This assignment covers context free grammars and parsing. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work, and you should indicate in your submission who you worked with, if applicable. Assignments can be submitted electronically through Gradescope as a PDF by 11:59 PM PDT. A LaTeX template for writing your solutions is available on the course website.

1. Give the context-free grammar (CFG) for each of the following languages. Any grammar is acceptable - including ambiguous grammars - as long as it has the correct language.

(a) The set of all strings over the alphabet \(\{1, 2, -, \ast\}\) representing valid products of integers where the expression evaluates to some value \(\geq 0\).

Example Strings in the Language:
- \(1*2\)
- \(21*-1*-121\)
- \(-222\)

Strings not in the Language:
- \(2*-2\)
- \(1*\epsilon\)
- \(-12\)
- \(12-12\)

(b) The set of all strings over \(\{[ , ] , \{ , \} , , \}\) (Note this set includes a ,), representing comma separated lists and sets. A set is a \{ followed by a comma separated sequence of list and sets followed by \}, if a set has at least 1 item it may have a trailing comma before its closing \}. A list is defined similarly except it must begin with \[ and end with \].

Example Strings in the Language:
- \(\{}\)
- \(\{[ ]\}, \{[ ] , \} , \{[ ] , ]\}\)
- \([[[[\{, \} , ]]],\]

Strings not in the Language:
- \(\{, , \}\)
- \([\}\}
- \([\{\{, \} , ]\]

(c) The set of all strings over the alphabet \(\{0, 1\}\) where the number of 1’s is at most 2 more than the number of 0’s.

Example Strings in the Language:
- \(111100\)
- \(01000\)
- \(\epsilon\)

Strings not in the Language:
- \(111\)
- \(011101101\)

(d) The set of all strings over the alphabet \(\{0, 1, (, ), +, \ast\}\) which are valid regular expression over the alphabet \(\{0, 1\}\) (note: for this problem we are using + for alternation (OR) as done in the lecture slides). Example Strings in the Language:
- \(111+100\)
- \((1+00)^\ast+0\)
- \(\epsilon\)

Strings not in the Language:
- \(111+\)
- \(0++1\)
- \)((0^\ast\))
2. Extended Backus–Naur form (EBNF) is commonly used syntax for describing CFGs. In EBNF each production rules are labeled with a = instead of -> (e.g. NONTERMINAL = RHS instead of NONTERMINAL -> RHS). Concatenation is explicit with symbols being joined with , and each production rules is terminated by a ;. Terminals are given as quoted strings (S = "a" ; or S = 'a' ; is equivalent to S -> a). Additionally it supports a number of regular expression like operators, a subset of which are described in the table below.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Sample EBNF Rule</th>
<th>Equivalent CFG rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation,</td>
<td>S = &quot;a&quot;, &quot;b&quot; ;</td>
<td>S -&gt; a b</td>
</tr>
<tr>
<td>Grouping ( ... )</td>
<td>S = ( A</td>
<td>B ), &quot;x&quot; ;</td>
</tr>
<tr>
<td>Optional [ ... ]</td>
<td>S = [A] ;</td>
<td>S -&gt; A</td>
</tr>
<tr>
<td>Iteration { ... }</td>
<td>S = {A} ;</td>
<td>S -&gt; A S</td>
</tr>
</tbody>
</table>

EBNF in its own grammar is:

```
grahmar = { rule } ;
grahle = nonterm , "w" , rhs , ";" ;
rhs = items , { "|" , items } ;
items = item , { "," , item } ;
item = "[" , rhs , "]" | "{" , rhs , "}" | atom ;
atom = "(" , rhs , ")" | term | nonterm ;
nonterm = LETTER , { LETTER | DIGIT | \_ } ;

term = ( '\'' , string , '\'' ) | ( "\," , string , "\," ) ;

string = char , { char } ;
char = LETTER | DIGIT | SYMBOL ;
```

Where LETTER is any [a-zA-Z], digit is [0-9] and SYMBOL is any other ascii character excluding " and ’ (in reality quotes would be allowed but this makes the string rules more complicated then necessary for the purposes of this problem).

Write CFG which recognizes the subset of of EBNF described above. You may use the above character classes. E.g. S -> DIGIT to mean S -> 0 | 1 | 2 | ... | 9. To avoid ambiguity in the matching of | and alternation use \\ to match a literal |.
3. (a) Left factor the following grammar:

\[
S \rightarrow S + S \mid S + P \\
P \rightarrow P * P \mid P * I \\
I \rightarrow -I \mid (S) \mid D \\
D \rightarrow 0 \mid 1N \\
N \rightarrow 0 \mid 1 \mid NN \mid \epsilon
\]

(b) Eliminate left recursion from the following grammar:

\[
S \rightarrow S a S \mid U \\
U \rightarrow U u U \mid T \\
T \rightarrow t \mid f \mid T n \mid (S)
\]

4. Consider the following CFG, where the set of terminals is \{0, 1, (, ), ;\}:

\[
S \rightarrow (T \\
T \rightarrow CA | ) \\
A \rightarrow ; B | ) \\
B \rightarrow CA | ) \\
C \rightarrow 0 \mid 1 \mid S
\]

(a) Construct the FIRST sets for each of the nonterminals.
(b) Construct the FOLLOW sets for each of the nonterminals.
(c) Construct the LL(1) parsing table for the grammar.
(d) Show the sequence of stack, input and action configurations that occur during an LL(1) parse of the string “( ( ) ; 0 )”. At the beginning of the parse, the stack should contain a single S.

5. What advantage does left recursion have over right recursion in shift-reduce parsing? **Hint:** Consider left and right recursive grammars for the language a*. What happens if your input has a million a’s?
6. Consider the following Grammar G over the alphabet \( \Sigma = \{a, b\} \):

\[
\begin{align*}
S & \rightarrow S \\
S & \rightarrow Aa \\
S & \rightarrow Bbb \\
A & \rightarrow aaA \\
A & \rightarrow \epsilon \\
B & \rightarrow Bbb \\
B & \rightarrow \epsilon 
\end{align*}
\]

You want to implement G using an SLR(1) parser (note that we have already added the \( S' \rightarrow S \) production for you).

(a) Construct the first state of the LR(0) machine, compute the FOLLOW sets of A and B, and point out the conflicts that prevent the grammar from being SLR(1).

(b) Show modifications to production to make the grammar SLR(1) while having the same language as the original grammar G. Explain the intuition behind this result.

7. (EXTRA CREDIT) Define a set of semantic actions (pseudo code is fine) on the CFG you generated for problem 2 that will transform a parsed EBNF into a CFG. Define new CFG productions with the function `add_rule(nt, rhs)` where `nt` is a nonterminal, and `rhs` is sequence of terminals and non-terminals. You may assume you have access to a function `fresh_nt()` which will generate a unique non-terminal. For example to create the rule \( S \rightarrow aA \mid b \mid \epsilon \):

\[
S = \text{NonTerminal('S')} \\
A = \text{NonTerminal('A')} \\
a = \text{Terminal('a')} \\
b = \text{Terminal('b')} \\
\text{add_rule}(S, [a, A]) \\
\text{add_rule}(S, [b]) \\
\text{add_rule}(S, []) /* adding the \( \epsilon \) production */
\]

Outline the basic idea behind your approach, and supply a semantic action for each production your grammar.