Where We’re Going
So far in this course, we have

- covered pretty much all the features of the Haskell language
- gotten significant practical experience with Haskell programming in homework assignments
- seen some of the most common idioms used in Haskell programs and libraries
  - higher-order programming (functions as data)
  - programming with type classes
  - monadic style
Where We’re Going

My aim in the rest of the course is to use all of this as a foundation for discussing and experimenting with a number of topics:

- modularity (modular decomposition, design by contract, unit testing, documentation, etc.)
- programming as writing (programming for people, not computers)
- more examples of interesting libraries and abstractions (pretty printing combinators, “arrows,” etc.)
- larger applications of Haskell’s idioms and features to practical programming problems
  - in particular: web site construction
Concretely

- The term projects (which will take up about the last 1/3 of the semester) will involve web site construction on top of the WASH/CGI framework.
- Most of the remaining homework assignments will be devoted to building up infrastructure for this (exploring various parts of the WASH/CGI framework, etc.)
- Class time will be divided between
  - examining various aspects of WASH/CGI
  - discussion of people’s solutions to assignments
  - discussion of programming from various angles
  - other miscellaneous examples and topics (suggestions welcome!)

Warning: All this will be a little less structured than what we’ve done up to this point!
As fodder for the programming discussions, we’ll be reading selections from

The Pragmatic Programmer
by Andrew Hunt and David Thomas

You’ll need a copy of this book in about a week. (Should we get the bookstore to order it?)
XML Processing Combinators
A number of people have built high-level libraries (a.k.a. embedded languages) for manipulating and transforming XML in Haskell.

- HaXml (Wallace and Runciman)
- HXML (English)
- etc.

We’ll look at the first one today, and (hopefully) come back to the second later in the course.
<album>
	<title>Time Out</title>
	<artist>Dave Brubeck Quartet</artist>
	<recording date="June-August 1959" place="NYC"/>
	<coverart style='abstract'>
		<location thumbnail='pix/small/timeout.jpg' fullsize='pix/covers/timeout.jpg'/>
	</coverart>
	<catalogno label='Columbia' number='CL 1397' format='mono'/>
	<catalogno label='Columbia' number='CS 8192' format='stereo'/>
	<catalogno label='Columbia' number='CPK 1181' format='LP' country='Korea'/>
	<catalogno label='Sony/CBS' number='Legacy CK 40585' format='CD'/>
	<personnel>
		-player name='Dave Brubeck' instrument='piano'/>
		-player name='Paul Desmond' instrument='alto sax'/>
		-player name='Eugene Wright' instrument='bass'/>
		-player name='Joe Morello' instrument='drums'/
	</personnel>
	<track title='Blue Rondo &agrave; la Turk' credit='Brubeck' timing='6m42s'/>
	<track title='Strange Meadow Lark' credit='Brubeck' timing='7m20s'/>
	<track title='Take Five' credit='Desmond' timing='5m24s'/>
	<track title='Three To Get Ready' credit='Brubeck' timing='5m21s'/>
	<track title='Kathy’s Waltz' credit='Brubeck' timing='4m48s'/>
	<track title='Everybody’s Jumpin’' credit='Brubeck' timing='4m22s'/>
	<track title='Pick Up Sticks' credit='Brubeck' timing='4m16s'/>
	<notes>
		Possibly the DBQ’s most famous album, this contains
		-trackref link='#3'>Take Five</trackref>, the most famous jazz track
		of that period. These experiments in different time signatures are
	what Dave Brubeck is most remembered for. Recorded Jun-Aug 1959 in
	NYC. See also the sequel,
	<albumref link='cbs-timefurthout'>Time Further Out</albumref>.
	</notes></album>
Essentially the same as the one we have been using:

```haskell
data Element = Elem Name [Attribute] [Content]
type Attribute = (Name, AttValue)
data Content = CElem Element
              | CText CharData

  type Name = String
  type AttValue = String
  type CharData = String
```

(N.b.: This is slightly simplified from the real implementation.)
All document transformations are content filters. A filter takes a single XML Content value and returns a sequence of Content values (possibly empty).

```haskell
type CFilter = Content -> [Content]
```
Selection Filters
Selecting Children

Throw away current node and return its children.

children :: CFilter

children (CElem (Elem _ _ cs)) = cs
children _ = []
Select the \( n \)’th positional result of a filter.

\[
\text{position} :: \text{Int} \rightarrow \text{CFilter} \rightarrow \text{CFilter}
\]

\[
\text{position} \ n \ f = (\cs \mapsto [\cs !! n]) \ . \ f
\]

(N.b.: not described in the paper.)
Applying

\[ \text{ex2} = \text{position 1 children} \]

to our sample XML yields:

\[ <\text{artist}>\text{Dave Brubeck Quartet}</\text{artist}> \]
Throw away current node and return one of its attributes.

\[
\text{showAttr} :: \text{String} \rightarrow \text{CFilter} \\
\text{showAttr key c@CElem (Elem _ as _)) =} \\
[\text{CText (lookfor key as)}] \\
\text{where} \\
\text{lookfor x as =} \\
\text{case (lookup x as) of} \\
\text{Nothing} \rightarrow \\
\text{(error ("missing attribute: "++show x))} \\
\text{Just v} \rightarrow \text{v} \\
\text{showAttr _ _ = []} \\
\]

(N.b.: the current library does this a little differently — see the on-line documentation.)
rename an element tag.

\[
\text{replaceTag} :: \text{String} \rightarrow \text{CFilter}
\]

\[
\text{replaceTag} \ n \ (\text{CElem} \ (\text{Elem} \ _ \ _ \ \text{cs})) = \\
[\text{CElem} \ (\text{Elem} \ \text{n} \ [] \ \text{cs})]
\]

\[
\text{replaceTag} \ n \ _ = \\
[]
\]

(Note that one might quibble with the details of this!)
Construction Filters
The filter \texttt{literal} \( s \) ignores its argument and yields \( s \):

\[
\text{literal} :: \text{String} \to \text{CFilter}
\]

\[
\text{literal} \ s = \_ \to [\text{CText} \ s]
\]

Also (in the paper, but not in the current library)...\[
(\!) = \text{literal}
\]

...so we can write \texttt{("hello"!)} instead of \texttt{(literal "hello")}, using Haskell’s “section” syntax.
Build an element with the given tag name. Its content is formed by concatenating the results of the given list of filters.

```
mkElem :: String -> [CFilter] -> CFilter
mkElem h cfs = \t-> [ CElem (Elem h []) (cat cfs t) ]

cat :: [CFilter] -> CFilter
cat fs = \e-> concat [ f e | f <- fs ]
```

Similarly, `mkElemAttr` builds an element with the given name, attributes, and content. The attributes are specified by a list of (string, filter) pairs.
ex1 =
    mkElem "html"
    [mkElem "body"
      [literal "Hello world"]]

Running this on our sample XML (or any other XML document!) yields:

<html><body>Hello world</body></html>
ex4 =
    mkElem "html"
       [mkElem "body"
            [position 1 children,
             position 2 children]]

Running this on our sample XML yields:

```
<html><body>
   <artist>Dave Brubeck Quartet</artist>
   <recording date="June-August 1959" place="NYC"/>
</body></html>
```
Predicate Filters
The `keep` filter takes any content and returns it; the `none` filter takes any content and returns nothing.

```haskell
keep, none :: CFfilter

keep = \x->[x]
none = \x->[]
```
These filters either keep or throw away some content based on a simple test: \texttt{elm} keeps only a tagged element, \texttt{txt} keeps only non-element text, \texttt{tag} keeps only an element with the named tag, \texttt{attr} keeps only an element with the named attribute, \texttt{attrval} keeps only an element with the given attribute value, \texttt{tagWith} keeps only an element whose tag name satisfies the given predicate.

\begin{verbatim}
elm, txt :: CFilter
tag :: String -> CFilter
attr :: Name -> CFilter
attrval :: Attribute -> CFilter
tagWith :: (String->Bool) -> CFilter
\end{verbatim}
elm x@(CElem _)  = [x]
elm _            = []

txt x@(CText _)  = [x]
txt _            = []

tag t x@(CElem (Elem n _ _)) | t==n  = [x]
tag t _          = []

tagWith p x@(CElem (Elem n _ _)) | p n   = [x]
tagWith p _     = []

attr n x@(CElem (Elem _ as _)) | n 'elem' (map fst as) = [x]
attr n _         = []

attrval av x@(CElem (Elem _ as _)) | av 'elem' as = [x]
attrval av _    = []
Combinators
Apply the left filter to the results of the right filter.

\[ o :: \text{CFilter} \rightarrow \text{CFilter} \rightarrow \text{CFilter} \]

\[ f \ 'o' \ g = \text{concat} \ . \ \text{map} \ f \ . \ g \]
Running

```
ex5 = tag "personnel" 'o' children
```

on our sample XML yields:

```
<personnel>
  <player name="Dave Brubeck" instrument="piano"/>
  <player name="Paul Desmond" instrument="alto sax"/>
  <player name="Eugene Wright" instrument="bass"/>
  <player name="Joe Morello" instrument="drums"/>
</personnel>
```
The filter

```python
ex3 = txt 'o' children 'o' tag "title"
```

means “only the plain-text children of the current element, provided the current element has the tag title.”
Binary parallel composition. Each filter gets a copy of the input, rather than one filter using the result of the other.

```
union :: CFilter -> CFilter -> CFilter

f 'union' g = \c -> f c ++ g c
```

(Called `|||` in the paper.)
Running

ex6 =
    mkElem "stuff"
        [(tag "personnel" union tag "catalogno") o children]

on our XML yields:

<stuff>
    <catalogno label="Columbia" number="CL 1397" format="mono"/>
    <catalogno label="Columbia" number="CS 8192" format="stereo"/>
    <catalogno label="Columbia" number="CPK 1181" format="LP"
              country="Korea"/>
    <catalogno label="Sony/CBS" number="Legacy CK 40585" format="CD"/>
    <personnel>
        <player name="Dave Brubeck" instrument="piano"/>
        <player name="Paul Desmond" instrument="alto sax"/>
        <player name="Eugene Wright" instrument="bass"/>
        <player name="Joe Morello"
              instrument="drums"/>
    </personnel>
</stuff>
On the other hand, running

```
ex7 =
    mkElem "stuff"
        [(tag "personnel" 'o' children)
         'union'
         (tag "catalogno" 'o' children)]
```

on our XML yields:

```
<stuff>
    <personnel>
        <player name="Dave Brubeck" instrument="piano"/>
        <player name="Paul Desmond" instrument="alto sax"/>
        <player name="Eugene Wright" instrument="bass"/>
        <player name="Joe Morello" instrument="drums"/></personnel>
    <catalogno label="Columbia" number="CL 1397" format="mono"/>
    <catalogno label="Columbia" number="CS 8192" format="stereo"/>
    <catalogno label="Columbia" number="CPK 1181" format="LP"
               country="Korea"/>
    <catalogno label="Sony/CBS" number="Legacy CK 40585"
               format="CD"/></stuff>
```
Pruning

\[ f \text{ `with` } g \text{ keeps (just) those results of } f \text{ for which } g \text{ also produces at least one result.} \]

```haskell
with :: CFFilter -> CFFilter -> CFFilter

f 'with' g = filter (not.null.g) . f
```
Running

```python
ex8 =
    (children 'o' tag "personnel" 'o' children)
'with'
    (attrval ("instrument","bass"))
```

on our XML yields:

```xml
<player name="Eugene Wright" instrument="bass"/>
```

(N.b.: Needs to be modified slightly to run with the real library.)
Similarly, \( f \text{ ‘without’ } g \) keeps (just) those results of \( f \) for which \( g \) does not produce any results.

\[
\text{without} :: \text{CFilter} \rightarrow \text{CFilter} \rightarrow \text{CFilter}
\]

\[
f \text{ ‘without’ } g = \text{filter} \ (\text{null}.g) \ . \ f
\]
Conditionals
These definitions provide C-like conditionals, lifted to the filter level.

The \((\text{cond} \ ? \ \text{yes} : \ \text{no})\) notation in C becomes \((\text{cond} \ ?> \ \text{yes} :> \ \text{no})\) in Haskell.

---

---

-- Conjoin the two branches of a conditional.
data ThenElse a = a :> a

-- Select between the two branches of a joined conditional.
(?>) :: (a->[b]) -> ThenElse (a->[b]) -> (a->[b])
p ?> (f :> g) = \c-> if (not.null.p) c then f c else g c
f |>| g returns g’s results only if there are no f-results

(|>|) :: (a->[b]) -> (a->[b]) -> (a->[b])

f |>| g =
\x-> let fx = f x in if null fx then g x else fx

Alternatively: f |>| g = f ?> f :> g
Path-like Selection

Pronounced “slash”, \( f \twoheadrightarrow g \) means \( g \) inside \( f \) :

\[
(/>): \text{CFilter} \to \text{CFilter} \to \text{CFilter}
\]

\( f /> g = g \circ \text{children} \circ f \)

Pronounced “outside”, \( f \leftarrow\neg\neg\neg\rightarrow g \) means \( f \) containing \( g \)

\[
(</): \text{CFilter} \to \text{CFilter} \to \text{CFilter}
\]

\( f <\neg\neg\neg\rightarrow g = f \text{with} (g \circ \text{children}) \)
Editing
Process **Children In Place.** The filter is applied to any children of an element content, and the element rebuilt around the results.

```
chip :: CFilter -> CFilter

chip f (CElem (Elem n as cs)) = [ CElem (Elem n as (concat (map f cs))) ]
chip f c = [c]
```
Recursion
Recursive application of filters: a fold-like operator.

foldXml :: CFfilter -> CFfilter

foldXml f = f ‘o‘ chip (foldXml f)
Running

```haskell
ex9 =
    foldXml
    ((replaceTag "MUSICIAN" 'o' tag "player") |> | keep)
```

on our XML yields:

```xml
<album>
    <title>Time Out</title>
    <artist>Dave Brubeck Quartet</artist>
    <recording date="June-August 1959" place="NYC"/>
    <coverart style="abstract">
        <location thumbnail="pix/small/timeout.jpg"
            fullsize="pix/covers/timeout.jpg"/>
    </coverart>
    <catalogno label="Columbia" number="CL 1397" format="mono"/>
    <catalogno label="Columbia" number="CS 8192" format="stereo"/>
    <catalogno label="Columbia" number="CPK 1181" format="LP"
        country="Korea"/>
    <catalogno label="Sony/CBS" number="Legacy CK 40585" format="CD"/>
    <personnel>
        <MUSICIAN/>
        <MUSICIAN/>
        <MUSICIAN/>
        <MUSICIAN/>
    </personnel>
    <track title="Blue Rondo à la Turk" credit="Brubeck"
        timing="6m42s"/>
    <track title="Strange Meadow Lark" credit="Brubeck"
        timing="7m20s"/>
</album>
```
deep, deepest, multi :: CFilter -> CFilter

deep f = f |>| (deep f ‘o‘ children)
deepest f = (deepest f ‘o‘ children) |>| f
multi f = f ‘union‘ (multi f ‘o‘ children)