**Documentación**

**Resumen del diseño**

* Se utilizó un arreglo estático para todas las banderillas enviadas por medio de la línea de comandos (en la clase Compiler), esto nos permite colocarlas en una posición definida para su futura utilización.
* Se manejaron **booleanos** en la librería ***debug*** para establecer si una clase debe permitir outputs.
* Definimos un método estático llamado **detectRule** el cual es invocado por la gramática al momento de utilizar una regla de producción, este método nos devuelve el nombre de la regla para luego escribirlo en el archivo de salida.
* Siguiendo las indicaciones de la fase 1, utilizamos antlr para la parte de la gramática (Lexer y Parser) en donde definimos los tokens y las reglas de producción permitidas para el lenguaje Decaf.
* La parte del scanner genera un archivo de salida en el que se especifica el listado de tokens reconocidos indicando la línea del token, el nombre del token y el lexema que fue reconocido por el Lexer.

**Top-level**

* Se creó una variable tipo String para guardar el target de la fase hasta la que se desea ejecutar el compilador, esta variable fue creada en la clase **Debug.java**
* Debido a que no le asignamos acciones a las reglas del lexer, utilizamos la clase DecafLexer.java y la case Token para obtener los tipos de tokes, el número de línea del token, los nombres de las reglas y los token ingresados con los métodos getType( ), ruleNames( ), getLine( ) y getText( ), sin embargo, tuvimos un conflicto debido a que en el Lexer definimos reglas “fragment”, para las cuales se estaba generando un tipo de token pero no se estaba generando un nombre para la regla debido a que no era un símbolo terminal como tal, sino era un fragmento o para de un terminal. Por esto decidimos colocar todas estas reglas “fragment” hasta el final de la definición y que no modificaran las posiciones de del arreglo de nombres de reglas.

**Temas implementados interesantes**

* Se creó una clase llamada **Log**, que pertenece al paquete ***lib***; esto permitirá centralizar el proceso de escritura a un archivo de salida (output), en cada fase.

**Cambios de fases anteriores**

* En la fase 0 no verificábamos que el archivo de entrada existiera, sin embargo, para la fase 1 realizamos esa modificación debido a las correcciones solicitadas.
* En la fase 0 imprimíamos los stage tanto en el archivo de salida como en la consola, sin embargo, debido a que las especificaciones de la fase 1 solicitan que sólo se impriman los errores en la consola, el stage lo estamos haciendo únicamente en el archivo de salida.
* Modificamos el módulo de –target debido a que estaba reconociendo bandera inválidas para la ejecución del compilador.

**Listado de problemas conocidos**

* Se sobrescribió el método emitErrorMessage para lograr la detección de errores en el parser, sin embargo, no obtuvimos resultados al momento de querer detectar los errores en dicha parte del compilador.
* Por precedencia en definición de reglas, tuvimos algunos problemas de reconocimiento de tokens en la fase del parser, es decir que cuando un token se es mandado a llamar por otro token (en el Lexer) se reconoce el nombre de la primera regla que se encuentra definida, pero en el Parser, hay ocasiones es necesario la utilización de ambas reglas, entonces tenemos un problema debido a que la regla del Lexer no coincide con el input.