

It represents the "world" by an un-forgeable token of type \text{World}, and implements \text{bind} and \text{return} as:

\[
\text{return} :: a \to \text{IO } a
\]
\[
\text{return} a = \lambda w \to (a, w)
\]
\[
(\gg=) :: \text{IO } a \to (a \to \text{IO } b) \to \text{IO } b
\]
\[
(\gg=) m k = \lambda w \to \text{case } m w \text{ of } (r, w') \to k r w'
\]

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?


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.