**Eric Huang**

**Ho Yin Pun**

**Jung Gi Lee**

**Hyangjin Lee**

**Report**

**Implementation:**

The implementation of this project was divided into a lexer, an abstract syntax tree, a mini parser and a mini virtual machine. The lexer is used to scan the regular expression and create tokens out of it which is used to build the abstract syntax tree. It is like a wrapper for the project one implementation. The abstract syntax tree holds the structure of the grammar specification for the language. The mini parser is a recursive descent parser that is used to accept token definitions and use the lexer to construct the abstract syntax tree. The mini virtual machine will walk the abstract syntax tree and evaluate the expressions to execute the script.

The lexer is composed of a tokenizer, a character class factory, a NFA factory, a DFA factory and a table walker. The tokenizer is responsible for scanning through the regular expression and only retrieves valid tokens specified in the grammar. It accepts a regular expression string as an input to operate on. It keeps track of the current token position in the string and only increment the position after a token was consumed. It also provides a peek method to look ahead by one or more tokens without actually consuming the tokens which provides a way to go backward on regular expression. The peek method serves as a future detection system for the range and exclude functions implementation. A token object contains an opcode and a value field. The opcode field determines that type of the token it is and the value field holds the character that is contained in the token.

The character class generator is responsible for creating a set of classes that contains valid characters for the regular expression to operate on. It uses the functions provided by the tokenizer to retrieve valid token objects and use them to determine the values in the character classes that the regular expression is allowed to use. A series of accepted characters, a range of characters and also excluding characters from another class are ways to populate the class. The union and intersection functions can also be performed between two classes when needed to combine elements or exclude elements. Character classes with a valid name are stored in a hash map for quick lookup.

The NFA generator is responsible for creating an NFA represented by a graph data structure from the regular expression and the character classes. Each state is an NFA node that contains fields that indicates the type of node it is (non accepting or accepting) and all transitions from the current state outward stored in a list structure. The transition triggers include valid characters and epsilon. Operations supported between different NFA’s are union and concatenation which adds edges to the appropriate nodes to connect them in the correct order. An NFA may also perform a kleene star operation on itself for to represent zero or more repetitions. The final output from the generator is a single NFA representing the given regular expression.

The DFA generator class makes a DFA from an NFA. A graph structure is used to represent the DFA. Each DFA node is mapped to a set of NFA nodes which is connected together by epsilon closure. Each DFA node contains information on the transitions to another DFA node, and whether it is final accepted node or not. The resulting DFA may not be minimal at this point. In order to reduce the number of DFA states, the minimization function can be performed to produce a minimal DFA. This function will check each state with one another to determine if they can be distinguished or not. Indistinguishable states that have the same transitions for the same set of character triggers can be combined into a single state.

The table walker is responsible for scanning the program and create tokens from the DFA of the program. When the table walker is called, it takes in a string input and a DFA list that contains multiple DFA’s from the DFA generator. Then on each distinct DFA, it walks on its nodes until it hits the accepting node and checks if each node has a valid transition by calling isTriggered() method. Only when the accepting node of each DFA is hit then it saves the collection of the input characters, or a string as a token. By this point, there must be one or more tokens, and the table walker compare each token to one another and finds the longest token that was obtained from walking each DFA’s. When the longest token is found, the table walker saves it in the token list. After the entire input file is processed, the table walker prints out the tokens from the token list.

The abstract syntax tree was constructed using a tree data structure. The tree contains a mapping of grammar rules with their respective rule ids. An abstract syntax tree will contains a set of tree nodes as children. This tree holds the structure of the grammar and so that it can be walked to evaluate the script after the recursive descent parser constructs it. The mini parser is a recursive descent parser that reads the token definitions provided as input. It then uses the lexer within to construct the abstract syntax tree based on the grammar rules. The mini parser will perform most of the error checking and handling of the program. The mini virtual machine will be responsible to walk the abstract syntax tree with the miniRE script and evaluate the code accordingly. It will perform the specified functions in the script.

The miniVM is for executing the abstract syntax tree. It walks on the abstract syntax tree, starting from its root, until it reaches the terminals. The terminals are defined in the file “token\_spec.txt”. Up until it reaches terminals, it has to handle each grammar according to the set of rule defined in the file “grammar.txt.” In the miniVM class, each grammar is defined in a distinct method so that upon calls and it carries out its expected action, breaks down into other rules and call the corresponding grammar methods, or checks for the terminals. The functional methods are replace, recursivereplace, and print. The replace method takes the content of an input file – the name of the input file is retrieved from a String array, finds every token that matches a targeted REGEX and replaces it with a given ASCII string. The recursivereplace method does the performs the same function as the replace method, except that it replaces the tokens that match the targeted REGEX recursively until the String input does not have anything to replace with a given ASCII string. Both methods write the executed String into the destination text file of which the name is again retrieved from the same String array that contains file names. The print method takes in an abstract syntax tree that has “print” directive and takes whatever is in between the parentheses as a Variable list. In a for-loop, it goes through the Variable list and prints out each Variable in it.