Haskell: I/O II
Haskell: Data

CS 331 Programming Languages

Lecture Slides

Monday, March 2, 2026

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Unit Overview
The Haskell Programming Language

Topics

- ✓ PL feature: type system
- ✓ PL category: functional PLs
- ✓ Introduction to Haskell
- ✓ Haskell: functions
- ✓ Haskell: lists
- ✓ Haskell: flow of control
- ✓ Haskell: I/O I
- Haskell: I/O II
- Haskell: data

Review

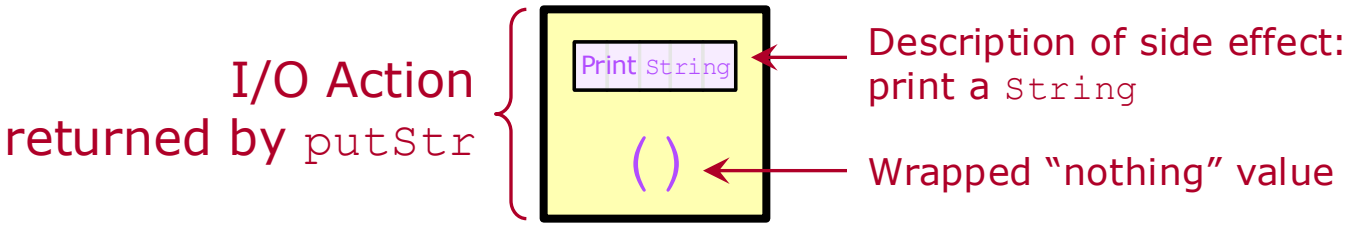
Haskell: I/O I — Simple Output, I/O Actions [1/2]

A Haskell **I/O action** holds a description of zero or more side effects plus a wrapped value.

Example. Function `putStr` takes a `String` and returns an I/O action representing printing the `String` to the standard output.

```
> :t putStr
putStr :: String -> IO ()
```

Return type: I/O action wrapping a "nothing" value



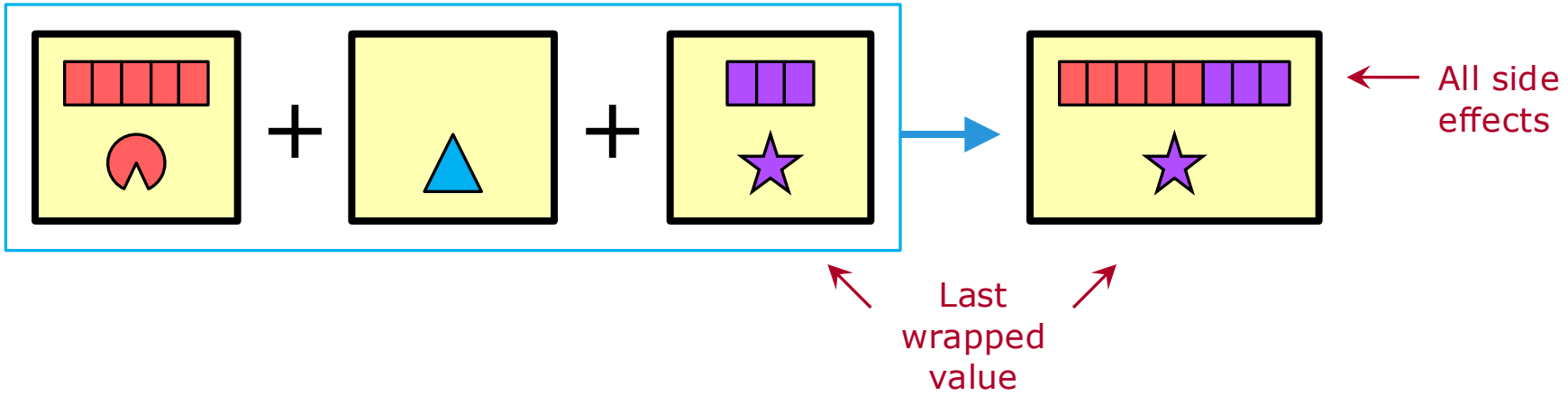
In a Haskell program, `main` needs to return an I/O action. The side effects described are performed by the runtime environment.

For code from this topic, see `io1.hs`.

Haskell: I/O I — Simple Output, I/O Actions [2/2]

Multiple I/O actions can be combined into one, which holds:

- A description of all side effects from the combined I/O actions.
- The wrapped value from the last of the combined I/O actions.



The >> operator combines I/O actions in this way.

```
> putStr "Today's number: " >> putStr (show (32+5))
                            >> putStrLn ""
Today's number: 37
```

Conversion to `String`

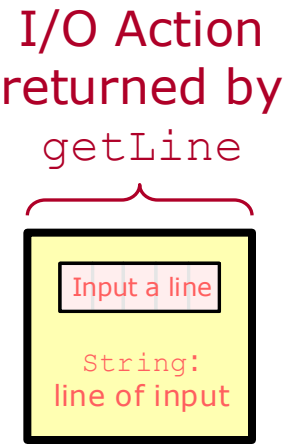
Haskell: I/O I — Simple Input [1/3]

When we do input, we get an I/O action that wraps the value we are inputting.

```
> :t getLine
getLine :: IO String
```

The returned I/O action wraps a `String`.

`getLine` returns an I/O action whose side effect is inputting a line of text from the standard input. The wrapped value is a `String`: the line of text, without the ending newline.



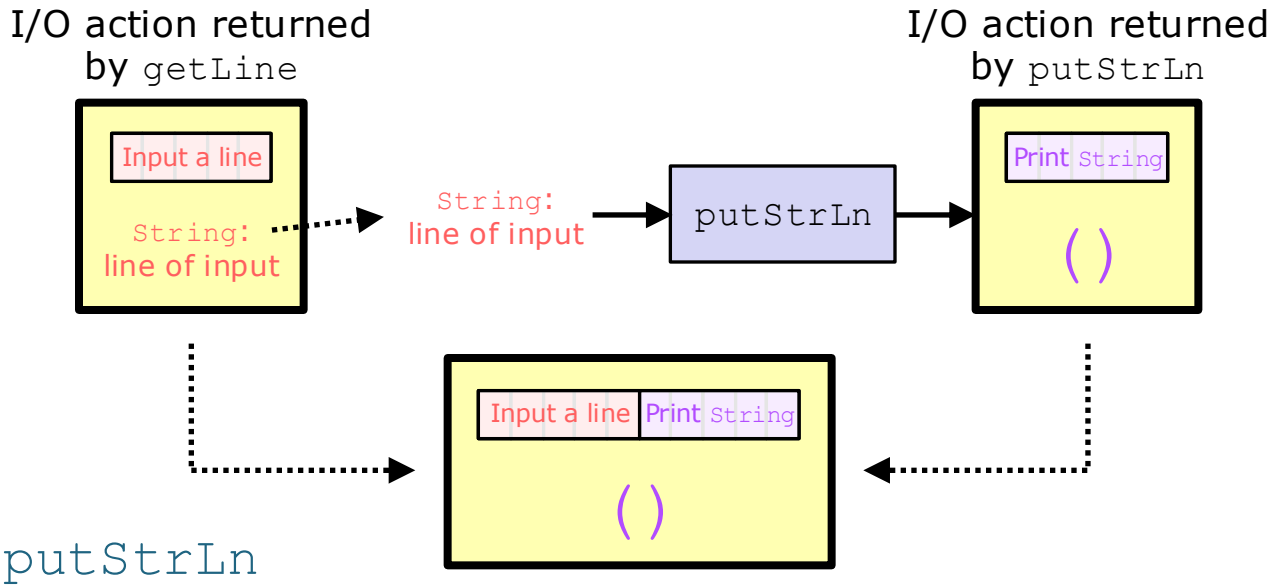
We can access the wrapped `String`, not by pulling it out of the I/O action—which cannot be done—but by pushing a function in, using the `>>=` operator.

Haskell: I/O I — Simple Input [2/3]

The `>>=` operator has two operands:

- an I/O action wrapping a value, and
- a function that takes such a value and returns an I/O action.

For example, suppose we do "`getLine >>= putStrLn`".




```
> getLine >>= putStrLn
Howdy!
Howdy!
```

Typed by user

Haskell: I/O I — Simple Input [3/3]

The `>>` and `>>=` operators can be used together:

```
> putStr "Type something: " >> getLine >>= putStrLn
Type something: I like hamsters!
I like hamsters!
```

We can give the parameter of `putStrLn` a name:

Same as `putStrLn` (right?)

```
> getLine >>= (\ line -> putStrLn line)
Hamsters rule ...
Hamsters rule ...
> getLine >>= (\ line -> putStrLn (reverse line))
... this planet and others like it.
.ti ekil srehto dna tenalp siht ...
```

Haskell: I/O II

Haskell: I/O II
Do-Expression [1/2]

Haskell's **do-expression** offers a cleaner way to write I/O.

The keyword `do` is followed by an indented block. I/O actions in the block are combined into a single I/O action. Internally, this is done using the `>>` and `>>=` operators.

For code from this topic, see `io2.hs`.

Using operators:

```
putStr s >> putStrLn t
```

Using a do-expression:

```
do
    putStr s
    putStrLn t
```

Using operators:

```
getLine >>=
    (\ line -> putStrLn line)
```

Using a do-expression:

```
do
    line <- getLine
    putStrLn line
```

Used inside a do-expression, "`<-`" binds the name `line` to the I/O-wrapped value. Variable `line` can then be used in the rest of the do-expression.

Haskell: I/O II
Do-Expression [2/2]

TO DO

- Write a function that inputs a line of text and then prints a message indicating the number of characters entered. Use a do-expression.

Done. See `io2.hs`.

Useful

- Function `hFlush` is given a *handle** to an open file; it returns an I/O action that *flushes*** the file. Do "`hFlush stdout`" after printing a prompt and before doing input, to ensure that the prompt appears before input is entered.
- `hFlush` is a Haskell standard-library function, but it is not in the prelude. To use it in a source file, do "`import System.IO`" near the beginning of the file.

***Handle**: object that identifies and allows access to an open file.

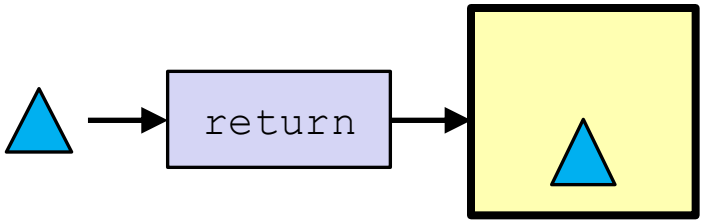
****Flush**: write any buffered characters.

Haskell: I/O II
return [1/2]

So far, every I/O action we have used has described one or more side effects. But an I/O action can also involve zero side effects.

To create a no-side-effect I/O action wrapping a particular value, pass the value to `return`.

- `return x` produces a do-nothing I/O action that wraps `x`.
- `return ()` produces a do-nothing, wrap-"nothing" I/O action.



Haskell's `return` does not return! It simply creates a do-nothing I/O action. However, generally we only use `return` as the last thing in a do-expression, so it *feels* like it returns.

Otherwise, it is pointless (right?).

Haskell: I/O II
return [2/2]

`getChar` does single-character input. It returns an I/O action wrapping a `Char`.

TO DO

- Using `getChar`, write `myGetLine`, which does the same thing as `getLine`.
- Write code that uses `myGetLine`.

Done. See `io2.hs`.

Now we have examples of flow of control involving I/O: both selection (`if … then … else`) and repetition (using recursion).

Haskell: I/O II
"let" in a Do-Expression [1/2]

One last bit of do-expression syntax remains. We can use `let` *NAME* `=` *EXPR* inside a do-expression to bind a name to a normal value (not I/O-wrapped) for the remainder of the block.

```
foo = do
    let n = 42
    putStr "n = "
    putstrLn $ show n
    let nsq = n*n
    putStr "n + n*n = "
    putStrLn $ show (n+nsq)
```

Haskell: I/O II
"let" in a Do-Expression [2/2]

TO DO

- Write a program of the kind that might be assigned early in Computer Science I. Input a number. If the number is zero, then quit. Otherwise, print a message giving some value computed from the number (its square?), and repeat.

Done. See `io2.hs`*.*
For a stand-alone version, see `squarenums.hs`*.*

A Haskell function must have a consistent return type. If one branch of a selection control structure (pattern matching, `if` … `then` … `else`, guards) returns an I/O action, then the other branches must also return I/O actions.

Summary of Haskell I/O

- Haskell string conversions are largely separate from I/O.
  - `show` converts a value to a `String`.
  - `read` gets a value from a `String`. A type annotation may be required.
- An **I/O action** holds a description of a sequence of side effects plus a wrapped value.
- I/O is performed by returning an I/O action from a program.
- Combine multiple I/O actions into one with the `>>` and `>>=` operators. The result includes all side effects, last wrapped value.
- A **do-expression** is nicer. It is syntactic sugar around `>>` and `>>=`.
- Inside an I/O do-expression:
  - "*NAME* `<-` *EXPR*" binds an identifier to an I/O-wrapped value.
  - "`let` *NAME* `=` *EXPR*" binds an identifier to a non-wrapped value.
  - "`return` *EXPR*" creates a do-nothing (no side effects) I/O action wrapping the given value.

Haskell: I/O II
Notes [2/2]

If a Haskell program uses values obtained via input, then its behavior is dependent on a side effect. So purity would seem to be compromised. (Right?)

Solution: the `>>=` operator (always used when doing input) actually includes the function passed to it in the returned I/O action—not the result of calling the function, but *the function itself*.

What ends up being included in the returned I/O action is a function to call to run *the entire remainder of the program*.

So when we do input, we might as well be saying, "This program is finished. Now do some input, and pass the value to a *separate program*; here is its code." And purity is not compromised.

Haskell: Data

Haskell: Data
data Declaration [1/3]

We finish our coverage of Haskell by looking at Haskell's facilities for defining new types and implementing data structures.

Consider a structure holding information about a product sold in a store. We need to keep the name of the product and the name of the manufacturer.

In C++, we might do something like this:

```
class Product {
private:
    string productName;
    string manufacturerName;
…
};
```

For code from this topic, see `data.hs`.

Haskell: Data
data Declaration [2/3]

The more-or-less equivalent Haskell is a **data declaration**.

```
data Product = Pr String String
  -- product name, manufacturer name
```

`Product` is the name of a new type.

`Pr` is a **constructor** for that type. Literals of type `Product` are marked by the fact that they begin with "`Pr`".

For example, here is a (mostly useless) function that takes a `Product` and returns the same `Product`.

```
doNothing :: Product -> Product
doNothing (Pr pn mn) = Pr pn mn
```

Haskell: Data
data Declaration [3/3]

Pattern matching works with constructors.

We can use this to retrieve names from a `Product` object.

```
-- pName - Get product name from a Product
pName :: Product -> String
pName (Pr pn _) = pn

-- mName - Get manufacturer name from a Product
mName :: Product -> String
mName (Pr _ mn) = mn
```

Haskell: Data
Overloading & Typeclasses [1/3]

Suppose we wish to test whether two `Product` values are the same. We consider this to be true if they have the same product name and the same manufacturer name.

```
sameProduct :: Product -> Product -> Bool
sameProduct (Pr pn1 mn1) (Pr pn2 mn2) =
    (pn1 == pn2) && (mn1 == mn2)
```

But it would be nicer if we could use the "==" operator.

And in fact we *can* do this.

Overloading in Haskell is done using typeclasses. To overload the "==" operator, we use typeclass `Eq`.

Haskell: Data
Overloading & Typeclasses [2/3]

To overload the "==" operator for type `Product`, we place this type into the `Eq` typeclass. We want type `Product` to be an **instance** of class `Eq`.

In the **instance declaration**, we provide a definition of the "==" operator for `Product`.

```
instance Eq Product  where
    Pr pn1 mn1 == Pr pn2 mn2 =
        (pn1 == pn2) && (mn1 == mn2)
```

Now we can use the "==" operator with `Product`.

And we can also use "/=" (the inequality operator). Haskell typeclasses typically include default definitions of overloaded functions in terms of others.

Haskell: Data
Overloading & Typeclasses [3/3]

We can similarly provide conversion to `String` for `Product` (overloading function `show`) by placing `Product` into the `Show` typeclass.

```
instance Show Product  where
    show (Pr pn mn) = pn ++ " [made by " ++ mn ++ "]"
```

In GHCi:

```
> Pr "Tide" "Procter & Gamble"
Tide [made by Procter & Gamble]
```

Haskell: Data
TO BE CONTINUED …

Haskell: Data will be continued next time.