CSCI 3415 – Principles of Programming Languages
Fall 2021 – Dr. Doug Williams
Program 1 – Leftmost Derivation (Python)
Algorithm Description

In Program 1, you are to write a Python program to read in a grammar and compute the leftmost derivations of sentences – if they exist. You were provided a hint program, `python-hint.py`, that reads the grammar file. This paper discusses the algorithm to compute the leftmost derivations.

Grammars

Consider the grammar file example_3.1.txt.

```
<program> -> begin <stmt_list> end
<stmt_list> -> <stmt>
             | <stmt> ; <stmt_list>
<stmt> -> <var> = <expression>
<var> -> A | B | C
<expression> -> <var> + <var>
              | <var> - <var>
              | <var>
```

The `derive-hint.py` program reads this file and produce the following structure as a result.

```
[('<program>', ['begin', '<stmt_list>', 'end']),
 ('<stmt_list>', ['<stmt>']),
 ('<stmt_list>', ['<stmt>', ';', '<stmt_list>']),
 ('<stmt>', ['<var>', '=', '<expression>']),
 ('<var>', ['A']),
 ('<var>', ['B']),
 ('<var>', ['C']),
 ('<expression>', ['<var>', '+', '<var>']),
 ('<expression>', ['<var>', '-', '<var>']),
 ('<expression>', ['<var>'])]
```

This is a list (or vector) of tuples where each tuple represents a rule in the grammar. The first element in each tuple is a string containing a nonterminal and the second element is a list of strings representing a sentential form.

Sentences

A sentence (or any sentential form) is represented by a list of strings typically created by using the `split` method. For example, the sentence `'begin A = B + C end'` is represented by the list:

```
['begin', 'A', '=', 'B', '+', 'C', 'end']
```

It is important for each lexeme be surrounded by whitespace for the `split` method to work properly and the grammar is case sensitive.

I use the following code to read sentences and print the leftmost derivations in the main program.

```
    # Read sentences from standard input until end of file and print
    # their leftmost derivations.
    while True:
        print('---')
        try:
            sentence_string = input('Enter a sentence:\n')
        except EOFError:
            sys.exit()
        sentence = sentence_string.split()
        derivation = leftmost_derivation(grammar, [start], sentence, 0)
        print(derivation)
.
        print(' '.join(sentence))
        print('Derivation:')
        print_derivation(grammar, derivation)
```

The start symbol is the LHS (left hand side) of the first rule in the grammar and can be retrieved as:

```
    start = grammar[0][0]
```

Leftmost Derivation Algorithm

The leftmost derivation algorithm is a recursive depth-first search algorithm that starts with a sentential form that is just the start symbol, `['<program>']` in the above grammar and generates successive sentential forms by expanding the leftmost nonterminal until a sentence is generated. This continues until the desired sentence is generated or the search space is exhausted in which case the sentence is not in the grammar.

The basic structure of a recursive depth-first search is:

```
procedure DFS(G, v) is
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G, w)
```

However, rather than being given an explicit graph, *G*, we are given a grammar that allows us to generate successors. The `applicable_rules` function returns a list of the rules in the grammar for a given nonterminal.

```
    def applicable_rules(grammar, nonterminal):
        """ Return a list of grammar rules for nonterminal. """
        return list(filter(lambda rule: rule[0] == nonterminal, grammar))
```

The `filter` function iterates over the `grammar` and returns the rules whose LHS matches the given nonterminal. The `list` function is needed to return the applicable rules as a list.

An important function we need is one to match a sentential form against a sentence to determine if the sentential form is part of a (potential) derivation of the sentence. It does this by (conceptually) finding the prefix of the sentential form that are terminal symbols matching the beginning of the sentence, the leftmost nonterminal (if one exists), and the suffix that is the remainder of the sentential form following the leftmost nonterminal.

For example, consider the grammar above, the sentence

```
['begin', 'A', '=', 'B', '+', 'C', 'end'],
```

and the sentential form

```
['begin', 'A', '=', '<expression>', 'end'].
```

The prefix is `['begin', 'A', '=']`, the leftmost nonterminal is `'<expression>'`, and the suffix is `['end']`. Our function `match_form` will return an integer that is the length of the prefix or -1 if no prefix exists. For a nonnegative result, the prefix is `form[:match]`, that is the first `match` items of the sentential form.

```
    def match_form(form, sentence):
        """ Find the longest prefix of form matching sentence. """
        for i, lex in enumerate(form):
            if i == len(sentence):
                return -1
            if isnonterminal(lex):
                return i
            if lex != sentence[i]:
                return -1
        return len(sentence) if len(sentence) == len(form) else -1
```

If the length of the prefix is the same as the length of the form then we have matched the entire sentence and otherwise our generated sentence is too short to match the given sentence. Otherwise, the length of the prefix must be less than the length of the sentential form and the match value is the index of the leftmost nonterminal and the suffix is `form[match+1:]`.

We can create the successor sentential form by applying a `rule` to a `form` given a `match`.

```
    def subst(rule, form, match):
        return form[:match] + rule[1] + form[match+1:]
```

Finally, we need the framework for the

```
    # Recursively search for forms matching the sentence.
    match = match_form(form, sentence)
    if match == -1:
        return None
    if match == len(sentence):
        return []
    for rule in applicable_rules(grammar, form[match]):
        . . .
    return None
```

The ellipsed code, which you must provide, must create the new form by applying the rule (using `subst`), calling `leftmost_derivation` recursively, and returning the derivation when the sentence is found. `None` is returned if the sentence is not found (in this recursive level).

At each recursive level, the derivation is formed by prepending its sentential form with that returned by the recursive call to `leftmost_derivation`. Note that the level that found the sentence returns `[]` to denote success.