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, -, \*\} representing valid products of integers where the expression evaluates to some value $\geq 0$.

\[
P \rightarrow P \* P \mid N \* N \mid I \mid -N \\
N \rightarrow N \* P \mid P \* N \mid -P \\
I \rightarrow DI \mid D \\
D \rightarrow 1 \mid 2
\]

(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 ].

\[
S \rightarrow \{L\} \mid [L] \\
L \rightarrow S,L \mid S \mid \epsilon
\]

(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.

\[
S \rightarrow TT \mid \epsilon \\
T \rightarrow 1Q \mid 0S \mid \epsilon \\
Q \rightarrow 0T \mid T0 \mid \epsilon
\]

(d) The set of all strings over the alphabet \{0, 1, (,), +, \*\} 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).

This problem was pretty poorly defined. Basically the only way to lose points on it were to not have balanced parentheses or to miss one of the operators (concatenation, iteration, alternation).

\[
S' \rightarrow S \mid \epsilon \\
S \rightarrow SS \mid C \mid A \mid I \mid G \\
C \rightarrow 0 \mid 1 \\
A \rightarrow S + S \\
I \rightarrow S\* \\
G \rightarrow (S)
\]
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:

```
grammar = { rule } ;
rule = 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 \(\backslash\) to match a literal |.

**ANSWER:**

```
GRAMMAR -> RULE GRAMMAR | \(\epsilon\)
RULE -> NONTERM = RHS ;
RHS -> ITEMS RHS ;
RHS' -> \(|\) ITEMS RHS' | \(\epsilon\)
ITEMS -> ITEM ITEMS' ;
ITEMS' -> , ITEM ITEMS' | \(\epsilon\)
ITEM -> [ RHS ] | { RHS } | ATOM
ATOM -> ( RHS ) | NONTERM | TERM
NONTERM -> LETTER NONTERM ;
NONTERM' -> NONTERM' , NOTERM' | \(\epsilon\)
NOTERM' -> LETTER | DIGIT | _
TERM -> " STRING " | \(' STRING \',
STRING -> CHAR STRING ;
STRING' -> CHAR STRING' | \(\epsilon\)
CHAR -> LETTER | DIGIT | SYMBOL
```
3. (a) Left factor the following grammar:

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

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

(b) Eliminate left recursion from the following grammar:

\[
S \rightarrow S \ast S \mid U \\
U \rightarrow U \ast U \mid T \\
T \rightarrow t \mid f \mid T \ast n \mid (S) \\
S \rightarrow US' \\
U \rightarrow TU' \\
T \rightarrow tT' \mid fT' \mid (S)T' \\
S' \rightarrow aSS' \mid \epsilon \\
U' \rightarrow uUU' \mid \epsilon \\
T' \rightarrow nT' \mid \epsilon
\]
4. Consider the following CFG, where the set of terminals is \{0, 1, (, ), ;\}:

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

(a) Construct the FIRST sets for each of the nonterminals.
- S: \{(\}
- T: {), 0, 1, (}
- A: {; ,}
- B: {), 0, 1, (}
- C: {0, 1, (}

(b) Construct the FOLLOW sets for each of the nonterminals.
- S: \{$, ;, )$}
- T: \{$, ;, )$}
- A: \{$, ;, )$}
- B: \{$, ;, )$}
- C: \{; ,\}

(c) Construct the LL(1) parsing table for the grammar.

| Nonterminal | ( | ) | ; | 0 | 1 | $ |
|-------------|----|----|---|---|---|
| S | ( | T |
| T | C | A | C | A |
| A | ) | ; | B |
| B | C | A | C | A |
| C | S | 0 | 1 |

(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.
<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S $</td>
<td>( ( ) ; 0 ) $</td>
<td>$S \rightarrow (T$</td>
</tr>
<tr>
<td>( T $</td>
<td>( ( ) ; 0 ) $</td>
<td>match (</td>
</tr>
<tr>
<td>T $</td>
<td>( ) ; 0 ) $</td>
<td>$A \rightarrow CA$</td>
</tr>
<tr>
<td>C A $</td>
<td>( ) ; 0 ) $</td>
<td>$C \rightarrow S$</td>
</tr>
<tr>
<td>S A $</td>
<td>( ) ; 0 ) $</td>
<td>$S \rightarrow (T$</td>
</tr>
<tr>
<td>( T A $</td>
<td>( ) ; 0 ) $</td>
<td>match (</td>
</tr>
<tr>
<td>T A $</td>
<td>) ; 0 ) $</td>
<td>$T \rightarrow )$</td>
</tr>
<tr>
<td>) A $</td>
<td>) ; 0 ) $</td>
<td>match )</td>
</tr>
<tr>
<td>A $</td>
<td>) ; 0 ) $</td>
<td>$A \rightarrow ;B$</td>
</tr>
<tr>
<td>; B $</td>
<td>; 0 ) $</td>
<td>match ;</td>
</tr>
<tr>
<td>B $</td>
<td>0 ) $</td>
<td>$B \rightarrow CA$</td>
</tr>
<tr>
<td>C A $</td>
<td>0 ) $</td>
<td>$C \rightarrow 0$</td>
</tr>
<tr>
<td>0 A $</td>
<td>0 ) $</td>
<td>match 0</td>
</tr>
<tr>
<td>A $</td>
<td>) $</td>
<td>$A \rightarrow )$</td>
</tr>
<tr>
<td>) $</td>
<td>) $</td>
<td>match )</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
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?

**ANSWER:** Consider what happens with the right recursive grammar for $a^*$. $S → aS | \epsilon$ The resulting shift-reduce machine will shift the entire input onto the stack, before performing the reduction. It will accept all strings in the language, but it requires an unbounded stack size.

Now consider $S → Sa | \epsilon$. This machine will initially reduce by $S → \epsilon$, and then alternate between shifting an "a" on to the stack and reducing by $S → Sa$. The stack space used is thus constant. In general, using left recursion in grammars for shift-reduce parsers helps limit the size of the stack.
6. Consider the Following Grammar G over the alphabet \( \Sigma = \{a, b\} \):

\[
S'' \rightarrow S \\
S \rightarrow Aa \\
S \rightarrow Bbb \\
A \rightarrow aaA \\
A \rightarrow \epsilon \\
B \rightarrow Bbb \\
B \rightarrow \epsilon
\]

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)

First state:

\[
S'' \rightarrow S \\
S \rightarrow Aa \\
S \rightarrow Bbb \\
A \rightarrow aaA \\
A \rightarrow \epsilon \\
B \rightarrow Bbb \\
B \rightarrow \epsilon
\]

FOLLOW(A) = \{a\} and FOLLOW(B) = \{b\}. We have a shift-reduce conflict between the productions \( A \rightarrow aaA \) and \( A \rightarrow \epsilon \) when reading an a. This is because we could either use the item \( A \rightarrow \). to reduce \( \epsilon \) to \( A \) (as \( a \in \text{FOLLOW}(A) \)) or use the item \( A \rightarrow .aaA \) to shift the \( a \).

(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.

**ANSWER:** If we change the either production \( A \rightarrow aaA \) to \( A \rightarrow Aaa \) or \( S \rightarrow Aa \) to \( S \rightarrow aA \) we eliminate the conflict. Intuitively, the issue is that we need to look two characters ahead to know if we are on the "last" A. If we make the first change ( \( A \rightarrow aaA \) to \( A \rightarrow Aaa \)) the grammar will have the property that an \( a \) is always preceded by an \( A \) (in the first state) so it doesn't matter if we are on the "last" A.

If we make the second change ( \( S \rightarrow Aa \) to \( S \rightarrow aA \)) we will get a some what different state machine but we will be able to decide whether or not we are parsing the last \( A \) as it will be followed by \$. 
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 \texttt{add\_rule(nt, rhs)} where \texttt{nt} is a nonterminal, and \texttt{rhs} is sequence of terminals and non-terminals. You may assume you have access to a function \texttt{fresh\_nt()} which will generate a unique non-terminal. For example to create the rule \( S \rightarrow aA | b | \epsilon \):

\[
S = \text{NonTerminal(‘S’)}
A = \text{NonTerminal(‘A’)}
a = \text{Terminal(‘a’)}
b = \text{Terminal(‘b’)}
add\_rule(S, [a, A])
add\_rule(S, [b])
add\_rule(S, []) /* adding the \epsilon production */
\]

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

Basic Idea:
replace \([A]\) with a new fresh nonterminal \(X\) with productions \(X \rightarrow A\) and \(X \rightarrow \epsilon\).
replace \((A|B)\) with new fresh nonterminal \(X\) with productions \(X \rightarrow A\) and \(X \rightarrow B\).
replace \(\{A\}\) with new fresh nonterminal \(X\) with productions \(X \rightarrow AX\) and \(X \rightarrow \epsilon\).
Otherwise just build rules up from the bottom and add them.

In the syntax of the lecture notes (see below for bison version):

```
0  GRAMMAR  ->  RULE  GRAMMAR
  |  \epsilon
1  RULE  ->  NONTERM = RHS ;
2  RHS  ->  ITEMS  RHS ’
3  RHS’ 0  ->  \| ITEMS  RHS’ 1
4  ITEMS  ->  ITEM  ITEMS’
5  ITEMS’ 0  ->  , ITEM  ITEMS’ 1
6  ITEM  ->  [ RHS ]
7  |  \{ RHS \}
8  |  ATOM
9  ATOM  ->  ( RHS )
10  |  NONTERM
11  |  TERM
12  NONTERM  ->  LETTER  NONTERM’
13  NONTERM’ 0  ->  NONTERM’’  NOTERM’ 1
14  |  \epsilon
15  NONTERM’’  ->  LETTER
16  |  DIGIT
17  TERM  ->  " STRING "
18  |  ‘ STRING ’
19  STRING  ->  CHAR  STRING’
20  STRING’ 0  ->  CHAR  STRING’ 1
21  |  \epsilon
22  CHAR  ->  LETTER
23  |  DIGIT
24  |  SYMBOL
```

8
class Symbol:
    val: str

class Terminal(Symbol): pass
class NonTerminal(Symbol): pass

class Component:
    val: Symbol

class Atom(Component): pass
class Item(Component): pass

class Items:
    val: List[Symbol]

class RHS:
    val: List[Items]

0 { }
1 { }
2 {
    for rule in RHS.val:
        add_rule(NONTERM.val, rule.val)
}
3 { RHS.val = concat([ITEMS], RHS'.val)}
4 { RHS'.val = concat([ITEMS], RHS'_1.val)}
5 { RHS'_0.val = [] }
6 { ITEMS.val = concat([ITEM.val], ITEMS'.val)}
7 { ITEMS'.val = concat([ITEM.val], ITEMS'1.val) }
8 { ITEMS'_0.val = [] }
9 {
    X = fresh_nt()
    for rule in RHS.val:
        add_rule(X, rule.val)
    add_rule(X, [])
    ITEM.val = X
}
10 {
    X = fresh_nt()
    for rule in RHS.val:
        add_rule(X, concat(rule.val, X))
    add_rule(X, [])
    ITEM.val = X
}
11 { ITEM.val = ATOM.val }
12 {
    X = fresh_nt()
    for rule in RHS.val:
        add_rule(X, rule.val)
    ATOM.val = X
}
13 { ATOM.val = NONTERM.val }
14 { ATOM.val = TERM.val }
15 { NONTERM.val = NonTerminal(concat(LETTER.val, NONTERM'.val)) }
16 { NONTERM\_0.val = concat(NONTERM\_\'.val, NONTERM\_1.val) }
17 { NONTERM\_0.val = '' }
18 { NONTERM\_\'.val = LETTER }
19 { NONTERM\_\'.val = DIGIT.val }
20 { NONTERM\_\'.val = '_' }
21 { TERM.val = Terminal(STRING.val) }
22 { TERM.val = Terminal(STRING.val) }
23 { STRING.val = concat(CHAR.val, STRING\_\'.val) }
24 { STRING\_0.val = concat(CHAR.val, STRING\_1.val) }
25 { STRING\_0.val = '' }
26 { CHAR.val = LETTER.val }
27 { CHAR.val = DIGIT.val }
28 { CHAR.val = SYMBOL.val }
In bison syntax with c++14 (using _ for ' in rule names):

%union {
    Items items;
    Symbols symbols;
    Symbol symbol;
    String string;
}

%type <items> RHS RHS_ ITEMS ITEMS_
%type <symbols> ITEM
%type <symbol> ATOM NONTERM TERM
%type <string> LETTER DIGIT SYMBOL CHAR
%type <string> STRING_ STRING NONTERM__ NONTERM_

GRAMMAR : RULE GRAMMAR {}
    | /* empty */ {};

RULE : NONTERM '=' RHS ';' {
    for (auto rule : $3) {
        add_rule($1, rule.val);
    }
}

RHS : ITEMS RHS_ {
    $$ = append_Items($1, $2);
}

RHS_ : '|' ITEMS RHS_ {
    $$ = append_Items($2, $3);
} | /* empty */ {
    $$ = nil_Items();
}

ITEMS : ITEM ITEMS_ {
    $$ = append_Items(single_Items($1), $2);
}

ITEMS_ : ' ,' ITEM ITEMS_ {
    $$ = append_Items(single_Items($2), $3);
} | /* empty */
ITEM: `\[` RHS `\]`
{
    auto X = fresh_nt();
    for (auto rule : $2) {
        add_rule(X, rule.val);
    }
    add_rule(X, nil_Symbols());
    $$ = single_Symbols(X);
}
|
\{`\{` RHS `\}`
{
    auto X = fresh_nt();
    for (auto rule : $2) {
        add_rule(X, append_Symbols(rule.val, single_Symbols(X)));
    }
    $$ = single_Symbols(X);
}
| ATOM
{
    $$ = single_Symbols($1);
}
|
ATOM: `\(` RHS `\)`
{
    auto X = fresh_nt();
    for (auto rule : $2) {
        add_rule(X, rule.val);
    }
    $$ = X;
}

/* using the default $$ = $1 for the following */
| NONTERM
| TERM
|
NONTERM: LETTER NONTERM_
{
    $$ = NonTerminal(append_String($1, $2));
}
|
NONTERM_: NONTERM__ NONTERM_
{
    $$ = append_String($1, $2);
} /* Empty */
{
    $$ = nil_String();
}
/* using the default $$ = $1 for the following */
NONTERM__ : LETTER | DIGIT | '_';

TERM: '"', STRING '"'
    { $$ = Terminal($2); }
    | "'" STRING "'"'
    { $$ = Terminal($2); }
    ;

STRING: CHAR STRING_
    { $$ = append_String($1, $2); }
    ;

STRING_: CHAR STRING_
    { $$ = append_String($1, $2); }
    | /* Empty */
    { $$ = nil_String(); }
    ;

/* using the default $$ = $1 for the following */
CHAR: LETTER | DIGIT | SYMBOL;