https://github.com/913-Florian-Vlad/LFTC/tree/main/lab7

Recursive Descendant Parsing

Recursive Descendant Parsing is a top-down parsing technique that starts with the topmost grammar rule and recursively explores the production rules to match the input string(sequence).

Class Structure

RecursiveDescendant

Attributes

Grammar (__grammar): Represents the grammar to be parsed, provided as an instance of the Grammar class.

Current State (__current_state): Tracks the current state of the parsing process: 'q' for normal, 'b' for backtracking, 'f' for final success, and 'e' for error.

Index Position (__index_position): Indicates the current position in the input string(sequence).

Working Stack (__working_stack): Maintains a stack of symbols being processed during parsing.

Input Stack (__input_stack): Represents the input string as a stack of symbols.

Methods

expand()

Expands the current state by moving the input stack's first element (non-terminal) to the working stack, along with its first production to the input stack.

advance()

Advances the current state by moving the input stack's first element (terminal) to the working stack and increments the index position.

momentary_insuccess()

Modifies the current parsing state to the back state ('b').

back()

Backtracks the current state by popping the working stack's last element and moving it back to the input stack.

another_try()

Modifies the current parsing state based on the non-terminal's productions:

If more productions exist, the state is set to normal, and the next production is moved to the input stack.

If no more productions exist:

• If the non-terminal is the start symbol and the index position is 1, the state is set to error.

• Otherwise, the non-terminal is moved back to the input stack, and the state remains in the back state.

success()

Sets the current state to final success ('f').

get_grammar()

Return the Grammar object associated with the RecursiveDescendant instance.

get_working_stack()

Get the working stack of symbols.

get_output_file()

Get the output file name.

__initialize_configuration()

Initialize the configuration by resetting the parsing state, index position and stacks.

__initialize_output_file()

Initialize the output file by clearing its content.

__read_sequence_from_file()

Read the input sequence from the specified file and returns a list of symbols in the input sequence. Raises exception if any element in the sequence is not a terminal.

parse()

Parse the input sequence using the Recursive Descendant Parsing strategy and the

corresponding methods described above. It was developed using the course approach notes.

Iterates through the following conditions based on the current parsing state:

1. When in state ‘q’:

• Checks if the index position is at the end of the input sequence, and the input stack is empty, indicating successful parsing.

• Expands the non-terminal if the current symbol on the input stack is a non-terminal.

• Advances the index position if the current symbol on the input stack is a terminal, matching it with the corresponding symbol in the input sequence.

• If the terminal symbols do not match, it transitions to the momentary insuccess state.

2. When in state ‘b’:

• Backtracks by popping the last symbol from the working stack and moving it back to the input stack if the last symbol is a terminal.

• Initiates another try by calling the another_try method if the last symbol is a non-terminal.

3. When in state ‘f’ or ‘e’:

• Exits the loop.

If the parsing state is 'e' (error), prints a message indicating that the sequence is not accepted.

If the parsing state is 'f' (final success), generates a table representation of the parsing tree using the get_tree_table_representation method from the ParserOutput class, and prints a message indicating the sequence is accepted, along with the table representation.

Reinitializes the parsing configuration using the __initialize_configuration method for potential subsequent parsing attempts.

str()

Return a string representation of the current parsing state.

Parsing Tree and Output Generation Documentation

Classes

1. Node Class

The Node class represents a node in the parsing tree. Each node has the following attributes:

parent: The parent node in the tree.

symbol: The symbol associated with the node.

children: A list of child nodes.

number: An index number assigned to the node in the tree, used to complete the parent-sibling table representation.

2. Position Class

The Position class is a simple class to manage a position value. It has methods to get the current value and increment it, and it was created as an auxiliary variable used to be modified in the recursive approach of building the tree.

3. ParserOutput Class

The ParserOutput class is responsible for generating the parsing tree and creating a table representation. It has the following methods:

__create_parsing_tree

This private method generates the parsing tree based on the parser's grammar and working stack. It uses depth-first search to traverse the tree and create nodes for non-terminal symbols. It results in an auxiliary representation, a default parent-child tree, which will be further used for the parent-sibling representation.

__create_parsing_tree_table_representation

This private method creates a table representation of the parsing tree, based on the parent-child tree obtained by using the “__create_parsing_tree” method. It assigns index numbers to nodes and records information such as the symbol, parent, and right sibling in a table, using a breath first search implementation.

get_tree_table_representation

This method returns the table representation of the parsing tree as a formatted string using the

Texttable library.

print_string_to_file

This method appends a given string to the output file specified by the parser.

print_string_to_file_and_console

This method prints a string to both the console and the output file.