Github: https://github.com/AidaCristea/FLCDPair

Documentation Parser

1. Overview

The RecursiveDescendentParser class is designed to implement a Recursive Descent Parser for a given grammar. It recognizes and parses sequences based on the grammar rules and generates a parse table using a recursive descent parsing technique.

1. Menu options

* Show non-terminals (1): This option prints out the set of non-terminals present in the grammar.
* Show terminals (2): This option displays the set of terminals in the grammar.
* Show productions (3): It prints the set of productions defined in the grammar.
* Show starting symbol (4): This option reveals the starting symbol specified in the grammar.
* Production for given non-terminal (5): Allows the user to input a non-terminal and displays the productions associated with that non-terminal.
* CFG check (6): Checks if the grammar is a Context-Free Grammar (CFG) and prints the result.
* Recursive Descendent Parser (7): Initiates the execution of the Recursive Descendent Parser, using the provided sequence file generated by the Scanner.
* Exit (0): Exits the program.

1. Initial configuration Recursive Descendent Parser

* Input String (w): The input string is read from the specified sequence file (seq) and stored in the list w.
* State (s): The parser starts in the state q, indicating that it is in the parsing process.
* Input Index (i): The input index i is set to 1, indicating the position in the input string that the parser is currently processing.
* Alpha Stack (alpha): The alpha stack is initially an empty list. It will be used to store symbols that are derived during the parsing process.
* Beta Stack (beta): The beta stack is initialized with the starting symbol of the grammar. For example, if the starting symbol is S, then beta will contain [S].
* Grammar (grammar): The grammar object holds the grammar rules and productions required for parsing.
* Parse Table (table): The table is an instance of the Table class, initially empty, which will be used to record parsing actions.

1. Recursive Descendent Parser

The parsing process involves the following steps:

1. Initialization: Setup of initial configuration and working stacks.
2. Parsing Loop: Continuously processes the input until the sequence is either successfully parsed or an error occurs.
3. Success: Indicates successful parsing when the sequence is fully processed.

private void Success(): Sets the state s to "f" indicating successful parsing.

Configuration: (q,n+1, 𝜶, 𝜀) ⊢ (f,n+1, 𝜶, 𝜀)

1. Backtracking: Handles backtracking when the parser encounters a situation that requires it.

private void Back(): Performs backtracking by decrementing the input index i and adjusting the working stack (alpha and beta).

Configuration: (b,i, 𝜶a, 𝜷) ⊢ (b,i-1, 𝜶, a𝜷)

When: head of working stack is a terminal

1. Momentary Insuccess: Marks a momentary failure during parsing, triggering backtracking.

private void MomentaryInsuccess(): Marks a momentary failure during parsing, setting the state s to "b."

Configuration: (q,i, 𝜶, ai𝜷) ⊢ (b,i, 𝜶, ai𝜷)

When: head of input stack is a terminal ≠ current symbol from input

1. Advance: Advances the parser to the next symbol in the input sequence.

private void Advance(): Advances the parser to the next symbol in the input sequence by incrementing the input index i and adjusting the working stack (alpha and beta).

Configuration: (q,i, 𝜶, ai𝜷) ⊢ (q,i+1, 𝜶ai, 𝜷)

When: head of input stack is a terminal = current symbol from input

1. Expand: Expands the working stack (beta) when the head of the stack is a non-terminal.

private void Expand(): Expands the working stack (beta) when the head of the stack is a non-terminal. This involves adding the productions of the non-terminal to the working stack.

Configuration: (q,i, 𝜶, A𝜷) ⊢ (q,i, 𝜶A1, 𝜸1𝜷)

Where: A → 𝜸1 | 𝜸2 | … represents the productions corresponding to A

1 = first prod of A

When: head of input stack is a nonterminal

1. Another Try: Attempts an alternative production when backtracking is not possible.

private void AnotherTry(): Attempts an alternative production when backtracking is not possible. This method handles the case when the current production cannot be applied, and the parser needs to try an alternative.

Configuration: (b,i, 𝜶 Aj, 𝜸j 𝜷) ⊢ (q,i, 𝜶Aj+1, 𝜸j+1𝜷) , if ∃ A → 𝜸j+1

(b,i, 𝜶, A𝜷), otherwise with the exception

(e,i, 𝜶,𝜷), if i=1, A =S, ERROR

When: head of working stack is a nonterminal

1. Generate Parse Table: If successful, prints the final configuration and generates a parse table.

private void generateTable(): Generates a parse table based on the parsing process. The table is a representation of the parsing actions taken during the parsing process.

The Table class represents the parse table used by the parser to record parsing actions.

Methods For Table:

public void add(String info, Integer parent, Integer leftSibling)

info: The information to be added to the table.

parent: The index of the parent node in the parse table.

leftSibling: The index of the left sibling node in the parse table.

Adds a new row to the parse table with the provided information, parent index, and left sibling index.

public List<TableRow> getTable()

Returns: A list of TableRow objects representing the parse table.

public int getCurrentIdx() => Returns the entire parse table.

Returns: The current index of the parse table.

Parser Table Generation:

1. Initialization:

Extract production numbers from the alpha list.

Set the starting symbol and initialize variables.

1. Iterate Through Productions:

For each production number, obtain the non-terminal and its index.

Retrieve the production associated with the non-terminal.

1. Process Production Elements:

Add elements to the parsing table, considering parent and left sibling indices.

Keep track of encountered non-terminals for potential expansion.

1. Update Indices:

Update the index for the next starting symbol based on non-terminals encountered.

Remove pairs from the list to mark non-terminals that will be expanded.

1. Return Parsing Table:

The constructed parsing table is returned.

1. Grammer input format files:
2. SET OF NON-TERMINALS: This section lists the non-terminal symbols in the grammar. Each non-terminal is separated by a comma (,).

Example:

SET OF NON-TERMINALS

program, decllist, declaration, listDeclaration, listInt, listString, cmpdstmt, stmtlist, stmt, ...

1. SET OF TERMINALS: This section lists the terminal symbols in the grammar. Each terminal is separated by a comma (,).

Example:

SET OF TERMINALS

{, }, ;, [, ], (, ), +, -, \*, /, %, read, write, if, else, whileLoop, forLoop, <, <=, ==, !=, >=, >, =, boolean, ...

1. PRODUCTIONS: This section defines the production rules of the grammar. Each production rule has the format non-terminal -> expansion. Alternatives in the expansion are separated by the pipe (|) symbol.

Example:

PRODUCTIONS

program -> decllist cmpdstmt

decllist -> declaration ; | declaration ; decllist

...

1. STARTING SYMBOL: This section specifies the non-terminal from which the derivation of the language begins.

Example:

STARTING SYMBOL

program