Universidad
Europea de Madrid

LAUREATE INTERNATIONAL UNIVERSITIES

ANÁLISIS SINTÁCTICO II

GENERADOR DE ANALIZADORES SINTÁCTICOS ASCENDENTES

Índice

Presentación

El objetivo de este tema es entender el funcionamiento de los generadores de analizadores sintácticos ascendentes, concretamente **Bison, basado en el método LALR.** Comenzaremos entendiendo los conceptos básicos de este tipo de herramientas y veremos la forma de construir un analizador sintáctico.

Los objetivos a conseguir en este tema son:

- Conocer el generador de analizadores sintácticos LALR Bison.
- Aprender el funcionamiento general de Bison.
- Comprender cómo funciona la sección de declaraciones.
- Comprender cómo funciona la sección de reglas.
- Entender la comunicación Flex-Bison.
- Incorporar precedencia y asociatividad con Bison.
- Comprender cómo se realiza la gestión de errores.
- Aprender a incorporar acciones semánticas.

Introducción a Bison

Bison es la versión GNU de YACC (*Yet Another Compiler of Compilers*). Es un generador de analizadores sintácticos de carácter general que convierte una gramática libre de contexto en un analizador determinista LR utilizando tablas del analizador sintáctico LALR.

Estructura de un fichero Bison

{definición}

%%

{reglas}

%%

{subrutinas de usuario}

¿Cómo son las reglas?

Reglas: <producción> acción

Producción	Son las producciones que pertenecen a una gramática tipo 2 en BNF.
Acción	Fragmentos de código C que especifican qué hacer cuando se reduce una producción.

La forma natural de trabajar es conjuntamente con Flex:

- yylex(): petición de siguiente *token.*

El analizador está contenido en la función **yyparse().**

Funcionamiento general de Bison

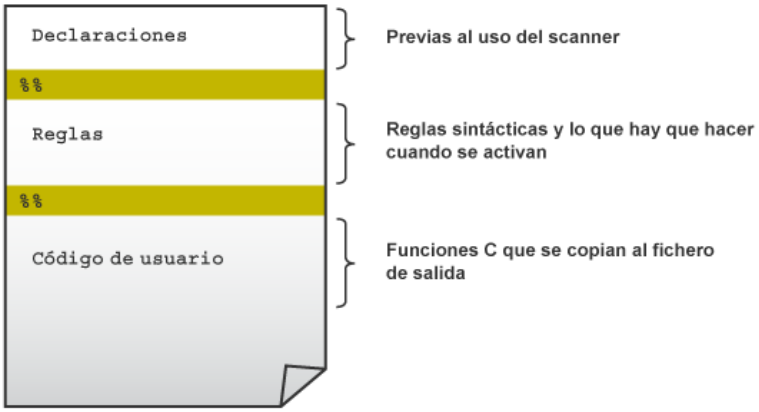
Hace un análisis previo de la gramática para construir el analizador LALR: notifica incidencias.

Funcionamiento general del analizador: pide *tokens* definidos al analizador léxico (yylex).

- Como es LALR(1), pide un *token* de pre-análisis.
- Cuando realiza desplazamiento pide el siguiente *token.*
- Cuando realiza reducciones de producciones y, por tanto, ejecuta el código en sus acciones.

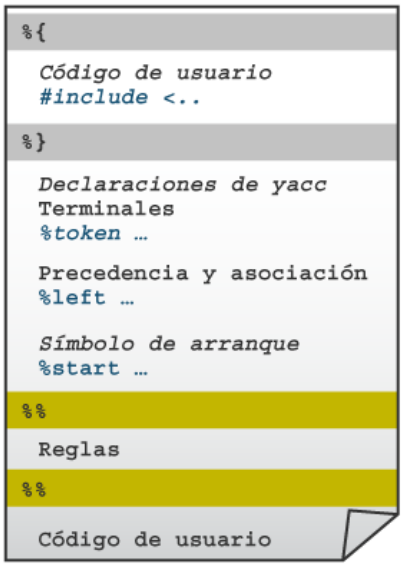
¿Cúal es el formato de un fichero en Bison?

Tres partes separadas por una línea con "%%".



Sección de declaraciones

Consta de dos partes:



1. Código de usuario
   - Se copia a la salida.
   - **%{....%}.**
2. Declaraciones
   - Terminales y no terminales (opcional):
     - **%token símbolo.**
     - **%type símbolo.**
   - Precedencia y asociatividad de operadores:
     - **%noassoc símbolos.**
     - **%left símbolos.**
     - **%right símbolos.**
   - Símbolo inicial de la gramática:
     - **%start símbolo.**
   - Directivas de comportamiento

En detalle Respecto de los terminales

Ejemplo Definición terminales en Bison En detalle Definición terminales en Flex

En detalle

Respecto de los terminales

- Es lo que devuelve la función **yylex()** (llamada automáticamente desde **yyparse().**
- Se convierten en **#define** en el fichero de salida.
- Se les asocia un número comenzando por 257.
- Se puede obligar a que tenga uno en concreto.
  - **%token T_LLAVE 345.**
- Los terminales de un carácter no hace falta declararlos (están implícitos).
  - En Bison se podrá utilizar directamente el terminal '('
  - Son tokens con valores numéricos el código ASCII (<257).

Ejemplo

Definición terminales en Bison

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

%token NUMERO, MAS, MENOS, POR, DIV, PAR_I, PAR_D
%start expr. /* simbolo axioma sentencial */
```

En detalle

Definición terminales en Flex

```
% {
#include "expresiones_tab.h"
%}
digito [0-9]


%%
[ \t]+    ;
{digito}+       {yylval=atoi(yytext) ;  return NUMERO; }
"+" return   MAS;
"-" return   MENOS;
"*" return   POR;
"/" return   DIV;
"(" return   PAR_I;
")" return   PAR_D;
.   {printf("token erroneo\n") ;}
```

Sección de reglas

Se utiliza un formato BNF simplificado:

LI: LD *acción*;

LI	Es un símbolo no-terminal del lenguaje.
LD	Secuencia de símbolos no-terminales y terminales.

¿Cómo se agrupan varias reglas del mismo no terminal?

```
expr: expr ´+´ expr {....}
    | expr ´-´ expr {....}
;
```

NOTA: Si se deja vacía es la regla de la palabra vacía (λ), como vemos en el siguiente ejemplo:

```
sentencias : sentencias ´;´ sentencia {....}
                  |
;
acción: { sentencias en C } (puede  ser vacío)
```

Ejemplo Declaración de reglas

Ejemplo

Declaración de reglas

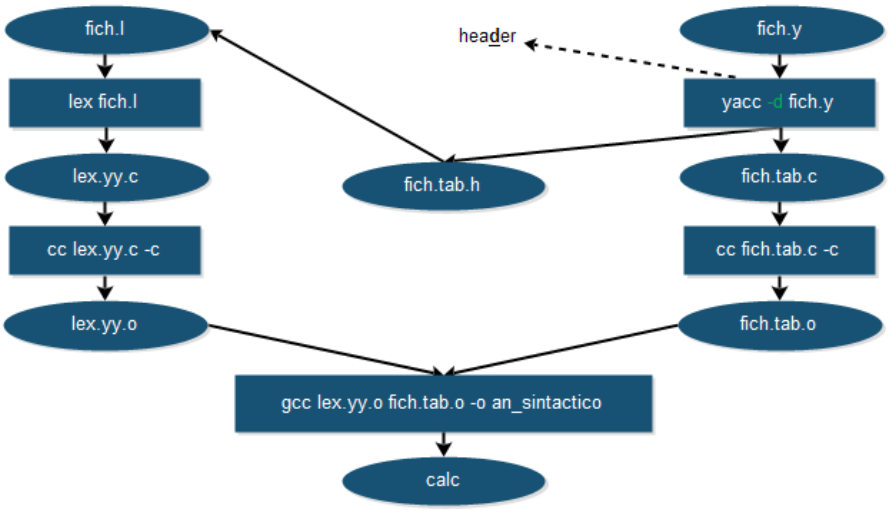
```
%token NUMERO, MAS, MENOS, POR, DIV, PAR_I, PAR_D
%start expr /* simbolo axioma sentencial */

%%
expr: expr  MAS  term {printf("expr   --> expr  MAS  term\n") ;}
          | expr  MENOS  term   {printf("expr   --> expr  MENOS term\n") ;}
          | term                          {printf("expr   --> term\n") ;}
;
term:    term POR factor         {printf("term  --> term  POR  factor\n") ;}
          | term  DIV  factor         {printf("term   --> expr  DIV  factor\n") ;}
          | factor                          {printf("term   --> factor\n") ;}
;
factor:  NUMERO                   {printf("factor--> NUMERO(%d) \n" ,$1) ;}
           | PAR_I  expr  PAR_D {printf("factor-->    (  expr  ) \n") ;}
;
%%
...
```

Comunicación Flex-Bison I

Por un lado tenemos el fichero "**fich.l**" con la especificación del analizador léxico escrita en formato flex y por otro lado tenemos el fichero "**fich.y**" con la especificación del analizador sintáctico escrita en formato Bison (lleva la extensión .y, porque deriva de yacc).

Si ejecutamos los pasos que aquí se indican obtenemos un fichero escrito en C para un compilador. En este ejemplo se obtiene una calculadora.



Comunicación Flex-Bison II

¿Qué debemos introducir en el fichero de Flex para producir la comunicación entre el analizador léxico y el sintáctico?

```
%{
#include "expresiones.tab.h"
%}
digito [0-9]
%option noyywrap
%%
[ \t]+  ;
{digito}+       {yylval=atoi(yytext); /*printf("lex: %s, %d\n ",yytext, yylval);*/  return NUMERO;}
"+" return MAS;
"-" return MENOS;
. . .
.  {printf("token erroneo\n");}
%%
```

```
%{
#include "expresiones.tab.h"
%}
```

Este fichero "expresiones.tab.h" lo genera Bison

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#defineNUMERO  258
#defineMAS     259
#defineMENOS   260
#definePOR     261
#defineDIV     262
```

```
.  {printf("token erroneo\n");}
%%
```

```
%{
#include "expresiones.tab.h"
%}
```

Los atributos, en este caso el valor, van en la variable yylval y se comunican así al analizador sintáctico. Representa el atributo del último token en el patrón reconocido.

```
[ \t]+  ;
{digito}+       {yylval=atoi(yytext); /*printf("lex: %s, %d\n ",yytext, yylval);*/  return NUMERO;}
"+" return MAS;
"-" return MENOS;
. . .
.  {printf("token erroneo\n");}
%%
```

```
%{
#include "expresiones.tab.h"
%}
digito [0-9]
```

No es necesario incluir un "main".

```
{digito}+       {yylval=atoi(yytext); /*printf("lex: %s, %d\n ",yytext, yylval);*/  return NUMERO;}
"+" return MAS;
"-" return MENOS;
. . .
.  {printf("token erroneo\n");}
%%
```

El valor de cada símbolo está disponible al realizar cada reducción, en el fichero "fich.y". Notación:

Producción


Ejemplo

En detalle

Producción (código acción)

expr: T ´+´ F {$$=$1+$3};

- $$ identifica a la parte izquierda de la regla de producción. En este caso $$ es "expr".
- $n se corresponde con el n-ésimo símbolo de la parte derecha. Por tanto $1 es T, $2 es + y $3 es F.

Por defecto se suponen valores enteros para el atributo de cada símbolo, aunque también pueden declararse tipos complejos para los atributos (números reales).

En detalle

Ejemplo

```
%%
%token NUMERO, MAS, MENOS, POR, DIV, PAR_I, PAR_D
%start S /* simbolo axioma sentencial */
%%
S: expr    {printf("resultado:  %d\n",  $$) ; }

expr:    expr MAS term                  {$$=$1+$3;}
          |expr MAS term              {$$=$1-$3;}
          |term                              {$$=$1;}
;
term:      term POR factor          {$$=$1*$3;}
          |  term DIV factor            {$$=$1/$3;;}
          |  term                            {$$=$1;}
;
factor:    NUMERO                    {$$=$1;}
          |  PAR_I  expr PAR_D    {$$=$1;}
;
...
```

Comunicación Flex-Bison III

¿Cómo incorporamos distintos tipos a los atributos que pasan los terminales de la gramática? Por defecto todos los terminales tienen un atributo de tipo entero, yyval, asignado en yylex():

- Para utilizar otro tipo de atributo puede usarse la variable YYSTYPE: #define YYSTYPE double.

Directiva %union:

- Da mayor flexibilidad, permitiendo definir varios atributos por símbolo y atributos alternativos.
- Declara los tipos posibles como una "unión" de C.

¿Cómo se utilizan estos atributos?

Uso de los tipos declarados en %union para que Bison esté informado del tipo de retorno de cada no terminal:

Se puede asociar a un terminal en su declaración	**%token <valent> NATURAL**
Para un no terminal en su declaración	**%type <tipo> NO_TERMINAL**
En una producción específica	**expr: NAT ´+´ NAT {$$=$<valent>1+$<valent>3};**

En un símbolo terminal (análisis léxico, "fich.l"): hay que especificar:

[-+]?{digito}+ { yyval.valent=atoi(yytext); return NUMERO;}

Veamos un ejemplo completo donde definimos la directiva %union con un struct denominado num:

Ejemplo Directiva %union

Variable YYSTYPE

De esta forma podríamos hacer operaciones con número en punto flotante.

Unión de C

```
%union{
   int valent;
   char *cadena;
   tblPos *ptr;
}
```

Ejemplo

Directiva %union

```
%union{
 struct{
  char tipo;
  int valor;
  float f_valor;
            } num
```

Ahora asignamos la estructura **num** a dos no terminales (N y E):

```
%type <num> N E
.
.
```

Ahora lo utilizamos en las reglas:

```
E ( E + E { if $1.tipo = $3.tipo
       then
  if $1.tipo = ENTERO
   $$.valor = $1.valor + $3.valor
   $$.tipo = ENTERO
    }
```

Precedencia y asociatividad

¿Cómo se especifica la asociatividad por la izquierda o por la derecha?

¿Cómo se especifica la precedencia?

Los declarados en líneas posteriores tienen más precedencia. Por ejemplo:

% left '+' '-'

% left '*' '/'

El último declarado es el que tiene más precedencia, en este caso la operación de multiplicar y la de división.

Un ejemplo de gramática con expresiones de prioridad:

```
%token NUMERO, MAS, POR, '(',  ')'
%left MAS
%left POR
%start S /* simbolo axioma sentencial */

%%
S:  expr {printf("resultado:  %d\n",  $$) ; }
expr:    |expr MAS expr
             |expr POR expr
             |'(' expr ')'
             |  NUMERO
%%
```

Gestión de errores

Las estrategias de recuperación de errores que conocemos son las siguientes:

- Modo de pánico/alarma.
- Nivel de frase.
- Producciones de error.
- Corrección global.

Podemos implementar una estrategia de producciones de error incorporando producciones con el error para que emitan un mensaje:

- E→ E op T | E->T
- E→ E T //**falta operador**
- T→ id | num

Inconvenientes

La estrategia que implementa Bison es la de "modo pánico" contextualizada a una producción. Para que detecte los errores se tiene que implementar una serie de directivas y variables globales definidas por el propio Bison.

Para que los errores se informen de una forma mas completa, en la sección de Declaraciones se incluye:

#define YYERROR_VERBOSE 1

Función de los errores

También se puede ampliar para incluir el número de línea, el lexema del error, etc.

En detalle

Inconvenientes

Esto presenta algunos inconvenientes:

- Es dificil ir más allá de los casos conocidos.
- Si se realizan acciones, aparte de la emisión del mensaje se pueden introducir ambigüedades.

En detalle

Función de los errores

La función que informa de los errores tiene que implementarse y puede hacerse de la siguiente manera:

```
yyerror (s)
char *s;
{
fprintf (stderr, "%s\n", s);
}
```

Incorporación de acciones semánticas

En una pantalla anterior hemos aprendido a asociar atributos a los símbolos de la gramática, que posteriormente podremos utilizar para almacenar información de contexto (semántica) que a su vez nos permite hacer comprobaciones, es decir, introducción de acciones que nos permiten evaluar estos atributos.

Estas reglas se denominan reglas semánticas y cada atributo representa una propiedad de un elemento del lenguaje, como por ejemplo: X.Tipo, X.Valor, etc.

Solo hay que recordar que el valor semántico de cada símbolo está disponible al realizar cada reducción.

¿Cómo se implementa esto en Bison?

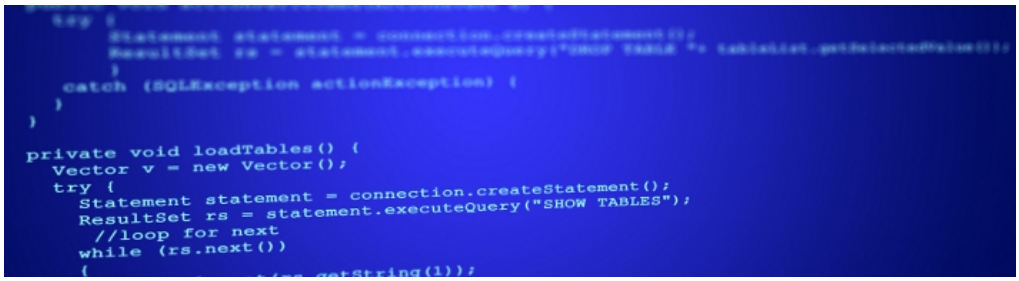
Se puede hacer mediante acciones embebidas, aunque es más sencillo ponerlas al final de las reglas, ya que si no, se pueden introducir conflictos en la resolución de la gramática:

X: Y {acción} Z;

Por ejemplo:

X: Y {$$:= 2*$1;} z {$$:= $2+$3;}

Establece el valor de x como 2*(valor de Y) + (valor de z)



Resumen

En este tema hemos aprendido a utilizar una herramienta denominada **Bison, cuya finalidad es generar analizadores sintácticos,** que en unión con el generador de analizadores léxicos nos va a permitir desarrollar las dos primeras fases de un compilador de una forma rápida y que permite una fácil modificación.

Por un lado, se han visto las **distintas partes que componen un fichero Bison** y la finalidad de las distintas directivas que trae el generador.

Por otro lado, se ha visto **cómo se comunica Flex con Bison,** lo que permite que le pase los componentes léxicos que el primero ha reconocido y cómo se implementan directivas para especificar la precedencia y asociatividad.

Además se ha visto **cómo se pueden gestionar los errores, mediante el denominado "modo pánico"** y/o con producciones de error.

Por último, se ha visto **cómo se pueden introducir acciones semánticas** que nos facilitarán la realización de determinadas comprobaciones o controles semánticos.

En cualquier caso es muy recomendable revisar el manual de Bison equivalente a la versión que esté instalada en el entorno de desarrollo.