9.7 Webprogramming in Haskell

- WASH: a Haskell library for server-side Web scripting
- Based on CGI (portability, ease of use)
- High-level functionality
  - X(HT)ML generation & syntax
  - Control flow in program = interaction
  - Typed interfaces, checked by compiler
  - No string-based interfaces necessary

9.7.1 Essence of Haskell

Haskell is a “purely functional programming language”

- higher-order functions
- automatic garbage collection
- separation between side-effect free evaluation and stateful computation
- lazy evaluation
- strong, static, and polymorphic type system
- Haskell’s 15th birthday in 2003:
  `http://research.microsoft.com/~simonpj/papers/haskell-retrospective/index.htm`

9.7.2 WASH Example: showDate

```
import CGI
import Time

main :: IO ()
main = run date

date :: CGI ()
date = do theDate <- io $ do clk <- getClockTime
                             cal <- toCalendarTime clk
                             return (calendarTimeToString cal)
          ask <html>
                <head><title>The current time</title></head>
                <body>
                  <h1>The current time</h1>
                  <%= theDate %>
                </body>
              </html>
```

Explanation

`main :: IO ()` the program's entry point

`CGI` WASH's action monad; handles sequencing of I/O actions on the server and between browser and server

`run` starts the CGI monad; first thing in a WASH program

```
run :: CGI () -> IO ()
```

`io` embeds an IO action into the CGI monad

```
io :: (Read a, Show a) => IO a -> CGI a
```

`ask` maps a document to a CGI action

```
ask :: WithHTML CGI () -> CGI ()
```

XHTML literals generate document fragments

9.7.3 Abstraction for X(HT)ML

- Webpages-as-text is not appropriate
  - phase errors (headers, main message)
  - structural errors (well-formedness, validity)
  - requires too much low-level knowledge
- WASH/CGI's approach
  - XML fragments represented by tree structure
  - constructed functionally
  - automatic conversion to XML-syntax (serialization) on output

XHTML Literals

- `WithHTML CGI a` type of XHTML literal
- *sequence* of document nodes (elements, attributes, or text nodes)
- corresponds to *contents* of a HTML element
- each XHTML tag or attribute creates a *singleton sequence*
- also computes a value of type `a` (*later*)
- syntax inspired by Java Server Pages

Syntax of XHTML Literals

an element literal:	`<title>MyTitle</title>`
an attribute literal:	`<[name="value"]>`
a string insertion:	`<%= aString %>`
	where `aString :: String`
a document insertion:	`<% aDoc %>`
	where `doc :: WithHTML m a`
	also in attribute context
attribute value:	`name=<% aString %>`
	where `aString :: String`
sequence of elements:	`<#>`*contents of XHTML element*`</#>`

Differences to JSP-style

JSP	WASH
starts in XML mode	starts in program mode
translation is oblivious to program syntax	XML elements become language expressions
scriptlets cannot be nested	arbitrary nesting of scriptlets and XML
expression language required to substitute in attributes	notation for attributes and attribute values
literal XML elements cannot be processed by program code	XML elements are first-class: they can be passed as parameters, stored in data structures, and returned from functions

9.7.4 Document Abstraction

- Documents are just monadic values

⇒ parameterized documents by value abstraction

- Example: a standard XHTML document template

```
standardPage :: String -> WithHTML x CGI a -> CGI ()
standardPage title contents =
  ask   <html>
          <head><title><%= title %></title></head>
          <body>
            <h1><%= title %></h1>
            <% contents %>
          </body>
        </html>
```

Constructing Documents by Hand

- `text :: String -> WithHTML x CGI ()`

  creates a *singleton sequence* with one text node

- for each HTML tag t, there is a constructor function

  `t :: WithHTML x CGI a -> WithHTML y CGI a`

  - it takes a sequence of child elements and attributes
  - creates an element with tag t
  - returns it in a *singleton sequence*

- Example: `p (text "This is my first CGI program!")`

Document Node Sequences

- the empty sequence

```
empty
```

- concatenation of sequences

```
seq1 ## seq2
```

or

```
seq1 >> seq2
```

or

```
do { seq1; seq2; ...; seqn }
```

or

```
  do    seq1
        seq2
        ...
        seqn
```

Example

```
standardQuery "Hello" $
do p (text "This is my second CGI program!")
   p (do text "My hobbies are"
         ul (do li (text "swimming")
                li (text "music")
                li (text "skiing")))
```

Example: showDate with raw constructors

```
date :: CGI ()
date = do theDate <- io $ do clk <- getClockTime
                             cal <- toCalendarTime clk
                             return (calendarTimeToString cal)
          ask (html (do
                head (title (text "The current time"))
                body (do
                  h1 (text "The current time")
                  text theDate)))
```

9.7.5 Working with Widgets

For programming interactive web pages, we need to specify

- an XHTML form
  to tell the browser that the web page accepts input and where this input should be delivered
- several input fields (widgets)
  each widget specifies a particular input mode
- an action taken on input

Creating a Form

- “raw” constructor for `form` element not available
- instead “cooked” constructor

  ```
  makeForm :: WithHTML CGI a -> WithHTML CGI ()
  ```

  creates form with standard attributes preset
- the WASH library provides the following parameterized document:

  ```
  standardQuery :: String -> WithHTML CGI a -> CGI ()
  standardQuery title xmlElems =
    ask (standardPage title (makeForm xmlElems))
  ```

Example: Adding two numbers (old style)

```
adder :: CGI ()
adder = standardQuery "Adder/1"
    <#> <p>First  number to add <% sum1F <- inputField empty %></p>
        <p>Second number to add <% sum2F <- inputField empty %></p>
        <% submit (F2 sum1F sum2F) addThem <[value="Perform addition"]> %>
    </#>

addThem (F2 sum1F sum2F) =
  let sum1, sum2 :: Int
      sum1 = value sum1F
      sum2 = value sum2F
  in
  standardQuery "Adder/2"
    <#> <p><%= show sum1 %> + <%= show sum2 %> = <%= show (sum1+sum2) %></p>
        <% submit0 adder <[value="Continue"]> %>
    </#>
```

Example: Adding two numbers (new style)

```
adder :: CGI ()
adder = standardQuery "Adder/1"
    <#> <p>First  number to add <input type="text" name="sum1"/></p>
        <p>Second number to add <input type="text" name="sum2"/></p>
        <input type="submit" value="Perform addition"
               WASH:callback="addThem" WASH:parms="sum1,sum2"/>
    </#>

addThem :: (Int, Int) -> CGI ()
addThem (sum1, sum2) =
  standardQuery "Adder/2"
    <#> <p><%= sum1 %> + <%= sum2 %> = <%= sum1+sum2 %></p>
        <input type="submit" value="Continue"
               WASH:callback="adder"/>
    </#>
```

Example: GuessNumber

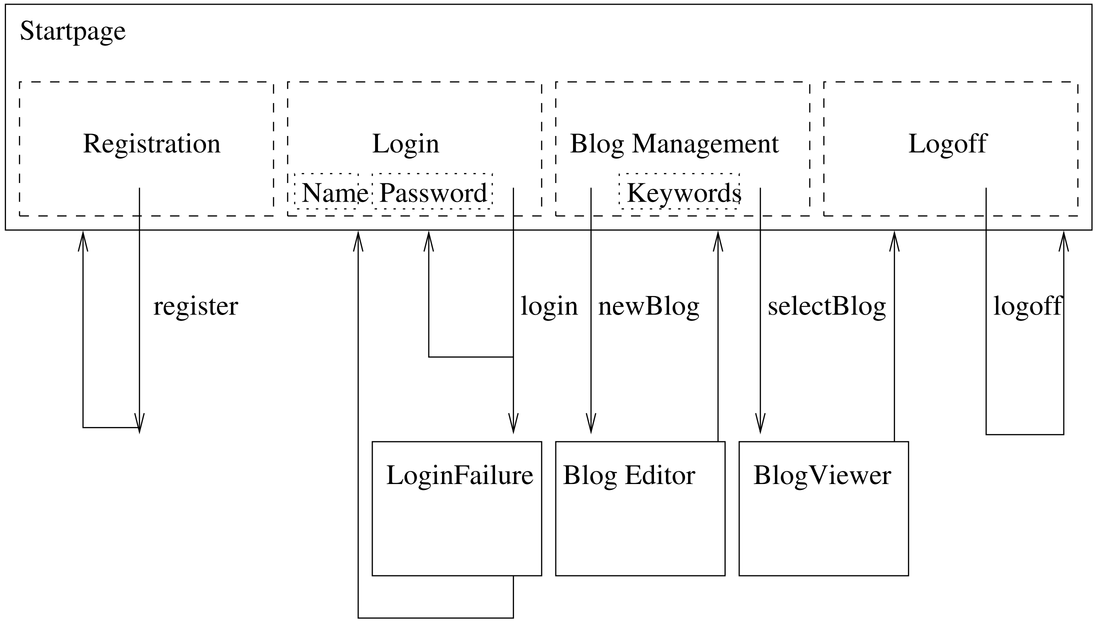
```
highScoreStore :: CGI (Persistent2.T [Score])
highScoreStore = Persistent2.init "GuessNumber" []

main :: IO ()
main =
  run mainCGI

mainCGI =
  io (randomRIO (1,100)) >>= \ aNumber ->
  standardQuery "Guess a number"
    let go = play 0 (aNumber :: Int) "Guess a number between 1 and 100
    <#><input type="submit" value="Play game" WASH:callback="go"/>
       <input type="submit" value="Hi scores"  WASH:callback="admin"/>
    </#>
```

```
play nGuesses aNumber aMessage =
  standardQuery "Guess a number"
    <#><% aMessage %> Make a guess
       <input type="text" name="aGuess"/>
       <input type="submit"
              WASH:callback="processGuess (nGuesses + 1) aNumber"
              WASH:parms="aGuess"/>
    </#>

processGuess nGuesses aNumber aGuess =
  if aNumber == aGuess then
    youGotIt nGuesses aNumber
  else if aGuess < aNumber then
    play nGuesses aNumber (show aGuess ++ " was too small.")
  else
    play nGuesses aNumber (show aGuess ++ " was too large.")
```

```
youGotIt nGuesses aNumber =
  standardQuery "You got it!"
  <#>CONGRATULATIONS!<br/>
     It took you <%= nGuesses %> tries to find out.<br/>
     Enter your name for the hall of fame
     <input type="text" name="name"/><br/>
     <input type="submit" value="ENTER"
            WASH:callback="addToHighScore nGuesses"
            WASH:parms="name"/>
  </#>

addToHighScore nGuesses name =
  if name == "" then admin else
  do highScoreList <- highScoreStore
     Persistent2.add highScoreList (Score name nGuesses)
     admin
```

```
admin =
  do highScoreList <- highScoreStore
     highScores <- Persistent2.get highScoreList
     standardQuery "GuessNumber - High Scores"
       <table border="border">
         <tr><th>Name</th><th># Guesses</th></tr>
         <% mapM_ oneEntry (sort highScores) %>
       </table>
  where
    oneEntry (Score name guesses) =
      <tr><td><% name %></td><td><% guesses %></td></tr>
```

9.7.6 A Blogger Application



- structured in pagelets and wiring
- each pagelet composed of logic and skin

Login Pagelet: Specification

```
type Skin = ...
type Name = String
type Password  = String
type PasswordChecker = Name -> Password -> IO Bool
type SuccessCont = Name -> CGI ()
type FailureCont = CGI ()
login :: Skin
      -> PasswordChecker
      -> SuccessCont
      -> FailureCont
      -> WithHTML x CGI ()
```

Login Pagelet: Skin

```
module LoginSkin where
import CGI
-- visual layout, only
loginSkin act =
  <table>
    <tr><td>Name</td>
        <td><input type="text" name="l"/></td></tr>
    <tr><td>Password</td>
        <td><input type="password" name="p"/></td></tr>
    <tr><td></td>
        <td><input type="submit" value="Login"
                   WASH:parms="l,p" WASH:callback="act"/>
        </td></tr>
  </table>
```

Login Pagelet: Logic

```
module Login where
import CGI

login skin pwCheck succCont failCont =
  skin $ \ (F2 l p) ->
    let logname = unNonEmpty (value l)
        pw      = unPassword (value p)
    in
    do registered <- io (pwCheck logname pw)
       if registered
          then succCont logname
          else failCont
```

Final Wiring: Composing the Pagelets

```
-- build pagelets from logic and skin
startPage= StartPage.startPage Skins.startSkin
login    = Login.login Skins.loginSkin
logoff   = Logoff.logoff Skins.logoffSkin
register = Register.register Skins.registerSkin
selector = Select.selector Skins.selectorSkin
```

Final Wiring: Only Control Logic

```
blogger =
  mainPage initialBloggerState ""

mainPage bs message =
  ask (startPage message (userManager bs) (blogManager bs))

userManager bs =
  case n bs of
    Nothing ->
      Skins.userManager1
         (login myPasswordCheck
                (\ user -> mainPage bs{ n = Just user } "Login successful")
                (mainPage bs{ n = Nothing } "Login failed"))
         (register myPasswordSaver
                (\ user -> mainPage bs{ n = Just user } "Registration successful"))
    Just user ->
      Skins.userManager2
         (logoff user (mainPage initialBloggerState (user ++ " logged off")))

blogManager bs@B{ st = Visiting } =
  selector myBlogTitles (BlogAccess.newBlog bs mainPage) (BlogAccess.oldBlog bs mainPage)
```

9.7.7 Conclusion

- simple, declarative approach to Web-based user interfaces
- types and type safety essential
- GUI-style programming interface
- natural interface to HTML
- ideas not tied to CGI
- applications: submission software, generic time table, ...
- available from `http://www.informatik.uni-freiburg.de/~thiemann/WASH`