Prácticas de Lenguajes, Gramáticas y Autómatas

Prácticas 3 y 4

Cuarto cuatrimestre (primavera) de Ingeniería en Informática

Curso 2010-2011

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

Profesor Responsable: Jorge Júlvez
Dpto. Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Práctica 3

Requisitos: *Haber hecho las prácticas anteriores y haberse leído la introducción a Bison que se os dio con los guiones anteriores y que también está en la página Web de la asignatura (para esta práctica no es necesario mirar la parte de semántica del lenguaje, ni la precedencia de operadores, ni la sección sobre la colección de tipos de valores).*

Objetivo: *El objetivo principal de esta tercera práctica de la asignatura es que el alumno se familiarice con el manejo de la herramienta Bison, un generador de analizadores sintácticos. y con el uso de éste conjuntamente con Flex. La práctica constará de tres ejercicios: el primero de ellos, tendrá como objetivo la creación guiada de un reconocedor de palabras del lenguaje formado por las expresiones enteras bien escritas que utilizan + y *. Mientras que en el segundo y en el tercero, el alumno pondrá en práctica los conocimientos adquiridos para la creación de reconocedores de palabras para otros lenguajes que se le especifican.*

Ejercicio 1

Bison es una herramienta de gran potencia que, en uso conjunto con Flex, permite construir compiladores. En esta práctica veremos una pequeña introducción a Bison, haciendo hincapié en el aspecto que más nos interesa en esta asignatura, que es su capacidad para reconocer palabras que pertenecen al lenguaje generado por una gramática libre de contexto.

Un fichero fuente Bison describe una gramática. El ejecutable correspondiente nos dice si una entrada textual corresponde o no al lenguaje generado por la gramática.

Es importante notar la diferencia con Flex: en este caso se toma la entrada como una única palabra para la que hay que ver si está o no en el lenguaje que genera la gramática.

Un fichero fuente Bison es de la forma:

%token α_1 α_2 α_3 ... α_n

%start β

%%

γ_1;

γ_2;

...

γ_k;

Donde $\{\alpha_1\ \alpha_2\ \alpha_3\ ...\ \alpha_n\}$ es el **alfabeto** (conjunto de terminales de la gramática), β es el **símbolo inicial** (no terminal de la gramática), y $\gamma_{1...}\ \gamma_k$ son las reglas o **producciones** de la gramática. **Los símbolos terminales se denominan en Bison tokens**.

Cada regla (producción) se escribe de la forma:

$X_1: X_2 \ldots X_j;$ ***(dos puntos en lugar de la flecha, y termina en punto y coma)***

donde X_1 es una variable (no terminal) y $X_2 \ldots X_j$ son variables (no terminales) y tokens (terminales). No es necesario declarar las variables de la gramática (conjunto de no terminales de la gramática), ya que todo lo que no sean tokens (que si se declaran) se consideran variables.

Supongamos que queremos reconocer las palabras del lenguaje formado por las expresiones enteras con paréntesis que utilizan + y * (por ejemplo "4", "4 + 2","(4 + 3) * 5" etc.) :

Para ello podríamos crear una gramática incontextual muy sencilla:

S → T | T + T | T * T

T → ENTERO | (S)

Que traducida a un fuente básico de Bison sería::

```
%token PARIZ PARD MAS POR INTEGER
%start S
%%
S: T | T MAS T | T POR T
 ;
T: INTEGER | PARIZ S PARD
 ;
```

Pero además, necesitaríamos un fichero fuente de Flex (fuente.l) que lea la entrada estándar y reconozca las ocurrencia de los diferentes tokens de nuestra gramática, es decir:

```
%{
#include "y.tab.h" /* GENERADO AUTOMÁTICAMENTE POR BISON */
%}
%%
[\-]?[0-9]+ return(INTEGER);
\+          return(MAS);
\*          return(POR);
\(          return(PARIZ);
\)          return(PARD);
\n          return(0);
[ \t]       {/* ignorar blancos y tabuladores */}
```

Añadimos algunas definiciones de funciones necesarias al fuente de Bison (fuente.y) , que quedaría como sigue:

```
%token PARIZ PARD MAS POR INTEGER
%start S
%%
S: T | T MAS T | T POR T
 ;
T: INTEGER | PARIZ S PARD
 ;
%%
int yyerror(char* s) {
          printf("\n%s\n", s);
    return 0;
}
main() {
    yyparse();
}
```

y compilamos (ojo con el orden):

```
bison -yd fuente.y      (esto genera y.tab.c e y.tab.h)
flex fuente.l           (esto genera lex.yy.c)
gcc y.tab.c lex.yy.c -lfl -o ejemplo
```

"ejemplo" contiene ahora un ejecutable que reconoce todas las expresiones enteras bien escritas. Este ejecutable puedes utilizarlo desde teclado como desde un fichero (redirigiendo la entrada, igual que se hace con Flex cuando está solo). Si la entrada es una palabra del lenguaje, no da ninguna salida. Si la entrada no está en el lenguaje, el resultado será "***parse error***".

Ejercicio 2

Tenéis otro ejemplo en la introducción a Bison, concretamente un lenguaje para expresiones de asignación a variable que en el lado derecho tienen una expresión aritmética formada por enteros, paréntesis y signos de suma.
Copiadlo, compiladlo y comprobad qué reconoce exactamente.
Después haced algunas modificaciones:

- Soportad productos, divisiones y restas.
- Soportad enteros negativos.
- Soportad que se puedan usar corchetes además de paréntesis (que estén correctamente "anidados" y asegurando que si abro un paréntesis lo cierre, y que si abro un corchete lo cierre).

Ejercicio 3

Utilizad Bison para generar ejecutables que reconozcan los siguientes lenguajes. Esto quiere decir que para las palabras del lenguaje la salida debe ser vacía y para las que no pertenecen al lenguaje "parse error".

1) $L = \{ z\, a^n\, z\, a^n\, b^m\, z\, b^m\, z \mid m, n \geq 0 \}$

2) $L = \{ a^i\, b^j\, c^j\, d^i \mid i,j \geq 1 \}$

3) $L = \{ 0^m\, 1^n \mid m > n \geq 0 \}$

Nota: cuidado con el nombre que le dais a los tokens en Bison. Si los llamáis a, b, c etc., se producen colisiones con algunas variables internas de Flex. Es mejor que los llaméis de forma algo más "complicada" como t_a, t_b etc.

Práctica 4

Requisitos: *Haber hecho las prácticas anteriores. No se requiere mirar nada más sobre Bison, aunque las acciones a media regla se pueden usar para hacer "trazas" de la ejecución de Bison (ver las notas del ejercicio de Bison).*

Objetivo: *El objetivo de esta cuarta práctica es verificar la sintaxis, parcialmente, de documentos XML con Flex y Bison. Se trata de ver un poco los aspectos básicos del estándar XML y seguir explorando las posibilidades de Flex y de Bison, no de construir un parser completo y correcto de XML. Vamos a construir un analizador que acepte algunos documentos XML bien formados y que rechace algunos que no lo estén, pero veremos que tendrá limitaciones que lo harán aceptar muchos mal formados y rechazar también muchos que son correctos. Profundizando un poco en las posibilidades de análisis semántico de Bison, el analizador se podría ampliar, de forma relativamente sencilla, para ser mucho más selectivo a la hora de aceptar documentos bien formados, pero no es el objetivo de este curso, ni de esta práctica, entrar en esas posibilidades.*

XML

XML es un estándar para la creación de documentos de texto con una estructura bien definida. Aunque originalmente pensado sobre todo como un formato de intercambio de datos entre plataformas, en la actualidad su uso se ha extendido, y se utiliza como formato nativo para los ficheros en muchas aplicaciones, para la descripción de servicios Web, como formato de almacenamiento en algunas bases datos, etc. El estándar es accesible desde http://www.w3.org/XML/.

En esta práctica vamos a diseñar un analizador, bastante limitado, de ficheros XML utilizando Flex y Bison. Nuestro analizador va a limitarse a una parte del aspecto léxico/sintáctico del reconocimiento de documentos XML, ya que el objetivo de la práctica es ver una aplicación "práctica" de los lenguajes regulares e incontextuales antes que crear un analizador "realista".

Estructura de documentos XML

Los documentos XML tienen una estructura definida con elementos delimitados mediante etiquetas y anidados de una forma apropiada. Las etiquetas XML son, básicamente, un texto entre los símbolos "<" y ">". En XML se distinguen mayúsculas de minúsculas, así que `<ETIQUETA>` y `<etiqueta>` son etiquetas diferentes. Hay etiquetas de apertura y etiquetas de cierre.

```
<ETIQUETA>Contenido de etiqueta</ETIQUETA>
```

Puede haber etiquetas que aparezcan sin etiqueta de cierre, pero es obligatorio que éstas lleven una barra (/) al final (se les llama elementos vacíos):

```
<ETIQUETA/>
```

Los elementos pueden tener atributos en sus etiquetas de apertura (o en su única etiqueta si son elementos vacíos). Los atributos tienen valores que deben ir entre comillas, ya sean dobles o simples. Si hay varios atributos, tienen que ir separados por al menos un espacio (o tabulador, fin de línea...).

```
<ETIQUETA ATRIBUTO = "Valor" ATRIBUTO2 = 'Valor2'>
```

Los comentarios se escriben así (pueden ocupar las líneas que se quiera):

```
<!-- Esto es un comentario en XML -->
```

XML bien formado

XML tiene una serie de reglas para la construcción de documentos bien formados. Un documento XML bien formado puede tener las etiquetas que queramos, pero debe cumplir con unas reglas sintácticas determinadas:

- Los documentos XML deberían empezar con una declaración de XML que especifique la versión del estándar que cumple el documento (esta declaración puede llevar otros atributos, pero son opcionales):

  ```
  <?xml version="1.0"?>
  ```

- El documento debe tener un elemento raíz (y sólo uno) que englobe a todos los demás entre una etiqueta de apertura y una de cierre. Los elementos englobados por otro se llaman sub-elementos de éste, o sus hijos.

- Todas las etiquetas de apertura deben tener su correspondiente etiqueta de cierra para los elementos que contengan a otros, o que contengan datos. Si aparece una etiqueta sola, debe tener una barra (/) al final de su texto (se llama elemento vacío).

- Entre dos etiquetas puede haber nada, espacios, texto y/o otras etiquetas.

- No pueden aparecer dentro del texto de un elemento (entre otros):

  ```
  <       (en su lugar poner &lt; )
  ```

- Los elementos deben anidarse "correctamente". Es decir, la etiqueta de cierre de un elemento hijo de otro, debe aparecer antes que la etiqueta de cierre de su elemento padre.

- Si aparecen atributos en algún elemento tienen que tener un valor, y éste debe ir entre comillas (dobles o simples).

Un ejemplo de documento XML bien formado:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Biblioteca>
<Libro>
  La catalogación de este libro se ha llevado a cabo a partir de
información histórica recopilada de varias fuentes
  <Título>
    El ingenioso hidalgo Don Quijote de la Mancha
  </Título>
```

```
  <Autor>
    <Nombre tipo = "Nombre">
      Miguel de Cervantes
    </Nombre>
    <Nombre tipo = "Apodo">
      El Manco de Lepanto
    </Nombre>
  </Autor>
  <Novela tipoNovela = "Caballerías"/>
</Libro>
<Libro>
  <Título>
    Introducción a la Teoría de Autómatas, Lenguajes y Computación
  </Título>
  <Autor>
    <Nombre tipo = "nombre">
      John E. Hopcroft
    </Nombre>
  </Autor>
  <LibroTexto clasificación = "Informática Teórica"/>
</Libro>
</Biblioteca>
```

XML válido

Un documento XML válido es aquel que además de bien formado, es conforme a cierta estructura previamente establecida. Esta estructura se especifica en forma de definición de tipo de documento (DTD) o mediante un esquema (Schema) que es el método más reciente y más potente. Un DTD o un esquema son fundamentalmente, una gramática que especifica qué elementos pueden/deben aparecer en un documento XML y como deben estar estructurados (los esquemas son más "potentes" y permiten definir con mucha precisión tipos de datos válidos en los elementos y atributos, cardinalidades complejas etc.). En esta práctica no entraremos para nada en la validez de documentos XML.

Lectura de XML con Flex

A continuación se proporciona un esqueleto del documento Flex que se usará para el análisis de ficheros XML. Se indica qué patrones hay que reconocer, y se proporciona un ejemplo comentado. La primera tarea será completarlo:

```
%{
#include "y.tab.h"
%}
... DEFINICIONES ...
%%
{OPEN_TAG}  {return openTag; /* "<HOLA a='45'>" */}
{CLOSE_TAG} {return closeTag; /* "</HOLA>" */}
{EMPTY_TAG} {return emptyTag; /* "<HOLA A='12'/>" */}
{INSTR}     {return instr;    /* "<?XML version="1.0"?>" */}
{COMMENT}   {/*LOS IGNORO*/   /* "<!-- COM -->" */}
{SPACES}    {/*LOS IGNORO*/   /* "  /n  /t" */}
{CDATA}     {return cData;    /* "Texto =3,&vale casi todo. \t\n" */}
.           {return badCar;   /* "<"*/}
```

Notas:

- Esta no es, ni mucho menos, la única o la mejor manera de analizar la entrada. Sólo es una que funciona razonablemente, es bastante completa y no es muy compleja.
- Los identificadores de etiqueta son parecidos a los de un lenguaje de programación típico, pero admiten expresamente cosas como dos puntos (:).
- Si una etiqueta admite atributos y lleva varios, deben estar separados por espacios.
- Ignoramos comentarios y espacios, salvo los que van entre los atributos que los trataremos dentro de la e.r. de la etiqueta que sea, dado que en general podemos organizar las etiquetas en un documento con margen para usar tabulaciones, espacios y fines de línea para indentarlo, y los comentarios los podemos poner más o menos donde queramos.
- La e.r. CDATA se emparejará con el texto libre que podemos escribir entre etiquetas. Se admite casi cualquier carácter (incluyendo espacios, saltos de línea, signos de puntuación...), salvo el de abrir etiqueta "<" y alguno más, que no vamos a ver.
- Tratamos de forma distinta espacios y CDATA (que puede llevarlos), porque CDATA debe aparecer en nuestra gramática, hay sitios donde es aceptable y sitios donde no, y los espacios, en general y salvo entre atributos, podremos ignorarlos y no devolvérselos a Bison, lo que facilita bastante la tarea.
- Devolvemos específicamente un token cuando encontremos un carácter que no se ha emparejado antes, porque casi con seguridad será un error del fichero XML y de esta forma es más sencillo que Bison lo detecte (aunque el token no aparecerá en ningún sitio en la gramática de Bison).

Análisis sintáctico con Bison

Hay que construir una gramática en Bison que acepte ficheros bien formados XML. Los tokens serán lo que devuelve Flex (están en el esqueleto que se os da).

Notas:

- En general queremos aceptar ficheros XML de la siguiente forma:

INSTR
OPEN_TAG
  RESTO DEL DOCUMENTO (CDATA, ELEMENTOS ANIDADOS, ELEMENTOS VACIOS...)
CLOSE_TAG

Es decir: requerimos que el documento empiece con una instrucción xml, exigimos que tenga una sola etiqueta de primer nivel (que englobe a todas las demás), y después permitiremos que tenga cualquier contenido aceptable: CDATA y elementos correctamente anidados.

- Atención: no podemos comprobar si la etiqueta de inicio corresponde con la de cierre sin entrar en aspectos más complejos de Bison, así que aceptaremos como buenos documentos incorrectos como este:

```
<?xml version="1.0"?>
<simple>
</tonto>
```

- Para hacer "trazas" en Bison, podéis utilizar acciones en las reglas. Aunque estas acciones sirven para muchas más cosas, como "depurador" os puede servir para localizar problemas en vuestra gramática. Por ejemplo:

```
S: {printf("Entra en ELEMENTOS\n");} ELEMENTOS {printf("Entra en
FIN\n");} FIN ;
```

Esto escribirá por pantalla una traza del parseo de la gramática.

Pruebas

En el directorio `/export/home/practicas/Practicas/LGA/pract4` tenéis varios ficheros XML que podéis usar para hacer pruebas:

- bueno1.xml hasta bueno4.xml son ficheros de tamaño y complejidad creciente que vuestro analizador deberá considerar como correctos (y que efectivamente son ficheros XML bien formados, el último de ellos un ejemplo real, mínimamente simplificado, de la descripción de un servicio web).
- fallo1.xml hasta fallo6.xlm son ficheros con distintos errores (sólo un error en cada fichero) que vuestro analizador deberá considerar como incorrectos (y que efectivamente son ficheros XML no bien formados). El error concreto aparece en un comentario en cada fichero.
- problema1.xml ilustra el principal problema que tiene el analizador, y es que el fichero es incorrecto (la etiqueta de apertura y de cierre son distintas) pero el analizador lo considerará correcto al no poder tener en cuenta cual es el nombre de las etiquetas.