Compiler Phase 2

Parser

Data structure:

* Node having:
* string name: the name of the node.
* bool terminal: if true then it’s terminal, otherwise it’s nonterminal.

Node represents the right side of each rule.

* Rule having:
* int id: id associated to this rule.
* int non\_terminal: id of the nonterminal of each rule (head).
* list of nodes: each node contains one of terminals or nonterminal.

One production rule can have more than one rule (if it has more than one production “or”).

* Non\_terminal having:
* string name: name of nonterminal.
* int id: id associated to this nonterminal.
* list of rule: containing all the productions for this nonterminal.
* list of (pair of int) first: this is the first set of nonterminal, each pair contains id of terminal and id of production.
* List of int follow: this is the follow set of nonterminal, contains id of terminals.
* Grammer having:
* string filename: path of file that contains the grammar productions.
* list of Non\_terminal non\_terminals: containing all nonterminal given in the grammer.
* list of string terminals: containing all terminals given in the grammer.
* list of rule allRules: containing all the rules in the given grammer.
* Int numNonterm: number of nonterminals.
* Int numRule: number of rules.
* Parser\_table having:
* Grammer
* List of (list of int) table: this is the parse table.
* Bool nonLL1: to indicate that the given grammer is LL1 or not.
* String error\_message: the error message if the grammar is not LL1.
* Parser having:
* Grammar\* g
* Link linker = Link()
* list of (list of int) table: this is the parse table.
* List of string output: the output to be printed in the output file.
* Stack of pair of (int ,bool) st: the int determine the id of the terminal or the nonterminal , the bool indicate if it is a terminal or not.
* int input: the id of the terminal which is the current input.
* bool getNext: indicate if we have to get the next token or not.
* Linker having:
* Rules rules
* Final\_NDFA ndfa
* DFA dfa
* Minimization mini
* Tokens tk

To link lexical generator with parser.

Algorithms:

* To compute first set:
* Iterate over the rules of this nonterminal

1. If the first node in the rule is a terminal add it to the first list of this nonterminal with bool equals true.
2. If the first node in the rule is a nonterminal

* If the first list of the nonterminal is not empty add it to the first list of that nonterminal.
* If the first list of the nonterminal is empty compute it then add it to the first list of that nonterminal.
* To compute follow set:
* Check if the nonterminal is start symbol, then add $ sign to the follow set.
* Iterate over all rules

1. check if that rule contains nonterminal (say e) and not the left most nonterminal

* check if the next one is terminal, add its id to the follow set of e.
* If nonterminal (say s), add its first set of s to the follow set of e and checks if the first set of s contain epsilon then add follow set of the nonterminal (head) of the rule to the follow set of e.

1. check if the rule contains e and it’s the left most nonterminal

* add the follow set of nonterminal (head) of the rule to the follow set of e.
* To construct parse table:
* Iterate over all nonterminals, for every nonterminal:
* Iterate over first set, for each terminal in the first set add its production id to the table in entry of (row: nonterminal id, column: terminal id).
* If the first set contains epsilon, then iterate over follow set of that nonterminal and add epsilon production to the table in entry of (row: nonterminal id, column: terminal id).
* Else iterate over follow set and add id of synch to indicate that this is a synchronization token.
* To drive the left most derivation for an input:
* Push -1 (which indicate “$”) to the stack then push the start nonterminal id with bool false to indicate that this is a nonterminal.
* If the top of the stack is a terminal:
* Check if it equals the input pop it from stack and let the bool getNext is true.
* Else indicate error to add the missing terminal to the input and pop it from the stack
* If the top of the stack is a nonterminal:
* Get the int in the parser table at index of the id of nonterminal which is at the top of the stack and the id of the terminal which is the input.
* If the int is -3 (indicate error) discard the current input and let the bool getNext is true.
* If the int is -2 (indicate synchronize) pop this nonterminal from the stack and parse the next top of the stack.
* Else get the rule which its id equals to that int, pop the top of the stack and then push the nodes of the rule in reverse order. Parse again.
* If the stack is empty and also the input is empty return.
* Else if the stack is empty and also the input is not or if the input is empty and the stack is not, indicate error.
* To get the next token:
* Class link is used to link between the lexical generator and the parser.
* By calling the function get\_next\_token in link it will call the get\_next\_token function in the lexical which return the next token.