3.1.1 Respuestas a las cuestiones de la Sesión 1

1.1 ¿Por qué aparece este error?

El lenguaje que genera la gramática es finito, sólo permite evaluar una única expresión. Al introducir obtenemos un error porque no está contemplada.

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

Una solución que se nos puede ocurrir instintivamente es emplear un bucle para aplicar el parser/scaner de forma reiterada a las expresiones:

```
while (1)
      yyparse () ;
```

Sin embargo, la idea que perseguimos aquí consiste en aprovechar el potencial de las gramáticas para descargar en ellas la tarea de programar un compilador/intérprete, o en este caso la calculadora.

Para reconocer más de una expresión, tenemos que ampliar el Lenguaje para que contenga secuencias de una o más expresiones. Nos basaremos por ello en una Gramática recursiva que permite generar Lenguajes infinitos.

Empleando la notación de las expresiones regulares, la primera versión del intérprete nos permite evaluar una expresión: <expresión>·<intro>

Ahora necesitamos evaluar:

(<expresión>·<intro>)$^+$ = (<expresión>·<intro>)·(<expresión>·<intro>)*

Las producciones gramaticales que nos permiten generar el mismo lenguaje las obtenemos de:

palabra base : <expresión>·<intro>

y repitiendo:

<expresión> <intro> <expresión> <intro> <expresión> <intro>

Por lo que empleamos el esquema de diseño recursivo de caso base+recursividad:

S → <palabra base> | <palabra base> S

Por ejemplo:

```
axioma:      expresion '\n'         { printf ("Expresion=%lf\n", $1); }
      |      axioma expresion '\n' { printf ("Expresion=%lf\n", $2); }/* $2 */
      ;
```

Otra posibilidad:

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

También podemos intercalar la acción semántica con los token de las producciones:

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

La primera parece funcionar correctamente. Sin embargo, estamos aplicando recursividad por la izquierda, cosa que intentaremos evitar por lo general.

En el segundo caso se puede observar que el programa no responde hasta que no termina al introducir un error. Y la salida aparece en secuencia invertida respecto a las expresiones de entrada. Analizando la definición sintáctica-semántica de la gramática, observamos que la acción semántica está situada después de la recursividad sobre axioma. Esto conlleva a que la acción semántica no será evaluada nunca antes de que finalice el proceso de *parsing*, lo cuál sucede cuando aparece una entrada no contemplada. En ese momento se empezarán a aplicar las acciones semánticas en sentido inverso, según se deshace el bucle recursivo.

La tercera funciona pero imprime los resultados de forma desfasada, es necesario pulsar dos veces <intro> para obtener un resultado.

Recordemos que al tener dos producciones parcialmente coincidentes, se puede producir una *ambigüedad* funcional (No Determinismo). Esto sucede en nuestro ejemplo al adelantar la acción semántica para situarla antes de la recursividad. No es posible determinar qué producción debemos aplicar hasta que no se evalúa por completo, de ahí que tengamos que introducir un segundo <intro> para que el parser imprima la respuesta.

Si adelantamos la acción semántica para situarla antes del símbolo '\n', *bison* tendrá un conflicto debido al No Determinismo ya que le estamos pidiendo que evalúe una acción semántica sin saber qué producción debe aplicar. De ahí que genere un conflicto *shift/reduce*.

Lo que debemos aplicar en estos casos es una factorización, proceso en el que extraemos la parte derecha común, y el resto lo representamos con un no terminal distinto:

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

La parte común es:

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

La parte diferenciada es:

`<lambda>` en un caso, y `axioma` en otro

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

resto_axioma:       /* lambda */
             |      axioma
      ;
```

Incluso podemos simplificar admitiendo el caso de palabra vacía:

$(\text{<expresión>}\cdot\text{<intro>})^*$ en vez de $(\text{<expresión>}\cdot\text{<intro>})^+$

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

Para salir de la calculadora sin provocar un error de sintáxis, se podría incluir una producción que reconozca algún símbolo especial (por ejemplo, la letra q) y que

lleve asociada en la semántica la terminación del programa. Aunque esta no es una solución muy ortodoxa.

```
axioma:       /* lambda */
       |      expresion '\n'      { printf ("Expresion=%lf\n", $1); } axioma
       |      'q' '\n'            { exit(0); }
       ;
```

NOTAS:

Sucede con algunos conflictos (shift/reduce) que la máquina de pila puede operar, e incluso proporcionar el resultado correcto. Sin embargo, no es conveniente dejarlo así, porque al ir ampliando la gramática, se pueden ir propagando, multiplicando y complicando los errores hasta ser inabordables. Las causas para su aparición son la presencia de no determinismo y de ambigüedades en la gramática. Es muy conveniente transformar la gramática para evitar estos casos.

Observad la notación empleada para representar el código para *bison*. Es una notación cuidada para facilitar la detección visual de errores. Las herramientas *bison* no son capaces de detectar muchos errores que pasan fácilmente inadvertidos. Por ejemplo, si falta un punto y coma, o si se juntan dos caracteres, errores al indexar un token, etc.

Prestad atención a los operadores empleados para referenciar a los *tokens* desde las acciones semánticas.
Por ejemplo:

```
expresión:   operando '+' expresion  { $$ = $1 + $2; }
. . .
```

`$2` hace referencia al segundo símbolo de la producción, en este caso el literal '+', y en este caso dará un resultado erróneo o sin sentido.

```
expresión:   operando { printf ("%d", $1); } '-' expresion  { $$ = $1 - $4; }
. . .
```

`$4` hace referencia (correcta) al token expresión, ya que la acción semántica `{printf("%d",$1);}` se contabiliza como segundo token, y el literal `'+'` como tercero.

EJERCICIOS:

Introduce en calc1.y las modificaciones comentadas en el apartado 1.2, genera las correspondientes calculadoras y comprueba los resultados al introducir varias expresiones.

3.2 Sesión 2. Ampliación de la gramática

Que admita de forma sucesiva:
- más de una expresión.
- expresiones con paréntesis
- números enteros de más de un dígito.
- números enteros con signo.

Planteamos la gramática que describe el lenguaje propuesto, también descrita en ***Diag2***:

G2a

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

La implementación la hacemos por partes, para analizar los problemas que puedan surgir, partiendo de la modificación que va a ser habitual de expandir los *Operadores*:

*Expresión → Operando | Operando + Expresión | Operando – Expresión | Operando * Expresión | Operando / Expresión*

En primer lugar, la implementación para permitir más de una expresión (una por línea), parece sencilla. Bastaría con añadir a la gramática en *bison*:

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

Adviértase que en este caso la acción semántica está intercalada en la descripción sintáctica. Esto no sólo es posible, sino que es necesario con frecuencia. Al fin y al cabo, deseamos imprimir el resultado de la evaluación al pulsar el salto de línea y antes de pasar a la siguiente expresión. No obstante, si incluimos este añadido en la versión anterior de la calculadora, podremos comprobar que al probarla con la secuencia de sumas:

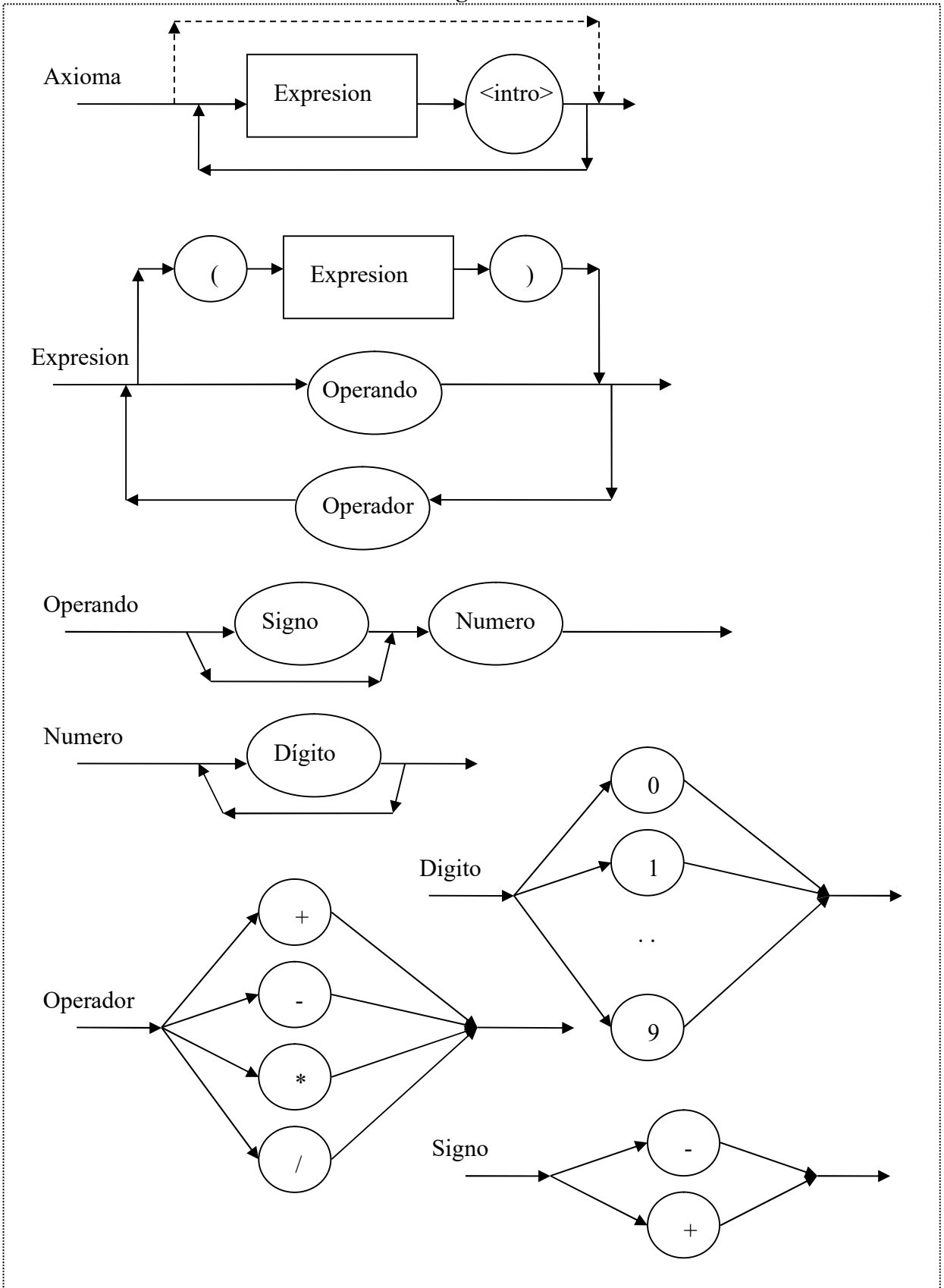
```
1+1<intro>
2+2<intro>              Expresion=2.000000
3+3<intro>              Expresion=4.000000
<intro>                 Expresion=6.000000
                    syntax error
```

la respuesta de la calculadora se producirá siempre con un retardo no deseado. El motivo por el que esto se produce se debe a que al intercalar la acción semántica en la gramática se produce una ambigüedad funcional: cuando la calculadora lee la secuencia `1+1<intro>` sabe resolverla perfectamente como *expresión*, pero dispone de dos ramas distintas para actuar (*expresión* única o *expresión* seguida de *axioma*). De ahí que postergue la acción hasta el momento de resolver el siguiente término que recibe por la entrada.

En el apartado anterior ya se trató brevemente la cuestión de la ambigüedad. De las tres soluciones se optó por la primera. Si planteamos la segunda solución (factorizar) adaptada al caso presente, tendremos:

Axioma → Expresión <intro> | Expresión <intro> RestoExpresión
RestoExpresion → Axioma| <lambda>

Diag2



Pero aquí se volvería a reproducir la ambigüedad funcional al intercalar las correspondientes acciones semánticas.

Por ello, elegiremos la tercera opción, que emplea <*lambda*>.
Axioma → *Expresión* <*intro*> *RestoExpresión*
RestoExpresion → <*lambda*> | *Axioma*

La implementación quedaría como sigue:

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

r_expr:                          /* lambda */
           | axioma
           ;
```

El símbolo correspondiente a <*lambda*> se implementa dejando una línea en blanco a continuación del *No Terminal* de la parte izquierda. Implementado y probado sobre la versión anterior de la calculadora, obtendríamos:

```
1+1<intro>              Expresion=2.000000
2+2<intro>              Expresion=4.000000
3+3<intro>              Expresion=6.000000
<intro>                 syntax error
```

en la secuencia adecuada.

También es válida la implementación mediante:

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

En este caso, se admitiría el uso del programa sin introducir una expresión. Pero al introducir sólo <intro> nos dará un error, no está del todo bien definida. Por eso optamos por la versión anterior. En el diagrama ***Diag2*** se representa mediante un flecha alternativa punteada.

La implementación de expresiones con paréntesis es más sencilla, sólo requiere añadir:

```
expresion:
…
           | '('  expresion  ')'      { $$ = $2 ; }
           ;
```

Para permitir números de más de un dígito, recurrimos a una definición recursiva:

Numero → *Digito* | *Digito Numero*
Digito → *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*

que implementaremos con:

```
numero:      digito                 { $$ = $1 ; pot = 1 ; }
           | digito numero          { pot *= 10 ; $$ = $1 * pot + $2 ; }
           ;

digito:      '0'                    { $$ = 0 ; }
…
           | '9'                    { $$ = 9 ; }
           ;
```

Aquí habremos encontrado una dificultad imprevista a la hora de reconstruir el valor del número que se define de forma recursiva. Una forma más natural sería emplear una definición con recursividad a izquierdas, pero en *bison* vamos a evitar usar estas estructuras. Por ello es necesario definir una variable (`pot`) que nos indique la potencia de diez con la que hay que sumar los dígitos al valor del número que se va construyendo. Es cierto que los dígitos más significativos son los primeros en ser leidos, pero la acción semántica situada después de la recursividad obliga a tratar primero los dígitos menos significativos. La solución es correcta, pero en el fondo es un artificio y sería preferible no tener que recurrir a ella. Más adelante veremos otra solución.

Hace falta añadir la definición de la variable (`pot`) en la primera sección:

```
%{                                  /* Seccion 1  Declaraciones de C-bison */
#include <stdio.h>
#define YYSTYPE  double
double pot ;
%}
```

-

Para permitir números con signo, podemos intentar ampliar el concepto de operando:

```
operando:    numero                 { $$ = $1 ; }
           | '-' numero             { $$ = -$2 ; }
           | '+' numero             { $$ = $2 ; }
           ;
```

Está solución puede ser válida, pero no muy elegante ya que puede crear conflictos como veremos en desarrollos futuros. En las pruebas podemos comprobar un comportamiento que parece el adecuado:

```
+1+23 <intro>           Expresion=24.000000
-1 <intro>              Expresion=-1.000000
-1-1 <intro>            Expresion=-2.000000
12—12 <intro>           Expresion=0.000000
-12- —12 <intro>        Expresion=0.000000
-12- +12 <intro>        Expresion=-24.000000
```

El conjunto del código desarrollado hasta aquí figura en la página siguiente (***calc2.y***).

calc2.y

```
%{                                  /* Seccion 1  Declaraciones de C-bison */
#include <stdio.h>
#define YYSTYPE  double
double pot ;
%}
                                    /* Seccion 2  Declaraciones de bison   */
%%
                                    /* Seccion 3  Gramática - Semántico   */

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

r_expr:                    /* lambda */
           | axioma
           ;

expresion:   operando                { $$ = $1 ; }
           | operando '+' expresion  { $$ = $1 + $3 ; }
           | operando '-' expresion  { $$ = $1 - $3 ; }
           | operando '*' expresion  { $$ = $1 * $3 ; }
           | operando '/' expresion  { $$ = $1 / $3 ; }
           | '('  expresion  ')'     { $$ = $2 ; }
           ;

operando:    numero                  { $$ = $1 ; }
           | '-' numero              { $$ = -$2 ; }
           | '+' numero              { $$ = $2 ; }
           ;

numero:      digito                  { $$ = $1 ; pot = 1 ; }
           | digito numero           { pot *= 10 ; $$ = $1 * pot + $2 ; }
           ;

digito:      '0'                     { $$ = 0 ; }
           | '1'                     { $$ = 1 ; }
           | '2'                     { $$ = 2 ; }
           | '3'                     { $$ = 3 ; }
           | '4'                     { $$ = 4 ; }
           | '5'                     { $$ = 5 ; }
           | '6'                     { $$ = 6 ; }
           | '7'                     { $$ = 7 ; }
           | '8'                     { $$ = 8 ; }
           | '9'                     { $$ = 9 ; }
           ;
%%
                                    /* Seccion 4  Código 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 () ;
}
```

CUESTIONES ABIERTAS:

2.1 Observa el resultado de diversas expresiones como:
2*3+1
1+2*3
2+3*1
1*3+2
1-1-1
1-1-1-1
1-1-1-1-1
1-1-1-1-1-1
1-2-3-4-5

¿Sábes determinar a qué se deben los resultados obtenidos?

2.2. Existe un problema en la solución desarrollada hasta ahora:

```
1 2 3 + 2  1 <intro>
Expresion=144.000000
```

¿A qué se debe esto? ¿Qué soluciones se te ocurren? Analiza el código del analizador léxico (*yylex*).

2.3. Hay un pequeño fallo tal como está definido *expresion*. ¿Sabes en qué casos aparece? ¿Y a qué es debido? Prueba diversos tipos de expresiones.

2.4. Se ha diseñado la definición de *numero* como:

```
numero:      digito                  { $$ = $1 ; pot = 1 ; }
           | digito numero           { pot *= 10 ; $$ = $1 * pot + $2 ; }
           ;
```

Usando una estructura recursiva similar a la que habíamos empleado inicialmente para *axioma (Caso Base, Baso Base + Recursividad).*:

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

Sin embargo, en el caso de axioma se tuvo que descartar por los problemas que causaba.
¿Sabes decir por qué vale en el primer caso y no en el segundo?