Generador de analizadores sintácticos BISON

TEORÍA DE AUTÓMATAS Y LENGUAJES FORMALES
3º Grado de Informática

Marzo-2021

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.
  - analizador determinista ascendente de salto-reducción.
- Permite asociar código C (*acciones semánticas*) a las reglas de la gramática
  - se ejecutará cada vez que se reconozca 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 **yyparse()**, desde la cual se llamará a la función **yylex()** del analizador léxico cada vez que necesite una categoría léxica (*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 tuvo éxito y 1 si el análisis falló
  - 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$^{\text{o}}$ 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
  - reconoce *tokens* y no terminales según las reglas de la gramática
  - utiliza una pila donde acumula los símbolos reconocidos
  - 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 símbolo de la regla se le asocia una variable de tipo **YYSTYPE** (atributo semántico)
  - 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.
  - Se escribe entre los símbolos "%{" y "%}" (y se copia en "XXXX.tab.c")
  - Generalmente serán **#include**, **#define**, tipos de datos o variables globales
- Especificación de **YYSTYPE** (tipo de los valores semánticos)
  - tipo de datos asociado a los símbolos de la gramática
  - se especifica mediante la directiva **%union**

```
XXXX.Y                          XXXX.TAB.H / XXXX.TAB.C

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

- 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
```

    Bison asocia a cada *token* un código numérico (mediante **enum**)
    Excepción: no es necesario declarar *tokens* compuestos por un único carácter
    - Se pueden usar en las reglas de la gramática escribiendo dicho carácter entre comillas simples.
    - Su código numérico será el código ASCII del carácter
    - Bison asigna al resto de *tokens* valores enteros a partir de 256

- directivas **%left, %right, %nonassoc**: especifica *tokens* (operadores) con una asociatividad determinada (repectivamente, izquierda, derecha, sin asociatividad)

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

- 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
```

- Otros:
  - directiva **%start no_terminal**: identifica al axioma de la gramática
    Si no se especifica, 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_n
;
```

   La reglas-ε tienen su lado derecho vacio

2. Acciones semánticas:
   - Contienen código C que generalmente se ejecutará cada vez que se reconozca una la regla asociada (los símbolos de su parte derecha).
   - **Formato:** 1 o más instrucciones C delimitadas por llaves (`{ }`)
   - 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 reconozcan los símbolos a la izquierda de la acción
3. Pseudo-variables:
   - Las pseudo-variables **$$, $1, $2, ...** permiten acceder a los atributos semánticos a asociados a los símbolos de la regla.
     $$: corresponde al símbolo cabeza de la regla
     $1, $2, ..., $N: corresponden a los símbolos de la parte derecha

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

   - 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* `{$$=$1;}` (cuando concuerden los tipos)
   - No hay acción por defecto en el caso de reglas-$\boldsymbol{\varepsilon}$

(c) Sección de rutinas de usuario

En esta sección 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**)
  - definidas en forma de tipo enumerado (**enum**)
- Con los terminales de un solo carácter no es necesario definir un nombre de *token* específico, dado que se pueden identificar por su código ASCII.
  - el analizador léxico simplemente deberá devolver su valor ASCII.
  - no hay conflicto con otros *tokens* (BISON les asigna valores entreros > 256)
- Para *tokens* con atributos semánticos se usa la var. global **yylval** (tipo **YYSTYPE**)
  - definido mediante la directiva **%union**.
  - la declaración de **yylval** e **YYSTYPE** se encuentran 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. Incluirlo en la sección de declaraciones C de la expecificación FLEX

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

3. En la acción asociada al patrón de cada *token*:
   - *(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)
   - *(obligatorio)* devolver el código del *token* encontrado mediante una orden **return NOMBRE_TOKEN** o **return yytext[0]** (para *tokens* de un solo carácter)

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

5. Resolución de conflictos

Tipos de conflictos (surgen cuando la gramática es ambigua)

salto-reducción: el analizador se encuentra con que puede saltar 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:

- *salto-reducción* : Elige el salto
- *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 salto-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 salto-reducción

- conflicto entre *token* y regla

```
SI la precedencia no está definida ---.> por defecto, saltar
SI token  MAYOR PRECEDENCIA que regla ---> saltar
SI regla MAYOR PRECEDENCIA que token ---> reducir
SI IGUAL PRECEDENCIA   (Mirar ASOCIATIVIDAD)
         SI token  asociativo por IZQUIERDA ---> reducir
         SI token  asociativo por DERECHA   ---> saltar
         SI token  no asociativo     --->  generar WARNING y, por defecto, saltar
```

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`

```
expresion : '-' expresion  %prec MENOS_UNITARIO
```

- Asocia a la regla la precedencia del *token* indicado.

6. Tratamiento de errores

Cuando hay un error sintáctico el analizador generado por BISON llama a la función **yyerror(char *s)**, que debe proporcionar el usuario (en su versión más simple es un **printf** del mensaje de error en la cadena **s**).

Por defecto, el analizador parará el análisis despues de detectar el error y de llamar a **yyerror()**, y la función **yyparse()** devolverá 1, indicando que la entrada no pertenece al lenguaje generado por la gramática.

En general esta forma de tratar los errores no es aceptable.

- Es deseable que despues de encontarse con un error, el analizador siga analizando el resto de la entrada para detectar los demás errores que pueden existir.

Simbolo especial 'error'

- Mecanismo para tratar y recuperar los errores en Bison
- Funciona como un *"token ficticio"* reservado para el manejo de errores.
- Siempre que se produce un error de sintaxis, el analizador de Bison genera el símbolo '**error**', que puede ser manejado en la reglas como un *token* normal.
- Si existe una regla que permita reconocer el símbolo '**error**' en el contexto actual, el analizador podrá recuperarse del error y continuar analizando la entrada
  - Se incluirá la macro **yyerrok** en la acción de esa regla para indicar que el error se ha recuperado

Ejemplo :

`instrucción : error ';' {yyerrok;}` $\rightarrow$ Acepta que durante el análisis de una instr. haya errores, pero sólo hasta leer el **';'** (indicando que la zona erronea ha terminado).

Funcionamiento del símbolo 'error'

- Si en el estado en el que se detecta el error es posible reconocer el símbolo '**error**' (por formar parte de una regla de la gramática), el analizador saltará sobre dicho símbolo y tratará de continuar el análisis.
- En el caso de que no se pueda reconocer el símbolo '**error**', el analizador comenzará a retirar símbolos de la cima de la pila hasta que:
  - encuentre un estado que permita el reconocimiento de '**error**', en cuyo caso se salta sobre él y se intenta proseguir el análisis de la entrada, desechando los símbolos retirados de la pila
  - o se vacíe la pila, con lo que **yyparse()** termina su ejecución devolviendo 1

Tres **modos de funcionamiento** del analizador:

Normal no hay errores y se avanza en el análisis usando las reglas

Sincronización despues de encontarse con un error, el analizador comienza a retirar símbolos de la pila buscando un estado que admita el símbolo '**error**'.

Recuperación (modo pánico) cuando se encuentra un estado que permita el reconocimiento de '**error**', se entra en *modo recuperación*, aceptando *tokens* a la espera de que la entrada vuelva a ser *"correcta"*

- Permanece en *modo pánico* hasta que { se lean 3 *tokens* consecutivos sin encontrar nuevos errores ó se llame a la macro **yyerrok** al ejecutar alguna acción
- Si se encuentra un nuevo error durante el *modo pánico* no se muestran el mensaje de error correspondiente (no se llama **yyerror(char *s)**) para evitar una cascada de mensajes de error hacia el usuario.
- Se sale del *modo pánico* cuando se considera que la entrada vuelve a ser *"analizable"* (el analizador puede seguir sin encontrar nuevos errores)

7. Depuración de gramáticas

Bison tiene un mecanismo para sacar por la consola información sobre el proceso de análisis con la macro `YYDEBUG`, en dos pasos:

1. Declarar dicha macro en la sección de declaraciones de Bison.

```
%{

  #include <stdio.h>
  extern FILE *yyin;
  extern int yylex();

  #define YYDEBUG 1

%}
```

2. Dar a la variable `yydebug` un valor distinto a 0 en alguna parte del código C del analizador, por ejemplo en el programa principal.

```
int main(int argc, char *argv[]) {

  yydebug = 1;

  ...

  yyparse();

  ...

  }
```

El analizador sintáctico listará por la consola, a medida que realiza el análisis:

- los tokens que recibe de `yylex()`
- las reglas que va reduciendo
- los estados por los que va transitando
- el contenido de la pila del analizador

Con esa información podeis ir al fichero '`simple.output`' para ver en qué parte del autómata está el fallo, y hacer las correcciones correspondientes en la gramática.

8. Implementación de gramáticas

Generalmente vamos a tener que representar tres tipos de estructuras típicas de los lenguajes de programación:

- Símbolos opcionales. Dos posibilidades:

$$ab?c$$

$$\begin{array}{ll} S \to ac & S \to aAc \\ S \to abc & A \to b \\ & A \to \varepsilon \end{array}$$

- Simbolos que se repiten, cero o más veces, o una o más veces, mediante reglas recursivas.

$$\begin{array}{ll} b^+ & b^* \\ & \\ S \to Sb & S \to Sb \\ S \to b & S \to \varepsilon \end{array}$$

- Símbolos que se repiten, cero o más veces, o una o más veces, separados por un delimitador.

$$\begin{array}{ll} (a,)^*a & (a,)^*a? \\ & \\ S \to S, a & S \to A \\ S \to a & S \to \varepsilon \\ & A \to A, a \\ & A \to a \end{array}$$

calculadora.y

```
%{  /* Código C */
#include <stdio.h>
#include "diccionario.h"
DICCIONARIO diccionario; /* variable global para el 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 */
lineas:  /* cadena vacia */
      |  lineas linea
;
linea: '\n'
     | IDENTIFICADOR '=' expresion '\n' {insertar_diccionario(&diccionario, $1, $3);}
     | expresion '\n'                   { printf ("resultado: %f\n", $1); }
     | error '\n'                       { yyerrok;}
;
expresion: CONSTANTE_REAL   { $$ = $1; }
         | CONSTANTE_ENTERA { $$ = (double) $1; }
         | IDENTIFICADOR    { ENTRADA * entrada = buscar_diccionario(&diccionario,$1);
                              if (entrada != NULL) { /* encontrada */
                                 $$ = entrada->valor;
                              }
                              else {
                                 printf("ERROR: variable %s no definida\n", $1);
                                 $$ = 0;
                              }
                            }
         | expresion '+' expresion   { $$ = $1 + $3; }
         | expresion '-' expresion   { $$ = $1 - $3; }
         | expresion '*' expresion   { $$ = $1 * $3; }
         | expresion '/' expresion   { $$ = $1 / $3; }
;

%%
```

```
int main(int argc, char** argv) {
    inicializar_diccionario(&diccionario);
    yyparse();
    liberar_diccionario(&diccionario);
}

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

calculadora.l

```
%{ /* Codigo C */
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include "calculadora.tab.h"
%}
DIGITO [0-9]
LETRA  [A-Za-z]

%%
{DIGITO}+             { yylval.valor_entero = atoi(yytext);
                        return (CONSTANTE_ENTERA);
                      }

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

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

{LETRA}({LETRA}|_)*   { yylval.texto = (char *) malloc (strlen(yytext) + 1);
                        strcpy(yylval.texto, yytext);
                        return (IDENTIFICADOR);
                      }

.                     ;
%%
```

Compilación

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