Recuperación de errores y eliminación de conflictos en Bison

Lenguajes de Programación y Procesadores de Lenguajes
Escuela Técnica Superior de Ingeniería Informática
Universitat Politècnica de València

Índice

1. Presentación

En este documento se presenta la forma de analizar y resolver conflictos en un analizador sintáctico generado por Bison, y cómo incorporar un mecanismo de recuperación de errores sintácticos.

Para realizar este trabajo se recomienda al alumno que, además de estudiar detenidamente este documento consulte el manual de Bison.

Para conocer con detalle cómo es el analizador sintáctica LALR(1) generado por Bison resulta de mucha utilidad la información que genera Bison. Al ejecutar Bison con la opción *-v* se vuelca toda la información del analizador LALR(1) en un fichero de texto cuyo nombre por defecto tendrá la extensión ".out". En este archivo se encuentran:

1. Las reglas de la gramática.
2. Los posibles conflictos.
3. El autómata LALR(1): estados, elementos LR(1), acciones en cada estado,...

2. Depuración y resolución de conflictos

2.1. Comportamiento por defecto

Frecuentemente la gramática que se usa para especificar el lenguaje a procesar no es LALR(1) lo que significa que la tabla del analizador que construye Bison contendrá conflictos. Esto es particularmente importante cuando se trabaja con gramáticas ambiguas. Por ejemplo, si se compila con Bison la especificación sintáctica siguiente:

```
%token PARA_ PARC_  CTE_  MAS_ MENOS_ POR_ DIV_

%%
expMat : exp
       ;
exp : exp MAS_   exp
```

```
    | exp MENOS_ exp
    | exp POR_   exp
    | exp DIV_   exp
    | PARA_ exp PARC_
    | CTE_
    ;
```

se obtendrá el siguiente mensaje:

```
$ ... conflictos: 16 desplazamiento(s)/ reducción(ones)
```

Los conflictos de este ejemplo surgen porque la gramática es ambigua.

La situación donde sería válido tanto un desplazamiento como una reducción se denomina *conflicto desplazamiento/reducción*. Bison está diseñado para resolver estos conflictos eligiendo por defecto el *desplazamiento*. Se produce un *conflicto de reducción/reducción* si hay dos o más reglas que pueden aplicarse en el mismo estado con el mismo símbolo de anticipación en la cadena de entrada. Por defecto Bison resuelve los conflictos reducción/reducción eligiendo la regla que *aparece primero* en la gramática.

Cuando la elección por defecto de Bison ante un conflicto no resulta adecuada, se puede usar la precedencia y asociatividad de los símbolos gramaticales y el orden de las reglas para indicar qué acción debe seleccionarse.

2.2. Uso de precedencia y asociatividad ante conflictos reducción/desplazamiento

Ante un conflicto **reducción/reducción** Bison elige la regla que aparece primero en la gramática. Por lo tanto para resolver este tipo de conflictos basta con ordenar de la forma deseada las reglas gramaticales en el fichero de entrada de Bison.

Para resolver un conflicto reducción/desplazamiento es necesario indicar la asociatividad y precedencia de algunos símbolos terminales. La asociatividad y precedencia de los símbolos léxicos (tokens) se indica al definir cada símbolo en la sección de declaraciones Bison mediante **%left** (símbolo/s asociativo/s a izquierdas) o **%right** (símbolo/s asociativo/s a derecha) en lugar de %token.

- La precedencia de cada símbolo viene determinada por el orden en que se definen con %left o %right en Bison: Los tokens están definidos de menor a mayor precedencia.
- La precedencia de cada *regla* es la del último terminal que aparece en su lado derecho.

La resolución de los conflictos **reducción/desplazamiento** se realiza comparando la precedencia de la regla a reducir y el símbolo a desplazar con los que se plantea el conflicto:

1. Si la precedencia del token es mayor que la precedencia de la regla se selecciona la acción "desplazar".
2. Si la precedencia del token es menor que la precedencia de la regla se "reduce".
3. Si las precedencias son iguales Bison considerará la asociatividad en ese nivel de precedencia.

Por ejemplo, se pueden resolver los conflictos de la gramática ambigua de la sección anterior definiendo la asociatividad (a izquierda) de los símbolos que representan los operadores matemáticos y estableciendo al mismo tiempo su precedencia relativa (mayor la del `POR_` y `DIV_` que la del `MAS_` y `MENOS_`):

```
%token PARA_ PARC_  CTE_
%left MAS_ MENOS_
%left POR_ DIV_
```

3. Recuperación de errores

El analizador sintáctico generado por Bison informa de los errores sintácticos llamando a la función `yyerror()`' que debe ser escrita por el programador. Esta función se llama cuando se encuentra un error y recibe como argumento una cadena (normalmente el mensaje de error que se desea mostrar). Por defecto, el análisis sintáctico acabará tan pronto como se detecte un error, pero cuando se compila un programa es deseable que con una única compilación se muestren todos los errores que contiene. Por esta razón no es aceptable que el análisis finalice ante el primer error del programa fuente.

Es posible definir cómo recuperarse de un error sintáctico en Bison escribiendo reglas para reconocer el token especial `error`. Éste es un símbolo terminal predefinido en Bison (no es necesario declararlo) y está reservado para el tratamiento de errores. El analizador generado por Bison genera un token `error` cada vez que detecta un error de sintaxis. Si se ha facilitado una regla que reconozca este token en el contexto en el que se ha producido el error, el analisis sintáctico podrá continuar. Para ello, el analizador sintáctico generado por Bison seguirá los tres pasos siguientes:

1. Se desapilan todos los estados y objetos de la pila del analizador LALR(1) hasta que se encuentre un estado en el que se acepte el token `error`.

2. Se desplazará el token `error`.

3. Se saltarán todos los tokens de la entrada hasta encontrar uno con el que se pueda realizar un desplazamiento en el estado al que ha pasado el autómata.

Además, para prevenir una cascada de mensajes de error, el analizador generado por Bison no mostrará mensajes de error para otro error de sintaxis que ocurra "poco" después del primero: solamente después de que se hayan desplazado con éxito **tres** tokens de entrada consecutivos se reanudarán los mensajes de error.

Para que este mecanismo de recuperación de errores funcione, el programador debe *añadir* a las reglas de su analizador nuevas reglas que incluyan el token `error`. La inclusión de reglas con el token `error` es un trabajo que difícilmente puede mecanizarse y requiere un estudio detallado del comportamiento que tendrá el analizador.

Por ejemplo, en la siguiente especificación Bison que reconoce expresiones matemáticas se ha añadido una producción con el símbolo `error`: $fac \rightarrow (error)$ Si aparece un error mientras se analiza una expresión entre paréntesis y, si previamente se ha leído un paréntesis abierto, el analizador saltará todos los símbolos de la entrada hasta leer un `)`.

```
expMat : exp
       ;
exp  : exp MAS_   term
     | exp MENOS_ term
     | term
     ;
term : term POR_ fac
     | term DIV_ fac
     | fac
     ;
fac  : PARA_ exp PARC_
     | PARA_ error PARC_
     | CTE_
     ;
```

Añadir pocas reglas con el símbolo `error` puede provocar que el analizador se salte demasiados símbolos del programa fuente antes de recuperarse del error. Pero añadir muchas reglas con el símbolo `error` puede provocar que se muestren más errores de los que realmente contiene el programa fuente. Por tanto, la inclusión de reglas con el símbolo `error` debe hacerse cuidadosamente.

4. Ejercicios

Considera la siguiente gramática que reconoce y calcula el valor de expresiones matemáticas sencillas:

```
expMat:  exp              { printf("Resultado: %d\n", $1); }
      ;
exp :  exp  MAS_  exp     { $$ = $1 + $3 ; }
    | exp  MENO_  exp     { $$ = $1 - $3 ; }
    | exp  POR_  exp      { $$ = $1 * $3 ; }
    | exp  DIV_  exp      { $$ = $1 / $3 ; }
    | PARA_  exp  PARC_   { $$ = $2 ;}
    | CTE_                { $$ = $1 ;}
    ;
```

donde `CTE_` representa una constante entera sin signo. El analizador léxico puede devolver el valor numérico de la constante a través de la variable `yylval` que obtendrá su valor convirtiendo a entero la cadena `yytext` (lexema leido). Para ello basta con poner en las acciones asociadas al patrón de `CTE_` del fichero de Flex:

```
{entero}        {ECHO; yylval = atoi(yytext); return(CTE_) ; }
```

1. Construye un analizador léxico-sintáctico para la gramática anterior.
2. Resuelve todos los conflictos del analizador usando precedencia y asociatividad.
3. Ejecuta el analizador con varias expresiones expresiones matemáticas de prueba y comprueba que la calculadora obtiene el resultado esperado.
4. Prueba a cambiar la precedencia y asociatividad entre los operadores y observa como se obtienen resultados distintos para la misma expresión matemática.
5. Añade a la gramática alguna/s regla/s para el tratamiento del token `error` que permita que el nuevo analizador se recupere en modo pánico ante errores en la entrada. Asegúrate de que el analizador resultante detecta los dos errores que contiene la siguiente cadena:

   `3 + 5 * (12 + /5) + ((7 + 2) -*3).`