CS4450 Lecture 4 Notes

Monday, August 27, 2018 2:58 PM

Data Declarations

A program in Haskell or other functional languages is some kind of data structure, definitions, and some kind of procedure definitions

Data declarations are how you define completely new types.

Data declarations themselves can also have parameters.

Now, we will go over using GHCi. GHCi is the Haskell interpreter we will be using.

Loading a file into GHCi:

:l ExtractList (:l is short for :load) will load the ExtractList.hs file.
Can also do `ghci ExtractList` (without backticks) in bash to load it.

:t head
    Outputs head :: [a] -> a

:t is short for type
Very useful for remembering what it is that a given library function does.

foo :: A -> B
Can comment out the type and see if it compiles, alternatively can do :t foo in ghci to show the type.

To reload, do :r (:r is short for :reload)
To quit, do :q (:q is short for :quit)

How to Write a Haskell Program

Very soon, we'll get a homework assignment.

Type-Driven Programming in Haskell

Writing a function with type A -> B, then you have a lot of information to use for fleshing out the function.

Recursive Types

In Haskell, new types can be declared in terms of themselves. This means types can be <u>recursive</u>

In Haskell, you don't want to write the whole program at once and then test it. Test as you go to ensure that each piece of the program is typed and working properly as they should.

How to write a Haskell program:

Step 1: Figure out the type of thing you're going to write

... The rest of these are on the online slides since we're out of time.

Additional notes with Haskell code shown in class:

```haskell
data Bool = False | True
-- New types can be used in the same way as built-in types.
-- For example, given
data Answer = Yes | No | Unknown
-- We can define:
answers :: [Answer]
answers = [Yes,No,Unknown]
-- Here is a function we define:
-- If we give it a Yes, it returns a No. If we give it a No, it returns
a Yes. If we give it an Unknown, it returns an Unknown
flip :: Answer -> Answer
flip Yes = No
flip No = Yes
flip Unknown = Unknown
-- Haskell has "type inference", which means you don't have to tell it
the types of anything; it will go through and tell things what type
they should be.
-- It's good practice to write down the type declaration, because when
you start implementing something, the type checker will notice that
something is wrong
something is wrong.
-- Many bugs can be caught by the type checker.
-- You can have constructors that take arguments.
data Shape = Circle Float | Rect Float Float
-- The constructors Circle and Rect are actually functions. The
declaration above says that Circle takes Float -> Shape, and Rect takes
Float -> Float -> Shape.
-- Not surprisingly, data declarations themselves can also have
parameters.
data Maybe a = Nothing | Just a
-- General form of a Haskell module:
module ModuleName where
import L1 -- imports
...
import Lk
data D1 = ... -- Type declaration
...
data Dn = ...
f1 = ...
...
fm = ...
-- The Prelude is included in Haskell by default and is a standard
module.
-- Imports must go at the top, but the data declarations and functions
can be implemented anywhere in the program.
-- A lot of languages require you to define these things from top to
bottom, but that really isn't the case with this language (with the
exception of imports).
-- newtype: Like data but only one constructor.
data Nat = Zero | Succ Nat
-- Nat is a new type, with constructors
Zero :: Nat
Succ :: Nat -> Nat -- Effectively, Succ means +1. If you give it a Nat,
it returns a Nat.
Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero)) represents (1 + (1 + (1 + 0))
-- Using recursion, it is easy to define functions that convert between
values of type Nat and Int:
nat2int :: Nat -> Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n
int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
-- Two naturals can be added by converting them to ints, add them as
ints, and then convert them into nats.
-- How do we add them if we don't want to convert them to ints via the
above "trick"? You can use Succ
```

```haskell
-- Write a function that:
-- 1. Takes a list of items
-- 2. Takes a function that returns either True or False on those
items,
-- 3. and returns a list of all the items on which the function is
true.
-- This is called filter, and it's a built in function in Haskell, but
here's how it would be written from scratch:
-- Function is called "myfilter" to avoid name clash with built-in
function.
-- The below is a constructor.
data [a] = [] | a:[a] -- A list of items is either an empty list of
items OR an item followed by a list of items.
myfilter :: [a] ->
(a -> Bool) ->
[a]
-- myfilter [] f = undefined -- Declare this as undefined first and
then change each undefined to the type one by one, compiling with ghci
as you go. If there is an error, you'll know that the line you just
changed is the issue.
-- myfilter (a:as) f = undefined
myfilter [] f = []
-- Type of f is a -> Bool
myfilter (a:as) f = if fa then a : myfilter as f else myfilter as f --
Recursion here.
-- The above is how to write a function in this style.
```