|  |
| --- |
| Universidad blas pascal |
| Trabajo Final |
| Programación eficiente |
|  |
| **Cenzano Sebastian** |
| **03/02/2019** |

|  |
| --- |
|  |

# Consigna

El objetivo de este Trabajo Final es tomar un programa de alguna materia anterior, proyecto personal o descargado de Internet y analizarlo en detalle suﬁciente como para poder determinar qué correcciones, mejoras y optimizaciones se deberían aplicar. Se deberán analizar los siguientes puntos:

* Cobertura del código fuente,
* Detección de las funciones que realizan el mayor consumo de CPU,
* Redundancia de operaciones aritmético–lógicas,
* Modo de utilización de la memoria asignada en forma dinámica,
* Modo de acceso a la memoria caché y principal.

Las pruebas a realizar deberán estar sistematizadas para así tener una medición objetiva y repetible. Una vez analizado el software, se deberán proponer mejoras a realizarle. Luego, se procederá a su implementación. Finalmente, se deberá hacer un estudio comparativo entre el software original y su versión mejorada aplicando, en caso de ser posible, las mismas pruebas de software sistematizadas. En esta etapa se incluirán todas las tablas y gráﬁcos necesarios para mostrar el impacto de los cambios realizados.

# Problemática abordada

El programa a utilizar fue provisto en la materia “Algoritmos y estructuras de datos 2”. El mismo, utiliza el Algoritmo de Huffman para comprimir un archivo provisto como entrada. Además, cuenta con la opción de descomprimir el archivo utilizando el mismo Algoritmo.

En la fase de compresión, se toma un archivo binario, se analiza su contenido y se crea un árbol de Huffman cargando los nodos con la información de los bytes y su frecuencia de aparición. Luego, información como la tabla de símbolos y el vector que representa al árbol se guardan en el archivo a comprimir, terminando con el proceso de compresión.

Para descomprimir el proceso es, a grandes rasgos, el inverso.

Con el objetivo de utilizar las herramientas vistas durante el dictado de la materia, el programa se compilara, ejecutara y testeara en una distribución del sistema operativo Linux. El software a testear, originalmente programado para Windows, requirió cambios mínimos para que su uso sea el esperado, como ejemplo corregir las instrucciones “system(‘pause’)” mediantes las cuales el programa espera alguna acción de parte del usuario para seguir con la ejecución.

# Herramientas

Las herramientas a utilizar para el profiling son las siguientes:

* Gcov: Para medición de cobertura de código. Se incluirán los reportes generados por la herramienta.
* Gprof: Para obtener información sobre consumo de CPU, discriminado por función. Orden y cantidad de llamada de las funciones. Se incluirán los reportes generados.
* Valgrind: Para obtener información sobre manejo de memoria.

# Ejecución & diagnostico

## Compilación

Se aborde el problema de analizar un archivo de texto con código fuente “C” ,y cumplir con los objetivos previamente mencionados en la consigna, a través de las etapas que realiza un compilador. Las cuales, son las siguientes:

Análisis léxico

En esta etapa de analiza el código fuente y se busca coincidencias en la secuencia de caracteres con los tokens (componentes léxicos) definidos en la gramática.

Análisis sintáctico

La datos necesarios para esta etapa son los tokens obtenidos previamente. Se crea un árbol en base al texto de entrada y los tokens que se obtuvieron y es aquí donde se valida la estructura jerarquica que se espera del código “C”.

Análisis semántico

En esta etapa se valida que el código tenga un significado valido mediante el uso de la tabla de símbolos.  
Aquí se genera el reporte de errores.

Tabla de símbolos

Es una estructura que se genera en el análisis sintáctico/semántico, donde para tokens y reglas especificas de la gramática, en este caso las variables y funciones del código, se almacena información de utilidad para la validación semántica del código.  
Por ejemplo, en el caso de la variables se registran “nombre”, “tipo”, “valor” y si fue utilizada o no.

# Solución propuesta

Se desarrollo un programa mediante la implementación de un proyecto en el entorno de desarrollo “Netbeans”, utilizando el plugin previamente mencionado “ ANTLR”. Este plugin permite generar un analizador para todas las etapas mencionadas en el punto anterior , agregando funcionalidades de Java, para simplificar la creación y uso de objetos como la Tabla de símbolos y los símbolos en sí mismos.

## Gramática

Se creó una gramática para una versión reducida de C. Los tokens definidos son los siguientes:

*CARACTER\_LLAVE\_ABRE:'{';*

*CARACTER\_LLAVE\_CIERRA:'}';*

*CARACTER\_PARENTESIS\_ABRE:'(';*

*CARACTER\_PARENTESIS\_CIERRA:')';*

*TIPO\_DATO\_INT: 'int ';*

*TIPO\_DATO\_DOUBLE: 'double ';*

*TIPO\_DATO\_FLOAT: 'float ';*

*TIPO\_DATO\_CHAR: 'char ';*

*TIPO\_DATO\_STRING : 'string ';*

*TIPO\_DATO\_VOID : 'void ';*

*ESTRUCTURA\_IF:'if';*

*ESTRUCTURA\_ELSE:'else';*

*ESTRUCTURA\_FOR: 'for';*

*ESTRUCTURA\_WHILE:'while';*

*INSTRUCCION\_RETURN : 'return ';*

*fragment LETRA: [a-zA-Z];*

*fragment DIGITO:[0-9];*

*ID: (LETRA|'\_')(LETRA | DIGITO | '\_')\*;*

*VALORNULL :'null';*

*NUMERO : DIGITO+('.'DIGITO+)?;*

*STRING : '\"'(~'\"')\*'\"';*

*COMPARADOR\_IGUAL : '==';*

*COMPARADOR\_DISTINTO : '!=';*

*COMPARADOR\_MAYOR : '>';*

*COMPARADOR\_MENOR : '<';*

*COMPARADOR\_MAYOR\_IGUAL : '>=';*

*COMPARADOR\_MENOR\_IGUAL : '<=';*

*OPERADOR\_ASIGNACION : '=';*

*OPERADOR\_SUMA : '+';*

*OPERADOR\_RESTA : '-';*

*OPERADOR\_PRODUCTO : '\*';*

*OPERADOR\_DIVISION : '/';*

*OPERADOR\_SUMA\_COMPUESTA : '+=';*

*OPERADOR\_RESTA\_COMPUESTA : '-=';*

*OPERADOR\_PRODUCTO\_COMPUESTO : '\*=';*

*OPERADOR\_DIVISION\_COMPUESTA : '/=';*

*OPERADOR\_INCREMENTO : '++';*

*OPERADOR\_DECREMENTO : '--';*

*OPERADOR\_AND : '&&';*

*OPERADOR\_OR : '||';*

*SIGNO\_COMA : ',';*

*SIGNO\_PUNTO : '.';*

*SIGNO\_PUNTO\_Y\_COMA : ';';*

*/\* For whitespace skiping \*/*

*WS:[ \n\r\t]->skip;*

## Sintaxis

Para el analizador sintáctico, se utilizaron las siguientes producciones:

*programa*

*instrucciones*

*termino\_matematico*

*lista\_expresiones\_matematicas*

*declaracion\_variable*

*declaraciones\_asignacion*

*declaracion\_funcion*

*parametros\_en\_declaracion*

*lista\_parametros\_en\_declaracion*

*llamada\_funcion*

*valor\_funcion*

*parametros\_en\_llamada*

*lista\_parametros\_en\_llamada*

*bloque\_instrucciones*

*valor*

*tipo\_dato*

*retorno*

*asignacion*

*operador\_incremento\_decremento*

*for\_asignacion*

*for\_lista\_asignaciones*

*operador\_asignacion\_compuesta*

*estructura\_control*

*estructura\_control\_if*

*estructura\_control\_else*

*expresion\_logica*

*lista\_expresiones\_logicas*

*termino\_logico*

*lista\_terminos\_logicos*

*estructura\_control\_for*

*estructura\_control\_while*

En base a las producciones resaltadas podemos entender cuáles son las prestaciones del lenguaje “C” que son soportadas por este proyecto.

## Implementación en Java

Junto con las prestaciones del plugin ANTLR, se crearon distintos Clases en Java para asistir el proceso en diferentes etapas que se explicaran en casa caso.

#### Clase Símbolo

Representa cada elemento que se encontrara en la Tabla de simbolos. La representación incluye 4 atributos que son comunes para funciones y variables. Por estos atributos en común es que las clases Variable y Función heredan esta implementación.

Símbolo

-tipo  
-nombre  
-usada  
-scope  
  
+ Simbolo()

+ isUsada()

+ setUsada(boolean usada)

+ getTipo()

+ setTipo(String tipo)

+ getNombre()

+ setNombre()

+ getScope()

+setScope()

#### Clase TablaSimbolo

Lista los simbolos y permite su manipulación.

TablaSimbolos

-LinkedList<Simbolo> simbolos;

+TablaSimbolos()

+getSimbolos()

+existeSimboloEnScope()

+existeSimbolo()

+agregarSimbolo()

+getScopeSimbolo()

+getSimbolo()

+removerSimbolo()

+removerSimbolos()

+setUsadoSimbolo()

+toStringScope()

#### Clase Variable

Representa a las variables del código propiamente dichas. Además de los atributos que posee por heredar de Símbolo, posee un atributo para saber si la variable esta inicializada y su valor.

Variable

-valor

-inicializada

+Variable()

+getValor ()

+setValor ()

+isInicializada ()

#### Clase Función

Representa a las funciones del código. Además de los atributos que posee por heredar de Símbolo, posee un atributo para saber si la variable esta inicializada y su valolista para listar los argumentos.

Funcion

-argumentos

+getArgumentos ()

+setArgumentos ()

+existeArgumento ()

+agregarArgumento ()

#### Clase Tac

Cumple una función similar a TablaSimbolos. Tiene el propósito de lista y manipular las instrucciones de código de 3 direcciones (Three Addres Code).

Tac

-instrucciones

-cantInstrucciones

-finalCode

+ Tac ()

+ getInstrucciones ()

+ setInstrucciones ()

+ getCantInstrucciones ()  
+ setCantInstrucciones()  
+ getInstruccion()  
+ addVariable()

#### Clase Instruccion

Representa instrucciones de código de tres direcciones.

Instruccion

- nombre

- usada

- id3ac

+ getNombre ()

+ setNombre ()

+ isUsada ()

+ setUsada ()  
+ getId3ac()  
+ setId3ac()

Durante el análisis de código otras funciones y métodos son utilizados como por ejemplo la Clase “Embellecedor”, pero al tener otro propósito que el análisis gramático, no se explicara en detalles en este trabajo.

# Conclusión

Luego de estudiar e implementar las distintas etapas de análisis de un compilar, la conclusión más fuerte que puedo obtener es que antes de comenzar con la implementación, la documentación y el diseño deben ser exhaustivos y muy detallados. En muchas etapas del desarrollo, a simple vista podía notarse que errores en el diseño de la gramática impactan en todas las etapas y el trabajo se vuelve exponencial. Esto se manifestó principalmente a la hora de generar el código de 3 direcciones.

El trabajo de un compilador debe ser preciso y sometido a constantes mejoras, por lo cual el trabajo presentado es un trabajo en desarrollo.   
Dentro del alcance propuesto para este trabajo, el programa desarrollado cumple con las siguientes características.

* Detectar declaración de variables
* Detectar declaración de funciones
* Asignar variables en declaración.
* Asignación de variables (fuera de la declaración).
* Detectar variables declaradas dos veces o más.
* Detectar funciones declaradas dos veces o más.
* Detecta Balance de paréntesis (mediante reglas gramáticas).
* Detectar el uso de variables que no existen.
* Declaración de variables con mismo nombre en otro scope.
* Generación de tabla de símbolos.
* Generación de código de 3 direcciones.