Grammar Parser

Hriscu Octavia & Hutupasu Stefana - 933/2

Link to git repo

Grammar

Implementation:

- A class that manages reading a grammar from a given file and printing found non-terminals, terminals, productions and start symbols; it also implements a function to check whether or not the given grammar is context-free

Operations:

- **parse_line(line)**: Takes one parameter, line, which should be a string. Returns a list of values that are stripped from the line.
- **parse_productions(rules)**: Takes one parameter, rules, which should be a list of strings. Returns a dictionary with the left-hand side of the rule as the key and the right-hand side of the rule as a list of tuples containing the rule value and index.
- **from_file(fileName):** This function reads a text file containing the necessary information to create a Grammar object. It reads the first two lines of the file to create the nonterminals and terminals of the grammar, the third line to create the start symbol, and the rest of the file to create the productions of the grammar. It then returns a Grammar object using these values.
- **get_productions_for_nonterm(self, nonterm)**: This function takes a Grammar object (self) and a nonterminal symbol (nonterm) as arguments and returns a list of all the productions associated with the given nonterminal symbol. - If the given symbol is not a nonterminal symbol, an error will be raised.
- **isCFG(self)**: This function takes a Grammar object (self) as an argument and returns a boolean value. - This function will check the Grammar object for the following properties: (1) all nonterminal symbols have at least one associated production, (2) all keys in the productions dictionary are single-character strings. - If the Grammar object satisfies both of these properties, the function will return True. Otherwise, it will return False.

Parser

- **read_sequence(sequence_file):**
  - Parameters: sequence_file (str): A file containing a sequence
  - Returns: sequence (list): A list of elements from the sequence file
  - This function reads a sequence from a given file and returns a list of elements from the sequence. If the file is "PIF.out", the elements are read from the first column of the file. Otherwise, the elements are read from the first character of each line of the file.
- **write_out(message, final=False)**
  - Writes a given message to the output file. If the optional argument final is True, the message will be preceded by a delimiter.
- **write_all_data()**
  - Writes all data about the current state to the output file. Includes the state, current position, working stack, and input stack.
- **expand(self)**
  - The expand function takes an object as an argument, pops the first element from the input stack, appends the popped element to the working stack and adds the new production to the input stack.
- **advance(self)**
  - This method advances the current position by one, adds the next item in the input stack to the working stack, and prints out the new stack.
- **momentary_insuccess(self)**
  - Updates the current state to 'b'
- **back(self)**
  - Removes the last object in the working stack, appends it to the input_stack, and updates the current position
- **success(self)**
  - This function sets the state to ‘f’
- **another_try(self)**
  - changes the state of the parser to "q".
  - It pops the last item from the working stack and checks if the production number is less than the number of productions for the non-terminal.
  - If it is, it appends a new tuple to the working stack and changes the production on the top of the input stack.
  - If the index is 0 and the last item is the start symbol, the state is changed to "e".

    - Otherwise, the last production is removed from the input stack and replaced with the last non-terminal.
- **run(self, sequence)**
    - 1. The function checks if the state is not equal to 'f' or 'e'
    - 2. The function write all data to the output file
    - 3. If the state is equal to 'q', it checks if the input stack is empty and if the current position is equal to the length of the input string
    - 4. If the input stack is empty and the current position is equal to the length of the input string, the success function is called
    - 5. If the input stack is empty but the current position is not equal to the length of the input string, the momentary insuccess function is called
    - 6. If the input stack is not empty and its head is a non terminal, the expand function is called
    - 7. If the index is less than the length of the input string and the head of the input stack is equal to the current symbol from the input, the advance function is called
    - 8. If none of the previous conditions is true, the momentary insuccess function is called
    - 9. If the state is equal to 'b', it checks if the working stack's last element is a terminal
- **create_parsing_tree(self)'**
    - Create an empty list called tree.
    - Iterate through the working stack
    - If the item in the working stack is a tuple, create a new Node object with the first element of the tuple as the node's value and append it to the tree list and sets the production rule to the second value of the tuple
    - Otherwise, create a new Node object with the item as the node's value and append it to the tree list
    - Set the father of the Node to the index of the item in the working stack
    - If the item in the working stack is a tuple, compute the length of the production of the non-terminal
    - Create a vector of indexes corresponding to the length of the production
    - For each item in the vector of indexes, if the corresponding tree node has a production, compute the length of its depth
    - Add the computed length to the vector of indexes
    - For each item in the vector of indexes, set the sibling of the corresponding tree node to the next item in the vector of indexes