3 Desarrollo de una calculadora programable

En las primeras siete sesiones se propone el diseño de una calculadora capaz de evaluar expresiones aritméticas. El proceso se basará en una construcción incremental, que permita abordar de forma gradual las particularidades de las herramientas *yacc/bison***.**

3.1. Calculadora elemental.
Empezamos con una calculadora elemental que sea capaz de interpretar y evaluar resultados a partir de expresiones como las que siguen (números de un solo dígito).

1<*intro*>
1+2<*intro*>
1+2*3-4/5<*intro*>

3.2. Calculadora con números enteros con signo y expresiones con paréntesis.

3.3. Diferenciación de los niveles léxico y sintáctico.
Descomponer la gramática en niveles *sintáctico* y *léxico* para descargar la fase de análisis sintáctico.

3.4. Calculadora con mayor complejidad.
Se añade mayor complejidad a la calculadora: número reales. Análisis léxico con *flex*.

3.5. Variables Simples.
Añadir a la calculadora el manejo de variables sencillas.

3.6. Variables Globales con la *Tabla de Símbolos*.

3.7. Funciones con la *Tabla de Símbolos*.

El desarrollo está basado libremente en la calculadora *HOC* de:

- B.W. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice Hall, Upper Saddle River, NJ, Second edition, 1984.

- B.W. Kernighan y R. Pike, *El entorno de programación UNIX*, Prentice-Hall, 1987.

A la hora de implementar un analizador léxico o sintáctico se podría optar por programar los autómatas finitos o de pila equivalentes a la gramática dada, pero esto es una opción extremadamente laboriosa. De ahí se explica la existencia de numerosas herramientas que básicamente realizan una traducción de Gramática a un AFD/AP.

Una de las primeras fue *yacc* (yet another compiler compiler), creada por Stephen C. Johnson en el laboratorio Bell de AT&T para su sistema operativo (Unix). Está documentado con un ejemplo interesante en el libro:

- B.W. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice Hall, Upper Saddle River, NJ, Second edition, 1984.
- B.W. Kernighan y R. Pike, *El entorno de programación UNIX*, Prentice-Hall, 1987.

Si *yacc* sirve para generar analizadores sintácticos, la herramienta homóloga para crear analizadores lexicográficos es *lex/flex*.
De entre las múltiples variantes creadas podemos destacar algunas derivaciones:

- *yacc* → *bison* (GNU)
- *lex* → *flex* (GNU)

Para Java se crearon otras versiones:

- *bison* → *jbison* (¿obsoleta?) → … → *cup*
- *flex* → *jlex* → *jflex* (GNU)
- Otra posibilidad interesante es *JavaCC* (Java Compiler Compiler) que integra los niveles léxico y sintáctico en la misma herramienta, aunque sólo permite la creación de analizadores descendentes.

Estas herramientas vienen documentadas en el libro

- "Compiladores. Traductores y Compiladores con Lex/Yacc, JFlex/Cup y JavaCC", Sergio Gálvez Rojas, Miguel Ángel Mora Mata:
  http://www.lcc.uma.es/~galvez/Compiladores.html

- Otra herramiente interesante es ANTLR, desarrollado por Terence Parr:
  http://www.antlr.org/

Un tutorial sobre el lenguaje C bastante completo es:

- "Aprenda el lenguaje ANSI C como si estuviera en primero", de Javier García de Jalón de la Fuente y otros:
  http://www.tecnun.es/asignaturas/Informat1/ayudainf/aprendainf/AnsiC/leng_c.pdf

3.1 Sesión 1. Calculadora elemental

Calculadora capaz de interpretar y evaluar resultados a partir de expresiones con números de un solo dígito
Ejemplos:

1<*intro*>
1+2<*intro*>
1+2*3-4/5<*intro*>

Podemos empezar por plantear la gramática que nos genera dicho lenguaje. :

G1a

Axioma → *Expresion* <*intro*>
Expresión → *Operando* | *Expresion Operador Expresion*
Operando → *Digito*
Operador → + | - | * | /
Digito → *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*

- La producción de *Operando* es de redenominación y un poco redundante, pero la conservamos para futuras ampliaciones de la gramática.
- <*intro*> es un *Terminal* que representa el símbolo 'final de línea' que será empleado para marcar el final de una expresión.

Si representamos esta gramática mediante un diagrama sintáctico (***Diag1a***), observamos varios detalles:

- Un pequeño fallo 'estético' aparece en el flujo correspondiente al *No Terminal Expresión* mediante una caja haciendo referencia al propio *No Terminal*. Esto se debe a que la producción tiene una doble recursividad a izquierdas y a derechas. Para este tipo de diagramas conviene tener la gramática sin recursividad por la izquierda.
- Además, podemos observar que la gramática es ambigua; es decir, es posible derivar la misma palabra generando más de un árbol de derivación (la expresión *1+2*3*<*intro*> tendría dos árboles de derivación). Nos interesa disponer de una gramática recursiva a derechas y que no sea ambigua, así que para subsanarlo, reescribimos la producción.

Si intentamos evitar estos defectos, podemos obtener diversas soluciones:
a - Por sustitución. En el ejemplo anterior observamos que la primera *Expresion* en la parte derecha de la producción original puede ser un simple *Operando*:

Expresion → *Operando* | *Operando Operador Expresion*

b - Por descomposición, obtendremos recursividad a varios pasos:

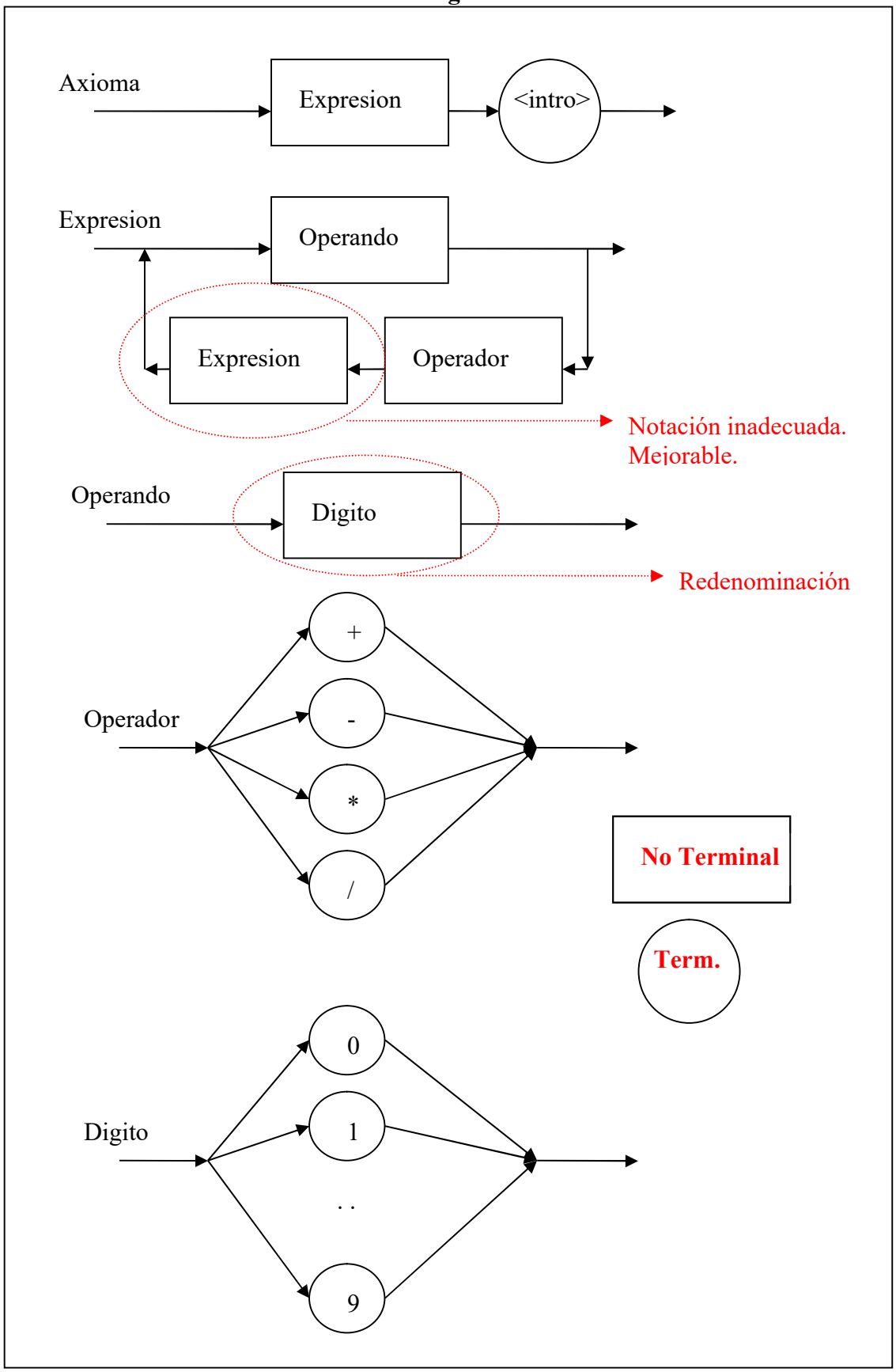
Expresion → *Operando* | *Operando RestoExpresion*
RestoExpresion → *Operador Expresion*

c - Por descomposición y factorizando, obtendremos:

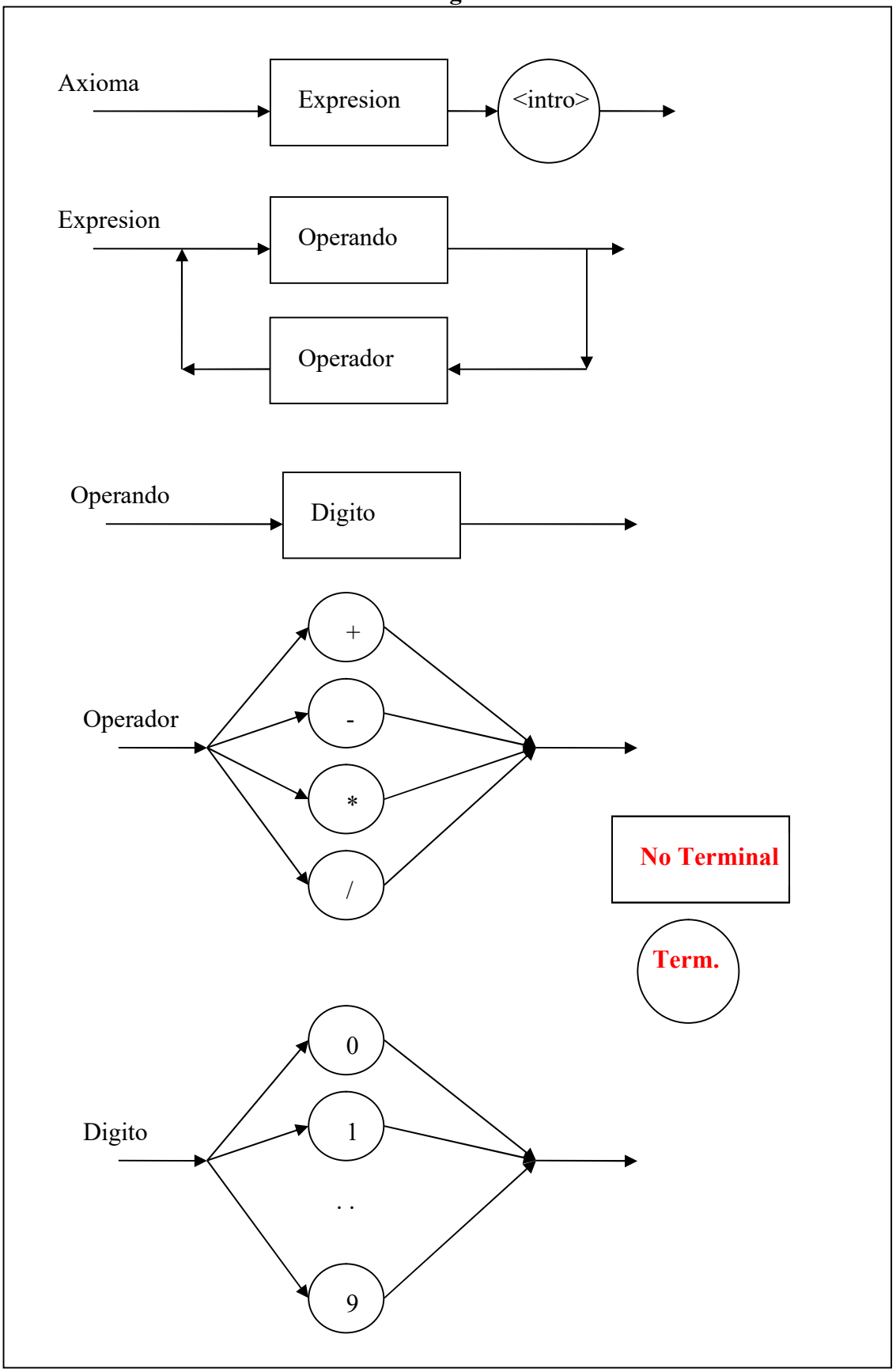
Expresion → *Operando RestoExpresion*
RestoExpresion → *lambda* | *Operador Expresion*

No nos vamos a pronunciar todavía sobre las ventajas de cada solución. En principio, nos puede valer cualquiera, pero vamos a emplear a continuación la primera de ellas (avj). El diagrama asociado se muestra en ***Diag1b***.

Diag1a


Axioma
Expresion
<intro>
Expresion
Operando
Expresion
Operador
Notación inadecuada.
Mejorable.
Operando
Digito
Redenominación
Operador
+
-
*
/
No Terminal
Term.
Digito
0
1
. .
9

Diag1b


Axioma
Expresion
<intro>
Expresion
Operando
Operador
Operando
Digito
Operador
+
-
*
/
No Terminal
Term.
Digito
0
1
. .
9

La gramática obtenida de la primera solución, que corresponde al diagrama ***Diag1b*** es la siguiente:

G1b

Axioma → Expresion <intro>
Expresión → Operando | Operando Operador Expresion
Operando → Digito
*Operador → + | - | * | /*
Digito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Antes de pasar a la implementación, comentamos algunos detalles de interés:

En las gramáticas definidas aquí, podemos distinguir dos tipos de símbolos, que por convención se denominan símbolos *Terminales*, y símbolos *No Terminales*. Al primer conjunto pertenecen los que componen las sentencias del Lenguaje, es decir los símbolos +, -, *, /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 e <intro>.

Los restantes símbolos empleados en la definición de la gramática son los *No Terminales* que sirven para construir estructuras gramaticales más complejas en base a los *Terminales* e incluso otros *No Terminales*.

Podemos añadir que la detección de los símbolos *Terminales* corresponde a una *fase léxica* (es decir, identificar *1*, + o *4* como símbolos *Terminales*), mientras que el análisis de la combinación de dichos símbolos para formar sentencias válidas (por ejemplo, *1+4*) corresponde a la denominada *fase sintáctica*. Hablaremos de una *fase semántica* cuando asociamos algún significado concreto a la sentencia formada o analizada (en el ejemplo, la *semántica* se refiere a la acción de aplicar el operador *suma* a los operandos *1* y *4*).

Por una cuestión práctica, relacionada con la fase semántica, vamos a reescribir la gramática ***G1b*** para obtener la equivalente ***G1e***:

G1e

Axioma → Expresion <intro>
*Expresion → Operando | Operando + Expresión | Operando – Expresion | Operando * Expresion | Operando / Expresion*
Operando → Digito
Digito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Se han duplicado las producciones de *Expresion* para sustituir el *No Terminal Operador* por cada uno de sus *Terminales* posibles.

Para implementar el programa nos vamos a basar en una conocida herramienta denominada *yacc* o *bison*, que permite representar de forma asequible las reglas sintácticas asociadas a sus correspondientes acciones semánticas. *bison* es un generador de código, esto es, a partir de unas especificaciones va a generar una serie de funciones y procedimientos que van a servir para realizar el análisis sintáctico mediante un *autómata a pila*. El analizador sintáctico generado se llamará *yyparse()*. El programa que realicemos debe comenzar haciendo una llamada a dicho procedimiento.

Aquí se va a comentar la implementación de forma desordenada respecto a la estructura física del código. Comenzamos por el procedimiento principal que debe llamar a *yyparse()*.

```
…

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

Como ya se ha comentado, *yyparse()* contiene un analizador basado en un *autómata a pila* que se va a encargar de comprobar que el flujo de entrada a nuestro programa (la calculadora) sea correcto e interpretable. Obviamente será necesario leer la entrada al programa, cosa que se hace a través de un analizador léxico. El procedimiento *yyparse()* hace una llamada a la función *yylex()* que debe encargarse de resolver el *análisis léxico*. Para implementar esta fase léxica podríamos utilizar la herramienta equivalente (*lex* o *flex*), pero la simplicidad de la tarea requerida en este caso nos permite emplear una sencilla función programada a medida.

Ya se había comentado que los símbolos *Terminales* (el léxico de nuestro lenguaje) son los símbolos +, -, *, /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 e <intro>. Así que la tarea de *yylex()* va a consistir en leer de la entrada, identificar dichos símbolos y entregárselos a *yyparse()*.

```
…
int yylex ()
{
    unsigned char c ;

    do {
         c = getchar () ;
    } while (c == ' ') ;

    return c ;
}
…
```

Se ha tenido en cuenta una pequeña circunstancia, y es que como separador de los símbolos pueden aparecer uno o varios espacios en blanco. Esto se resuelve mediante el bucle *do…while*. No se admite ningún otro símbolo separador.
Cualquier símbolo leido que sea distinto de espacio en blanco será considerado como un *Terminal*, y por ello será entregado a *yyparse()* mediante un *return*.

Queda por definir el bloque sintáctico que contiene las reglas que debe contemplar *yyparse()* para analizar el flujo de entrada. Este bloque se representa de forma análoga al de la notación BNF. Si partimos de la gramática ***G1e***, obtenemos el siguiente bloque de código en notación para *bison*.

```
axioma:      expresion '\n'
           ;

expresion:   operando
           | operando '+' expresion
           | operando '-' expresion
           | operando '*' expresion
           | operando '/' expresion
           ;

operando:    numero
           ;

numero:      '0'
           | '1'
           | '2'
           | '3'
           | '4'
           | '5'
           | '6'
           | '7'
           | '8'
           | '9'
           ;
```

Aquí hacemos hincapié en varios detalles. El primer *No Terminal* que aparece en el listado denominado *axioma*, va a ser el que represente al nodo padre del árbol de derivación. Los símbolos *Terminales*, que son entregados desde yylex como caracteres se representan como tales en esta gramática. En la notación de *bison* son denominados *literales* y deben ir encerrados por comillas símples. [1]

Queda el detalle de la representación del *Terminal* <*intro*>, que en *bison* vamos a hacer con el *literal* '\n' usado en el lenguaje C para representar al salto de línea.Es necesario asociar una *fase semántica* a la *sintáctica* para que el programa que vamos a generar pueda operar en función del flujo de entrada. *bison* permite realizar esto entremezclando acciones semánticas con las descripciones sintácticas.

```
axioma:      expresion '\n'          { printf ("Expresion=%lf\n", $1); }
           ;

expresion:   operando                { $$ = $1; }
           | operando '+' expresion  { $$ = $1 + $3; }
           | operando '-' expresion  { $$ = $1 - $3; }
           | operando '*' expresion  { $$ = $1 * $3; }
           | operando '/' expresion  { $$ = $1 / $3; }
           ;

operando:    numero                  { $$ = $1; }
           ;

numero:      '0'                     { $$ = 0 ; }
           | '1'                     { $$ = 1 ; }
           | '2'                     { $$ = 2 ; }
           | '3'                     { $$ = 3 ; }
           | '4'                     { $$ = 4 ; }
           | '5'                     { $$ = 5 ; }
           | '6'                     { $$ = 6 ; }
           | '7'                     { $$ = 7 ; }
           | '8'                     { $$ = 8 ; }
           | '9'                     { $$ = 9 ; }
           ;
```

[1] Es importante tener en cuenta que en *bison* no es válido representar un operador como ++ mediante '++' o "++". La única forma válida por ahora, aunque un poco discutible sería '+' '+'.

Tomemos como ejemplo una entrada al programa de
0+7<intro>

El primer caracter leido y entregado por *yylex()* será '0'. *Yyparse()* resolverá la rama de la gramática correspondiente a

```
numero:      '0'                    { $$ = 0 ; }
```

en la cuál la acción semántica $$ = 0 ; se encargará de asociar el valor 0 al *No Terminal numero* ($$ representa siempre al *No Terminal* de la parte izquierda de cada producción). El valor asociado será propagado hacia arriba para ser empleado en las resoluciones subsiguientes.

numero se resuelve con la producción

```
operando:    numero                 { $$ = $1; }
```

y al mismo tiempo se asocia al *No Terminal operando* el valor que tiene *numero*. En este caso, $1 representa el valor asociado al primer símbolo de la parte derecha (*numero*), mientras que $$ representa en este caso al *No Terminal* de la izquierda (*operando*). El valor de *operando* será propagado hacia arriba.

El siguiente Terminal leido y entregado por *yylex()* será '+' que permite a yyparse() abordar la resolución por la producción

```
expresión:
…
           | operando '+' expresion  { $$ = $1 + $3; }
```

En este momento todavía no es posible aplicar la acción semántica ya que por su disposición previamente será necesario resolver el símbolo *expresión* de la parte derecha.

Esto se logrará, una vez leido el carácter '7', a través de las producciones:

```
numero:
…
           | '7'                    { $$ = 7 ; }
```

```
operando:    numero                 { $$ = $1; }
```

```
expresion:   operando               { $$ = $1; }
```

Al retroceder de nuevo a

```
expresion:
…
           | operando '+' expresion  { $$ = $1 + $3; }
```

Tendremos los valores $1 = 0, $3 = 7, por lo que como resultado de la acción semántica al *No Terminal expresión* se le asigna el valor de la suma, es decir, 7.

El siguiente paso es propagar la resolución hacia arriba, a la producción

```
axioma:      expresion '\n'         { printf ("Expresion=%lf\n", $1); }
```

que, una vez leido el símbolo <intro>, podrá llevar a cabo la acción semántica correspondiente, la impresión del valor asociado a *expresión*.

Los últimos detalles respecto a la implementación en *bison*, corresponden a la declaración del tipo de valor que se va asociar a los *No Terminales*, en este caso números de coma flotante (`double`), a través de la definición de la constante

```
#define YYSTYPE  double
```

También será necesario incluir los ficheros de cabeceras necesarios para el buen funcionamiento del programa. En este caso, sólo son necesarias declaraciones para la función printf, así que basta con:

```
#include <stdio.h>
```

Bison permite al usuario definir un control de errores imprescindible en cualquier compilador, intérprete o traductor. En este caso vamos a ceñirnos a un control mínimo, que por ahora se limitará a imprimir el mensaje de error que transmita *yyparse()*. La función empleada se denominará *yyerror()* con la cadena de caracteres que reciba como parámetro:

```
int yyerror (char *mensaje)
{
    fprintf (stderr, "%s\n", mensaje) ;
}
```

En la siguiente hoja se ha recopilado el conjunto del código detallado hasta ahora. Obsérvese su particular estructura, incluidos algunos símbolos de notación imprescindibles para marcar las secciones (`%%, %{, %}`) que requiere *bison.*

Este código debe incluirse en un fichero con extensión .y , por ejemplo `calc1.y`
Una vez terminado, se aplicará la herramienta *bison* para generar el analizador sintáctico:

```
bison calc1.y
```

Como salida de la herramienta se obtendrá un fichero que combina el código en C de las funciones de usuario con las que correspondan al autómata a pila. En unix/linux es habitual que el fichero resultante se denomine calc1.tab.c (antiguamente `y.tab.c`).

El siguiente paso consistirá en compilar:

```
gcc calc1.tab.c –o calc1
```

Obteniendo el ejecutable `calc1`.

Si lanzamos el proceso e introducimos la siguiente secuencia:

```
./calc1<intro>
2+3<intro>
```

El programa nos debe responder con:

```
Expresion=5.000000
```

Si introducimos a continuación otra expresión, sin lanzar el programa de nuevo:

```
1+1<intro>
```

El programa nos responderá con:

```
syntax error
```

Cuestiones abiertas:

1.1 ¿Por qué aparece este error?

1.2 ¿Eres capaz de evitar que aparezca? Describe qué hay que añadir a la gramática (también las acciones semánticas) y qué efectos produce en el resultado.

calc1.y

```
%{                                  /* Seccion 1  Declaraciones de C y bison */
#include <stdio.h>
#define YYSTYPE  double
%}
                                    /* Seccion 2  Declaraciones de bison   */
%%
                                    /* Seccion 3  Sint - Semantico   */

axioma:      expresion '\n'           { printf ("Expresion=%lf\n", $1); }
           ;

expresion:   operando                 { $$ = $1; }
           | operando '+' expresion   { $$ = $1 + $3; }
           | operando '-' expresion   { $$ = $1 - $3; }
           | operando '*' expresion   { $$ = $1 * $3; }
           | operando '/' expresion   { $$ = $1 / $3; }
           ;

operando:    numero                   { $$ = $1; }
           ;

numero:      '0'                      { $$ = 0 ; }
           | '1'                      { $$ = 1 ; }
           | '2'                      { $$ = 2 ; }
           | '3'                      { $$ = 3 ; }
           | '4'                      { $$ = 4 ; }
           | '5'                      { $$ = 5 ; }
           | '6'                      { $$ = 6 ; }
           | '7'                      { $$ = 7 ; }
           | '8'                      { $$ = 8 ; }
           | '9'                      { $$ = 9 ; }
           ;

%%
                                       /* Seccion 4  Codigo en C   */
int yyerror (char *mensaje)
{
    fprintf (stderr, "%s\n", mensaje) ;
}

int yylex ()
{
    unsigned char c ;

    do {
         c = getchar () ;
    } while (c == ' ') ;

    return c ;
}

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