

3. Usar Bison para reconocer y computar

1 Utilizar Bison para reconocer y computar

En líneas generales, la función `yyparse` obtiene de `yylex` una secuencia de tokens, y verifica que dicha secuencia se pueda generar con una gramática. Para ello, va leyendo la secuencia y comprobando si alguna subsecuencia de lo leído hasta el momento `sj sk sl` coincide con alguna parte derecha de regla `nti: sj sk sl`. De ser así, `yyparse` puede optar por reducir (reemplazar) `sj sk sl` a `nti` y continuar.

Durante el proceso de reconocimiento de la entrada, se puede ir realizando un cierto cómputo. Justo antes de reducir por una regla, se pueden ejecutar instrucciones asociadas a esta regla y trasladar la información calculada para cómputos posteriores. Para ello, se permite al usuario añadir código C en las reglas gramaticales del siguiente modo: `nti: sj sk sl {Código C}`. En éste código, se puede acceder a valores ya calculados previamente cuando se obtuvieron `sj`, `sk` y `sl` mediante las variables abstractas `$1`, `$2` y `$3`, respectivamente. A dichos valores se les llama atributos. La variable abstracta `$$` corresponde al atributo que guardará `nti` tras hacer la reducción.

En el siguiente ejemplo, vemos cómo la gramática de expresiones de sumas de enteros nos sirve, no sólo para verificar que la entrada es una expresión bien construida, sino también para calcular el valor de dicha expresión.

```
%start expresion_total
%%
expresion_total: expresion {printf("suma: %d", $1);}
expresion: expresion MAS expresion { $$=$1+$3; } |
          INTCONST { $$=$1; }
```

En el caso de la producción `expresion: INTCONST`, el valor calculado en `$1` no proviene de una reducción anterior, pues `INTCONST` es un token, sino que se obtiene de una variable global definida por el Bison llamada `yylval`. Aquí se asume que la función `yylex` habrá modificado esta variable asignándole el correspondiente valor entero; es decir, se supone que en el fichero flex tenemos una regla del tipo:

```
{DIGITO}+      { yyval.valor=atoi(yytext); return INTCONST; }
```

Cuando se vuelve de una llamada a `yylex`, `yyparse` asume que la variable `yylval` ha sido modificada, y asigna su contenido sobre la variable abstracta correspondiente al token reconocido.

Para ello es necesario definir desde el fichero Bison los posibles valores que puede adquirir `yylval`. Con este fin, se declara un conjunto de pares (`tipo, identificador`) dentro de un ámbito `%union`, el cual se acabará traduciendo a una `union` de C. Veamos el siguiente ejemplo.

```
%union{
    int valor;
    char texto_identificador[20];
}
```

La declaración anterior, tras ejecutar Bison sobre el fichero que la contenga, quedará convertida en la siguiente.

```
typedef union {
    int valor;
    char texto_identificador[20];
} YYSTYPE;

YYSTYPE yylval;
```

2 Estructura detallada de un fichero Bison

El formato de la entrada para el Bison es:

```
%{
  declaraciones para el compilador de C
}%
Declaraciones de Bison
%%
Reglas
%%
Subrutinas del usuario
```

La sección de declaraciones para el compilador de C tiene un uso similar a la del Flex.

2.1 La sección de declaraciones Bison

En la sección de declaraciones Bison se indican, entre otras cosas, los tipos de los atributos, los tokens, las prioridades y el no-terminal inicial de la gramática. Por ejemplo, consideremos:

```
%union {
  atrib_ident ai;
  int          numlin;
  arbol        ar;
}
%token PROGRAM ENDPROGRAM VARS ENDVARS
%token IF THEN ELSE END IF WHILE ENDWHILE
%token MAS MENOS PROD DIV
%token <ai> IDENT
%type <ar> programa
%type <ar> dec_vars
%left MAS MENOS
%left PROD DIV
%...
%start programa
```

- Aquí la **union** de C indica los distintos *tipos* que puede tener cada atributo asociado a un token o no-terminal. (Nota para los familiarizados con gramáticas de atributos: solamente consideraremos atributos *sinetizados* y no los *heredados*). Cada campo de la **union** es dado como **<tipo> <nombre tipo>**; donde el **<tipo>** ha sido declarado (o incluido) en la sección de declaraciones para el compilador de C. Recordemos: la **union** de C es un **record** variable, es decir, se reserva el espacio correspondiente al **<tipo>** que más ocupe, (pero no el espacio de la suma de los tamaños de todos los tipos, como en una **struct**).
- A continuación se han declarado varios tokens, algunos de ellos con el tipo del atributo asociado. Por ejemplo, al token **IDENT** se le ha asociado el tipo **atrib_ident**. Este tipo podría ser una **struct** con un campo **string** con el string del identificador y un campo **lin** con el número de línea. Los atributos de los tokens se asignan en el Flex, por ejemplo así:

```
{LETRA}({LETRA}|{DIGITO})* { strcpy(yylval.ai.string,yytext);
                               yylval.ai.lin = numlinea;
                               return(IDENT); }
```

- Después figuran declaraciones como `%type <ar> programa`, que indican el tipo del atributo asociado a cada no-terminal (en este caso, `programa`). Aquí, `arbol` podría ser el tipo del árbol sintáctico (seguramente, un puntero a un nodo, donde cada nodo tiene, al menos, punteros a los hijos, además de información sobre el no-terminal o token del nodo).
- A continuación hay varias declaraciones sobre prioridades entre tokens. 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, como es habitual, hemos dado mayor prioridad al producto y cociente (sin prioridad entre sí) que a la suma y resta (que también tienen igual prioridad).
- Finalmente, `%start programa` indica que `programa` es el no-terminal inicial de la gramática.

2.2 La sección de reglas

En el apartado de reglas, escribiremos las reglas gramaticales en el siguiente formato:

```
programa: PROGRAM dec_vars l_instrucciones ENDPROGRAM
        { printf("programa");
          $$=crear_nodo_2hijos(PROG,$2,$3); }
;
dec_vars: VARS lista_decs ENDVARS
        { printf("decl. vars no-vacia");
          $$=crear_nodo_1hijo(DECV,$2); }
        | { printf("decl. vars vacia"); ... }
;
```

donde la parte derecha de la segunda regla gramatical de `dec_vars` es λ (es decir, es vacía). Cada regla gramatical debe tener una acción C asociada (aunque sea la acción vacía `{ ; }`), que se escribe al final de su parte derecha. (También es posible insertar las acciones en otros puntos, no al final, de la parte derecha, pero aquí no consideraremos esa posibilidad.) En estas acciones C podemos asignar cosas a `$$`, que denota el atributo del no-terminal a la izquierda de la regla. Asimismo, `$1`, `$2`, etc., denotan el atributo del primer, segundo, etc. símbolo (token o no-terminal) de la parte derecha de la regla. De esta manera, la instrucción `$$=crear_nodo_2hijos(PROG,$2,$3)` podría ser una llamada a una función (creada por nosotros), que genera un árbol sintáctico cuya raíz es un nodo etiquetado con la constante `PROG`, y que tiene como hijos los árboles (ya generados) de `dec_vars` y `l_instrucciones` respectivamente.

En el apartado de subrutinas de usuario, en primer lugar incluiremos el C generado por el Flex: `#include "nombrelex.c"`. Además tendremos nuestras funciones C auxiliares. Si queremos utilizar la función `yyparse()` independientemente (y no, como es también habitual, desde otro programa, como un compilador), también incluiremos en este apartado un `main` que llame a la función `yyparse()`. Asimismo, incluiremos una función `int yyerror(char *)` que es llamada en caso de error sintáctico (de momento, nos basta con un `yyerror` que escriba siempre un mismo mensaje de error).

3 Estructura general para combinar Flex y Bison

Entrada Bison:		Fichero C generado
=====		=====
%{		
Declaraciones C		Declaraciones C /* se copian tal cual */
%}		
/* Definiciones Bison */		
%union {	\	typedef union {
/* Declaración de campos estilo C */		/* se copia tal cual: */
tipo1 identificador1;	---->	tipo1 identificador1;
tipo2 identificador2;		tipo2 identificador2;
...		...
}	/	} YYSTYPE;
		/* se declara la variable
		global yylval */
		YYSTYPE yylval;
/* Declaración de tokens */		/* Cada token será una constante entera */
%token <identificador1> t1 t2		#define t1 num1
%token <identificador2> t3 t4 t5 ...		#define t2 num2
...		#define t3 num3
		...
		/* Cabecera de yylex(), para que
		/* yyparse la pueda llamar.
		int yylex(void);
/* Resolucion de prioridades */		
%left t3 t5		
%left t2		
...		
/* Declaración de no-terminales */		
%type <identificador1> nt1 nt2		
%type <identificador2> nt3 nt4 ...		
...		
/* Simbolo inicial de la gramatica */		
%start nt1		
%%		
/* Reglas gramaticales */	\	int yyparse() {
/* proviene de una gramática con reglas		/* Función que llama a yylex
nti -> sj sk sl, donde los s son		obteniendo así una secuencia
tokens o no-terminales	*/---->	de tokens y verifica si la
nt1 : s1 s2 s3 {Codigo C }		gramática genera dicha secuencia.
nt2 : s4 s5 {Codigo C }		Durante el proceso se ejecutará
...	/	el Codigo C correspondiente a
%%		una parte derecha reconocida */
#include "ficheroflex.c"	----->	#include "ficheroflex.c"
Rutinas C	----->	Rutinas C /* se copian tal cual */
		/* Estas rutinas pueden contener una
		funcion main que llame a yyparse, y
		la función yyerror que será llamada
		por yyparse en caso de error sintáctico */

Supongamos que la función `yyparse` trata de reconocer la parte derecha de la regla “`nt1: s1 t2 s3`”, que ya ha reconocido ‘`s1`’ y que al llamar a `yylex` esta le devuelve ‘`t2`’. En este momento, se asigna el valor `yylval.identificador1` (que se supone modificado por `yylex`) sobre ‘`$2`’ (por ser ‘`t2`’ el segundo símbolo de la lista “`s1 t2 s3`”), que es una variable abstracta de tipo ‘`tipol`’, donde en ‘`%union`’ se declaró el campo “`tipol identificador1`” y el token ‘`t2`’ se declaró de tipo ‘`identificador1`’.

Supongamos que finalmente se reconoce “`s1 t2 s3`” reduciéndose así a ‘`nt1`’, y supongamos también que se intentaba reconocer ‘`nt1`’ debido a que estábamos reconociendo a su vez “`nt5: t1 t6 nt1 t4`” y ya habíamos obtenido “`t1 t6`”. En este momento, el contenido de la variable abstracta ‘`$$`’ de la producción “`nt1: s1 t2 s3`” se copia sobre la variable abstracta ‘`$3`’ de la producción “`nt5: t1 t6 nt1 t4`”.

Desde el flex, en el código C asociado a cada expresión regular, se asume la existencia de una variable global `yylval`, y un conjunto de constantes `t1`, `t2`, ... que llamamos tokens.

Entrada Flex:	Fichero C generado
=====	=====
<code>%{</code>	
<code>Declaraciones C</code>	<code>-----> Declaraciones C /* se copian tal cual */</code>
<code>%}</code>	
<code>Definiciones</code>	<code>\ int yylex() {</code>
<code>%option noyywrap</code>	<code> // Función que lee la entrada,</code>
<code>%%</code>	<code> // y cada vez que encuentra</code>
<code>Reglas del tipo:</code>	<code> // un prefijo que está en el</code>
<code>ExpReg { asignaciones sobre yylval; -></code>	<code> // lenguaje de una ExpReg, ejecuta</code>
<code>return token;} /</code>	<code> // las ‘asignaciones sobre yylval’</code>
<code>%%</code>	<code> // y devuelve el control al ejecutar</code>
	<code>/ // ‘return token;’</code>
	<code>...</code>
	<code>}</code>
<code>Rutinas C</code>	<code>-----> Rutinas C /* se copian tal cual */</code>

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 implemente una calculadora que lea expresiones (con constantes enteras, +, *, etc.) y escriba los resultados.
2. Implementa con Flex+Bison un programa que genere y escriba (mas o menos legiblemente) los arboles sintácticos para programas muy sencillos (por ejemplo, solo con if-then-else-endif y asignación). Para ello, se recomienda utilizar las funciones del fichero `ast.c` que se encuentra en la página web de la asignatura.