Decaf Compiler

Chandan Kharbanda (201301034)
Tanmay Sahay (201301173)

Phase 1 (Tokenizing using Flex)

1. To tokenize given decaf code we used flex as a tool.

2. Regular expressions were given to flex tool by specifying them in decaf.l

   Example:

   ```
   DIGIT [0-9]

   {DIGIT}+ {
      return INT_VALUE;
   }
   ```

3. Precedence for token matching is decided using following rules:
   - Match the longest possible string every time the scanner matches input.
   - In the case of a tie, use the pattern that appears first in the program.

4. Regular expressions include keywords like “if”, “else”, “for”, “break”, some simple terminals like “{”, “}” and some terminals like INT_VALUE (see example above), STRING_VALUE, etc.

5. If the string does not match with any token, then a counter is increased. We used `error_count` variable for this purpose and detected ‘no token match situation’ using ‘.’ regular expression in the end of decaf.l file.

6. Flex is implemented using DFA and runs in O(n).

Phase 2 (Expressing Decaf Grammar using Bison)

1. To express decaf grammar bison was used.

2. Rules as given below were used for expressing grammar.
   expr : expr PLUS expr

3. Explanation of a common rule (List formation) using below example.
   ```
   filed_decls :      field_decl
               |      field_decls field_decl
   ```

   - When first field_decl is constructed from rule : "field_decl : type identifier_opt_arrs SEMICOLON"
   - It makes a filed_decls using rule : "field_decls : field_decl".
   - When next field_decl is constructed field_decls is updated using "field_decls : field_decls field_decl"
4. To get the values of terminals like INT_VALUE, STRING_VALUE, CHAR_VALUE union is used in bison. yylval is sort of a struct with all the fileds in union. yylval is used to pass values from lexical analyser to parser.
   Example :

   ```
   DIGIT [0-9]

   {DIGIT}+ {
      yylval.ival = atoi(yytext);
      return INT_VALUE;
   }
   ```

   Here atoi was used to convert string (yytext) to integer and returned with token. This integer value can be accessed in bison program using token.ival.
   Similarly strdup is used in case of STRING_VALUE.

5. Shift/Reduce ERRORS : During project most challenging part was dealing with shift/reduce errors. They occur when specified grammar is ambiguous and bison cannot decide whether to shift or reduce when a token is received from lexical analyzer. They were solved using verbose flag while using bison. It generated decaf.output file which contains detailed information about states made and how states change when they encounter a terminal/non-terminal. It can be read to conclude which rule(s) have ambiguity. This is how shift/reduce errors were resolved. Similarly reduce/reduce errors were dealt with.

6. Tokens were declared in decaf.y using %token.

7. %left was used to set precedence of tokens.

8. When moving to phase 2, error_count was not being used before giving “success” or “syntax error” verdict. Later, error_count was declared in decaf.y file and extern was used to refer it in decaf.l file. Thus error_count check was imposed before giving final verdict.