# DOCUMENTATION

### PLCC Compiler Project 2013 – Stage 1

Batch #28

Udbhav Prasad 2009B4A7523P

Abhijit Lal 2009B3A7577P

STAGE 1 MODULES:

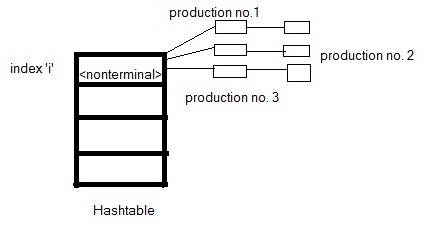
* Lexical Analyser: Uses the DFA to generate tokens. The DFA is implemented in form of a .csv file. The module buffers the input source code and the tokens are generated depending upon the DFA transitions. Also returns errors, if the DFA transitions to a dead state.

Data structures:

* Transition table: The DFA is implemented as a 2D array of integers. No. of rows = No. of states in the DFA (numbered from 0-171). No. of columns = No. of ASCII characters (numbered from 0-130). Each entry in the DFA table DFA[i][j] contains the state number to which the DFA should transition, when it is in the ith state and sees a character whose ascii value is ‘j’. The 2D array is populated by reading from the csv file at runtime.
* Input Buffer: Comprises of two 1-D arrays of characters. Each array has the same size specified in the code as ‘GRAMBUF’. The input source code is read block by block from the source file and stored in this buffer.
* Parser: Uses the tokens given by the lexical analyser, verifies whether the code adheres to the grammar and generates the parse tree.

Data structures:

* Hash table: The grammar for the language is stored as a Hash table with linked lists and multiple such lists from each index in the Hash table. Each list corresponds to a production for a particular non terminal. Since there can be many such productions for a non-terminal, many such linked lists are maintained. Max no. of such lists for each non terminal is 10. LHS of a grammar production is hashed and stored at an index. Collisions are resolved through probing. The ‘grammar.txt’ file contains the representation of the grammar. In the grammar file, each nonterminal is represented as <nonterminal> and each terminal is represented as [terminal].



* Parse table: The parse table is used to parse the input token stream generated from the lexical analyser and aids in the generation of the parse tree. The parse table is a two-dimensional integer array. The rows refer to non-terminals (each non-terminal has been associated to an integer by using a suitable hashing function) and the columns refer to token IDs (which have been assigned integers using an enumerator data structure). Each cell in the matrix refers to the production number (as mentioned above) of the unique rule corresponding to the non-terminal that generates that token.
* Parse tree: The parse tree is generated with the help of the parse table and the hash table of rules. The parse tree node structure contains information about the lexeme that it refers to, the production number that generates the children of the node, a pointer to its parent node, and an array of pointers to its children. The parse tree is generated using an iterative algorithm and an explicit stack to store states.
* Abstract syntax tree: The parse tree is compressed to eliminate the redundant nodes, epsilon productions and create a more concise, understandable tree.

Data structures

* + The abstract syntax tree uses a different node structure from the parse tree. Whereas the latter uses an array structure to store children, the former stores a pointer to its next sibling and a pointer to its first child. It also stores information regarding the lexeme it represents, and the production number which it uses to generate its children. The algorithm to generate the abstract syntax tree works recursively.
* AST rules:
  + Here we explain the encoding used in representing the semantic rules for abstract syntax tree generation. The first string (enclosed in < >) creates a one-to-one mapping from the grammar rules to the semantic rules. The second string, actually an integer from 0-3, denotes the 4 cases which a compression rule may fall under.
    1. Case 0: Pull up one child and keep a few children. The first integer denotes the index of the child to be kept, and the following indices are the children to retain.
    2. Case 1: The string denotes that child whose children have to be pulled up.
    3. Case 2: Keep as is. No changes.
    4. Case 3: Remove that node. Generally done in case that node leads to an epsilon production.
* First and Follow Sets:
  + We have automated the computation of all the first sets but four. These correspond to the cases where the elements need to be calculated with derivations of more than one level, i.e. The direct production has an epsilon.
  + In the follow sets, we have automated the computation of most of them, barring the ones (again) which are derived in more than one step from epsilon productions