Compilers - Phase II Report

1. Enas Gaber
2. Dina Hossam
3. Monica Salama
4. Mostafa Gabriel

1. Description of used data structures

- CFG & Elimination of left recursion/factoring
  - Set<int> terminals;
  - Vector<pair <int, vector<vector<int> > > > nonTerminals;
  - Trie (Used to get the longest common prefix in eliminating left factoring)

- First, Follow & Parsing table:

  - vector<set<int> > follow:

    Each non terminal is mapped to an integer id .
    follow[id] → contains the corresponding id's of its follow terminals set (To avoid repetition)

  - map<int, vector<pair <int,int> >>mapInputProduction

    Mapping non terminal's ID with terminal's ID and production's ID.
  - map<int,vector<int> >mapProduction
    Mapping production's ID with production vector.
  - map<int,int>termCol
    Mapping terminal's ID with column number in parsing table.
  - Map <int,int>nonTermRow
    Mapping non terminal's ID with row number in parsing table.
    Vector <int, int > parsingtable
    Storing the transitions.

- Predictive parsing:

  - Stack <int> s;

    non recursive predictive parser can be built by maintaining a stack to add
    The terminals then pop it when a match found.

2. Algorithms and techniques used

- ## CFG & Elimination of left recursion/factoring

    - It reads the CFG file by line to parse each non terminal
    - For each non terminal, vector<vector<int> > is used to save its productions
    - Before adding this non terminal it eliminates the left recursion and left factoring from it
    - Eliminating left recursion:

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i-1$) {
4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions
7) }

    - Then it eliminate the left factoring problem by getting the longest common prefix among all the productions using a Trie

- ## First :

- Terminals:
    - Each terminal has ‘first’ as itself
- For non-terminals:
    - Starting from the last non-terminal
    - For each non-terminal :
        - For each production :
            - Starting from the first element of the production
            - If this element’s ‘first’ has not been calculated yet, recursively compute ‘first’ of this element
            - Add ‘first’ of this element to the first of the non terminal
            - While epsilon is in the ‘first’ of current element continue through the elements of the production.
            - If the end of the production is encountered add epsilon to the ‘first’ of the non-terminal

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.

2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \ldots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ, then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.

3. If $X \to \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

- Follow :

1. Place \$ in $\text{FOLLOW}(S)$, where S is the start symbol, and \$ is the input right endmarker.

2. If there is a production $A \to \alpha B \beta$, then everything in $\text{FIRST}(\beta)$ except ϵ is in $\text{FOLLOW}(B)$.

3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where $\text{FIRST}(\beta)$ contains ϵ, then everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$.

- Parsing Table:

```
Filling each cell in the table with -1;
For(each non terminal id ){
    vector<pair<int,int>> inputFirst = mapINputProduction[id];
    for(j : 0 -> inputFirs.size())
    {
       pair<int,int>termPro = inputFirst[j];
      Col = termCol[termPro.first];
     parseTable[i][col]=termPro.second;
    }
  Set s = follow of non terminal id;
  for(each terminal in the set )
  {
    Col = termCol[terminal];
    if(parsetable[row][col]=-1)
        {
```

```
            if(hasEps[nonterminalId])parsetable[i][col]= epsilon;
            Else parsetable[i][col]=sync;

          }

    }
}
```

- Predictive parsing:

```
set ip to point to the first symbol of w ;
set X to the top stack symbol;
while ( X =I=- $ )
 {
          /* stack is not empty * /
}
if ( X is a )
          pop the stack and advance ip;
else if ( X is a terminal )
          error ;
else if ( M [X, a] is an error entry )
          error ;
else if ( M[X, a] = X -t Y1 Y2 • .• Yk ) {
}
output the production X -t Y1 Y2 • • • Yk ;
pop the stack;
push Yk , Yk -l , ... , Y1 onto the stack, with Y1 on top;
set X to the top stack symbol;
}
```

Function parse: function to compare the elements in the stack with the input string
If matched or not, to know whether to pop from the stack or increment
The pointer of the input.
Function match: used to get the nonterminal and the terminal to access the parsing
Table to get the productions.
Function print: used to print the stack.

3. Parsing Tables if any

- Attached in the submission

4. Any assumptions made and their justification

- Predictive parsing:

If the input string have error from phase 1 (lexical analysis) skip this error and continue.