Introducción a Bison

Lenguajes de Programación y Procesadores de Lenguajes
Escuela Técnica Superior de Ingeniería Informática
Universitat Politècnica de València

Curso 2020-21

Índice

1. Presentación

Bison es un generador automático de analizadores sintácticos que también permite asociar acciones semánticas a las reglas gramaticales. Tal y como muestra la Figura 1, Bison recibe como entrada un fichero de texto (que por convención lleva la extensión `.y`) con las reglas de una gramática independiente del contexto, y genera código C que implementa un analizador sintáctico ascendente LALR(1) correspondiente a la especificación proporcionada.

El analizador sintáctico necesita que el analizador léxico le proporcione los símbolos léxicos. Este analizador léxico puede implementarse por el programador o usarse uno generado por la herramienta Flex tal y como se verá en la sección 3.

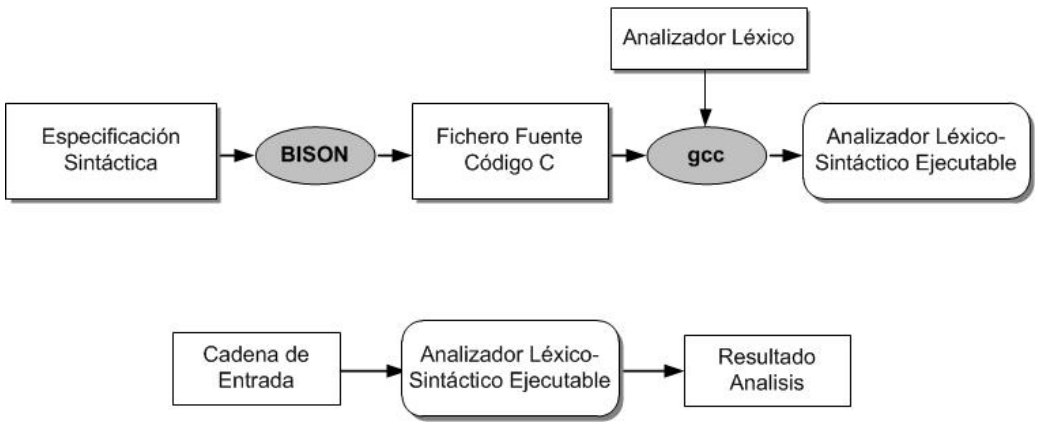

Figura 1: Uso de Bison

2. Descripción de un analizador sintáctico mediante Bison

Un fichero fuente de Bison se divide, al igual que un fichero fuente de Flex, en tres secciones separadas por una línea donde aparecen únicamente los caracteres `%%` :

1. **Declaraciones**: En esta sección aparece una subsección destinada a realizar declaraciones en código C delimitada mediante los caracteres `%{` y `%}` y otra destinada a declaraciones propias de Bison.

2. **Reglas Gramaticales**: En esta sección se definen las reglas gramaticales que constituyen la especificación sintáctica.

3. **Código C adicional**: En está sección opcional se definen funciones en el lenguaje C que se incluirán en el fichero C generado.

A continuación se presentan cada una de estas secciones en detalle.

2.1. Sección de declaraciones

Esta sección se encuentra dividida en dos subsecciones: declaraciones escritas en el lenguaje C y declaraciones propias de Bison.

2.1.1. Declaraciones en C

La sección de declaraciones en C contiene definiciones de macros y declaraciones de funciones y variables que se utilizan en las acciones asociadas a las reglas de la gramática.

A continuación vemos un ejemplo en el que se utiliza la directiva `#include` para incluir el archivo de cabecera `stdio.h` además de la declaración de la variable externa `int yylineno`. La variable `int yylineno` contiene los números de las líneas que se van procesando y ya está definida en el analizador léxico generado por Flex (por eso aqui se indica que es `extern`). Podemos utilizarla gracias a la integración de ambas herramientas que se verá en la Sección 3.

```
%\{
#include <stdio.h>
extern int yylineno;
%}
```

2.1.2. Declaraciones de Bison

La sección de declaraciones de Bison contiene la definición de los símbolos terminales (tokens o símbolos léxicos) y, opcionalmente, los no-terminales de la gramática. Los nombres de los tokens se declaran precedidos de la palabra `%token`, y serán los nombres que deberá devolver el analizador léxico en sus acciones asociadas a cada patrón. A modo de ejemplo, se muestra la declaración de alguno de ellos:

```
%token PARA_ PARC_ MAS_ MENOS_ POR_ DIV_
%token CTE_
```

Adicionalmente también puede aparecer una declaración `%error-verbose` para activar la verbosidad en la detección de errores. Esto hará que se muestren mensajes de error más detallados durante el análisis sintáctico.

2.2. Sección de Reglas Gramaticales

La sección de la reglas gramaticales contiene las reglas de la gramática que definen el lenguaje que se desea procesar. Debe haber siempre al menos una regla gramatical. Una regla gramatical de Bison tiene la siguiente forma general:

```
no-terminal: lado_derecho {acciones semánticas};
```

donde las acciones_semánticas (opcionales) estarán formadas por código C que se ejecutará cuando se aplique la reducción a la que están asociadas[1].

Para indicar que un no-terminal produce la cadena vacía basta con dejar el lado derecho de una de sus producciones vacío. Para separar las distintas producciones de un mismo no-terminal se usa la barra vertical | . El símbolo punto y coma (";") aparece al final de la última producción de cada no-terminal.

Por ejemplo, si un no-terminal S de una gramática tiene las reglas $S- > \varepsilon \mid abA \mid E$ su representación en Bison quedaría:

```
S:   |   a b A   |   E ;
```

que significa que S produce "cadena vacía", "abA" o "E".

En Bison se asume por defecto que el símbolo inicial de la gramática es el primer no-terminal que se encuentra en la sección de especificación de la gramática. Si se desea especificar explícitamente cuál es el símbolo inicial se debe usar la declaración `%start` seguida del símbolo inicial de la gramática.

2.3. Código C adicional

En esta sección opcional se incluyen funciones escritas por el usuario en el lenguaje C que se incluirán en el fichero generado por Bison. Estas funciones podrán invocarse desde las acciones de las reglas.

Cada vez que el analizador sintáctico generado por Bison detecta un error llamará a la función `yyerror`. La implementación de esta función debe realizarla el usuario/programador de Bison y puede incluirse en esta sección, en la sección equivalente de Flex, o en algún fichero externo, pero solo en una de ellas. Una sencilla implementación de esta función podría ser la siguiente:

```
void yyerror(const char *msg) {
   fprintf(stderr, "\nError en la linea %d: %s\n", yylineno, msg);
}
```

A modo de ejemplo se muestra a continuación el código completo de la especificación Bison para un sencillo lenguaje que reconoce expresiones matemáticas con números e identificadores, incluyendo la función main().

[1] La definición y uso de estas acciones no forma parte de este manual introductorio a Bison.

```
%{
#include <stdio.h>
#include "header.h"
%}

%token PARA_ PARC_ MAS_ MENOS_ POR_ DIV_
%token CTE_

%%
expMat : exp
        ;
exp    : exp MAS_   term
        | exp MENOS_ term
        | term
        ;
term   : term POR_ fac
        | term DIV_ fac
        | fac
        ;
fac    : PARA_ exp PARC_
        | CTE_
        ;

%%
void yyerror(const char *msg) {
   fprintf(stderr, "\nError en la linea %d: %s\n", yylineno, msg);
}

int main(int argc, char **argv) {
  int i, n=1 ;

  for (i=1; i<argc; ++i)
      if (strcmp(argv[i], "-v")==0) { verbosidad = TRUE; n++; }
  if (argc == n+1)
      if ((yyin = fopen (argv[n], "r")) == NULL)
          fprintf (stderr, "El fichero '%s' no es valido\n", argv[n]) ;
      else yyparse ();
  else fprintf (stderr, "Uso: cmc [-v] fichero\n");
return (0);
}
```

3. Integración Flex-Bison.

3.1. Generación de Analizadores Léxico-Sintácticos

El analizador sintáctico generado por Bison necesita interactuar con un analizador léxico para su correcto funcionamiento. El analizador léxico debe encargarse de analizar los lexemas de la entrada con el fin de identificar aquellos que corresponden con los tokens definidos en la sección de declaraciones de Bison y que se utilizan como terminales en las reglas gramaticales.

El proceso de análisis de un analizador sintáctico generado por Bison se inicia llamando a la función `yyparse()`. Esta función se encarga de llamar a la función `yylex()` que debe contener la implementación del analizador léxico. Generalmente este analizador léxico se encuentra en un fichero externo generado por Flex.

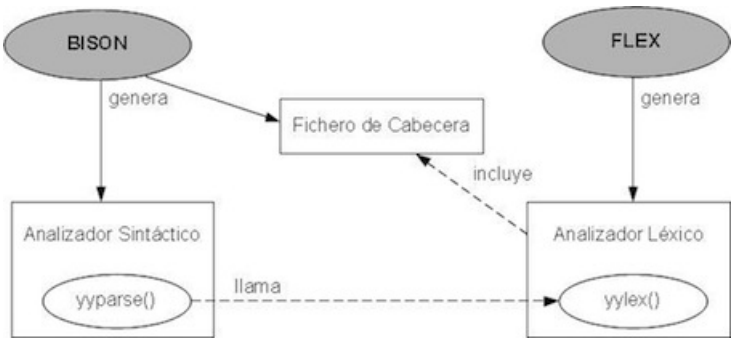

Figura 2: Integración de Bison con Flex

El analizador léxico generado por Flex debe detectar en la cadena de entrada los tokens definidos en Bison, por lo tanto se debe utilizar un mecanismo que permita compartir los nombres de los símbolos léxico en Bison y Flex. Dado que tanto el analizador léxico como el sintáctico están implementados en C, este mecanismo será un fichero de cabecera.

A modo de resumen en la Figura 2 se muestra cómo se realiza la integración Flex-Bison: Por un lado Bison se encarga de generar el analizador sintáctico que contiene la función `yyparse()`. También se encarga de generar un fichero de cabecera que le permitirá a Flex conocer los nombres de los símbolo léxicos definidos. Por otro lado, Flex se encarga de generar el analizador léxico que incluye la función `yylex()`, en cuya implementación se incluirá el fichero de cabecera generado por Bison.

Pasos para la integración Flex-Bison:

1. En primer lugar se debe ejecutar la herramienta Bison pasándole como argumento el nombre del fichero donde se encuentra la especificación sintáctica. Por convención, estos ficheros llevan la extensión `.y` (por ejemplo, `asin.y`). Además, para indicarle a Bison que debe generar el archivo de cabecera que permitirá compartir las declaraciones de tokens con Flex, se debe utilizar la opción `-d`. Si la especificación es correcta, Bison genera dos ficheros:

   - Por un lado el fichero C que constituye la implementación del analizador sintáctico. Por defecto el nombre de este fichero generado se construye reemplazando la extensión `.y` del nombre del fichero fuente Bison por la extensión `.tab.c`.
   - Bison también genera el fichero de cabecera necesario para integrase con Flex. Por defecto el nombre de este fichero se construye reemplazando la extensión `.y` del nombre del fichero fuente Bison por la extensión `.tab.h`.

   De este modo, a partir de un fichero Bison llamado `asin.y` se obtendrían los ficheros `asin.tab.c` y `asin.tab.h`.

   ```
   >   bison -d asin.y
   ```

   En caso de querer cambiar el nombre de los ficheros generados se puede usar la opción `-o` seguida del nombre del fichero C. En la llamada a Bison del siguiente ejemplo se generarán los ficheros `asin.c` y `asin.h`.:

   ```
   >  bison -oasin.c -d asin.y
   ```

2. Se ejecuta la herramienta Flex para generar la implementación del analizador léxico[2]. Hay que tener en cuenta que se debe incluir el fichero `.h` generado por Bison en la especificación léxica de Flex (fichero `.l`) mediante la directiva `#include`.

[2]Para más información de la compilación con Flex consultar la documentación de esta herramienta.

```
> flex -oalex.c alex.l
```

3. Los ficheros C generados por Bison y Flex se compilan con `gcc` y se enlazan con la librería `lfl` para obtener el ejecutable de un analizador léxico-sintáctico.

```
> gcc -omianalizador alex.c asin.c -lfl
```

4. Se puede ejecutar el analizador léxico-sintáctico para que analice la entrada estándar (sin argumentos) o el texto de un fichero (pasado por argumento) si se ha preparado el analizador léxico para leer y usar este argumento de entrada.

```
> mianalizador ejemplo
```

3.2. Adaptación de un analizador léxico para integrarse con Bison

Al integrar Bison con Flex, el analizador léxico generado por este último debe encargarse de analizar la cadena de entrada, identificar los tokens definidos en Bison y comunicar al analizador sintáctico la detección de dichos tokens. Por otro lado, la función `main` debe llamar a la función `yyparse` para que comience el análisis sintáctico. Si ya se dispone de una analizador léxico para Flex y se desea integrar con el analizador sintáctico generado por Bison, deben realizarse las siguientes tareas:

1. Incluir el fichero de cabecera generado por Bison en la sección de declaraciones C de Flex.
2. Las acciones de las reglas léxicas se encargarán de informar a Bison de los tokens identificados mediante la sentencia `return` seguida del nombre del token reconocido (uno de los nombres definidos en Bison mediante `%token`). Si se desea mostrar por pantalla el lexema reconocido puede emplearse la orden ECHO de Flex.
3. La función `main` debe llamar al analizador sintáctico: `yyparse()`.

A modo de ejemplo se muestra la especificación Flex necesaria para generar un analizador léxico que se integrará con el analizador sintáctico del ejemplo de expresiones matemáticas mostrado anteriormente:

```
%{
#include <stdio.h>
#include "header.h"

#define  retornar(x) {if (verbosidad) ECHO; return x ; }

%}
/*--------------------------------------------------------------------
Dado que las funciones "input()" y "unput(c)" son costosas y no las
utilizaremos, evitaremos que se generen automaticamente sus rutinas
correspondientes desactivandolas mediante las siguientes opciones:
--------------------------------------------------------------------*/
%option noinput
%option nounput
/*--------------------------------------------------------------------
Para mantener el numero de linea actual en la variable global yylineno
--------------------------------------------------------------------*/
%option yylineno

delimitador     [ \t\n]+
digito          [0-9]
```

```
entero          {digito}+

%%

{delimitador}   {if (verbosidad) ECHO ; }
"+"             { retornar (MAS_) ; }
"-"             { retornar (MENOS_) ; }
"*"             { retornar (POR_) ; }
"/"             { retornar (DIV_) ; }
"("             { retornar (OPAR_) ; }
")"             { retornar (DIV_) ; }
{entero}        { retornar (CTE_) ; }
. { yyerror("Caracter desconocido") ;}

%%
int verbosidad = FALSE;

void yyerror(const char *msg){
  fprintf(stderr, "\nError en la linea %d: %s\n", yylineno, msg);
}

int main(int argc, char **argv) {
  int i, n=1 ;

  for (i=1; i<argc; ++i)
      if (strcmp(argv[i], "-v")==0) { verbosidad = TRUE; n++; }
  if (argc == n+1)
      if ((yyin = fopen (argv[n], "r")) == NULL)
          fprintf (stderr, "El fichero '%s' no es valido\n", argv[n]) ;
      else yylex ();
  else fprintf (stderr, "Uso: cmc [-v] fichero\n");

  return (0);
}
```