

GUÍA BÁSICA DE FLEX Y BISON

PRÁCTICAS - PROCESAMIENTO DE LENGUAJES

GRADO EN INGENIERÍA INFORMÁTICA

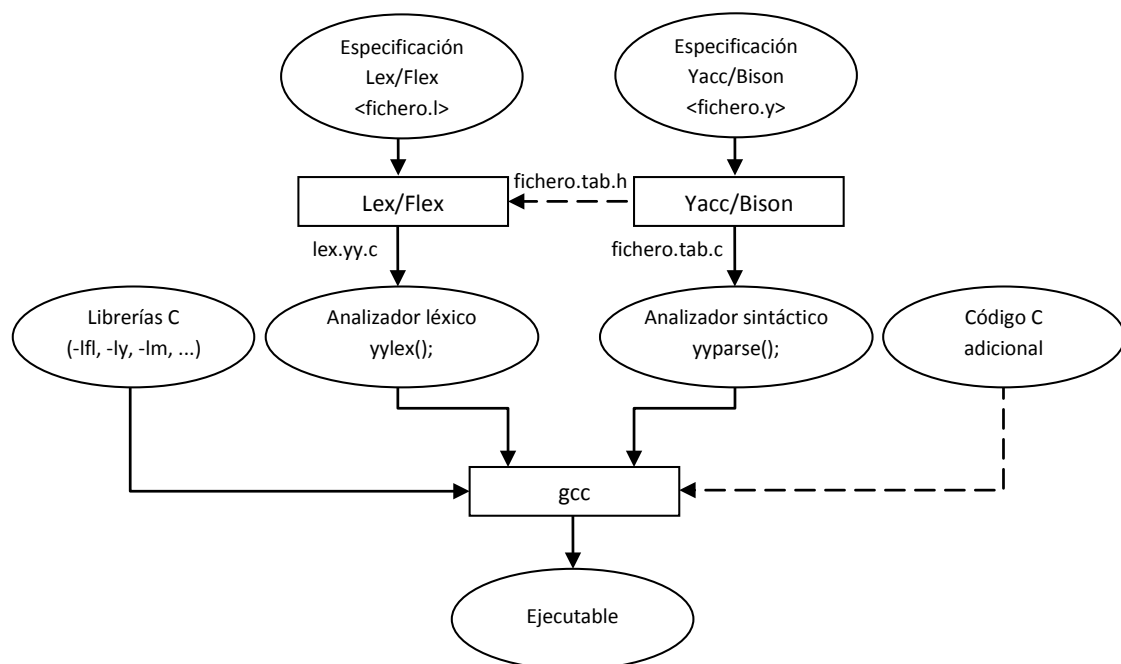
Curso 2016/2017

YACC/BISON: GENERADOR DE ANALIZADORES SINTÁCTICOS

1. Introducción

YACC (“Yet Another Compiler-Compiler”) es una herramienta generadora de analizadores sintácticos (“parsers”) de propósito general. Un analizador sintáctico es la parte del compilador que trata de darle sentido a la entrada reconocida por el analizador léxico.

YACC fue desarrollado por Stephen C. Johnson en AT&T para el sistema operativo UNIX y ha sido el precursor de este tipo de herramientas. Posteriormente, Robert Corbett desarrolló BISON que es la implementación equivalente de la herramienta YACC en el proyecto GNU; Richard Stallman lo hizo compatible con YACC y Wilfred Hansen le añadió nuevas funcionalidades como el soporte para literales multicarácter. Todas las gramáticas escritas para YACC deberían funcionar en BISON sin necesidad de ser modificadas.



Esquema general de compilación utilizando el analizador léxico y sintáctico

2. Estructura de un programa en BISON

BISON es un generador de analizadores sintácticos de propósito general que convierte una descripción para una gramática independiente del contexto (en realidad trabaja con una subclase de éstas, las LALR) en un programa en C que analiza dicha gramática. Por lo tanto, para que BISON analice un lenguaje éste debe poder ser descrito por una gramática independiente del contexto.

El sistema formal más común para representar las reglas que permiten definir una gramática es la forma normal BNF (Backus-Naur). Cualquier gramática expresada en BNF es una gramática independiente del contexto. La entrada de BISON es en esencia una BNF legible por una máquina ($A \rightarrow \alpha$).

La entrada de BISON

BISON acepta como entrada la especificación de una gramática independiente del contexto y produce una función en el lenguaje C que reconoce las instancias correctas de la gramática. El formato del archivo de entrada a BISON es el siguiente:

```
Declaraciones
%%
Reglas gramaticales
%%
Código C adicional
```

Los “%%” son signos de puntuación que aparecen en todo archivo de gramática de BISON para separar las secciones.

En la sección de **declaraciones** se definen los símbolos terminales y no terminales, los tipos de datos de los valores semánticos de los diferentes símbolos y, en caso necesario, la precedencia de operadores. Al igual que en LEX, la sección de declaraciones de BISON puede comenzar con un bloque de **declaraciones en C** para definir tipos y variables que se utilizan en la acciones (este bloque estaría asimismo delimitado por los signos de puntuación “%{” y “}%”).

Las **reglas gramaticales** son las **producciones** de la gramática, que pueden además llevar asociadas acciones (especificadas en código en C) que se ejecutan cuando el analizador encuentra las reglas correspondientes.

La sección de **código C adicional** puede contener cualquier código C que se desee utilizar. Se suelen incluir aquí las subrutinas invocadas por las acciones en las reglas gramaticales y la definición del analizador léxico *yyllex*. También se incluirá en esta sección la llamada al analizador sintáctico *yyparse*.

Dentro de las reglas gramaticales a cada tipo de unidad sintáctica o agrupación se le denomina **símbolo**. Como se ha mencionado, los símbolos especificados en las reglas se definen en la sección de declaraciones y pueden ser de los siguientes tipos:

- **Token:** fragmento de la entrada que se corresponde a un solo **símbolo terminal**. Por convención se suelen escribir en mayúsculas. P. ej., en el lenguaje C los tokens serían las palabras reservadas, los operadores aritméticos, las marcas de puntuación, etc. En BISON este tipo de símbolos se declaran del siguiente modo:

```
%token TOKEN1 TOKEN2
...
%token NUMBER /* equivale a #define NUMBER ??? */
%token NUMBER 300 /* equivale a #define NUMBER 300 */
```

Un token representa una clase de símbolos terminales equivalentes sintácticamente. Se emplea el símbolo en las reglas de la gramática para indicar que está permitido un token de esa clase. El símbolo se representa en BISON por un código numérico y la función del analizador léxico devuelve un código de tipo de token para indicar qué tipo de Token se ha leído. **A efectos de análisis sintáctico sólo se necesita el símbolo para representarlo, no el valor concreto.**

Nota: el valor devuelto por la rutina de análisis léxico es el código numérico de uno de los tokens.

- Los **símbolos no terminales** se construyen agrupando otros símbolos. Por convención se suelen escribir en minúsculas y se declaran en BISON con la siguiente definición:

```
%type NoTerminal1 NoTerminal2
```

E.g:

```
%type expr (expr -> NUM '+' NUM)
```

El **símbolo de arranque o axioma** es el símbolo no terminal que define una declaración completa del lenguaje. En un compilador este símbolo representaría un programa completo. Por ejemplo, en C sería la secuencia completa de definiciones y declaraciones. La expresión $1 + 2$ sería válida dentro de un programa en C pero no como programa en C completo, es por ello que un símbolo que represente una expresión, como en este caso, no podría ser el símbolo de arranque. A no ser que se le indique otra cosa, BISON asume que el símbolo inicial para la gramática es el primer no terminal que encuentra en la sección de especificación de la gramática; se puede indicar el axioma específicamente mediante la siguiente declaración:

```
%type NoTerminal1 NoTerminal2
```

```
%start S
```

En síntesis, el analizador BISON acepta como entrada una secuencia de tokens y los agrupa empleando las reglas gramaticales. Si la entrada es válida, finalmente dicha secuencia de tokens entera se reduce a una sola agrupación cuyo símbolo es el símbolo de arranque de la gramática.

La salida de BISON: el archivo del analizador

Cuando se ejecuta, BISON toma como entrada un fichero (normalmente con extensión .y) en el que se especifica la gramática. La salida producida es un programa en código C capaz de analizar el lenguaje descrito por la gramática. Así, este programa generado será el analizador para el lenguaje aceptado por la gramática especificada.

3. Sintaxis de las reglas gramaticales

Una regla gramatical de BISON tiene la siguiente forma general, donde **resultado** es un símbolo no terminal que describe la regla y **c1, c2, ..., cn** son componentes y se corresponden con una serie de símbolos terminales y no terminales:

```
resultado: c1 c2 ... cn  
;
```

Cuando existen varias reglas para el mismo resultado se pueden escribir por separado o bien unirse mediante el carácter “|”. Es posible que alguno de los componentes se corresponda con la cadena vacía (reglas *lambda*), siendo habitual en este caso escribir el comentario */*vacío*/* en la cadena sin componentes.

```
r:      /*vacío*/  
| componentes_1  
| componentes_2  
;
```

Una regla se dice **recursiva** cuando el no-terminal resultado aparece también el lado derecho de la misma. En BISON casi todas las gramáticas utilizan la recursión, ya que es la única manera de definir una secuencia de cualquier número de elementos.

```
expseq1: exp  
| expseq1 ' , ' exp  
;
```

Ejemplo de recursión por la izquierda

```
expseq1: exp  
      | exp ',' expseq1  
      ;
```

Ejemplo de recursión por la derecha

Cualquier tipo de secuencia se puede definir utilizando ya sea la recursión por la izquierda o la recursión por la derecha, aunque es recomendable utilizar siempre recursión por la izquierda, ya que de esta forma es posible analizar una secuencia de elementos sin ocupar espacio de pila, es decir, de forma mucho más eficiente en memoria.

La recursión indirecta o mutua ocurre cuando el resultado de la regla no aparece directamente en el lado derecho, sino que aparece en las reglas de otros no terminales del lado derecho. En el siguiente ejemplo se definen dos no-terminales recursivos mutuamente, pues cada uno referencia al otro:

```
expr: primario  
     | primario '+' primario  
     ;  
primario: constante  
        | '(' expr ')'  
        ;
```

4. Acciones semánticas

El analizador sintáctico tiene en cuenta los tokens por su significado, así si en una regla se menciona el símbolo terminal `NUMBER` en representación de los números enteros, esto quiere decir que cualquier número entero será gramaticalmente válido en esa posición sin importar su valor, ya que el valor preciso de la constante es irrelevante cuando se analiza la entrada. El valor semántico (valor preciso) es muy importante en la acción a realizar una vez llevado a cabo el análisis de la entrada.

En una gramática BISON cada token tiene dos partes: un **valor sintáctico** (tipo del token) y un **valor semántico** (valor concreto del mismo). Por ejemplo, un token podría declararse como un tipo de token `NUMBER` y tener el valor semántico 4. Para hacer referencia al valor semántico de uno de los componentes de la regla se emplea la notación `$i`, donde `i` es el número de orden (empezando en uno) que tiene el componente de la regla que se quiere referenciar. Con la notación `$$` se indica el valor semántico del resultado.

```
expr: expr '+' expr { $$ = $1 + $3; }
```

```
    expr ($$)
```

```
   /  |  \
  1($1) '+' 2($3)
```

En un programa sencillo podría ser suficiente con emplear el mismo tipo de datos para los valores semánticos de todas las construcciones del lenguaje (por defecto BISON emplea el tipo `int` para todos los valores semánticos). Sin embargo, en la mayoría de los programas se precisarán diferentes tipos de datos para diferentes clases de tokens y agrupaciones: una constante numérica podría necesitar el tipo `int` o `long`, una cadena el tipo `char *`, un identificador podría necesitar un puntero a la tabla de símbolos, etc. Para dar cabida a esta variedad de tipos de datos se puede emplear la definición **`%union`**, especificando para cada símbolo no terminal y terminal el tipo concreto que se va a emplear en lugar del tipo por defecto (`int`).


```

%union {
    int tipo_int;
    double tipo_double;
    char * tipo_string;
}

%token <tipo_int> INTEGER
%token <tipo_double> REAL
%token <tipo_string> CADENA

```

Los tipos de datos enumerados dentro de **%union** no tienen que ser necesariamente tipos simples, sino que se pueden incluir estructuras de datos todo lo complejas que se precisen (structs en C, punteros a funciones, etc.).

```

typedef struct { ... } tipo_complejo;
%union {
    tipo_complejo * tc;
}
%type <tc> NT

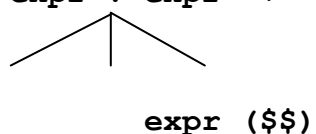
```

Como ya se ha indicado, una regla gramatical puede tener una acción compuesta de sentencias en C (de modo análogo a lo que sucedía en LEX con los patrones basados en expresiones regulares). Cada vez que el analizador reconozca una correspondencia para esa regla se ejecutará la acción asociada a la misma. La mayoría de las veces el propósito de una acción es computar el valor semántico de la construcción a partir del valor semántico de sus componentes; por ejemplo, supongamos una regla que indica que una expresión puede ser la suma de dos expresiones, cada una de las sub-expresiones poseerá un valor semántico. La acción para esta regla debería crear un tipo de valor similar para la expresión mayor que se acaba de reconocer.

```

expr : expr '+' expr { $$ = $1 + $3; }

```



expr (\$\$)

5. Obtención de los tokens a partir del texto de entrada

El analizador sintáctico espera que el usuario haya definido previamente la función de análisis léxico (denominada también “yylex”) que proporcionará los tokens de entrada. Recordemos que al generar el escáner con la herramienta Flex el nombre de la rutina de análisis léxico es precisamente ese: `int yylex()`.

En la definición del analizador léxico el valor que se debe devolver será el código del token que se ha reconocido.

```
[0-9]+    { return NUMBER; }
```

El analizador léxico, además de devolver el tipo del próximo token, puede poner su valor asociado en la variable global `yylval` que siempre está disponible. Dicha variable global es por defecto de tipo entero, aunque si se han declarado tipos de datos propios para los tokens se convierte en tipo “union”, con tantos campos como se hayan declarado en el archivo de la gramática BISON con la definición.

```
Declaración:
%union {
    int val;
}
Acceso:
yylval.val = 1;
```

Ejemplo completo para un sumador de números enteros simple (ficheros *ejemplo.l* y *ejemplo.y*):

```
%{
#include "ejemplo.tab.h"
}%
digito [0-9]
%%
{digito}+ { yylval.val = atoi(yytext); return NUMBER; }
\+ { return '+'; }
```

Código fuente “ejemplo.l”

```
%{
#include <stdio.h>
%}
%union {
    int val;
}
%token <val> NUMBER
%type <val> expr
%%
expr: NUMBER '+' NUMBER {
    $$ = $1 + $3; printf("El resultado es : %d\n", $$);
}
%%
main () {
    yyparse();
}
```

Código fuente "ejemplo.y"

6. Tratamiento de errores

Cuando el analizador sintáctico BISON lee un token que no satisface ninguna de las reglas de la gramática se genera un error sintáctico (por defecto se muestra la cadena de salida `"syntax error"`) y, salvo que se le indique, se finaliza el análisis de la entrada. Incluyendo la directiva específica `"%error-verbose"` dentro de la sección de declaraciones del programa en BISON obtendremos una mayor información sobre el error producido.

El analizador sintáctico utiliza la función `yyerror(s)` para informar de los errores que se producen durante la ejecución del programa. Esta función puede ser redefinida dentro del fichero en BISON para adaptarse a las necesidades del problema a resolver.

```
%{
/* declaraciones en C opcionales */
/* ... */
void yyerror (char const *); /* declaración del prototipo de la función */
}%
sección de declaraciones
%%
sección de reglas gramaticales
%%
sección de código C adicional
/* ... */
void yyerror (char const *s) {
    fprintf(stderr, "Error: %s\n", s);
}
/* ... */
```

Dentro de las reglas de la gramática podemos definir cómo tratar los errores sintácticos que se produzcan mediante reglas específicas para casos de error ó utilizando el token especial `"error"`. Este token es un símbolo terminal que siempre está definido (no hace falta declararlo) y que está reservado para el tratamiento de errores. El analizador sintáctico BISON generará un token `error` siempre que se produzca un `"syntax error"` y si existe una regla que lo reconozca el análisis sintáctico podrá continuar.

```
stmt: '(' expr ')'    { /* acción regla correcta */}
      | '(' error ')' { /* error: se esperaba una expr */
                      yyerror("Error en expr"); yyclearin; }
      ;
```

7. Compilación y ejecución

El proceso de diseño de lenguajes empleando BISON se compone de cuatro etapas:

1. Especificar la gramática en un formato que reconozca BISON. Para cada regla gramatical se ha de describir la acción que se va a tomar cuando una instancia de esa regla sea reconocida.
2. Escribir un analizador léxico para procesar la entrada y proporcionar los tokens al analizador sintáctico.
3. Escribir una función de control que llame al analizador producido por BISON.
4. Escribir las rutinas de informe de errores.

Para generar código ejecutable a partir del código fuente escrito en el fichero de especificaciones de la gramática (.y), son necesarios los siguientes pasos:

- **flex ejemplo.l**
Generará el fichero “lex.yy.c” (fichero que contiene llamada a “yylex()”)
- **bison ejemplo.y -yd**
Generará los ficheros “y.tab.c” y “y.tab.h”
- **gcc -o ejemplo.out lex.yy.c y.tab.c -lfl -ly**
Se obtendrá el fichero ejecutable “ejemplo.out”

YACC genera por defecto como salida los ficheros “y.tab.c” y “y.tab.h”; BISON en cambio genera los ficheros “<nombre_fichero>.tab.c” y “<nombre_fichero>.tab.h” pero compilado con la opción “-y” dará como resultado ficheros con la nomenclatura utilizada por YACC. Esto se puede modificar utilizando la opción “-o <nombre_fichero>.tab.c”:

```
bison -o ejemplo.tab.c ejemplo.y -yd
```

Con la opción “-y” se indica a BISON que genere código estándar POSIX YACC. La opción “-d” indica que es necesario generar el fichero de cabeceras “y.tab.h”, que contiene las definiciones de los tokens para incluirlo en el correspondiente programa LEX/FLEX. La opción “-ly” le indica al compilador gcc que debe enlazar la librería YACC/BISON.

Si hemos utilizado la opción anterior con bison, habrá que modificar la llamada a gcc:

```
gcc -o ejemplo.out lex.yy.c ejemplo.tab.c -lfl -ly
```

8. Ejemplos

Ejemplo: Cálculo sobre mediciones de temperatura por horas. Dado un registro de temperaturas, en grados centígrados y fahrenheit, con el formato de línea: *hora temperatura unidad*, obtener la temperatura media total y la temperatura a las 12h en grados centígrados.

Ejemplo fichero entrada (temps.txt):

```
10 21.2 C
11 24.3 F
12 24.5 C
13 23.0 C
```

Reglas de la gramática:

```
S -> lista_temperaturas
lista_temperaturas -> lista_temperaturas medicion
                        | medicion
medicion -> HORA temperatura
temperatura -> VALOR_TEMPERATURA CENTIGRADOS
                | VALOR_TEMPERATURA FAHRENHEIT
```

Programa en flex (temperatura.l):

```
%{
#include <stdlib.h>
#include "temperatura.tab.h"
}%
entero "-"?[0-9]+
real "-"?[0-9]+"."[0-9]+
%%
C {return CENTIGRADOS;}
F {return FAHRENHEIT;}
{entero} {yyval.valInt = atoi(yytext); return HORA;}
{real} {yyval.valFloat = atof(yytext); return VALOR_TEMPERATURA;}
. ;
\n ;
%%
```

Programa en bison (temperatura.y):

```
%{
#include <stdio.h>
int numMedidas = 0;
%}
%union{
    int valInt;
    float valFloat;
}
%token CENTIGRADOS FAHRENHEIT
%token <valInt> HORA
%token <valFloat> VALOR_TEMPERATURA
%type <valFloat> temperatura medicion lista_temperaturas
%start S
%%
S : lista_temperaturas {printf("Temperatura Media en Total: %f C\n",
                             $1/(float)numMedidas);}
    ;
lista_temperaturas : lista_temperaturas medicion {$$ = $1 + $2;
numMedidas++;}
    | medicion {$$ = $1; numMedidas++;}
    ;
medicion : HORA temperatura { if ($1 == 12) {
                             printf ("Temperatura a las 12H: %f C\n", $2);
                             }
                             $$ = $2;}
    ;
temperatura : VALOR_TEMPERATURA CENTIGRADOS {$$ = $1;}
    | VALOR_TEMPERATURA FAHRENHEIT {$$ = (($1 - 32.0) * 5.0) / 9.0;}
    ;
%%
main() {
    yyparse();
}
```

Ejemplo Makefile:

```
FUENTE = temperatura
PRUEBA = temps.txt

all: compile run

compile:
    flex $(FUENTE).l
    bison -o $(FUENTE).tab.c $(FUENTE).y -yd
    gcc -o $(FUENTE) lex.yy.c $(FUENTE).tab.c -lfl -ly

run:
    ./$(FUENTE) < $(PRUEBA)

clean:
    rm $(FUENTE) lex.yy.c $(FUENTE).tab.c $(FUENTE).tab.h
```

9. Bibliografía

- Introduction to compiling techniques. A first course using ANSI C, LEX and YACC / Jeremy Peter Bennet. McGraw-Hill, [1996].
- Lex and Yacc / John R. Levine, Tony Mason, Doug Brown. Sebastopol. O'Reilly, [1992] XXII, 364 p.
- Flex and Bison. Text Processing Tools / John R. Levine. Sebastopol. O'Reilly Media, [2009], 304 p.
- Tutoriales y manuales de Lex/Flex en la web.
 - Web oficial bison:
<http://www.gnu.org/software/bison/>
<http://www.gnu.org/software/bison/manual/bison.html>
 - Web oficial flex:
<http://flex.sourceforge.net/>
<http://flex.sourceforge.net/manual/>
 - Otros recursos:
<http://dinosaur.compilertools.net/>
<http://gnu.org/2009/09/18/writing-your-own-toy-compiler/>
<http://www.mactech.com/articles/mactech/Vol.16/16.07/UsingFlexandBison>
http://www.medina-web.com/programas/documents/tutoriales/lex_yacc/