Introducción a Flex y Bison

Prácticas de *Lenguajes, Gramáticas y Autómatas,* cuarto cuatrimestre (primavera) de Ingeniería en Informática

http://webdiis.unizar.es/asignaturas/LGA

Profesor Responsable: Rubén Béjar Hernández
Dpto. Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Introducción a Flex

Flex es un una herramienta que permite generar analizadores léxicos. A partir de un conjunto de expresiones regulares, Flex busca concordancias en un fichero de entrada y ejecuta acciones asociadas a estas expresiones. Es compatible casi al 100% con Lex, una herramienta clásica de Unix para la generación de analizadores léxicos, pero es un desarrollo diferente realizado por GNU bajo licencia GPL.

Los ficheros de entrada de Flex (normalmente con la extensión .l) siguen el siguiente esquema:

```
%%
patrón1 {acción1}
patrón2 {acción2}
...
donde:
```

patrón: expresión regular

acción: código C con las acciones a ejecutar cuando se encuentre concordancia del patrón con el texto de entrada

Flex recorre la entrada hasta que encuentra una concordancia y ejecuta el código asociado. El texto que no concuerda con ningún patrón lo copia tal cual a la salida. Por ejemplo:

```
%%
a*b     {printf(“X”);};
re      ;
```

El ejecutable correspondiente transforma el texto:

```
abre la puertaab
```

en

```
X la puertX
```

pues ha escrito X cada vez que ha encontrado ab o aab y nada cuando ha encontrado re.

`Flex` lee los ficheros de entrada dados, o la entrada estándar si no se le ha indicado ningún nombre de fichero, con la descripción de un escáner a generar. La descripción se encuentra en forma de parejas de expresiones regulares y código C, denominadas **reglas**. `Flex` genera como salida un fichero fuente en C, `‘lex.yy.c’`, que define una función `‘yylex()’`. Este fichero se compila y se enlaza con la librería de Flex para producir un ejecutable. Cuando se arranca el fichero ejecutable, este analiza su entrada en busca de casos de las expresiones regulares. Siempre que encuentra uno, ejecuta el código C correspondiente.

El fichero de entrada de `Flex` está compuesto de tres secciones, separadas por una línea donde aparece únicamente un `‘%%’` en esta:

```
definiciones
%%
reglas
%%
código de usuario
```

La sección de **definiciones** contiene declaraciones de definiciones de **nombres** sencillas para simplificar la especificación del escáner, y declaraciones de **condiciones** de arranque, que se explicarán en una sección posterior. Las definiciones de nombre tienen la forma:

```
nombre definición
```

El "nombre" es una palabra que comienza con una letra o un subrayado (`'_'`) seguido por cero o más letras, dígitos, `'_'`, o `'-'` (guión). La definición se considera que comienza en el primer carácter que no sea un espacio en blanco siguiendo al nombre y continuando hasta el final de la línea. Posteriormente se puede hacer referencia a la definición utilizando "{nombre}", que se expandirá a "(definición)". Por ejemplo,

```
DIGITO   [0-9]
ID       [a-z][a-z0-9]*
```

define "DIGITO" como una expresión regular que empareja un dígito sencillo, e "ID" como una expresión regular que empareja una letra seguida por cero o más letras o dígitos. Una referencia posterior a

```
{DIGITO}+"."{DIGITO}*
```

es idéntica a

```
([0-9])+"."([0-9])*
```

y empareja uno o más dígitos seguido por un `'.'` seguido por cero o más dígitos.

La sección de *reglas* en la entrada de `Flex` contiene una serie de reglas de la forma:

```
patrón   acción
```

donde el patrón debe estar sin sangrar y la acción debe comenzar en la misma línea.

Finalmente, la sección de código de usuario simplemente se copia a `'lex.yy.c'` literalmente. Esta sección se utiliza para rutinas de complemento que llaman al escáner o son llamadas por este. La presencia de esta sección es opcional; Si se omite, el segundo `'%%'` en el fichero de entrada se podría omitir también.

Patrones

Los patrones en la entrada se escriben utilizando un conjunto extendido de expresiones regulares y usando como alfabeto cualquier carácter ASCII. Cualquier símbolo excepto el espacio en blanco, tabulador, cambio de línea y los caracteres especiales se escriben tal cual en las expresiones regulares (patrones) de Flex. Los caracteres especiales son:

“ \ [^ - ? . * + | ( ) $ / { } % < >

Algunos de los patrones de Flex son:

`x`
: empareja el carácter `'x'`

`.`
: cualquier carácter excepto una línea nueva

`[xyz]`
: un conjunto de caracteres; en este caso, el patrón empareja una `'x'`, una `'y'`, o una `'z'`

`[abj-oZ]`
: un conjunto de caracteres con un rango; empareja una `'a'`, una `'b'`, cualquier letra desde la `'j'` hasta la `'o'`, o una `'Z'`

`[^A-Z]`

cualquier carácter menos los que aparecen en el conjunto. En este caso, cualquier carácter EXCEPTO una letra mayúscula.

`[^A-Z\n]`
: cualquier carácter EXCEPTO una letra mayúscula o una línea nueva

r`*`
: cero o más *r*'s, donde *r* es cualquier expresión regular

r`+`
: una o más *r*'s

r`?`
: cero o una *r* (es decir, "una *r* opcional")

r`{2,5}`
: entre dos y cinco concatenaciones de *r*

r`{4}`
: exactamente 4 *r*'s

`{nombre}`
: la expansión de la definición de "nombre" (ver más abajo)

`"[xyz]\"foo"`
: la cadena literal: [xyz]"foo

`\`*x*
: si *x* es una `'a'`, `'b'`, `'f'`, `'n'`, `'r'`, `'t'`, o `'v'`, entonces la interpretación ANSI-C de *x* (por ejemplo \t sería un tabulador). En otro caso, un literal `'`*x*`'` (usado para la concordancia exacta de caracteres especiales (\\ \. \?)

`(`*r*`)`
: empareja una *R*; los paréntesis se utilizan para anular la precedencia (ver más abajo)

rs
: la expresión regular *r* seguida por la expresión regular *s*; se denomina "concatenación"

r`|`*s*
: bien una *r* o una *s*

r`/`*s*
: una *r* pero sólo si va seguida por una *s*.

`^`*r*
: una *r*, pero sólo al comienzo de una línea

r`$`
: una *r*, pero sólo al final de una línea (es decir, justo antes de una línea nueva). Equivalente a "*r*/\n".

`<`*s*`>`*r*
: una *r*, pero sólo en la condición de arranque *s* (ver más adelante).

`<`*s1*`,`*s2*`,`*s3*`>`*r*
: lo mismo, pero en cualquiera de las condiciones de arranque *s1*, *s2*, o *s3*

Emparejamiento de la entrada

Cuando el escáner generado está funcionando, este analiza su entrada buscando cadenas que concuerden con cualquiera de sus patrones. Si encuentra más de un emparejamiento, toma el que empareje el texto más largo. Si encuentra dos o más emparejamientos de la misma longitud, se escoge la regla listada en primer lugar en el fichero de entrada de `Flex`.

Una vez que se determina el emparejamiento, el texto correspondiente al emparejamiento (denominado el *token*) está disponible en el puntero de carácter global `yytext`, y su longitud en la variable global entera `yyleng`. Entonces la *acción* correspondiente al patrón emparejado se ejecuta y luego la entrada restante se analiza para otro emparejamiento.

Si no se encuentra un emparejamiento, entonces se ejecuta la **regla por defecto**: el siguiente carácter en la entrada se considera reconocido y se copia a la salida estándar.

Un ejemplo clarificador:

```
%%
aa      {printf(“1”);}
aab     {printf(“2”);}
uv      {printf(“3”);}
xu      {printf(“4”);}
```

Con el texto de entrada `Laabdgf xuv,` daría como salida `L2dgf 4v`. El ejecutable copiará L, reconocerá `aab` (porque es más largo que `aa`) y realizará el printf(“2”), copiará `dgf ` , reconocerá `xu` (porque aunque tienen la misma longitud que `uv`, y `uv` está antes en el fuente, en el momento de reconocer la `x` flex no reconoce el posible conflicto, puesto que sólo `xu` puede emparejarse con algo que empieza por `x` (el manual de flex no es muy claro al respecto, pero parece que usa un analizador de izquierda a derecha) y ejecutará el printf(“4”) y luego copiará la `v` que falta).

Acciones

Cada patrón en una regla tiene una acción asociada, que puede ser cualquier código en C. El patrón finaliza en el primer carácter de espacio en blanco que no sea una secuencia de escape; lo que queda de la línea es su acción. Si la acción está vacía, entonces cuando el patrón se empareje el token de entrada simplemente se descarta. Por ejemplo, aquí está la especificación de un programa que borra todas las apariciones de "zap me" en su entrada:

```
%%
"zap me"
```

(Este copiará el resto de caracteres de la entrada a la salida ya que serán emparejados por la regla por defecto.)

Aquí hay un programa que comprime varios espacios en blanco y tabuladores a un solo espacio en blanco, y desecha los espacios que se encuentren al final de una línea:

```
%%
[ \t]+        putchar( ‘ ‘ );
[ \t]+$       /* ignora este token */
```

Si la acción contiene un `‘{‘`, entonces la acción abarca hasta que se encuentre el correspondiente `‘}’`, y la acción podría entonces cruzar varias líneas. `Flex` es capaz de reconocer las cadenas y comentarios de C y no se dejará engañar por las llaves que encuentre dentro de estos, pero aun así también permite que las acciones comiencen con `‘%{‘` y considerará que la acción es todo el texto hasta el siguiente `‘%}’` (sin tener en cuenta las llaves ordinarias dentro de la acción).

Las acciones pueden incluir código C arbitrario, incluyendo sentencias `return` para devolver un valor desde cualquier función llamada `‘yylex()’`. Cada vez que se llama a `‘yylex()’` esta continúa procesando tokens desde donde lo dejó la última vez hasta que o bien llegue al final del fichero o ejecute un return.

También se pueden llamar a funciones definidas por el usuario en la parte de código de usuario del fuente de Flex. Por ejemplo:

```
%{
  int y = 0;
%}
%%
```

```
aa      {subrutina(y); y++; };
%%
subrutina(d)
int d;
{
  int j = 5;
  if (d==j) {
    printf(“aa”);
  }
}
```

Hay unas cuantas directivas especiales que pueden incluirse dentro de una acción:

- `ECHO` copia yytext a la salida del escáner.
- `BEGIN` seguido del nombre de la condición de arranque pone al escáner en la condición de arranque correspondiente.

El analizador generado

La salida de `Flex` es el fichero `‘lex.yy.c’`, que contiene la función de análisis `‘yylex()’`, varias tablas usadas por esta para emparejar tokens, y unas cuantas rutinas auxiliares y macros. Por defecto, `‘yylex()’` se declara así

```
int yylex()
    {
    ... aquí van varias definiciones y las acciones ...
    }
```

Siempre que se llame a `‘yylex()’`, este analiza tokens desde el fichero de entrada global `yyin` (que por defecto es igual a stdin). La función continúa hasta que alcance el final del fichero (punto en el que devuelve el valor 0) o una de sus acciones ejecute una sentencia `return`.

Condiciones de arranque (sensibilidad al contexto)

`Flex` dispone de un mecanismo para activar reglas condicionalmente. Cualquier regla cuyo patrón se prefije con "<sc>" únicamente estará activa cuando el analizador se encuentre en la condición de arranque llamada "sc". Por ejemplo,

```
<STRING>[^"]*        { /* se come el cuerpo de la cadena ... */
            ...
            }
```

estará activa solamente cuando el analizador esté en la condición de arranque "STRING", y

```
<INITIAL,STRING,QUOTE>\. { /* trata una secuencia de escape ... */
            ...
            }
```

estará activa solamente cuando la condición de arranque actual sea o bien "INITIAL", "STRING", o "QUOTE".

Las condiciones de arranque se declaran en la (primera) sección de definiciones de la entrada usando líneas sin sangrar comenzando con `%S` seguida por una lista de nombres. Una condición de arranque se activa utilizando la acción `BEGIN`. Hasta que se ejecute la próxima acción `BEGIN`, las reglas con la condición de arranque dada estarán activas y las

reglas con otras condiciones de arranque estarán inactivas. Las reglas sin condiciones de arranque también estarán activas.

`BEGIN(0)` retorna al estado original donde solo las reglas sin condiciones de arranque están activas. Este estado también puede referirse a la condición de arranque INITIAL, así que `BEGIN(INITIAL)` es equivalente a `BEGIN(0)`. (No se requieren los paréntesis alrededor del nombre de la condición de arranque pero se considera de buen estilo.)

Algunas variables disponibles para el usuario

Esta sección resume algunos de los diferentes valores disponibles al usuario en las acciones de las reglas.

- `char *yytext` apunta al texto del token actual (última palabra reconocida en algún patrón). Por ejemplo printf("%s", yytext) lo escribiría por pantalla.
- `int yyleng` contiene la longitud del token actual.

Interacción con Bison

Uno de los usos principales de Flex es como acompañante del analizador de gramáticas Bison (o de Yacc). Los analizadores Bison necesitan una función llamda 'yylex()' para devolverles el siguiente token de la entrada. Esa función devuelve el tipo del próximo token y además puede poner cualquier valor asociado en la variable global yylval. Para usar Flex con Bison, normalmente se especifica la opción –d de Bison para que genera el fichero 'y.tab.h' que contiene las definiciones de todos los '%tokens' que aparecen el fuente Bison. Este fichero de cabecera se incluye después en el fuente de Flex. Por ejemplo, si uno de los tokens es "TOK_NUMBER", parte del fichero Flex podría ser:

```
%{
#include "y.tab.h"
%}
%%
[0-9]+        yylval = atoi( yytext ); return TOK_NUMBER;
```

Compilación y ejecución de un programa Flex

Al ejecutar el comando f`lex nombre_fichero_fuente` se creará un fichero en C llamado "`lex.yy.c`". Si se compila este fichero con la instrucción "`gcc lex.yy.c –lfl –o nombre_ejecutable`" (`-lfl` indica enlazar con la biblioteca de flex) se obtendrá como resultado un fichero ejecutable llamado `nombre_ejecutable`).

Una vez se ha creado el fichero ejecutable se puede ejecutar directamente. Flex esperará que se introduzca texto por la entrada estándar (teclado) y lo analizará cada vez que se pulse el retorno de carro. Para terminar con el análisis normalmente hay que pulsar `<CTRL>-D`. Para poder probarlo con entradas más largas, lo más sencillo es crear archivos de texto y usar redirecciones para que el ejecutable lea estos ficheros como entrada a analizar. Por ejemplo si hemos creado un analizador llamado `prueba1` y un fichero de texto con datos para su análisis, llamado `entrada.txt`, podemos ejecutar $`prueba1 < entrada.txt`.

Si se quiere ejecutar algún código al final de un análisis de Flex (para mostrar resultados por ejemplo) hay al menos dos opciones:

```
%%
reglas...
```

```
%%
main() {
  yylex();
  código a ejecutar al final del análisis
}

o bien

%%
reglas...
%%
yywrap() {
  código a ejecutar al final del análisis
  return 1;
}
```

Notas finales

Flex requiere un formato bastante estricto de su fichero de entrada. En particular los caracteres no visibles (espacios en blanco, tabuladores, saltos de línea) fuera de sitio causan errores difíciles de encontrar. Sobre todo es muy importante no dejar líneas en blanco de más ni empezar reglas con espacios en blanco o tabuladores.

Introducción a Bison

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción para una gramática independiente del contexto (en realidad de una subclase de éstas, las LALR) en un programa en C que analiza esa gramática. Es compatible al 100% con Yacc, una herramienta clásica de Unix para la generación de analizadores léxicos, pero es un desarrollo diferente realizado por GNU bajo licencia GPL. Todas la gramáticas escritas apropiadamente para Yacc deberían funcionar con Bison sin ningún cambio. Usándolo junto a Flex esta herramienta permite construir compiladores de lenguajes.

Un fuente de Bison (normalmente un fichero con extensión .y) describe una gramática. El ejecutable que se genera indica si un fichero de entrada dado pertenece o no al lenguaje generado por esa gramática. La forma general de una gramática de Bison es la siguiente:

```
%{
declaraciones en C
%}

Declaraciones de Bison

%%
Reglas gramaticales
%%
Código C adicional
```

Los `'%%'`, `'%{'` y `'%}'` son signos de puntuación que aparecen en todo archivo de gramática de Bison para separar las secciones.

Las declaraciones en C pueden definir tipos y variables utilizadas en las acciones. Puede también usar comandos del preprocesador para definir macros que se utilicen ahí, y utilizar `#include` para incluir archivos de cabecera que realicen cualquiera de estas cosas.

Las declaraciones de Bison declaran los nombres de los símbolos terminales y no terminales, y también podrían describir la precedencia de operadores y los tipos de datos de los valores semánticos de varios símbolos.

Las reglas gramaticales son las **producciones** de la gramática, que además pueden llevar asociadas acciones, código en C, que se ejecutan cuando el analizador encuentra las reglas correspondientes.

El código C adicional puede contener cualquier código C que desee utilizar. A menudo suele ir la definición del analizador léxico `yylex`, más subrutinas invocadas por las acciones en las reglas gramaticales. En un programa simple, todo el resto del programa puede ir aquí.

Símbolos, terminales y no terminales

Los **símbolos terminales** de la gramática se denominan en Bison **tokens** y deben declararse en la sección de definiciones. Por convención se suelen escribir los tokens en mayúsculas y los **símbolos no terminales** en minúsculas.

Los nombres de los símbolos pueden contener letras, dígitos (no al principio), subrayados y puntos. Los puntos tienen sentido únicamente en no-terminales.

Hay tres maneras de escribir símbolos terminales en la gramática. Aquí se describen las dos más usuales:

- Un **token declarado** se escribe con un identificador, de la misma manera que un identificador en C. Por convención, debería estar todo en mayúsculas. Cada uno de estos nombres debe definirse con una declaración de `%token`.
- Un **token de carácter** se escribe en la gramática utilizando la misma sintaxis usada en C para las constantes de un carácter; por ejemplo, `'+'` es un tipo de token de carácter. Un tipo de token de carácter no necesita ser declarado a menos que necesite especificar el tipo de datos de su valor semántico, asociatividad, o precedencia. Por convención, un token de carácter se utiliza únicamente para representar un token consistente en ese carácter en particular.

Sintaxis de las reglas gramaticales (producciones)

Una regla gramatical de Bison tiene la siguiente forma general:

```
resultado: componentes...
        ;
```

donde *resultado* es el símbolo no terminal que describe esta regla y *componentes* son los diversos símbolos terminales y no terminales que están reunidos por esta regla. Por ejemplo,

```
exp:      exp '+' exp
        ;
```

dice que dos agrupaciones de tipo `exp`, con un token `'+'` en medio, puede combinarse en una agrupación mayor de tipo `exp`.

Los espacios en blanco en las reglas son significativos únicamente para separar símbolos. Puede añadir tantos espacios en blanco extra como desee.

Distribuidas en medio de los componentes pueden haber *acciones* que determinan la semántica de la regla. Una acción tiene el siguiente aspecto:

```
{sentencias en C}
```

Normalmente hay una única acción que sigue a los componentes.

Se pueden escribir por separado varias reglas para el mismo *resultado* o pueden unirse con el caracter de barra vertical `'|'` así:

```
resultado:    componentes-regla1...
        | componentes-regla2...
        ...
        ;
```

Estas aún se consideran reglas distintas incluso cuando se unen de esa manera. Si los *componentes* en una regla están vacíos, significa que *resultado* puede concordar con la cadena vacía (en notación formal sería ε). Por ejemplo, aquí aparece cómo definir una secuencia separada por comas de cero o más agrupaciones `exp`:

```
expseq:   /* vacío */
        | expseq1
        ;
```

```
expseq1:  exp
        | expseq1 ‘,’ exp
        ;
```

Es habitual escribir el comentario `‘/* vacío */’` en cada regla sin componentes.

Una regla se dice **recursiva** cuando su no-terminal *resultado* aparezca también en su lado derecho. Casi todas las gramáticas de Bison hacen uso de la recursión, ya que es la única manera de definir una secuencia de cualquier número de cosas. Considere esta definición recursiva de una secuencia de una o más expresiones:

```
expseq1:  exp
        | expseq1 ‘,’ exp
        ;
```

Puesto que en el uso recursivo de `expseq1` este es el símbolo situado más a la izquierda del lado derecho, llamaremos a esto **recursión por la izquierda**. Por contraste, aquí se define la misma construcción utilizando **recursión por la derecha**:

```
expseq1:  exp
        | exp ‘,’ expseq1
        ;
```

Cualquier tipo de secuencia se puede definir utilizando ya sea la recursión por la izquierda o recursión por la derecha, pero debería utilizar siempre recursión por la izquierda, porque puede analizar una secuencia de elementos sin ocupar espacio de pila (es decir, de forma mucho más eficiente en memoria). La recursión **indirecta** o **mutua** sucede cuando el resultado de la regla no aparece directamente en su lado derecho, pero aparece en las reglas de otros no terminales que aparecen en su lado derecho.

Por ejemplo:

```
expr:     primario
        | primario ‘+’ primario
        ;

primario:  constante
        | ‘(‘ expr ‘)’
        ;
```

define dos no-terminales recursivos mutuamente, ya que cada uno hace referencia al otro.

Semántica del lenguaje

Las reglas gramaticales para un lenguaje determinan únicamente la sintaxis. La semántica viene determinada por los valores semánticos asociados con varios tokens y agrupaciones, y por las acciones tomadas cuando varias agrupaciones son reconocidas.

En un programa sencillo podría ser suficiente con utilizar el mismo tipo de datos para los valores semánticos de todas las construcciones del lenguaje. Por defecto Bison utiliza el tipo `int` para todos los valores semánticos. Para especificar algún otro tipo, defina `YYSTYPE` como una macro, de esta manera:

```
#define YYSTYPE double
```

Esta definición de la macro debe ir en la sección de declaraciones en C del fichero de la gramática.

En la mayoría de los programas, necesitará diferentes tipos de datos para diferentes clases de tokens y agrupaciones. Por ejemplo, una constante numérica podría necesitar el tipo `int` o `long`, mientras que una cadena constante necesita el tipo `char *`.

Para utilizar más de un tipo de datos para los valores semánticos en un analizador, Bison requiere dos cosas:

- Especificar la colección completa de tipos de datos posibles, con la declaración de Bison `%union`.
- Elegir uno de estos tipos para cada símbolo (terminal o no terminal). Esto se hace para los tokens con la declaración de Bison `%token` y para los no terminales con la declaración de Bison `%type`.

Acciones

Una acción acompaña a una regla sintáctica y contiene código C a ser ejecutado cada vez que se reconoce una instancia de esa regla. La tarea de la mayoría de las acciones es computar el valor semántico para la agrupación construida por la regla a partir de los valores semánticos asociados a los tokens o agrupaciones más pequeñas.

Una acción consiste en sentencias de C rodeadas por llaves, muy parecido a las sentencias compuestas en C. Se pueden situar en cualquier posición dentro de la regla; la acción se ejecuta en esa posición.

El código C en una acción puede hacer referencia a los valores semánticos de los componentes reconocidos por la regla con la construcción `$`*n*, que hace referencia al valor de la componente *n*-ésima. El valor semántico para la agrupación que se está construyendo es `$$`. Aquí hay un ejemplo típico:

```
exp:    ...
        | exp ‘+’ exp
            { $$ = $1 + $3; }
```

Esta regla construye una `exp` de dos agrupaciones `exp` más pequeñas conectadas por un token de signo más. En la acción, `$1` y `$3` hacen referencia a los valores semánticos de las dos agrupaciones `exp` componentes, que son el primer y tercer símbolo en el lado derecho de la regla. La suma se almacena en `$$` de manera que se convierte en el valor semántico de la expresión de adición reconocida por la regla. Si hubiese un valor semántico útil asociado con el token `'+'`, debería hacerse referencia con `$2`.

Si no especifica una acción para una regla, Bison suministra una por defecto: `$$ = $1`. De este modo, el valor del primer símbolo en la regla se convierte en el valor de la regla entera. Por supuesto, la regla por defecto solo es válida si concuerdan los dos tipos de datos. No hay una regla por defecto con significado para la regla vacía; toda regla vacía debe tener una acción explícita a menos que el valor de la regla no importe.

Tipos de Datos de Valores en Acciones

Si ha elegido un tipo de datos único para los valores semánticos, las construcciones `$$` y `$`*n* siempre tienen ese tipo de datos.

Si ha utilizado `%union` para especificar una variedad de tipos de datos, entonces debe declarar la elección de entre esos tipos para cada símbolo terminal y no terminal que

puede tener un valor semántico. Entonces cada vez que utilice $$ o $*n*, su tipo de datos se determina por el símbolo al que hace referencia en la regla. En este ejemplo,

```
exp:    ...
        | exp '+' exp
            { $$ = $1 + $3; }
```

$1 y $3 hacen referencia a instancias de exp, de manera que todos ellos tienen el tipo de datos declarado para el símbolo no terminal exp. Si se utilizase $2, tendría el tipo de datos declarado para el símbolo terminal '+', cualquiera que pudiese ser.

De forma alternativa, puede especificar el tipo de datos cuando se hace referencia al valor, insertando '<*tipo*>' después del '$' al comienzo de la referencia. Por ejemplo, si ha definido los tipos como se muestra aquí:

```
%union {
  int tipoi;
  double tipod;
}
```

entonces puede escribir $<tipoi>1 para hacer referencia a la primera subunidad de la regla como un entero, o $<tipod>1 para referirse a este como un double.

Acciones a Media Regla

Ocasionalmente es de utilidad poner una acción en medio de una regla. Estas acciones se escriben como las acciones al final de la regla, pero se ejecutan antes de que el analizador llegue a reconocer los componentes que siguen.

Una acción en mitad de una regla puede hacer referencia a los componentes que la preceden utilizando $*n*, pero no puede hacer referencia a los componentes subsecuentes porque esta se ejecuta antes de que sean analizados.

Las acciones en mitad de una regla por sí mismas cuentan como uno de los componentes de la regla. Esto produce una diferencia cuando hay otra acción más tarde en la misma regla (y normalmente hay otra al final): debe contar las acciones junto con los símbolos cuando quiera saber qué número *n* debe utilizar en $*n*. Por ejemplo:

```
      $1      $2   $3           $4             $5  $6

a :  token1 token2 b { acción a media regla; } c token3

  ;
```

No hay forma de establecer el valor de toda la regla con una acción en medio de la regla, porque las asignaciones a $$ no tienen ese efecto. La única forma de establecer el valor para toda la regla es con una acción corriente al final de la regla.

Declaraciones en Bison

La sección de **declaraciones de Bison** de una gramática de Bison define los símbolos utilizados en la formulación de la gramática y los tipos de datos de los valores semánticos. Todos los nombres de tokens (pero no los tokens de carácter literal simple tal como '+' y '*') se deben declarar. Los símbolos no terminales deben ser declarados si necesita especificar el tipo de dato a utilizar para los valores semánticos.

La primera regla en el fichero también especifica el símbolo inicial, por defecto. Si desea que otro símbolo sea el símbolo de arranque, lo debe declarar explícitamente.

Nombres de Token

La forma básica de declarar un nombre de token (símbolo terminal) es como sigue:

`%token` *`nombre1 nombre2 nombre3...`*

De forma alternativa, puede utilizar `%left`, `%right`, o `%nonassoc` en lugar de `%token`, si desea especificar la precedencia.

En el caso de que el tipo de la pila sea una union, debe aumentar `%token` u otra declaración de tokens para incluir la opción de tipo de datos delimitado por ángulos.

Por ejemplo:

```
%union {              /* define el tipo de la pila */
  double val;
  int intVal;
}

%token <val> NUM       /* define el token NUM y su tipo */
```

Puede asociar un token de cadena literal con un nombre de tipo de token escribiendo la cadena literal al final de la declaración `%type` que declare el nombre. Por ejemplo:

Precedencia de Operadores

Use las declaraciones `%left`, `%right` o `%nonassoc` para declarar un token y especificar su precedencia y asociatividad, todo a la vez. Estas se llaman **declaraciones de precedencia**.

La sintaxis de una declaración de precedencia es la misma que la de `%token`: bien

`%left` *`símbolos...`*

o

`%left <`*`tipo`*`>` *`símbolos...`*

Y realmente cualquiera de estas declaraciones sirve para los mismos propósitos que `%token`. Pero además, estos especifican la asociatividad y precedencia relativa para todos los *símbolos*:

- La asociatividad de un operador *op* determina cómo se anidan los usos de un operador repetido: si *`'x op y op z'`* se analiza agrupando *x* con *y* primero o agrupando *y* con *z* primero. `%left` especifica asociatividad por la izquierda (agrupando *x* con *y* primero) y `%right` especifica asociatividad por la derecha (agrupando *y* con *z* primero). `%nonassoc` especifica no asociatividad, que significa que *`'x op y op z'`* se considera como un error de sintaxis.

- La precedencia de un operador determina cómo se anida con otros operadores. Todos los tokens declarados en una sola declaración de precedencia tienen la misma precedencia y se anidan conjuntamente de acuerdo a su asociatividad. Cuando dos tokens declarados asocian declaraciones de diferente precedencia, la última en ser declarada tiene la mayor precedencia y es agrupada en primer lugar.

La Colección de Tipos de Valores

La declaración `%union` especifica la colección completa de posibles tipos de datos para los valores semánticos. La palabra clave `%union` viene seguida de un par de llaves conteniendo lo mismo que va dentro de una `union` en C.

Por ejemplo:

```
%union {
  double val;
  int valInt;
}
```

Esto dice que los dos tipos de alternativas son `double` y `int`. Se les ha dado los nombres `val` y `valInt`; estos nombres se utilizan en las declaraciones de `%token` y `%type` para tomar uno de estos tipos para un símbolo terminal o no terminal.

Note que, a diferencia de hacer una declaración de una `union` en C, no se escribe un punto y coma después de la llave que cierra.

Símbolos No Terminales

Cuando utilice `%union` para especificar varios tipos de valores, debe declarar el tipo de valor de cada símbolo no terminal para los valores que se utilicen. Esto se hace con una declaración `%type`, como esta:

`%type <`*`tipo`*`> `*`noterminal`*`...`

Aquí *noterminal* es el nombre de un símbolo no terminal, y *tipo* es el nombre dado en la `%union` a la alternativa que desee. Puede dar cualquier número de símbolos no terminales en la misma declaración `%type`, si tienen el mismo tipo de valor. Utilice espacios para separar los nombres de los símbolos.

Puede también declarar el tipo de valor de un símbolo terminal. Para hacer esto, utilice la misma construcción <*tipo*> en una declaración para el símbolo terminal. Todos las clases de declaraciones de tipos permiten <*tipo*>.

El Símbolo Inicial

Bison asume por defecto que el símbolo inicial para la gramática es el primer no terminal que se encuentra en la sección de especificación de la gramática. El programador podría anular esta restricción con la declaración `%start` así:

`%start `*`símboloNoTerminal`*

Precedencia de operadores

Considere el siguiente fragmento de gramática ambigua (ambigua porque la entrada `‘1 - 2 * 3’` puede analizarse de dos maneras):

```
expr:     expr ‘-’ expr
        | expr ‘*’ expr
        | ‘(‘ expr ‘)’
        ...
        ;
```

Suponga que el analizador ha visto los tokens `‘1’`, `‘-’` y `‘2’`; ¿debería reducirlos por la regla del operador de adición? Esto depende del próximo token. Por supuesto, si el siguiente token es un `‘)’`, debemos reducir; el desplazamiento no es válido porque ninguna regla puede reducir la secuencia de tokens `‘- 2 )’` o cualquier cosa que comience con eso. Pero si el próximo token es `‘*’` tenemos que elegir: ya sea el desplazamiento o la reducción permitiría al analizador terminar, pero con resultados diferentes (el primer caso haría la resta antes del producto y en el segundo se haría el producto antes de la resta)

Especificando Precedencia de Operadores

Bison le permite especificar estas opciones con las declaraciones de precedencia de operadores `%left` y `%right`. Cada una de tales declaraciones contiene una lista de tokens, que son los operadores cuya precedencia y asociatividad se está declarando. La declaración `%left` hace que todos esos operadores sean asociativos por la izquierda y la declaración `%right` los hace asociativos por la derecha. Una tercera alternativa es `%nonassoc`, que declara que es un error de sintaxis encontrar el mismo operador dos veces "seguidas".

La precedencia relativa de operadores diferentes se controla por el orden en el que son declarados. La primera declaración `%left` o `%right` en el fichero declara los operadores cuya precedencia es la menor, la siguiente de tales declaraciones declara los operadores cuya precedencia es un poco más alta, etc.

En nuestro ejemplo, queríamos las siguientes declaraciones (precedencia típica en operadores matemáticos):

```
%left ‘-’
%left ‘*’
```

En un ejemplo más completo, que permita otros operadores también, los declararíamos en grupos de igual precedencia. Por ejemplo, `‘+’` se declara junto con `‘-’`:

```
%left ‘=‘
%left ‘+’ ‘-’
%left ‘*’ ‘/’
```

Precedencia Dependiente del Contexto

A menudo la precedencia de un operador depende del contexto. Esto suena raro al principio, pero realmente es muy común. Por ejemplo, un signo menos típicamente tiene una precedencia muy alta como operador unario, y una precedencia algo menor (menor que la multiplicación) como operador binario.

Las declaraciones de precedencia de Bison, `%left`, `%right` y `%nonassoc`, puede utilizarse únicamente para un token dado; de manera que un token tiene sólo una precedencia declarada de esta manera. Para la precedencia dependiente del contexto, necesita utilizar un mecanismo adicional: el modificador `%prec` para las reglas.

El modificador `%prec` declara la precedencia de una regla en particular especificando un símbolo terminal cuya precedencia debe utilizarse para esa regla. No es necesario por otro lado que ese símbolo aparezca en la regla. La sintaxis del modificador es:

`%prec` *`símbolo-terminal`*

y se escribe despúes de los componentes de la regla. Su efecto es asignar a la regla la precedencia de *símbolo-terminal*, imponiéndose a la precedencia que se deduciría de forma ordinaria. La precedencia de la regla alterada afecta entonces a cómo se resuelven los conflictos relacionados con esa regla.

Aquí está cómo `%prec` resuelve el problema del menos unario. Primero, declara una precedencia para un símbolo terminal ficticio llamada `UMINUS`. Aquí no hay tokens de este tipo, pero el símbolo sirve para representar su precedencia:

```
...
%left ‘+’ ‘-’
%left ‘*’
%left UMINUS
```

Ahora la precedencia de `UMINUS` se puede utilizar en reglas específicas:

```
exp:    ...
        | exp ‘-’ exp
        ...
        | ‘-’ exp %prec UMINUS
```

Funcionamiento del analizador

El fuente de Bison se convierte en una función en C llamada `yyparse`. Aquí describimos las convenciones de interfaz de `yyparse` y las otras funciones que éste necesita usar.

Tenga en cuenta que el analizador utiliza muchos identificadores en C comenzando con `‘yy’` e `‘YY’` para propósito interno. Si utiliza tales identificadores (a parte de aquellos descritos en el manual) en una acción o en código C adicional en el archivo de la gramática, es probable que se encuentre con problemas.

La Función del Analizador `yyparse`

Se llama a la función `yyparse` para hacer que el análisis comience. Esta función lee tokens, ejecuta acciones, y por último retorna cuando se encuentre con el final del fichero o un error de sintaxis del que no puede recuperarse. Usted puede también escribir acciones que ordenen a `yyparse` retornar inmediatamente sin leer más allá.

El valor devuelto por `yyparse` es 0 si el análisis tuvo éxito (el retorno se debe al final del fichero). El valor es 1 si el análisis falló (el retorno es debido a un error de sintaxis).

La Funcion del Analizador Léxico `yylex`

La función del **analizador léxico**, `yylex`, reconoce tokens desde el flujo de entrada y se los devuelve al analizador. Bison no crea esta función automáticamente; usted debe escribirla de manera que `yyparse` pueda llamarla.

En programas simples, `yylex` se define a menudo al final del archivo de la gramática de Bison. En programas un poco más complejos, lo habitual es crear un programa en Flex que genere automáticamente esta función y enlazar Flex y Bison.

Un ejemplo sencillo

Vamos a ver un ejemplo sencillo en Bison para una gramática como la que sigue:

instrucciones -> instrucciones NL instrucción | instrucción

instrucción -> IDENTIFICADOR OPAS expresión

expresión -> término | expresión MÁS término

término -> IDENTIFICADOR | CONSTENTERA | APAR expresión CPAR

dónde:

IDENTIFICADOR: típico identificador en cualquier lenguaje de programación	
CONSTENTERA: entero sin signo	
OPAS:	:=
MÁS:	+
APAR:	(
CPAR:	)
NL:	\n

```
%{
/* fichero instrucciones.y */
#include <stdio.h>
%}
%token IDENTIFICADOR OPAS CONSTENTERA NL MAS APAR CPAR
%start instrucciones
%%
instrucciones : instrucciones instruccion
              | instruccion
              ;
instruccion : IDENTIFICADOR OPAS expresion NL
            ;
expresion : termino
          | expresion MAS termino
          ;
termino : IDENTIFICADOR
        | CONSTENTERA
        | APAR expresion CPAR
        ;
%%
int yylex()
{ Sólo necesaria si se trabaja sin Flex }

yyerror (s)  /* Llamada por yyparse ante un error */
     char *s;
{
  printf ("%s\n", s); /* Esta implementación por defecto nos valdrá */
  /* Si no creamos esta función, habrá que enlazar con –ly en el
momento de compilar para usar una implementación por defecto */
}

main()
{
  Acciones a ejecutar antes del análisis
```

```
  yyparse();
  Acciones a ejecutar después del análisis
}
```

Normalmente `int yylex()` no hay que declararla porque se asume el uso conjunto con Flex. Si se usara sólo Bison entonces si habría que declararla e implementarla. Si usamos Flex como analizador léxico, este podría ser el fuente de Flex:

```
/*
  fichero instrucciones.l
*/
%{
#include <stdio.h>
#include “y.tab.h” /* GENERADO AUTOMÁTICAMENTE POR BISON */
%}
separador       ([ \t””])+
letra           [a-zA-Z]
digito          [0-9]
identificador   {letra}({letra}|{digito})*
constEntera     {digito}({digito})*
%%
{separador}     {/* omitir */}
{constEntera}   {return (CONSTENTERA);}
“:=”            {return (OPAS);}
“+”             {return (MAS);}
{identificador}{return (IDENTIFICADOR);}
“(“             {return (APAR);}
“)”             {return (CPAR);}
\n              {return (NL);}
.               ECHO;
%%
```

El fichero y.tab.h es creado automáticamente por Bison, si usamos la opción –d al ejecutarlo, y contiene básicamente macros para definir como números los identificadores de tokens de la gramática de Bison que se van a leer en Flex (los CONSTENTERA, OPAS, MAS etc.).

Otra opción para implementar la gramática en Bison habría sido sustituir los tokens declarados que corresponden a un carácter, con el carácter en si (tokens de carácter). Algunas de las producciones en Bison cambiarían, por ejemplo:

```
instrucciones : instrucciones ‘\n’ instrucción
              | instrucción
              ;
```

Y el fuente de Flex también cambiaría. Desaparecerían las reglas que devolvían los tokens MAS, APAR, CPAR y NL y aparecería una nueva regla para devolver esos caracteres literalmente (como un carácter de C):

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

Compilación y ejecución de un programa Bison

Al ejecutar el comando `bison nombre_fuente.y` se crea un fichero en C llamado `nombre_fuente.tab.c`. Por compatibilidad con Yacc, existe la opción `-y` que fuerza a que el archivo de salida se llame `y.tab.c`. A partir de ahora supondremos que se usa siempre esta opción para facilitar la escritura de este documento. Otra opción útil es la

opción `-d`, que genera el archivo con las definiciones de tokens que necesita Flex (si se ha usado la opción `-y`, el archivo se llama `y.tab.h`). Este archivo `y.tab.h` normalmente se incluye en la sección de declaraciones del fuente de Flex (ver el ejemplo de la sección anterior). El fichero `y.tab.c` se puede compilar con la instrucción `gcc y.tab.c`. Los pasos para usar conjuntamente Flex y Bison serán normalmente:

```
1. bison -yd fuente.y
2. flex fuente.l
3. gcc y.tab.c lex.yy.c –lfl –o salida
```

Estos pasos generan un ejecutable llamado salida que nos permite comprobar qué palabras pertenecen al lenguaje generado por la gramática descrita en el fichero Bison. Si se compilan los ejemplos dados en la sección anterior y se ejecuta el fichero salida, podemos escribir una expresión válida:

```
var1 := var2 + 5
```

y cuando pulsemos `<Return>` bison no dará ningún error, señal de que la palabra pertenece al lenguaje de las expresiones correctas, sin embargo si escribimos algo que no es una expresión válida:

```
var1 := var2 ++ 34
```

al pulsar `<Return>` bison nos da un “parse error” (o un error de sintaxis) que nos dice que la palabra no pertenece al lenguaje.

Notas finales

Cuando una gramática resulta “demasiado ambigua” Bison dará errores. Definir cuando una gramática es “demasiado ambigua” queda fuera del alcance de esta introducción. Una posible solución a algunos tipos de ambigüedad es el uso adecuado de las declaraciones de precedencia de operadores. Si surgen otros conflictos, es útil invocar a Bison con la opción –v, lo que generará un archivo de salida y.output (si se usa la opción –y) donde se puede mirar qué reglas son las problemáticas (aquellas donde aparezca la palabra `conflict`). Hay algunas pistas más sobre esto en el manual de Bison.

Finalmente tened en cuenta algunas cosas:

- no uséis el mismo nombre para no terminales y variables que definái en el código C
- Bison distingue mayúsculas de minúsculas en los nombres de variables y tokens
- Bison no lee directamente la entrada, es necesario que exista una función yylex() que lea la entrada y la transforme en tokens (por ejemplo una como la que se crea usando Flex).

Bibliografía

Esta introducción a Flex y Bison se ha creado fundamentalmente a partir de la traducción de los manuales (“Manual de GNU Flex 2.5” escrito por Vern Paxson y traducido por Adrián Pérez Jorge, y el “Manual de GNU Bison 1.27” escrito por Charles Donnelly y Richard Stallman y traducido por Adrián Pérez Jorge). También se han incorporado algunas ideas y ejemplos de las varias versiones de los enunciados de prácticas de esta asignatura que se han ido utilizando de año en año y unos cuantos párrafos aclaratorios.

Tenéis links a los manuales completos, así como a las traducciones, en la página Web de la asignatura (http://webdiis.unizar.es/asignaturas/LGA).