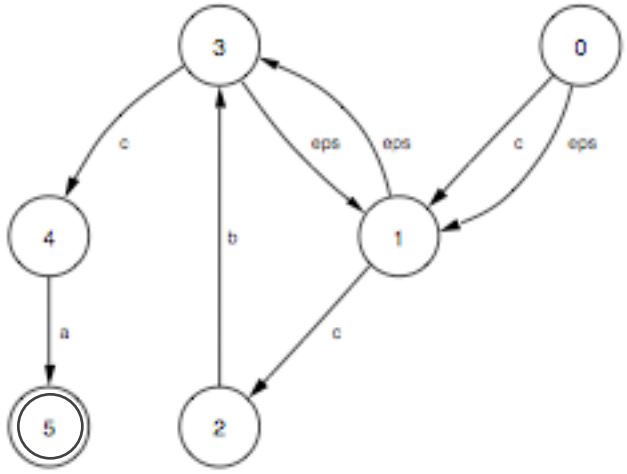
Solution Set 1: CFGs and Top-Down Parsing

This week's assignment consists of written exercises on regular expressions, formal grammars, and top-down parsing. You are to write up solutions and hand them in on paper.

1) Consider the regular language described by (ε + c)(cb)*ca.
   a) Develop an ε-NFA using Thompson's Rules that accepts this language. It's okay to collapse unimportant states.

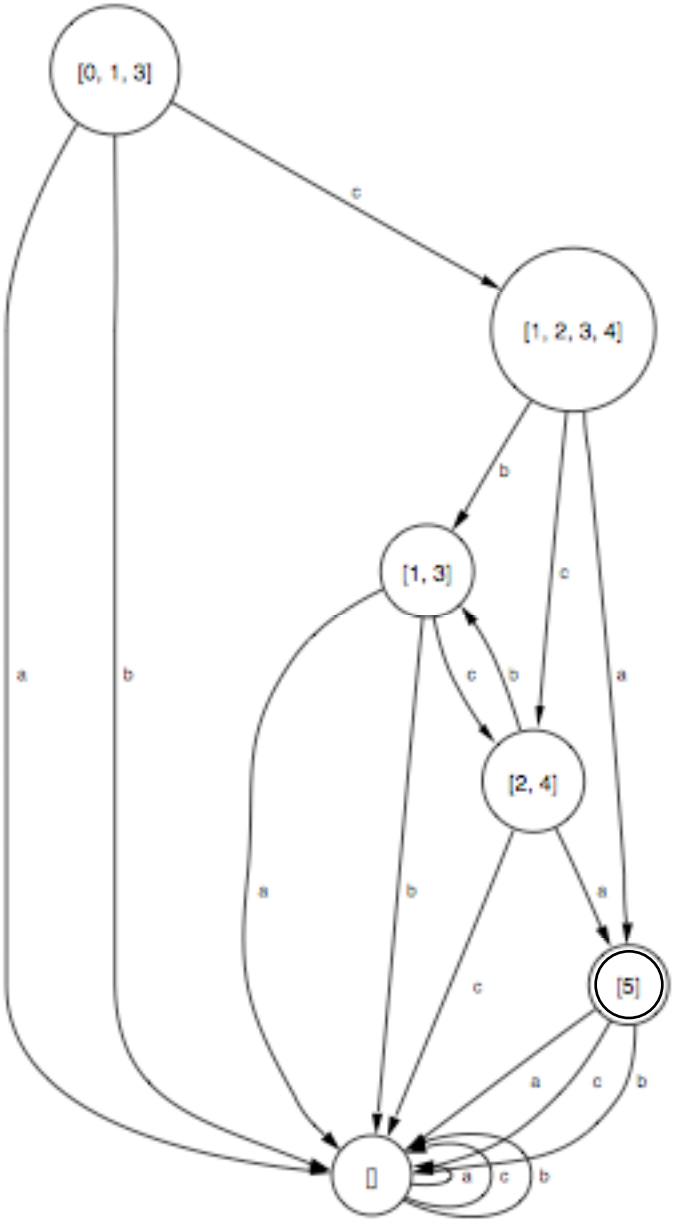
Our ε-NFA:



   b) Convert the ε-NFA into a DFA using subset construction. Show your work. You do not need to minimize the DFA.

Here are the subset construction steps for the NFA:
`eps-closure` is short for eplison closure, which computes how a subset in the construction extends to include even more NFA states if free rides along epsilon transitions are taken. **`move`** simultaneously tracks all of the states you can travel to while processing a single character:

```
start_state=0
eps-closure(0)=[0,1,3]
considering T=[0,1,3]
move([0,1,3], "a")=[]
eps-closure([])=[]
move([0,1,3], "c")=[2,4,1]
eps-closure([2,4,1])=[1,2,3,4]
move([0,1,3], "b")=[]
eps-closure([])=[]
considering T=[]
move([], "a")=[]
eps-closure([])=[]
move([], "c")=[]
eps-closure([])=[]
move([], "b")=[]
eps-closure([])=[]
considering T=[1,2,3,4]
move([1,2,3,4], "a")=[5]
eps-closure([5])=[5]
move([1,2,3,4], "c")=[2,4]
eps-closure([2,4])=[2,4]
move([1,2,3,4], "b")=[3]
eps-closure([3])=[1,3]
considering T=[1,3]
move([1,3], "a")=[]
eps-closure([])=[]
move([1,3], "c")=[2,4]
eps-closure([2,4])=[2,4]
move([1,3], "b")=[]
eps-closure([])=[]
considering T=[2,4]
move([2,4], "a")=[5]
eps-closure([5])=[5]
move([2,4], "c")=[]
eps-closure([])=[]
move([2,4], "b")=[3]
eps-closure([3])=[1,3]
considering T=[5]
move([5], "a")=[]
eps-closure([])=[]
move([5], "c")=[]
eps-closure([])=[]
move([5], "b")=[]
eps-closure([])=[]
```


[0, 1, 3]
[1, 2, 3, 4]
[1, 3]
[2, 4]
[5]
[]
a
b
c

2) In PP1, you used a **`lex`** pattern similar or equal to **`[0-9]+`** to recognize decimal integers.
   a) Specify a **`lex`** pattern that recognizes only decimal integers that are divisible by 2.

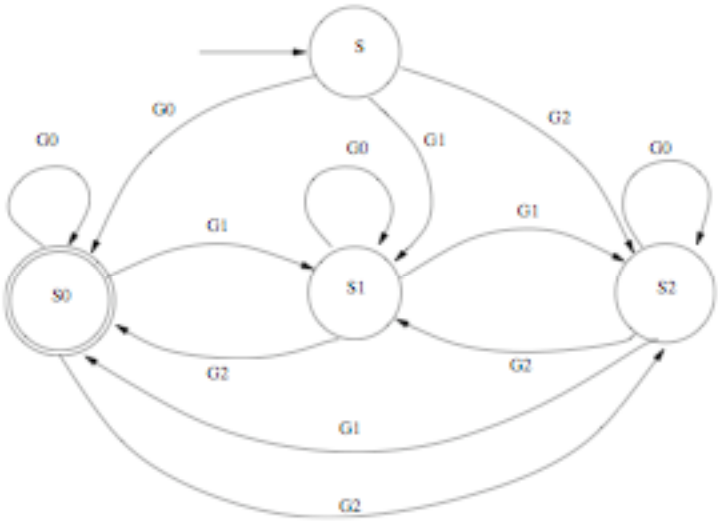
      We're hoping you thought this was easy: **`[0-9]*[24680]`** ☺

   b) Is there also a lex pattern that recognizes only decimal integers that are divisible by 3? Justify your answer!

      Believe it or not, there is! Why is there a **`lex`** pattern? Because there's a DFA that can track the modulo 3 characteristics of a decimal digit string and accept just those equal to 0 modulo 3. Check this out:

      Let G0 stand for the character class [0369].
      Let G1 stand for the character class [147].
      Let G2 stand for the character class [258].

      The top state is the start state, and the leftmost one is final.



      Now, it is a routine matter to turn this DFA into a lex pattern. It is given as follows:

```
%}
T (2|5|8)
O (1|4|7)
Z (0|3|6|9)
TZO {T}{Z}*{O}
OZO {O}{Z}*{O}
OZT {O}{Z}*{T}
TZT {T}{Z}*{T}
DIV3 ( (({TZT}|{O})({OZT}|{Z})*({OZO}|{T})) | {TZO} | {Z})+
%%
```

3) Consider the following grammar for regular expressions:

$$\mathbf{N} = \{R\}$$
$$\mathbf{T} = \{ \mid, *, (,), \langle id \rangle \}$$
$$\mathbf{P} = \{R \rightarrow R\ R,\ R \rightarrow R \mid R,\ R \rightarrow R^*,\ R \rightarrow (R),\ R \rightarrow \langle id \rangle\}$$

The start symbol is R.

a) Show that this grammar is ambiguous. Give an example.

There are two distinct parse trees for <id> | <id> *.

b) How would the ambiguity cause problems when interpreting the parse trees?

The ambiguity interferes with our ability to reliably interpret operator precedence. The above string could be semantically equivalent to <id> | (<id>*), or it could be equivalent to (<id> | <id>)*. The grammar says nothing about how tightly the Kleene star should bind to it operand.

c) Construct an LL(1) grammar for regular expressions that gives * higher precedence than |.

This isn't trivial, but it relies on the same ad-hoc inspection techniques we used to disambiguate the expression grammar. Here's what I came up with:

R → T R'
R' → '|' T R'
R' → ε

} Allow T | T |… T | T to be built up

T → F T'
T'→ F T'
T'→ ε

} Allow each T to expand to F F F … F

F → <id> K
F → (R) K
K → * K
K → ε

4) Consider the following context-free grammar:

$$\mathbf{N} = \{S, G, R, L, C\}$$
$$\mathbf{T} = \{n, s, c, y\}$$
$$\mathbf{P} = \{ S \rightarrow G,\ G \rightarrow R,\ G \rightarrow G\ R,\ R \rightarrow n\ L\ s,\ L \rightarrow C,\ L \rightarrow L\ c\ C,\ C \rightarrow \varepsilon,\ C \rightarrow C\ y \}$$

The start symbol is S.

a) Is this grammar ambiguous?

Nope, it's fine. The predictive parsing table from part d tells us this, but we don't even need it. Notice that there's at most one leftmost derivation of each string in the language.

b) Explain why the grammar is not LL(1).

A grammar can't be LL(1) if there's left recursion, which this grammar certainly has. It doesn't mean the language isn't LL(1); it just means the grammar we've chosen to represent it isn't.

c) Transform the grammar into a LL(1) grammar that accepts the same language.

Just eliminate the left recursion the way the LL(1) handout says to. Here's what I got:

$S \rightarrow G$
$G \rightarrow RG'$
$G' \rightarrow RG'$
$G' \rightarrow \varepsilon$
$R \rightarrow nLs$
$L \rightarrow CL'$
$L' \rightarrow cCL'$
$L' \rightarrow \varepsilon$
$C \rightarrow C'$
$C' \rightarrow yC'$
$C' \rightarrow \varepsilon$

d) Develop the parse table for a top-down predictive parser for your LL(1) grammar.

We have the following parsing table:

	n	s	c	y	$
S	S → G				
G	G → RG'				
G'	G'→ RG'				G'→ ε
R	R → nLs				
L		L → CL'	L → CL'	L → CL'	
L'		L'→ ε	L'→ cCL'		
C		C → C'	C → C'	C → C'	
C'		C'→ ε	C'→ ε	C'→ yC'	

And because we don't have any multiply defined entries, we see that the grammar is LL(1). Woo.

e) Trace through the parse for the input **nycys**.

Stack	Input	Action
$S\$$	$nycys\$$	pop S, push G, predict $S \to G$
$G\$$	$nycys\$$	pop G, push RG', predict $G \to RG'$
$RG'\$$	$nycys\$$	pop R, push nLs, predict $R \to nLs$
$nLsG'\$$	$nycys\$$	pop n, match n
$LsG'\$$	$ycys\$$	pop L, push CL', predict $L \to CL'$
$CL'sG'\$$	$ycys\$$	pop C, push C', predict $C \to C'$
$C'L'sG'\$$	$ycys\$$	pop C', push yC', predict $C' \to yC'$
$yC'L'sG'\$$	$ycys\$$	pop y, match y
$C'L'sG'\$$	$cys\$$	pop C', predict $C' \to \epsilon$
$L'sG'\$$	$cys\$$	pop L', push cCL', predict $L' \to cCL'$
$cCL'sG'\$$	$cys\$$	pop c, match c
$CL'sG'\$$	$ys\$$	pop C, push C', predict $C \to C'$
$C'L'sG'\$$	$ys\$$	pop C', push yC', predict $C' \to yC'$
$yC'L'sG'\$$	$ys\$$	pop y, match y
$C'L'sG'\$$	$s\$$	pop C', predict $C' \to \epsilon$
$L'sG'\$$	$s\$$	pop L', predict $L' \to \epsilon$
$sG'\$$	$s\$$	pop s, match s
$G'\$$	$\$$	pop G', predict $G' \to \epsilon$
$\$$	$\$$	Success

f) Sketch the functions required for a recursive-descent parser for your LL(1) grammar.

```
static int lookahead;

void matchToken(int expected)
{
   if (lookahead != expected) {
      fprintf(stderr, "Error: expected %c, saw %c\n", expected, lookahead);
      exit(-1);
   }

   lookahead = getchar();
}
```

```
static int S()
{
   switch (lookahead) {
      case 'n': G();
                return 0;
      default: fprintf(stderr, "Error.\n");
               exit(-1);
   }
}

static int G()
{
   switch (lookahead) {
      case 'n': R();
                Gprime();
                return 0;
      default: fprintf(stderr, "Error.\n");
               exit(-1);
   }
}

static int Gprime()
{
   switch (lookahead) {
      case 'n': R();
                Gprime();
                return 0;
      case EOF: // G' is nullable
                return 0;
      default: fprintf(stderr, "Error.\n");
               exit(-1);
   }
}

static int R()
{
   switch (lookahead) {
      case 'n': matchToken('n');
                L();
                matchToken('s');
                return 0;
      default: fprintf(stderr, "Error.\n");
               exit(-1);
   }
}

static int L()
{
   switch (lookahead) {
      case 'y':
      case 'c': C();
                Lprime();
                return 0;
      case 's': // L is nullable
                return 0;
      default: fprintf(stderr, "Error.\n");
               exit(-1);
   }
}
```

```
static int Lprime()
{
   switch (lookahead) {
      case 'c': matchToken('c');
                C();
                Lprime();
                return 0;
      case 's': // Lprime is nullable
                return 0;
      default: fprintf(stderr, "Error.\n");
               exit(-1);
   }
}

static int C()
{
   switch (lookahead) {
      case 'y': Cprime();
                return 0;
      case 's':
      case 'c': // C is nullable
                return 0;
      default: fprintf(stderr, "Error.\n");
               exit(-1);
   }
}

static int Cprime()
{
   switch (lookahead) {
      case 'y': matchToken('y');
                Cprime();
                return 0;
      case 's':
      case 'c': // Cprime is nullable
                return 0;
      default: fprintf(stderr, "Error.\n");
               exit(-1);
   }
}

int main()
{
   matchToken(0); // init lookahead
   S();
   printf("Success!\n");
}
```

g) Is the language described by the grammar regular?

Sure is. It's given by the regular expression `(ny*(cy*)*s)+`.
S generates a string of one or more G's. That's where the outer **+** comes from.
Each G eventually becomes an R, which generates a string that begins with **n** and ends with **s**. Between the **n** and the **s** sits whatever can be generated by L. L generates sentential forms that look like C**c**C**c**…C**c**C, where C expands to one or more **y**'s.