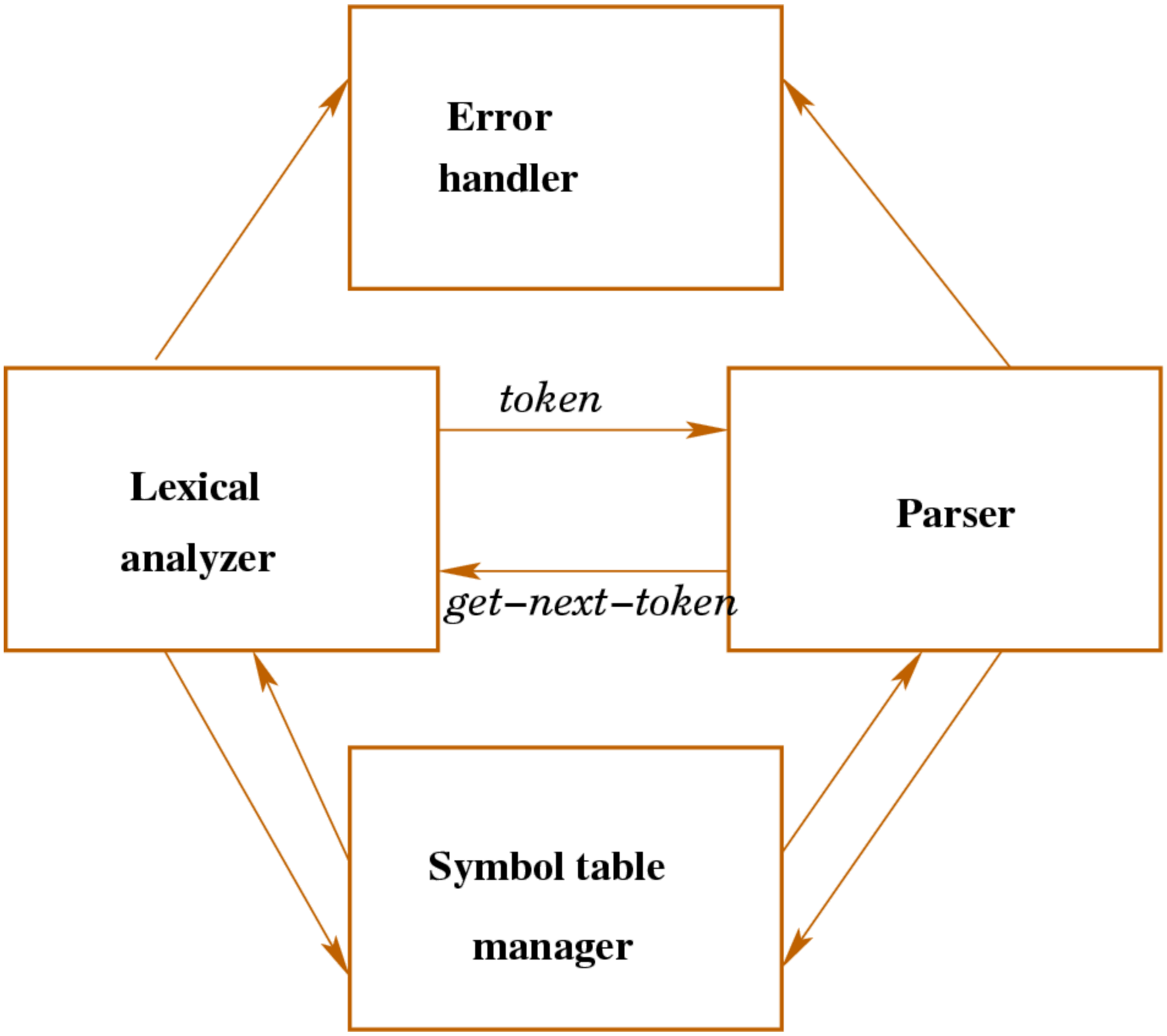
Compilers

Phase2: Parser Generator



Developed by:

1. **Rita Samir 26**
2. **Shereen El kordi 33**
3. **Merit Victor 69**
4. **Nancy Abd ElKarem 70**

Introduction

This phase aims to practice techniques for building automatic parser generator tools. It is about designing and implementing an LL (1) parser generator tool. The parser generator expects an LL (1) grammar as input. It should compute First and Follow sets and uses 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 tested using the given context free grammar of a small subset of Java. we combine the lexical analyzer generated in phase 1 and parser such that the lexical analyzer is to be called by the parser to find the next token.

Used Data Structures

We defined the following class models:

1. **Production.h**

   Represents the production in the grammar by its left hand side “nonterminal” and right hand side “set of terminals and nonterminals”

2. **ParserTable.h**

Represents the parser table, contains functions used to fill data in the table, get the content of specific cell, check if the cell is empty or not.

We used the following data structures:

1. **vector<Production>**

As data structure to hold all productions in the grammar. Each production is a class that its attribute the left hand side and the right hand side.

- String for LHS.
- Vector<String> for RHS contains strings of terminals and non terminals with specific order as the order of the context free grammar.

2. map<string, vector<vector<string>>>

As data structure represents all non terminals and its productions that goes to, the key is the nonterminal and the value is vector of productions the nonterminal goes to in the grammar, each production is vector of string.

3. set<string>

Used to hold the all terminals in the grammar.

4. map<string, set<string>>

As data structure to hold the first set or follow set of each nonterminal, the key is the nonterminal and the value is set of strings to represent the terminals.

5. map<string, map<string,vector<string>>>

As data structure to hold the parser table, the first key is nonterminal and the second key is the input/terminal and the value is the vector of strings to represent the production.

6. map<string, bool>

Used during the calculations of the follow set of each nonterminal to represent the nonterminal that we just get its follow set so add it to the map and set its bool value to true.

7. **map<string,vector<string>>**

   Used during the calculations of the follow set of each nonterminal, the key is the nonterminal and the value is the vector of all nonterminal that the key depends on. It's useful to prevent the occurrence of the cycles between the nonterminals while trying to compute their follow set.

8. **stack<string>**

   A stack of grammar symbols used for parsing the input file. It is initiated with a dollar sign and the start symbol.

9. **vector<string>**

   A vector holding the leftmost derivation output format that is required in the project. The vector is manipulated as well as the stack while keeping the required format that is then printed in a file.

Algorithms and Techniques Used

Regex for grammar parsing

Reading the CFG.txt the input file line by line and classify each line either a new production line or a remaining of previous production. after that sending every production line to clarify the components of this production and assign it to its data structure Vector<Production>.

Getting the LFS and RHS of each production and also dividing the long productions with many ors to multiple smaller productions without any ors so the RHS will be a vector of only terminals and non terminals with specific order.

First and follow sets calculation

We used the rules defined in the lecture to calculate the first and the follow set of each nonterminal in the grammar.

To calculate the first(X), we have have three cases of the start symbol in strings derived from x:

1. It's terminal or epsilon, so we simply add it to the first set of (X).
2. It's nonterminal, so we calculate its first set then add it to the first of (X).
3. It's nonterminal but after calculating its first set we found that its first set contains epsilon, so we add its first set to the first set of (X) except the epsilon. Then we go to to the following symbol and repeat this operation.

To calculate the follow (X), we have have three cases of the symbol that follows x in strings derived from the starting symbol:

1. It's terminal, so we simply add it the follow set of (X).
2. It's non terminal T, so we get the first set of this non terminal and it to the follow set of (X).
    - Before inserting the first set of (T) to follow set of (x), we check if it contains epsilon. If so, we need to calculate the follow set of (T) then add its follow set to the follow set of (X).

3. It's epsilon , so we get the follow set of its parent means:

Y -> aX is a production rule -----> everything in FOLLOW(Y) is in FOLLOW(X).

First, we need to check if X is the start symbol, then we add "$" to its follow set.

While calculating the follow set, we need to consider the dependencies between the nonterminals to prevent the occurrence of the cycle. We consider this case when getting follow(X) and it showed in the production rule as

Y -> a X or Y -> a X Z and epsilon in first(Z)

Then we need to calculate the follow(Y) to add it to follow(X) here X dependent on Y so we keep it in vector contains all the non terminals that X dependent on and continue calculating. After finishing we add the follow set of each non terminal exists in the vector that represents the dependencies to the follow set of the nonterminal that depends on them.

Constructing parser table

We loop through the nonterminals , for every non terminal we go through the list of expressions/productions and get the first of every production and add new input in the parser table with this expression. If the first contains epsilon then add epsilon to all follows of this nonterminal.

After That , we add “synch” to all the follows of each nonterminal which is useful in Panic-Mode Error Recovery.

The empty cells in the parser table marked as “error”.

Parsing input tokens

The stack and the derivation vector are both initialized by $ and the start symbol. The input vector ends with the dollar sign. The parsing then starts. The symbol at the top of stack and the input symbol determines the action that is to be taken. The following possibilities exists:

1. Top of stack (TOS) is a non terminal: the parser table is checked to get the production at the (parserTable[terminal])[non terminal]. TOS is then popped and the production rules is inserted backwards into the stack. Same thing happens to the printing vector.

2. A possibility is: a production may not be found and there is an error, empty cell, the panic mode recovery is then applied: an error is reported in the derivation file and the input symbol is discarded.
3. Another possibility is: production is not found but there is a synch, the input is one of the follows of the non terminal. An error is then reported and the non terminal symbol (TOS) is discarded (popped). The derivation vector is modified and rewritten.
4. The production may be an Epsilon: in that case the TOS is popped and the symbol is removed from the derivation vector.
5. TOS may be a terminal matching input symbol. They are matched and then removed from stack and input. The derivation vector pointer moves to the next symbol.
6. TOS is a terminal that doesn't match the input symbol: An error is reported and the the top of the stack is popped (it is considered as if the input is missing the symbol appearing in the grammar). That symbol is also deleted from the derivation file.
7. TOS and the input symbol are both $, the parsing is then completed successfully.

A check is made to see if the stack or the input are emptied before the other and an error is then produced.
A check is also made if the input contains a symbol that is not in the defined terminals and an error is produced.

Integration To Phase 1

The parser iterates over the tokens generated by the lexical analyzer that are sent line by line in case of declarations or send as a total expression like the while loop expression and the parser takes the vectors and parses them with the algorithm discussed.

Input Files

Lexical Rules Input File

Link: **Lexical Rules**

Grammar Rules Input File

Link: **Grammar Rules**

The resultant transition table for the minimal DFA

Link: **Minimized DFA Table**

The resultant parser table

Link: **Parser Table**

Test Program

Output file for the given test program

Assumptions

1. In the CFG.txt input file, there must be at least one space between every terminal or non terminal of the RHS of production.
2. Character ' cannot be considered as terminal or nonTerminal in any production.
3. Each line would be parsed starting from the beginning of the grammar not the previous line output.

Bonus

Elimination of left recursion

A grammar is left recursive if its first non terminal (variable) as the LHS non terminal so we checks for the presence of left recursion and if it is exist we eliminate it by replacing that production by some production as follow :

A -> Aa	**to**	A -> ß A’
A -> ß		A’ -> a A’
		A’ -> \L

Elimination of left factoring

A grammar is said to be left factored when the productions start with the same terminal (or set of terminals). On detecting the similarity of starting of the RHS we do left factoring step by step at each step we do factoring for the first element of the RHS of all the production that have the same first element, and we do more steps after that in the same way until we remove all the common part between the productions of the same LHS non terminal.

The left factoring will be as follow :

A -> aß1	**to**	A -> aA’
A -> aß2		A’ -> ß1
		A’ -> ß2