Lenguaje para construcción de *software*

Juan Francisco Cardona Mc'Cormick

15 de Marzo

1. Introducción

El manejo de un proyecto de software es una tarea complicada, que requiere la combinacinación de cientos o miles de programas fuentes, generar cientos de bibliotecas de funciones, para generar varios ejecutables que conforman un proyecto complejo.

Este tipo de tareas no se puede realizar a mano y para ello existen los IDE (*Integrated Development Enviroment*) que permiten manejar proyectos complejos, pero no son en muchos casos lo suficiente, por que ellos están centrados en el desempeño del programador, suministrando editores y visualizadores especializados del código. Es aquí donde aparencen las herramientas llamadas constructores autómaticos (*Build Automation*) que están centrados más en construir todo el código que conforman un proyecto que en facilitar las tareas del programador.

2. Modelo

Partamos del modelo más sencillo que podemos hacer para construir un proyecto de software pequeño. Supongamos que tenemos el famoso programa *Hello World* en lenguaje C

```
#include <stdio.h>

int main() {
  printf("Hello World! n");
  return 0;
}
```

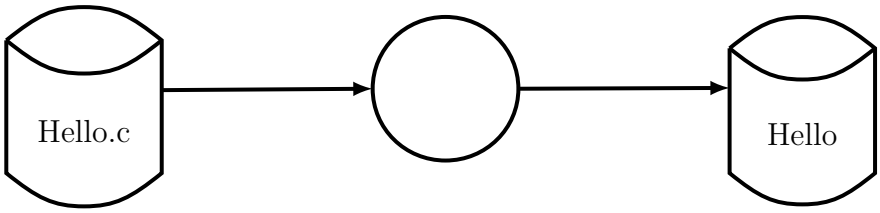

Figura 1: Diagrama de la transformación del ejecutable `Hello.c`

y que generamos su correspondiente ejecutable:

```
$ cc -o Hello Hello.c
```

Toma el código en el archivo `Hello.c` y genera el ejecutable `Hello`[1], utilizando el programa `cc`[2]. En la figura 1 nos muestra dicha transformación y en ella se observa que tenemos un grafo, que toma un archivo y aplicando un programa lo convertimos en otro archivo.

Pero en realidad esta transformación es puede hacer en más etapas como lo hacemos en siguiente compilación manual[3]:

```
$ cpp Hello.c Hello-tmp.c
$ cc -c Hello-tmp.c -o Hello.o
$ ld -o Hello Hello.c /usr/lib/crt0.o /usr/lib/libc.a
```

En la figura 2 se observa el grafo la transformación total. Se puede observar el comportamiento parcial de un proyecto pequeño. Se transforma un archivo en otro archivo utilizando un programa, por ejemplo la obtención de los archivos `Hello-tmp.c` y `Hello.o`. También, para generar un archivo se depende de varios archivos como es el caso para obtener el archivo `Hello`.

En el caso más simple, un proyecto puede ser de una sola etapa, en la cual el objetivo del proyecto es construir un archivo X que depende de los archivos $Y_0, \ldots, Y_{n-1}$. En los proyectos más complejos se pueden hacer describiendo

[1]En Windows los programas ejecutables tienen la extensión `.exe`

[2]El nombre original del compilador de lenguaje C en Unix, hoy en día existen diferentes compiladores: `gcc`, `cl`, etc.

[3]Esta transformación en sistemas Unix, no se puede aplicar directamente en Linux. Para ver las etapas en Linux *The True Story of Hello World*

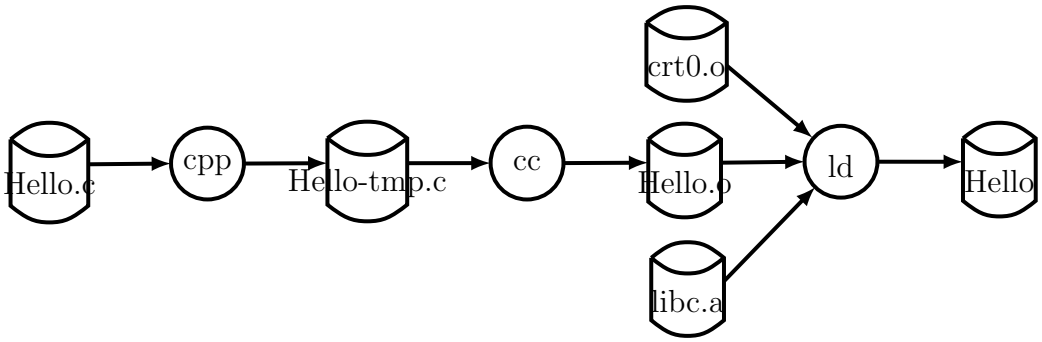

Figura 2: Transformación completa del programa *Hello*

cada una de las etapas particulares. Hasta que finalmente todas las etapas describan un grafo.

Un modelo inicial nos indica que tenemos dos entidades básicas: *archivos* y *programas*. Entonces para obtener el archivo X, este depende de los archivos $Y_0, \ldots, Y_{n-1}$ y es transformado utilizando el programa P. Este primer acercamiento es todavía simple, puesto que en algunos casos la obtención de un ejecutable se puede hacer por otro camino. De nuevo el caso de `Hello`:

```
$ cc -E Hello.c -o Hello-tmp.c
$ cc -c Hello-tmp.c -o Hello.o
$ ld -o Hello Hello.c /usr/lib/crt0.o /usr/lib/libc.a
```

Observe que el programa `cc` es utilizado de dos maneras distintas, es decir que es invocado (o *aplicado*) de dos formas distintas para obtener comportamiento distintos. El programa nos ayuda a describir cuales son los programas, pero en las transformaciones necesitamos definir algo que muestra los diferentes usos de los programas, estos serán las *aplicaciones*. Una aplicación de un programa es el que realmente se encarga de transformar uno (o varios) archivo(s) en un archivo.

En algunos proyectos ciertas tareas no están asociadas a los archivos en general, sino a sus tipos. Supongamos que tenemos un proyecto en el cual tenemos que generar un ejecutable llamado `proyecto` y este depende de tres archivos fuentes: `fuente1.c`, `fuente2.c` y `fuente3.c`, para generar el ejecutable, primer transformamos cada archivo fuente a su correspodiente objeto y luego combinamos los tres objetos para generar el ejecutable:

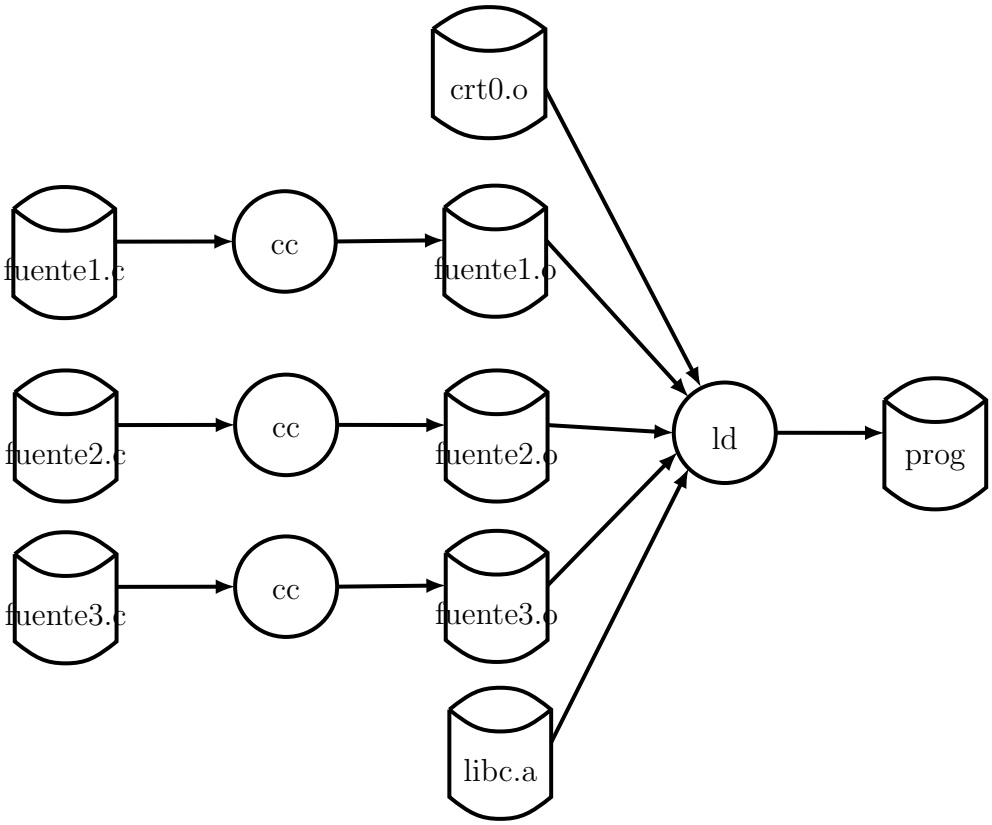

Figura 3: Un proyecto con varias dependencias

```
$ cc -c fuente1.c
$ cc -c fuente2.c
$ cc -c fuente3.c
$ ld -o prog fuente1.o fuente2.o fuente3.o /usr/lib/crt0.o -lc
```

En la figura 3 se observa dicha transformación. Si observamos bien, esta transformación está aplicada al tipo de archivo, los archivos con extensión `.c` en la compilación se transforma primero en archivos objeto. Esto nos indica que podríamos crear una *clase* que agrupe a varios archivos basados en una o varias propiedades específicas de los mismos, como por ejemplo su extensión. Las clases de archivos también se comportarán como archivos y podrán ser transformados 4.

En algunas situaciones, las transformaciones son más complicadas y son

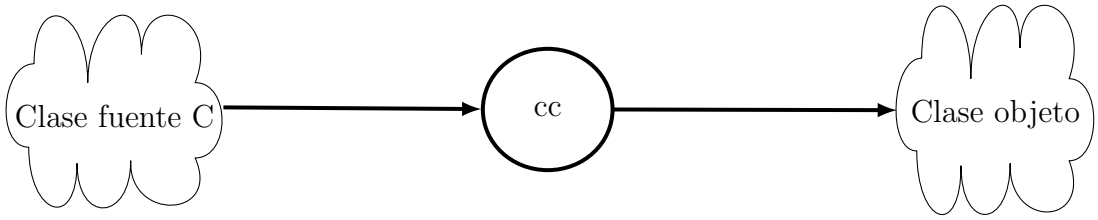

Figura 4: Transformación de clases de archivos

llevadas a cabo por una secuencia de operaciones que transforman uno o más archivos (o clases) de entrada en un archivo de salida. En el siguiente caso el archivo fuente `forma.frm` tiene un formulario de entrada en binario y el archivo `forma.ctr` tiene la información que controla la ejecución de un formulario. El siguiente proceso combian los formularios para generar un archivo de tipo `.xml` que luego es utilizado para transformar su respectivo programa en c.

```
$ genForm forma.frm forma.ctr -o forma.xml
$ xsltproc --encoding toCLanguage -o forma.c forma.xml
$ rm -f forma.xml
```

En este caso, el archivo `forma.xml` puede ser visto como un archivo intermedio que no es necesario en las demás etapas. Esta transformación puede ser vista como una función que toma uno o más archivos (o clases) de entrada y produce un archivo (o clase) de archivo de salida. En la figura 5 se observa tal comportamiento.

3. Lenguaje de construcción de *software*

La idea del proyecto es generar un lenguaje que nos permita construir un proyecto de *software* de manera sistematica que siga el modelo visto en la sección 2. El lenguaje es realmente una secuencia de varias cadenas que pertenecen a diferentes lenguajes que se describiran a continuación. En algunos casos estos lenguajes hace uso de identificadores de cada uno de los componentes.

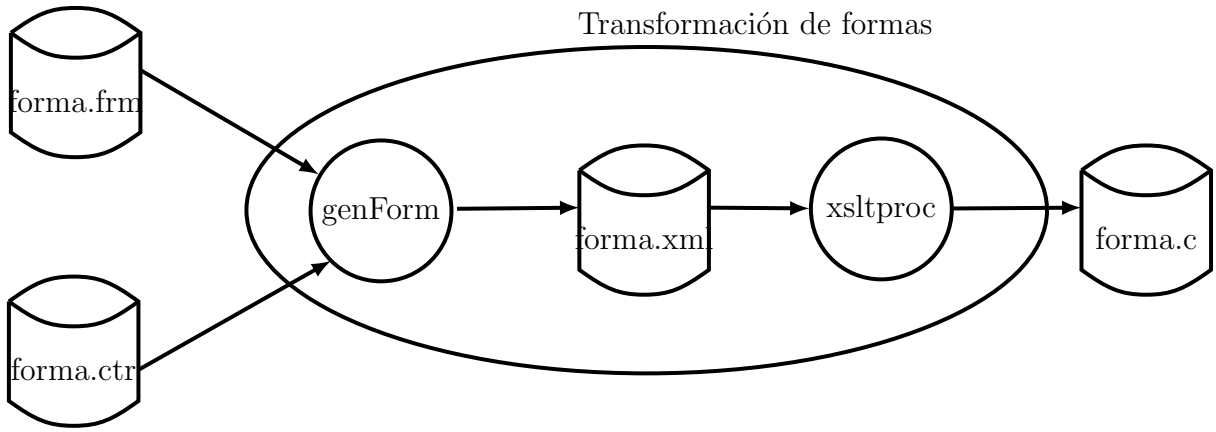

Figura 5: Secuencia de transformaciones

3.1. Lenguaje de archivo

Un archivo es descrito con un lenguaje que permite identificar los siguientes elementos: El *nombre* del archivo que es una secuencia de letras y dígitos que solamente se permite iniciar con dígitos. La *extensión* que permite declarar el tipo del archivo, este es una secuencia de carácteres y símbolos. Un archivo puede ser identificado con la combinación del *nombre*

Las siguientes propiedades de los archivos son opcionales:

La *ubicación* puede ser dada de dos formas: *absoluta* y *relativa*. En cualquier caso, es una secuencia de identificadores de archivos (son semejantes a los nombres de los archivos) separados por el carácter `/`. En el caso *absoluto* comienza con el carácter `/`, en el caso *relativa* comienza con el carácteres `./` o los caracteres `../`. En ningún caso el nombre de una ubicación puede terminar con el carácter `/`.

La *fecha de creación* la fecha, en el formato `DD/MM/AAAA`, en cual el archivo fue creado.

La *fecha de modificación* la fecha de modificación, en el formato `DD/MM/AAAA`, en cual el archivo fue modificado.

La *hora de creación*, en el formato `HH:MM:SS`, determina la hora exacta en la cual el archivo fue creado.

La *hora de modificación*, en el formato `HH:MM:SS`, determina la hora exacta en la cual el archivo fue modificado.

En muchas ocasiones, la definición de un archivo se puede omitir dado que

el archivo es identificado de manera única con el nombre del archivo (*nombre* y *extensión*). En otros caso con una combinación de la *ubicación* y el *nombre* con la extensión.

3.2. Lenguaje de comandos

Los comandos que transforma los programas se identifican por dos elementos: el *nombre*, es el nombre del comando; *ubicación*, donde está ubicado el comando en el sistema de archivos.

3.3. Lenguaje de aplicación

La aplicación utiliza el comandos definido en la sección 3.2 para darle un significado nuevo. Cada aplicación recibe una *identificación* que es única dentro de todo el archivo de definición del proyecto. El nombre del *comando* asociado. Recibe una secuencia de argumentos y el nombre del argumento a retornar, describe como se va aplicar el comando con las opciones y los argumentos que van a ser aplicados. En algunos casos el lenguaje de aplicación tiene una *descripción* sobre el significado del comando.

3.4. Lenguaje de clase de archivos

Como ya vimos en la sección 2 un grupo de archivos se pueden clasificar en un clase, ya sea por una propiedad o la combinación de varias propiedades.

Cada clase de archivos tiene una *identificación* que será única entre todas las clases definidas. Opcionalmente, puede tener una *descripción*.

Las clases de archivos pueden definir propiedades, que es una secuencia de tripletas que son: *clave*, *valor* y *tipo* (ver sección 3.7).

Una clase se define utilizando las propiedades de los archivos: *fecha creación*, *hora creación*, *nombre*, *extensión*, *id*[4]. La definición de una clase se da a través de una expresión booleana del lenguaje de expresiones 3.5.

[4]Recuerde la identificación es la combinación del nombre, el punto (`.`) y la extensión

3.5. Lenguaje de expresiones

$$
\begin{array}{rcl}
\textit{Expresión} & \rightarrow & \textit{Conjunción} \mid \textit{Expresión} \mid\mid \textit{Conjunción} \\
\textit{Conjunción} & \rightarrow & \textit{Relación} \mid \textit{Conjución} \;\&\&\; \textit{Relación} \\
\textit{Relación} & \rightarrow & \textit{Adición} \mid \textit{Relación} < \textit{Adición} \\
 & \mid & \textit{Relación} <= \textit{Adición} \mid \textit{Relación} > \textit{Adición} \\
 & \mid & \textit{Relación} >= \textit{Adición} \mid \textit{Relación} == \textit{Adición} \\
 & \mid & \textit{Relación} \mathrel{!}= \textit{Adición} \\
\textit{Adición} & \rightarrow & \textit{Término} \mid \textit{Adición} + \textit{Térm} \\
 & \mid & \textit{Adición} \mid \textit{Término} \\
\textit{Término} & \rightarrow & \textit{Negación} \\
 & \mid & \textit{Término} * \textit{Negación} \\
 & \mid & \textit{Término} \;/\; \textit{Negación} \\
\textit{Negación} & \rightarrow & \textit{Factor} \mid !\textit{Factor} \\
\textit{Factor} & \rightarrow & \textit{Identificador} \mid \textit{Propiedad} \mid \textit{Literal} \mid (\textit{Expresión})
\end{array}
$$

3.6. Lenguaje de funciones de comandos

Las aplicaciones de comandos 3.2 se pueden combinar para forma funciones que reciban varios argumentos que son archivos o clases de archivos y estos producen un archivo o una clase de archivo. Este es básicamente un lenguaje de definición de funciones, con una lista de argumentos de entrada, y un tipo de salida. Una función esta dividad en dos partes: una declaración de variables, que es una secuencia de declaraciones de variables y una secuencia de instrucciones.

Las intrucciones son de los siguientes tipos: bloques, asignación, instrucción condicional y ciclo (`while`). El lenguaje también acepta las expresiones como están definidas en la sección 3.5.

3.7. Lenguaje de tipos

Los tipos básicos del lenguaje son: *entero*, *booleano*, *fecha*, *hora*, *archivos*, *clases de archivos* y *cadenas de carácteres*.

4. Requerimientos

4.1. Técnicos

Lenguaje de programación: El lenguaje de programación Java [1].

Control de versiones: Se va utilizar el manejador de versiones *subversion* [3].

Repositorio de control de versiones: Assembla (`http://www.assembla.com`).

Estructura de directorios: La siguiente la estructura de directorios o carpetas o *folders* que tendrá el cd, donde `+` es un directorio y `-` es un archivo.

```
|= - miembros.xml
|  - build.xml
|  + gramatica
|  |= gramatica.txt
|  + src
|  += + main
|  |  |= - Main<nombregrupo>Lexer.java
|  |  |= - Main<nombregrupo>LL1Parser.java
|  |  |= - Main<nombregrupo>LR1Parser.java
|  |= + lexer
|  |  |= + jflex
|  |= + grammardesc
|  |  |= - DefANTLR<nombregrupo>.g
|  |  |= - DefCup<nombregrupo>.cup
|  |  + co
|  |  | + edu
|  |  |  |= + eafit
|  |  |  |  |= + dis
|  |  |  |  |  |= + st0270
|  |  |  |  |  |  |= + <nombregrupo>
|  |  |  |  |  |  |  |= + lexer
|  |  |  |  |  |  |  |= + token
|  |  |  |  |  |  |  |= + parser
```

```
|  |  |  |  |  |  |  |= + ast
|  + examples
```

`miembros.xml` archivo que contiene la definición de los miembros del grupo. El siguiente es el contenido de un posible grupo.

```
<grupo>
   <nombre>Principal</nombre>
   <url>https://subversion.assembla.com/svn/LenguajeProyectos/</url>
   <miembro>
      <nombre>Juan Francisco Cardona McCormick</nombre>
      <codigo>198610250010</codigo>
      <email>fcardona@eafit.edu.co</email>
   </miembro>
   <miembro>
      <nombre>Alejandro Gómez Londoño</nombre>
      <codigo>201010001010</codigo>
      <email>alegomez544@eafit.edu.co</email>
   </miembro>
   <mainclases>
     <mainclass  dir="src/main" clase="MainPrincipalAntlr"/>
     <mainclass  dir="src/main" clase="MainPrincipalCup"/>
   </mainclases>
</grupo>
```

El *`url`* identifica el repositorio donde está almacenada la práctica. El nombre del grupo es de una sola palabra, en minúsculas. Se debe añadir el correo electrónico en la segunda entrega. *`mainclases`* es una lista con las clases principales del proyecto tanto para ANTRL como para CUP.

`gramatica` directorio que contiene la definición de la gramática.

`gramatica.txt` archivo de texto que contiene la definición basada en *EBNF* que sigue el formato que está definido en interactiva.

`gramaticadesc` un directorio donde todas las descripciones de las gramáticas son almacenadas.

`src` directorio donde residen los fuentes de los archivos de clases de java y los archivos generados por las descripciones de las gramáticas.

4.2. Administrativos

Grupo de trabajo: Grupo máximo de dos estudiantes. *Cada* miembro del equipo debe registrarse en assembla (`http://www.assembla.com`) con un usuario propio. Uno de los miembros debe crear un repositorio donde los miembros del equipo son los dueños del repositorio. Deben enviar una invitación al monitor de la materia y al profesor de la misma.

Primera entrega: Domingo 15 de Abril a las 18:00 pm. Esta se hará automática a través del sistema de control de versiones. Se *debe* cumplir todos los requerimientos señalados en este documento. Recuerde que la idea es entregar una gramática en el archivo de las gramáticas.

Control de versiones: El control de versiones no es solamente un herramienta que facilite la comunicación entre los miembros del grupo y del control de versiones, sino que también ayudará al profesor a llevar un control sobre el desarrollo de la práctica. Se espera que las diferentes registros dentro del control de versiones sean cambios graduales. En caso contrario, se procederá a realizar un escutrinio a fondo del manejo de control de versiones para evitar fraudes.

Segunda entrega: El objetivo de la segunda entrega en produccir un analizador sintáctico descente tipo $LL(*)$[5] a transformando la gramática obtenida en la primera entrega y comentada por el docente. Esta gramática debe aceptar la entrada de uno o varios archivos a través de la línea de comandos.

Entrega domingo 20 de Mayo a las 18:00 pm. Al igual que la primera entrega ésta se hará a través del manejador de versiones.

Tercera entrega: El objetivo de la tercera entrega es implementar a través de ***cups*** un análisador sintáctico ascendente utilizando a *CUP Parser Generator for Java*[6]. Este analizador debe aceptar la entrada de uno o varios archivos a través de la línea de comandos.

La entrega debe hacerse para el día 4 de Junio a las 18:00.

[5] ANTLR

[6] `http://www2.cs.tum.edu/projects/cup/`

Bibliografía

[1] Arnold, Ken. Gosling, James. The Java Programming Language. 2005.

[2] Crespi Reghizzi, Stefano. Formal Languages and Compilation. (2008). 83–87.

[3] Pilato, Michael C. Collins-Sussman, Ben. Fitzpatrick, Brian W. Version Control with Subversion. 2008.