# 0

```
import Text.ParserCombinators.Parsec
import Data.List
```

# 1

A HWEB file is composed of a *prologue* and a sequence of *sections*.

```
type Web = (Prologue, [Section])
```

# 2

The HWEB file begins with the prologue, which may be empty. It is ignored by HTANGLE and copied essentially verbatim by HWEAVE, so its function is to provide any additional formatting instructions that may be desired by the HTML output. Indeed it is customary to begin a HWEB file with a prologue that imports or defines scripts and styles.

```
type Prologue = Text
```

# 3

Each section has two parts:

- A HTML part, explaining what is happening in the section.
- A Haskell part, containing a piece of code to be output by HTANGLE. This piece should be short enough that a reader can readily perceive its structure and easily understand it as a unit.

The two halves of a section must appear in this order; i.e. the HTML commentary must come before the Haskell code. Either part may be empty.

```
type Section = (Title, Commentary, SectionInfo, Code)
type Title = Text
type Commentary = Text
type SectionInfo = (Name, IsFile)
type Name = Maybe Text
```

```
type IsFile = Bool
type Code = [Either Text Link]
type Text = String
```

# 4

The construct " @ < section name@ > " can appear any number of times in the code part of a section: Subsequent appearances indicate that a named section is being linked to rather than defined.

```
type Link = Text
```

# 5

```
web :: Parser Web
web = do
    prol ← prologue
    secs ← many1 section
    return (prol, secs)


prologue :: Parser Prologue
prologue = do
    txt ← manyTill anyChar (try (string "@*"))
    return txt
```

# 6

The *control code* @* denotes the beginning of a new section. The title of the new group should appear after the @* followed by a period.

The Haskell part of an unnamed section begins with @h . This causes HTANGLE to append the code to the first order program text.

The control code @ < introduces a section name which consists of HTML and extends to the matching @ > . The whole construct is conceptually a Haskell element. The behaviour differs depending on the context: A @ < appearing in the commentary part attaches the following section name to the current section. The closing @ > should be followed by "=".

The control code @( introduces a special kind of section: a *file* section. The code in such a section will be output to a separate file, the name of which is found between the @( and the @> .

```
section :: Parser Section
section = do
    title ← manyTill anyChar (try (char '.'))
    txt ← manyTill anyChar (try (char '@'))
    info ← section_info
    cod ← code
    return (title, txt, info, cod)


section_info :: Parser SectionInfo
section_info = do
    c ← anyChar
    case c of
        '<' → do {name ← manyTill anyChar (try (string "@>="));   return (Just name, False)}
        '(' → do {name ← manyTill anyChar (try (string "@>="));   return (Just name, True)}
        'h' → return (Nothing, False)
```

# 7

In the code part, @< indicates that a named section is being used—its Haskell definition is splicd in by HTANGLE.

```
code :: Parser Code
code = do
    txt ← manyTill anyChar (try (char '@'))
    c ← anyChar
    case c of
        '*' → return [Left txt]
        '<' → do
            link ← manyTill anyChar (try (string "@>"))
            rest ← code
            return ((Left txt) : (Right link) : rest)
```

# 8

```
pretty :: [(Int, Section)] → String
pretty [] = ""
pretty (s : ss) = pretty_s s ++ pretty ss
```

```
pretty_s :: (Int, Section) → String
pretty_s (n, (title, commentary, info, code)) =
    let name =
        case info of
            (Nothing, _) → ""
            (Just txt, _) → txt
    in "<h2>" ++ show n ++ ". " ++ title ++ "</h2>" ++
        "<p id=\"" ++ name ++ "\">" ++ commentary ++ "</p>" ++
        (if name ≠ ""
    then "<h3>" ++ "&lt;" ++ name ++ "&gt; ::=" ++ "</h3>"
    else "") ++
        "<pre>" ++ prettify_code code ++ "</pre>"

prettify_code :: Code → String
prettify_code [] = ""
prettify_code ((Left txt) : code) = clean txt ++ prettify_code code
prettify_code (Right link : code) = "<a href=\"#" ++ link ++ "\">" ++ link ++ "</a>" ++ prettify_cod

clean [] = []
clean ('<' : xs) = "&lt;" ++ clean xs
clean ('>' : xs) = "&gt;" ++ clean xs
clean (x : xs) = x    :    clean xs
```

# 9

```
weave :: IO String
weave = do
    result ← parseFromFile web "test0.hweb"
    case result of
        Right (limbo, sections) → return (limbo ++ pretty (zip [0..] sections))

hweave :: FilePath → FilePath → IO ()
hweave in_file out_file = do
    result ← parseFromFile web in_file
    case result of
        Right (limbo, sections) → do
            writeFile out_file $
                "<head>" ++ limbo ++ "</head>" ++
                "<body><div id=\"content\">" ++ pretty (zip [0..] sections) ++ "</div></body
        Left err → print err

hweave_ :: FilePath → IO ()
hweave_ in_file = hweave in_file $ takeWhile (≠ '.') in_file    ++    ".shtml"
```

```
htangle_ :: FilePath → String → IO ()
htangle_ in_file ext = htangle in_file $ takeWhile (≢ '.') in_file ++ ext
```

# 10

The main idea of HTANGLE is to make a compiler-ready Haskell program out of individual sections, named and unnamed. This is done as follows:

1. The Haskell parts of unnamed sections are collected, in order; this constitutes the first-order approximation to the compiler-ready code. (There should be at least one unnamed section, otherwise there will be no program.)
2. An association list of section names and corresponding code is built. If the same name has been given to more than one section, the Haskell text for that name is obtained by putting together all the Haskell parts in the corresponding sections. This feature is useful, for example, in a section named "Module imports", since one can then import a module in whatever section its exports are used.
3. All section names that appear in the first-order approximation are replaced by the Haskell parts of the corresponding sections, and this substitution process continues until no section names remain.

```
unnamed_code :: [Section] → Code
unnamed_code ss = concat [cod | (_, _, (name, is_file), cod) ← ss, name ≡ Nothing]

named_code :: [Section] → [(Name, Code)]
named_code ss = (compress ∘ sort) [(name, cod) | (_, _, (name, is_file), cod) ← ss, name ≢ Nothing,

file_sections :: [Section] → [(Name, Code)]
file_sections ss = [(name, cod) | (_, _, (name, is_file), cod) ← ss, name ≢ Nothing, is_file ≡ True]

compress [(x, y)] = [(x, y)]
compress ((x, ys) : (x′, ys′) : zs) =
    if x ≡ x′
    then compress ((x, ys ++ ys′) : zs)
    else (x, ys) : compress ((x′, ys′) : zs)

tangle :: [(Name, Code)] → Code → Code
tangle db xs =
```

```
    let xs' = (concat ∘ map (visit db)) xs
    in if no_links xs' then xs' else tangle db xs'


visit db (Left txt) = [Left txt]
visit db (Right link) = case lookup (Just link) db of
    Nothing → error $ "Section named " ++ link ++ " not defined!"
    Just cod → cod


p (Left txt) = True
p (Right link) = False


no_links ys = all p ys


htangle :: FilePath → FilePath → IO ()
htangle input_file output_file = do
    result ← parseFromFile web input_file
    case result of
        Right (limbo, sections) → code_out output_file sections
        Left err → print err


code_out :: FilePath → [Section] → IO ()
code_out output_file sections = do
    let cod = tangle (named_code sections) (unnamed_code sections)
    let (names, codes) = unzip (file_sections sections)
    let codes' = map (tangle (named_code sections)) codes
    writeFile output_file ((concat ∘ map (λ(Left txt) → txt)) cod)
    output (zip names codes')


output :: [(Name, Code)] → IO ()
output [] = do {putStrLn "Output: Done"; return ()}
output ((Just name, cod) : xs) = do
    writeFile name $ (concat ∘ (map (λ(Left txt) → txt))) cod
    putStrLn $ "Output: " ++ name
    output xs


test0 = htangle "test0.hweb" "test0.hs"


main = test0
```