Parser

Compiler Phase2

18-Paula Boules

70-Mirna Fayez Boules

34-Shereen Abo El dahab

68-Menna Salah Selim

❖ Data Structure

- ## Element

/* the single element in the production rule */

```
struct PR_ELEM{
    string st
    bool terminal
    set <int> first
    set <int> follow
   vector<int> owners
   bool first_has_epsilon
}
```

- ## InputParser

vector<struct PR_ELEM> elements
map<int , vector<vector<int>>> prod_rules_indexes
map<string , vector<vector<string>>> prod_rules
map<string,int> elem_index_map

- ## ParsingTable

vector<struct PR_ELEM> elements
map<pair< int, string>, vector<int> > parsing_table
map<int, vector<vector<int>>> prod_rules

- ## Matching

vector<struct PR_ELEM> elements
map<pair< int, string> , vector<int> > parsing_table
vector <pair<string ,vector <int> >> derivatins_steps
vector <string> tokens
stack <int > rules

❖ Algorithms

- ## LL_Productions

  - Read the **rules** from the file , and construct them internally in the Parser
  - Check every rule if contains **left recursion** or **left factor** or **both**
  - If found , two rules will be created , the **rule_new** without the left factor/recursion and the **rule_X** which represents the dash non-terminal element
  - Write the rules back to another file that rules will be parsed from

- ## InputParser

  - Read the **modified rules** from file
  - Give every nonterminal/terminal element a unique index
  - Create a new **struct** for this element
  - Add it to **all_elements** vector , the unique index is its index in the vector
  - Generate the rule internally using hash map , which takes string **element name** as key , and return the vector of vector of string contains the **rule conjunctions and disjunctions** i.e. and(s)/or(s)
  - Map the strings in the rules map into the **indexes** of the **structs** in the all_elements vector

- ## ParsingTable

```
/**
 *  iterate over all nonterminals to compute the first list
 *  @return void
 */
construct_first()
   for i=0  to size of elements
      if elements[i] is not terminal and the first list ofelements[i] isn't computed yet
         get_first of elements[i]
/**
 *  params : index of the NT in all elements list
 *  finds the first for a specific nonterminal , given it's index
 *  @return void
 */
get_first(int NT_index)
   struct PR_ELEM cur_elem ←elements[NT_index]

   /* iterates over all the production rule's parts of the current nonterminal */
   for j ← 0 to size of the current NT production rules
```

```
single_pr ← cur_elem production rule's part number j
frst_index ← 0
has_epsilon← true

/* iterates over the single production rule's elements */
While current single production rule hasn't finished  and  previous first has epsilon
   has_epsilon ←false;

   /* if the first was an epsilon */
   if single_pr[frst_index] is EPS_INDEX
      insert EPS_INDEX to the first list of cur_elem
      set first_has_epsilon with true for the cur_elem
      frst_index++
      has_epsilon ← true
      continue while loop

   term ← first element in the current single production for this NT
   /* if the first was a terminal */
   if term is terminal
      insert the term's index to the first of the cur_elem
      insert to parsing_table map , a key of (cur_elem's index , term's string) and the value is the cur
single production rule

   /* if first was a non terminal -NT- */
   else
      /* if the first NT's first wasn't computed */
      if the term's first wasn't computed
         get_first for the term
      /* iterate over the first NT's first list and add to to the cur NT's first list*/
      for k ← 0 to all term's first
         insert the term's first element to cur_elem first
         /* if the first has epsilon  */
         if current term's first elem is EPS_INDEX
            has_epsilon ← true
            set first_has_epsilon with true for cur_elem
            frst_index ++
            continue the while loop
         insert to parsing_table map , a key of (cur_elem's index , string of term's first) and the value
is the cur single production rule
   update the element in the elements list
```

```
/**
 *  iterate over all nonterminals to compute the follow list
 *  @return void
 */
construct_follow()
   for i=0  to size of elements
      if elements[i] is not terminal and the follow list of elements[i] isn't computed yet
         get_follow(i)


/**
 *  params : index of the NT in all elements list
 *  creates the follow list  for each non terminal
 *  @return void
 */

get_follow(int NT_index)

   struct PR_ELEM curr_non_ter←elements[NT_index];
   if curr_non_ter is a terminal
      insert curr_non_ter to the follow of it self
      return

   /* if start symbol put $ */
   If NT_index = 0
      Insert DOLLAR_SIGN to follow list
      Update the parsing table

   /* search for the symbol */
   For  i=0 to the owners of curr_non_ter
      owner_to_find ← (curr_non_ter.owners[i])

      //i am the owner of my self
      If owner_to_find = NT_index
         Continue to the next owner

      temp_prod_rule ← get the production rule of the current owner

      // iterate inside each rule of owner
      For j=0 to prod rules of the owner
         //element in the current rule
         temp_single_pr←temp_prod_rule[j];
         //if not found in this production
         if NT_index not found in the current temp_single_pr
            continue to next single_pr
         //case nothing after it
         If curr_non_ter at the right most hand side of the prod rule
            curr_owner←elements[owner_to_find]
```

```
        If follow not computed yet
           get_follow(owner_to_find)
        needed_follow ←curr_owner.follow;
        add needed follow to follow list
        update the parsing table


    //case something after it
    else If curr_non_ter at the right most hand side of the prod rule
        is_epsilon←false
        get first of next element
        //terminal
        If next element was a terminal
           follow.insert(next)
           update parsing table
           continue to the next prod rule

        if first contained epsilon
              is_epsilon←true
        else
             add the next element's first to the follow list
             update the parsing table
        //the first list contains epsilon
        If is_epsilon
           if follow not computed yet
              get_follow(next)
           add the follow of the next element to the follow list
           update the parsing table

curr_non_ter.follow ← follow
elements[NT_index]←curr_non_ter
```

- Matching

  -First we combine lexical and parser by read output file of lexical and passed it to parser to applied o it algorithm .

```
-push $ on stack
-push start non_terminal on stack
While true
        get top os stack
        if  top_stack equal $
```

```
            if tokens[i] equals $
                  stop and accept
            else
                  Error still $ : stack finished and break
        /* get struct element */
      if top of stack terminal and equal tokens
          then match
          pop from stack
         move to next token
      else if top of stack is not terminal
         /* search in  parsing table to get produnctions*/
        pair <int,string> index_table← std::make_pair(top_stack,tokens[i])
        if  entry of parsing table is not empty
                  /* get productions */
           vector <int> prod_entry ←parsing_table.find(index_table)->second
           if  entry equal  epsilon
                  then  pop non_terminal from stack
           else if entry equal sync
              then  pop non_terminal from stack
                   and Error:follow missing
           else
              then  pop non_terminal from stack and put production reversed  in stack
        else
                 /* if entry is empty*/
            then error and input is discarded token input
      else if terminal and don`t match
          then error end pop top of stack and discrded token input
-finally write output in file
```

❖ Parsing Tables

We attached “parsing_table.txt” file

❖ Output

We attached “output.txt” file

❖ Assumptions

- Last line in the code file must be an empty line .