First Exam
CS 3366 Programming Languages

KEY
Thursday February 19, 2015
Instructor Muller
Boston College
Spring 2015

Before reading further, please arrange to have an empty seat on either side of you. Now that you are seated, please write your name **on the back** of this exam.

This is a closed-notes and closed-book exam. Computers, calculators, and books are prohibited.

- Partial credit will be given so be sure to show your work.
- Feel free to write helper functions if you need them.
- **Please write neatly.**

Problem	Points	Out Of
1		6
2		8
Total		14

A click on a link in a web browser leads to all manner of network to-ing and fro-ing and, if everything works out, eventually to the delivery of an HTML document to your web browser. As you probably already know, HTML is a text-based structured document layout language. So of course the front end of your browser includes a parser for HTML.

HTML is famously convoluted so a proper context-free grammar for it runs to hundreds of lines. (Google it sometime!) This problem chips off a very small part. In particular, we'll deal with documents that have only bodys and tables. We'll call it mini-HTML. Here's an example:

```
<HTML>
 <BODY>
  <TABLE>
   <TR>
    <TD>A</TD> <TD>B</TD>
   </TR>
   <TR>
    <TD>C</TD> <TD>D</TD>
   </TR>
  </TABLE>
 </BODY>
</HTML>
```

As you might know, the items above in the angle brackets are called *tags*. Each part of an HTML document is supposed to have an opening tag paired up with a closing tag with a backslash prefix. In the simple example, we can see that the content of a mini-HTML document is enclosed between opening and closing HTML tags. Normally there is a HEADER but in mini-HTML there is just a BODY containing a list of zero or more TABLEs. Each table contains a list of zero or more rows (TR) (that your browser lays out vertically) and each row has zero or more table data (TD) items (that your browser lays out horizontally). Each table data item can contain some text.

1. (6 Points) Assuming that you are given the definition of a nonterminal `Text`, write a context free grammar for mini-HTML as described above. Your grammar should have a start symbol `Html` as shown below and non-terminals for the body, for tables, for table rows and table data. Note that the opening and closing tags make it easy to define a non left-recursive grammar.

```
Text   ::= ...
Html   ::= <HTML> Body </HTML>
```

Answer:

```
Text   ::= ...
Html   ::= <HTML> Body </HTML>
Body   ::= <BODY> Tables </BODY>
Tables ::= <TABLE> TRows </TABLE> Tables | empty
TRows  ::= <TR> TData </TR> TRows | empty
TData  ::= <TD> Text </TD> TData | empty
```

2. (8 Points) Using `HTMLO` for “html open” and `HTMLC` for “html close” (and likewise for the others), consider the following F# type for tokens for mini-HTML documents:

```
type tokens = HTMLO | HTMLC | BODYO | BODYC | TABLEO | TABLEC
            | TRO | TRC | TDO | TDC | Text of string
```

For example, the mini-HTML document above could be represented in F# by the list of tokens:

```
[HTMLO; BODYO; TABLEO; TRO; TDO; Text("A"); TDC; TDO; Text("B"); TDC; ...]
```

Now consider the following type for abstract syntax trees for mini-HTML documents:

```
type td = Td of string
type tr = Tr of list<td>
type table = Table of list<tr>
type body = Body of list<table>
type html = Html of body
```

Using your grammar as a guide, write a recursive descent parser in F# for min-HTML documents. Your parser should have type: `parse : list<tokens> -> html`.

Answer:

```
let parse tokens =
  let rec html tokens =
    match tokens with
    | HTMLO::tokens -> let (ast, tokens) = body tokens
                       match tokens with
                       | HTMLC::tokens -> (Html ast, tokens)
                       | _ -> failwith "missing html close"
    | _ -> failwith "missing html open"

  and body tokens =
    match tokens with
    | BODYO::tokens -> let (ts, tokens) = tables tokens
                       match tokens with
                       | BODYC::tokens -> (Body ts, tokens)
                       | _ -> failwith "missing body close"
    | _ -> failwith "missing body open"

  and tables tokens =
    match tokens with
    | TABLEO::tokens -> let (tab, tokens) = table (TABLEO::tokens)
                        let (tabs, tokens) = tables tokens
                        (tab::tabs, tokens)
    | _ -> ([], tokens)

  and table tokens =
    match tokens with
    | TABLEO::tokens -> let (rs, tokens) = rows tokens
                        match tokens with
                        | TABLEC::tokens -> (Table rs, tokens)
                        | _ -> failwith "missing table close"
    | _ -> failwith "missing table open cannot happen"
```

```
and rows tokens =
  match tokens with
  | TRO::tokens -> let (r,  tokens) = row (TRO::tokens)
                   let (rs, tokens) = rows tokens
                   (r::rs, tokens)
  | _ -> ([], tokens)

and row tokens =
  match tokens with
  | TRO::tokens -> let (ds, tokens) = data tokens
                   match tokens with
                   | TRC::tokens -> (Tr ds, tokens)
                   | _ -> failwith "missing table row close"
  | _ -> failwith "missing table row open"

and data tokens =
  match tokens with
  | TDO::tokens -> let (d, tokens) = datum (TDO::tokens)
                   let (ds, tokens) = data tokens
                   (d::ds, tokens)
  | _ -> ([], tokens)

and datum tokens =
  match tokens with
  | TDO::Text(s)::TDC::tokens -> (Td s, tokens)
  | _ -> failwith "bad table data item"

match html tokens with
| (ast, []) -> ast
| _ -> failwith "bad html"
```