UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

"TRADUCTOR ENTRE HERRAMIENTAS CIENTÍFICAS"

T E S I S

QUE PARA OBTENER EL TÍTULO DE INGENIERO EN COMPUTACIÓN
PRESENTA:

MARIO ARTURO NIETO BUTRÓN

DIRECTOR DE TESIS:

DR. FREDERIC TRILLAUD PIGHI

2013

JURADO ASIGNADO

Presidente: Ing. Alberto Templos Carbajal

Vocal: Dr. Frederic Trillaud Pighi

Secretario: Dr. Jesús Savage Carmona

1$^{er.}$ Suplente: Dr. Alejandro Farah Simón

2$^{do.}$ Suplente: Ing. Julio Alfonso De León Razo

Ciudad Universitaria, México, D.F.

Abril 2013

DIRECTOR DE TESIS:

DR. FREDERIC TRILLAUD PIGHI

A mi madre y a mis hermanos

Agradecimientos

A mi familia y especialmente a Beatriz mi madre que me acompañó en esta gran aventura, por sus horas de paciencia, sus sabios consejos, su cálido amor y por su constante apoyo en todas y cada una de mis locuras, a mis hermanos Jacob y Lía por todos aquellos momentos de constante felicidad a su lado, aquellas peleas pero sobretodo por su inmenso amor, finalmente a mis abuelos Elías y Bertha que no alcanzaron a ver este momento pero estoy seguro que donde estén estarán orgullosos.

A mi director el Dr. Frederic Trillaud Pighi por permitirme cumplir uno de mis sueños el cual fue pertenecer al Instituto de Ingeniería, por sus horas de paciencia, por aquellas horas de reflexión constante, por todas aquellas platicas de sensibilización , por todo ese apoyo sin medida que me brindó, por permitirme divertirme con su servidor linux Olli, pero sobretodo mostrarme como debe ser un ingeniero.

A mi Codirector el Ing. Julio Alfonso De León Razo, gracias por todas esas horas dedicadas a ayudarme, a guiarme por el camino correcto, por los sabios consejos, por las revisiones constantes, por presionarme para ser un gran ingeniero.

Al Dr. Alexis A. Aguilar-Arévalo por su recomendaciones y consejos sobre el uso de ROOT ya que sin él mi trabajo hubiera resultado mucho mas complicado.

El servicio de cómputo del IINGEN en particular al Ing. Marco Ambriz Maguey, al Ing. Fernando José Maldonado S. y a la Ing. María de Jesús Ortega por permitirme trabajar en los servidores del Instituto, por compartir su conocimiento de forma amable y enseñarme a ser una mejor persona.

A mis amigos de SAFIR por todas esas noches de contemplación estelar, por todos aquellos recuerdos que llevaré siempre conmigo.

A mi alma mater la Universidad Nacional Autónoma de México, particularmente a mi amada Facultad de Ingeniería que me llenó de conocimientos, momentos de felicidad y me formó como un gran profesionista.

A todos ustedes, que por espacio, no menciono en estos renglones pero que pertenecen a este gran sueño.

¡Muchas gracias a todos!

Índice general

Pag.

Índice de figuras VII

Índice de tablas IX

1 Introducción 1
- 1.1 Antecedentes 1
- 1.2 Definición del problema 3
- 1.3 Objetivos de la tesis 4
- 1.4 Metodología de solución 4

2 Herramientas Científicas y Técnicas 5
- 2.1 Plataforma Salome 5
  - 2.1.1 Funcionalidades 6
  - 2.1.2 Módulo geométrico GEOMPY 7
  - 2.1.3 Ejemplo 8
- 2.2 ROOT 10
  - 2.2.1 Funcionalidades 11
  - 2.2.2 Paquete Geométrico TGeoManager 12
  - 2.2.3 Ejemplo 13

3 Presentación breve de la teoría de lenguajes formales 17
- 3.1 Conjuntos 17
- 3.2 Alfabeto, cadenas y lenguaje 18
- 3.3 Operaciones aplicadas a lenguajes 18
- 3.4 Expresiones Regulares 19
- 3.5 Gramáticas 21

4 Presentación del traductor salomeToROOT 26
- 4.1 Análisis del problema 29
- 4.2 Arquitectura de salomeToROOT 30

5 Filtro 32
- 5.1 Expresiones regulares en C 33
- 5.2 Arquitectura del Filtro 33

5.3 Prueba al filtro 34

6 Análisis Léxico **37**
6.1 Componentes Léxicos, patrón y lexema 37
6.2 Manejo de errores léxicos 38
6.3 Flex 38
6.4 Analizador Léxico en Flex 42
6.5 Prueba al analizador 44

7 Análisis Sintáctico **47**
7.1 Análisis sintáctico descendente 48
7.2 Análisis sintáctico ascendente 50
7.3 Diagramas de Sintaxis 53
7.4 Gramática para salomeToROOT 55
7.5 Bison 60
7.6 Analizador Sintáctico en Bison 64

8 Integración y liberación **66**
8.1 Integración y Pruebas 66
8.2 Liberación del traductor 72

9 Conclusiones **75**

Apéndices **77**

A Código fuente salomeToROOT **78**

B Makefile **88**

C Manual de Usuario **92**

D Reglas para compilar en Flex y Bison **94**

Referencias **95**

Referencias **95**

Índice de figuras

Pag.

Figura 1.1 Ejemplo de un detector usando CCD's similar al detector DAMIC construido en Salome (cortesía del Dr. Frederic Trillaud Pighi). 2
Figura 1.2 Ejemplo de un detector usando CCD's similar al detector DAMIC construido en ROOT (cortesía del Dr. Alexis A. Aguilar-Arévalo). 3

Figura 2.1 Plataforma Salome (2012). Recuperado de SALOME [10]. 6
Figura 2.2 Módulos de Salome (2012). Modificado de SALOME [11]. 6
Figura 2.3 Se abre Salome. 9
Figura 2.4 Se carga el archivo de órdenes en Salome. 9
Figura 2.5 Cubo en Salome. 10
Figura 2.6 ROOT (2012). Recuperado de ROOT [44]. 11
Figura 2.7 Administrador del paquete geométrico (2012). Recuperado de TGeomanager [47]. 13
Figura 2.8 Ejecución del archivo cube.c en ROOT. 16
Figura 2.9 Cubo en ROOT. 16

Figura 3.1 Árbol. 24
Figura 3.2 Árbol sintáctico. 25

Figura 4.1 Proceso de compilación según Aho [3]. 26
Figura 4.2 Fases de un compilador según Aho [3]. 28
Figura 4.3 Arquitectura de salomeToROOT. 30

Figura 5.1 Arquitectura del filtro. 33
Figura 5.2 Compilando el Filtro. 35
Figura 5.3 Entradas inválidas al Filtro. 36
Figura 5.4 Salida Filtro. 36

Figura 6.1 Proceso Flex. 39
Figura 6.2 Ejecución scanner. 39
Figura 6.3 Estructura archivo "*.l". 39
Figura 6.4 Función main() por defecto en Flex. 41

Figura 6.5 scanner.l sección de declaraciones. 45
Figura 6.6 Ejecución del analizador léxico. 46

Figura 7.1 Interacción entre el analizador sintáctico y el analizador léxico (modificado de Aho [3]). 47
Figura 7.2 Organigrama descendente. 49
Figura 7.3 Organigrama gramáticas ascendentes. 52
Figura 7.4 Diagrama para C según Ruíz [12]. 53
Figura 7.5 Diagrama para D->A|B según Ruíz [12]. 54
Figura 7.6 Diagrama para (A)* según Ruíz [12]. 54
Figura 7.7 Diagrama para $E- > A|\xi$ según Ruíz [12]. 54
Figura 7.8 Diagrama de sintaxis del Cubo. 56
Figura 7.9 Diagrama de sintaxis del Cilindro. 56
Figura 7.10 Diagrama de sintaxis Esfera. 57
Figura 7.11 Diagrama de sintaxis del Cono. 57
Figura 7.12 Diagrama de sintaxis del Toroide. 57
Figura 7.13 Diagrama de sintaxis del Rectángulo. 58
Figura 7.14 Diagrama de sintaxis del Disco. 58
Figura 7.15 Reglas de producción para primitivas geométricas. . . . 59
Figura 7.16 Estructura Bison. 60

Figura 8.1 Yacc/Bison. Recuperado de [18]. 69
Figura 8.2 Toroide en la plataforma Salome. 70
Figura 8.3 Ejecución salomeToROOT. 70
Figura 8.4 Toroide ROOT. 72
Figura 8.5 Licencia GPL de GNU. 74

Figura B.1 Estructura archivo Makefile 88

Índice de tablas

Pag.

Tabla 6.1 Ejemplos componentes léxicos. 38
Tabla 6.2 Reglas para especificar las expresiones regulares en Flex. 41
Tabla 6.3 Categoría 1: Palabras Reservadas. 43
Tabla 6.4 Categoría 3: Símbolos especiales. 43
Tabla 6.5 Categoría 5: Otros. 43
Tabla 6.6 Expresiones Regulares para cada categoría. 44

Tabla 7.1 Análisis Descendente Ejemplo 4. 49
Tabla 7.2 Frases y Handle de la gramática. 51
Tabla 7.3 Acciones de un analizador sintáctico ascendente para el ejemplo 8. 52
Tabla 7.4 Consideraciones de nuestra gramática. 55

Tabla B.1 Objetivos especiales comunes 91

Capítulo 1

Introducción

EN la gran mayoría de los proyectos científicos involucrados en el desarrollo de equipos experimentales, físicos e ingenieros necesitan colaborar y compartir experiencias. Ambos utilizan herramientas computacionales complejas y específicas al problema que les compete. Generalmente, esas herramientas son muy diferentes entre sí y no toman en cuenta el intercambio de información útil que debe haber entre físicos e ingenieros, lo que provoca la duplicación de esfuerzos.

La incompatibilidad entre herramientas presenta a los desarrolladores de software un reto complejo, debido a que plantea diferentes problemas que deben ser resueltos de manera distinta. En este trabajo se presenta una propuesta de solución entre dos herramientas numéricas: la plataforma Salome [10] y el marco de trabajo ROOT [44].

Después del análisis de las características de esas herramientas se llegó a la conclusión que la mejor solución a dicha incompatibilidad es la construcción de un compilador permitiendo la traducción de datos geométricos. Este traductor se denomina “salomeToROOT” indicando que se traducirán geometrías desde Salome hasta ROOT.

Se presentan los conceptos claves que deben tenerse en cuenta en la construcción de compiladores como expresiones regulares y gramáticas, además se exponen herramientas libres y especializadas en la construcción de tales compiladores.

1.1. Antecedentes

Este proyecto nació debido a la necesidad, de un proyecto que contempla el desarrollo de un detector de partículas, de intercambiar entidades geométricas entre el Instituto de Ciencias Nucleares (ICN) y el Insituto de Ingeniería (IINGEN) de la Universidad Nacional Autónoma de México (UNAM) participando en una colaboración internacional *Dark Matter In CCDs* (DAMIC) [19].

Se cree que el espacio esta formado principalmente por materia oscura, dicha

materia no puede ser detectada por métodos científicos comunes. El laboratorio estadounidense Fermilab (*Laboratorio Nacional Fermi*) se formó hace varios años a partir del desarrollo de un instrumento científico con base a dispositivos de carga acoplada. En 2012, se creó alrededor de este grupo un proyecto internacional llamado DAMIC-SOUTH (Dark Matter in CCDs south) contemplando la participación de Argentina, Brasil, Chile, Estados Unidos, y México con el objetivo de construir e instalar un detector de partículas en el hemisferio sur del planeta.

DAMIC intenta detectar materia oscura utilizando dispositivos de carga acoplada. Estos son una versión mejorada de la tecnología usada en las cámaras digitales. Los sensores se encuentran en un criostato cubierto por un blindaje de plomo. Un criorefrigerador permite mantener los sensores a menos de 150°C. Este equipo será instalado en 2014 en una mina alrededor de 1,500 km bajo tierra.

El ICN y el IINGEN participan activamente en el proyecto DAMIC-SOUTH por lo cual en 2011 el Dr. Frederic Trillaud Pighi construyó un dibujo asistido por computadora (en inglés, *Computer-Aided Drafting* or CAD) del detector el cual puede ser visto en la Figura 1.1.

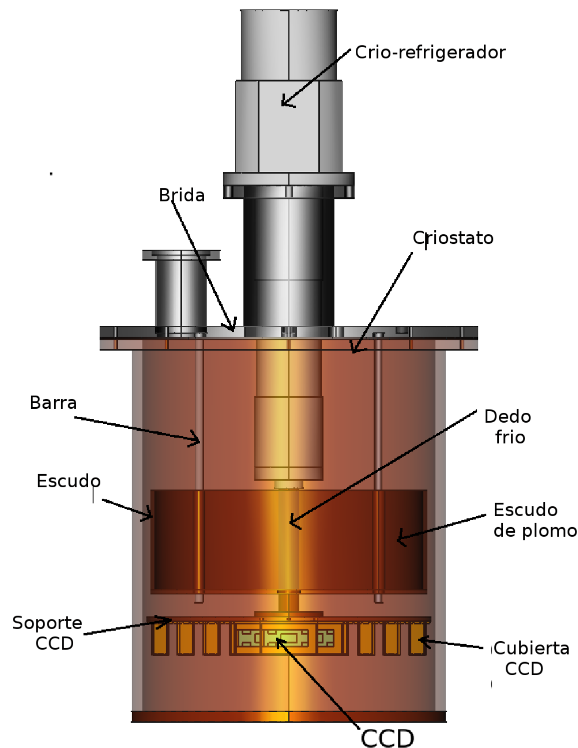

Figura 1.1. Ejemplo de un detector usando CCD's similar al detector DAMIC construido en Salome (cortesía del Dr. Frederic Trillaud Pighi).

A partir del modelo CAD se construyó un modelo geométrico del detector en la herramienta ROOT (*Véase la Figura 1.2*), para poder realizar simulaciones físicas usando GEANT4 [16] por el Dr. Alexis A. Aguilar-Arévalo del ICN.

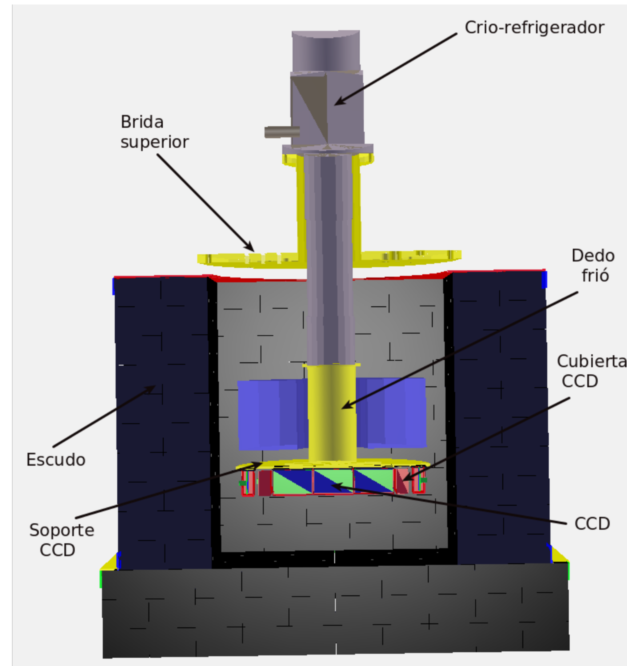

Figura 1.2. Ejemplo de un detector usando CCD's similar al detector DAMIC construido en ROOT (cortesía del Dr. Alexis A. Aguilar-Arévalo).

1.2. Definición del problema

En la construcción de modelos geométricos, la plataforma Salome utiliza el lenguaje interpretado Python y el marco de trabajo ROOT utiliza el lenguaje de programación C++ con un intérprete llamado CINT [46]. La necesidad de este trabajo surge por la actual inexistencia de un puente entre la plataforma Salome y el marco de trabajo ROOT, lo cual dificulta el intercambio de datos y la duplicación de los esfuerzos entre investigadores del Instituto de Ciencias Nucleares y el Instituto de Ingeniería.

1.3. Objetivos de la tesis

Al final de este trabajo se esperan alcanzar los siguientes objetivos:

1. Construir un traductor sencillo entre la plataforma Salome y el marco de trabajo ROOT, para las siguientes primitivas:

   - Cubo
   - Cilindro
   - Cono
   - Esfera
   - Toroide
   - Rectángulo
   - Disco

   El traductor debe permitir cambiar una geometría escrita en código Python que interpreta la plataforma Salome a su equivalente en código C++ que interpreta el marco de trabajo ROOT.

2. Adquirir experiencia en el uso y desarrollo de software especializado.

3. Contribuir a la comunidad del Software Libre.

1.4. Metodología de solución

Para llegar al traductor funcional, salomeToROOT, se propóne escribir :

1. Un filtro que discrimine la entrada de datos.
2. Un analizador léxico con Flex, para procesar la salida del filtro y enviar componentes léxicos al analizador sintáctico.
3. Un analizador sintáctico con Bison, especificando una acción al reconocer una instancia de las reglas gramaticales.
4. Una función de control que llame al traductor.

Capítulo 2

Herramientas Científicas y Técnicas

La plataforma Salome y el marco de trabajo (en inglés, *framework*), ROOT, son dos ejemplos de herramientas distintas y con filosofías diferentes. De un lado, Salome se usa para desarrollar diseños asistidos por computadora y para realizar análisis termo-mecánicos de estructuras mecánicas utilizando el método de los elementos finitos del inglés *Finite Element Analysis* (FEA). Del otro lado, ROOT fue elaborado en el marco del desarrollo de la física de altas energías permitiendo el análisis de datos de gran volumen, la modelización física de algunos fenómenos y mucho más. Ambas contienen módulos distintos para generar modelos geométricos.

2.1. Plataforma Salome

La plataforma Salome es una herramienta dedicada a la generación de modelos 3D, para la visualización y mallado de estos, dando soluciones a problemas mecánicos, térmicos y de fluidos apoyándose de las herramientas libres que utilizan el método de elementos finitos que son: Code_Aster y Code_Saturne. Esta plataforma esta siendo desarrollada por el laboratorio francés "Commissériat à l'Energie Atomique" (CEA) [1] y la empresa francesa "Electricité de France" (EDF) [17] desde el año 2001, con un equipo de trabajo de 20 personas. Actualmente está formada por 1,300,000 líneas de código en donde el 90 % de las mismas están escritas en C++ y el 10 % restante están escritas en Python [10].

Salome fue creado sobre la tecnología abierta Open CASCADE [9] agregando un intérprete Python y una interfaz gráfica. La plataforma Salome se esta distribuyendo bajo la licencia LGPL [23] de GNU [25].

Figura 2.1. Plataforma Salome (2012). Recuperado de SALOME [10].

2.1.1. Funcionalidades

La plataforma Salome esta compuesta por un conjunto de módulos, los cuales pueden ser observados en la Figura 2.2.

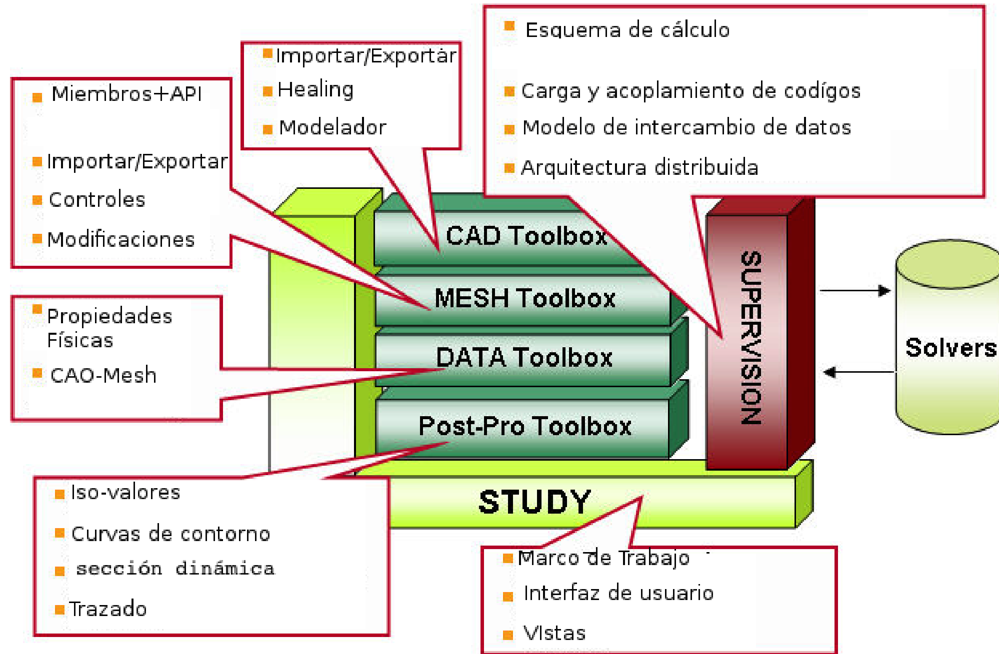

Figura 2.2. Módulos de Salome (2012). Modificado de SALOME [11].

Algunos de estos módulos son:

- Módulo GUI: proporciona la interfaz gráfica del usuario.
- Módulo de geometría (GEOMPY): facilita la construcción y optimización de los modelos geométricos utilizando una amplia gama de funciones de CAD.
- Módulo de mallado (SMESH): genera una malla en los modelos geométricos creados en esta plataforma o importados.
- Módulo de post-procesamiento: lleva a cabo la visualización de datos.

Con la plataforma SALOME se puede:

- Definir modelos geométricos.
- Definir mallado de estos elementos geométricos.
- Manipular las propiedades físicas y las cantidades unidas a los elementos geométricos.
- Realizar cálculos utilizando un programa de solución.
- Visualizar campos de resultado en 1D, 2D y 3D.

2.1.2. Módulo geométrico GEOMPY

Este módulo proporciona un rico conjunto de comandos para crear, editar, importar o modificar un complejo modelo geométrico. Las funcionalidades de este módulo se pueden acceder a través de la interfaz gráfica de usuario (GUI) o mediante programación en Python.

Este módulo esta destinado a:

- Importar y exportar modelos geométricos en IGES [39], STEP [30], BREP [14] y otros formatos.
- Construir objetos geométricos utilizando una amplia gama de funciones.
- Visualizar objetos geométricos.
- Transformación de los objetos geométricos.
- Optimización de los objetos geométricos.
- Diseño de formas a partir de imágenes.

Proporciona un conjunto de herramientas para crear:

- Objetos básicos - puntos, líneas, círculos ...,etc.
- Primitivas - cubos, esferas, conos...,etc.
- Objetos geométricos avanzados -tuberias...,etc.
- Objetos complejos por extrusión, transformaciones (rotación, translación, fusión, etc...), y interpolación de otros objetos.
- Entre otras.

Una vez creado o importado el modelo geométrico se puede visualizar a través del visor 3D OCC.

2.1.3. Ejemplo

Para crear un modelo geométrico se puede hacer desde el entorno gráfico o desde un archivo de órdenes, en este ejemplo se mostrará la creación de un cubo desde un archivo de órdenes Python (*script*) el cual esta formado como sigue:

```
import geompy
import Salome
gg=Salome.ImportComponentGUI("GEOM")

#La filosofía para crear modelos geométricos
#en Salome se basa en crear las geometrías
#en un sistema de coordenadas de tres dimensiones

#Función para crear un cubo
#en el origen del sistema
#cada lado medira 200
caja1=geompy.MakeBoxDXDYDZ(200,200,200)

#Agrega los objetos al estudio
id_caja1=geompy.addToStudy(caja1,"caja1")

#Dibuja los objetos sobre OCC
gg.createAndDisplayGO(id_caja1)
gg.setDisplayMode(id_caja1,1)
```

Para ejecutar el archivo de órdenes, se abre la interfaz de Salome y en "File" se selecciona new :

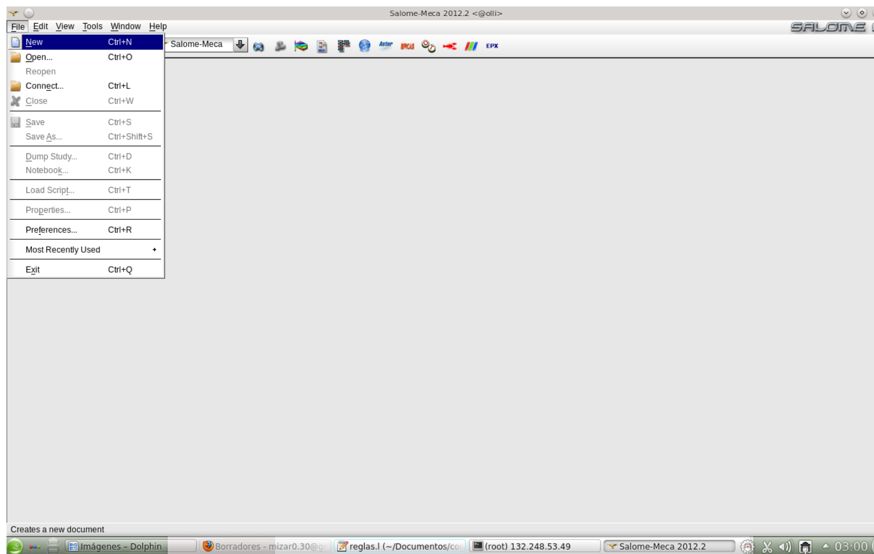

Figura 2.3. Se abre Salome.

Se carga el archivo de órdenes desde "File -> Load" archivo de órdenes.

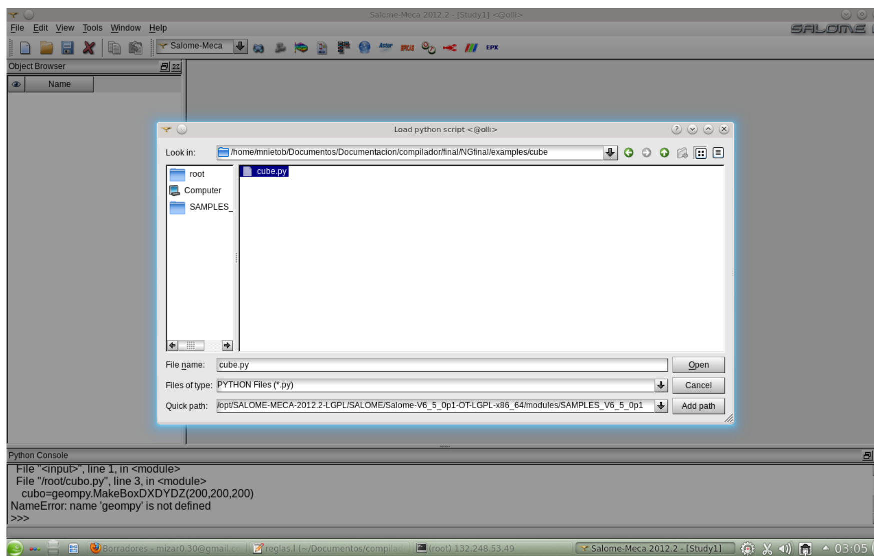

Figura 2.4. Se carga el archivo de órdenes en Salome.

Finalmente el cubo es dibujado en el lienzo de Salome:

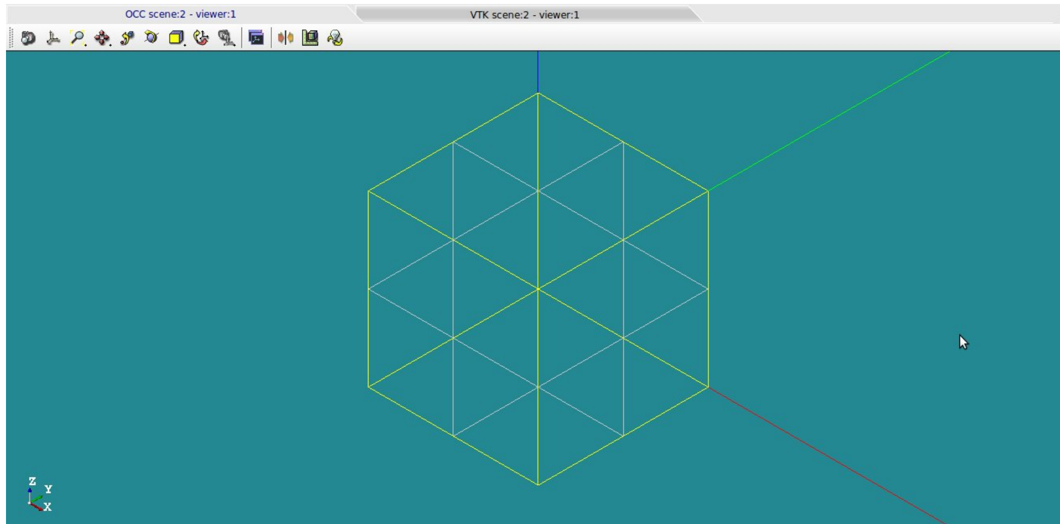

Figura 2.5. Cubo en Salome.

2.2. ROOT

ROOT es un marco de trabajo orientado a objetos, escrito en el lenguaje de programación C++ que incorpora el intérprete CINT. Esta dirigido al análisis de datos en la física de altas energías.

Este marco cuenta con todo el poder del lenguaje de programación C++, por tanto se pueden incorporar de manera sencilla, nuevas clases y funciones, permitiendo también gestionar y analizar grandes cantidades de datos, de forma eficiente.

ROOT es un desarrollo del CERN (Organización Europea para la Investigación Nuclear) [13] desde mediados de los 90s y nació en el contexto del experimento NA49 [15]. Esta formado por más de 1,500,000 líneas de código, desarrollado por 12 personas y está disponible bajo la licencia LGPL.

Figura 2.6. ROOT (2012). Recuperado de ROOT [44].

2.2.1. Funcionalidades

Actualmente se trabaja ROOT en su versión 5-34-00. La columna vertebral de éste es una jerarquía de clases conteniendo alrededor de 1200's agrupadas en 60 bibliotecas divididas en 19 categorías principales [45]. Las clases, heredan de una sola clase común, Tobject, definiendo protocolos para comparar objetos entra otras funcionalidades, es similar a la arquitectura que utiliza Java [45] .

ROOT provee clases y funciones para:

- Histogramas.
- Gráficas.
- Árboles.
- Bibliotecas matemáticas.
- Álgebra líneal.
- Colecciones.
- Vectores.
- Paquete Geométrico.
- Redes.
- Procesamiento en paralelo

El intérprete CINT es un producto independiente desarrollado por Masaharu Goto [46] que traduce y ejecuta cada instrucción recibida por el usuario.

Un intérprete es la herramienta perfecta para código que cambia con frecuencia y se ejecuta varias veces.

Las principales características de CINT son:

- Soporta los estándares K&R-C, ANSI-C, and ANSI-C++ [48].
- Es compatible con la herencia múltiple, función virtual, sobrecarga de funciones, la sobrecarga de operadores, los parámetros por defecto, las plantillas, y mucho más. CINT es lo suficientemente robusto como para interpretar su propio código fuente.
- Intepreta código C/C++.
- Maneja enormes cantidades de código fuente.
- Trabaja en varias plataformas: HP-UX, Linux, SunOS, Solaris, AIX, Alpha-OSF, IRIX, FreeBSD, NetBSD, NEC EWS4800, NewsOS, BeBox, WindowsNT, Windows9x, MS- DOS, MacOS, VMS, NextStep, Convex.

2.2.2. Paquete Geométrico TGeoManager

ROOT provee un paquete geométrico para la construcción, navegación y visualización de geometrías. Este paquete define el medio, el material, el campo magnético, con el objetivo de ser capaz de utilizar la misma geometría con varios fines, como seguimiento, reconstrucción, visualización o para resolver problemas físicos con herramientas externas como GEANT4. El modelador ofrece un paquete de dibujo de gran alcance, apoyándose en varias opciones de visualización.

El administrador del paquete geométrico es la clase TGeoManager donde se incorporan todas las Interfaces de Programación de Aplicaciones (IPA, en inglés, *API* o *Application Programming Interface*) necesarias para la creación y el seguimiento de una geometría. Se define un apuntador global, gGeoManager, con el fin de ser totalmente accesible desde código externo, es el propietario de todos los objetos geométricos definidos en una sesión. Contiene las listas de los:

- medios.
- materiales.
- transformaciones.
- formas y volúmenes.
- acciones de visualización

A continuación se muestra el esquema general de la clase manager.

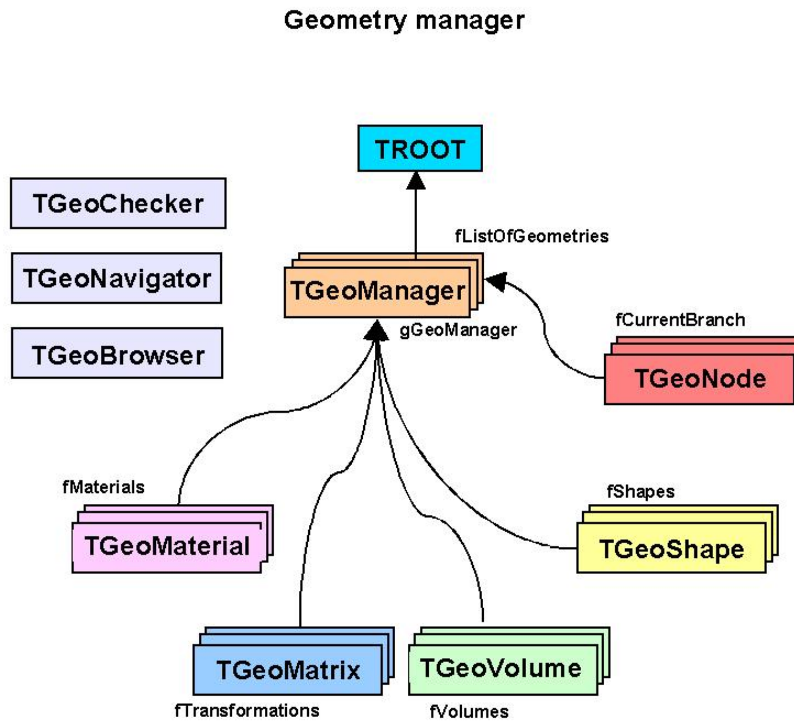

Figura 2.7. Administrador del paquete geométrico (2012). Recuperado de TGeomanager [47].

2.2.3. Ejemplo

Para crear un modelo geométrico en ROOT, se puede hacer desde el intérprete CINT construyéndolo línea a línea o bien se puede crear un archivo de órdenes “*.c”, lo cual se usa regularmente debido a que es necesario guardar la geometría para después reutilizarla.

Para crear los archivos de órdenes se puede hacer de dos formas distintas:

- archivos de órdenes sin nombre

- archivos de órdenes con nombre

Archivos de órdenes sin nombre

En este caso los archivos de órdenes sin nombre son aquellos que contienen toda la definición de la geometría dentro de llaves {}.

Supongamos que el archivo de órdenes donde se almacena es una geometría se llama ordenes1.c, si se requiere ejecutar dicho archivo se hace en la línea de comandos del marco ROOT (*prompt* "root[i]", donde i es el número de líneas que se han ejecutado en el intérprete), con la instrucción:

root[0] **.x ordenes1.c**

Esto carga el contenido del archivo ordenes1.c y ejecuta todas las sentencias en el ámbito global del intérprete.

Si se requiere volver a cargar el archivo de órdenes se hace con la instrucción:

root[1] **.which archivo de ordenes1.c**

Archivos de órdenes con nombre

En este caso es idéntica a definir una función en C o C++. Para cargar en memoria el archivo de órdenes se usa la instrucción:

root[0] **.L name.C**

y para ejecutar se llama el nombre de la función desde el intérprete. Por ejemplo, si se supone que se definió una función con el nombre "run()", entonces se llama a la geometría desde el intérprete con:

root[1] **run()**

Cubo

La creación de un cubo se hace de la siguiente manera:

```
/*Se comienza por incluir la cabecera que
llama al administrador de geometrias.*/
#include "TGeoManager.h"

//Se define el nombre de la función:
void cube()
{

/*
La filosofía de creación de geometrías en ROOT se basa
en tener  un mundo y dentro de  el definir  todos  los
objetos geométricos que se necesiten es decir antes que
nada hay que definir el mundo.
*/
new TGeoManager("world","simple")

//El mundo debe tener un medio y un material asociado.
TGeoMaterial *mat=new TGeoMaterial("Vacuum",0,0,0);
TGeoMedium *med=new TGeoMedium("Vacuum",1,mat);

/*
Se debe definir la forma del mundo en este caso cubica y
se establece a esta geometría como el mundo.
*/
TGeoVolume *top=gGeoManager->MakeBox("top",med,500.,500.,500.);
gGeoManager->SetTopVolume(top);

/*
Para definir una geometría dentro del mundo se debe
establecer, la posición en donde aparecerán
*/
TGeoRotation *rot = new TGeoRotation("rot",0.,180,0);
TGeoCombiTrans *transf = new TGeoCombiTrans(0,0,-100,rot);

//Se define un cubo y lo se añade al mundo
TGeoVolume *mycube=gGeoManager->MakeBox("mycube",med,200.,200.,200);
top->AddNode(mycube,0,transf);

//Se cierra la geometria
gGeoManager->CloseGeometry();

//Se inidica el color del mundo
top->SetLineColor(kMagenta);

//Se dibuja el cubo
gGeoManager->SetTopVisible();
top->Draw();
}
```

La ejecución del cubo puede ser vista en la siguiente impresión del terminal:

```
(mnietob) 132.248.53.49
Archivo  Editar  Ver  Marcadores  Preferencias  Ayuda
  *******************************************
  *                                         *
  *        W E L C O M E  to  R O O T       *
  *                                         *
  *   Version   5.34/02 21 September 2012   *
  *                                         *
  *  You are welcome to visit our Web site  *
  *          http://root.cern.ch            *
  *                                         *
  *******************************************

ROOT 5.34/02 (tags/v5-34-02@46115, Sep 21 2012, 16:29:13 on linuxx8664gcc)

CINT/ROOT C/C++ Interpreter version 5.18.00, July 2, 2010
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] .L cube.c
root [1] cube()
Info in <TGeoManager::TGeoManager>: Geometry world, simple created
Info in <TGeoManager::SetTopVolume>: Top volume is top. Master volume is top
Info in <TGeoNavigator::BuildCache>: --- Maximum geometry depth set to 100
Info in <TGeoManager::CheckGeometry>: Fixing runtime shapes...
Info in <TGeoManager::CheckGeometry>: ...Nothing to fix
Info in <TGeoManager::CloseGeometry>: Counting nodes...
Info in <TGeoManager::Voxelize>: Voxelizing...
Info in <TGeoManager::CloseGeometry>: Building cache...
Info in <TGeoManager::CountLevels>: max level = 1, max placements = 1
Info in <TGeoManager::CloseGeometry>: 2 nodes/ 2 volume UID's in simple
Info in <TGeoManager::CloseGeometry>: ----------------modeler ready----------------
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
root [2] █
(mnietob) 132.248.53.49
```

Figura 2.8. Ejecución del archivo cube.c en ROOT.

Y el resultado:

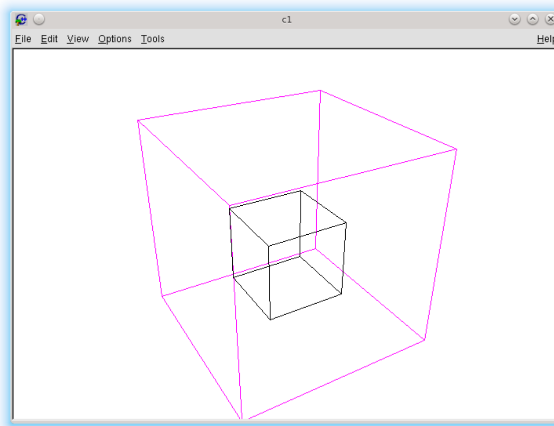

Figura 2.9. Cubo en ROOT.

Capítulo 3

Presentación breve de la teoría de lenguajes formales

En el capítulo anterior se presentaron las herramientas Salome y ROOT. Este capítulo, introduce de manera breve la teoría de los lenguajes formales aplicada en la computación, sobre la cual se esta apoyando la construcción del traductor salomeToROOT, para lo cual se definen algunos conceptos fundamentales como el caso de los conjuntos que son la base del estudio del lenguaje, se revisan temas importantes como las expresiones regulares utilizadas hoy en día ampliamente en varios lenguajes de programación y finalmente se dedica parte de este trabajo al estudio de gramáticas.

3.1. Conjuntos

Un conjunto es un término primitivo en matemáticas por que no existen términos más simples con los cuales definirlo, no obstante de manera intuitiva se puede decir que un conjunto es una colección de elementos o como lo describe George Cantor: "Un conjunto es la unidad de pluralidad" [6].

Normalmente los conjuntos se expresan con letras mayúsculas con o sin subíndices y los elementos de dichos conjuntos son representados con letras minúsculas con o sin subíndices. Para expresar matemáticamente a los conjuntos se tienen dos notaciones:

1. la primera es conocida como notación tabular o por extensión, consiste en definir un conjunto por la enumeración de sus elementos por ejemplo un conjunto A que esta formado por los números 1, 2 y 3 se define como A={1, 2, 3}.

2. la segunda es llamada notación por comprensión o constructiva, en esta los elementos de un conjunto quedan definidos por una propiedad en común para todos ellos por ejemplo si quisiéramos un conjunto formado por todos los números pares el conjunto se denotaría como $C = \{x| \quad x \quad es \quad par\}$.

La notación por extensión sirve para definir conjuntos finitos y pequeños, mientras que la notación por comprensión se utiliza en la definición de conjuntos finitos muy grandes o conjuntos infinitos.

En la teoría de conjuntos existen dos conjuntos singulares que deben ser definidos, el primero es el conjunto vacío, este es el conjunto que carece de elementos y es denotado con el símbolo ϕ, el otro conjunto singular es el conjunto universal o referencial en el cual todos los elementos están contenidos en él, éste conjunto es diferente dependiendo del estudio que se este realizando se define con la letra E.

3.2. Alfabeto, cadenas y lenguaje

Para comprender lo expuesto en esta sección es necesario definir el concepto de símbolo, en este trabajo un símbolo es un trazo geométrico que tiene asociado un significado, por ejemplo, las letras y los números.

El alfabeto es un conjunto finito de símbolos y se denota con la letra griega $\sum$. Por ejemplo el alfabeto binario es un conjunto finito formado por los digitos 0 y 1 y se denota como $\sum = \{0, 1\}$.

Una cadena, también conocida como palabra, es una secuencia de símbolos yuxtapuestos, que son tomados de un alfabeto. Una cadena A tiene una longitud que se denota como $|A|$ y es el número de símbolos que componen a la cadena. Existe una cadena especial llamada cadena vacía denotada por el símbolo ξ y es aquella cadena que tiene una longitud de 0 es decir no esta formada por ningún símbolo.

Un lenguaje es el conjunto de cadenas tomadas de un alfabeto. El conjunto vacio ϕ y el conjunto formado por la cadena vacía $\{\xi\}$ también se consideran lenguajes.

3.3. Operaciones aplicadas a lenguajes

Varias operaciones de la teoría de conjuntos pueden ser aplicadas a los lenguajes. Se presentan adelante las mas importantes según Aho [3]:

1. Unión de conjuntos entre dos lenguajes L y M:

   $L \cup M = \{s \,|\, s \in L \; o \; s \in M\}$

2. Concatenación:

   $LM = \{st \,|\, s \in L \; y \; t \in M\}$

3. Cerradura de Kleene. La cerradura o unión de conjuntos de un lenguaje incluyendo la cadena vacía queda definida por:

   $L^* = \cup_{i=0}^{\infty} L^i$

4. Cerradura positiva. La cerradura positiva es muy similar a la cerradura de Kleene, con la diferencia de que en ésta no se incluye la cadena vacía y esta denotada por:

   $L^+ = \cup_{i=1}^{\infty} L^i$

3.4. Expresiones Regulares

Una expresión regular se forma con un conjunto de símbolos de un alfabeto. Es utilizada para representar patrones con varias aplicaciones modernas en la computación y en lenguajes de programación.

Una expresión regular compleja se define a partir de expresiones regulares simples que se unen con un conjunto de reglas.

Dado un alfabeto $\sum$ se establecen las reglas que definen las expresiones regulares:

1. ξ es una expresión regular designada por $\{\xi\}$ que representa el conjunto que contiene la cadena vacía.
2. Si a es un símbolo de $\sum$, entonces a es una expresión regular designada por $\{a\}$.
3. Sean r y s dos expresiones regulares que denotan a los lenguajes L y S, entonces:

- *(r)|(s)* es una expresión regular que denotan al conjunto $L \cup S$.
- *(r)(s)* es una expresión regular que denotan al conjuntoLS.
- *(r)** es una expresión regular que denotan al conjunto (L)*.

Las dos primeras reglas se refieren a símbolos básicos del alfabeto y son la base de la definición, mientras tanto la última regla denota las tres operaciones básicas que pueden ser utilizadas en la definición de las expresiones regulares, las cuales son:

1. La unión, que se define mediante el metacarácter : $|$.
2. La concatenación, que se define por la yuxtaposición de caracteres.
3. Cerradura de Kleene, la cual se indica mediante el metacarácter : *.

Para mostrar de forma más clara los conceptos mencionados anteriormente se presenta el ejemplo 1.

Ejemplo 1. Para el alfabeto binario definido por $\sum = \{0, 1\}$, se aplican las tres operaciones básicas.

1. La primera de las operaciones básicas es la unión que queda definida por la expresión regular $0|1$ la cual genera el siguiente conjunto de elementos $\{0, 1\}$
2. La segunda operación es la concatenación y queda definida por la expresión regular $(0|1)(0|1)$ la cual genera el conjunto $\{00, 01, 10, 11\}$
3. La tercer y última expresión regular es la cerradura de Kleene la cual queda definida por $1*$ y genera el conjunto $\{\epsilon, 1, 11, 111, ...\}$

Entre las 3 operaciones básicas, la cerradura tiene precedencia sobre la concatenación y sobre la unión. La concatenación tiene precedencia sobre la unión. Si se requiere modificar estas precedencias se debe recurrir al uso de paréntesis.

Por medio de estas tres operaciones es posible representar cualquier patrón, hay ocasiones en que una expresión puede resultar demasiado compleja para su construcción y puede ser difícil de entender lo que perjudica el mantenimiento de ella como se presenta en el ejemplo 2.

Ejemplo 2. Dado el alfabeto de todas las letras minúsculas del español $\sum = \{a, b, c, ..., z\}$ y el alfabeto de los números naturales $\prod$=$\{0, 1, 2, ...9\}$, se escribe una expresión regular, que reconozca identificadores donde el primer carácter de estos debe ser una letra y puede ser seguido de letras o dígitos.

Esto se puede realizar mediante la expresión regular:
$(a|b|c...|z)\,(a|b|c|...|z|0|1|...|9)$*

En el ejemplo 2 se observa que usando solo las tres operaciones básicas, se forman expresiones largas y complicadas, por tal motivo una expresión regular puede también estar formada de símbolos especiales denominados metacaracteres, de tal forma que las expresiones sean más claras y sencillas.

Los metacaracteres tipicos son:

- **Cero o una repetición**, dada una expresión regular para repetirla 0 o 1 vez se utiliza el metacarácter: ?.
- **Una o más repeticiones**, para definir una o más repeticiones de una expresión regular se utiliza el metacarácter: $+$.
- **Cualquier carácter**, para definir cualquier carácter que se encuentre en el alfabeto se utiliza el metacarácter punto : . .
- **Rango de caracteres**, se usan metacaracteres "$[-]$" , en donde se especifica el rango de caracteres a reconocer, si se desea excluir un rango de caracteres se utiliza el metacarácter "^" añadiendolo al principio del rango de caracteres.

Para quitar su significado especial a los metacaracteres hay que envolverlos entre comillas dobles: "", por ejemplo si necesitamos reconocer el símbolo $+$ tendríamos que definir el patrón de la siguiente manera: "$+$".

Ejemplo 3. Reformulando la expresión regular del Ejemplo 2 usando metacaracteres, la nueva expresión regular puede expresarse como:
$[a-z]\,([a-z]\,|\,[0-9])*$
de lo cual se obtiene una expresión más concisa y sencilla.

3.5. Gramáticas

Una gramática es una construcción matemática que define el conjunto de reglas que describen la estructura de una secuencia de símbolos que pertenecen a un lenguaje.

De manera formal Glenn [8] dice que se llama gramática a la cuádrupla:

$G = (\sum_T, \sum_N, S, P)$
donde:

$\sum_T$ es el alfabeto de símbolos terminales, y $\sum_N$ es el alfabeto de símbolos no terminales. Se verifica que:
$\sum_T \cap \sum_N = \phi$ y $\sum = \sum_T \cup \sum_N$.

$S \in \sum_N$ se conoce como símbolo inicial, o símbolo distinguido. P es un conjunto finito de reglas de produccion.

Notación de Forma Normal de Backus (en inglés *Backus Naur From*)

La Forma Normal de Backus (*FNB*) es una notación abreviada utilizada para representar las reglas de producción de una gramática [4].

La sintaxis de estas reglas son:

$$nombre_regla -> expresión1|expresión2$$

La regla está formada por dos partes separadas por la cadena "->" y es leida como nombre_regla produce o deriva "expresión1", la parte de la izquierda "nombre_regla" es un símbolo no terminal que define el nombre de la regla. La parte derecha correspondiente a las expresiones puede estar formada por símbolos terminales o símbolos no terminales. El símbolo "|" representa un "o" lógico, lo cual es equivalente a tener dos reglas de producción de la siguiente forma:

$$nombre_regla -> expresión1$$

$$nombre_regla -> expresión2$$

Las reglas especifican cómo se pueden combinar los terminales y los no terminales para formar cadenas, en este caso el "nombre_expresion" deriva o produce "expresión1 o expresión2".

En las reglas de producción se pueden encontrar todos lo elementos que conforman una gramática veámoslo en el Ejemplo 4.

Ejemplo 4. A partir de las siguientes reglas de producción:

$$1 : E- > E + T$$

$$2 : E- > T$$

$$3 : T- > T * F$$

$$4 : T- > F$$

$$5 : F- > (E)$$

$$6 : F- > a$$

se identifica el conjunto de símbolos terminales, símbolos no terminales , el símbolo inicial.

Por convención los no terminales se representan con letras mayúsculas mientras que los terminales se denotan con letras minúsculas.

El conjunto de símbolos terminales, son los símbolos básicos con los que se forma una cadena:

$$\sum T = \{+, *, (,), a\}$$

Los no terminales son variables que denotan conjuntos de cadenas que ayudan a definir el lenguaje:

$$\sum N = \{E, T, F\}$$

El símbolo inicial es un no terminal y corresponde a el símbolo con el que inicia la gramática, en este caso corresponde a:

$$S = \{E\}$$

Gramáticas Tipo 2 [8]

Si cada regla de producción en una gramática tiene la forma:

$$A- > \delta \qquad donde \quad A \quad \epsilon \quad N, \delta \quad \epsilon\,(NUT)^*$$

la gramática recibe el nombre de gramática libre de contexto [34].

Éstas gramáticas no tienen restricciones con respecto a la forma del lado derecho de sus reglas de producción, aunque tienen como restricción que el lado izquierdo éste formado unicamente de un no terminal. Se les llama libres de contexto debido a que puede aplicarse sin importar el contexto.

La sintaxis de los lenguajes de programación pueden ser descritos por las gramáticas libres de contexto, también la mayoría de los lenguajes humanos.

Derivación

Las reglas gramáticales determinan un lenguaje válido, y por medio de derivaciónes se determinan las cadenas sintácticamente legales.

Una derivación es una secuencia de reemplazos de los símbolos no terminales, por los símbolos terminales que se encuentran en los lados derechos de las reglas de producción, o la aplicación de las producciones de una gramática para obtener una cadena de terminales.

Con la derivación se puede comprobar si una determinada cadena pertenece o no a un lenguaje gramatical.

Se puede hacer derivación de dos modos distintos, por la derecha o por la izquierda. Cuando se hace una derivación por la izquierda se inicia tomando el símbolo inicial de la gramática, después se toma el símbolo terminal más a la izquierda y se remplaza por su terminal según convenga. Para realizar una derivación por la derecha de igual forma se comienza tomando el símbolo inicial de la gramática y después se toma el no terminal que se encuentre más a la derecha y es remplazado por el terminal que convenga. En ambos caso el proceso se repite recursivamente hasta obtener una sentencia de terminales.

Ejemplo 5. Dada la gramática del ejemplo 4, la derivación por la izquierda se muestra a continuación, para validar la cadena (a+a)*a.

Para llevar a cabo el proceso de derivación es necesario hacerlo desde el símbolo inicial, en éste caso se tienen dos posibles producciones para el símbolo inicial y se toma la regla:

$$E- > T$$

A partir de esta regla se elige el no terminal más a la izquierda para llevar a cabo la sustitución, en esta regla solo se tiene el no terminal T así que lo sustituiremos por la regla 3 la cual dice $T- > T * F$, aplicando esto:

$$^{3}- > T * F$$

note los caracteres "$^{3}- >$" ellos se leen: "aplicando la regla 3 de las reglas de producción se obtiene", notación que usaremos para denotar que se ésta llevando el proceso de derivación. Siguiendo el proceso se toma ahora al terminal T que es el que se encuentra más a la izquierda y se sustituye usando la regla 4 por:

$$^{4}- > F * F$$

el proceso se hace de manera recursiva:

$$^{5}- > (E) * F$$
$$^{1}- > (E + T) * F$$
$$^{2}- > (T + T) * F$$
$$^{4}- > (F + T) * F$$
$$^{6}- > (a + T) * F$$
$$^{4}- > (a + F) * F$$
$$^{6}- > (a + a) * F$$
$$^{6}- > (a + a) * a$$

Al finalizar la derivación se observa que a partir de dicha gramática fue posible llegar a la cadena con lo que se concluye que es aceptada dicha cadena.

Árbol de análisis sintáctico

Un árbol es una gráfica compuesta por nodos y arcos, un árbol comienza su construcción desde un nodo llamado raíz hasta las hojas del mismo. La raíz no tiene padres solo nodos hijos, veamos esto en la Figura 3.1.

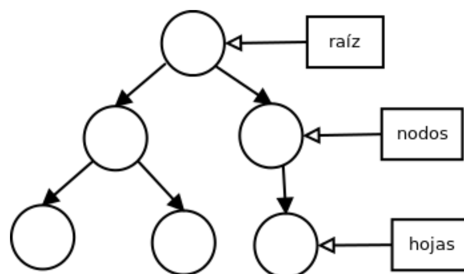

Figura 3.1. Árbol.

Un árbol de análisis sintáctico es aquel cuyas hojas interiores representan los terminales y los nodos hoja representan los no terminales, la raíz del árbol corresponde al símbolo inicial de la gramática y las hojas de cada nodo no terminal son los símbolos que reemplazan a ese no terminal, en la derivación.

Un árbol sintáctico puede ser visto como la representación gráfica al proceso de la derivación, tal como se observa en el siguiente ejemplo.

Ejemplo 6. Dada la derivación en el ejemplo 5, se construye un árbol de análisis sintáctico.

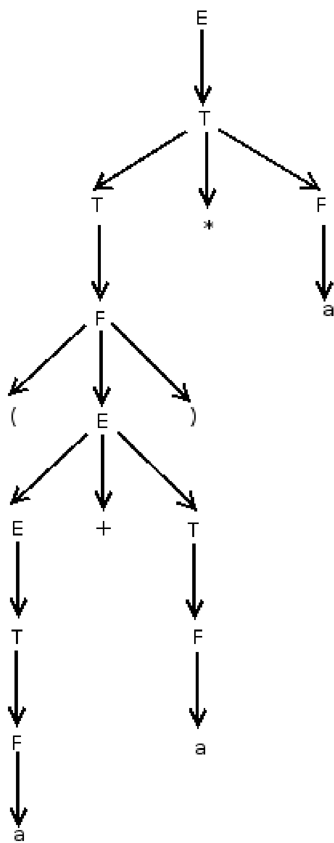

Figura 3.2. Árbol sintáctico.

Capítulo 4

Presentación del traductor salomeToROOT

Prácticamente todos los autores dedicados al estudio de compiladores, definen al compilador de manera similar como lo hace Aho [3] diciendo:

> "un compilador es un programa que lee un programa escrito en un lenguaje, el lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto; y como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente "

La mayoría de los compiladores tienen como objetivo realizar un puente de comunicación entre el ser humano y las computadoras, por tanto es común que se piense que la traducción vaya de un lenguaje de alto nivel a un lenguaje de bajo nivel sin embargo existen también compiladores que devuelven programas en alto nivel como lo son Flex y Bison [33].

Gráficamente la definición puede ser vista en la Figura 4.1.

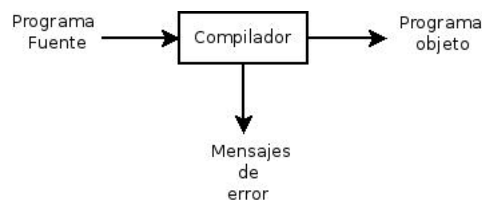

Figura 4.1. Proceso de compilación según Aho [3].

Para un ingeniero de software, un compilador es un programa complejo, debido a que está compuesto por un gran número de módulos, además que su construcción implica un gran estudio de ambos lenguajes: fuente y destino, manejo de errores, interacción con el usuario, etc... , de tal manera que, un sólo desarrollador no implementará un compilador por si mismo, sólo tomará aquellos módulos que necesite.

El proceso de compilación se puede agrupar en dos etapas principales:

1. Análisis
2. Síntesis

Estas dos etapas a su vez se subdividen en distintas fases como se puede ver en la Figura 4.2, en cada uno de los módulos se procesa y transforma el programa de entrada de una a otra representación. En la fase de análisis se busca verificar el lenguaje fuente.

A continuación se presentan las partes que conforman el análisis:

1. **Análisis léxico,** el archivo se lee de izquierda a derecha y se agrupa la salida en palabras con un significado especifico.
2. **Analisis sintáctico**, dadas las palabras obtenidas en la fase anterior, las secuencias de caracteres se reducen a cadenas siguiendo la gramática del lenguaje fuente para comprobar la instrucción.
3. **Analizador semántico**, la semántica de la instruccion se esta comprobando para verificar su significado.

Una vez que la parte del análisis indique la posibilidad de llevar cabo la traducción del lenguaje, será la parte de síntesis quien se encargue de realizar la traducción, a continuación se describe dicho proceso:

1. **Generación de código intermedio,** antes de entregar el código final algunos compiladores producen una representación intermedia con el fin de optimizar el código.
2. **Optimización de código,** mejora el código intermedio, de modo que resulte un código de máquina más rapido a la ejecución.
3. **Generación de código**, fase final del compilador donde se genera el código objeto.

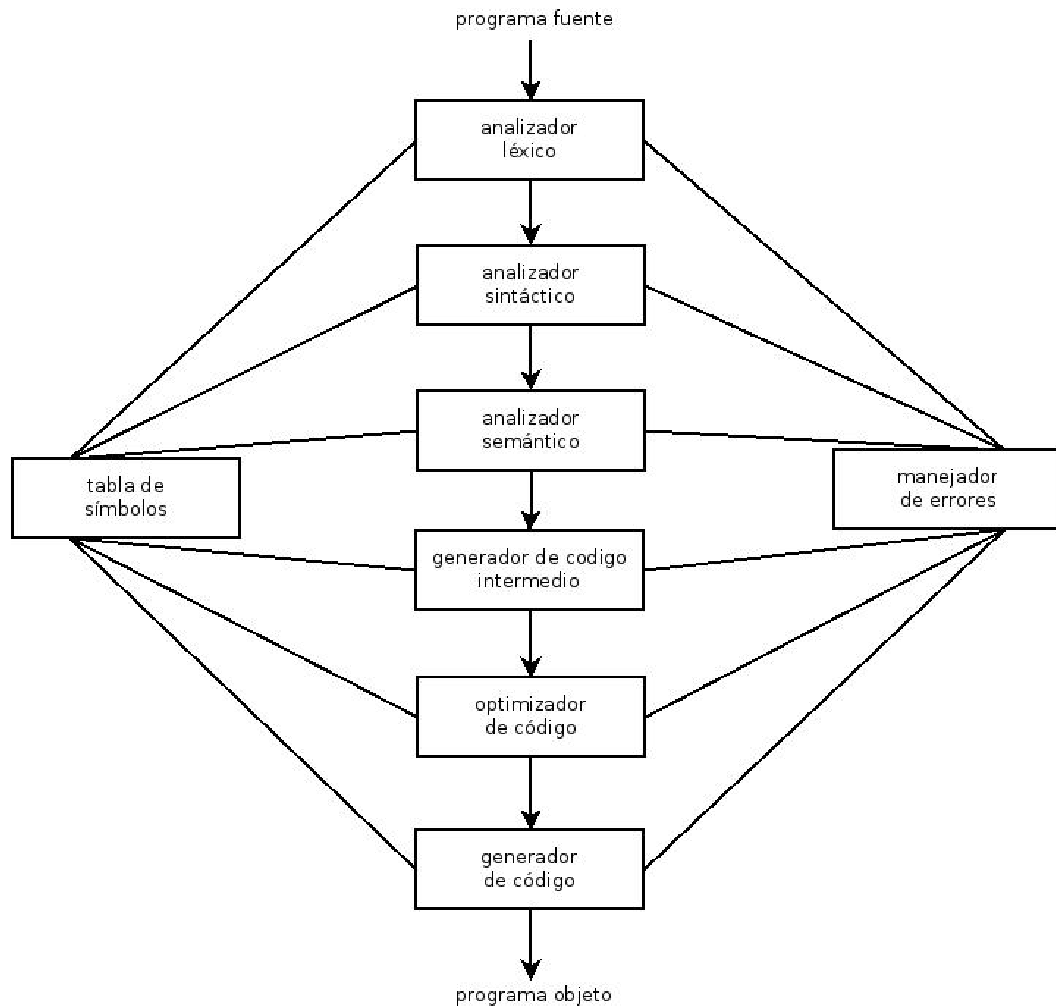

Figura 4.2. Fases de un compilador según Aho [3].

La tabla de símbolos y el manejador de errores se definen como:

1. **La tabla de símbolos**: estructura de datos dinámica que contiene los identificadores del programa de entrada.

2. **Detección y mensajes de errores**, el compilador recolecta e informa de todos los errores generados durante el proceso de compilación.

4.1. Análisis del problema

La plataforma Salome es una herramienta que utilizan ingenieros y técnicos para definir modelos geométricos y llevar a cabo análisis de mallas, para generar tales modelos se utiliza la filosofía de ir construyendo pieza a pieza sobre un plano de tres dimensiones, las cuales corresponden a altura, ancho, profundidad. La plataforma Salome puede ser usada de manera gráfica o a través de archivos "*.py" que contienen la o las geometrías a dibujar.

ROOT es por su parte un marco de trabajo utilizado por físicos en el cual se pueden definir modelos geométricos también. La filosofía de construcción consiste en crear un "mundo", el cual tendrá las piezas geométricas, este mundo puede tomar distintas formas y distintos tamaños dependiendo de las necesidades del usuario, en este caso se define un archivo "*.c", que contiene el mundo y la o las geometrías que forman parte del modelo geométrico.

Los ingenieros deben ser capaces de compartir sus modelos geométricos con los físicos, resulta necesario que los físicos puedan tener el código en lenguaje C++ para realizar ciertas modificaciones o añadir algunas funcionalidades, dado estas observaciones se propone la implementación de un traductor entre plataformas. Un primer paso, es la traducción de geometrías sencillas que son:

1. Cubo
2. Cilindro
3. Esfera
4. Toroide
5. Cono
6. Rectángulo
7. Disco

Para construir alguna de estas geometrías en la plataforma Salome, se ofrecen varios métodos distintos. Por ejemplo, para crear un cubo, se tienen tres diferentes métodos*:*

1. Uno con el cual podemos crear un cubo en el origen.
2. Otro que dados dos puntos del espacio crean un cubo.
3. Finalmente, uno más, que pide las coordenadas de un vértice.

Tantos métodos y la definición del sistema de coordenadas diferentes entre ROOT y Salome agregan complejidad al problema.

Este trabajo preliminar se limita a la traducción de geometrías, con un sólo método de creación sin respetar la disposición espacial del objeto geométrico.

4.2. Arquitectura de salomeToROOT

Un compilador es la mejor solución al problema, puesto que permite llevar un conjunto de instrucciones de un lenguaje fuente a un conjunto de instrucciones equivalentes a un lenguaje destino. No obstante un compilador es un programa complejo para ser desarrollado por una sola persona. Por lo tanto se propone tomar solo algunas partes de la arquitectura del compilador para llevar a cabo el proceso de traducción. La arquitectura propuesta para tal fin puede verse en la Figura 4.3.

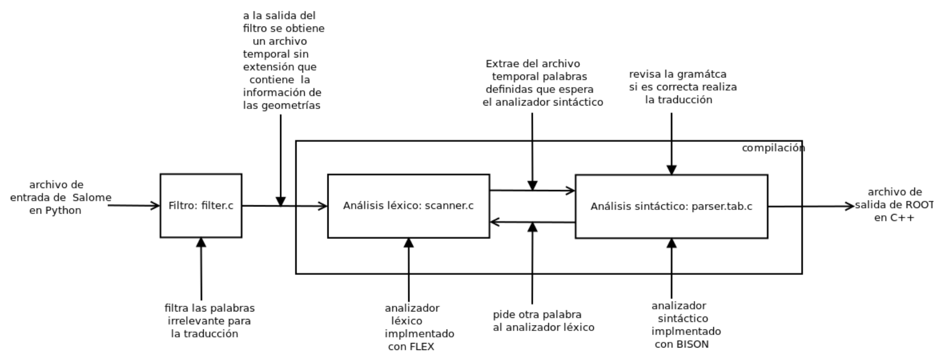

Figura 4.3. Arquitectura de salomeToROOT.

Filtro

Esta es la primera fase propuesta para llevar a cabo la traducción entre la plataforma salome y ROOT, tiene como objetivos:

- Comprobar que el archivo de entrada exista.
- Comprobar que el traductor sea alimentado con un archivo “*.py”.
- Eliminar algunas palabras innecesarias para el proceso de traducción.

A la salida de este módulo se obtendrá un archivo temporal que el analizador léxico leerá para continuar con el proceso de traducción.

Analizador Léxico

La segunda fase propuesta del traductor es un analizador léxico cuyo objetivo principal es leer caracteres para agruparlos en secuencias con un significado definido que serán la entrada del analizador sintáctico. En éste se eliminarán los espacios en blanco, los saltos de línea y los comentarios. Para la construcción de dicho analizador se eligió la herramienta Flex [2].

Analizador Sintáctico

La última fase del traductor es un analizador sintáctico cuyos objetivos son los siguientes:

- Comprobar la sintaxis de las geometrías.
- Manejo de errores.
- Traducir las geometrías a ROOT en un archivo con nombre.

Para la construcción de este analizador se eligió la herramienta Bison [20], a la salida de ésta fase se obtendrá la geometría en ROOT escrita en lenguaje C.

En los siguientes tres capítulos se define cada uno de los módulos del traductor.

Capítulo 5

Filtro

PAra entender la necesidad de construir un filtro que revise la geometría a traducir, se debe observar con mas atención la definición de una geometría en la plataforma Salome, en este caso la definición de un cubo es:

```
import geompy
import Salome
gg=Salome.ImportComponentGUI("GEOM")

#La filosofía para crear modelos geométricos
#en Salome se basa en crear las geometrías
#en un sistema de coordenadas de tres dimensiones

#Función para crear un cubo
#en el origen del sistema
#cada lado medira 200
caja1=geompy.MakeBoxDXDYDZ(200,200,200)

#Agregamos los objetos al estudio
id_caja1=geompy.addToStudy(caja1,"caja1")

#Dibujamos los objetos sobre OCC
gg.createAndDisplayGO(id_caja1)
gg.setDisplayMode(id_caja1,1)
```

Del código solo una línea contiene la información requerida para la traducción esta es:

```
caja1=geompy.MakeBoxDXDYDZ(200,200,200)
```

La información contenida en dicha línea indica el tipo de figura, sus dimensiones y el nombre con el cual se identifica.

Debido a la cantidad de datos innecesarios para la traducción se implementa como primer fase un filtro, el cual se encargará de discriminar datos de la entrada.

5.1. Expresiones regulares en C

Una de las principales aplicaciones para las expresiones regulares es la de validar y verificar datos de entrada, actualmente existen varios lenguajes especializados para su uso como lo son Perl y Python, éstas, también pueden ser utilizadas en lenguaje C usando la librería "regex", cuyas funciones principales son:

- regcomp, compila una expresión regular, escrita como una cadena, en un formato interno.
- regerror, transforma códigos de error devueltos por regcom() y regexec en mensajes legibles.
- regexec, compara una cadena con la expresión regular compilada previamente con regcomp(), si se encuentra una coincidencia se devuelve un 0 en caso contrario el valor será distinto de cero.

Estas funciones implementan el estándar 1003.2 ("POSIX.2") de la IEEE [27] el cual se usa para referirse a expresiones regulares.

Debido a todo esto se eligieron a la expresiones regulares como herramienta para la construcción del filtro.

5.2. Arquitectura del Filtro

Dada la naturaleza de la entrada del traductor, la cual es un archivo, es necesario que el filtro pueda determinar la existencia o inexistencia de la entrada, también que la extensión sea la correcta en este caso: ".py" por lo cual la arquitectura del filtro queda definida como sigue:

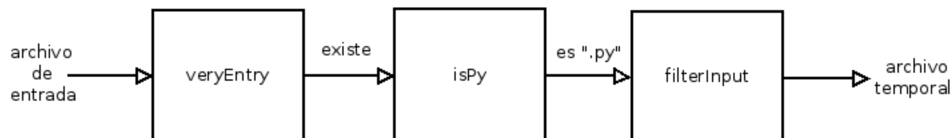

Figura 5.1. Arquitectura del filtro.

Cada módulo representado como una caja en la Figura 5.1, corresponde a una función en C, las cuales son descritas a continuación:

VeryEntry

Es una función entera, es la encargada de determinar la existencia del archivo de entrada, si éste no existe termina la ejecución del traductor y envía un mensaje de error, si existe llama a la función isPy.

isPy

Es una función entera, que toma como entrada el nombre del archivo y verifica que la extensión sea ".py" si es así devuelve un uno sino cero. Si la extensión es diferente termina la ejecución del traductor y envía un mensaje de error al usuario.

filterInput

Es la función principal. Esta fase del traductor, se encarga de filtrar las líneas que no son necesarias, recibe como entrada un archivo, verifica su existencia y su extensión con las funciones verEntry y isPy respectivamente para seguir con el filtrado si no es encontrado ningún error.

Para realizar el filtrado se toma una línea del archivo y es compara contra una expresión regular, si coincide con ésta, la línea es almacenada en un archivo temporal, en caso contrario, se toma otra línea de comparación hasta leer por completo el archivo.

La expresión regular utilizada en el filtrado es: .*Make.*, debido a que el Módulo geompy cuenta con más de 250 funciones para la visualización, la creación, la modificación y otras operaciones de un modelo geométrico empezando por "Make".

El código de este Módulo se proporciona en el Apéndice A.

5.3. Prueba al filtro

A continuación se muestra una prueba realizada al filtro, cuya entrada es el código fuente desarrollado por el usuario.

Para poder llevar a cabo la prueba del filtro se implemento una función main que llama al filtro, ésta es presentada a continuación:

```
#include <stdio.h>
#include "filter.h"
int main(int argc, char *argv[])
{
    filterInput(argv[1],argc);
  return 0;
}
```

Para está prueba se uso un cubo cuyo código es el siguiente:

```
import geompy
import salome
gg=salome.ImportComponentGUI("GEOM")
#Creamos un cubo
caja1=geompy.MakeBoxDXDYDZ(200,200,200)
#Agregamos los objetos al estudio
id_caja1=geompy.addToStudy(caja1,"caja1")
#muestra las cajas
gg.createAndDisplayGO(id_caja1)
gg.setDisplayMode(id_caja1,1)
```

Compilando:

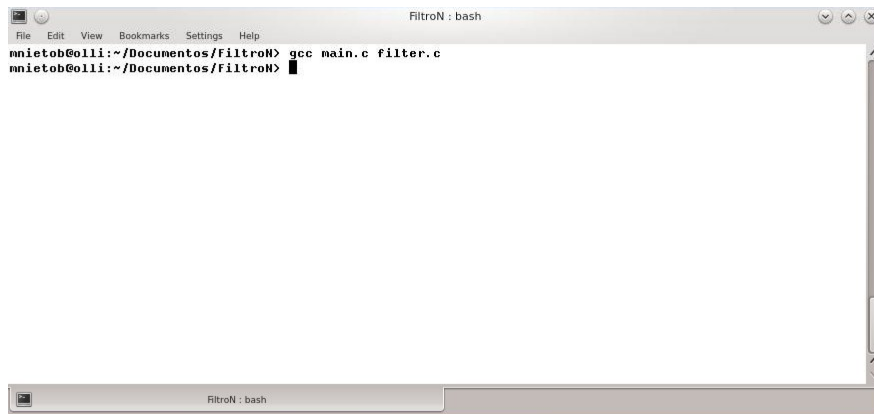

Figura 5.2. Compilando el Filtro.

Alimentado al filtro con entradas no validas se tiene:

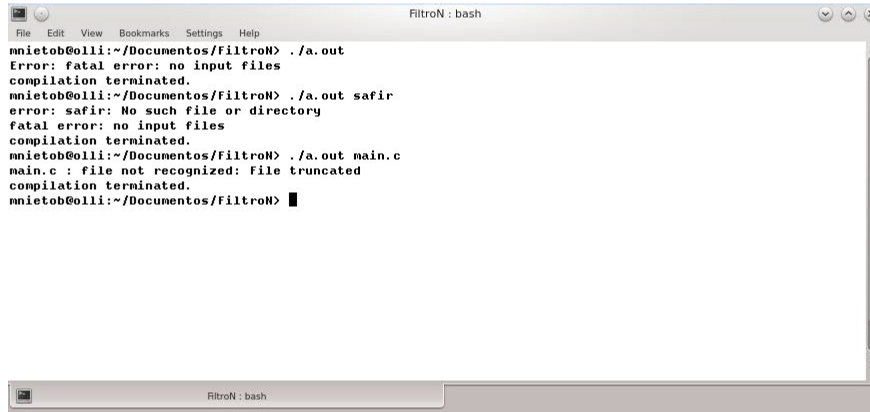

Figura 5.3. Entradas inválidas al Filtro.

Si es alimentado con una entrada válida, se obtiene un archivo temporal cuyo contenido es:

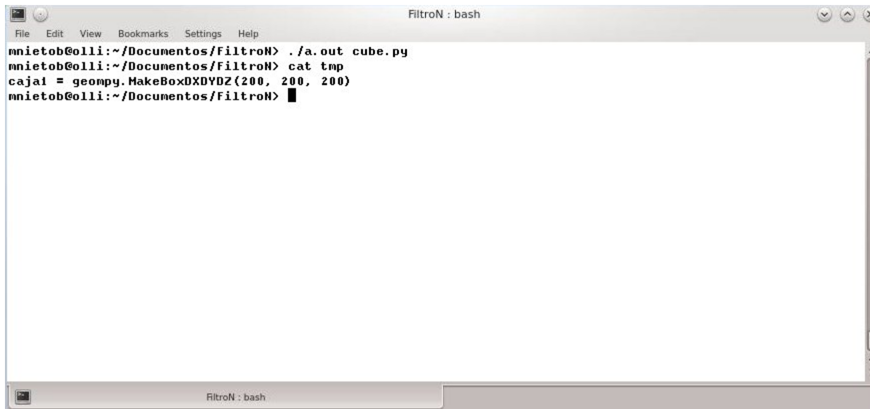

Figura 5.4. Salida Filtro.

Capítulo 6

Análisis Léxico

El analizador léxico es la primera fase de un compilador. Su principal tarea consiste en leer la entrada de izquierda a derecha y de arriba hacia abajo buscando caracteres para agruparlos en cadenas con un significado definido, las cuales serán manipuladas más tarde por el analizador sintáctico.

En esta fase:

- Los espacios en blanco son eliminados.
- Los comentarios son eliminados.
- Se agrupan caracteres en cadenas.
- Se crea y se llena la tabla de símbolos.

Este análisis puede ser implementado como parte del analizador sintáctico, no obstante se prefiere que se desarrollen como módulos independientes, esto permite simplificar la fase léxica y la fase sintáctica además de facilitar el mantenimiento del código.

6.1. Componentes Léxicos, patrón y lexema

La fase del análisis léxico; lee el archivo de entrada y agrupa los caracteres en secuencias especiales denominados componentes léxicos o del inglés “tokens” cada uno de éstos representa una unidad de información del código fuente. Los componentes léxicos caen en diversas categorías, pueden ser palabras reservadas, símbolos especiales, pueden representar un gran número de cadenas como los identificadores o los números.

Dado que un componente léxico puede caer en diversas categorías se necesita de una regla que sea capaz de distinguir entre grupos de componentes léxicos y los agrupe según su categoría. Por ejemplo si uno imagina que en un archivo C se encuentra las palabras “for” y “cubo”, el analizador léxico deberá ser capaz de reconocer y establecer que “for” es una palabra reservada y “cubo” es un

identificador.Tales reglas, que pueden distinguir entre componentes léxicos, son llamados patrones.

Cuando una cadena concuerda con un patrón se dice que se ha encontrado un lexema.

Los conceptos expuestos son ejemplificados en la siguiente tabla:

Componente Léxico	Patrón	Lexema
palabras reservadas	if,for,else...etc	if,for,else...etc
identificadores	letra seguida de letras y números	cubo,nombre...etc
números	digitos con o sin punto decimal	1.6180,9.81

Cuadro 6.1. Ejemplos componentes léxicos.

6.2. Manejo de errores léxicos

En este nivel son muy pocos los errores que se pueden detectar ya que el analizador léxico tiene una visión muy restringida de un programa fuente. Por ejemplo si se tiene lo siguiente:

fro(i=0; i<10; i++)
{
expresión...1
expresión...2
...
expresión n
}

Dado que el analizador léxico sólo reconoce los componentes léxicos de los cuales esta formado un archivo, no tendrá la capacidad para darse cuenta que "fro" es un error, puesto que "fro" cumple con el patrón de identificadores, todo lo que puede hacer el analizador léxico es dejar que otra fase del compilador se encarge de tratar dichos errores.

6.3. Flex

Del inglés "Fast Lexical Analizer" es una herramienta para la construcción de analizadores léxicos(*del inglés "scanners"*), éstos son generados a partir de un lenguaje especial denominado lenguaje LEX, que se basa en expresiones regulares, recibe como entrada un archivo de texto plano, con la extensión "*.l"

y después de compilarlo regresa como salida un analizador léxico escrito en lenguaje C.

En la Figura 6.1, se puede observar la secuencia de pasos a seguir, para desarrollar el archivo que contiene la definición del analizador léxico "scanner", se crea el archivo "*.l", éste pasara por el compilador Flex el cual nos entregara un archivo "*.c".

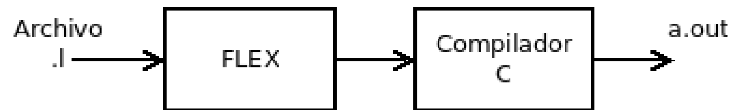

Figura 6.1. Proceso Flex.

Una vez hecho lo anterior, se puede ejecutar el programa como se puede ver en la Figura 6.2.

Figura 6.2. Ejecución scanner.

Por defecto, un analizador léxico desarrollado con Flex es alimentado desde la entrada estándar (el teclado), por tanto debe ser modificado para que lea desde un archivo.

Especificación de Flex

Un programa en Flex se divide en tres secciones, las cuales se puede ver en la Figura 6.3.

```
Declaraciones
%%
Reglas de Reconocimiento
%%
código de usuario
```

Figura 6.3. Estructura archivo "*.l".

Declaraciones

En esta sección se pueden hacer dos tipos de declaraciones:

1. Código en C global, el cual pasará exactamente al analizador léxico, regularmente en esta sección se declaran variables globales, funciones y directivas a través de la palabra clave "#include", ésta parte del código debe ir envuelta entre los caracteres %{ %} y de la siguiente forma:

   %{

   ...código c ...

   %}

2. Definiciones propias para expresiones regulares, en donde las definiciones tienen la siguiente forma:

   Identificador ExpresiónRegular

En donde el identificador es el nombre para referirse a la expresión regular como si fuera una variable, para utilizarla se tiene que envolver entre llaves {}. También se proporciona la forma de escribir comentarios sobre el archivo "*.l" encerrándolos entre los caracteres: /* */.

Reglas de Reconocimiento

En la segunda sección van los patrones que están asociados con los componentes léxicos a reconocer y que están definidos por expresiones regulares.

Resulta sumamente importante establecer correctamente las expresiones regulares para el reconocimiento de los componentes léxicos que se buscan, si no es así, el analizador léxico tendrá un funcionamiento erróneo.

La sintaxis de esta sección es la siguiente:

expresiónregular1 {acción}

expresiónregular2 {acción}

...

expresiónregularN {acción}

En donde expresionregular1 es el patrón a reconocer y la acción es lo que hará el analizador léxico una vez identificado el patrón.

Los metacaracteres para definir una expresion regular en Flex son representados a continuación:

<table>
<tr><th>Patrón</th><th>Definición</th></tr>
<tr><td>carácter</td><td>Cualquier carácter se representa a si mismo</td></tr>
<tr><td>.</td><td>Cualquier carácter excepto a una línea nueva</td></tr>
<tr><td>^</td><td>Representa el inicio de una línea</td></tr>
<tr><td>$</td><td>Representa el fin de una línea</td></tr>
<tr><td>[-]</td><td>Representa a un rango de caracteres</td></tr>
<tr><td>[^]</td><td>Todos los caracteres excepto los del rango</td></tr>
<tr><td>*</td><td>Concatenación del carácter 0 o más veces</td></tr>
<tr><td>+</td><td>Concatenación del carácter 1 o más veces</td></tr>
<tr><td>?</td><td>Representa a una opcion es decir 0 o 1</td></tr>
<tr><td>{m,n}</td><td>Concatenación de un carácter m o n veces</td></tr>
<tr><td>caráctercarácter</td><td>Concatenación de caracteres</td></tr>
<tr><td>cacarter|carácter</td><td>Es decir que Flex soporta la unión de caracteres</td></tr>
<tr><td>()</td><td>Para modificar la prioridad de los caracteres</td></tr>
</table>

Cuadro 6.2. Reglas para especificar las expresiones regulares en Flex.

Código de Usuario

Esta parte es opcional y contendrá código escrito en C por el usuario el cual sera copiado directamente al analizador léxico, usualmente es aquí en donde se define la función "main".

Si no se definió ninguna función "main", Flex proporciona una por defecto cuya definición se observa en la Figura 6.4.

Aquí se pueden definir otras funciones, con otros propósitos distintos dependiendo de la necesidad del compilador y del usuario.

```
main()
{
  yylex();
  return 0;
}
```

Figura 6.4. Función main() por defecto en Flex.

Directivas y funciones especiales de Flex

Existen directivas y funciones especiales que pueden ser incluidas dentro de una acción algunas de ellas son:

1. yytext apunta al texto del componente léxico actualmente reconocido.
2. ECHO copia el lexema encontrado en la salida del analizador de léxico.
3. BEGIN con una condición de inicio, posiciona ahí al analizador de léxico.

las funciones especiales son:

1. yymore() indica al autómata que la próxima vez que se reconozca un patrón, la salida se concatena a yytext en lugar de reemplazar el valor que tenía.
2. yyless(n) Regresa todos, excepto los primeros n caracteres de la entrada.
3. unput(c) Pone el carácter c de regreso en la entrada yyin, por lo que será el siguiente carácter escaneado.
4. input() lee el siguiente carácter de la entrada.
5. yyterminate() termina el funcionamiento del escáner y regresa un 0 al invocador, indicando que ha terminado. Por default, yyterminate es llamado cuando se encuentra un fin de archivo.

6.4. Analizador Léxico en Flex

El analizador léxico es implementado en el archivo “scanner.l” y se apoya sobre diferentes categorías las cuales son:

1. Palabras reservadas
2. Símbolos especiales
3. Identificadores
4. Números
5. Otros

La categoría 1 se especifica para las palabras reservadas que son las instrucciones encargadas de generar las distintas geometrías.

En la tabla 6.3 se presentan estas palabras más detalladamente:

Palabra Reservada	Breve Descripción
geompy	Es la clase del paquete geométrico
MakeBoxDXDYDZ	Crear un cubo con dimensiones especificadas
MakeCylinderRH	Crear un cilindro con radio y altura
MakeSphereR	Crear una esfera con radio
MakeConeR1R2H	Crear un cono con altura y radios
MakeTorusRR	Crear un toroide con un radio
MakeFaceHW	Crear una cara con dimensiones específicas
MakeDiskR	Crear un disco con las dimensiones especificadas

Cuadro 6.3. Categoría 1: Palabras Reservadas.

La segunda corresponde a los símbolos especiales presentados en la tabla 6.4.

símbolos	Descripción
(	Define el inicio de un método
)	Define el fin de un método
,	Define la separacion de argumentos
.	Define el uso de métodos
=	Define la operacion de asignacion

Cuadro 6.4. Categoría 3: Símbolos especiales.

En el caso de las categorías 3 y 4 no se puede crear una tabla finita para cada una de ellas pues la cantidad de números o identificadores es infinita. Finalmente la categoría 5 la cual puede ser vista en la tabla 6.5, recordemos que el analizador léxico también es encargado de eliminar comentarios y espacios en blanco, se añade una más para otros caracteres que podría introducir el usuario.

Otros	Descripción
Comentarios	Son las lineas que comienzan con #
Espacios en blanco	Son lineas vacias,tabuladores, o espacios en blanco
Cualquier otra cosa	Símbolos como ^,?,&,...,etc

Cuadro 6.5. Categoría 5: Otros.

La construcción del analizador léxico en Flex se centra en la sección de reglas. En esta se agregan los patrones que reconocerán los componentes léxicos. A continuación se presentan las expresiones regulares para cada lexema:

<table>
<tr><th>Posición</th><th>Lexema</th><th>Expresión Regular</th></tr>
<tr><td>0</td><td>geompy</td><td>geompy</td></tr>
<tr><td>1</td><td>MakeBoxDXDYDZ</td><td>MakeBoxDXDYDZ</td></tr>
<tr><td>2</td><td>MakeCylinderRH</td><td>MakeCylinderRH</td></tr>
<tr><td>3</td><td>MakeSphereR</td><td>MakeSphereR</td></tr>
<tr><td>4</td><td>MakeConeR1R2H</td><td>MakeConeR1R2H</td></tr>
<tr><td>5</td><td>MakeTorusRR</td><td>MakeTorusRR</td></tr>
<tr><td>6</td><td>MakeFaceHW</td><td>MakeFaceHW</td></tr>
<tr><td>7</td><td>MakeDiskR</td><td>MakeDiskR</td></tr>
<tr><td>8</td><td>(</td><td>“(”</td></tr>
<tr><td>9</td><td>)</td><td>“)”</td></tr>
<tr><td>10</td><td>,</td><td>“,”</td></tr>
<tr><td>11</td><td>.</td><td>“.”</td></tr>
<tr><td>12</td><td>=</td><td>“=”</td></tr>
<tr><td>13</td><td>Identificadores</td><td>([A-Za-z]|_)([A-Za-z]|[0-9]|_)*</td></tr>
<tr><td>14</td><td>Números</td><td>[0-9]+("."[0-9]+)?</td></tr>
<tr><td>15</td><td>Comentarios</td><td>#.*</td></tr>
<tr><td>16</td><td>Espacio en blanco</td><td>[\t\n]</td></tr>
<tr><td>17</td><td>Cualquier otro símbolo</td><td>.</td></tr>
</table>

Cuadro 6.6. Expresiones Regulares para cada categoría.

Es importante mantener la secuencia de los patrones cuando sean implementados en Flex, las palabras reservadas siempre anteceden al patrón de identificador, ya que si éste es puesto primero, cada vez que encuentre una palabra clave será reconocida como un identificador.

El código completo se proporciona en el apéndice A.

6.5. Prueba al analizador

Al observar el analizador en el Apéndice A, se notará que como acción de las expresiones regulares se envían datos con la sentencia “return” los cuales serán recibidos por el analizador sintáctico en este caso para llevar a cabo las pruebas, se modifican las secciones del archivo “scanner.l” y se añade una función “main” en la sección de código de usuario, veámoslo a continuación:

En la Figura 6.5 se muestra la sección de declaraciones.

```
%{
/*Define un analizador léxico para SALOME*/
#include <stdio.h>
%}

/*GENERALES*/
digito [0-9]
letra [A-Za-z]

/*númeroS*/
número {digito}+("."{digito}+)?

/*IDENTIFICADORES*/
id ({letra}|_)({letra}|{digito}|_)*
```

Figura 6.5. scanner.l sección de declaraciones.

Se añade la librería "stdio.h" y se definen las expresiones regulares generales que serán las encargadas de reconocer a los números y a los identificadores.

Sección de Reglas

En esta sección se modifican las acciones para que una vez reconocida una palabra se imprima en pantalla como se ve a continuación:

```
"geompy"            printf("\n%s es una palabra reservada \n",yytext);
"MakeBoxDXDYDZ"     printf("\n%s es una palabra reservada \n",yytext);
"MakeCylinderRH"    printf("\n%s es una palabra reservada \n",yytext);
"MakeSphereR"       printf("\n%s es una palabra reservada \n",yytext);
"MakeConeR1R2RH"    printf("\n%s es una palabra reservada \n",yytext);
"MakeTorusRR"       printf("\n%s es una palabra reservada \n",yytext);
"MakeFaceHW"        printf("\n%s es una palabra reservada \n",yytext);
"MakeDiskR"         printf("\n%s es una palabra reservada \n",yytext);
{id}                printf("\n%s es un identificador \n",yytext);
{número}            printf("\n%s es un número   \n",yytext);
"("                 printf("\n%s es un símbolo especial   \n",yytext);
")"                 printf("\n%s es un símbolo especial   \n",yytext);
","                 printf("\n%s es un símbolo especial   \n",yytext);
"."                 printf("\n%s es un símbolo especial   \n",yytext);
"="                 printf("\n%s es un símbolo especial   \n",yytext);
.                   {printf("\n%s es un ERROR\n",yytext);};
```

Código de usuario

La función "main" indica que la entrada será leída desde un archivo y llama a la función yylex() encargada del análisis léxico.

```
int main(int argc, char *argv[])
{
  if(argc==2)
  {
    yyin=fopen(argv[1],"r");
     if(yyin==NULL)
    {
      printf("El archivo %s no se puede abrir",argv[1]);
      exit(-1);
    }
  }
  else
    yyin=stdin;
  yylex();
  return 0;
}
```

Dado que el analizador léxico trabajará en conjunto con el filtro, se usará la salida del filtro para que sea analizada con el programa creado con Flex:

```
caja1=geompy.MakeBoxDXDYDZ(200,200,200)
```

Ejecutando el analizador:

```
asul@cmp120p094a000s:~/Documentos/Documentacion/compilador/lexico/alexicop> ./analizador tmp.txt

caja1 es un identificador

= es un simbolo especial

geompy es una palabra reservada

. es un simbolo especial

MakeBoxDXDYDZ es una palabra reservada

( es un simbolo especial

200 es un numero

, es un simbolo especial

200 es un numero

, es un simbolo especial

200 es un numero

) es un simbolo especial
asul@cmp120p094a000s:~/Documentos/Documentacion/compilador/lexico/alexicop>
```

Figura 6.6. Ejecución del analizador léxico.

Capítulo 7

Análisis Sintáctico

Es la segunda fase del compilador y la más importante debido a que es la encargada de determinar la sintaxis o la estructura de un programa a partir de los componentes léxicos reconocidos por un analizador léxico.

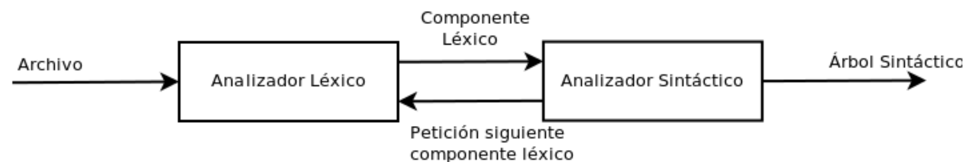

Figura 7.1. Interacción entre el analizador sintáctico y el analizador léxico (modificado de Aho [3]).

De la Figura 7.1, se puede observar que el analizador sintáctico recibe un componente léxico y una vez que lo utiliza, envia una petición al analizador léxico para que éste envie otro componente léxico y así recursivamente hasta que se termine de leer el archivo de entrada, una vez terminado este proceso el analizador sintáctico devuelve como salida un árbol.

Existen tres metódos distintos para realizar un análisis sintáctico estos son:

1. Universales
2. Descendentes
3. Ascendentes

En los métodos universales se tiene por ejemplo el algoritmo de Cocke-Younger-Kasimi el cual puede analizar cualquier gramática, lamentablemente

este tipo de método resulta ser ineficiente, por lo que no se usa en el desarrollo de compiladores [3].

Usualmente los métodos usados en los compiladores son los descendentes y ascendentes. En donde los descendentes construyen el árbol sintáctico desde la raiz hasta las hojas, mientras que el ascendente lo construye desde las hojas y sube hasta la raíz.

7.1. Análisis sintáctico descendente

También conocido del inglés como *top-down*, este análisis se basa en el reconocimiento de la cadena de entrada a partir del símbolo inicial de la gramática y sus derivaciónes por la izquierda, construyendo el árbol de análisis sintáctico de la raíz a las hojas.

Un analizador sintáctico esta formado por:

- Entrada.
- Pila.
- Programa para el análisis.
- Tabla de análisis.
- Salida

La entrada corresponde a la cadena a analizar, esta cadena debe ser seguida del símbolo que indica fin de cadena, cuando la pila se encuentre vacía y en la entrada sólo esté el fin de cadena, el analizador descendente habrá terminado exitosamente. La pila contiene una secuencia de símbolos gramáticales, al principio del análisis, la pila tendrá el símbolo inicial de la gramática, la tabla es una matriz bidimensional *M[A,a]* en donde A es un no terminal y a es un símbolo terminal o el símbolo.

Se presenta el proceso del análisis descendente en el ejemplo siguiente.

Ejemplo 7. Dada la gramática:

$1 : S- > dSA$
$2 : S- > bAc$
$3 : A- > dA$
$4 : A- > c$

se realiza un análisis descendente para la cadena *ddbccdcc*.

Dado que el análisis descendente se basa en el proceso de derivación se cominenza realizándolo:

$S -> dsA$
$^4 -> ddSAA$
$^2 -> ddbAcAA$
$^4 -> ddbccAA$
$^3 -> ddbccdAA$
$^4 -> ddbccdcA$
$^4 -> ddbccdcc$

Se construye una pequeña tabla para ir realizando el análisis sintáctico.

Entrada	Stack	Acción
ddbccdccΓ	S∇	1
dbccdccΓ	SA∇	1
bccdccΓ	SAA∇	2
ccdccΓ	AcAA∇	4
cdccΓ	cAA∇	3
dccΓ	AA∇	4
ccΓ	AA∇	4
cΓ	A∇	4
Γ	∇	ACEPTA

Cuadro 7.1. Análisis Descendente Ejemplo 4.

Para que una gramática pueda ser analizada por el método descendente debe cumplir ciertas características especiales. En general se tienen tres gramáticas, que se pueden observar en el organigrama de la Figura 7.2.

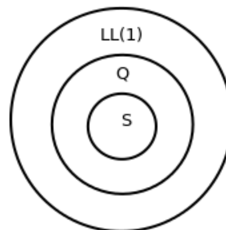

Figura 7.2. Organigrama descendente.

Las gramáticas *LL(1)* son un conjunto de las gramáticas libres de contexto, su nombre viene *Left-Left* en donde la primera L indica que la cadena será analizada de izquierda a derecha y la segunda L se refiere al hecho de que se realiza derivación por la izquierda. Para este tipo de gramáticas el 1 sólo indica que se tomará de 1 a 1 de la cadena de entrada.

Una gramática es *LL(1)* si y solo sí:

- Dos producciones que tienen el mismo lado izquierdo tienen conjuntos de selección disjuntos.

Una gramática es Q si cumple con:

- Sus producciones en el lado derecho inician con un *terminal* o son producciones ε.
- Si dos producciones tienen el mismo lado izquierdo entonces tienen conjuntos de selección disjuntos.

Se dice que una gramática es S si cumple con dos condiciones:

- Todas las reglas de producción comienzan con un símbolo terminal en su lado derecho.
- Si dos producciones tienen el mismo lado izquierdo entonces sus lados derechos deben ser distintos.

Un conjunto de selección, es un conjunto que contiene únicamente elementos de la gramática, éstos indicarán, qué acción se debe tomar en la tabla del analizador sintáctico, para esa producción. Para cada regla de producción tendremos un conjunto de selección y éstos no pueden estar vacíos.

Dado que los objetivos de este trabajo no son presentar todos los métodos descendentes, si no más bien dar una visión general de este tipo de análisis, no se profundizará en ellos.Para más información podría ser consultada la referencia [3].

7.2. Análisis sintáctico ascendente

También conocido como *Bottom-up*, este análisis consiste en crear el árbol de análisis sintáctico desde las hojas hasta la raíz, para esto toma la cadena de entrada e intenta llegar al símbolo inicial de la gramática.

Este tipo de análisis resulta ser mucho más potente que el análisis descendente y puede reconocer muchas más gramáticas pero con el costo de una implementación más compleja.

La implementación del analizador sintáctico se hace por medio de una pila la cual es la encargada de realizar el análisis sintáctico. La pila estará formada

por terminales, no terminales e información adicional. Al principio del análisis la pila está vacía y al final del mismo si es exitoso tendrá el símbolo inicial.

A parte de aceptar la cadena, un analizador sintáctico tiene dos acciones mas:

- Desplazar, transfiere un terminal de la entrada hasta la parte superior de la pila.
- Reducir, sustituye una cadena en la pila por un no terminal.

Debido a estas dos operaciones, también se el conoce como analizador sintáctico de reducción por desplazamiento del inglés *shift-reduce*.

Ejemplo 8. Dada la gramática siguiente:

```
1.E'->E
2.E->E+n
3.E->n
```

se realiza un análisis sintáctico ascendente de la cadena n+n.

Para llevar a cabo la operación de reducción se necesita identificar el mango *(del inglés handle)* de cada forma sentencial y para encontrar el mango es necesario identificar la frase de cada forma sentencial. La frase es la cadena que surge de aplicar derivación por la derecha y el mango es la frase que se encuentra más a la izquierda en cada forma sentencial.

Para identificar las frases y el mango de cada forma sentencial se realiza derivación por la derecha de la gramática para obtener la cadena *n+n*.

Derivación por la derecha	frases	handle
E+E	E+E	E+E
E+E*E	E*E	E*E
E+E*id1	id1	id1
E+id2+id1	id2,id1	id2
id3+id2+id1	id3,id2,id1	id3

Cuadro 7.2. Frases y Handle de la gramática.

Finalmente el proceso de análisis es:

Entrada	Estado de la Pila	Handle	Acción
n+nΓ	∇		desplaza
+nΓ	∇n	n	reducción por 3
+nΓ	∇E		desplaza
nΓ	∇E+		desplaza
Γ	∇E+n	E+n	reducción por 2
Γ	∇E	E	reducción por 1
Γ	∇E'		acepta

Cuadro 7.3. Acciones de un analizador sintáctico ascendente para el ejemplo 8.

Cuando se tiene en la entrada fin de cadena y en la pila el símbolo inicial, la cadena se acepta.

De la misma manera en que se tenían distintas gramáticas en el análisis descendente, aquí también se tienen distintas gramáticas, como se muestra en el organigrama 7.3. La gramática más general es conocida como *LR(Left to Right)* en donde la *L* se refiere a que la cadena es procesada de izquierda a derecha mientras que la *R* indica que realizan derivación por la derecha [3].

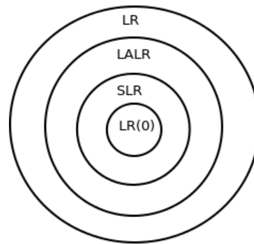

Figura 7.3. Organigrama gramáticas ascendentes.

Para construir un analizador sintáctico existen varios métodos estos son:

- ***SLR*** o ***LR simple*** es el método que reconoce menos gramáticas. Su objetivo es construir un autómata finito determinístico.
- ***LR*** o ***LR canónico*** es la técnica mas general que provee el análisis descendente y la más poderosa, debido a la gran cantidad de gramáticas que pueden ser analizadas con este método. No obstante, implementar este tipo de análisis resulta ser demasiado complejo.

- ***LALR*** o ***Lookahead*** este método tiene una complejidad y una potencia intermedia entre los métodos *SLR* y *LR*, funciona con la mayoría de las gramáticas de los lenguajes de programación.

Las Gramáticas libres de contexto se pueden categorizar dependiendo del análisis *LR* que les sea aplicado, es decir si para una gramática G, es posible analizarla por el método *SLR*, *LALR* o por *LR* sin conflictos se puede afirmar que esa gramática es *SLR*, *LALR* o *LR*.

Del mismo modo en que no se profundizó en el análisis descendente tampoco se hará aquí, de igual modo se invita al lector a revisar la bibliografía [37].

7.3. Diagramas de Sintaxis

Los diagramas de sintaxis o diagramas de tren, son una herramienta de gran utilidad puesto que sirven para representar gráficamente una gramática con sus respectivas reglas de producción.

Un diagrama de este tipo, va de un origen a un destino, se dibujan de izquierda a derecha, están compuestos por cajas que representan a los no terminales mientras los símbolos terminales son representados por óvalos.

Para entender mejor como crear los diagramas de sintaxis, se consideran los siguientes ejemplos. Una regla de producción como la que sigue:

$$C \rightarrow AB$$

Donde A y B son dos no terminales, por lo tanto el diagrama seria como en la Figura 7.4.

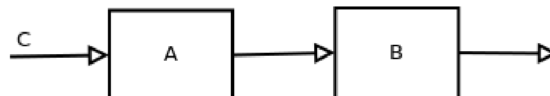

Figura 7.4. Diagrama para C según Ruíz [12].

Si se tiene que la regla de producción es:

$$D \rightarrow A|B$$

Cuando se usa el símbolo "|" en una gramática se indica la existencia de dos reglas de producción con el mismo símbolo inicial o:

$$D \rightarrow A$$

$$D \rightarrow B$$

El diagrama que representa este caso queda definido en la Figura 7.5.

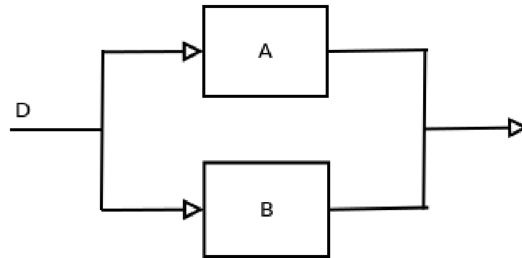

Figura 7.5. Diagrama para D->A|B según Ruíz [12].

En estos diagramas también se deben tener en cuenta construcciones de repetición o recursividad y para indicarla se usa el símbolo "*" de la siguiente manera:

$$D \rightarrow A*$$

El diagrama de sintaxis queda definido como en la Figura 7.6.

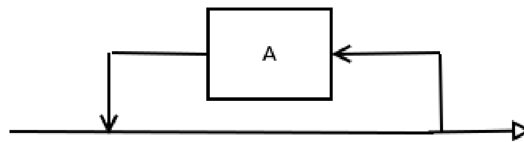

Figura 7.6. Diagrama para (A)* según Ruíz [12].

Si se tiene una regla de producción de la siguiente manera:

$$E \rightarrow A|\xi$$

El diagrama de sintaxis queda definido en la Figura 7.7.

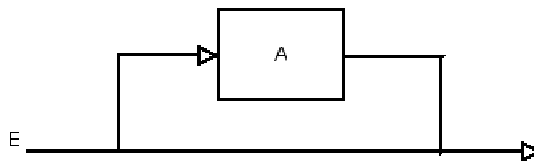

Figura 7.7. Diagrama para $E- > A|\xi$ según Ruíz [12].

7.4. Gramática para salomeToROOT

Para definir la gramática del traductor se deben tomar en cuenta las consideraciones de la tabla 7.4.

símbolos terminales y no terminales	Variable que representaran a los símbolos
Entrada	E
Auxiliar	A
Símbolo de inicio	S
Cubo	C
Cilindro	X
Cono	O
Esfera	S
Toroide	T
Rectangulo	R
Disco	D
Geometria	G
número	n
Identificador	i
geompy	g
MakeBoxDXDYDZ	m
MakeCylinderRH	c
MakeShpereR	s
MakeConeR1R2H	o
MakeTorusRR	t
MakeFaceHW	r
MakeDiskR	d

Cuadro 7.4. Consideraciones de nuestra gramática.

Símbolos terminales de la gramática

El conjunto de símbolos terminales de la gramática es:
$\sum_T = \{i, n, "(", ")", ".", ",", g, m, c, s, o, t, r, d\}$

Símbolos no terminales de la gramática

El conjunto de símbolos no terminales de la gramática es:
$\sum_N = \{E, A, C, X, O, S, T, R, D, G\}$

Símbolo inicial

El símbolo inicial es:
$S = \{S\}$

Reglas de producción

Para definir las reglas de producción se diseñan los diagramas de sintaxis de cada primitiva.

Cubo:

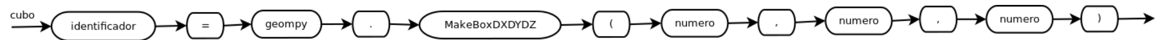

Figura 7.8. Diagrama de sintaxis del Cubo.

Del diagrama visto en la Figura 7.8, se puede ver que la regla de producción correspondiente para el cubo es:

$$cubo- > identificador = geompy.MakeBoxDXDYDZ(numero, numero, numero)$$

Cilindro:

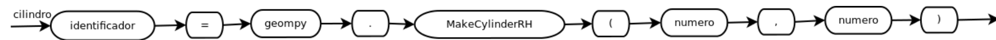

Figura 7.9. Diagrama de sintaxis del Cilindro.

Del diagrama visto en la Figura 7.9, se puede ver que la regla de producción correspondiente para el cilindro es:

$$cilindro- > identificador = geompy.MakeCylinderRH(numero, numero)$$

Esfera:

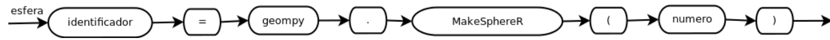

Figura 7.10. Diagrama de sintaxis Esfera.

Del diagrama visto en la Figura 7.10, se puede ver que la regla de producción correspondiente para la esfera es:

$$esfera- > identificador = geompy.MakeSphereR(numero)$$

Cono:

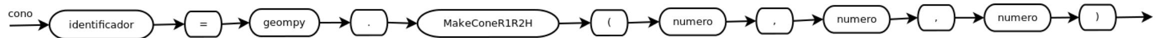

Figura 7.11. Diagrama de sintaxis del Cono.

Del diagrama visto en la Figura 7.11, se puede ver que la regla de producción correspondiente para el cono es:

$$cono- > identificador = geompy.MakeConeR1R2H(numero, numero, numero)$$

Toroide:

Figura 7.12. Diagrama de sintaxis del Toroide.

Del diagrama visto en la Figura 7.12, se puede que la regla de producción correspondiente para el toroide es:

$$toroide- > identificador = geompy.MakeTorusRR(numero, numero)$$

Rectángulo:

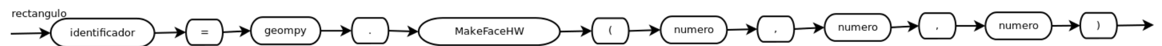

Figura 7.13. Diagrama de sintaxis del Rectángulo.

Del diagrama visto en la Figura 7.13, se puede ver que la regla de producción correspondiente para el rectangulo es:

$$rectangulo- > identificador = geompy.MakeFaceHW(numero, numero, numero)$$

Disco:

Figura 7.14. Diagrama de sintaxis del Disco.

Del diagrama visto en la Figura 7.14, se puede ver que la regla de producción correspondiente para el disco es:

$$disco- > identificador = geompy.MakeDiskR(numero, numero)$$

Finalmente agrupadas las reglas de producción para las primitivas quedan definidas en la Figura 7.15.

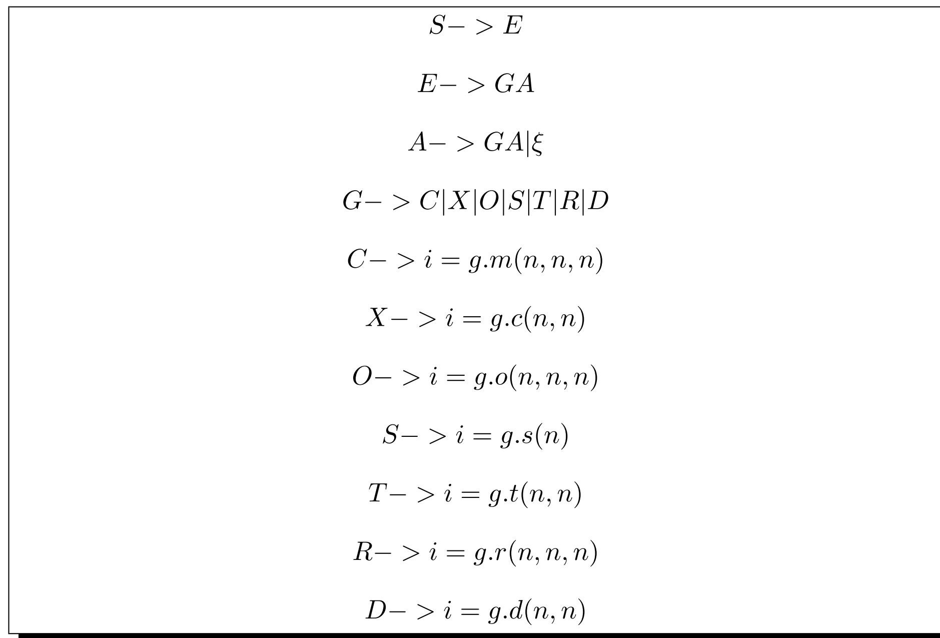

Figura 7.15. Reglas de producción para primitivas geométricas.

De la Figura 7.15, se observa que el símbolo inicial sólo tiene una producción, el símbolo no terminal E define una entrada que produce una geometría o un auxiliar, después se define el auxiliar, el que puede ser una geometría seguida de si mismo, lo cual da a la gramática recursividad para permitir que se acepte más de una geometría, una geometría puede ser un cubo, cilindro, cono, esfera, toroide, rectángulo o disco y finalmente se agregan a la gramática las reglas de cada primitiva geométrica.

7.5. Bison

Bison es una herramienta desarrollada por el proyecto GNU que permite generar analizadores sintácticos de propósito general. Para que Bison pueda analizar un lenguaje, éste se debe poder describir por una gramática libre de contexto, particularmente un lenguaje que pueda ser analizado con el algoritmo *LALR*.

A pesar de estar distribuida bajo la licencia GPL se pueden utilizar los analizadores sintácticos construidos por Bison en software no libre [20].

Es totalmente compatible con la herramienta Yacc de Unix [33], por lo cual todas las gramáticas escritas en Yacc funcionarán sin problema en Bison, si ésta se combina con Flex se puede construir un compilador.

Bison toma como entrada una gramática de contexto libre definida en un archivo y la transforma a un programa en lenguaje C que sea capaz de analizar dicha gramática.

El analizador léxico devolverá un conjunto de componentes léxicos, estos serán validados por un analizador sintáctico.

Para construir dicho analizador, se debe escribir un archivo con la extensión ".y" expresando la gramática del lenguaje a analizar, éste archivo se divide en tres partes como se muestra en la Figura 7.16.

```
%{
    prologo
%}
declaraciones
%%
descripcion gramátical
%%
código de usuario
```

Figura 7.16. Estructura Bison.

Para separar cada sección de otra se utlizará el símbolo " %%".

Prólogo

El prólogo del archivo Bison comienza con el símbolo " %{". En éste se pueden definir macros, prototipos de funciones y declaraciones de variables, que serán ocupadas mas tarde por las reglas gramáticales, para cerrar este bloque se coloca el símbolo " %}".

En caso de no necesitar ninguna declaración de este tipo es posible omitir este bloque.

Declaraciones

El objetivo de esta sección es declarar los símbolos terminales que usará la gramática, todo lo que no sea declarado como símbolo terminal Bison lo considerará como un símbolo no terminal y por lo tanto tendrá que haber una regla gramátical para él.

En la gramática reconocida por Bison, a los símbolos terminales se les llama también *tokens* y deben declararse en la sección de declaraciones. Por convención se escriben a los símbolos terminales en mayúsculas y a los símbolos no terminales en minúsculas.

Existen tres formas para escribir los símbolos terminales en una gramática definida por Bison, estas son:

- ***Token* designado**, se define en esta sección como si se tratara de un identificador en C, se debe definir con una declaración de Bison, como “ %token”.
- ***Token* carácter**, se escriben en la gramática directamente utilizando la sintaxis de C para los caracteres, es decir ubican al carácter entre comillas simples. Éste no necesita ser declarado a menos que se necesite asociar un valor a él. Por conveniencia se utiliza únicamente para definir un *token* que consiste sólo en un carácter, por ejemplo los caracteres aritméticos ’+’,’-’,’*’ y ’/’.
- ***Token* cadena,** se define cono una cadena en C es decir entre comillas dobles. De igual modo no necesita ser declarado a menos que se desee asignarle un valor particular. Este tipo de *tokens* no funcionan en Yacc, por convención, se usan para definir una cadena en particular. Por ejemplo para representar el símbolo “<=”.

La sintaxis para declarar un *token* es como sigue:

```
%token nombre
```

Bison convertirá esta definición en un macro “#define”, de manera que la función “yylex()” podrá utilizarla para representar el código de este *token.*

Se puede usar también las directivas “ %left”, “ %rigth” o “ %nonassoc” si lo que desea es especificar precedencia.

Descripcion gramátical

La definición de esta sección es de carácter obligatorio y ésta formada por las reglas de producción gramátical, en esta sección debe haber al menos una regla de producción.

Para definir una regla gramátical se usa la notación BNF, la sintaxis en Bison para esta notación es:

```
resultado:componente1{acción1}|componente2{acción2}|...|componenten{acciónn};
```

donde, resultado, es un símbolo no terminal que describe a las reglas y los componentes, son los diversos símbolos terminales y no terminales que están reunidos por esta regla. Se pueden añadir tantos espacios a la regla de producción como sean necesarios. Con estos sólo se da claridad a las reglas y no serán tomados en cuenta por Bison para generar el analizador sintáctico.

Con esto Bison creará un programa capaz de analizar las reglas gramáticales definidas por el usuario. Sin embargo, para que pueda realizar un trabajo, éstas reglas deberán ir acompañadas de acciones en lenguaje C que determinan las tareas a realizar. Estas deben ir entre llaves "{}". Varias reglas que compartan su lado derecho pueden ser escritas de manera independiente o pueden unirse con el carácter "|".

Una regla es recursiva cuando su no terminal aparezca también en su lado derecho. Casi todas las gramáticas hacen uso de esta característica debido a que es la única forma de definir una secuencia de cualquier número de cosas.

Código de usuario

En esta sección, se puede agregar código C. El código definido en esta sección sera copiado exactamente igual al analizador definido por Bison. Es, en esta sección, donde usualmente se define la funcion "main", la funcion "yylex" que corresponde al analizador léxico y la funcion "yyerror" la cual se encargará del tratamiento de errores.

Al igual que otras secciones ésta puede ser omitida por el usuario, si se hace esto Bison proporcionará por defecto una funcion "main", la cual se encargara de llamar a la función "yyparse" la encargada de llevar a cabo el análisis sintáctico.

Bison entrega al programador una serie de macros y variables las cuales comienzan con "yy" o "YY", una buena práctica de programacion es evitar definir variables o funciones que inicien de ese modo.

Valores Semánticos

Las reglas gramáticales determinan únicamente la sintaxis. La semántica es determinada por los valores semánticos asociados con *tokens* y por las acciones tomadas cuando varias reglas son reconocidas. Por ejemplo en una entrada el analizador léxico encuentra el número 5, su tarea es informar que se encontró un número, pero para trabajar también se necesita el número en sí mismo por lo tanto el analizador sintáctico debe recibir la instrucción de que se reconoció un número pero también el valor de dicho número.

En un programa simple puede ser suficiente utilizar el mismo tipo de datos para los valores semánticos de todas las construcciones del lenguaje. Bison nor-

malmente utiliza el tipo entero “int” para los valores semánticos si su programa utiliza los mismos tipo de datos para todas las construcciones del lenguaje. Para especificar algún otro tipo, se usa la macro “YYSTYPE” de la siguiente manera:

```
# define YYSTYPE doble
```

“YYSTYPE” lista de reemplazo debe ser un nombre de tipo que no contiene paréntesis o corchetes. Esta definición de la macro debe ir en el prólogo del archivo.

En la mayoría de los programas, se necesitan diferentes tipos de datos para diferentes tipos de componentes léxicos. Por ejemplo, una constante numérica puede necesitar tipo “int” o “long int” entero largo, mientras que una cadena necesita constantes de tipo “char *”. Para utilizar más de un tipo de datos en los valores semánticos en un analizador, Bison pide que se hagan dos cosas:

- Especificar la colección completa de tipos de datos posibles, en una declaración unión de Bison. Por ejemplo:

  ```
  %union
  {
          double typeReturn;
          char *string;
  }
  ```

- Elegir uno de los tipos para cada símbolo (terminal) para que los valores semánticos se utilicen. Por ejemplo:

  ```
  %token <string>IDENTIFIER
  ```

Después cualquier referencia a un valor es a través de la notación “$$”, “$1”, etc..., el cual accede al campo de la estructura unión.

Recuperación de Errores

No es aceptable que un programa termine en un error de sintaxis. Por ejemplo, un compilador debe ser capaz de recuperarse lo suficiente como para analizar el resto del archivo de entrada y revisar si hay más errores. En Bison la recuperación del error se hace a través de producciones de error. Se eligen, los no terminales que tendrán recuperación de errores, después de debe añadir producciones de la forma $A \rightarrow error$, donde A es un no terminal y error es una palabra reservada de Bison a partir de ésto se genera un analizador sintáctico pero considerando las producciones de error como si fueran producciones normales.

Función yyerror()

Si Bison detecta un error sintáctico, es decir, recibe un componente léxico que no espera, llama a la función “yyerror()”. Esta función debe ser proporcionada por el desarrollador, el prototipo de la función es:

- void yyerror(char *s);

La implementación debe añadirse en la sección de usuarios.

7.6. Analizador Sintáctico en Bison

Consiste en definir las tres secciones del archivo “parser.y” se comienza con la sección de declaraciones. En el prólogo se define el prototipo de la función error que se debe implementar. Más tarde se define el tipo de valores semánticos a recibir con la estructura unión y los símbolos no terminales, de la siguiente manera:

```
%{
  void yyerror(char const *);
%}

%unión
{
double typeReturn;
char *string;
}
/*general tokens*/
%token <string>IDENTIFIER
/*general tokens geometry*/
%token GEOMPY
/*Numbers*/
%token <typeReturn> NUM
/*Tokens for a cube*/
%token MAKEBOXDXDYDZ
.
.
.
```

A continuación se definen las reglas de producción usando la gramática de la sección 7.4. Se incluye una producción de error indicando que la geometría enviada puede estar mal escrita, o no existe.

```
/*Defined start symbol*/
start: input;

/*Define input*/
input:geometry aux
     |
     ERROR
     {
       yyerror("\n");
       yyerrok;
       fprintf(stderr,"error: %s: unknown symbol \n",$1);
     };

/*Define aux*/
aux:/*cadena vacía*/|geometry aux;

/*We define geometry types*/
geometry : cube|cylinder|sphere|cone|torus|face|disc|error;

/*Grammar rule for a cube*/
cube:IDENTIFIER '=' GEOMPY '.' MAKEBOXDXDYDZ '(' NUM ',' NUM ',' NUM ')'
{
fprintf(archSal,"\t//We define a geometry \n");
fprintf(archSal,"\tTGeoVolume *%s=gGeoManager->//
MakeBox(\"%s\",med,%lf,%lf,%lf);\n",getID($1),getID($1),$7,$9,$11);
fprintf(archSal,"\ttop->AddNode(%s,0, transf);\n\n",getID($1));
};
.
.
.
```

Si se observa de nuevo la sección de reglas se notará que si es reconocido un cubo, este cambiara a código de ROOT y se guarda en un archivo.

El código completo esta proporcionado en el Apéndice A. Para trabajar el analizador sintáctico creado por Bison requiere un analizador léxico, por tal motivo en este capítulo no se realizaron pruebas.

Capítulo 8

Integración y liberación

Se ha llegado a la última parte del desarrollo con la integración y la liberación de la herramienta. Este capítulo se centra en la integración de todos los módulos, es decir, el filtro, el analizador léxico realizado con Flex y el analizador sintáctico realizado con Bison. Se llevan a cabo pruebas de integración y para terminar se ponen en contexto los requerimientos necesarios para liberar un proyecto de software bajo una licencia libre de GNU.

8.1. Integración y Pruebas

La integración del software es el proceso de unir todos los módulos desarrollados de manera independiente para obtener una sola herramienta. En este caso se unieron el filtro, al analizador léxico y el analizador sintáctico respetando los pasos siguientes:

1. Implementar la función principal del traductor.
2. Implementar funciones extras e incluirlas en el archivo "parser.y".
3. Generar el archivo "parser.tab.h" que contiene la definición de los componentes léxicos y se crea con la opción de compilación Bison "-d".
4. Incluir en la sección de declaraciones del "scanner.l" el archivo ".h".
5. En la acción de cada patrón del "scanner.l" se contemplan los siguientes pasos:

    a) Devolver el componente léxico encontrado mediante una orden "return TOKEN".

    b) Cargar en el campo que corresponda de la variable "yyval" los valores de los atributos léxicos que fueron reconocidos.

6. Compilar cada módulo y crear un sólo archivo ejecutable llamado "salomeToRoot".

Función Principal

Todo proyecto que quiera ser ejecutado, debe incluir la función especial "main" en algún lado del desarrollo, está función se anexa en el código de usuario del archivo "parser.y". La función principal llama al filtro, si éste, puede completar su ejecución regresa el control a la función "main" la cual crea un archivo ".c" para almacenar la nueva geometría escrita en C++. Por ultimo elimina el archivo temporal "tmp" creado por el filtro.

Funciones Extras

Existen ciertas acciones extras que deben ser realizadas para llevar a cabo la traducción, por lo cual se incluye también un archivo llamado "functions.c". Este archivo contiene 4 funciones más que son:

- getName, char *getName(char name[]), devuelve una cadena y es usada por "main" para nombrar el nuevo archivo ".c".
- getID, char *getID(char name[]), toma como entrada una cadena y si ésta contiene el símbolo "=" la divide y la devuelve. Es usada por la función "yyparse" para poner nombre a las primitivas.
- deleteFile, void deleteFile(char *name), elimina el archivo temporal creado por el filtro.
- captureWorld, void captureWorld(double *num1,double *num2,double *num3). Como se mencionó anteriormente ROOT necesita un mundo para dibujar las primitivas, el traductor construirá dicho mundo como un cubo y con unas dimensiones indicadas por el usuario. Esta función toma de la entrada estandar los valores y los devuelve a la función principal, para ser añadidos.

Esta archivo se añade al prólogo del "parser.y" por medio de un archivo de cabecera.

Macros

Bison exige que se le proporcione una función de nombre "yylex" encargada de llevar a cabo un análisis léxico. Por defecto Flex y Bison fueron diseñados para trabajar juntos por lo cual se requiere un archivo que contenga la definición de los componentes léxicos. Bison genera automáticamente un archivo de cabecera que contenga esta información, por defecto de nombre "y.tab.h". En el caso de traductor su nombre es "parser.tab.h" cuya parte más importante del contenido es el siguiente:

```
# define YYTOKENTYPE
/* Put the tokens into the symbol table,
so that GDB and other debuggers
know about them. */
enum yytokentype
{
IDENTIFIER = 258,
GEOMPY = 259,
NUM = 260,
MAKEBOXDXDYDZ = 261,
MAKECYLINDERRH = 262,
MAKESPHERER = 263,
MAKECONER1R2H = 264,
MAKETORUSRR = 265,
MAKEFACEHW = 266,
MAKEDISKR = 267,
ERROR = 268
};
#endif
/* Tokens. */
#define IDENTIFIER 258
#define GEOMPY 259
#define NUM 260
#define MAKEBOXDXDYDZ 261
#define MAKECYLINDERRH 262
#define MAKESPHERER 263
#define MAKECONER1R2H 264
#define MAKETORUSRR 265
#define MAKEFACEHW 266
#define MAKEDISKR 267
#define ERROR 268
```

Este archivo es incluido en el "scanner.l" que necesita las definiciones de los componentes léxicos, para devolver el tipo de componente léxico reconocido.

Adicionalmente algunos de los componentes léxicos deben enviar también un valor semántico, el cual es almacenado en la variable global "yylval" , el tipo de esta variable es "YYSTYPE" o unión definidos en el prólogo.

Los pasos para crear el archivo ejecutable se observan en la Figura 8.1, y estan descritos en el apéndice D.

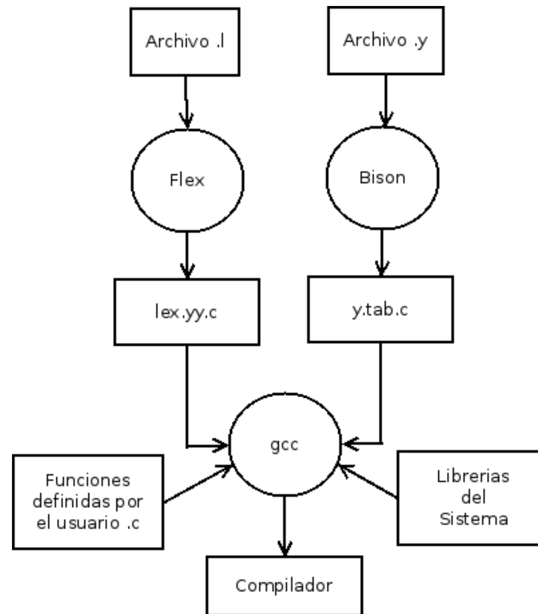

Figura 8.1. Yacc/Bison. Recuperado de [18].

Prueba

La ejecución e instalación del traductor estan detallados en el apéndice C. Para ésta prueba se muestra el código que define un toroide el cual es:

```
import geompy
import salome
gg = salome.ImportComponentGUI("GEOM")

#creamos los toroides
torus2 = geompy.MakeTorusRR(300, 150)

#agregamos los objetos al estudio
id_torus1 = geompy.addToStudy(torus1,"Torus1")

#dibujamos los toroides
gg.createAndDisplayGO(id_torus1)
gg.setDisplayMode(id_torus1,1)
```

Dibujando con la plataforma Salome queda como:

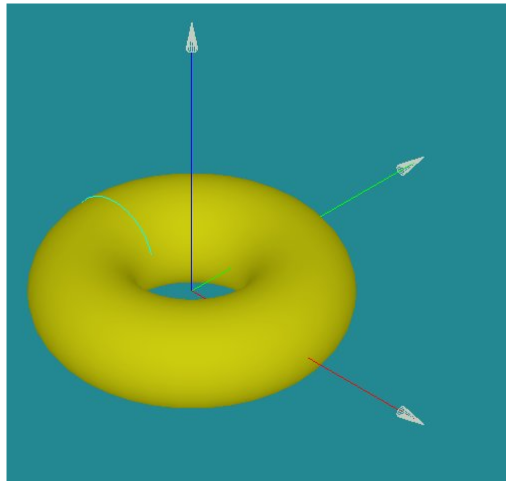

Figura 8.2. Toroide en la plataforma Salome.

Ejecutando la herramienta salomeToROOT de la siguiente forma:

(mnietob) olli

Archivo Editar Ver Marcadores Preferencias Ayuda

```
mnietob@olli:~/Documentos/prueba> ls
torus.py
mnietob@olli:~/Documentos/prueba> salomeToRoot torus.py
Dimensions of the world in ROOT
size in x:
500
size in y:
500
size in z:
500

temporary file removed

mnietob@olli:~/Documentos/prueba> ls
torus.c  torus.py
mnietob@olli:~/Documentos/prueba> 
```

(mnietob) olli

Figura 8.3. Ejecución salomeToROOT.

La salida del traductor es un archivo "*.c" que contiene la especificación del toroide reconocido por ROOT:

```
#include "TGeoManager.h"
void torus()
{
  //We call the admin. geometry
  new TGeoManager("world","simple");

  //We define the means and the material
  TGeoMaterial *mat=new TGeoMaterial("Vacuum",0,0,0);
  TGeoMedium *med=new TGeoMedium("Vacuum",1,mat);

  //Define the world
  TGeoVolume *top=gGeoManager->MakeBox("top",med,500,500,500);
  gGeoManager->SetTopVolume(top);

  //We define position
  TGeoRotation     *rot = new TGeoRotation("rot",0.,180,0);
  TGeoCombiTrans   *transf = new TGeoCombiTrans(0,0,-100,rot);

  //We define a geometry
  TGeoVolume *torus2=gGeoManager->MakeTorus("torus2",med,300,0,150,0,360)
  top->AddNode(torus2 ,0,transf);

  //We closed geometry
  gGeoManager->CloseGeometry();

  //Indicate the color cube
  top->SetLineColor(kMagenta);

  //We sent a draw the cube
  gGeoManager->SetTopVisible();
  top->Draw();
}
```

Ejecutando y cargando el archivo “*.c” en ROOT, el resultado obtenido es el siguiente:

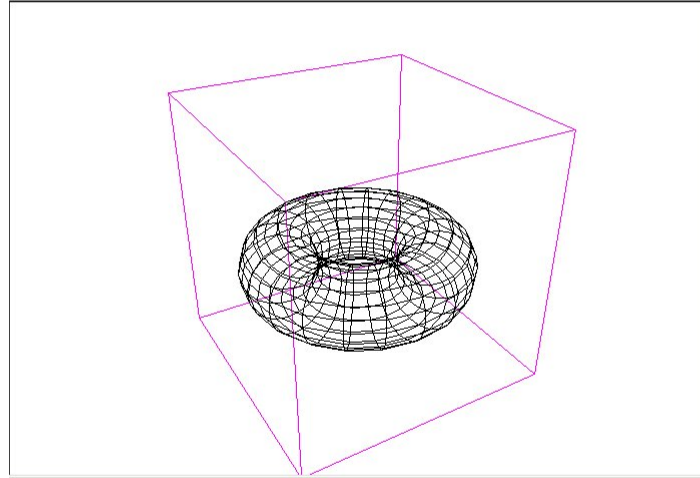

Figura 8.4. Toroide ROOT.

Del mismo modo se hicieron pruebas existosas para las siguientes geometrias: cono, cilindro, cubo, esfera, toroide, disco y rectángulo.

8.2. Liberación del traductor

Richard Stallman creador del proyecto GNU y la Free Software Foundation definió el Software Libre como una herramienta que respeta la libertad de los usuarios que permite a la comunidad a copiar, distribuir, estudiar, modificar y mejorar el código fuente [42].

Un programa es software libre si tiene las siguientes libertades:

- Libertad 0. Se puede ejecutar el programa para cualquier propósito.
- Libertad 1. Se puede estudiar cómo funciona el programa, y cambiarlo para que haga lo que el usuario quiera. El acceso al código fuente es una condición necesaria para ello.
- Libertad 2. Se puede redistribuir copias para ayudar a su prójimo.
- Libertad 3 .Se puede distribuir copias de sus versiones modificadas a terceros. Esto le permite ofrecer a toda la comunidad la oportunidad de beneficiarse de las modificaciones. El acceso al código fuente es una condición necesaria para ello.

Las primeras dos libertades son para el usuario mientras que las dos restantes son para la comunidad.

El software libre no tiene ninguna relación con el aspecto económico, en otras palabras, este no es sinónimo de “software gratis” como usualmente se cree. El software libre se puede vender si es lo que se desea, en casos más especializados se cobra el servicio de soporte y las actualizaciones, sin embargo según las libertades del usuario, que adquiere el software está en su derecho de regalarlo, venderlo, alquilarlo, a diferencia del software propietario en donde distribuir copias ya sea de forma gratuita o con un precio es considerado un delito.

El software privativo es otro nombre para designar el software que no es libre, en éste se prohíbe su uso, redistribución, modificación, o requiere que se solicite permiso, o tiene tantas restricciones que de hecho no se puede hacer libremente. Este tipo de software tiene el control sobre el usuario y no el usuario sobre el software debido a que es el software quien determina que es lo que puede y no puede hacer el usuario. Los usuarios no tienen el derecho de autor conocido como *copyright,* que es un concepto legal que da al autor los derechos exclusivos de la obra, por un tiempo limitado. Esencialmente es la manera en que un autor puede proteger su trabajo y beneficiarse económicamente de él, no obstante, esto tiende a ser un problema en el software, ya que si uno necesita modificarlo se estará incurriendo en un delito.

La forma más sencilla de hacer al software libre es poniéndolo bajo dominio público, es decir sin ningún tipo de derechos de autor, esto presupone un gran problema, ciertamente la comunidad podrá tomar el código y mejorarlo si así se necesita, no obstante existe el riesgo que alguien tome el trabajo, haga algunas modificaciones a él y lo convierta en software privativo y lamentablemente la comunidad que reciba este producto habrá perdido las libertades relacionadas con el software libre.

El proyecto GNU, usa el concepto de *copyleft* para proteger legalmente las libertades del software libre. Su idea fundamental está basada en que se permite su ejecución, su copia, modificación y distribución del código original o su versión modificada sin que se añada ninguna restricción.

Copyleft sólo es un concepto por tanto no puede ser aplicado de manera directa, se debe implementar de otra forma, para esto GNU ofrece a los usuarios varias opciones de licencia, la más común es GPL de GNU la cual se eligió para liberar la herramienta [24].

La licencia GPL del inglés “General Public License” de GNU está basada en la idea *copyleft*, la licencia está aquí para proteger y para dar libertad a los usuarios, ésta se ocupa en prácticamente todos los desarrollos de la Free Software Foundation.

Al poner el desarrollo bajo la licencia GPL se adquieren ciertas responsabilidades, una, es que si se distribuye el software ya sea gratuitamente o a cambio de una compensación monetaria, debe darle a los usuarios las cuatros libertades que hacen un software libre para asegurar que ellos puedan recibir o conseguir el código fuente de algún modo.

Con esta licencia se protege al o a los desarrolladores de dos maneras:

1. Se impone derecho al software
2. Se da derecho legal a distribuir, copiar y a modificar el desarrollo.

La primera protege a cada autor pues no se proporciona ninguna garantía para este software, es decir si el software original es modificado y distribuido, los usuarios finales sabrán que no es el original y cualquier problema no recae a los autores originales.

Al usar esta licencia, el autor renuncia a su derecho de patentar el desarrollo asi como a terceros que quieran tomar el código y convertirlo en software propietario.

Actualmente se trabaja con la versión 3 de esta licencia [22].

Para aplicar esta licencia al desarrollo se debe añadir al principio de cada archivo fuente, las líneas contenidas en la Figura 8.5, también se debe agregar información sobre como contactar al autor mediante correo electrónico y postal.

Es importante también agregar una copia de los términos y condiciones en alguna parte, regularmente se hace en un archivo denominado COPYING.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
```

Figura 8.5. Licencia GPL de GNU.

Aplicando todo esto al desarrollo, añadimos una copia de los términos y condiciones en el archivo COPIYNG y a los archivos fuentes lo siguiente:

```
Copyright (C) 2012 by MARIO ARTURO NIETO BUTRON
mizar0.30@gmail.com
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
```

Capítulo 9

Conclusiones

El trabajo fue realizado en un poco más de 6 meses con el objetivo de aprender las herramientas Salome y ROOT, para proporcionar un traductor preliminar que podría traducir algunas geometrías sencillas. El objetivo fue alcanzado con la realización del traductor salomeToROOT usando Flex y Bison. Se llevaron a cabo pruebas para cada uno de los módulos y pruebas en la integración del traductor. Trabajos adicionales faltan para agregar más funcionalidades al traductor. Sin embargo, un programador con experiencia en C no tendrá problema en retomar y emplear el trabajo hecho. La forma adecuada es definir cada nueva funcionalidad como un módulo del traductor permitiendo a terceras partes participar en el proyecto. Por eso, el traductor será en el futuro liberado bajo la licencia GPL de GNU.

Para el autor, el software libre es la mejor manera posible de hacer crecer una herramienta científica.

Ésta es una versión preliminar y no se encuentra lista para ser liberada en la comunidad del software libre debido a su poca funcionalidad.

Más que un compilador, ésta primera versión de la herramienta "salomeToROOT" es un traductor por las siguientes razones:

1. Un compilador esta formado por un gran número de módulos. En este desarrollo son omitidos varios de ellos debido a la complejidad y a la falta de recursos humanos.

2. Un compilador es una herramienta que toma por completo un lenguaje y traduce todas las instrucciones (a excepción de los espacios en blanco y los comentarios) a su equivalente en el lenguaje destino. En este trabajo, sólo importa obtener las instrucciones que especifican una geometría, las demás son eliminadas.

3. Por último un compilador es una herramienta que no pide datos extras al usuario para llevar a cabo el proceso de compilación, en este caso se pide al usuario las dimensiones del mundo.

Pero con la ampliación de las funcionalidades del traductor, se creará un compilador que contemplará:

1. Obtención del mundo
2. Manejo de gramáticas ambiguas
3. Interfaz gráfica
4. ROOT a Salome

Obtención del mundo
El problema más importante a solucionar después de este trabajo es la creación de un algoritmo que genere un mundo y un marco de referencia equivalente (sistema de coordenadas global) entre la plataforma Salome y ROOT.

Manejo de gramáticas ambiguas
La plataforma Salome da otras funciones para crear las primitivas mostradas en este trabajo, por tal motivo se requiere trabajar con gramáticas ambiguas ya que, por ejemplo, para la creación de un cubo se tienen tres formas distintas de hacerlo en Salome.

Interfaz gráfica
Hoy en día prácticamente todas las herramientas cuentan con una interfaz gráfica, para ser más amigable la experiencia del usuario, por lo tanto se plantea la creación de una interfaz gráfica sencilla. Se está contemplando también la integración en la plataforma Salome del traductor.

ROOT a Salome
No solo se debe pensar en un camino de la plataforma Salome a ROOT, también es muy importante crear un sentido de ROOT a Salome.

Con el desarrollo de una herramienta abierta, el autor ha adquirido nuevos conocimientos y tuvo la suerte de practicar conocimientos impartidos en sus estudios universitarios, no obstante él quiere resaltar que el desarrollo de esta herramienta es un pequeño aporte a la gran cantidad de temas computacionales y a la comunidad científica. Se espera que el trabajo hecho se utilice, con el objetivo de proporcionar a la comunidad del software libre el compilador "salomeToROOT".

Apéndices

Apéndices A

Código fuente salomeToROOT

A continuación se presenta el código que compone al traductor salomeToROOT:

```
/*********************************/
/*       Archivo: filter.c       */
/*       código Fuente           */
/*********************************/

1 #include "filter.h"
2 //Filter.c is the first part on translator salomeToRoot
3 /*
4 This function receives a file
5 and determines if the file
6 contains the extesion .py
7 */
8 int isPY(char *path)
9 {
10     char c='.';
11     char *divide=strchr(path,c);
12     //If the file not contains extension, the compilation ends and error sending
13     if(divide==NULL)
14     {
15       printf("%s: file not recognized: File truncated \n",path);
16       printf("compilation terminated. \n");
17       exit(-1);
18     }
19     //If the extension is .py then return 1
20     else if(!strcmp(divide,".py"))
21     {
22       return 1;
23     }
24     //If the extension is not .py, the compilation ends and error sending
25     else
26     {
27       printf("%s : file not recognized: File truncated \n",path);
28       printf("compilation terminated. \n");
29       exit(-1);
30     }
31   }
32   /*
33   This function checks if the
```

```
34  input file exists, if file
35  exits called the funtion isPy.
36  */
37  int verEntry(char *path,int arg)
38  {
39    FILE *file;
40    //The input not contains any file
41    if(arg == 1)
42    {
43      printf ("Error: fatal error: no input files \n");
44      printf("compilation terminated. \n");
45      exit(-1);
46    }
47    //The input exists
48    else if (arg == 2)
49    {
50      file=fopen(path,"r");
51      //The input not exists
52      if(file==NULL)
53      {
54        printf("error: %s: No such file or directory\n",path);
55        printf("fatal error: no input files\n");
56        printf("compilation terminated.\n");
57        exit(-1);
58      }
59      //The input exists and contains the extension .py
60      else if(isPY(path))
61        return 1;
62    }
63    else
64      return 0;
65  }
66
67  /*
68  This function performs filtering
69  and returns a temporary file
70  with keywords to the
71  translation
72  */
73  int filterInput(char *path,int arg)
74  {
75    FILE *file;
76    FILE *tmp;
77    regex_t re;
78    char sAux[1000]="";
79    char *prueba;
80    //Define regular expression
81    regcomp(&re,".*Make.*",REG_EXTENDED|REG_NOSUB);
82    //READING FILE
83    if(verEntry(path,arg))
84    {
85      //Open file for read
86      file=fopen(path,"r");
87      //The temporary file is created
88      tmp=fopen("tmp","w");
89      //Read the file line by line
90      while(!feof(file))
91      {
92        fgets(sAux,1000,file);
93        //search pattern matches and ignores
94        if(!regexec(&re,sAux,(size_t)0,NULL,0))
95        {
96          fputs(sAux,tmp);
97        }
98        else
99          continue;
100     }
101     //close the file
```

```
102       fclose(file);
103       //close the temporary file
104       fclose(tmp);
105       return 1;
106    }
107 }
```

```
/*********************************/
/*        Archivo: filter.h       */
/*Archivo de cabecera,incluye:    */
/*    -Librerias necesarias       */
/*    -Prototipos de las Funciones*/
/*********************************/

108 #ifndef Filtro_filter_h
109 #define Filtro_filter_h
110 //Define the necessary libraries for the filter
111 #include <stdio.h>
112 #include <stdlib.h>
113 #include <regex.h>
114 #include <string.h>
115 //Prototypes of the functions necessary
116 //Function to verify that the file exists
117 int verEntry(char *,int);
118 //Function to verify that the file containing the extension ".py"
119 int isPY(char *);
120 //Filter function returns certain words
121 int filterInput(char *,int);
122 #endif
```

```
/*********************************/
/*        Archivo: scanner.l      */
/*         Analizador LEXICO      */
/*********************************/

123 %{
124 /*Defines a lexer for SALOME*/
125 #include "parser.tab.h"
126 int lineno=1;
127 %}
128
129 /*Regural Expressions: GENERAL*/
130 digit [0-9]
131 letter  [A-Za-z]
132
133 /*Regural Expressions: NUMBERS*/
134 number {digit}+("."{digit}+)?
135
136 /*Regural Expressions: IDENTIFIERS*/
137 id      ({letter}|_)({letter}|{digit}|_)*
138
139 /*Regural Expressions: BLANK*/
140 blank [ \t\n]
141
142
143 %%
144 #.*                     ;
145 {blank}                 ;
146 "geompy"                        {
147         return GEOMPY;
148             };
149 "MakeBoxDXDYDZ"                 {
150         return MAKEBOXDXDYDZ;
151             };
```

```
152 "MakeCylinderRH"                {
153           return MAKECYLINDERRH;
154               };
155 "MakeSphereR"                   {
156           return MAKESPHERER;
157               };
158 "MakeConeR1R2H"                 {
159           return MAKECONER1R2H;
160               };
161 "MakeTorusRR"                   {
162           return MAKETORUSRR;
163               };
164 "MakeFaceHW"                    {
165           return MAKEFACEHW;
166               };
167 "MakeDiskR"                     {
168           return MAKEDISKR;
169               };
170
171
172 {id}                {
173         yylval.string=yytext;
174         return IDENTIFIER;
175         }
176
177 {number}            {
178             yylval.typeReturn=atof(yytext);
179             return NUM;
180         };
181
182 "("             {
183             return '(';
184         };
185
186 ")"             {
187           return ')';
188          };
189
190 ","                {
191           return ',';
192          };
193
194 "."                 {
195           return '.';
196          };
197
198 "="                 {
199           return '=';
200          };
201
202
203 .        {
204         yylval.string=yytext;
205         return ERROR;
206          };
```

```
/*********************************/
/*        Archivo: parser.y      */
/*         Analizador SINTACTICO */
/*********************************/

207 %{
208   #include "scanner.c"
209   #include "functions.h"
210   //Define a file for output
211   FILE *archSal;
```

```
212   //name for the output file that will contain the geometry in ROOT
213   char *name;
214   //Function for the treatment of errors
215   void yyerror(char const *);
216 %}
217 %union
218 {
219   double typeReturn;
220   char *string;
221 }
222
223 /*general tokens*/
224 %token <string>IDENTIFIER
225 /*general tokens geometry*/
226 %token GEOMPY
227 /*Numbers*/
228 %token <typeReturn> NUM
229 /*Tokens for a cube*/
230 %token MAKEBOXDXDYDZ
231 /*Tokens for a cylinder*/
232 %token MAKECYLINDERRH
233 /*Tokens for a sphere*/
234 %token MAKESPHERER
235 /*Tokens for a cone*/
236 %token MAKECONER1R2H
237 /*Tokens for a torus*/
238 %token MAKETORUSRR
239 /*Tokens for a rectangle*/
240 %token MAKEFACEHW
241 /*Tokens for a disc*/
242 %token MAKEDISKR
243 /*Token to recognize a mistake*/
244 %token <string>ERROR
245
246 %%
247 /*Defined start symbol*/
248 start: input;
249
250 /*Define input*/
251 input:geometry aux|ERROR {yyerror("\n"); yyerrok; fprintf(stderr,"error: %s: unknown symbol \n",$
252
253 /*Define aux*/
254 aux:/*cadena vacia*/|geometry aux;
255
256 /*We define geometry types*/
257 geometry : cube|cylinder|sphere|cone|torus|face|disc|error;
258
259 /*Grammar rule for a cube*/
260 cube:IDENTIFIER '=' GEOMPY '.' MAKEBOXDXDYDZ '(' NUM ',' NUM ',' NUM ')'
261 {
262   fprintf(archSal,"\t//We define a geometry \n");
263   fprintf(archSal,"\tTGeoVolume *%s=gGeoManager->MakeBox(\"%s\",med,%lf,%lf,%lf);
      \n",getID($1),getID($1),$7,$9,$11);
264   fprintf(archSal,"\ttop->AddNode(%s,0, transf);\n\n",getID($1));
265 };
266
267
268 /*Grammar rule for a cylinder*/
269 cylinder:IDENTIFIER '=' GEOMPY '.' MAKECYLINDERRH '(' NUM ',' NUM ')'
270 {
271   fprintf(archSal,"\t//We define a geometry \n");
272   fprintf(archSal,"\tTGeoVolume *%s=gGeoManager->MakeTube(\"%s\",med,0,%lf,%lf);
      \n",getID($1),getID($1),$7,$9);
273   fprintf(archSal,"\ttop->AddNode(%s,0, transf);\n\n",getID($1));
274 };
275
276 /*Grammar rule for a sphere*/
277 sphere:IDENTIFIER '=' GEOMPY '.' MAKESPHERER '(' NUM ')'
```

```
278 {
279   fprintf(archSal,"\t//We define a geometry \n");
280   fprintf(archSal,"\tTGeoVolume *%s=gGeoManager->MakeSphere(\"%s\",med,0,%lf,0,180,0,360);
      \n",getID($1),getID($1),$7);
281   fprintf(archSal,"\ttop->AddNode(%s,0,transf);\n\n",getID($1));
282 };
283
284 /*Grammar rule for a cone*/
285 cone:IDENTIFIER '=' GEOMPY '.' MAKECONER1R2H '(' NUM ',' NUM ',' NUM ')'
286 {
287   fprintf(archSal,"\t//We define a geometry \n");
288   fprintf(archSal,"\tTGeoVolume *%s=gGeoManager->MakeCone(\"%s\",med,%lf,0,%lf,0,%lf);
      \n",getID($1),getID($1),$11,$7,$9);
289   fprintf(archSal,"\ttop->AddNode(%s,0,transf);\n\n",getID($1));
290 };
291 /*Grammar rule for a torus*/
292 torus:IDENTIFIER '=' GEOMPY '.' MAKETORUSRR '(' NUM ',' NUM ')'
293 {
294   fprintf(archSal,"\t//We define a geometry \n");
295   fprintf(archSal,"\tTGeoVolume *%s=gGeoManager->MakeTorus(\"%s\",med,%lf,0,%lf,0,360);
      \n",getID($1),getID($1),$7,$9);
296   fprintf(archSal,"\ttop->AddNode(%s,0,transf);\n\n",getID($1));
297 };
298 /*Grammar rule for a rectangle*/
299 face:IDENTIFIER '=' GEOMPY '.' MAKEFACEHW '(' NUM ',' NUM ',' NUM ')'
300 {
301   int tmp=$11;
302   switch (tmp)
303   {
304     case 1:
305       fprintf(archSal,"\t//We define a geometry \n");
306       fprintf(archSal,"\tTGeoVolume *%s=gGeoManager->MakeBox(\"%s\",med,%lf,%lf,0);
        \n",getID($1),getID($1),$7,$9);
307       fprintf(archSal,"\ttop->AddNode(%s,0, transf);\n\n",getID($1));
308       break;
309
310     case 2:
311       fprintf(archSal,"\t//We define a geometry \n");
312       fprintf(archSal,"\tTGeoVolume *%s=gGeoManager->MakeBox(\"%s\",med,0,%lf,%lf);
        \n",getID($1),getID($1),$7,$9);
313       fprintf(archSal,"\ttop->AddNode(%s,0, transf);\n\n",getID($1));
314       break;
315
316     case 3:
317       fprintf(archSal,"\t//We define a geometry \n");
318       fprintf(archSal,"\tTGeoVolume *%s=gGeoManager->MakeBox(\"%s\",med,%lf,0,%lf);
        \n",getID($1),getID($1),$7,$9);
319       fprintf(archSal,"\ttop->AddNode(%s,0, transf);\n\n",getID($1));
320       break;
321     default:
322       printf("ERROR: %d: Invalid option \n",tmp);
323       deleteFile(name);
324       break;
325   }
326 };
327 /*Grammar rule for a disc*/
328 disc:IDENTIFIER '=' GEOMPY '.' MAKEDISKR '(' NUM ',' NUM ')'
329 {
330   int tmp=$9;
331   switch(tmp)
332   {
333     case 1:
334       fprintf(archSal,"\t//We define a geometry \n");
335       fprintf(archSal,"\tTGeoVolume *%s=gGeoManager->MakeTorus(\"%s\",med,%lf,0,0,0,360);
        \n",getID($1),getID($1),$7);
336       fprintf(archSal,"\ttop->AddNode(%s,0,transf);\n\n",getID($1));
337       break;
338     case 2:
```

```
339         fprintf(archSal,"\t//We define a geometry \n");
340         fprintf(archSal,"\tTGeoVolume *%s=gGeoManager->MakeTorus(\"%s\",med,%lf,0,0,0,360);
            \n",getID($1),getID($1),$7);
341         fprintf(archSal,"\ttop->AddNode(%s,0,transf);\n\n",getID($1));
342         break;
343       case 3:
344         fprintf(archSal,"\t//We define a geometry \n");
345         fprintf(archSal,"\tTGeoVolume *%s=gGeoManager->MakeTorus(\"%s\",med,%lf,0,0,0,360);
            \n",getID($1),getID($1),$7);
346         fprintf(archSal,"\ttop->AddNode(%s,0,transf);\n\n",getID($1));
347         break;
348       default:
349                   printf("ERROR: %d: Invalid option \n",tmp);
350         deleteFile(name);
351                   break;
352
353     }
354   };
355   %%
356   /*
357   Main function of the translator
358   */
359   int main(int argc, char *argv[])
360   {
361     double world[3];
362     if(filterInput(argv[1],argc))
363     {
364       yyin=fopen("tmp","r");
365       /*
366         It creates the file that will contain
367         the geometry and is called as the input
368         file but the extension is replaced .py by the of .c
369       */
370       archSal=fopen(strcat(getName(argv[1]),".c"),"w");
371       if(yyin==NULL)
372       {
373         printf("\nERROR\n");
374       }
375
376       /*It stores some of the geometry and syntactic
377       analyzer is called which in turn calls the lexical analyzer*/
378       else
379       {
380         //We define the output file
381
382         name=strcat(getName(argv[1]),".c");
383         //We define the size of world
384         captureWorld(&world[1],&world[2],&world[3]);
385
386         //Program structure in ROOT
387         fprintf(archSal,"#include \"TGeoManager.h\" \n");
388         fprintf(archSal,"void %s()\n",getName(argv[1]));
389         fprintf(archSal,"{\n");
390
391         fprintf(archSal,"\t //We call the admin. geometry\n");
392         fprintf(archSal,"\t new TGeoManager(\"world\",\"simple\");\n\n");
393
394         fprintf(archSal,"\t //We define the means and the material\n");
395         fprintf(archSal,"\t TGeoMaterial *mat=new TGeoMaterial(\"Vacuum\",0,0,0);\n");
396         fprintf(archSal,"\t TGeoMedium   *med=new TGeoMedium(\"Vacuum\",1,mat);\n\n");
397
398         fprintf(archSal,"\t //Define the world \n");
399         fprintf(archSal,"\t TGeoVolume *top=gGeoManager->MakeBox(\"top\",med,%f,%f,%f);
            \n",world[1],world[2],world[3]);
400         fprintf(archSal,"\t gGeoManager->SetTopVolume(top);\n\n");
401         ////We define position
402         fprintf(archSal,"\t //We define position \n");
403         fprintf(archSal,"\t TGeoRotation   *rot = new TGeoRotation(\"rot\",0.,180,0); \n");
```

```
404        fprintf(archSal,"\t TGeoCombiTrans  *transf = new TGeoCombiTrans(0,0,-100,rot); \n\n");
405
406        //We call the parser, which in turn call the lexical analyzer
407        yyparse();
408
409        //Final part of the root file
410        fprintf(archSal,"\t //We closed geometry\n");
411        fprintf(archSal,"\t gGeoManager->CloseGeometry();\n\n");
412
413        fprintf(archSal,"\t //Indicate the color cube\n");
414        fprintf(archSal,"\t top->SetLineColor(kMagenta);\n\n");
415
416        fprintf(archSal,"\t //We sent a draw the cube\n");
417        fprintf(archSal,"\t gGeoManager->SetTopVisible();\n");
418        fprintf(archSal,"\t top->Draw();\n");
419        fprintf(archSal,"}\n");
420      }
421    }
422    fclose(archSal);
423    //delete the file temporary
424    deleteFile("tmp");
425    return 0;
426 }
427 //Function for error handling, if you find one just send as message syntax error
428 void yyerror(char const *s)
429 {
430    fprintf(stderr," %s \n",s);
431    /*
432    Remove the output file because if an
433    error is found, the translation shall
434    imcompleta and would send wrong file
435    */
436    remove(strcat(name,".c"));
437 }
```

```
/*********************************/
/*       Archivo: functions.c    */
/*        Funciones extras       */
/*********************************/

438 #include "functions.h"
439
440 //get filename
441 char *getName(char name[])
442 {
443     char temp[1] = ".";
444     char *ptr;
445     ptr = strtok( name,temp);
446     return ptr;
447 }
448 //get ID for parser
449 char *getID(char name[])
450 {
451     char temp[1] = "=";
452     char *ptr;
453     ptr = strtok( name,temp);
454     return ptr;
455 }
456 //Funcion for delete a file
457 void deleteFile(char *name)
458 {
459    if(remove(name)==-1)
460      perror("\n Error deleting \n");
461    else
462      puts("\ntemporary file removed\n");
463 }
```

```
464
465 //Get the size of the world
466 void captureWorld(double *num1,double *num2,double *num3)
467 {
468     printf("Dimensions of the world in ROOT \n");
469     printf("size in x: \n");
470     scanf("%lf",num1);
471     printf("size in y: \n");
472     scanf("%lf",num2);
473     printf("size in z: \n");
474     scanf("%lf",num3);
475
476 }
```

```
/**********************************/
/*       Archivo: functions.h     */
/*        Funciones extras        */
/**********************************/

477 #ifndef FUNCTIONS_H
478 #define FUNCTIONS_H
479
480 //Libraries
481 #include <stdio.h>
482 #include <stdlib.h>
483 #include <string.h>
484 //Get filename
485 char *getName(char []);
486 //Get ID for parser
487 char *getID(char []);
488 //Delete a file
489 void deleteFile(char *);
490 //Get the size of the world
491 void captureWorld(double *,double *,double *);
492
493 #endif
```

```
494 # Top level Makefile for salomeToRoot System
495 # Copyright (c) 2012
496 #
497 # Author: Mario Arturo Nieto Butron, 19/11/2012
498
499 LEX=flex -o
500 BISON=bison -d
501 CC=gcc -o
502 INSTALL=/usr/bin/salomeToRoot
503 DELETE=unlink
504
505 salomeToRoot: scanner.c parser.tab.c functions.c filter.c
506   $(CC) salomeToRoot parser.tab.c functions.c filter.c -lfl
507   @echo " "
508   @echo "   ================================================================"
509   @echo "   ===          salomeToroot BUILD SUCCESSFUL.
===" 
510   @echo "   === Run 'make install' now with user root.
===" 
511   @echo "   ================================================================"
512
513 scanner.c:scanner.l
514   $(LEX) scanner.c scanner.l
515
516 parser.tab.c parser.tab.h: parser.y
517   $(BISON) parser.y
518
```

```
519
520 install: salomeToRoot
521   install $< $(INSTALL)
522
523 .PHONY: clean
524 clean:
525   rm -f scanner.c parser.tab.h parser.tab.c salomeToRoot
526
527 .PHONY: uninstall
528 uninstall:
529   $(DELETE) $(INSTALL)
```

Apéndices B

Makefile

Normalmente los proyectos de software se componen por mas de un archivo fuente de tal forma que la construcción y el mantenimiento se vuelven tareas mas sencillas no obstante la complejidad de compilar el código crece debido a que se necesitaran líneas del compilador mas complicadas. Para simplificar la compilación, se tiene una herramienta llamada "make" desarrollo por el proyecto GNU, que permite simplificar el proceso de compilacion mediante la invocacion de un solo archivo de texto.

Cuando se ejecuta el comando "make", este buscara por defecto un archivo con el nombre de "GNUmakefile", "makefile" o "Makefile".

Un archivo "makefile" esta compuesto por un conjunto de reglas cuya estructura se puede ver en la Figura B.1.

La estructura de una regla consta de tres partes que son:

- Objetivo
- Dependencias
- Comandos

En donde el objetivo es un archivo binario o un archivo objeto el cual se quiere construir, las dependencias son los requisitos previos que se necesitan para construir el objetivo y los comandos son los pasos necesarios para construir el objeto.

Una regla puede tener mas de un objetivo

```
objetivo: dependencia1 dependencia2 dependencia3...
          comando1
          comando2
          ...
```

Figura B.1. Estructura archivo Makefile

Una regla puede estar definida por un gran número de dependencias con las mismas extensiones, para simplificar el proceso de escritura los archivos "makefile" soportan comodines o metacaracteres idénticos a los definidos por el *Bourne Shell* [49].

Se pueden añadir comentarios a un archivo "make" para lo cual utilice el símbolo especial "#", estos pueden ir colados en donde se necesiten.

Además de los objetivos normales o absolutos también se cuentan con objetivos ficticios los cuales representan tan solo una secuencia de comandos que deben ser ejecutados y no representan a archivos reales. Por ejemplo en el archivo "makefile" se puede insertar un objetivo de limpieza de la siguiente manera:

```
clean:
rm -f *.olexer.c
```

La mayoría de los objetivos falsos no tienen dependencias por lo tanto siempre "make" los va a considerar actualizados por lo tanto no se van a ejecutar, para evitar este problema la herramienta incluye un objetivo especial "*.PHONY" este le indica que no se trata de un archivo real. Cualquier objetivo puede ser declarado falso por su inclusión como requisito previo farsante, de donde el objetivo "clean" se define de la siguiente manera:

```
.PHONY: clean
clean:
        : rm -f *.o lexer.c
```

Variables

Con el fin de simplificar la construcción y el mantenimiento del "makefile", "make", permite la creación y utilización de variables, en este archivo una variable no es mas que un identificador que representa una cadena de texto.

"make" reconoce cuatro tipos de variables:

1. Variables de usuario
2. Variables de entorno
3. Variables automáticas
4. Variables predefinidas

Variables de usuario

Las variables de usuario son las variables que define el usuario valga la redundancia, generalmente este tipo de variables se definen al principio del archivo "make".

Su sintaxis es la siguiente:

```
NOMBRE_DE_LA_VARIABLE=VALOR [ . . . ]
```

Por convención este tipo de variables se definen en mayúsculas y se usan de esta manera “$(NOMBRE_DE_LA_VARIABLE)”. El fin de usar este tipo de variables es que si por algún motivo es necesario modificar un parámetro o un comando solo se modifique una vez.

Variables de entorno

Son todas la variables definidas en el entorno de la interfaz que son copiadas por “make” con los mismos nombres y los mismos valores. No obstante en caso de existir variables de usuario con el mismo nombre, las variables de entorno serán invalidas.

Variables automáticas

Estas variables son las que se evalúan cada vez que es ejecutada una regla, basándose en el objetivo y en las dependencias de esa regla, son las que se emplean para crear reglas patrón, las cuales serán genéricas. Por ejemplo una regla que indique como compilar un archivo “*.c” arbitrario a su correspondiente archivo “*.o”.

Variables predefinidas

“make” predefine un cierto número de otras variables que son utilizadas ya sea como nombres de programas o para transferir indicadores y argumentos a esos programas.

Reglas Implícitas

Hasta este momento se han utilizado reglas definidas por el programador a las cuales se les llaman reglas explicitas sin embargo se tienen también un conjunto de reglas definidas por “make” las cuales son conocidas como reglas explicitas o predefinidas. Por ejemplo si se tienen dos objetivos a los cuales no se les agregaron los comandos necesarios para su construcción, en lugar de terminar con la ejecución del “makefile”, “make” intentara buscar reglas implícitas que les permitan la construcción de los objetivos.

Objetivos especiales

Estos son los objetivos ficticios como el caso del objetivo “clean” cuyo fin consistía en realizar una limpieza de los códigos objetos construidos por “make”,

para construir el objetivo principal. Además de "clean" en los archivos "make" se definen otros tipos de objetivos. Se listan en la siguiente tabla los mas comunes:

Nombre	Descripción
all	Realiza todas las tareas necesarias para crear la aplicación.
install	Realiza la instalación de los binarios compilados.
clean	Realiza una limpieza de archivos generados desde las fuentes.
distclean	Elimina todos los archivos fuentes que no estaban en la fuente original.
TAGS	Crea una tabla de etiquetas para los editores en uso.
info	Crea archivos de informacion GNU desde sus fuentes.
check	Ejecuta algunas pruebas asocidas con esta aplicacion.

Cuadro B.1. Objetivos especiales comunes

Apéndices C

Manual de Usuario

Requerimientos del sistema

- Sistema Operativo: GNU/LINUX
- Compilador: gcc
- Compilador: Bison
- Compilador: Flex
- Libreria: libfl.a

Instalación

Se cuenta con un archivo "makefile" con los pasos siguientes:

- Descomprimir el archivo salomeToROOT.tar.gz en /opt

```
> cd /opt
> tar -xzvf salomeToROOT.tar.gz
```

- Ingresamos a la carpeta salomeToROOT:

```
> cd salomeToROOT
```

- Ejecutar el comando make:

```
> make
```

- Ejecutar con permiso de súper usuario el comando “make install”:

```
# make install
```

Adicionalmente se incluyen las opciones de “uninstall” y “clean”, para desinstalar y limpiar respectivamente.

Uso

salomeToROOT es una herramienta de líneas de comandos, para usarla desde líneas de comandos llamamos al traductor con la siguientes sintaxis:

```
> salomeToROOT nombre_archivo.py
```

Apéndices D

Reglas para compilar en Flex y Bison

Se presentan las principales opciones para compilar en Flex y Bison.

Sintaxis para compilar en Flex:

```
> flex -o nombre.c archivo.l
```

La opción -o sirve para especificar el nombre del archivo de salida, si se omite esta opción por defecto Flex nos regresara una archivo de nombre “lex.yy.c”, en donde estará contenido el analizador léxico.

El archivo de entrada a Flex debe tener la extensión “*.l” .

Sintaxis para compilar en Bison :

```
> bison -d archivo.y
```

Por defecto Bison creara dos archivos uno con la especificación del analizador sintáctico por lo tanto contara con la extensión “*.c” y el otro con la definición de los macros para los símbolos terminales y contara con la extensión “*.h” debido a que es un archivo de cabecera.

Finalmente para unir ambos desarrollos se usa el compilador gcc de GNU.

La sintaxis para unir ambos archivos con el compilador gcc es la siguiente:

```
> gcc -o nombre.c archivo.tab.c otro1.c ,otro2.c... otron.c -lfl
```

Se indican otros en el caso de que el desarrollo necesite de algún otro archivo “*.c” necesario para la compilación y la ejecución, con la opción -o definimos el nombre que llevara el archivo final.

Referencias

[1] Commissériat à l'Energie Atomique. Identity. Consultado el 15 de octubre de 2013, de http://www.cea.fr/english-portal/cea/identity.

[2] Anthony A. Aaby. Compiler Construction using Flex and Bison, 2004.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compiladores.Principios,técnicas y herramientas. 2000.

[4] Manuel Alfonseca Cubero Alfonseca Moreno. *Compiladores e Interpretes: Teoría y Practica*. PEARSON EDUCACION, 2006.

[5] López Tecillo Miguel Angel. *Compilador en español basado en lenguaje JAVA en español*. PhD thesis, 2008.

[6] Andrade Delgado Arnulfo, Castañeda de Isla Puga Erik, Oregel Sánchez Felipe, and Parada Avila Jaime. *Antecedentes de álgebra elemental*. 1990.

[7] Gough Brian and Richard M. Stallman. *An Introduction to GCC*, 2004.

[8] J. Glenn Brookshear. *Teoría de la Computación.Lenguajes formales,autómatas y complejidad*. ADDISON-WESLEY, 1993.

[9] Open Cascade. Open cascade technology, 3d modeling and numerical simulation. Consultado el 10 de septiembre de 2012, de http://www.opencascade.org/.

[10] Open Cascade. Salome 6.6.0 documentation. Consultado el 10 de marzo del 2013, de http://www.salome-platform.org/user-section/documentation/current-release.

[11] Open Cascade. Salome architecture. Consultado el 23 de agosto de 2012, de http://docs.salome-platform.org/salome_6_5_0/gui/gui/salome_architecture_page.html.

[12] Jacinto Ruiz Catalán. *Compiladores Teoría e implementación*. Alfaomega, 2010.

[13] CERN. About cern. Consultado el 20 noviembre de 2012, de http://public.web.cern.ch/public/.

[14] CERN. Boundary representation. Consultado el 15 de junio de 2012, de http://cadd.web.cern.ch/cadd/cad_geant_int/thesis/ node23.html.

[15] CERN. Large acceptance hadron detector for an investigation of pb-induced reactions at the cern sps. Consultado el 6 de noviembre de 2012, de http://na49info.web.cern.ch/na49info/.

[16] CERN. User support. Consultado el 12 de agosto de 2012, de http://geant4.cern.ch/.

[17] Electricité de France. Electricité de france. Consultado el 5 de octubre de 2012, de http://france.edf.com/france-45634.html.

[18] Marina de la Cruz. Yacc/bison. Consultado el 7 de diciembre de 2012, de http://arantxa.ii.uam.es/˜ mdlcruz/docencia/ compiladores/2002_2003/compiladores_02_03_yacc_bison.pdf.

[19] Laboratorio Detección de Partículas y Radiación. Dark matter identification with ccd. Consultado el 10 de diciembre del 2012, http://fisica.cab.cnea.gov.ar/particulas/html/labdpr/damic-south/.

[20] Charles Donnelly and Richard Stallman. *Bison*, 2012.

[21] Hopcroft John E. and Jeffrey D. Ullman. *Introducción a la teoría de autómatas, lenguajes y computación*. CECSA, primera edición edition, 1993.

[22] Free Software Foundation. Gnu general public license. Consultado el 18 de diciembre de 2012, de http://www.gnu.org/licenses/gpl-3.0.html.

[23] Free Software Foundation. Gnu lesser general public license. Consultado el dia 15 de noviembre de 2012, de http://www.gnu.org/copyleft/lesser.html.

[24] Free Software Foundation. Licencias. Consultado el 15 de diciembre de 2012, de http://www.gnu.org/licenses/licenses.es.html.

[25] Free Software Foundation. Sistema operativo gnu. Consultado el 3 agosto de 2012, de http://www.gnu.org/philosophy/free-sw.es.html.

[26] Noah Gift and Jeremy M.Jones. *Python para administración de sistemas Unix y Linux*. Primera edición edition, 2008.

[27] Jan Goyvaerts. Posix basic regular expressions. Consultado el 4 de noviembre de 2012, de http://www.regular-expressions.info/posix.html.

[28] H.M.Deitel and P.J.Deitel. *Como Programar en C/C++*. Segunda edicion edition, February 1998.

[29] Sommerville Ian. *Ingeníeria del software*. PEARSON, séptima edición edition, 2005.

[30] ISO. Iso 10303-203:2011. consultado el 6 de junio de 2012, de http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=44305.

[31] Goyvaerts Jan and Levithan Steven. *Regular Expressions CookBook*. O'Reilly, primera edición edition, 2009.

[32] Laura Alicia Leonideas Jiménez. *Técnicas para construir compiladores eficientes en haskell*. PhD thesis, 2005.

[33] R. Levine John, Manson Tony, and Brown Doug. *Lex & Yacc*. Second edition edition, February 1995.

[34] Richard Johnsonbaugh. *Matemáticas Discretas*. PEARSON, sexta edición edition, 2005.

[35] Karen A. Lemone. *Fundamentos de Compiladores: Cómo traducir al lenguaje de computadora*. CECSA, 1996.

[36] Seymour Lipschutz. *Teoría de conjuntos y temas afines*. Schaum.

[37] Kenneth C. Louden. *Construcción de compiladores*.

[38] Mark Lutz. *Learning Python*. Tercera edición edition, October 2007.

[39] Francisco Medina. Sistemas cad / cam. Consultado el 13 de junio de 2012, de http://www.computec.org/GibbsCAM/IGES%20for%20Surfaces%20and%20Solids.htm.

[40] Mecklenburg Robert. *GNU Make*. O'Reilly, tercera edición edition, 2004.

[41] Team ROOT. *ROOT: Users Guide 5.26*, 2009.

[42] Richard M. Stallman. *Software libre para una sociedad libre*. traficantes de sueños, 2004.

[43] Richard M. Stallman, McGrath Roland, and Smith Paul D. *GNU Make*. GNU, 2010.

[44] The ROOT Team. About. Consultado el 20 de octubre del 2012, de http://root.cern.ch/drupal/content/about.

[45] The ROOT Team. Architectural overview. Consultado el 20 de agosto de 2012, de http://root.cern.ch/drupal/content/architectural-overview.

[46] The ROOT Team. Cint. Consultado el 30 de agosto de 2012, de http://root.cern.ch/drupal/content/cint.

[47] The ROOT Team. class tgeomanager: public tnamed. Consultado el 23 de septiembre de 2012, de http://root.cern.ch/root/html/tgeomanager.html.

[48] M. Ritchie Dennis W. Kernighan Brian. *El lenguaje de programacion C*. PEARSON EDUCACION.

[49] Kurt Wall. *Programacion en Linux con ejemplos*. Segunda edicion edition, 2002.