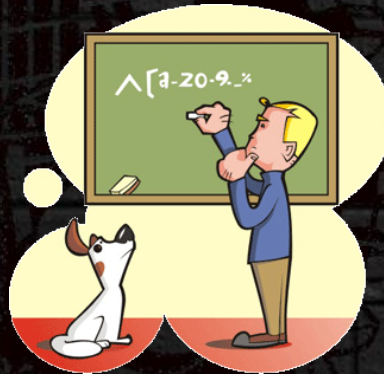




# Introducción a Bison



Compiladores  
Prof. Luis Enrique Hernández Olvera



## Bison

- Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción para una gramática independiente del contexto en un programa en C que analiza esa gramática.
- Un archivo fuente de Bison (un fichero con extensión .y) describe una gramática. El ejecutable que se genera indica si un fichero de entrada dado pertenece o no al lenguaje generado por esa gramática.

Compiladores  
Prof. Luis Enrique Hernández Olvera

2



## Compilación

### \$ bison fichero.y

- Compila la especificación del analizador y crea el fichero **fichero.tab.c** con el código y las tablas del analizador.

### \$ bison -d fichero.y

- Con la opción **-d** además del fichero **.c** se genera **fichero.tab.h** con las definiciones de las constantes asociadas a los tokens, además de variables y estructuras de datos necesarias para el analizador léxico.



## Compilación

### \$ gcc fichero.tab.c (ficheros .c)

- El usuario deberá de proporcionar sus propias funciones **main()**, **yyerror()** y **yylex()**.
- Dentro del código de usuario se deberá llamar a la función **yparse()** que a su vez llamará a la función **yylex()** del analizador léxico cada vez que necesite un TOKEN.



## Forma general de una gramática de Bison

- La forma general de una gramática de Bison es la siguiente:

### Declaraciones de Bison

%%

%{  
declaraciones en C  
%}

### Reglas y acciones gramaticales

%%

### Código C adicional

Los '%%', '%{' y '%}' son signos de puntuación que aparecen en todo archivo de gramática de Bison para separar las secciones.



## Sección de declaraciones en C

- Las declaraciones en C pueden definir tipos y variables utilizadas en las acciones. Puede también usar comandos del preprocesador para definir macros que se utilicen ahí, y utilizar #include para incluir archivos de cabecera que realicen cualquiera de estas cosas.





## Sección de declaraciones de Bison

- Las declaraciones de Bison declaran los nombres de los símbolos terminales y no terminales, y también podrían describir la precedencia de operadores y los tipos de datos de los valores semánticos de varios símbolos.



## Sección de reglas gramaticales

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



## Sección de código C adicional

- El código C adicional puede contener cualquier código C que desee utilizar. A menudo suele ir la definición del analizador léxico yylex, más subrutinas invocadas por las acciones en las reglas gramaticales. En un programa simple, todo el resto del programa puede ir aquí.



## Especificación de TOKENs y no terminales.

- Los **símbolos terminales** de la gramática se denominan en Bison **tokens** y deben declararse en la sección de definiciones. Por convención se suelen escribir los tokens en mayúsculas y los **símbolos no terminales** en minúsculas.
- Los nombres de los símbolos pueden contener letras, dígitos (no al principio), subrayados y puntos.



## Producciones en Bison

- Formato de una producción en Bison:

```
no_terminal:    componentes...
;
```

- Cuando varias reglas tienen el mismo no terminal a la derecha, se puede abreviar la notación en la forma :

```
no_terminal :    componentes_1
                | componentes_2
                | componentes_3
;
```



## Producciones en Bison

- *Los componentes* son los diversos símbolos terminales y no terminales que están reunidos por esta regla.
- Por ejemplo:

```
exp: exp '+' exp
;
```

- dice que dos agrupaciones de tipo exp, con un token '+' en medio, puede combinarse en una agrupación mayor de tipo exp.





## Producciones en Bison

- Si los *componentes* en una regla están vacíos, significa que *resultado* puede concordar con la cadena vacía.

```
exp1: /* vacío */  
      | exp2  
      ;  
exp2: exp3  
      | exp2 '+' exp3  
      ;
```



## Producciones en Bison

- Una regla se dice **recursiva** cuando su no-terminal también se encuentre en su lado derecho. Casi todas las gramáticas de Bison hacen uso de la recursión.

```
exp2: exp3  
      | exp2 '+' exp3  
      ;
```



## Semántica del lenguaje

- Las reglas gramaticales para un lenguaje determinan únicamente la sintaxis. La semántica viene determinada por los valores semánticos asociados con varios tokens y agrupaciones, y por las acciones tomadas cuando varias agrupaciones son reconocidas.



## Semántica del lenguaje

- Para utilizar más de un tipo de datos para los valores semánticos en un analizador, Bison requiere dos cosas:
  - Especificar la colección completa de tipos de datos posibles, con la declaración de Bison %union.
  - Elegir uno de estos tipos para cada símbolo (terminal o no terminal). Esto se hace para los tokens con la declaración de Bison %token y para los no terminales con la declaración de Bison %type.





## Semántica del lenguaje

- Directiva **%union**
- Especificación de YYSTYPE (tipo de todos los posibles valores semánticos) Incluye la declaración del tipo de datos asociado a los tokens y a los no terminales.

```
%union {
    int  entero;
    double real;
    char * texto;
}
```



## Semántica del lenguaje

- Directiva **%token**  
Declara un TOKEN indicado su nombre y opcionalmente su tipo (el tipo deberá ser uno de los identificadores declarados en la directiva %union).  
Formato : %token <tipo> nombre

```
%token BEGIN END IF THEN ELSE
%token <entero> CONSTANTE_ENTERA
%token <real> CONSTANTE_REAL
%token <texto> NOMBRE_VARIABLE NOMBRE_FUNCION
```



## Semántica del lenguaje

- Directiva **%type**  
Especifica el tipo de un no terminal.  
No es necesario declarar previamente los no terminales (se diferencian por que aparecen en el lado izq. de las reglas)  
Pero si es necesario especificar su tipo en el caso de que tengan asociado valores semánticos.  
Formato : %type <tipo> nombre\_no\_terminal

```
%type <entero> expresion_entera
%type <real> expresion_real
```



## Semántica del lenguaje

- Directiva **%start** : Identifica el no terminal que sirve como axioma de la gramática  
Formato : %start no\_terminal  
Por defecto bison utiliza como axioma de la gramática el no\_terminal en el lado izquierdo de la primera regla.



## Acciones

- Una acción acompaña a una regla sintáctica y contiene código C a ser ejecutado cada vez que se reconoce una instancia de esa regla. La tarea de la mayoría de las acciones es computar el valor semántico para la agrupación construida por la regla a partir de los valores semánticos asociados a los tokens.
- Una acción consiste en sentencias de C rodeadas por llaves. Se pueden situar en cualquier posición dentro de la regla; la acción se ejecuta en esa posición.



## Acciones

- El código C en una acción puede hacer referencia a los valores semánticos de los componentes reconocidos por la regla con la construcción  $\$n$ , que hace referencia al valor de la componente  $n$ -ésima (Pseudo-variables).

```
exp: ...
    | exp '+' exp
    { $$ = $1 + $3; }
```





## Acciones (Pseudo-variables)

- Las pseudo-variables \$\$, \$1, \$2,... permiten que dentro de las acciones se pueda acceder a los valores semánticos asociados a los símbolos de la regla.
- \$\$ contiene el valor semántico asociado al no terminal del lado izquierdo de la regla.
- \$1, \$2, ...,\$N contiene los valores semánticos asociados a los símbolos (TOKENs y no terminales) del lado derecho de la regla.



## Acciones (Pseudo-variables)

- El tipo de esas pseudo-variables será el tipo que se le haya asociado al símbolo correspondiente en la sección de declaraciones (directivas ).
- Cuando no se indica ninguna acción en una regla, BISON añade por defecto la acción  $$$=$1$ , en caso de que concuerden los tipos.



## Ejemplo

exp: exp '+' exp { \$\$ = \$1 + \$3; }

- En la acción, **\$1** y **\$3** hacen referencia a los valores semánticos de las dos agrupaciones exp, que son el primer y tercer símbolo en el lado derecho de la regla. La suma se almacena en **\$\$** de manera que se convierte en el valor semántico de la expresión de adición reconocida por la regla. Si hubiese un valor semántico útil asociado con el token '+', debería hacerse referencia con \$2.



## Especificando Precedencia de Operadores

- Bison le permite especificar estas opciones con las declaraciones de precedencia de operadores %left y %right. Cada una de tales declaraciones contiene una lista de tokens, que son los operadores cuya precedencia y asociatividad se está declarando. La declaración %left hace que todos esos operadores sean asociativos por la izquierda y la declaración %right los hace asociativos por la derecha.



## Especificando Precedencia de Operadores

- La precedencia relativa de operadores diferentes se controla por el orden en el que son declarados. La primera declaración `%left` o `%right` en el fichero declara los operadores cuya precedencia es la menor, la siguiente de tales declaraciones declara los operadores cuya precedencia es un poco más alta.



## Función del Analizador yyparse

- El código fuente de Bison se convierte en una función en C llamada `yyparse`.
- Se llama a la función `yyparse` para hacer que el análisis comience. Esta función lee tokens, ejecuta acciones, y por último retorna cuando se encuentre con el final del fichero o un error de sintaxis del que no puede recuperarse.
- El valor devuelto por `yyparse` es 0 si el análisis tuvo éxito y 1 si el análisis falló.





## Funcion del Analizador Léxico yylex

- La función del **analizador léxico**, `yylex`, reconoce tokens desde el flujo de entrada y se los devuelve al analizador (Bison no crea esta función automáticamente).
- En programas simples, `yylex` se define a menudo al final del archivo de la gramática de Bison. En programas un poco más complejos, lo habitual es crear un programa en Flex que genere automáticamente esta función y enlazar Flex y Bison.



## INTEGRACIÓN CON EL ANALIZADOR LÉXICO

- En BISON los TOKENs son constantes numéricas que identifican una clase de símbolos terminales equivalentes. El analizador léxico debe proporcionar una función **`yylex()`** que cada vez que sea llamada identificará el siguiente símbolo terminal en la entrada y devolverá la constante entera que identifica a ese tipo de TOKEN.
- Las constantes enteras asociadas a los TOKEN se definen en el fichero de cabecera **`.tab.h`** (generado usando la opción `-d`) que contiene los `#define` que asocian el nombre del TOKEN con su valor numérico.



## INTEGRACIÓN CON EL ANALIZADOR LÉXICO

- Para los caracteres simples no es necesario definir un TOKEN específico, dado que ya están edificados por su código ASCII. El analizador léxico simplemente deberá devolver su valor ASCII. No existe conflicto con los demás TOKENs ya que BISON les asigna valores enteros empezando en 256.
- En el caso de que los TOKENs tengan atributos asociados se usará la variable global de BISON `yylval` de tipo `YYSTYPE` (declarado mediante la directiva `%union`).



## Ejemplo (Compilación)

```
olvera@olvera-N551JW: ~/Desktop/Ejemplo2_Flex_Bison
File Edit View Search Terminal Help
olvera@olvera-N551JW:~/Desktop/Ejemplo2_Flex_Bison$ ls
ejemplo2.l  ejemplo2.y
olvera@olvera-N551JW:~/Desktop/Ejemplo2_Flex_Bison$ flex ejemplo2.l
olvera@olvera-N551JW:~/Desktop/Ejemplo2_Flex_Bison$ ls
ejemplo2.l  ejemplo2.y  lex.yy.c
olvera@olvera-N551JW:~/Desktop/Ejemplo2_Flex_Bison$ bison ejemplo2.y -d
olvera@olvera-N551JW:~/Desktop/Ejemplo2_Flex_Bison$ ls
ejemplo2.l  ejemplo2.tab.c  ejemplo2.tab.h  ejemplo2.y  lex.yy.c
olvera@olvera-N551JW:~/Desktop/Ejemplo2_Flex_Bison$ gcc lex.yy.c ejemplo2.tab.c
```



## Ejemplo (Compilación)

- Los mensajes warning se deben a que el ejemplo tienen lo mínimo necesario para funcionar.

```
olvera@olvera-N551JW:~/Desktop/Ejemplo2_Flex_Bison$ gcc lex.yy.c ejemplo2.tab.c
ejemplo2.tab.c: In function 'yyparse':
ejemplo2.tab.c:1127:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
    yychar = yylex ();
                   ^
ejemplo2.y:25:7: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]
    | exp '\n' { printf ("\tresultado: %d\n", $1); }
                  ^
ejemplo2.y:25:7: note: include '<stdio.h>' or provide a declaration of 'printf'
ejemplo2.tab.c:1286:7: warning: implicit declaration of function 'yyerror'; did you mean 'yyerrok'? [-Wimplicit-function-declaration]
    yyerror (YY_("syntax error"));
    ^~~~~~
    yyerrok
ejemplo2.y: At top level:
ejemplo2.y:42:1: warning: return type defaults to 'int' [-Wimplicit-int]
    yyerror (char *s)
    ^~~~~~
ejemplo2.y: In function 'yyerror':
ejemplo2.y:44:3: warning: incompatible implicit declaration of built-in function 'printf'
    printf ("--%s--\n", s);
    ^~~~~
ejemplo2.y:44:3: note: include '<stdio.h>' or provide a declaration of 'printf'
olvera@olvera-N551JW:~/Desktop/Ejemplo2_Flex_Bison$ ls
a.out  ejemplo2.l  ejemplo2.tab.c  ejemplo2.tab.h  ejemplo2.y  lex.yy.c
```




## Ejemplo (ejecución)

```
olvera@olvera-N551JW:~/Desktop/Ejemplo2_Flex_Bison$ ./a.out
5
Numero entero 5
Salto de línea
    resultado: 5
S+5
Numero entero 5
Signo op
Numero entero 5
Salto de línea
    resultado: 10
S*5
Numero entero 5
Signo op
Numero entero 5
Salto de línea
    resultado: 25
S+S*5
Numero entero 5
Signo op
Numero entero 5
Signo op
Numero entero 5
Salto de línea
    resultado: 30
Mod(5,3)
Modulo
Signo parentesis
Numero entero 5
coma
Numero entero 3
Signo parentesis
Salto de línea
    resultado: 2
```

Entradas

- Una vez recibida la entrada al programa el analizador léxico se encarga de identificar los componentes léxicos uno por uno de la trama ingresada y los manda al analizador sintáctico.





# Ejemplo (ejecución)

```

olvera@olvera-N551JW: ~/Desktop/Ejemplo2_Flex
olvera@olvera-N551JW:~/Desktop/Ejemplo2_Flex_Bison$ ./a.out
5
Numero entero 5
Salto de línea
      resultado: 5
5+5
Numero entero 5
Signo op
Numero entero 5
Salto de línea
      resultado: 10
5*5
Numero entero 5
Signo op
Numero entero 5
Salto de línea
      resultado: 25
5+5*5
Numero entero 5
Signo op
Numero entero 5
Signo op
Numero entero 5
Salto de línea
      resultado: 30
Mod(5,3)
Modulo
Signo parentesis
Numero entero 5
coma
Numero entero 3
Signo parentesis
Salto de línea
      resultado: 2

```

```


1  %{
2  /**include <math.h>
3  %}
4
5  /* Declaraciones de BISON */
6  %union{
7      int entero;
8  }
9
10 %token <entero> ENTERO
11 %token MOD
12 %type <entero> exp
13
14 %left '+'
15 %left '*' MOD
16
17 /* Gramática */
18 %%
19
20 input: /* cadena vacia */
21 ;
22
23 line: '\n' { printf ("\nresultado: %d\n", $1); }
24 ;
25
26 exp: ENTERO { $$ = $1; }
27 | exp '+' exp { $$ = $1 + $3; }
28 | exp '*' exp { $$ = $1 * $3; }
29 | MOD '(' exp ',' exp ')' { $$ = $3 % $5; }
30 ;
31
32
33
34

```

**Resultado  
del análisis  
sintáctico**

35

Prof. Luis Enrique Hernández Olvera



# Práctica

- Diseñar un programa que permita reconocer los lexemas de:
  - Números enteros con y sin signo. Ejemplo (5, 34, -100).
  - Números decimales con y sin signo. Ejemplo (.05, 0.51, -13.1, -3.1416).
- Diseñar programa que permita reconocer las gramáticas para las siguientes formas:
  - Operaciones matemáticas (+, -, \*, /)
  - Modulo como función Ejemplo( Mod(5,3) ó mOd(8.5,3) )
- Unir ambos programas para que el primero alimente de tokens al segundo.

36

Compiladores  
Prof. Luis Enrique Hernández Olvera