1. We can use recursion to implement a CFG recognizer

Recall the definition of \( L(G) \) for any CFG \( G = \langle C, X, S, R \rangle \):

\[
L(G) = \{ x_1..x_n \mid \exists c \in S . c \xrightarrow{G}^* x_1..x_n \}
\]

where:

\[
c \xrightarrow{G}^* x_i..x_j \iff \begin{cases} 
  i = j : c = x_i \\
  i < j : \exists k, d, e \text{ s.t. } c \rightarrow d e \in R \text{ and } d \xrightarrow{G}^* x_i..x_k \text{ and } e \xrightarrow{G}^* x_{k+1}..x_j
\end{cases}
\]

For CFG \( G \), we can convert this to a recursive function to recognize \( L(G) \):

```python
import sys
import re
import model

S = model.Model('S')
C = model.Model('C')
R = model.Model('R')

def Rec(c, i, j, X):
    if i == j:
        return (c==X[i])
    else:
        v = False
        for k in range(i, j):
            for d in C:
                for e in C:
                    if (c, d, e) in R:
                        v = v or (R[c, d, e] and Rec(d, i, k, X) and Rec(e, k+1, j, X))
        return v

for line in sys.stdin:
    S.read(line)
    C.read(line)
    R.read(line)
    m = re.search('I (. *)', line)
    if m != None:
        I = re.split(' +', m.group(1))
        print (Rec('S', 0, len(I)-1, I))
```

When run on the model file `cfg.model`:

\[
\begin{array}{l}
S : S = 1 \\
C : S = 1 \\
C : VP = 1 \\
C : NP = 1 \\
\end{array}
\]
and the input file `cat-toy.in`:

```plaintext
I the cat hit the toy off the mat
```

(e.g. `cat cfg.model cat-toy.in | python parser.py`)

produces:

```plaintext
True
```

Correctness:

Use ‘recursion invariant’: \( \text{Rec}(c,i,j,X) \) computes \( c \xrightarrow{*} G^{x_{i..j}} \)

Complexity:

\[
\tau(\text{Rec}, n) \geq n \cdot |C| \cdot |C| \cdot \tau(\text{Rec}, n-1) \geq n! \cdot |C|^{2n} \notin O(n^k) \quad \text{— not polynomial!}
\]

(NOTE: technically, ‘lazy evaluation’ saves us, but only for boolean semirings)

2. Memoized algorithm: record partial results

Avoid duplication of effort by recording partial results in \( V \), checking for duplicates:

```python
V = {}

def Rec(c, i, j, X):
    if (c, i, j) not in V:
        if i == j:
            return (c == X[i])
        else:
            V[c, i, j] = False
            for k in range(i, j):
                for d in C:
                    for e in C:
```
if (c,d,e) in R:
    V[c,i,j] = V[c,i,j] or (R[c,d,e] and
    Rec(d,i,k,X) and
    Rec(e,k+1,j,X))

return V[c,i,j]

Correctness:
Same recursion invariant: Rec(c, i, j, X) computes $c \xrightarrow{G} x_i..x_j$
Only change was to add first line to check for duplicates

Complexity:
Recursion only explored once for each (c, i, j)
Since only $|C| \cdot n \cdot n$ possible instances of (c, i, j):
$\tau(\text{Rec}, n) \in \mathcal{O}(|C| \cdot n \cdot n \cdot |C| \cdot |C|) = \mathcal{O}(|C|^3 n^3)$ — now it is polynomial!
This is called ‘dynamic programming’

3. Tabular/’bottom-up’ dynamic programming algorithm:
Consider ‘recursion tree’ defined by usage of program stack in memoized DP algo:

```
Rec('PP',5,7,X)
  Rec('on',5,5,X)   Rec('NP',6,7,X)
    Rec('the',6,6,X) Rec('mat',7,7,X)
```

This can be simplified to remove function recursion.
Loop over each Rec(c, i, j, X) from bottom to top of recursion tree
(has to be bottom-up to ensure sub-solutions are there when you need them):

V = {}

def Rec(cS,_,n,X):
    for j in range(0,n+1):
        for i in range(j,-1,-1):
            for c in C:
                if i == j:
                    V[c,i,j] = (c==X[i])
                else:
                    V[c,i,j] = False
                    for k in range(i,j):
                        for d in C:
                            for e in C:
                                if (c,d,e) in R:
                                    V[c,i,j] = V[c,i,j] or (R[c,d,e] and
                                    V[d,i,k] and
                                    V[e,k+1,j])
    return V[cS,0,n]
Any memoized recursive algorithm can be rewritten this way!

Here's an example run:

```
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>the</td>
<td>NP</td>
<td></td>
<td></td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>i=1</td>
<td>cat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=2</td>
<td>hit</td>
<td>VP</td>
<td>VP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=3</td>
<td>the</td>
<td>NP</td>
<td>NP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=4</td>
<td></td>
<td>toy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=5</td>
<td></td>
<td>off</td>
<td>PP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=6</td>
<td></td>
<td></td>
<td>the</td>
<td>NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>the</td>
<td>mat</td>
</tr>
</tbody>
</table>
```

Correctness:
Loop invariant instead of recursion, but still: \( c, i, j \) computes \( c \xrightarrow{*} G x_i \ldots x_j \)

Only change in outer loops

Complexity:

\[
\tau(\text{Rec}, n) \in O(n \cdot |C| \cdot n \cdot |C| \cdot |C|) = O(|C|^3 n^3) \] — still polynomial!