Compilers Phase II (Parser)

Names :

Ahmed Khaled Saad Seif 07

Ahmed Reda Amin Selim 09

Hisham Osama Ghazy 78

Overview:

- The task in this phase of the assignment is to design and implement an **LL (1) parser generator** tool. The parser generator expects an LL (1) grammar as input.
- We first enclosed the implementation of phase 1 in a class that can communicate with phase 2 through the function **next token ( )** that enables the communication between the two phases
- It should compute **First** and **Follow** sets and use them to construct a **predictive parsing table** for the grammar.
- The table is to be used to drive a **predictive top-down parser**. If the input grammar is not

  LL (1), an appropriate error message should be produced.
- The generated parser is required to produce some representation of the **leftmost derivation** for a correct input. If an error is encountered, a **panic-mode error** recovery routine is to be called

  to print an error message and to **resume parsing**.
- The parser generator is required to be tested using the given context free grammar of a small subset of Java. Of course, you have to modify the grammar to allow predictive parsing.

Used Data Structures:

- **MainController**: A class that controls the flow of the compiler and calls all modules respectively. (eg: Lexical Analyzer, Parser, etc.)
- **NonTerminal**: A data structure used to include data about the non-terminals , this is an implemented data structure , created to accommodate many of the requirements of the data we are dealing with , it carries the first and follow sets of each non-terminal .
- **Pair<NonTerminal,String>**: The main building unit of the work we are doing and it is used to represent both terminals and non terminals , it is also used as the main vessel of communication between the different classes .
- **Vectors**: Used extensively during the phase to represent various containers and to carry data structures of interest , like the representation of the parsing grammar.
- **Map**: Used to represent the parsing table as a key value pair .
- **Stack**: Used in the stack matching .

Algorithms:

- **First Algorithm:** Firstly, we have constructed a function for the computation of the first-set of some non-terminal symbol, it looks up whether the first token is a terminal or

not. If that token was a terminal symbol then we add it directly to the first-set but if it is not we recurse and find the first-set of the current non-terminal. Finally, **Dynamic programming** is used while recursing to achieve memoization and avoid computing the first-set of some non-terminal symbol more than once.

- **Follow Algorithm :** The same idea of *First-Algorithm* was implemented in the computation of the follow-set but with some special conditions. To make the implementation of *Follow-algorithm* easy for us, we have made a new data structure that holds the set of next symbols that follow the non-terminal symbol we are computing its follow-set. Also we used dynamic programming to avoid recomputation of follow-sets we have already computed before.

- **Left Factoring :** The basic implementation of left factoring is used with the help of a utility algorithm of common prefix is used during the calculation to merge the number of productions having the same start symbols under the rule

  $A = \alpha X \mid \alpha Y \Rightarrow A = \alpha A' , A' = X \mid Y$

  The refactored production is pushed again to refactoring for a possibility of higher order of the left factoring

- **Eliminating left - recursion :** The basic implementation of eliminating non-immediate left recursion is used where we loop on every grammar rule and apply the rule of $A = A\,\alpha \mid \beta \Rightarrow A = \beta\,A'$, $A' = \alpha\,A' \mid \varepsilon$ is used for substitution.
  Also a substitution with the previously formed grammar is used to reveal any other recursion that also helped us in removing the possibility of finding cycles in the case of the first and the follow.

Used Tools :

All the steps were self-implemented , no used tools .

Work-Flow :

I. Lexical - Analyzer :

Reads the grammar and input file , matches lexemes and sends a reference to its output to the Parser.

II. Parser :

Step[1] : Reads the parsing grammar file .

Step[2] : Identifies and eliminates any left recursion or left factoring to make it LL(1) compatible .

Step[3] : Builds the first and follow sets of each non-Terminal based on the grammar .

Step[4] : Constructs the parsing table .

Step[5] : Reads tokens from the lexical analyzer via next_token() and uses the stack to match them .

Step[6] : Prints the output of the left derivation in real time as the stack matching is underway .

Assumptions:

No special assumptions are made , the process should align completely with the requirements stated in the assignment body , including accepting non LL(1) compliant grammars .