CMSC 330 Exam #2 solutions Fall 2012

1. [16 pts.] The grammar in this problem has two aspects that didn't come up in the examples of left factoring shown in class. Firstly, the productions where left factoring must be performed are recursive. However, The left factoring procedure that was shown works in this situation, but it's just a little different. Applying the procedure would result in this grammar (we added new nonterminals L and M):

   $\mathrm{S} \rightarrow c\mathrm{L} \mid db\mathrm{T} \mid bd\mathrm{T} \mid \mathrm{T}$
   $\mathrm{L} \rightarrow a\mathrm{S} \mid o\mathrm{S}$
   $\mathrm{T} \rightarrow a\mathrm{M} \mid b$
   $\mathrm{M} \rightarrow a\mathrm{T} \mid b\mathrm{T} \mid \epsilon$

   The second aspect that comes up more indirectly here than in the examples from class is that the grammar above still doesn't have unique first sets for its productions. b is in first(T), as well as in first(bd), so the S production has b in the first of two of its right sides; consequently the procedure would have to be applied again. Almost no one realized this, so we just gave full credit for applying the procedure once, resulting in the grammar above. But a correct grammar produced by left factoring again would be (now adding the new nonterminal N):

   $\mathrm{S} \rightarrow c\mathrm{L} \mid db\mathrm{T} \mid b\mathrm{N}$
   $\mathrm{L} \rightarrow a\mathrm{S} \mid o\mathrm{S}$
   $\mathrm{T} \rightarrow a\mathrm{M} \mid b$
   $\mathrm{M} \rightarrow a\mathrm{T} \mid b\mathrm{T} \mid \epsilon$
   $\mathrm{N} \rightarrow d\mathrm{T} \mid \epsilon$

   Going back to the first grammar, note that just using the same nonterminal in the right sides of the new productions added, rather than the original nonterminal, would lead to productions that could never generate anything:

   $\mathrm{S} \rightarrow c\mathrm{L} \mid db\mathrm{T} \mid bd\mathrm{T} \mid \mathrm{T}$
   $\mathrm{L} \rightarrow a\mathrm{L} \mid o\mathrm{L}$
   $\mathrm{T} \rightarrow a\mathrm{M} \mid b$
   $\mathrm{M} \rightarrow a\mathrm{M} \mid b\mathrm{M} \mid \epsilon$

   Any derivation that uses the nonterminal L will never terminate and derive a string.

   And note that just making the first added production recursive and adding an alternative for ϵ, as in the grammar below, will allow generating invalid strings such as c, aaa, aoa, etc.

   $\mathrm{S} \rightarrow c\mathrm{L} \mid db\mathrm{T} \mid bd\mathrm{T} \mid \mathrm{T}$
   $\mathrm{L} \rightarrow a\mathrm{L} \mid o\mathrm{L} \mid \epsilon$
   $\mathrm{T} \rightarrow a\mathrm{M} \mid b$
   $\mathrm{M} \rightarrow a\mathrm{T} \mid b\mathrm{T} \mid \epsilon$

2. [20 pts.]

   a. This grammar can't generate some valid strings: those without any *b*s and *c*s, such as *d*, *add*, etc., or any strings that have an unequal number of *b*s and *c*s, such as *abbbd*. It's also ambiguous for any strings with more than one *b* and *c*, such as *bbccd*, *bbbcccddd*, *abbccdd*, etc.

   b. This grammar generates invalid strings that have *d*s before *b*s, such as *adbbbd* and *aadbbbdd*. It also can't generate some valid strings: those without any *b*s and *c*s, such as *d*, *add*, etc. Likewise *abcdd* and *acccdd* can't be generated.

   c. This grammar is ambigous for any strings other than *d*, *bc*, *bbcc*, and any strings that have *b*s but no *c*s, or *c*s but no *b*s (such as *abbbdd*, *aacccccccddd*, *bbbd*, etc.).

   d. Correct!

3. [32 pts.] Where the question talks about the grammar giving the interpretation that pipelines are performed or grouped in left–to–right order this means assocativity. Writing a **left–recursive** production for generating pipelines will force the first pipeline in a sequence to be the lowest one in a parse tree, the next one to be applied to the result of the first one, etc. (Note that nothing in the problem says that the grammar has to be suitable for parsing by a recursive descent parser, which would be the only case where a left recursive production would be a problem, and the requirement that pipelines be performed in left to right order necessitates a left–recursive production.)

   Where the question talks about input redirection having priority over pipelining, and pipelining having priority over output redirection, this means precedence, i.e., a hierarchy of productions must be introduced.

   We use nonterminals F for filenames and C for commands, and we made the vertical bar separating right–hand sides of productions really big so as to differentiate it from the pipeline terminal symbol.

$$
\begin{aligned}
\mathrm{S} &\rightarrow \mathrm{T} \;\Big|\; \mathrm{T}\ \texttt{>}\ \mathrm{F}\\
\mathrm{T} &\rightarrow \mathrm{T}\ \texttt{|\&}\ \mathrm{C} \;\Big|\; \mathrm{U}\\
\mathrm{U} &\rightarrow \mathrm{C}\ \texttt{<}\ \mathrm{F} \;\Big|\; \mathrm{C}\\
\mathrm{F} &\rightarrow \texttt{f} \;\Big|\; \texttt{g} \;\Big|\; \texttt{h}\\
\mathrm{C} &\rightarrow \texttt{a} \;\Big|\; \texttt{b} \;\Big|\; \texttt{c}
\end{aligned}
$$

   Note this version of the grammar (which differs only in the first alternative for T) is close to being correct, but it would allow generating invalid commands in which programs in the middle of a pipeline can have input or output redirection, such as the example `a |& b < f |& c`.

$$
\begin{aligned}
\mathrm{S} &\rightarrow \mathrm{T} \;\Big|\; \mathrm{T}\ \texttt{>}\ \mathrm{F}\\
\mathrm{T} &\rightarrow \mathrm{T}\ \texttt{|\&}\ \mathrm{U} \;\Big|\; \mathrm{U}\\
\mathrm{U} &\rightarrow \mathrm{C}\ \texttt{<}\ \mathrm{F} \;\Big|\; \mathrm{C}\\
\mathrm{F} &\rightarrow \texttt{f} \;\Big|\; \texttt{g} \;\Big|\; \texttt{h}\\
\mathrm{C} &\rightarrow \texttt{a} \;\Big|\; \texttt{b} \;\Big|\; \texttt{c}
\end{aligned}
$$

Also note that using recursion for the productions generating input or output redirection, as in the productions for S and U in the grammar below, would allow generating invalid commands in which a program has redirection from multiple input or output files, such as `a < f < g`.

$$
\begin{aligned}
S &\rightarrow T \mid S \texttt{ > } F \\
T &\rightarrow T \texttt{ |\& } C \mid U \\
U &\rightarrow U \texttt{ < } F \mid C \\
F &\rightarrow \texttt{f} \mid \texttt{g} \mid \texttt{h} \\
C &\rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{c}
\end{aligned}
$$

4. [32 pts.] Perhaps the most elegant way to solve this problem is using a higher–order functional programming paradigm not discussed in class yet this semester, so you would probably not know about it, but we explain it here. It is a higher–order function named filter, which takes a predicate (a boolean–valued function) and a list, and applies the function to every element of the list, returning a list with only the elements of its parameter list for which the predicate was true. We can create one function that will return true for elements of quicksort's list argument that are less than the pivot, and one that will return true for the elements of its argument list that are greater than or equal to the pivot, and apply them both to the list to partition the list.

   In order to do this we use the fact that OCaml's built–in operators can be passed as arguments to functions by surrounding them in parentheses. Or we could pass anonymous functions like `(fun x y -> x >= y)` and `(fun x y -> x < y)` instead.

   Here is what this solution would look like:

```
let rec filter predicate = function
    [] -> []
  | h::t -> let rest = filter predicate t in
              if predicate h
                then h::rest
                else rest;;

let rec quicksort = function
    [] -> []
  | pivot::t ->
      let lessthan_pivot = ((>=) pivot) in
        let greaterthanorequal_pivot = ((<) pivot) in
          let firstpart = quicksort (filter lessthan_pivot t) in
            let secondpart = quicksort (filter greaterthanorequal_pivot t) in
              firstpart @ [pivot] @ secondpart;;
```

   Notice that the `filter` function's first argument is a predicate that is applied to one argument– each element of the list– while the `<` and `>=` operators in OCaml are applied to two arguments. What we need is to create two functions that will each take one argument and compare it (using `<` or `>=`) to a single element. This is easy with currying– we create two functions (called `lessthan_pivot` and `greaterthanorequal_pivot`) that are each closures consisting of one of the operators `<` or `>=` plus they capture the pivot in their environment. They only need one additional argument to be supplied, which will be compared to the pivot. Lastly, note that since `<` and `>=` both compare their two arguments, returning true if the relationship is true between them, to create a function that returns true if its argument is less than the pivot what we really need is `((>=) pivot`, and to create a function that returns true if its argument is greater than or equal to the pivot what we really need is `((<) pivot`.

Here is a version that does not use the filter function paradigm or higher–order functions:

```
let rec all_lessthan element list =
  match list with
     [] -> []
   | h::t -> if (h < element)
               then h::(all_lessthan element t)
             else (all_lessthan element t);;

let rec all_greaterthanorequal element list =
  match list with
     [] -> []
   | h::t -> if (h >= element)
               then h::(all_greaterthanorequal element t)
             else (all_greaterthanorequal element t);;

let rec quicksort list =
  match list with
    [] -> []
  | pivot::t -> (quicksort (all_lessthan pivot t)) @ [pivot] @
                (quicksort (all_greaterthanorequal pivot t));;
```

This solution does not use filter or higher–order functions yet only has one helper function, but one that returns a tuple:

```
let rec split pivot = function
    [] -> ([], [])
  | h::t ->
      let (lt, gt) = split pivot t in
        if h < pivot
          then (h::lt, gt)
          else (lt, h::gt);;

let rec quicksort = function
    [] -> []
  | pivot::t -> let (lt, gt) = split pivot t in
      (quicksort lt) @ [pivot] @ (quicksort gt);;
```