Compiler Project

Phase2: Parser Generator

Team members

Aya Lotfy

Salma Mohamed

Salma Yehia

Mohamed Gaber

Data Structure

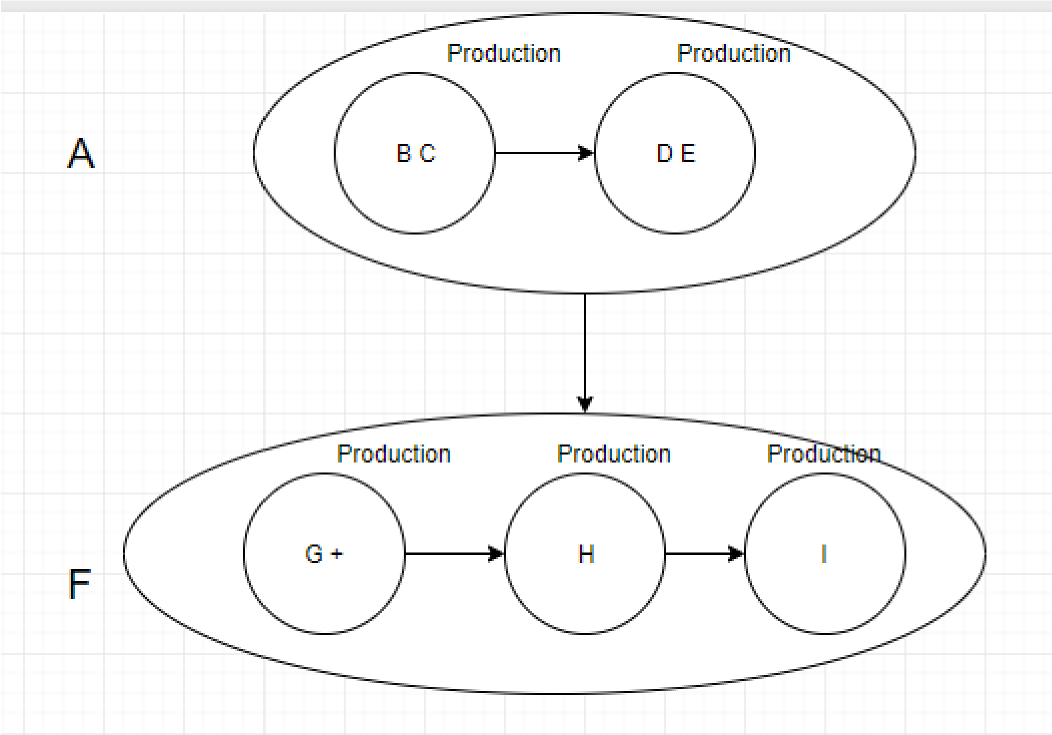
2D vector for grammar

To represent grammar scanned from grammerFile, we use 2D vector of production.

For example

#A = B C | D E

#F = G '+' | H | I



HashMap

HashMap is used to hold terminal and non-terminal names with their indexes.

So each terminal and non-terminal is easily represented by index.

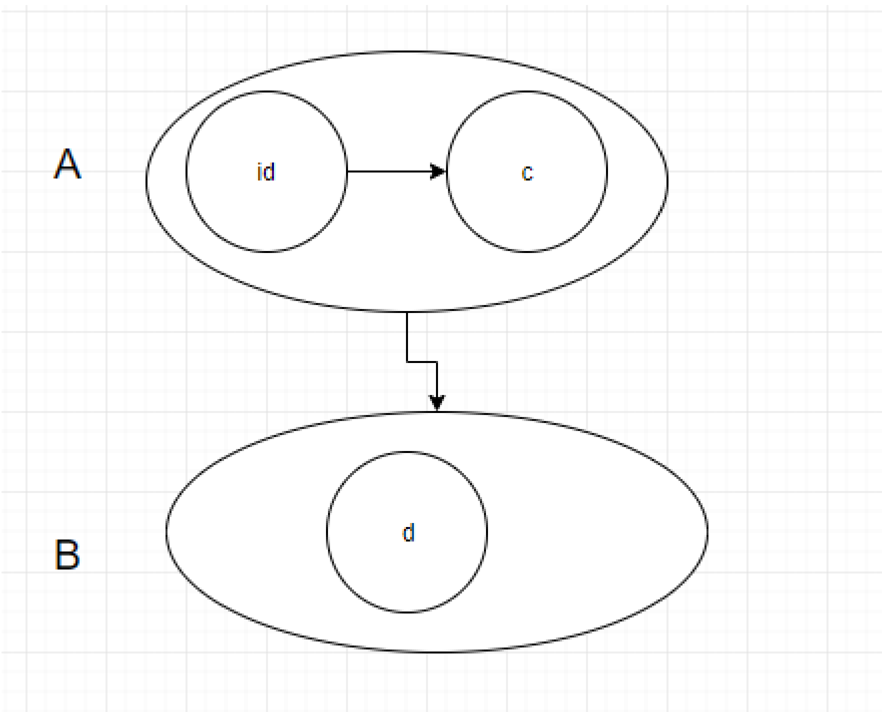
2D vector for first terminals

We use 2D vector of integers to hold the first terminal of each non terminal.

For example

#A = 'id' B | 'c'

#B = 'd' F



2D vector for follow terminals

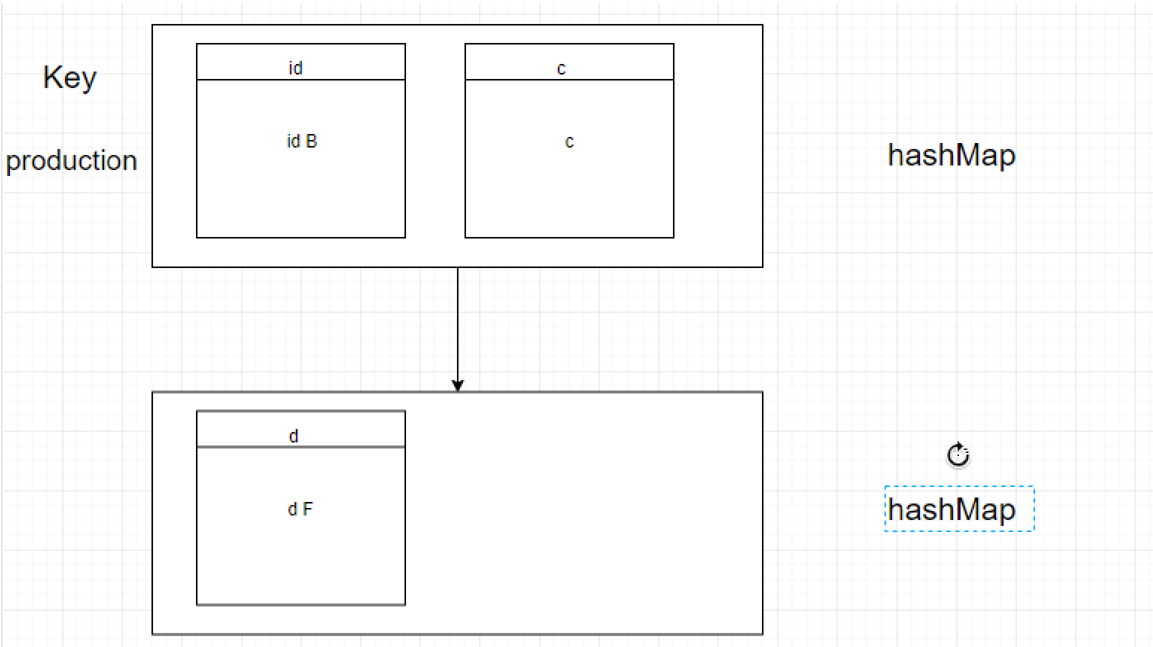
The same strategy as first terminals.

Vector of hashmap for parsing table

For example:

#A = 'id' B | 'c'

#B = 'd' F



Algorithms and Techniques

Detecting left recursion

For detecting left recursion, we used DFS traversal for the grammar so we could detect indirect left recursion.

First computation

For 'first' computation, we used algorithm described in lecture.

- If X is a terminal symbol ➔ FIRST(X)={X}
- If X is a non-terminal symbol and $X \rightarrow \varepsilon$ is a production rule
  ➔ ε is in FIRST(X).
- If X is a non-terminal symbol and $X \rightarrow Y_1Y_2..Y_n$ is a production rule
  ➔ if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then **a** is in FIRST(X).
  ➔ if ε is in all FIRST(Y_j) for j=1,...,n then ε is in FIRST(X).

Follow computation

For 'first' computation, we used algorithm described in lecture.

- If S is the start symbol ➔ \$ is in FOLLOW(S)

- if $A \rightarrow \alpha B \beta$ is a production rule
  ➔ everything in FIRST(β) is FOLLOW(B) except ε

- If ($A \rightarrow \alpha B$ is a production rule) or
  ($A \rightarrow \alpha B \beta$ is a production rule and ε is in FIRST(β))
  ➔ everything in FOLLOW(A) is in FOLLOW(B).

We apply these rules until nothing more can be added to any follow set.

Parsing table construction

For 'first' computation, we used algorithm described in lecture.

- for each production rule $A \rightarrow \alpha$ of a grammar G
  - for each terminal a in FIRST(α)
    ➔ add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α)
    ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α) and \$ in FOLLOW(A)
    ➔ add $A \rightarrow \alpha$ to M[A,\$]

- All other undefined entries of the parsing table are error entries.

Error handling

For matching error handling, we used panic-mode error recovery as it is descried in the lecture.

- So, a simple panic-mode error recovery for the LL(1) parsing:
  - For each nonterminal A, mark the entries M[A,a] as ***synch*** if a is in the follow set of A. So, for an empty entry, the input symbol is discarded. This should continue until either:

  1) an entry with a production is encountered. In the case, parsing is continued as usual.
  2) an entry marked as ***synch*** is encountered. In this case, the parser will pop that non-terminal A from the stack. The parsing continues from that state.

  - To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that that unmatched terminal is inserted.

Parsing table example

```
#E = T E1
#E1 = '+' T E1 | \L
#T = F T1
#T1 = '*' F T1| \L
#F = '(' E ')' | 'id'
```

Console

```
<terminated> lexWin6.exe [C/C++ Application] E:\cpp\lexWin
3 prod.PSymbols.size()
printParsingTable
E :(: T E1
E :id: T E1
T :(: F T1
T :id: F T1
E1 :+: + T E1
E1 :): \L
E1 :$: \L
F :(: ( E )
F :id: id
T1 :+: \L
T1 :*: * F T1
T1 :): \L
T1 :$: \L
```

Main Functions

createProductionGrammer()

It parses grammar file and set productions, terminal and non-terminal in 'grammar' 3D.

detectLeftRecusion()

Use DFS algorithms to detect direct and indirect left recursion in the 'grammar'.

createFirstTable()

It initializes 'first' vector and 'prodFirst' vector and fill them with first terminal of non-terminals and production.

createFollowTable()

Using 'grammar', 'first' and 'prodFirst' vectors, it fills 'follow' vector with the follow terminals of each non terminal.

createParsingTable()

Using 'first' and 'follow' vectors, this method create parsing tree and detect if there is ambiguity in the grammar.

matchTokens()

This method parse the tokensFile then match it with the given grammar using the stack method and panic-mode error recovery method then print the left most derivation and productions used to parse the tokens.