

Generador de analizadores sintácticos BISON

PROCESADORES DE LENGUAJES
4º Informática

<http://ccia.ei.uvigo.es/docencia/PL>

23 de enero de 2008

1. Introducción

- Traduce la especificación de una gramática de contexto libre a un programa en C que implementa un analizador LALR(1) para esa gramática.
- Permite asociar código C (*acciones semánticas*) a las reglas de la gramática
 - se ejecutará cada vez que se aplique la regla correspondiente
- Esquema de funcionamiento:

```
fichero.y ---> BISON ---> fichero.tab.c
fichero.tab.c + (ficheros .c) ---> GCC ---> ejecutable
|
|-- main()
|-- yyerror()
|-- yylex()
```

Compilación:

```
$ bison fichero.y
```

Compila la especificación de la gramática y crea el fichero `fichero.tab.c` con el código y las tablas del analizador LALR(1)

```
$ gcc fichero.tab.c (ficheros .c)
```

El usuario debe proporcionar las funciones **main()**, **yyerror()** y **yylex()**. El código de usuario deberá llamar a la función **yyvsparse()**, desde la cual se llamará a la función **yylex()** del analizador léxico cada vez que necesite un `TOKEN`.

Opciones:

bison -d 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.

bison -v genera "`fichero.output`" con un resumen legible del autómata LALR(1) y señala los conflictos y/o errores presentes en la gramática de entrada

2. Funcionamiento del analizador

- El fichero "XXXX.tab.c" contiene las tablas del analizador y la función **int yyparse(void)**
 - **yyparse()** simula el analizador LALR(1)
 - devolverá 0 si el análisis tubo éxito y 1 si el análisis fallo
 - deberá de ser llamada desde el código del usuario
- Cada vez que **yyparse()** necesite un nuevo TOKEN, llamará a la función **int yylex()**
 - **yylex()** devuelve un n° entero que identifica al siguiente TOKEN
 - esas constantes enteras aparecen en el fichero "XXXX.tab.h"
 - **yylex()** devolverá el token EOF (*end of file*) cuando alcance el final del fichero
- Para comunicar los atributos de los TOKENS se usa la variable global **yylval**
 - es de tipo YYSTYPE y está declarada en "XXXX.tab.h"
 - **yylex()** escribe en **yylval** los valores que después usará **yyparse()**
- La función **yyparse()** realiza un análisis ascendente salto-reducción
 - agrupa TOKENS y no terminales según a las reglas de la gramática
 - utiliza una pila donde acumula símbolos
 - cuando en la cima de la pila se localiza el lado derecho de una regla, se eliminarán de la pila y se meterá en ella el no terminal del lado derecho de la regla. (REDUCCIÓN)
- Por regla general el código asociado a una regla (*acciones semánticas*) se ejecutará en el momento en que se reduzca la regla
 - es posible incluir código en mitad de la parte derecha de las reglas
 - a cada elemento de la pila se le asocia asociada una variable de tipo YYSTYPE
 - el código de las acciones semánticas puede acceder a esas variables usando las pseudo-variables \$\$, \$1, \$2, \$3, ...

3. Especificación de gramáticas en BISON

3 partes separadas por símbolo "%%" (2 primeras obligatorias, pueden ir vacías)

<sección de declaraciones>

%%

<sección de reglas y acciones>

%%

<sección de rutinas de usuario>

(a) Sección de Declaraciones.

- Código C necesario para las acciones semánticas.
 - Irá entre los símbolos "%{" y "%}" (se copia tal cual en "XXXX.tab.c")
 - Generalmente serán **#includes** y/o estructuras y variables
- Definiciones de BISON.
 1. Especificación de YYSTYPE (tipo de los valores semánticos)
 - tipo de datos asociado a los elementos de la pila (tokens y no terminales)
 - se usa la directiva **%union**

XXXX.Y

XXXX.TAB.H / XXXX.TAB.C

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

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

2. Especificación de TOKENS y de sus propiedades (asociatividad, precedencia)
 - directiva %token: indica el nombre de un TOKEN y opcionalmente su *tipo* (será uno de los identificadores declarados en **%union**)

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

- No es necesario declarar TOKENS compuestos por un único caracter
- Cada caracter se identifica por el valor numérico de su código ASCII
- Los demás TOKENS tienen constantes numéricas empezando en 256

- **directivas %left, %right, %nonasoc**: especifica un TOKEN con su asociatividad (izquierda, derecha o no asociativo)
Determina como actuará el analizador sintáctico (normalmente son operadores)

```
%left  '-' '+' '*' '/'
%right '^' '='
```

3. Especificación de *tipo* de los no terminales

- No es necesario declarar previamente los no terminales (son los que aparecen en lado izquierdo)
- **directiva %type**: especifica el *tipo* de un no terminal
Sólo para los no terminales que tengan asociados valores semánticos

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

4. Otros:

- **directiva %start no_terminal**: identifica al axioma de la gramática
Por defecto bison usa como axioma al no terminal del lado izquierdo de la primera regla.

(b) Sección de reglas

1. Formato de las reglas:

```
no_terminal : componente1 componenete2 ... componenteN
;
```

Varias reglas con el mismo lado izquierdo pueden abreviarse:

```
no_terminal : lado_derecho_1
             | lado_derecho_2
             | lado_derecho_3
;
```

La reglas- ϵ tienen su lado derecho vacío

2. Acciones semánticas:

- Contienen código C que se ejecutará cada vez que se reconozca una instancia de la regla asociada.
- **Formato**: sentencias C incluidas entre llaves ("{" y "}")
- Suelen ir al final de la regla y se ejecutan cuando se reconoce completamente su parte derecha
- También se admiten en otras posiciones dentro de las reglas, que se ejecutarán en cuanto se reconozca esa fracción de la regla

3. Pseudo-variables:

- Las pseudo-variables `$$`, `$1`, `$2`, ... permiten acceder a los valores semánticos asociados a los símbolos de la regla.
`$$`: valor semántico asociado al no terminal del lado izquierdo de la regla
`$1`, `$2`, ..., `$N`: valores semánticos asociados a los símbolos (`TOKENS` y no terminales) del lado derecho
- El *tipo* de esas pseudo-variables será el que fue asignado al símbolo correspondiente en la sección de declaraciones (directivas **`%token`**, **`%type`**, **`%left`**, ..)
- Si no hay acción, BISON añade la *acción por defecto* `{$$=$1;}` (cuando concuerden los tipos)

(c) Sección de rutinas de usuario

Esta sección zona se puede escribir código C adicional

- Suelen ser funciones llamadas desde las acciones de las reglas
- En programas pequeños, suelen incluirse las funciones **`main()`**, **`yyerror()`** y/o **`yylex()`** que aporta el usuario

4. 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 **`int yylex()`**
 - cada vez que sea llamada identificará el siguiente símbolo terminal en la entrada y devolverá la constante entera que lo identifica
 - esas constantes enteras están definidas en el fichero de cabecera `"XXXX.tab.h"` (generado con la opción **`-d`**)
 - declaraciones **`#define`** que asocian nombre de `TOKEN` con su valor numérico
- Con los caracteres simples no es necesario definir un específico, dado que ya están identificados por su código ASCII.
 - el analizador léxico simplemente deberá devolver su valor ASCII.
 - no hay conflicto con otros `TOKENS` (BISON les asigna valores enteros > 256)
- Para `TOKENS` con atributos se usa la var. global **`yylval`** (de tipo `YYSTYPE`)
 - definido mediante la directiva **`%union`**.
 - la declaración de **`yylval`** e `YYSTYPE` está en `"XXXX.tab.h"`

Integración con FLEX

Pasos para utilizar los analizadores léxicos generados con FLEX:

1. Generar el fichero `"XXXX.tab.h"` con **`bison -d`**
2. Incluirllo en la sección de declaraciones C de la especificación FLEX

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

3. En la acción asociada al patrón de cada `TOKEN`:
 - a) (*opcional*) cargar en el campo que corresponda de la variable **`yylval`** los valores de los atributos léxicos que han sido reconocidos (serán utilizados en las acciones de bison)
 - b) (*obligatorio*) devolver el tipo del `TOKEN` encontrado mediante una orden **`return TOKEN`**

```
%%  
[0-9]+ { yylval.entero = atoi(yytext[0]);  
        return CONSTANTE_ENTERA; }  
"+"    { return yytext[0]; }  
...  
%%
```

Nota: Para los caracteres simples, basta devolver su valor en ASCII.

5. Resolución de conflictos

Tipos de conflictos (surgen cuando la gramática es ambigua)

desplazamiento-reducción: el analizador se encuentra con que puede desplazar un nuevo TOKEN a la pila o reducir una regla con el contenido actual de la pila

reducción-reducción: se reconoce una parte derecha de dos reglas distintas (habrá 2 posibles reglas a reducir)

Nota: suele deberse a problemas "graves" en el diseño de la gramática

Por defecto BISON informa del error mediante un *warning* y resuelve el conflicto de la siguiente manera:

- desplazamiento-reducción : Elige el desplazamiento
- reducción-reducción : Elige reducir la regla que aparezca primero en la definición de la gramática

Otras soluciones :

- rediseño de la gramática
- uso del mecanismo de precedencias (sólo para desplazamiento-reducción)

Mecanismo de precedencias

- A cada TOKEN se le puede asociar una precedencia y una asociatividad.
- A las reglas se les asocia la precedencia del último TOKEN de su parte derecha.
- La asociatividad de un TOKEN se especifica con las directivas **%left**, **%right** y **%nonassoc**
- La precedencia de los TOKENS se determina por el orden en que fueron declarados. (menor precedencia para los TOKENS declarados primero)

```
%left '+' '-'  
%left '*' '/'  
%nonassoc MENOS_UNITARIO  
%right '^'
```

PRECEDENCIA : " + - " < " * / " < MENOS_UNITARIO < " ^ "

Resolución conflictos desplazamiento-reducción

- conflicto entre TOKEN y regla

SI la precedencia no está definida ---> por defecto, desplazar

SI token MAYOR PRECEDENCIA que regla ---> desplazar

SI regla MAYOR PRECEDENCIA que {\sc token} ---> reducir

SI IGUAL PRECEDENCIA (Mirar ASOCIATIVIDAD)

SI token asociativo por IZQUIERDA ---> reducir

SI token asociativo por DERECHA ---> desplazar

SI token no asociativo ---> generar WARNING y, por defecto, desplazar

Nota: Se puede especificar la precedencia de una regla empleando el modificador **%prec**

- Se añade despues de los componentes de la regla con el siguiente formato
%prec nombre_token
- Asocia a la regla la precedencia del TOKEN indicado.

ejemplo.y

```
/* Codigo C*/
%{
#include "diccionario.h" /* definiciones de insertar_tabla(), buscar_tabla()*/
DICCIONARIO *diccionario;
%}

/* Declaraciones de BISON */
%union {
    int    valor_entero;
    double valor_real;
    char * texto;
}

%token <valor_real> CONSTANTE_REAL
%token <valor_entero> CONSTANTE_ENTERA
%token <texto> IDENTIFICADOR

%left '-' '+'
%left '*' '/'

%type <valor_real> expresion

%%
/* Gramatica */
input: /* cadena vacia */
    | input linea
    ;
linea: '\n'
    | IDENTIFICADOR '=' expresion '\n' { insertar_tabla(diccionario, $1, $3); }
    | expresion '\n' { printf ("\t%d\n", $1); }
    ;
expresion: CONSTANTE_REAL { $$ = $1; }
    | CONSTANTE_ENTERA { $$ = (double) $1; }
    | IDENTIFICADOR { $$ = buscar_tabla(diccionario,$1); }
    | expresion '+' expresion { $$ = $1 + $3; }
    | expresion '-' expresion { $$ = $1 - $3; }
    | expresion '*' expresion { $$ = $1 * $3; }
    | expresion '/' expresion { $$ = $1 / $3; }
    ;
%%
int main() {
    inicializar(diccionario);
    yyparse();
}

yyerror (char *s) { printf ("%s\n", s); }
int yywrap() { return 1; }
```

ejemplo.l

```
%{
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include "ejemplo.tab.h"
%}

NUM [0-9]+
%%

{NUM}      {
            yylval.valor_entero = atoi(yytext);
            return (CONSTANTE_ENTERA);
        }

{NUM}\.{NUM} {
            yylval.valor_real = atof(yytext);
            return (CONSTANTE_REAL);
        }

"+"|"-"|"*"|"/" {
            return (yytext[0]);
        }

"\n"      {
            return (yytext[0]);
        }

[A-Za-z]* {
            yylval.texto = (char *) malloc (strlen(yytext) + 1);
            strcpy(yytext, yylval.texto);
            return (IDENTIFICADOR);
        }

.         ;
%%
```

Compilación

```
$ bison -d ejemplo.y
$ flex ejemplo.l
$ gcc -o ejemplo ejemplo.tab.c lex.yy.c diccionario.c
```