Generation of Left-Hand Derivations

Purpose

The purpose of my program is for the user to be able to enter an arbitrary string and evaluate whether the string is a member of the language generated by a given context-free grammar. If the string is a member of the context-free grammar's language, a left-hand derivation of that string is given.

Usage

My program is implemented in Python (tested with version 3.3.1) and can executed from a command line or terminal using the following syntax:

```
python Derive.py INPUT
```

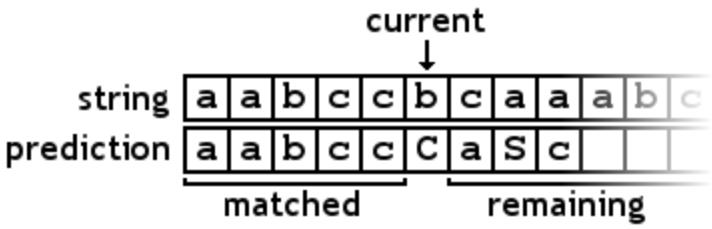
where `INPUT` is an optional command line argument for the filename of the file containing the context-free grammar. If `INPUT` is not given, a filename is asked for inside of the program until the filename of a file containing a properly formatted context-free grammar is given. I have provided several example grammar files with my program. The context-free grammar within the plain text file is to be formatted in Backus-Naur Form. The usage of Backus-Naur Form ensures unambiguity between terminals and nonterminals and also conforms to an already widely used standard for grammar notation. When the grammar has been read by the program, it is displayed in a readable format.

The user is then asked to input a string to be derived by the current grammar and a separator for the terminals of the string if the string is longer than a single character. If the string is derivable by the current grammar, a left-hand derivation will be given. Otherwise, the user will be informed that the given string cannot be derived using the rules of the current grammar. During any time in the usage of the program, the user can exit simply by pressing Enter at any of the prompts.

Implementation

In order to represent a context-free grammar within my program, I took an object-oriented approach by creating classes for terminal and nonterminal symbols, production rules, and the context-free grammar itself. Each of these components build upon one another in a way that makes implementation much less technically confusing. A context-free grammar is represented in my program just as it is formally, consisting of four sets: N (nonterminals), Σ (terminals), P (production rules), and S (the start symbol).

The grammar parsing technique I implemented with this program is a top-down, depth-first recursive algorithm. The recursive algorithm takes as arguments: the string to be derived, the current predicted string, a counter to track where the current derivation is taking place, a recursive counter for curtailment of infinite left recursive loops, and the list of sentential forms which make up the left-hand derivation.



As the algorithm progresses through the string, rules are derived by replacing nonterminals with right-hand sides of their rules. My algorithm uses backtracking through recursion to terminate derivation for invalid predictions. When the prediction matches the original input string, the algorithm terminates derivation and adds sentential forms (the valid predictions) to the left-hand derivation as it travels up the recursive call stack.

Left Recursion

The main problem faced in the implementation of my parser is one that is inherent to naive top-down parsers: left recursion. All that is needed to solve this problem is to count how many times a left-recursive rule is derived and then curtail parsing for that prediction when this count exceeds the length of the input string. Identification of these rules is the main problem.

Recursive rules create “loops” that the parser can become stuck in. These loops are trivial to identify in direct left-recursive rules, where all that needs to be checked is if a rule’s left hand nonterminal is equal to the first symbol of its right-hand side. With indirect left recursion, these loops can consist of multiple rules, eventually leading back to the rule that started the loop. In my program, I implemented an algorithm that marks the start rules of these recursive loops as recursive so that the parsing algorithm will curtail derivation for indirect left recursion just as it would with direct left recursion.

One problem that curtailment introduces is that parsing becomes exponentially inefficient when left recursive rules are used extensively to derive the input string. Regardless, the use of curtailment ensures a robust method for deriving strings from context-free grammars of any form. For the most efficient parsing, it is recommended to not have any left recursive rules, direct or indirect, in the rules of the input grammar.