Práctica 3
Introducción al Manejo de *Bison*

Tareas

1. Estudia la sección sobre *Bison* en el documento `Intro_Flex_Bison.pdf` (página 10 a final) y descarga el manual de *Bison* como soporte para consultas (`bison.pdf`).
2. Lee la introducción de esta práctica y realiza los ejercicios 1 a 4 propuestos.
3. Elabora la memoria de la práctica y entrégala junto con los ficheros fuente de los ejercicios 2 a 4 por el mismo procedimiento que en las prácticas anteriores. La fecha tope de entrega será hasta el día anterior al comienzo de la Práctica 4.

- Debes crear un directorio que contenga **exclusivamente** el fichero con la memoria en formato *PDF*, los ficheros fuente con tu código (*.l* de *Flex* y *.y* de *Bison*) y los de prueba (*.txt* de texto). No uses subdirectorios.
- En caso de que el fichero .zip resultante tenga un tamaño mayor de 512 KB deberás repetir la creación del directorio dividiéndolo en varios ficheros .zip de como máximo 512 KB cada uno, en ese caso llama a los ficheros resultantes `nipPrX1.zip`, `nipPrX2.zip`, etc.

Introducción

El objetivo principal de esta práctica es familiarizarse con la herramienta de creación de analizadores sintácticos *Bison*, entender cómo utilizarla en conjunción con *Flex* y aplicar lo aprendido sobre teoría lenguajes. Para ello, se plantea un ejercicio guiado para desarrollar un reconocedor de palabras del lenguaje formado por las expresiones de enteros bien escritos que utilizan '+' y '∗' y otros dos ejercicios para desarrollar reconocedores de palabras para otros lenguajes.

Ejercicio 1

Bison es una herramienta que, usada conjuntamente con *Flex*, permite construir compiladores. En la asignatura *Teoría de la Computación* nos interesa para producir programas

capaces de reconocer palabras que pertenecen a un lenguaje generado por una gramática libre de contexto. Así, en esencia, un fichero fuente *Bison* contendrá la descripción de una gramática y el ejecutable que se genere permitirá determinar si una entrada textual corresponde o no al lenguaje generado por dicha gramática.

A diferencia de *Flex*, se considera toda la entrada como una única expresión sobre la que se verifica si pertenece o no al lenguaje generado por la gramática.

Un fichero fuente Bison tiene una estructura como la siguiente:

$$
\begin{array}{l}
\%token\ \ \alpha_1\ \alpha_2\ \ldots\ \alpha_n \\
\%start\ \ \beta_1 \\
\%\,\% \\
\beta_1 :\ \ \beta_i\ \ldots\ \beta_j; \\
\ldots \\
\beta_i :\ \ \alpha_i\beta_i \mid \beta_k; \\
\beta_j :\ \ \beta_j\alpha_i \mid \beta_k; \\
\beta_k : \alpha_i \mid \lambda;
\end{array}
$$

Donde $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ representa el alfabeto o conjunto de símbolos terminales de la gramática, β_1 sería el símbolo inicial de la gramática (no terminal), y $\beta_1, \ldots, \beta_i, \beta_j, \beta_k$ serían reglas de producción de la gramática. Para el ejemplo de estructura expuesto β_1 produciría no terminales, β_i produciría terminales por la izquierda con no terminales o sólo no terminales, β_j produciría terminales por la derecha con no terminales o sólo no terminales y, finalmente β_k sólo produciría terminales. Los símbolos terminales se denominan *tokens* en *Bison*.

En la descripción de la gramática en *Bison* 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 considerarán variables.

Supongamos que se quiere reconocer las palabras del lenguaje formado por las expresiones enteras con paréntesis que utilizan '+' y '∗' (p. ej. "4", "$4+2$","$(4+3)*5$", ...). Para ello se podría crear la siguiente gramática incontextual:

$$
\begin{array}{l}
S \rightarrow\ \ T \mid\ T+T \mid T*T \\
T \rightarrow\ \ ENTERO \mid (S)
\end{array}
$$

Una **primera aproximación** a la codificación en *Bison* (fichero *fuente.y*):

```
%token PARIZ PARD MAS POR ENTERO
%start s
%%
s : t | t MAS t | t POR t
;
t : ENTERO | PARIZ s PARD
;
%%
```

Adicionalmente, se necesitaría crear un analizador léxico con *Flex* que permitiera reconocer las ocurrencia de los diferentes tokens de la gramática (fichero *fuente.l*):

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

Se añaden algunas definiciones de funciones necesarias al código fuente del programa en *Bison* (*fuente.y*):

```
%{
#include <stdio.h>
%}
%token PARIZ PARD MAS POR ENTERO
%start s
%%
s: t | t MAS t | t POR t
;
t: INTEGER | PARIZ s PARD
;
%%
int yyerror(char * s){
printf( " %s\n", s);
return 0;
}

int main(){
yyparse();
}
```

Para compilar:

```
bison -yd fuente.y   (genera y.tab.c e y.tab.h)
flex fuente.l        (genera lex.yy.c)
gcc y.tab.c lex.yy.c -lfl -o ejecutable
./ejecutable <argumento_1>...<argumento_n>
```

El programa de nombre *ejecutable* reconoce todas las expresiones enteras bien escritas de acuerdo con la gramática propuesta. Este ejecutable puedes probarlo desde teclado y también usando ficheros de prueba. Para este último caso, habría que redirigir la entrada estándar con el *pipe* '<' como se explicó en clase de prácticas para el caso de *Flex*. Si la entrada es una palabra del lenguaje, no da ningún mensaje a la salida. En caso contrario, el resultado será ***"parse error"***.

Ejercicio 2

Implementa el ejemplo de la introducción a *Bison* sobre el lenguaje para expresiones de asignación a variable (página 18 del documento `Intro_Flex_Bison.pdf` que podeis encontrar en la sección "material docente - laboratorio" de la página web de la asignatura) que en el lado derecho tienen una expresión aritmética formada por enteros, paréntesis y signos de suma.

Compílalo y haz pruebas para entender qué reconoce exactamente. A continuación, realiza las siguientes modificaciones para que reconozca:

- Operadores de resta y multiplicación.
- Enteros negativos y números en punto flotante (Ejemplo: -0.1234E-5).
- Corchetes correctamente anidados.

Ejercicio 3

Implementa un analizador sintáctico para la siguiente gramatica:

$$\boxed{\begin{array}{l} A \to CxA \mid \epsilon \\ B \to xCy \mid xC \\ C \to xBx \mid z \end{array}}$$

- Construye ejemplos de árboles de derivación y compruébalos con el programa.
- ¿Qué lenguaje reconoce?

Ejercicio 4

Implementa en *Bison* analizadores sintácticos que reconozcan los siguientes lenguajes:

- $L = \{0^n\, 1^m \mid m \geqslant 1, n = 3m\}$
- $L = \{a^i\, (bc)^j \mid i = 2j + 1, i > 1\}$
- $L = \{a^m\, b^n \mid m > n \wedge m, n > 0\}$

Notas:

Es importante que documentes tus decisiones de diseño de las gramáticas y su implementación con *Bison*. Así, es posible que encuentres conflictos de desplazamiento-reducción o reducción-reducción que puedes tratar de resover modificando las gramáticas y/o introduciendo directivas de asociatividad en las producciones de *Bison*.

En caso de que no logres eliminar los conflictos, puedes asumir el comportamiento estandar de *Bison* (desplazar) y documentar los problemas de tu implementación, acompañándolos de trazas de ejecución.

En cuanto a la ejecución de los programas *Bison*, para las entradas válidas, los programa no han mostrar nada por pantalla y para las que no pertenezcan al lenguaje deben mostrar por pantalla ***"parse error"***.

Se recomienda no usar nombres para los *tokens* como $a, b, c, \ldots$ en *Bison*, ya que se pueden producir colisiones con algunas variables internas de *Flex*. Para el caso anterior, utiliza mejor $T\_a, T\_b, T\_c, \ldots$.