**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.
* Para identificar los diferentes contextos en el análisis semántico se utilizó un *key* en una hashtable que para los métodos es el *ID* del método, el *tipo* de parámetros, orden de parámetros y *cantidad* de parámetros que recibe dicho método. Para las variables, el *key* es el *scope* y el *ID* de la variable.

**Temas implementados interesantes**

* Se creó una clase llamada **Log.java**, que pertenece al paquete ***lib***; esto permitirá centralizar el proceso de escritura a un archivo de salida (output), en cada fase.
* Dentro de la clase **Log** implementamos los métodos **log()** y **logln()** que utilizamos en lugar del print y println al momento de agregar impresiones en los archivos de salida.
* Se creó una clase llamada **Var.java**, la cual es utilizada como una estructura para guardar nodos tipo variables, parámetros, arreglos y fields. Se declaró un constructor para los siguientes casos:
  + Cuando es un arreglo: este constructor tiene 3 parámetros; uno que nos indica el estilo de estructura, que en este caso es *“ARRAY”,* otro tipo String que nos indica el *ID* del arreglo y por último un tipo int en el que viene el *length* del arreglo.
  + Cuando es un arreglo en el que su length es una expresión: recibe la misma cantidad de parámetros que el constructor anterior, lo único que los diferencia es que en éste en lugar de recibir un int para su *lenght*, se recibe un objeto tipo *Node*, el cual hace referencia a una expresión.
  + Cuando es una variable: en este caso, el constructor tiene 2 parámetros; uno que nos indica el estilo de estructura, que en este caso es “*VARIABLE”*  y otro que nos indica el ID de la variable.Como se puede observar, en este constructor no existe un tipo de variable (*int* o *boolean*) debido a que luego la clase *Var* es utilizada en un nodo tipo *Field*, el cual posee una lista encadenada de tipo Var. Entonces, es en la clase *Field* en donde se guarda el tipo de la lista de variables que se están declarando.
  + Cuando son parámetros que recibe un método: este constructor tiene 3 parámetros; uno que nos indica el estilo de estructura, que en este caso es “*PARAMETRO”,* otro tipo String que nos indica el *ID* del parámetros y un tipo String que nos indica el *tipo* del parámetros que se está recibiendo.
* Se implementó un método llamado **check()** en todas las clases utilizadas como nodos en el AST, esto nos servirá en la fase posterior del análisis semántico debido a que éstos devolverán un booleano indicando si el semántico es correcto o si posee algún error. Este método recibe de parámetro un objeto tipo *Node* que hace referencia al nodo superior o nodo padre.
* Debido a que el método **chech()** es recursivo, solo es necesario mandar a llamar al método de la clase *Root* del AST y éste manda a llamar a todos los métodos de los nodos hijo.

**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.
* Modificamos la gramática en la fase 1 porque teníamos problemas de ambigüedad con la precedencia en el reconocimiento de tokens, además cambiamos las reglas del lexer para dejar correctamente implementada la jerarquía de operaciones.