```
class CFGParser:
    def __init__(self, grammar):
        self.grammar = grammar

    def parse(self, tokens, rule='S'):
        if not tokens and not rule:
            return True
        if not rule or not tokens:
            return False
        for production in self.grammar.get(rule, []):
            if self.match(tokens, production):
                return True
        return False

    def match(self, tokens, production):
        if not production:
            return not tokens
        if production[0] in self.grammar:
            return self.parse(tokens, production[0]) and self.match(tokens[1:], prc
        return tokens and tokens[0] == production[0] and self.match(tokens[1:], prc

grammar = {
    'S': [['NP', 'VP']],
    'NP': [['D', 'N']],
    'VP': [['V', 'NP']],
    'D': [['the']],
    'N': [['dog']],
    'V': [['sees']]
}

tokens = ['the', 'dog', 'sees']
parser = CFGParser(grammar)
print(parser.parse(tokens))
```

```
False
```

```
class EarleyParser:
    def __init__(self, grammar):
        self.grammar = grammar

    def parse(self, sentence):
        # Initialize chart with lists for each position in the sentence
        chart = [[] for _ in range(len(sentence) + 1)]

        # Initialize Earley items for starting rule 'S' -> . NP VP
        self.predict(chart, 0, ('S', 'NP', 'VP'))

        # Process each position in the sentence
        for i in range(len(sentence) + 1):
            while True:
                added = False
```

```
                # Scan
                if i < len(sentence):
                    for state in chart[i]:
                        if not state.completed() and state.next() == sentence[i]:
                            new_state = state.advance()
                            self.add_state(chart, i + 1, new_state)
                            added = True

                # Complete
                for state in chart[i]:
                    if not state.completed():
                        continue
                    for completed_state in chart[state.origin]:
                        if completed_state.next() == state.name() and not completec
                            new_state = completed_state.advance()
                            self.add_state(chart, i, new_state)
                            added = True

                # Predict
                if not added:
                    break

        # Check if the goal state 'S -> NP VP .' is in the chart for the full sente
        goal_state = ('S', 'NP', 'VP')
        return any(state == EarleyItem('S', ('NP', 'VP'), 0) for state in chart[ler

    def add_state(self, chart, index, state):
        if state not in chart[index]:
            chart[index].append(state)

    def predict(self, chart, index, rule):
        lhs, rhs = rule[0], rule[1:]
        for production in self.grammar.get(lhs, []):
            self.add_state(chart, index, EarleyItem(lhs, production, index))

class EarleyItem:
    def __init__(self, name, production, origin):
        self.name = name
        self.production = production
        self.origin = origin
        self.current = 0

    def next(self):
        return self.production[self.current]

    def advance(self):
        new_state = EarleyItem(self.name, self.production, self.origin)
        new_state.current += 1
        return new_state

    def completed(self):
        return self.current == len(self.production)

    def __eq__(self, other):
        return self.name == other.name and self.production == other.production and
```

```
    def __str__(self):
        production_str = ' '.join(self.production[:self.current] + ['•'] + self.pro
        return f'{self.name} -> {production_str} [{self.origin}]'

# Example usage
grammar = {
    'S': [('NP', 'VP')],
    'NP': [('Det', 'N')],
    'VP': [('V', 'NP')],
    'Det': ['the', 'a'],
    'N': ['dog', 'cat'],
    'V': ['chases', 'sees']
}

parser = EarleyParser(grammar)
sentence = ['the', 'dog', 'chases', 'a', 'cat']
result = parser.parse(sentence)

if result:
    print("Sentence is grammatically correct.")
else:
    print("Sentence is not grammatically correct.")
```

```
Sentence is not grammatically correct.
```

```
class ParseTreeGenerator:
    def __init__(self, grammar):
        self.grammar = grammar

    def parse(self, tokens, rule='S'):
        tree = self.match(tokens, rule)
        if tree:
            return tree
        return "Invalid Sentence"

    def match(self, tokens, rule):
        if not tokens and not rule:
            return None
        for production in self.grammar.get(rule, []):
            subtree = []
            remaining_tokens = tokens
            for symbol in production:
                if symbol in self.grammar:
                    sub_tree_result = self.match(remaining_tokens, symbol)
                    if sub_tree_result:
                        subtree.append((symbol, sub_tree_result))
                        remaining_tokens = remaining_tokens[len(sub_tree_result):]
                    else:
                        break
                else:
                    if remaining_tokens and remaining_tokens[0] == symbol:
                        subtree.append((symbol, [symbol]))
                        remaining_tokens = remaining_tokens[1:]
```

```
                else:
                    break
            if len(subtree) == len(production):
                return subtree
        return None

grammar = {
    'S': [['NP', 'VP']],
    'NP': [['D', 'N']],
    'VP': [['V', 'NP']],
    'D': [['the']],
    'N': [['dog']],
    'V': [['sees']]
}

tokens = ['the', 'dog', 'sees']
parser = ParseTreeGenerator(grammar)
tree = parser.parse(tokens)
print(tree)
```

```
Invalid Sentence
```

```
class AgreementChecker:
    def __init__(self, grammar, agreement_rules):
        self.grammar = grammar
        self.agreement_rules = agreement_rules

    def check_agreement(self, tree):
        for rule in self.agreement_rules:
            if not rule(tree):
                return False
        return True

def subject_verb_agreement(tree):
    # Implement specific rules for subject-verb agreement
    return True

agreement_rules = [subject_verb_agreement]
checker = AgreementChecker(grammar, agreement_rules)
print(checker.check_agreement(tree))
```

```
True
```

```
import math

class PCFGParser:
    def __init__(self, grammar, probabilities):
        self.grammar = grammar
        self.probabilities = probabilities

    def parse(self, tokens):
        return self._parse(tokens, 'S')
```

```
    def _parse(self, tokens, rule):
        if not tokens and not rule:
            return 1
        best_prob = 0
        for production in self.grammar.get(rule, []):
            prob = self.probabilities.get((rule, tuple(production)), 0)
            remaining_tokens = tokens
            for symbol in production:
                if symbol in self.grammar:
                    prob *= self._parse(remaining_tokens, symbol)
                    remaining_tokens = remaining_tokens[1:]
                else:
                    if remaining_tokens and remaining_tokens[0] == symbol:
                        remaining_tokens = remaining_tokens[1:]
                    else:
                        prob = 0
                        break
            best_prob = max(best_prob, prob)
        return best_prob

grammar = {
    'S': [['NP', 'VP']],
    'NP': [['D', 'N']],
    'VP': [['V', 'NP']],
    'D': [['the']],
    'N': [['dog']],
    'V': [['sees']]
}

probabilities = {
    ('S', ('NP', 'VP')): 0.9,
    ('NP', ('D', 'N')): 0.6,
    ('VP', ('V', 'NP')): 0.8,
    ('D', ('the',)): 1.0,
    ('N', ('dog',)): 0.5,
    ('V', ('sees',)): 0.7
}

parser = PCFGParser(grammar, probabilities)
tokens = ['the', 'dog', 'sees']
print(parser.parse(tokens))
```

```
0
```