3.2.1 Cuestiones resueltas de la Sesión 2

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?

La conclusión de que los resultados son aparentemente erróneos se basa en que estamos acostumbrados a que los lenguajes y compiladores apliquen ciertas reglas y precedencias al evaluar expresiones.
En este caso la explicación es sencilla. Todas las expresiones se evaúan de derecha a izquierda. En otras palabras, todos los operadores tienen la misma precedencia y se aplica la asociatividad de derecha a izquierda. Esto se debe a la definición de *expresion*.

Por ejemplo:

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

La acciones semánticas están situadas al final de la las producciones, lo cuál quiere decir que antes de aplicar dichas acciones, deben resolverse todos los términos previos. Es decir, hasta que no se resuelve el último término de *expresion* el parser no operará con ellos. Y llegado a ese punto, aplicará las operaciones retrocediendo en el descenso recursivo. De ahí que las operaciones se calculen de derecha a izquierda.

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*).

Hay unas líneas en el analizador léxico que determinan que los espacios en blanco se van a ignorar siempre que aparezcan:

```
int yylex ()
…
    do {
         c = getchar () ;
    } while (c == ' ') ;
…
}
```

Esto sería la explicación a bajo nivel.
A otro nivel, la explicación está en que el espacio en blanco pertenece al lenguaje que debemos reconocer/procesar, pero no lo recogemos de forma explícita en la gramática.

En la práctica la función del carácter espacio en blanco suele ser la de separador, lo cuál quiere decir que sirve para delimitar determinados símbolos del lenguaje (números, identificadores, etc.). En este desarrollo hemos optado por ignorar los espacios, porque es una solución sencilla. Debido a esto una expresión como la planteada será interpretada como una expresión válida. Quien diseñe el intérprete/compilador tiene que decidir si esto tiene sentido. La interpretación lógica sería asumir que dígitos separados por espacios deben ser tratados como números diferentes. Y si falta un operador entre ellos, dar error.
Solución 1: Incluir el espacio en blanco dentro del alfabeto y añadirlo como un símbolo más dentro de la gramática: incluirlo sólo en aquellas producciones donde pueda aparecer:

```
espaciado:                  /* lambda */
          | ' ' espaciado
          ;
     . . .
expresion:    operando espaciado '+' espaciado expresion
```

Lo más seguro es que lleguemos a la conclusión de que resulta tedioso y poco práctico, sobre todo de cara a aplicarlo en el nivel sintáctico o una herramienta como *bison*.

Solución 2: Hacer un analizador lexicográfico algo más elaborado y sofisticado. Lo veremos a partir de la tercera sesión de la práctica guiada.

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

Seguramente habréis detectado más de un fallo, dado que la gramática todavía no está muy elaborada.
La gramática no permite evaluar expresiones como:

```
(1 + 2) + 3 <intro>
```

Al reformar la gramática para eliminar la doble recursividad (a izquierda y derecha) se ha obviado este caso para simplificar. El primer elemento de una expresión se considera que debe ser un operando, lo cuál parece un poco limitante.

Una solución posible para el primer problema es trasladar la producción de expresión:

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

a:

```
operando:    numero                 { $$ = $1 ; }
. . .
          | '('  expresion  ')'     { $$ = $2 ; }
          ;
```

Otros problemas detectados son que no se contemplan casos como:

```
+(3) <intro>
```

Ni expresiones vacías (solo <intro>).

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 un caso y no en otro?

Si analizamos la gramática a secas, veremos que ambas partes siguen la misma estructura:

PI ::= <terminos>
PI ::= <terminos> PI

¿Cuál es la diferencia que hace que esta estructura valga para un caso y no para el otro?
Se trata de las acciones semánticas (el código C entre llaves) asociadas a cada producción. En el primer caso se sitúan al final de la producción, mientras que en el segundo están intercaladas con los terminos de la producción. El intercalado provoca en el segundo caso una ambigüedad funcional. Al llegar a la acción semántica, el parser no tiene forma de saber en qué rama de la gramática se debe situar. De hecho, no lo puede resolver hasta que no resuelva los términos que aparecen después de la acción semántica. Este es el motivo por el que algunos habréis observado que la primera versión del programa evalúa correctamente las expresiones, pero responde sólo cuando se ha introducido la siguiente expresión (responde con un retraso de una expresión).

3.3 Sesión 3. Descomposición en niveles *sintáctico* y *léxico*

El objetivo es:
- Descargar la fase de análisis sintáctico.
- Aumentar el nivel léxico mejorando la función *yylex*.
- Admitir números con signo

Podríamos proseguir indefinidamente introduciendo mejoras y funciones más avanzadas en el desarrollo de la calculadora, pero en algún momento apreciaríamos una creciente ralentización en el tiempo de operación. Para evitar esto se propone descargar parte del trabajo que debe realizar el analizador sintáctico. Para ello potenciamos el nivel léxico, creando un analizador léxico capaz de asumir funciones que hasta ahora hemos delegado en el sintáctico.

Sabemos que el lenguaje que genera la gramática ***G2*** tiene un alfabeto compuesto por los símbolos +, -, *, /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,) e <intro>, y el espacio en blanco como separador. Podemos dividir la gramática en dos secciones, léxico y sintáctico, de forma que los anteriores símbolos sigan formando el alfabeto del primero. La sección sintáctica tendría un nuevo "alfabeto" compuesto por los símbolos *No Terminales* que define la sección léxica, en este caso, *Numero, Operador* y *Signo*. Esta división viene recogida en la gramática ***G3a***. Observaremos siempre que las producciones del nivel léxico deben ser de tipo 3 Lineal Derecha, o generar un sublenguaje representable mediante expresiones regulares.

En términos de *bison*, estos nuevos símbolos se van a denominar *token*. Es importante tener presente que cada *token* que sea enviado desde el nivel léxico al sintáctico debe llevar asociado un valor. En el caso de *Numero*, el valor numérico, en el caso de *Operador*, el tipo de operador, en el caso de *Signo*, el tipo de signo.

G3a

[Nivel Sintáctico]
Axioma → Expresion <intro> R_axioma
R_Axioma → <lambda> | Axioma
Expresion → Operando | Operando Operador Expresion
Operando → Numero | Signo Numero | (Expresion)

[Nivel Léxico]
Numero → Digito | Digito Numero
Operador → + | - | * | /
Signo → - | +
Digito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Para establecer la comunicación entre el *analizador léxico* y el *sintáctico* existen una serie de recursos:
- Se definen los *token*
- Se envían del *léxico* al *sintáctico* los *token* mediante un `return`.
- Se almacena en una variable específica, *yylval*, el valor asociado al *token*.
- En algunos casos el léxico envía caracteres sueltos (*literales)* al sintáctico (es el caso de los paréntesis o <intro> en G3a).

Por cuestiones técnicas por ahora sólo vamos a incluir en el nivel léxico las producciones correspondientes a *Numero.*

G3b

[Nivel Sintáctico]
Axioma → Expresion <intro> R_axioma
R_Axioma → <lambda> | Axioma
*Expresión → Operando | Operando + Expresión | Operando – Expresión | Operando * Expresión | Operando / Expresión*
Operando → Numero | + Numero | - Numero | (Expresión)

[Nivel Léxico]
Numero → Digito | Digito Numero
Digito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Obviamente, los símbolos +, -, *, /, (,) e <intro> serán transmitidos como *literales* del *analizador lexicográfico* al *sintáctico*.

Aunque el nivel *léxico* se ha vuelto a descargar, la complejidad del *sintáctico* no aumentará en exceso, ya que sólo tendrá que lidiar con literales aislados que no forman parte de definiciones recursivas. Aunque la propuesta de descargar el nivel sintáctico era buena, veremos que a veces priman cuestiones técnicas a la hora de definir el límite con el nivel léxico. En este caso hay varios motivos para optar por la versión G3b. Uno de ellos es que en *G3a* el analizador léxico tendría que resolver a la hora de leer un signo '-' si se trata de un *signo* o de un *operador* y eso no es tan sencillos si no recurrimos al contexto. Este caso concreto resulta más sencillo de resolver en el nivel sintáctico. En el diseño de nuestro intérprete/compilador siempre tendremos en cuenta que algunos *No Terminales* se pueden resolver con mayor o menor facilidad en el nivel sintáctico o léxico, y optar por una de las dos opciones.

La definición de los *token* se realiza en la segunda sección del fichero para *bison*:

```
%{                                  /* Seccion 1  Declaraciones de C-bison */
#include <stdio.h>
#define YYSTYPE  double             /* tipo de la pila del parser          */
%}
%token NUMERO               /* Seccion 2  Declaraciones de bison   */
```

La obtención del *token* `NUMERO` se realiza en `yylex()`*:*

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

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

    if (c >= '0' && c <= '9') {
         ungetc (c, stdin) ;
         scanf ("%d", &valor) ;
         yylval = (double) valor ;
         return NUMERO ;
    }

    return c ;
}
```

Se da por supuesto que cualquier símbolo que comience por un dígito será un número entero. La función `ungetc` devuelve el dígito leído para poder emplear una lectura única del número. La función `scanf("%d"…` tiene la particularidad de leer la secuencia de dígitos completa hasta la aparición de un carácter que no pueda pertenecer a un número. Para ser consecuentes con nuestra definición actual de la gramática (sólo se admiten números enteros), la lectura se hace mediante un valor auxiliar que luego es asignado mediante un cast `(double)` a la variable `yylval` (la pila del parser está definida con el tipo YYSTYPE double). En este caso, yylex() devuelve a yyparse() el identificador del token NUMERO.

Para cualquier otro carácter, se devuelve el literal mediante un return.

La única modificación que requiere la sección sintáctica es la inclusión de las nuevas definiciones de *operando* y *numero*:

```
operando:    NUMERO                  { $$ = $1; }
           | '('  expresion  ')'      { $$ = $2; }
           | '-' NUMERO              { $$ = -$2; }
           | '+' NUMERO              { $$ = $2; }
           ;
```

Se recomienda repasar el diagrama sintáctico y el código completo transcritos en las siguientes páginas.

Con estas modificaciones quedaría terminada la propuesta para este apartado.

-

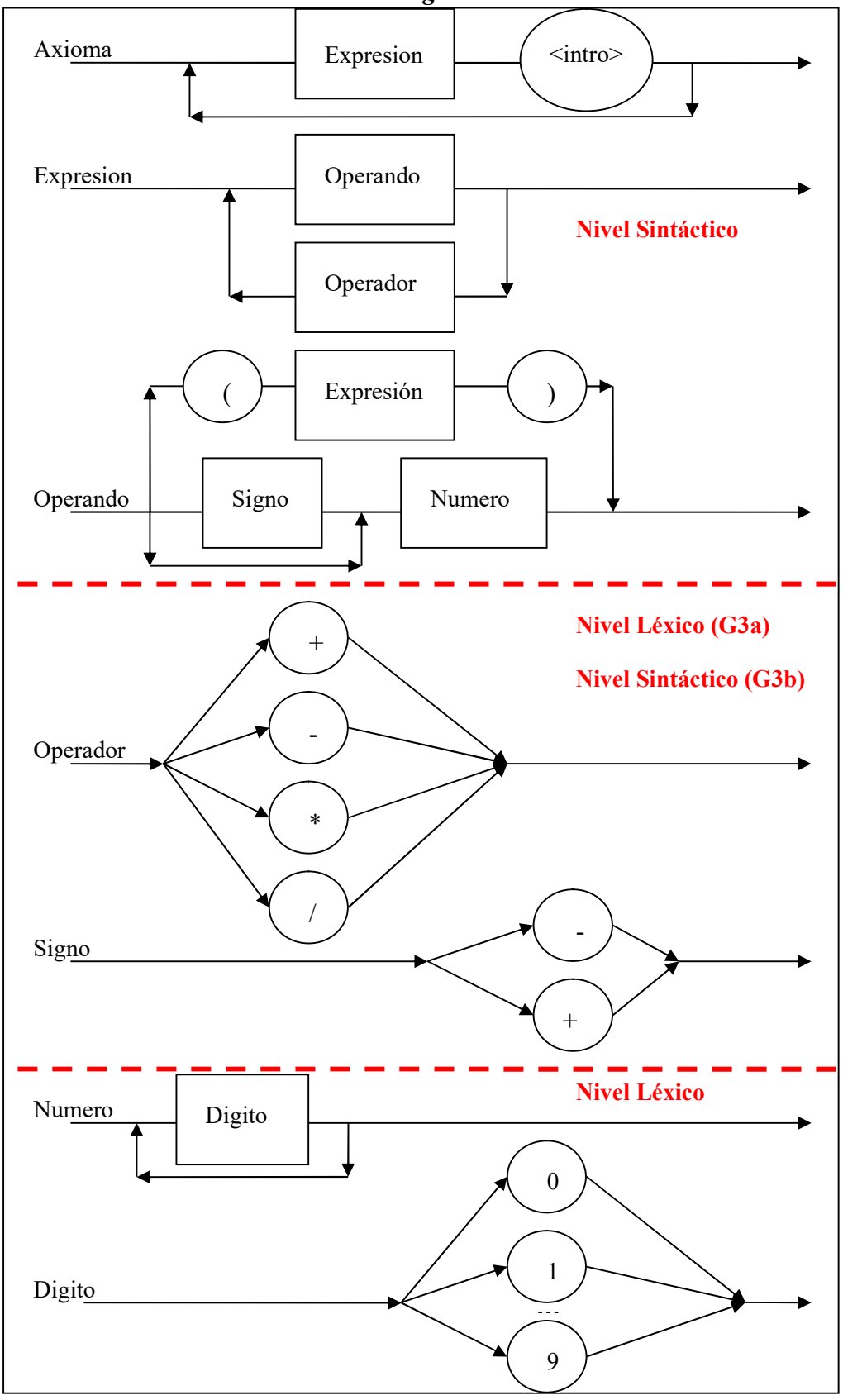
CUESTIONES ABIERTAS:

3.1 Adapta la gramática para que las expresiones encerradas con paréntesis puedan ir precedidas de un signo negativo (o positivo). Indicar sólo las producciones nuevas necesarias. Se recomienda probar la solución con *bison*. ¿Funciona para la expresión –(–3)?

3.2 Adapta la gramática para reconocer números reales (ejemplos: 00.33, 0.33, .33, 100, 0., 100.33). Representa las nuevas producciones de la gramática en BNF y en forma de Diagrama Sintáctico. No se pide ni código C ni acciones semánticas. Tampoco se pide diferenciar nivel léxico o sintáctico, sólo incluir las producciones necesarias para reconocer números reales (como los indicados como ejemplo).

3.3 Revisa la gramática de *calc3.y* (represéntala con diagramas sintácticos) y contrástala con el diagrama *diag3*. ¿Ves alguna diferencia, incluso sutil? ¿Los lenguajes generados o reconocidos por ambas representaciones difieren en algo? ¿Ves algún problema en implementar el analizador sintáctico, junto con su semántica siguiendo el diagrama?

Diag3[2]



[2] Este tipo de diagramas se denominan en origen como Diagramas de Conway o Diagramas Sintácticos, pero conviene hacer notar que sirven para representar tanto el nivel sintáctico como el léxico.

Calc3.y

```
%{                                  /* Seccion 1  Declaraciones de C-bison */
#include <stdio.h>
#define YYSTYPE  double             /* tipo de la pila del parser         */
%}
%token NUMERO                       /* Seccion 2  Declaraciones de bison   */
%%
                                    /* Seccion 3  Sintactico - Semantico   */

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; }
           ;

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

%%
                                    /* Seccion 4  Codigo en C   */

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


int yylex ()
{
    unsigned char c ;
    int valor ;

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

    if (c >= '0' && c <= '9') {
         ungetc (c, stdin) ;
         scanf ("%d", &valor) ;
         yylval = (double) valor ;
         return NUMERO ;
    }

    return c ;
}


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