

Trabajo de Laboratorio de CL-1

2. El Bison

1 Introducción

El Bison es un programa que toma como entrada una especificación sintáctica (el *fuelle* Bison *nombrebison.y*). Este *fuelle* Bison contiene, entre otras cosas, una gramática de contexto libre con acciones C asociadas a las reglas gramaticales. A partir del *fuelle*, el Bison genera un programa C con la función `yyparse()` que realiza el parsing ascendente de la secuencia de tokens de la entrada. Estos tokens los va obteniendo del analizador léxico mediante llamadas a la función `yylex()` generada por el programa Flex (véase “Trabajo de Laboratorio de CL-1, 1. El Flex”).

A continuación damos un formato simplificado de un fichero de entrada de Bison, junto con el C generado:

Entrada Bison:	Fichero C generado
=====	=====
%{	
Declaraciones C	-----> Declaraciones C // se copian tal cual
%}	
/* Declaracion de tokens */	\ // Cada token sera una constante entera
%token t1 t2	#define t1 1
%token t3 t4 t5 ...	-> #define t2 2
...	/ #define t3 3
	...
	\
/* Simbolo no-terminal inicial	int yyparse() {
de la gramatica: */	/* Esta funcion presupone
%start nt1	la existencia de una funcion
%%	'int yylex()' que devuelve
/* Reglas gramaticales	valores definidos por los tokens
de una gramatica con reglas	-> t1, t2, t3, ...
nti -> sj sk sl	
donde los s son simbs. terminales	Llama a yylex sucesivamente
(tokens) o no-terminales */	obteniendo asi una secuencia
	de tokens, y verifica si la
	gramatica genera dicha secuencia */
nt1 : s1 s2 s3	...
nt2 : s4 s5	}
...	/
%%	
Rutinas C	-----> Rutinas C // se copian tal cual
	/* Estas rutinas pueden contener
	una funcion yylex o incluirla desde
	algún fichero */

2 Flex + Bison

En el “Trabajo de Laboratorio de CL-1, 1.” vimos usos del Flex en los que la función `yylex()` no acababa hasta que la entrada llegaba a final de fichero. Pero también existen casos en los que nos interesa que la función `yylex()` devuelva un resultado *cada vez que reconoce un prefijo de la entrada*. Cuando es llamada de nuevo, la función devolverá un resultado para el siguiente prefijo, y así sucesivamente. En combinación con el Bison, el resultado devuelto

por `yylex()` se llama un *token*, una constante de C que nos dice a qué expresión regular corresponde el prefijo reconocido. Así podemos tener el siguiente fragmento de Flex:

```

":="                {return(ASIG);}
"+"                {return(MAS);}
If                  {return(IF);}
Then                {return(THEN);}
EndIf               {return(ENDIF);}
{LETRA}({LETRA}|{DIGITO})* {return(IDENT);}
{DIGITO}+           {return(INTCONST);}

```

donde ASIG, MAS, etc., son constantes C que identifican a los distintos tokens. En el fuente Bison podemos expresar reglas gramaticales en las que los símbolos terminales son tokens que hemos declarado previamente. Normalmente se escriben los tokens en mayúsculas, y los símbolos no-terminales en minúsculas. Un fragmento incompleto de fuente Bison es:

```

%{
// Declaraciones C
%}
% token MAS MENOS IF THEN ELSE IDENT INTCONST // declaración de tokens

% start lista_instrucciones // símbolo inicial de la gramática

%%
                                // ahora vienen las reglas gramaticales:
lista_instrucciones : lista_instrucciones instruccion | instruccion ;
instruccion:          IF expresion THEN lista_instrucciones ENDIF |
                      IDENT ASIG expresion ;
expresion:            IDENT |
                      INTCONST |
                      expresion MAS expresion;

%%
// Más declaraciones C:
#include "lexico.c" // Incluimos la funcion yylex() generada por el Flex.
int yyerror(char *error) // Función que será llamada por yyparse()
{printf(error);} // en caso de error de parsing.
main(){ yyparse(); } // main llama a yyparse(), que es quien va llamando
                      // a yylex() cada vez que necesita un nuevo token.

```

Cuando el Bison se ejecuta sobre este fuente, el C generado tiene la siguiente estructura:

```

Declaraciones C           // Las declaraciones C del principio se copian tal cual

#define ASIG      1 // declaración de los tokens como constantes C
#define MAS      2
#define IF       3
#define THEN     4
#define ENDIF    5
#define IDENT    6
#define INTCONST 7

int yylex(void); // cabecera de yylex(), para que yyparse la pueda llamar

function yyparse(){ // funcion yyparse() generada por el Bison a partir
... } // de las reglas gramaticales

Declaraciones C           // Este bloque de decl. C también se copia tal cual:

function yylex(){ // se incluye el yylex() aqui y no arriba, para que
... } // conozca los #define's de los tokens!

```

```
int yyerror(char *error) // Función que será llamada por yyparse()
{printf(error);}        // en caso de error de parsing.
main(){ yyparse(); }
```

Prioridades entre tokens

Muchas situaciones de ambigüedad de la gramática se pueden resolver utilizando prioridades entre tokens. Esto es preferible a tener que modificar la gramática, de modo que continúe siendo sencilla y legible.

Podemos declarar cada token como **left**, **right** o **nonassoc**, que indican asociatividad izquierda, derecha, o ninguna. Básicamente, si al parser le falta información para decidir si realizar un *shift* o un *reduce* por una regla gramatical, mira si el token que aparece más a la derecha en la regla tiene asociada una asociatividad. Una declaración como **nonassoc** puede ser útil para operadores como **=** o **>**, para que cosas como $a = b = c$ se consideren erróneas sintácticamente. Además, los tokens declarados así en líneas anteriores tienen prioridad más baja.

En el ejemplo de gramáticas de expresiones podríamos hacer:

```
%token MAS MENOS PROD DIV
%left MAS MENOS
%left PROD DIV
```

Para compilar haremos `bison nombrebison.y -o nombrebison.c`, con lo que se genera `nombrebison.c`.

Ejercicios para pensar en casa y/o laboratorio

1. Escribe un programa Flex+Bison que detecte si la entrada es una secuencia de paréntesis bien parentizada. Recuerda que una gramática para ese lenguaje es $S \rightarrow (S)S \mid \lambda$.
2. Amplía la gramática anterior para secuencias con paréntesis, corchetes y llaves, todos ellos bien imbrincados.
3. Escribe un programa Flex+Bison que compruebe si la entrada es una expresión correcta sobre enteros, operaciones de suma, multiplicación y resta, y paréntesis.
4. Implementa con Flex+Bison un programa que compruebe la corrección sintáctica de programas muy sencillos (por ejemplo, solo con if-then-else-endif y asignación). Amplíalo añadiendo otras construcciones.