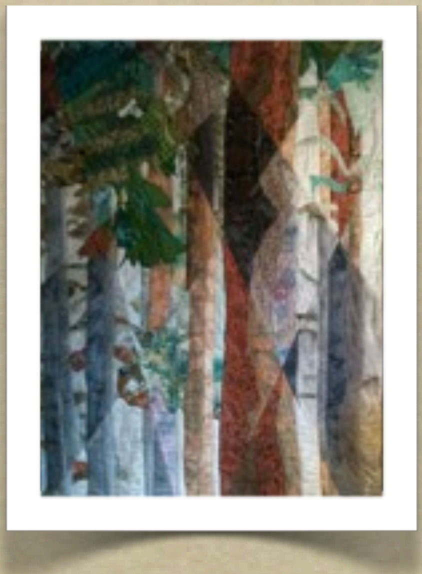


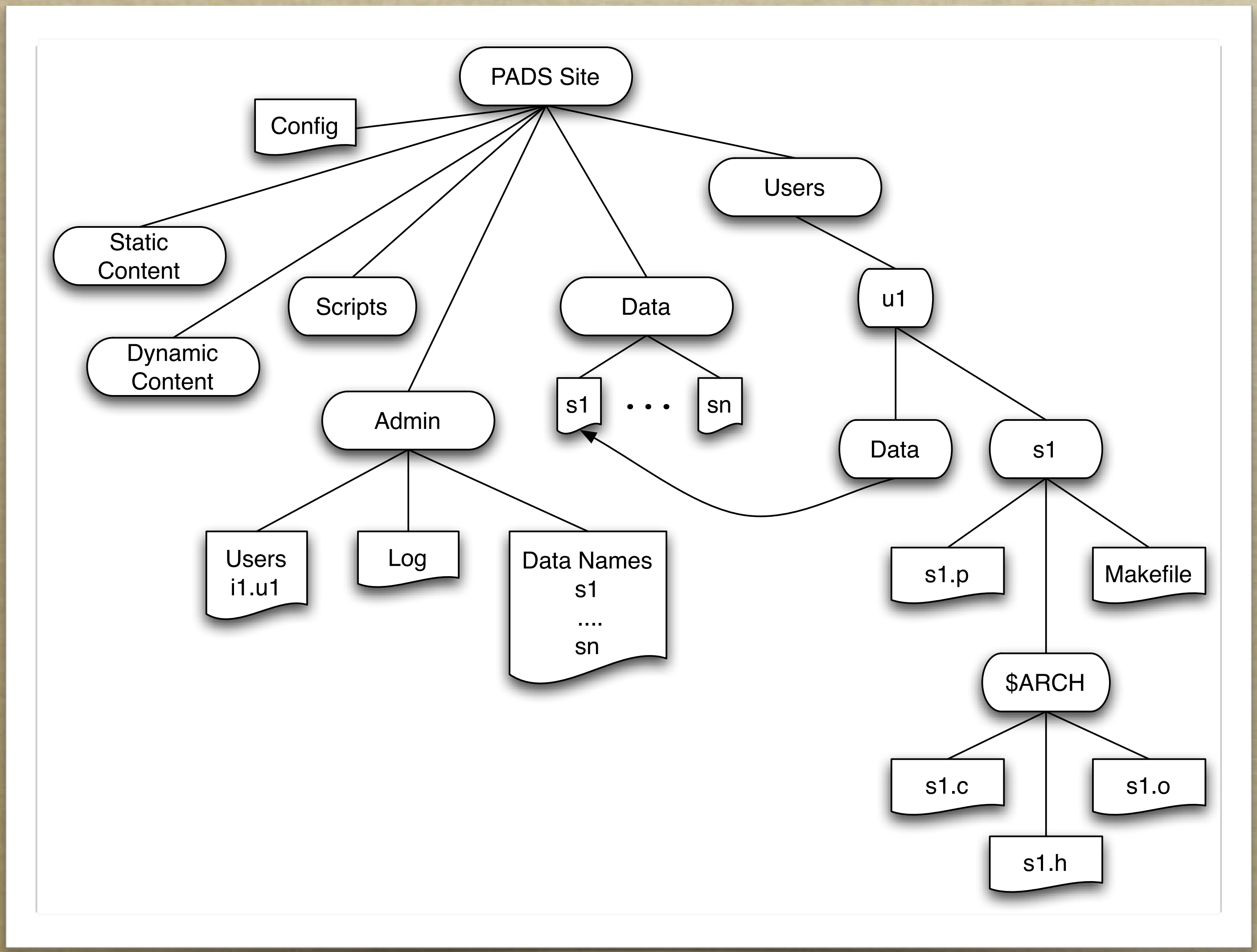
Forest: Typing FileStores

Kathleen Fisher

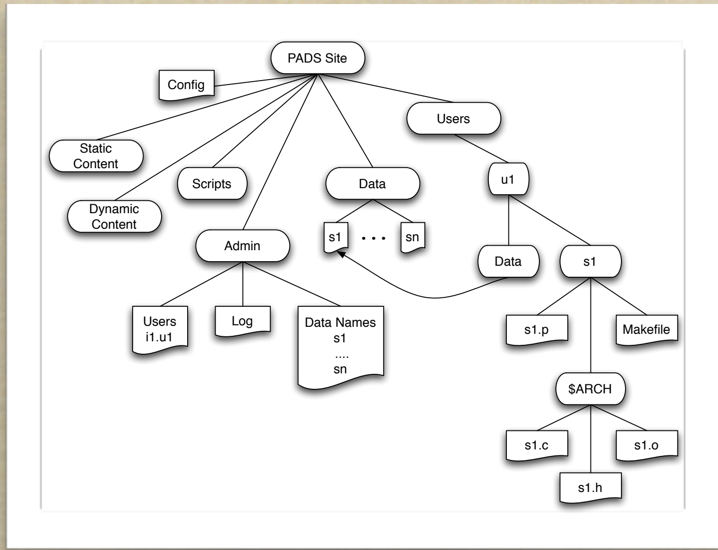
Joint work with Nate Foster, David Walker, and Kenny Zhu.

What is a FileStore?!!?

*A **FileStore** is a collection of directories, files, and symbolic links that constitutes a coherent collection of data.*



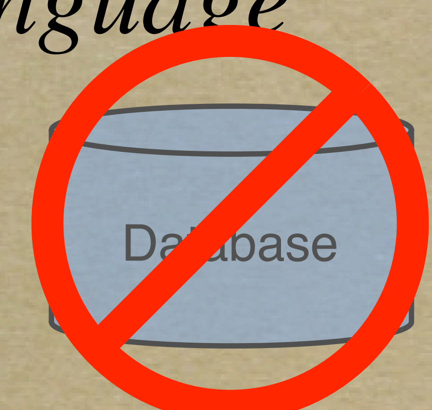
Example FileStores



- *Monitoring Systems:*
  - *Various AT&T monitoring applications*
  - *CoralCDN*
- *Scientific data sets:*
  - *Astronomy: Huge Data but Small Programs, ...*
  - *Ecology: CORIE system for Columbia river estuary, ...*
  - *Physics: The physics of soft matter, ...*
- *Code bases:*
  - *Pads, Haskell, Linux, Websites, ...*
- *Ad hoc databases:*
  - *Princeton computer science department records*

Why not a database?

- ***Up-front cost****: choosing a database, possibly paying for it, learning how to use it.*
- ***Challenges loading data****: potentially long-load times, high indexing overheads, and tedious data transformations.*
- ***Losing control****: all access must be through database interface.*
- ***Programming overhead****: tedium of interfacing the database to conventional programming language*



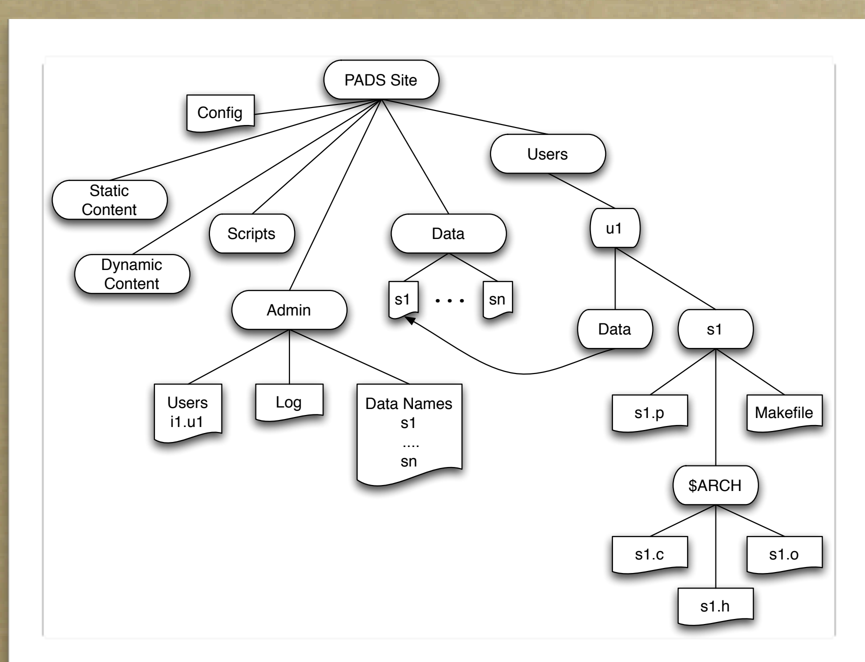
Challenges with FileStores

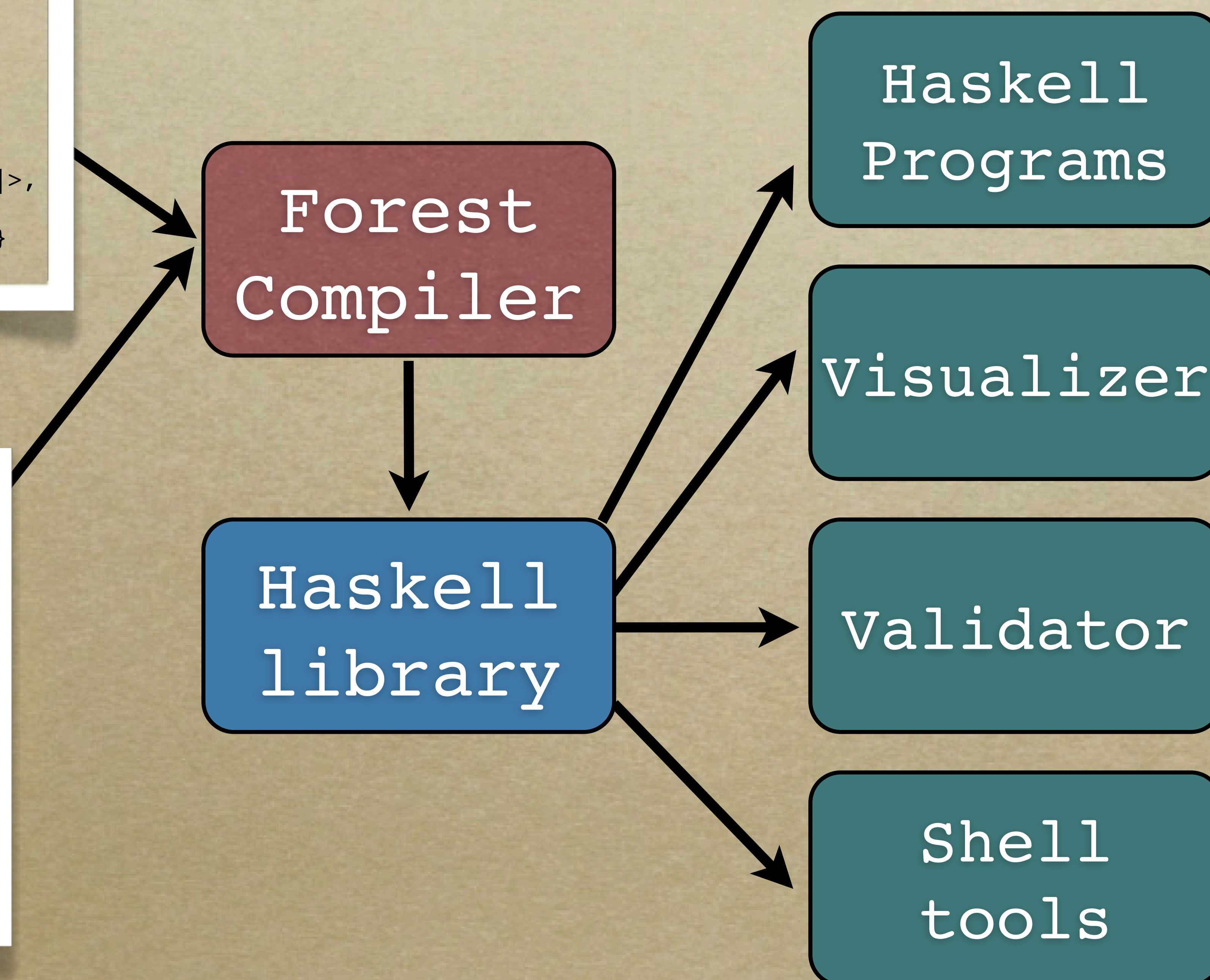
- *Documentation typically lacking: difficult maintenance, hard to learn.*
- *No systematic way to detect errors in FileStore.*
- *Tools must be built from scratch.*
- *Tools can't document assumptions about FileStore, so format evolution can cause silent failures.*
- *Scale (numbers of files, size of files) complicates things further.*

A Solution: Forest

Type-based specification language for FileStores.

```
[| forest |
 type Website_d(config::FilePath)  = Directory {
  c       is    config     :: Config,
  static  is <| gdst c  |> :: Static_d,
  dynamic is <| gcgi c  |> :: Cgi_d,
  scripts is <| gspt c  |> :: Scripts_d,
  adm     is <| gdst c  |> :: Info_d,
  data    is <| (gln c)++"/examples/data" |>
          :: DataSource_d <|(gSrc admin_info)|>,
  usr     is <| groot c  |>
          :: Usr_d <|(gpath data_md, adm)|>  }
|]
```





Benefits of Forest

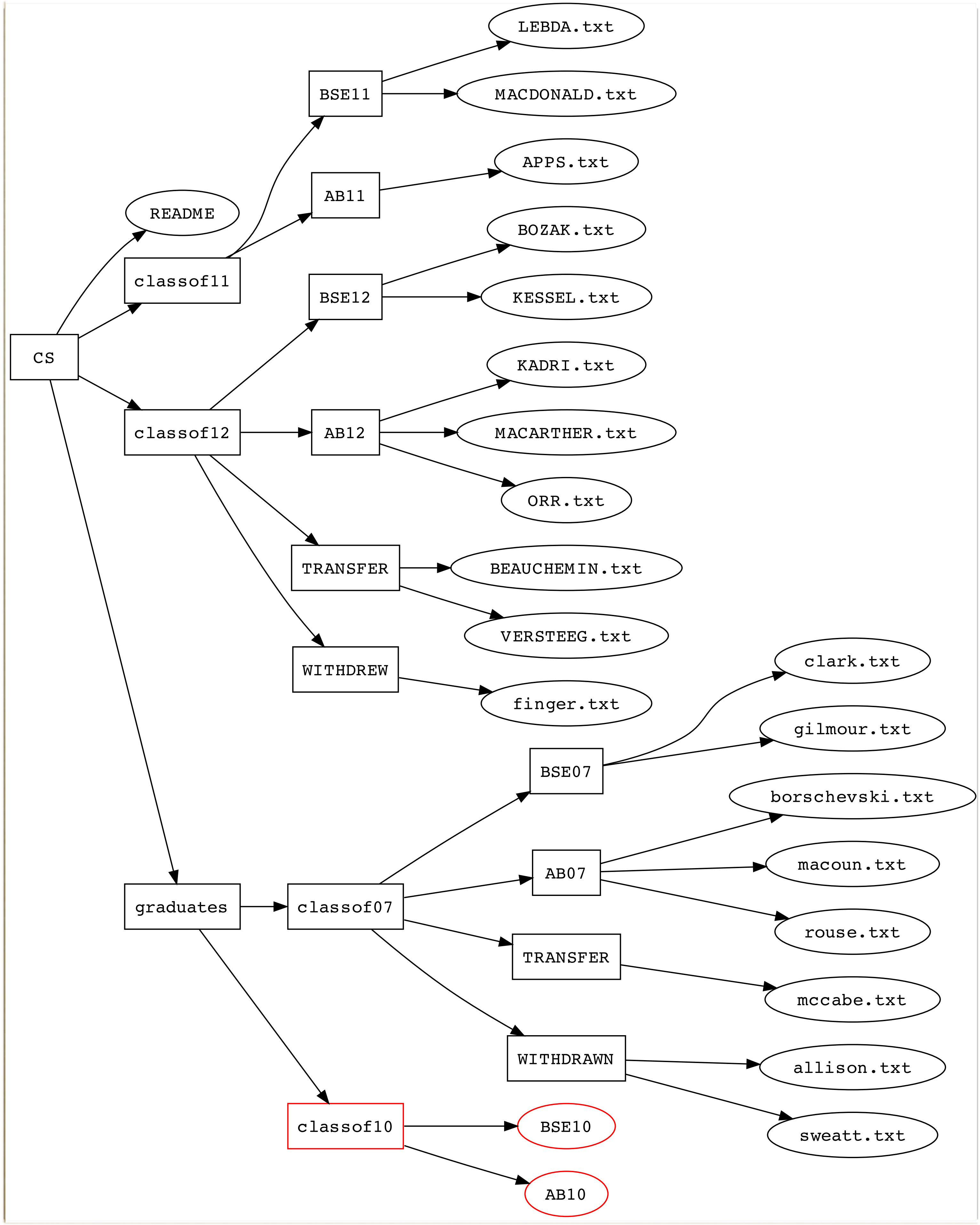
- *Executable documentation*
- *Generated data structures for in-memory representation of data and meta-data*
- *Generated (lazy) loading function*
- *Class instances to enable generic programming*
- *Access to existing generic tools*

```
[| forest |
 type Website_d(config::FilePath)  = Directory {
  c        is    config      :: Config,
  static   is <| gdst c |> :: Static_d,
  dynamic  is <| gcgi c |> :: Cgi_d,
  scripts  is <| gspt c |> :: Scripts_d,
  adm      is <| gdst c |> :: Info_d,
  data     is <| (gln c)++"/examples/data" |>
           :: DataSource_d <|(gSrc admin_info)|>,
  usr      is <| groot c |>
           :: Usr_d <|(gpath data_md, adm)|>  }
|]
```

Outline

- *Introduction*
- *Forest design*
- *Programming with Forest*
- *Tools*
- *Implementation*
- *Future work*

Princeton CS Department



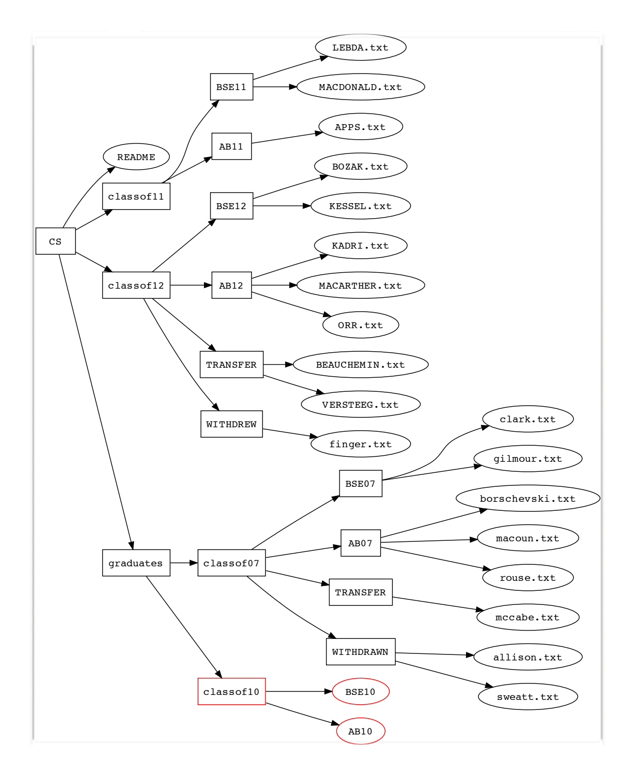
Princeton CS Department

```
[forest|
   type PrincetonCS (y::Integer) = Directory
    { notes      is "README" :: Text
    , seniors    is <|mkClass y        |> :: Class y
    , juniors    is <|mkClass (y + 1)|> :: Class <| y + 1 |>
    , graduates :: Grads
    }

  type Class (y :: Integer) = Directory
    { bse is <|"BSE" ++ (toStrN y 2)|> :: Major
    , ab  is <|"AB"  ++ (toStrN y 2)|> :: Major
    , transfer  matches transferRE  :: Maybe Major
    , withdrawn matches withdrawnRE :: Maybe Major
    , leave     matches leaveRE     :: Maybe Major
    }

  type Grads =
     Map [ c :: Class <| getYear c |> | c <- matches cRE ]

  type Major = Map
    [ s :: File (Student <| dropExtension s |>)
    | s <- matches "GL *.txt",  <| (not . template) s |>   ]
|]
```



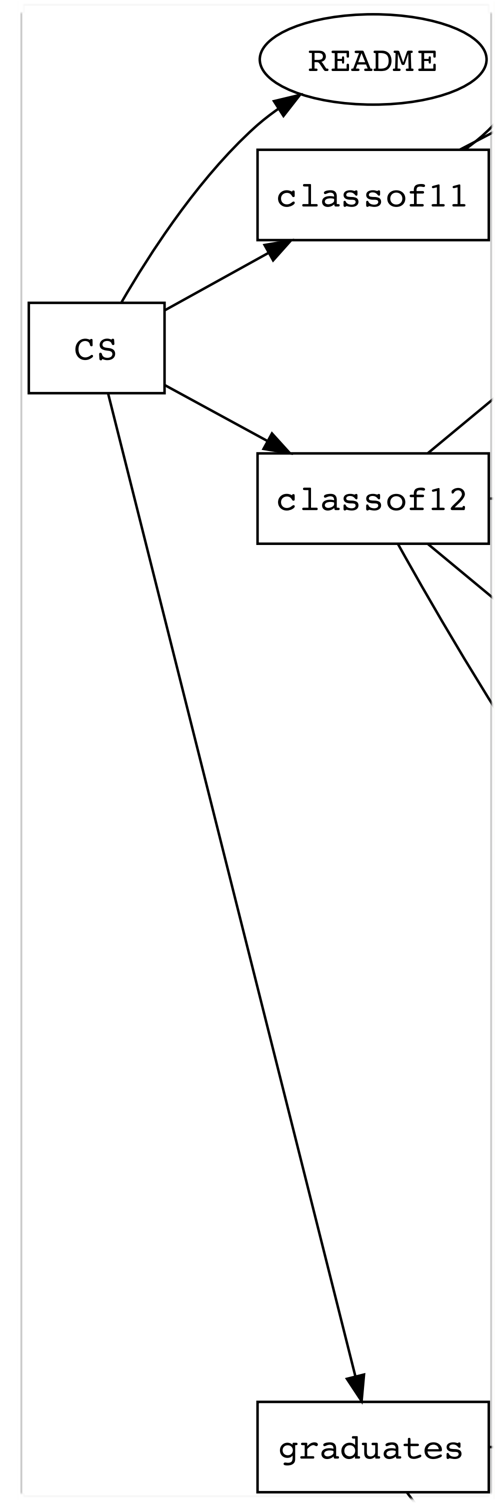
Princeton CS Department

```
[forest|
   type PrincetonCS (y::Integer) = Directory
    { notes      is "README" :: Text
    , seniors    is <|mkClass y        |> :: Class y
    , juniors    is <|mkClass (y + 1)|> :: Class <| y + 1 |>
    , graduates :: Grads
    }

  type Class (y :: Integer) = Directory
    { bse is <|"BSE" ++ (toStrN y 2)|> :: Major
    , ab  is <|"AB"  ++ (toStrN y 2)|> :: Major
    , transfer  matches transferRE  :: Maybe Major
    , withdrawn matches withdrawnRE :: Maybe Major
    , leave     matches leaveRE     :: Maybe Major
    }

  type Grads =
     Map [ c :: Class <| getYear c |> | c <- matches cRE ]

  type Major = Map
    [ s :: File (Student <| dropExtension s |>)
    | s <- matches "GL *.txt",  <| (not . template) s |>  ]
|]
```



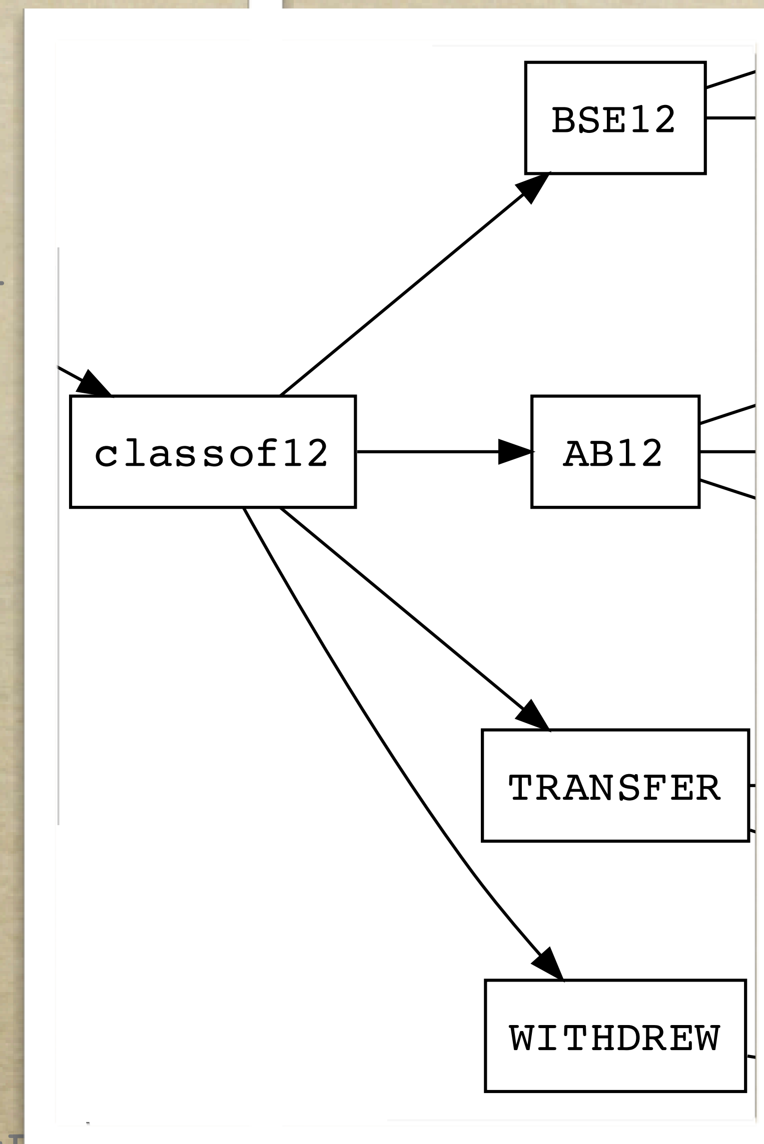
Princeton CS Department

```
[forest|
   type PrincetonCS (y::Integer) = Directory
    { notes      is "README" :: Text
    , seniors    is <|mkClass y       |> :: Class y
    , juniors    is <|mkClass (y + 1)|> :: Class <| y +
    , graduates :: Grads
    }

  type Class (y :: Integer) = Directory
    { bse is <|"BSE" ++ (toStrN y 2)|> :: Major
    , ab  is <|"AB"  ++ (toStrN y 2)|> :: Major
    , transfer  matches transferRE  :: Maybe Major
    , withdrawn matches withdrawnRE :: Maybe Major
    , leave     matches leaveRE     :: Maybe Major
    }

  type Grads =
     Map [ c :: Class <| getYear c |> | c <- matches cRE ]

  type Major = Map
    [ s :: File (Student <| dropExtension s |>)
    | s <- matches "GL *.txt",  <| (not . template) s |>  ]
|]
```



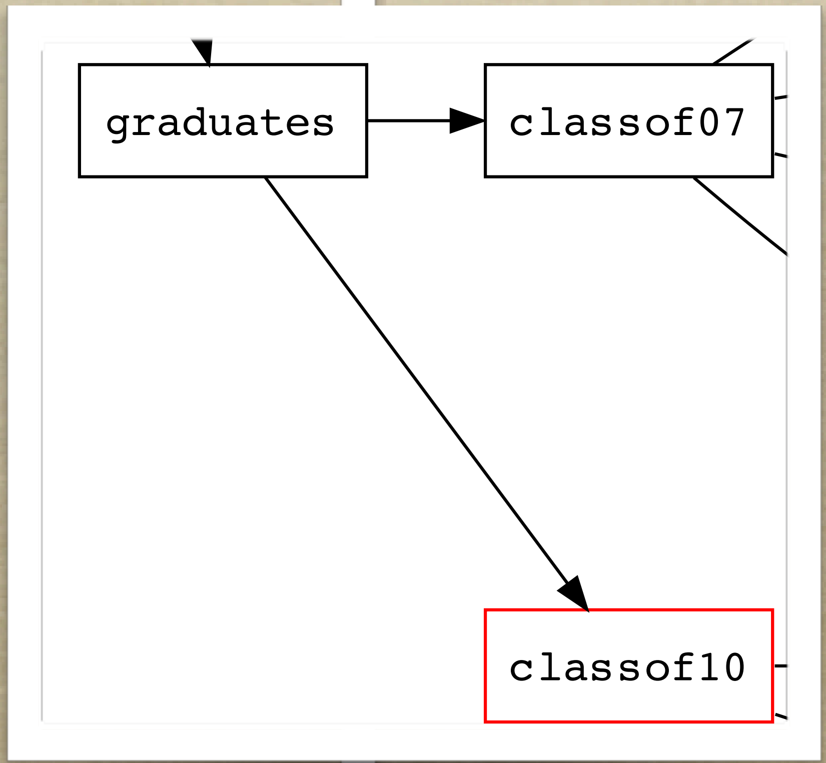
Princeton CS Department

```
[forest|
   type PrincetonCS (y::Integer) = Directory
    { notes     is "README" :: Text
    , seniors   is <|mkClass y      |> :: Class y
    , juniors   is <|mkClass (y + 1)|> :: Class <| y + 1 |>
    , graduates :: Grads
    }

  type Class (y :: Integer) = Directory
    { bse is <|"BSE" ++ (toStrN y 2)|> :: Major
    , ab  is <|"AB"  ++ (toStrN y 2)|> :: Major
    , transfer  matches transferRE  :: Maybe Major
    , withdrawn matches withdrawnRE :: Maybe Major
    , leave     matches leaveRE     :: Maybe Major
    }

  type Grads =
     Map [ c :: Class <| getYear c |> | c <- matches cRE ]

  type Major = Map
    [ s :: File (Student <| dropExtension s |>)
    | s <- matches “GL *.txt”,  <| (not . template) s |>  ]
|]
```



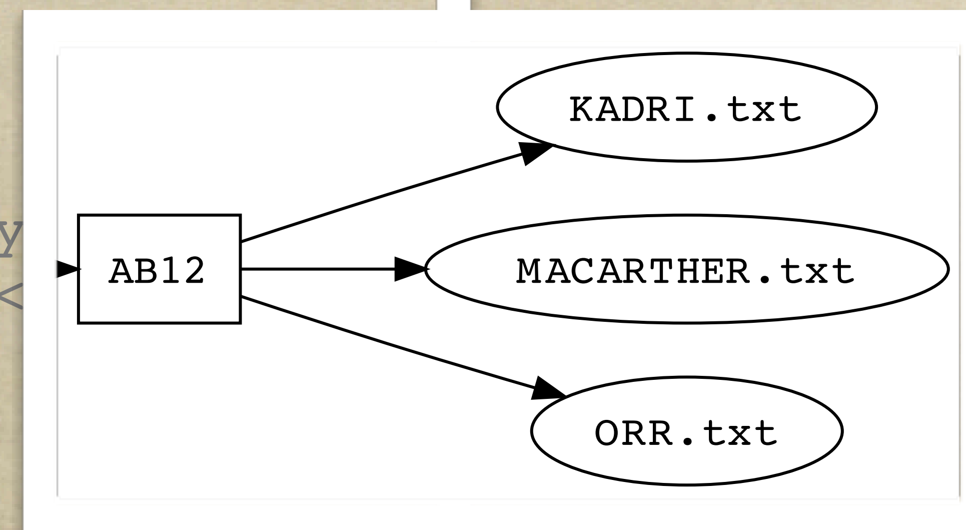
Princeton CS Department

```
[forest|
   type PrincetonCS (y::Integer) = Directory
    { notes      is "README" :: Text
    , seniors    is <|mkClass y       |> :: Class y
    , juniors    is <|mkClass (y + 1)|> :: Class <
    , graduates :: Grads
    }

  type Class (y :: Integer) = Directory
    { bse is <|"BSE" ++ (toStrN y 2)|> :: Major
    , ab  is <|"AB"  ++ (toStrN y 2)|> :: Major
    , transfer  matches transferRE  :: Maybe Major
    , withdrawn matches withdrawnRE :: Maybe Major
    , leave     matches leaveRE     :: Maybe Major
    }

  type Grads =
     Map [ c :: Class <| getYear c |> | c <- matches cRE ]

  type Major = Map
    [ s :: File (Student <| dropExtension s |>)
    | s <- matches GL "*.txt",  <| (not . template) s |>  ]
|]
```



Princeton CS Department

```
[forest|
   type PrincetonCS (y::Integer) = Directory
    { notes     is "README" :: Text
    , seniors   is <|mkClass y        |> :: Class y
    , juniors   is <|mkClass (y + 1)|> :: Class <| y + 1 |>
    , graduates :: Grads
    }

  type Class (y :: Integer) = Directory
    { bse is <|"BSE" ++ (toStrN y 2)|> :: Major
    , ab  is <|"AB"  ++ (toStrN y 2)|> :: Major
    , transfer  matches transferRE  :: Maybe Major
    , withdrawn matches withdrawnRE :: Maybe Major
    , leave     matches leaveRE     :: Maybe Major
    }

  type Grads =
     Map [ c :: Class <| getYear c |> | c <- matches cRE ]

  type Major = Map
    [ s :: File (Student <| dropExtension s |>)
    | s <- matches GL "*.txt ", <| (not . template) s |>  ]
|]
```

Red indicates Haskell expression

Princeton CS Department

```
[forest|
   type PrincetonCS (y::Integer) = Directory
    { notes     is "README" :: Text
    , seniors   is <|mkClass y        |> :: Class y
    , juniors   is <|mkClass (y + 1)|> :: Class <| y + 1 |>
    , graduates :: Grads
    }

  type Class (y :: Integer) = Directory
    { bse is <|"BSE" ++ (toStrN y 2)|> :: Major
    , ab  is <|"AB"  ++ (toStrN y 2)|> :: Major
    , transfer  matches transferRE  :: Maybe Major
    , withdrawn matches withdrawnRE :: Maybe Major
    , leave     matches leaveRE     :: Maybe Major
    }

  type Grads =
     Map [ c :: Class <| getYear c |> | c <- matches cRE ]

  type Major = Map
    [ s :: File (Student <| dropExtension s |>)
    | s <- matches "GL *.txt",  <| (not . template) s |>  ]
|]
```

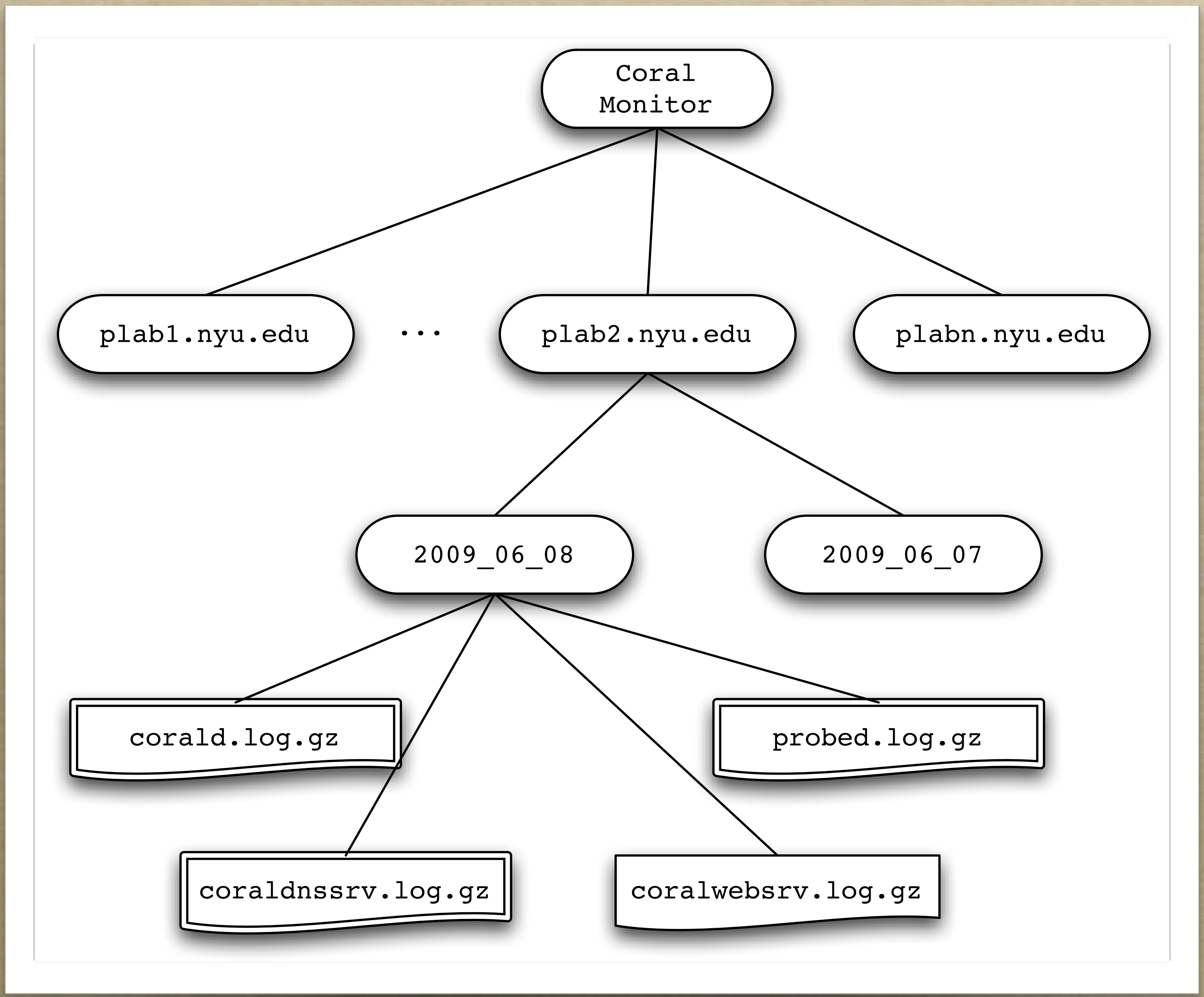
Blue indicates Pads description

Universal Description

```
[forest|
  type Universal_d = Directory
   { ascii_files  is [ f :: Text
                     | f <- matches (GL "*"),
                    <| get_kind  f_att == AsciiK       |> ]
   , binary_files is [ b :: Binary
                     | b <- matches (GL "*"),
                    <| get_kind  b_att == Binary       |> ]
   , directories is [ d :: Universal_d
                     | d <- matches (GL "*"),
                    <| get_kind  d_att == DirectoryK   |> ]
   , symLinks     is [ s :: SymLink
                     | s <- matches (GL "*"),
                    <| get_isSym s_att == True         |> ]
   }
|]
```

Coral CDN

Hosts in Coral CDN periodically send usage statistics to a central server.



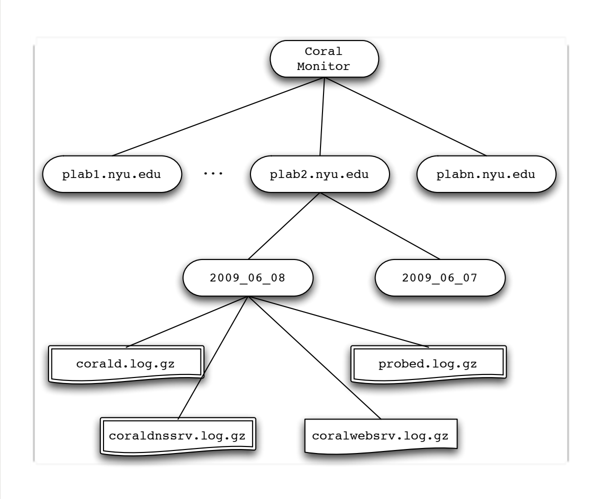
Coral CDN Description

```
[forest|
  type Top  = [ s :: Site | s <- matches siteRE ]
  type Site = [ d :: Log  | d <- matches timeRE ]

  type Log = Directory
    { web is "coralwebsrv.log.gz" :: Gzip  (File Coral),
      dns is "coraldnssrv.log.gz" :: Maybe (Gzip (File Ptext)),
      prb is "probed.log.gz"      :: Maybe (Gzip (File Ptext))
      dmn is "corald.log.gz"      :: Maybe (Gzip (File Ptext)) }
|]

[pads|
  type Coral = [Line Entry]
  data Entry =
    { header  :: Header,   comma_ws
    , payload :: InOut }                |]

comma_ws = REd ",[ \t]*" ", "
timeRE   = RE  "[0-9]{4}_[0-9]{2}_[0-9]{2}-[0-9]{2}_[0-9]{2}"
siteRE   = RE  "[^.].*"
```



Outline

- *Introduction*
- *Forest design*
- *Programming with Forest*
- *Tools*
- *Implementation*
- *Future work*

Forest Rep Types

```
[forest|
  type Top  = [ s :: Site | s <- matches siteRE ]
  type Site = [ d :: Log  | d <- matches timeRE ]

  type Log = Directory
    { web is "coralwebsrv.log.gz" :: Gzip  (File Coral),
      dns is "coraldnssrv.log.gz" :: Maybe (Gzip (File Ptext)),
      prb is "probed.log.gz"      :: Maybe (Gzip (File Ptext))
      dmn is "corald.log.gz"      :: Maybe (Gzip (File Ptext)) }
|]
```

```
newtype Top   = Top   [(Path, Site)]
newtype Site  = Site  [(Path, Log) ]

data Log = Log { web :: Coral
               , dns :: Maybe Ptext,
               , prb :: Maybe Ptext,
               , dmn :: Maybe Ptext }
```

Forest MetaData Types

```
[forest|
  type Top  = [ s :: Site | s <- matches siteRE ]
  type Site = [ d :: Log  | d <- matches timeRE ]

  type Log = Directory
    { web is "coralwebsrv.log.gz" :: Gzip  (File Coral),
      dns is "coraldnssrv.log.gz" :: Maybe (Gzip (File Ptext)),
      prb is "probed.log.gz"      :: Maybe (Gzip (File Ptext))
      dmn is "corald.log.gz"      :: Maybe (Gzip (File Ptext)) }
|]
```

```
type Log_md = (Forest_md, Log_inner_md)
data Log_inner_md = Log_inner_md
  { web_md :: (Forest_md, Coral_md)
  , dns_md :: (Forest_md, Maybe (Forest_md, Ptext_md))
  , prb_md :: (Forest_md, Maybe (Forest_md, Ptext_md))
  , dmn_md :: (Forest_md, Maybe (Forest_md, Ptext_md))
  }
```

Base Forest MetaData

```
data Forest_md = Forest_md
    { numErrors :: Int
    , errorMsg  :: Maybe ErrMsg
    , fileInfo  :: FileInfo
    }

data FileInfo = FileInfo
    { fullpath    :: FilePath
    , owner       :: String
    , group       :: String
    , size        :: COff
    , access_time :: EpochTime
    , mod_time    :: EpochTime
    , read_time   :: EpochTime
    , mode        :: FileMode
    , isSymLink   :: Bool
    , kind        :: FileType
    }
```

Forest Type Class

- *Compiler generates instance declarations:*
  - `Data` *for each rep and md type*
  - `ForestMD` *for each md type*
  - `Forest` *for each pair of rep, md types*

```
class (Data rep, ForestMD md) =>
      Forest rep md | rep -> md  where
  load :: FilePath -> IO(rep, md)
  fdef :: rep

class Data md => ForestMD md where
  get_fmd_header :: md -> Forest_md
  replace_fmd_header :: md -> Forest_md -> md
  get_fileInfo :: md -> FileInfo
  get_fullpath :: md -> String
  ...
```

Programming with Forest

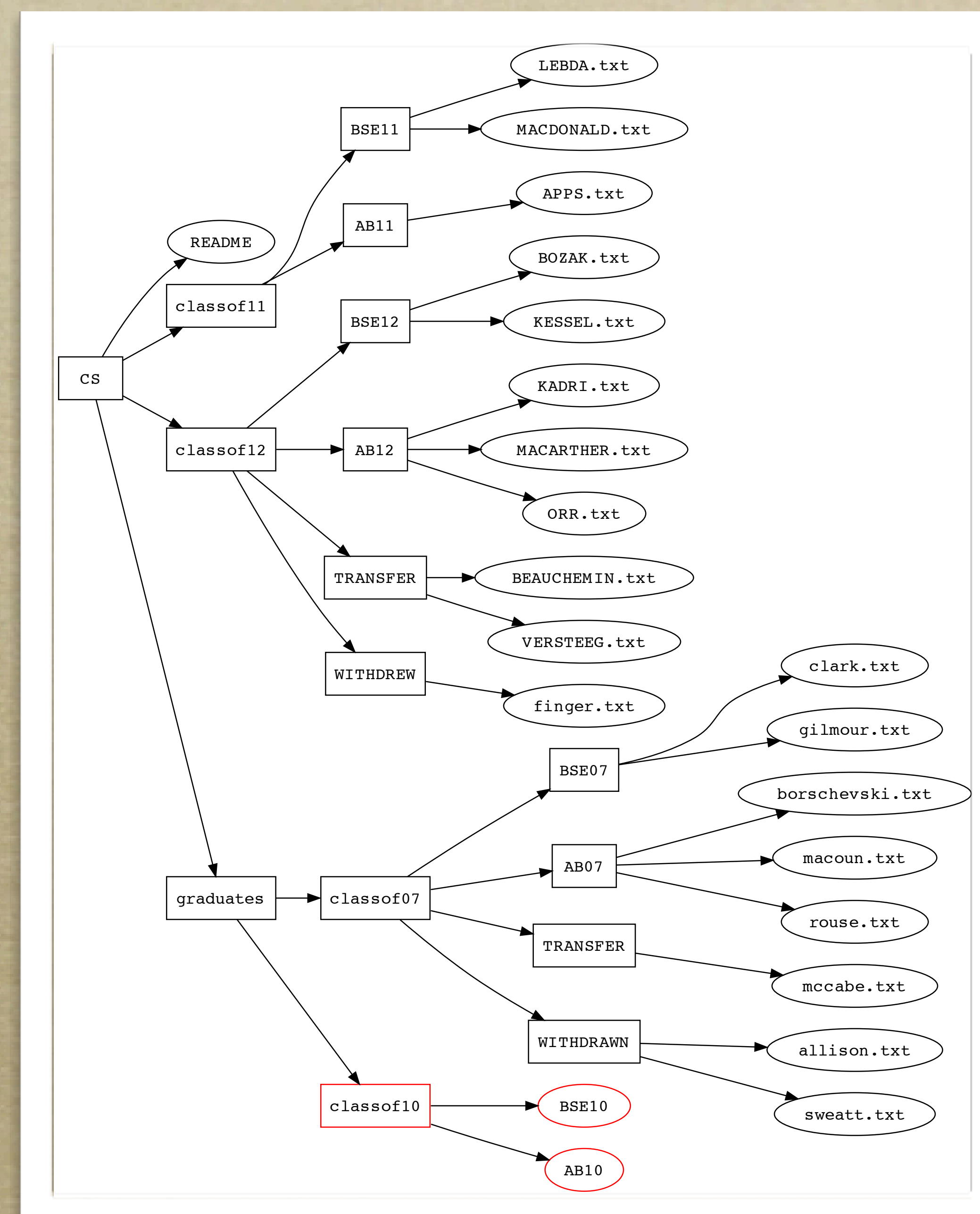
- *Forest/Pads combination blurs distinction between on-disk and in-memory data.*
- *Example program computes time when CDN statistics last reported for each site:*

```
sites_mod () = do
  (rep,md) <- coral_load "/var/log/coral1"
  return (case (rep,md) of (Coral rs, (_,ms)) ->
             map (get_site *** get_mod) (zip rs ms))

get_site = fst
get_mod (_,(f,_)) = mod_time . fileInfo $ f
```

Generic Programming

- *Third-party developers can use generic programming to build tools that work over any Forest description.*
  - *Pretty printer*
  - *File system visualization*
  - *Generic querying*
  - *Permission checker*
  - *Description-specific versions of shell tools*
  - *Description “inferencer”*



Example: Generic Querying

- *Function* `findFiles` *returns all file names in metadata* `md` *that satisfy predicate* `pred`*:*

```
findFiles :: (ForestMD md) =>
             md -> (FileInfo -> Bool) -> [FilePath]
findFiles md pred = map fullpath (listify pred md)
```

- *Example uses:*

```
dirs  = findFiles cs_md
       (\(r::FileInfo) -> (kind r) == DirectoryK)
other = findFiles cs_md
       (\(r::FileInfo) -> (owner r) /= "bwk")
```

Semantics

Strings	$n \quad \in \Sigma^*$
Paths	$r, s ::= \bullet \mid r \,/\, n$
Attributes	$\mathsf{a} ::= \ldots$
Filesystem Contents	$T ::= \mathsf{File}(n)$ $\mid \mathsf{Dir}(\{n_1, \ldots, n_k\})$ $\mid \mathsf{Link}(r)$
Filesystems	$F ::= \{\!\mid r_1 \mapsto (\mathsf{a}_1, T_1), \ldots r_k \mapsto (\mathsf{a}_k, T_k) \mid\!\}$
Values	$v ::= \mathsf{a} \mid n \mid r \mid \mathsf{True} \mid \mathsf{False} \mid () \mid (v_1, v_2)$ $\mid \mathsf{Just(v)} \mid \mathsf{Nothing} \mid \{v_1, \ldots, v_k\}$
Expressions	$e ::= x \mid v \mid \ldots$
Environments	$\mathcal{E} ::= \bullet \mid E, x \mapsto v$
Specifications	$s ::= k_{\tau_r}^{\tau_m} \mid \mathsf{Adhoc}(b_{\tau_r}^{\tau_m}) \mid e :: s \mid \langle x{:}s_1, s_2 \rangle$ $\mid \{s \mid x \in e\} \mid \mathsf{Pred}(e) \mid s?$

Semantics

$$\frac{\mathsf{eval}_{(\tau\ set)}(\mathcal{E}, F, r, e) = \{v_1, \ldots, v_k\} \quad S = \{(v, d) \mid v' \in \{v_1, \ldots, v_k\} \text{ and } \mathcal{E}[x \mapsto v']; F; r \models s \rightsquigarrow v, d\}}{\mathcal{E}; F; r \models \{s \mid x \in e\} \rightsquigarrow \pi_1\ S, (\bigwedge\ valid(\pi_2\ S), \pi_2\ S)}$$

$$\frac{}{\mathcal{E}; F; r \models \mathsf{Pred}(e) \rightsquigarrow (), (\mathsf{eval}_{bool}(E, F, r, e), ())}$$

$$\frac{r \notin \mathsf{dom}(F)}{\mathcal{E}; F; r \models s? \rightsquigarrow \mathsf{Nothing}, (\mathsf{False}, \mathsf{Nothing})}$$

$$\frac{r \in \mathsf{dom}(F) \qquad \mathcal{E}; F; r \models s \rightsquigarrow v, d}{\mathcal{E}; F; r \models s? \rightsquigarrow \mathsf{Just(v)}, (valid(d), \mathsf{Just(d)})}$$

$$\frac{}{\mathcal{E}; F; r \models k_{\tau_r}^{\tau_m} \rightsquigarrow ck(k_{\tau_r}^{\tau_m}, F, r)}$$

$$\frac{F(r) = (\mathsf{a}, \mathsf{File}(n)) \qquad b_{\tau_r}^{\tau_m}(E, n) = v, d}{\mathcal{E}; F; r \models \mathsf{Adhoc}(b_{\tau_r}^{\tau_m}) \rightsquigarrow v, (valid(d), (d, \mathsf{a}))}$$

$$\frac{F(r) = (\mathsf{a}, T) \qquad T \neq \mathsf{File}(n) \qquad b_{\tau_r}^{\tau_m}(E, \epsilon) = (v, d)}{\mathcal{E}; F; r \models \mathsf{Adhoc}(b_{\tau_r}^{\tau_m}) \rightsquigarrow v, (\mathsf{False}, (d, \mathsf{a}))}$$

$$\frac{r \notin \mathsf{dom}(F) \qquad b(E, \epsilon) = (v, d)}{\mathcal{E}; F; r \models \mathsf{Adhoc}(b_{\tau_r}^{\tau_m}) \rightsquigarrow v, (\mathsf{False}, (d, \mathsf{a}_{\mathsf{default}}))}$$

$$\frac{\mathcal{E}; F; \mathsf{eval}_{path}(E, F, r, r\,/\,e) \models s \rightsquigarrow v, d}{\mathcal{E}; F; r \models e :: s \rightsquigarrow v, d}$$

$$\frac{\mathcal{E}; F; r \models s_1 \rightsquigarrow v_1, d_1 \quad \mathcal{E}[x \mapsto v_1, x_d \mapsto d_1]; F; r \models s_2 \rightsquigarrow v_2, d_2}{\mathcal{E}; F; r \models \langle x{:}s_1, s_2\rangle \rightsquigarrow (v_1, v_2), (valid(d_1) \wedge valid(d_2), (d_1, d_2))}$$

Implementation

- *Requires >= GHC 7.0*
- *Quasi-quoter and Template Haskell*

```
QuasiQuoter
  quoteExp  :: String -> Q Exp
  quotePat  :: String -> Q Pat
  quoteType :: String -> Q Type
  quoteDec  :: String -> Q [Dec]
```

- *Generic programming: so far, SYB.*
- *Available for download from* www.padsproj.org

Current Work: Printing

Read-only FileStores are already very useful, but the ability to print would be nice.

Pads/Haskell already supports printing.

Challenges:

- *What if same file is mentioned multiple times?*
- *How to handle large FileStores?*