Reporte de Investigación:

Flex & Bison

Pablo Fallas

Leonardo Picado

Estructuras de Datos 2
Segundo Cuatrimestre, 2013.
UCenfotec

Primera Parte

Introducción

Lex y Yacc (“*Yet Another Compiler-Compiler”*) fueron desarrollados en la década de los 70's en los Laboratorios Bell de AT&T, estuvieron disponibles desde la 7ma Edición de UNIX hasta la aparición de Flex y Bison (análogos a Lex y Yacc respectivamente) que cuentan con algunas características adicionales así como un mejor soporte para reducciones o expresiones muy largas o complejas.

Flex y Bison, son especificaciones que sirven para generar *tokenizers* y *parsers* en lenguaje C que reconozcan gramáticas libres de contexto, como lenguajes de programación.

Flex es el encargado de leer de la entrada, típicamente *stdin* y extraer de la misma los tokens reconocidos por el basado en un lenguaje de expresiones regulares.

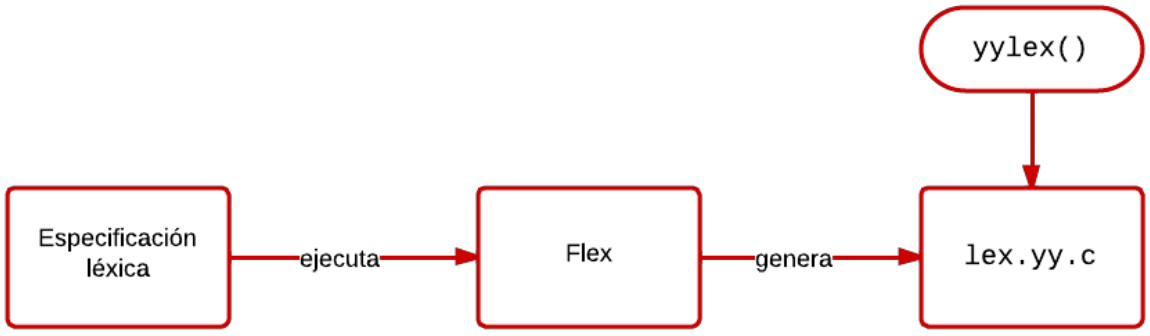
Bison sirve para generar *parsers*, usa a Flex para leer y reconocer sus *tokens* y basado en reglas sencillas en formato similar al *Backus-Naur Form*, es capaz de ir reduciendo una expresión con bastante eficiencia.

Flex

Flex es una herramienta que permite generar analizadores léxicos conocidos como escáneres (programas que reconocen patrones léxicos en el texto). Flex lee los archivos de entrada que serán usados como la especificación del analizador léxico, o la entrada estándar si no se dan los nombres de un archivo.

La especificación se organiza como pares de expresiones regulares y código C, denominadas reglas, después de su ejecución, Flex genera como salida un archivo fuente C llamado `lex.yy.c` correspondiente al analizador léxico a partir de esta especificación, que define una rutina `yylex()` que será la que tendremos que llamar para ejecutar el analizador léxico.

`lex.yy.c` se compila y se enlaza con la biblioteca `-lFL` para producir un ejecutable. Cuando se corre el ejecutable, este analiza su entrada en busca de casos de las expresiones regulares. Cada vez que encuentra uno, ejecuta el código C correspondiente.



Especificación Flex

El archivo de entrada de Flex es uno de texto que contiene tres secciones, cada una de ellas separadas por el carácter especial "%%". Las secciones son: definiciones, reglas léxicas y por último una sección opcional para el código de usuario.

- ➔ **Definiciones:** usadas para para declarar pares, se especifica un nombre asociado a una expresión regular, `<nombre> <expresión_regular>`.
  Ejemplo: `digito [0-9]`
  Digito a partir de este momento va a ser un nombre que va a ser utilizado cuando escribamos una expresión regular que corresponde exactamente a la expresión regular [0-9]. Vale recalcar que cuando queramos utilizar ese nombre deberá ir encerrado entre llaves “{}”.

- ➔ **Reglas**: es la sección más importante del archivo Flex, debido a que es donde vamos a indicar los patrones que se van a reconocer del fichero de entrada, en este caso tenemos el par <patrón> <acción>, el patrón será una acción regular y la acción será básicamente código C que se ejecutará cuando se reconozca en la entrada una secuencia de caracteres que calcen con este patrón o expresión regular.
  Ejemplo: `{digito} {ECHO;}`

- ➔ **Código de usuario**: esta sección es opcional, y el código que pongamos en ella se copiara al código de salida generado por Flex.

Ejemplo completo

```
digito [0-9]        // Definiciones
```

```
%%
{digito}  {ECHO;} // Reglas
{digito}+ {ECHO;} //
%%
main();           // Código de usuario
{ yylex(); }      //
```

Expresiones Regulares

Metacaracter	**Busca**
.	Cualquier carácter excepto cambio de línea.
\n	Cambio de línea.
*	Exactamente cero o más copias de la expresión anterior.
+	Al menos una o más copias de la expresión anterior.
?	Exactamente cero o una copia de la expresión anterior.
^	Principio de la línea.
$	Final de la línea.
a|b	a o b .
(ab)+	Una o más copias de ab (agrupación).
"a + b"	Literal "a + b".
[]	Clase o rango de caracteres.

Conflictos

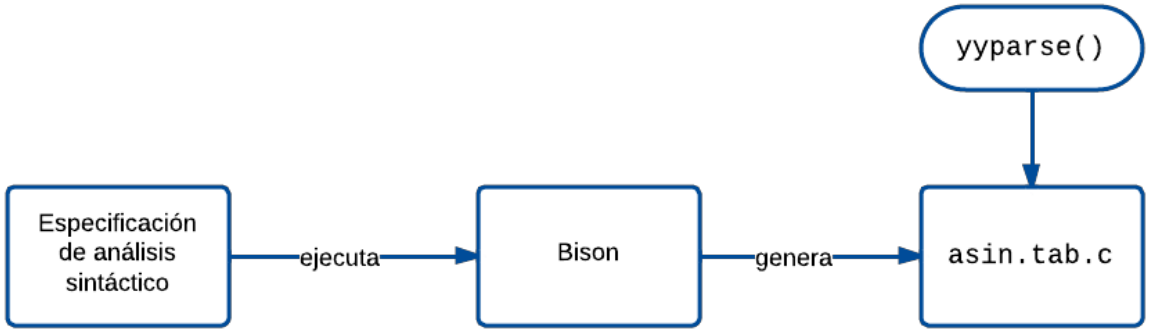
Si los caracteres de entrada coinciden con más de un patrón:

- ➔ El analizador escoge la secuencia que contiene más caracteres de ese patrón.
- ➔ En caso de igualdad en el número de caracteres en los patrones, se escoge el patrón que aparece de primero en el archivo Flex.

Bison

Bison es un generador de analizadores sintácticos (*parsers*) de propósito general que convierte una descripción de una gramática libre del contexto en un programa en C que analiza esa gramática. Básicamente es un programa que genera un programa que analiza la estructura de un archivo de texto. En lugar de escribir el programa, el usuario especifica qué cosas deben ser conectadas y con esas reglas se genera un programa que analiza el archivo.

El proceso parte de un archivo que contiene una gramática libre de contexto que es la especificación del analizador sintáctico, tras procesar la entrada Bison genera código C que por defecto usará el nombre `asin.tab.c`, en ese archivo existira una función llamada `yyparse()` que será la encargada de ejecutar el analizador sintáctico generado por Bison.



Especificación de Bison

La entrada de Bison es una gramática libre de contexto que contiene tres partes, la primera parte donde aparecerán declaraciones, una segunda parte separada por el carácter "%%" que contendrá las reglas gramaticales y una tercera parte

opcional separada también por los símbolos "%%" que contendrá el código del usuario.

- **Declaraciones** se definen los símbolos léxicos (*tokens*) que se van a utilizar en la parte sintáctica. Usualmente son los mismos tokens que se definieron en el analizador léxico. Opcionalmente junto a la declaración de estos símbolos léxicos podremos indicar asociatividad y precedencia asociada a ellos y tipos semánticos que puedan contener valores.

  Ejemplo: `%token CTE`

  Podemos observar que se ha declarado un símbolo léxico llamado CTE, este símbolo léxico se supone que estará definido en el analizador léxico quien será quien nos lo devuelva.

- **Reglas Bison:** es la parte más importante, donde escribiremos reglas de producción correspondientes a una gramática libre de contexto.

  Formato:
  ```
  <auxiliar> : <lado_derecho> { <accion> }
             | <lado_derecho> { <accion> }
             |
  ```

  Tenemos un lado izquierdo que es un símbolo no terminal, a continuación ":" y el lado derecho de la producción que estará formada por símbolos terminales y no terminales. Vale recalcar que los símbolos terminales son los símbolos devueltos por el analizador léxico, mientras que los auxiliares son los que se definen mediantes las reglas de producción. Además asociada a cada producción podremos escribir código en C que se ejecutará cada vez que el analizador sintáctico aplique esa producción. Las distintas producciones correspondientes al mismo auxiliar irán separadas entre sí por medio de una barra vertical "|", cuando escribamos la última producción correspondiente a un auxiliar finalizamos con ";".

Ejemplo:
```
expression  : expresion OPSUMA expression
            | CTE
            ;
```

En el ejemplo anterior indicamos que una expresión va a ser unida a otra expresión unida mediante un terminal llamado OPSUMA, OPSUMA tendrá que haber sido establecido en bloque de definiciones y además el analizador léxico nos lo tendrá que devolver. También podemos ver otra producción para el mismo no terminal separado por "|" que se reescribe como un terminal CTE, probablemente es una constante entera y como se trata de la última producción de un no terminal se finaliza con un ";".

Los terminales de la gramática se escriben en mayúscula y son devueltos por medio del analizador léxico. Además los símbolos auxiliares se escriben en minúscula y por defecto la primera regla que se escribe corresponde al símbolo inicial de la gramática.

➔ **Código de usuario**: sección opcional, donde se podrá escribir código C que se copiara al código generado por Bison.

Ejemplo completo

```
%{
#include "cabec.h"
%}

%token CTE FINLINEA              \\ Declaraciones
%token OPSUMA OPMUL

%%
inicio: expr FINLINEA { printf("\nFin"); } \\ Reglas
```

```
expr:   expr OPSUMA expr
      | expr OPMUL expr
      | CTE
      ;

%%
                        \\ Codigo de Usuario
```

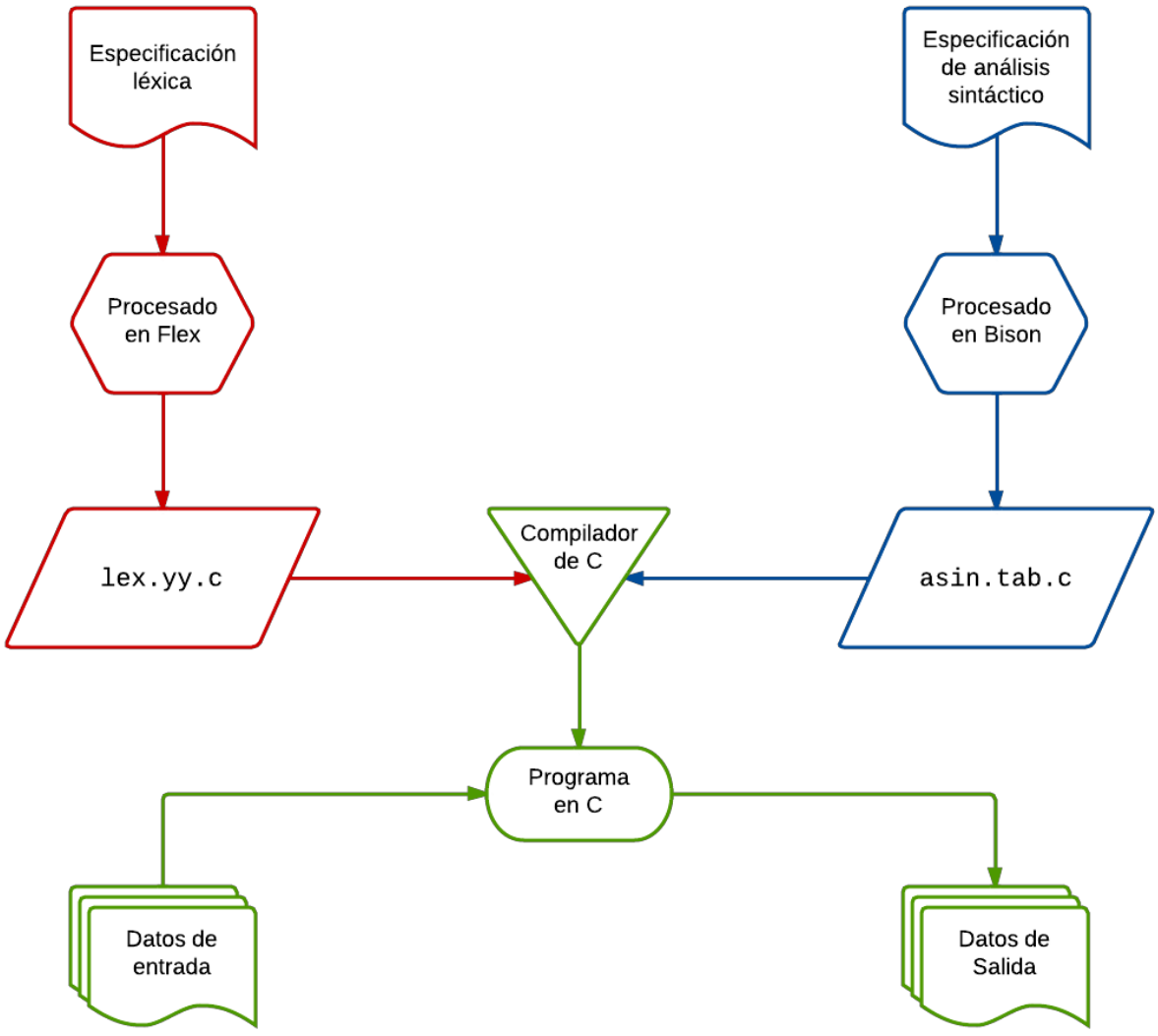
Acerca del código generado por Bison

El código del analizador sintáctico se va a llamar a través de la función `yyparse()`, esta función es generada por Bison cuando desde nuestro programa invoquemos al analizador sintáctico.

- ➔ El analizador sintáctico generado por Bison llama a la función `yylex()`, esperando que se le devuelva un token (analizador léxico).
- ➔ Bison está diseñado para facilitar el uso del analizador léxico generado por Flex, por lo tanto debido a la integración de estas dos herramientas es bastante sencilla.

Diagrama de flujo

A groso modo, podríamos graficar el flujo de uso de Flex y Bison hasta C de la siguiente manera:



Instalación

Instalación en Windows

Para poder instalar Flex y Bison en Windows es necesario descargar los ejecutables win_bison.exe y win_flex.exe, luego de esto procedemos a buscarlos y ejecutarlos utilizando la terminal CMD. Una forma de obtener información acerca de estas herramientas es utilizando el parámetro "help". Ejemplo:

```
c:\ Win_flex.exe --help
```

Instalación en Linux

Al ser utilidades del ambiente Linux, su instalación se hace mediante el manejador de paquetes que pertenezca a su distribución, ya sea *yum*, *aptitude* o cualquier otro. Para proceder a instalarlo abrimos una terminal y ejecutamos los siguientes comandos:

```
$: apt-get install flex
$: apt-get install bison
```

Después del segundo podemos ejecutar el siguiente comando para verificar que se haya instalado correctamente los programas:

```
$: flex -V && bison -V
```

Deberíamos tener una salida similar a:

```
flex 2.5.35
bison (GNU Bison) 2.3
Written by Robert Corbett and Richard Stallman.

Copyright (C) 2006 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
 There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
$:
```

Instalación en Mac OS X

Tanto Flex como Bison son instalados como parte de las herramientas de Xcode, para verificar que se encuentren instaladas podemos abrir una ventana de terminal y ejecutar el siguiente comando:

```
$: xcodebuild -version && flex -V && bison –V
```

Deberíamos tener una salida similar a:

```
Xcode 3.2
Component versions: DevToolsCore-1608.0;
DevToolsSupport-1591.0
BuildVersion: 10A432
flex 2.5.35
bison (GNU Bison) 2.3
Written by Robert Corbett and Richard Stallman.


Copyright (C) 2006 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE.
$:
```

Estamos listos para empezar.

Segunda Parte

Se tomó como base la primera calculadora del ejemplo brindado y se hicieron las siguientes modificaciones:

- Se cambió el nombre del *token* “integer” por “entero”.
- Se agregó un *token* nuevo “potencia”, usado para elevar un número a una potencia dada.
- Se agregaron tres elementos más al lenguaje:
  - Multiplicación: usando la expresión regular “por”, E-> E * E.
  - Raíz cuadrada: usando la expresión regular “raíz de”, E-> √E, su funcionalidad es relegada a la función sqrt del lenguaje.
  - Elevar a la potencia: usando el token “a la”, de la siguiente manera: E-> E^n, su funcionalidad es relegada a la función pow del lenguaje.

Las pruebas que se realizaron estaban orientadas a probar la exactitud de estos tres nuevos elementos, ya que fueron los tres únicos modificados. Es importante notar que no se otorgó ninguna prioridad a los nuevos operadores, por lo que los resultados son producto de resolver las operaciones usando una pila barriendo de derecha a izquierda.

Pruebas

El *setup* de cada prueba incluye la eliminación de todos los archivos previamente compilados y la re-compilación iniciando desde bison y flex hasta la generación de un archivo ejecutable.

Pruebas para multiplicación

- ➔ Se espera que 2 * 2 resulte en 4
- ➔ Se espera que 1 * 1 resulte en 1
- ➔ Se espera que 0 * 0 resulte en 0
- ➔ Se espera que 2 + 2 * 2 resulte en 6
- ➔ Se espera que 2 * 2 + 2 resulte en 8

Resultados para multiplicación

```
$: rm a.out && rm lex.yy.o y.tab.o && rm lex.yy.c y.tab.c && rm y.tab.h
$: bison -y -d calc1.y && flex calc1.l && gcc -c y.tab.c lex.yy.c && gcc y.tab.o lex.yy.o && ./a.out
conflicts: 20 shift/reduce
2 por 2
4 ✓
1 por 1
1 ✓
0 por 0
0 ✓
2 + 2 por 2
6 ✓
2 por 2 + 2
8 ✓
^C
```

```
$:
```

Pruebas para raíz cuadrada

- ➜ Se espera que √4 resulte en 2
- ➜ Se espera que √0 resulte en 0
- ➜ Se espera que √1 resulte en 1
- ➜ Se espera que √3+22 resulte en 5

Resultados para raíz cuadrada

```
$: rm a.out && rm lex.yy.o y.tab.o && rm lex.yy.c y.tab.c &&
rm y.tab.h
$: bison -y -d calc1.y && flex calc1.l && gcc -c y.tab.c
lex.yy.c && gcc y.tab.o lex.yy.o && ./a.out
conflicts: 20 shift/reduce
raiz de 4
2 ✓
raiz de 0
0 ✓
raiz de 1
1 ✓
raiz de 3 + 22
5 ✓
^C
$:
```

Pruebas para elevar a potencia

- ➜ Se espera que 1 potencia de 1 resulte en 1
- ➜ Se espera que 0 potencia de 0 resulte en 1
- ➜ Se espera que 2 potencia de 2 resulte en 4
- ➜ Se espera que 2 potencia de 2 + 2 resulte en 16

Resultados para elevar a potencia

```
$: rm a.out && rm lex.yy.o y.tab.o && rm lex.yy.c y.tab.c && rm y.tab.h
$: bison -y -d calc1.y && flex calc1.l && gcc -c y.tab.c lex.yy.c && gcc y.tab.o lex.yy.o && ./a.out
conflicts: 20 shift/reduce
1 a la 1
1 ✓
0 a la 0
1 ✓
2 a la 2
4 ✓
2 a la 2 + 2
16 ✓
^C
$:
```

Código fuente

calc1.l

```
 1 /* calculator #1 */
 2 %{
 3     #include <stdlib.h>
 4     #include "y.tab.h"
 5     extern int yylval;
 6     void yyerror(char *);
 7 %}
 8
```

```
 9 %%
10
11 [0-9]+        {
12                 yylval = atoi(yytext);
13                 return ENTERO;
14               }
15
16 [-+\n]        { return *yytext; }
17
18 "por"         { return *yytext; }
19
20 "raiz de"     { return *yytext; }
21
22 "a la"        { return POTENCIA; }
23
24
25 [ \t]       ;       /* skip whitespace */
26
27 .           yyerror("Error de sintaxis.");
28
29 %%
30
31 int yywrap(void) {
32     return 1;
33 }
```

calc1.y

```
 1 %{
```

```
 2     #include <stdio.h>
 3     #include <math.h>
 4     int yylex(void);
 5     void yyerror(char *);
 6 %}
 7
 8 %token ENTERO
 9 %token POTENCIA
10
11 %%
12
13 program:
14         program expr '\n'       { printf("%d\n", $2); }
15         |
16         ;
17
18 expr:
19         ENTERO
20         | expr '+' expr         { $$ = $1 + $3; }
21         | expr '-' expr         { $$ = $1 - $3; }
22         | expr 'por' expr       { $$ = $1 * $3; }
23         | 'raiz de' expr        { $$ = sqrt($2); }
24         | expr POTENCIA expr    { $$ = pow($1, $3); }
25         ;
26
27 %%
28
29 void yyerror(char *s) {
30     fprintf(stderr, "%s\n", s);
```

```
31 }
32
33 int main(void) {
34     yyparse();
35     return 0;
36 }
```

Referencias

- ➜ http://bdhacker.wordpress.com/2012/05/05/flex-bison-in-ubuntu/
- ➜ https://www.youtube.com/watch?v=vfDLKxbTGQo
- ➜ http://sourceforge.net/projects/flex
- ➜ http://sourceforge.net/projects/bison/
- ➜ http://www.escomposlinux.org/lfs-es/lfs-es-4.0/appendixa/bison.html
- ➜ http://webdiis.unizar.es/asignaturas/LGA/material_2004_2005/Intro_Flex_Bison.pdf
- ➜ http://www.manpages.info/macosx/lex.1.html
- ➜ https://www.youtube.com/watch?v=_zbIOMp63mo