Simple recursive substitution parser for CFGs

Main theory:

1. Start with the target string and the start symbol of G as the first sentential form
2. Find all substitution you can make
3. Recursively feed the new sentential forms into the parser
4. If the target string appears, return True

Now, that's not always (ever?) going to work, because the parser can get lost in an infinite derivation.
Of course, you can play against that with a clever sorting of the rules (recursive one last), but as soon as you have more than one recursive rule, you're obviously lost.

So we're introducing a depth limiter for the parse that just cuts of longer branches and returns False. So, False means "not with this depth" - nothing more.

Simple Rule class: LHS and RHS, nothing more

In [3]:
```
class Rule:
    def __init__(self,lhs,rhs):
        self.lhs = lhs
        self.rhs = rhs

    def __str__(self):
        return self.lhs + ' --> ' + self.rhs

    def __repr__(self):
        return self.__str__()
```

Little wrapper for Sentential Forms

It just wraps the string but also
allows for computation of all derivations given a rule set

In [4]:
```
class SententialForm:
```

```
    # Typical functional style: filter out all rules from "ruleSet" with "symbol" on the LHS
    @classmethod
    def select(cls,ruleSet,symbol):
        return filter(lambda rule: rule.lhs == symbol, ruleSet)

    def __init__(self, string):
        self.string = string

    def __str__(self):
        return self.string

    # Just a combinatorial problem:
    # 1. Check the string for non-terminals (always uppercase)
    # 2. Find all rules with this non-terminal on the LHS
    # 3. Produce the next sentential form bu substituting the LHS for the symbol
    # 4. Collect everything and return it
    def possibleDerivs(self, rules):
        result = []
        for i in range(len(self.string)): # Check every symbol
            c = self.string[i]
            if c.isupper():               # Uppercase --> non-terminal
                # Check all rules with c on the LHS
                for rule in SententialForm.select(rules,c):
                    # The magic substitution step!
                    newForm = self.string[:i] + rule.rhs + self.string[i+1:]
                    result.append(SententialForm(newForm))
        return result
```

Simple CFG class

Contains just the start symbol and a set of rules

Well, it also contains a recursive, brute force, depth first, left to right (DFS) parser

AND NOW even a BFS parser

In [5]:
```
class CFG:
    def __init__(self, start, rules):
        self.start = start
        self.rules = rules
        self.queue = []

    def addRule(self, rule):
        self.rules.append(rule)
```

```
    # The mighty parser Part 1: Depth First Search version
    # Step 1: Wrap the start symbol into a Sentential Form and jump in
    def parseDFS(self,string,maxDepth = 10):
        return self.parseDFSAux(SententialForm(self.start),string,maxDepth)

    # That's the recursive workhorse
    def parseDFSAux(self,current,string,maxDepth):
        # Recursion base cases - trivial
        if current.string == string: return True    # Got it!
        if maxDepth == 0: return False              # Give up!

        # Here we go down. Compute all possible next steps
        nextForms = current.possibleDerivs(self.rules)

        # For each check if there's a successful parse to the target string
        for form in nextForms:
            print(current.string,'==>',form)
            # Ok. Did this work? If yes, break out with True and leave the recursion
            if self.parseDFSAux(form,string,maxDepth-1): return True

        # Didn't find anything, so that's a blind alley
        return False

    # The mighty parser Part 2: Breadth First Search version
    # Step 1: Wrap the start symbol into a Sentential Form, initialize queue and jump in
    def parseBFS(self,string):
        self.queue = [SententialForm(self.start)]

        # Anything left to check?
        while self.queue:
            # Extract first (head) element from queue (dequeue)
            current = self.queue.pop(0)
            print(current)

            # Check for success
            if current.string == string: return True    # Got it! Get outa here...

            # Compute all possible next steps and place them at end of queue (enqueue)
            self.queue.extend(current.possibleDerivs(self.rules))

        # Didn't find anything, queue empty - think about this for a moment!
        return False
```

In [6]:
```
cfg = CFG('S',[])
#cfg.addRule(Rule('S','aSa'))
```

```
#cfg.addRule(Rule('S','bSb'))
cfg.addRule(Rule('S','aSb'))
cfg.addRule(Rule('S','c'))
print(cfg.rules)
print(cfg.parseDFS('aaaaaaacbbbbbbb',8))
cfg.parseBFS('aaaaaaacbbbbbbb')
#cfg.parseBFS('aabcbaa')
```

```
[S --> aSb, S --> c]
S ==> aSb
aSb ==> aaSbb
aaSbb ==> aaaSbbb
aaaSbbb ==> aaaaSbbbb
aaaaSbbbb ==> aaaaaSbbbbb
aaaaaSbbbbb ==> aaaaaaSbbbbbb
aaaaaaSbbbbbb ==> aaaaaaaSbbbbbbb
aaaaaaaSbbbbbbb ==> aaaaaaaaSbbbbbbbb
aaaaaaaSbbbbbbb ==> aaaaaaacbbbbbbb
True
S
aSb
c
aaSbb
acb
aaaSbbb
aacbb
aaaaSbbbb
aaacbbb
aaaaaSbbbbb
aaaacbbbb
aaaaaaSbbbbbb
aaaaacbbbbb
aaaaaaaSbbbbbbb
aaaaaacbbbbbb
aaaaaaaaSbbbbbbbb
aaaaaaacbbbbbbb
True
```

Out[6]:

In [ ]: