3.3.1 Cuestiones Resueltas de la Sesión 3

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)?

Hay varias posibilidades. La más rápida es duplicar las producciones para *expresion*:

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

Otra opción sería resumir mediante *signo*:

```
operando:    signo '(' expresion ')' { if ($1 == -1)    /* una forma */
                                         $$ = -$3;
                                       else
                                         $$ = $3; }
           | signo NUMERO            { $$ = $2 * $1; }  /* otra forma */
           ;

signo:       /* lambda */            { $$ = 1;  /*  1 identifica suma  */ }
           | '-'                     { $$ = -1; /* -1 identifica resta */ }
           | '+'                     { $$ = 1;  /*  1 identifica suma  */ }
           ;
```

En esta opción podría discutirse la elegancia porque con los recursos de que disponemos por ahora, nos vemos obligados a representar con el tipo de la pila *bison* (en este caso YYSTYPE=double) los casos posibles de signos. Incluimos esta solución para apuntar la posibilidad de descargar el nivel sintáctico en la semántica.

Una opción más estructurada sería poner en común *expresion* y *NUMERO* como *resto_operando*:

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

resto_operando:
             NUMERO                  { $$ = $1; }
           | '('  expresion  ')'      { $$ = $2; }
           ;
```

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).

Antes de ponerse a reformar la gramática, conviene tener algunas ideas claras. En este caso podemos a) retocar definiciones ya hechas, o b) incluir nuevas categorías. Optamos por b), la definición de NUMERO es válida y podemos basarnos en ella para obtener la de un número *real*. Consideramos que un número real consta de tres partes (parte entera, punto, parte decimal) de las cuáles parte entera y parte decimal se pueden resolver como

números. Siguiendo los ejemplos mostrados, estas tres partes pueden ser opcionales en diversas combinaciones.

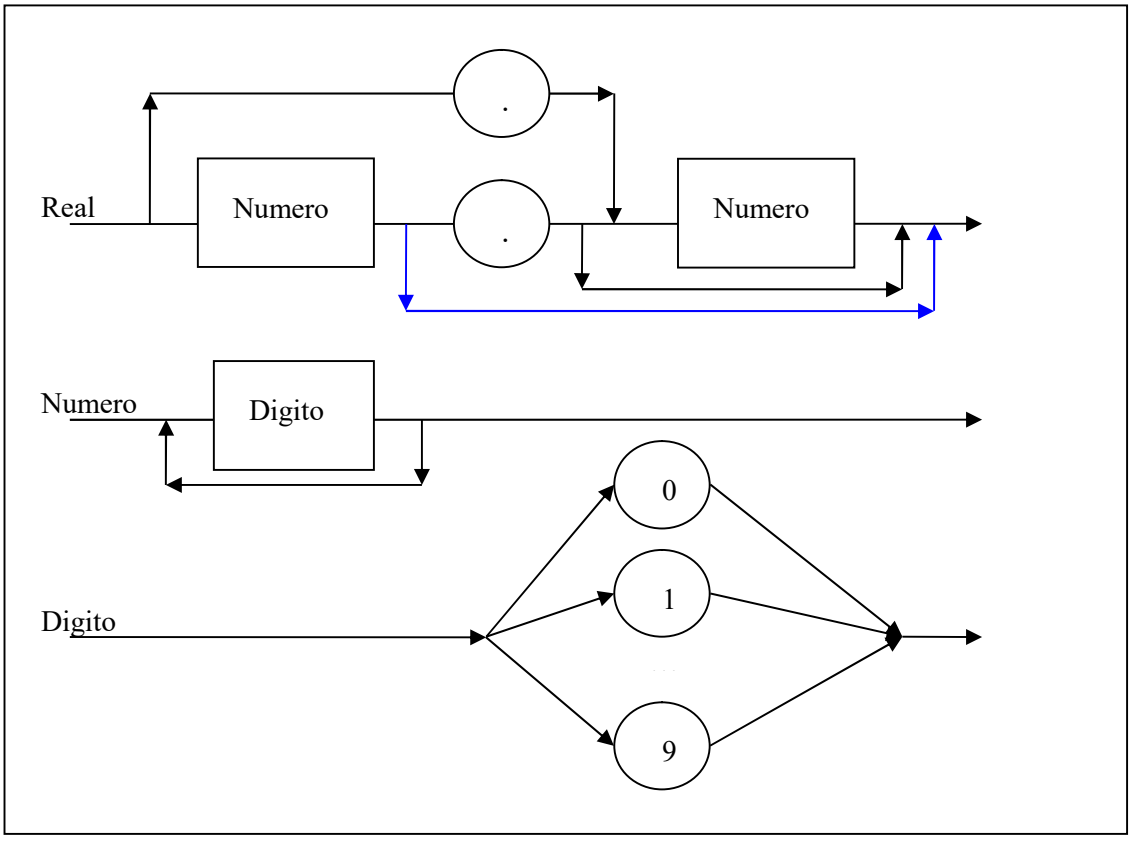
```
real ::= NUMERO '.' | NUMERO '.' NUMERO | '.' NUMERO | NUMERO
```

Podemos observar que los números reales se pueden representar mediante una expresión regular, lo cuál es un indicio de que se pueden resolver mediante el analizador lexicográfico:

$$[0-9]^{+}.[0-9]^{*} + .[0-9]^{+} + [0-9]^{+}$$

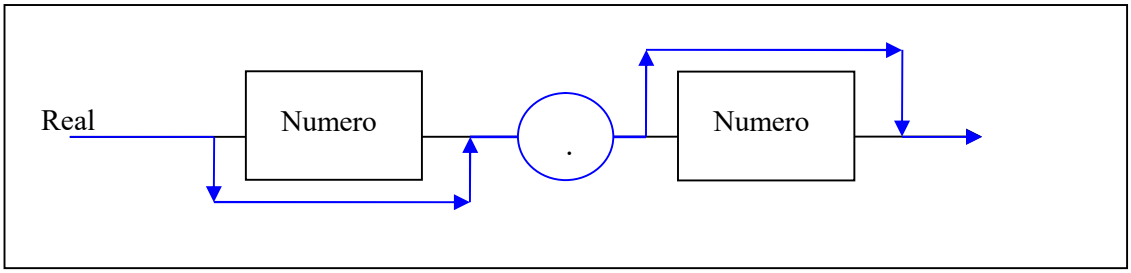
En este caso el símbolo . se interpreta como punto literal.

En el diagrama podemos resumir las combinaciones de la parte entera y la decimal de la siguiente forma:



La bifurcación en azul se incluye para contemplar casos de números enteros (por ejemplo, 100).

Otra solución como la siguiente tiene el problema de incluir un caso que en principio no se contemplaba (un punto decimal solitario obtenido con el recorrido marcado en azul). Y excluye la posibilidad de definir un número entero.



El objetivo de los ejercicios anteriores es hacer ver que a la hora de diseñar una gramática, y sobre todo, al ampliarla, hay que tener muy en cuenta que pueden aparecer casos que inicialmente no estaban previstos y que pueden dar origen a fallos o lagunas en nuestro desarrollo.

Si queremos resolver esto mediante producciones en el analizador sintáctico, tedríamos que prescindir del token `NUMERO`, ya que además yylex nos devuelve el número completo, incluido el punto decimal. Aquí nos limitamos a plantear una solución que habría que adaptar al resto de la gramática. Dado que `NUMERO` deja de ser un token, habrá que resolverlo mediante otros token o símbolos, en nuestro caso digitos:

```
numero:      n_entero '.' n_decimal     { $$ = $1 + $3 ; }
           ;

n_entero:    dig                        { $$ = $1 ; pot = 1 ; }
           | dig n_entero               { pot *= 10 ; $$ = $1 * pot + $2 ; }
           ;

n_decimal  : dig                        { $$ = $1 / 10.0 ; }
           | dig n_decimal              { $$ = ($1 + $2) / 10.0 ; }
           ;
```

Se deja al alumno la corrección de los errores que puede haber, y la integración con el resto de la gramática.

Una solución como:

```
numero:      n_entero '.' n_entero      { $$ = $1 . $3 ; }
           ;
. . .
```

No es válida porque el punto ni sive para concatenar variables ni es un operador válido en C, y recordamos, en la semántica se debe trabajar con código en C. Sí sería una solución factible concatenar los dos números enteros separados por punto y luego leerlos como real con sscanf. Aproximadamente:

```
numero:      n_entero '.' n_entero      { sprintf (temp, "%d.%d", $1, $3) ;
                                          sscanf (temp, "%lf", &aux) ;
                                          $$ = aux ; }
           ;
```

En la cuarta parte de la práctica guiada veremos una solución adicional.

3.3 ¿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?

-

-

Las diferencias más visibles atañen a la forma de resumir las operaciones +, -, *, / mediante una caja (*operador*) cuando en la gramática están desglosadas en otras tantas producciones. También los signos + y – se tratan como literales en producciones duplicadas cuando en el diagrama se resumen en la caja *signo*.
Si comparamos el diagrama con el código veremos que *Axioma* se resuelve con un único diagrama mientras que en el código se emplea un *No Terminal* auxiliar.

En todos los casos, la representación del diagrama y en BNF son equivalentes, y reconocen la misma gramática. Pero sí aprovechamos para recordar que la diferencia en el axioma, la introducción de `r_expr` se debía a la ambigüedad funcional causada por la inserción de la semántica. Obviamente, los diagramas no representan la parte semántica, con lo cuál aparentemente son correctos. En la práctica, es necesario transformar las producciones.

En el caso de los operadores que están representados con su propio No Terminal (Operador), podemos implementarlo sin problema en la gramática. Obtendríamos una factorización de cuatro producciones. Pero tendremos más dificultad para aplicar la semántica apropiada a cada caso.

3.4 Sesión 4. Mejoras en la calculadora

Inclusión de:
- número reales
- precedencia de operador
- mejoras en la detección de errores
- generar un analizador léxico con flex

La gramática requiere modificaciones para admitir los números con punto decimal. En este caso vamos a suponer que se van a admitir números como:

```
100.33, 00.33, .33, 100, 100.
```

Pero no podrá aparecer:

```
un punto aislado
```

G4

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

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

Se ha introducido un nuevo *No Terminal* (*Decimal*) para resolver la parte decimal. Tanto *Numero* como *Decimal* comparten la misma estructura, pero al primero se le añade la alternativa de completar el numero con una parte decimal.

Las nuevas producciones se han incluido en este caso en el dominio del nivel léxico, puesto que se pueden resolver de forma sencilla en el lenguaje C:

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

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

    if (c == '.' || (c >= '0' && c <= '9')) {
         ungetc (c, stdin) ;
         scanf ("%lf", &yylval) ;
         return NUMERO ;
    }
…
}
```

En la entrega anterior, ya se resolvió el uso de números con signo, aunque se advirtió del peligro de posibles confusiones con los operadores aritméticos. Existe otra solución posible en *bison*, que además es la más apropiada. Se basa en definir precedencias entre operadores. Sabemos que algunas operaciones deben ser realizadas de forma prioritaria sobre otras. Tal es el caso de las operaciones * y / que tienen mayor precedencia que la

suma y la resta. A su vez, los signos unarios + o – deben tener mayor precedencia que los demás operadores. La forma de definir estas relaciones se muestra a continuación:

```
%token NUMERO                /* Seccion 2  Declaraciones de bison   */
%left  '+' '-'   /* menor orden de precedencia */
%left  '*' '/'   /* mayor orden de precedencia */
%left  SIGNO_UNARIO /* define la mayor precedencia, ademas de nuevo token */
%%
```

Y añadimos a la sección gramatical:

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

Otra posible solución es:

```
expresion:   operando                { $$ = $1; }
           | '+' operando %prec SIGNO_UNARIO          { $$ = $2; }
           | '-' operando %prec SIGNO_UNARIO          { $$ = -$2; }
           | operando '+' expresion  { $$ = $1 + $3; }
. . .
operando:    NUMERO                                  { $$ = $1; }
           | '('  expresion  ')'                      { $$ = $2; }
           ;
```

Ahora mismo no es posible juzgar cuál de las dos es preferible, pero sí veremos en el futuro que si deseamos incluir signos unarios antes de una variable, al igual que delante de un número, se realizará de forma más práctica con la segunda solución.

Mejoramos la detección de errores llevando la cuenta de las líneas leídas para imprimir un poco más de información útil.

```
int n_linea = 1 ;

int yyerror (mensaje)
char *mensaje ;
{
    fprintf (stderr, "%s en la línea %d\n", mensaje, n_linea) ;
}

int yylex ()
{
…
    if (c == '\n')
          n_linea++ ;

    return c ;
}
```

Herramienta *flex*

Otra forma de resolver el análisis léxico es empleando una herramienta específica. Aunque hasta ahora la dificultad de programar este analizador es reducida, puede suceder que deseemos abordar problemas mucho más complejos descargando el nivel sintáctico en el léxico. En tal caso puede que sea necesario identificar *token* con una estructura mucho más compleja y resulte incómodo de abordar mediante las estructuras de control típicas de un lenguaje de programación. Si somos capaces de representar los token mediante expresiones regulares, una herramienta como *lex* o *flex* nos facilitará en gran medida el trabajo.

En el transcurso de la práctica aquí propuesta la complejidad del nivel léxico no va a aumentar excesivamente, así que se planteará siempre una función a medida en C y se dejará al alumno la creación del analizador equivalente en *flex*.

Para emplear *flex* crearemos un nuevo fichero (calc4.lex) en el que tendremos varias secciones. En el caso básico, una primera con declaraciones de ficheros cabecera, variables externas, etc que sean necesarios para el código generado por *flex* y para comunicar con el fichero generado por *bison*. En la segunda sección se consignarán, una por línea todas las expresiones regulares que identifiquen un *token* particular, y entre llaves, las acciones en C asociadas. Una tercera sección permite incluir código C.

Calc4.lex

```
%{                              /* Primera Seccion */
#include "calc4.tab.h"          /* Cabecera con decl. de token etc. */
extern int n_linea;
%}
%%                              /* Segunda Seccion */
[ \t]                       { ; } /* ignorar espacios y tabuladores */
[0-9]+\.?|[0-9]*\.[0-9]+ { sscanf (yytext, "%lf", &yylval); return (NUMERO); }
\n                          { n_linea++; return ('\n'); }
.                           { return (yytext [0]); }   /* literales */
%%                              /* Tercera Seccion */
int yywrap ()     /* se incluye para evitar un error de compilacion */
{
    return (0) ; /* para el control de fin de fichero */
}
```

Cosas a tener en cuenta:

`[0-9]` es equivalente a 0+1+2+3+4+5+6+7+8+9, es decir **un** dígito del rango 0-9.

`.` quiere decir cualquier carácter.

`\.` quiere decir punto literal, en el ejemplo se refiere a un punto decimal.

`\n \t` son los literales de salto de línea y tabulador.

`+ * ?` son indicadores de ocurrencia (de 1 a n veces, de 0 a n veces, 0 o 1 vez).

`|` es la barra alternativa, representa al + que estamos acostumbrados de EERR

`yytext` es una variable tipo (char *), equivalente a un string que contendrá el flujo de entrada. Operaremos sobre ella para obtener los *token*.

`yytext[0]` será por tanto el primer carácter del flujo de entrada pendiente de leer.

A la hora de poner las expresiones regulares en el fichero .lex conviene observar una estricta secuencia. Para evitar solapes entre una expresión regular y las posteriores.

Por ejemplo, no tendría sentido consignar:

```
.           { . . . }
[0-9]       { . . . }
```

El punto literal cubre cualquier carácter incluidos los dígitos 0-9, con lo que la segunda línea sería inoperativa.

La definición de la función *yywrap* se incluye ya que no todas las versiones de *flex* disponen de una por omisión. Su misión consiste en definir el tipo de comportamiento necesario al detectarse un fin de fichero en el analizador lexicográfico. Es útil de cara a leer una secuencia de ficheros.

Para la integración con el fichero calc4.y bastará con suprimir en este la función yylex (mejor la ponemos con comentarios), puesto que *Flex* se encargará de generar dicha función. Crearemos en primer lugar el código del parser:

```
bison calc4.y –d
```

La opción –d fuerza a *bison* a generar el fichero de cabeceras calc4.tab.h necesario en calc4.lex para disponer de la definición de los *token*, tipo de pila, etc. Que serán necesarias para el analizador léxico. Después procesaremos con flex:

```
flex calc4.lex
```

Con lo que obtenemos un fichero lex.yy.c

Antes de compilar, comprobad qué ficheros os han generado las herramientas. Deberían ser calc4.tab.h, calc4.tab.c y lex.yy.c. Pero en algunas versiones se generan con otros nombres.

En función de eso compilaremos el conjunto con:

```
gcc lex.yy.c calc4.tab.c –o calc4
```

o con:

```
gcc y.lex.c y.tab.c –o calc4
```

CUESTIONES ABIERTAS:

4.1 Prueba las siguientes expresiones para comprobar el funcionamiento de la precedencia. ¿a qué conclusión llegas?

2*3+1 2+3*1 1+3*2 1*3+2

4.2 Reforma la gramática sustituyendo el *no terminal operando* por el de *expresion* en aquellas producciones que contengan una operación binaria. Recuerda que habíamos introducido operando para evitar la ambigüedad debida a la doble recursivdad (expresion→expresion+expresion). Prueba de nuevo las expresiones anteriores.

4.3 Si nos plantearan usar variables como operandos dentro del lenguaje, ¿qué cambios plantearías en la gramática? Partimos de la idea de que las variables estarán identificadas por una secuencia alfanumérica, siendo el primer carácter una letra. No se contempla la definición previa de variables.

4.4 Incluye la operación de asignación de una expresión a una variable en la gramática.

4.5 Intenta adaptar el analizador lexicográfico correspondiente.

4.6 ¿Qué problema ves a la hora de implementar un analizador sintáctico que sea totalmente operativo (a nivel semántico) con las últimas cuestiones propuestas?

calc4.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      */
%left '+' '-'     /* menor orden de precedencia */
%left '*' '/'     /* orden de precedencia intermedio */
%left SIGNO_UNARIO /* define la mayor precedencia, ademas de nuevo token */
%%
                                    /* Seccion 3 Gramatica - 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                                    { $$ = $1; }
            | '+' NUMERO %prec SIGNO_UNARIO             { $$ = $2; }
            | '-' NUMERO %prec SIGNO_UNARIO             { $$ = -$2; }
            | '(' expresion ')'                         { $$ = $2; }
            ;
%%
                                        /* Seccion 4  Codigo en C    */
int n_linea = 1 ;

int yyerror (char *mensaje)
{
    fprintf (stderr, "%s en la linea %d\n", mensaje, n_linea) ;
}

/* suprimir la funcion yylex () si se usa flex */
///*
int yylex ()
{
    unsigned char c ;

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

    if (c == '.' || (c >= '0' && c <= '9')) {
         ungetc (c, stdin) ;
         scanf ("%lf", &yylval) ;
         return NUMERO ;
    }

    if (c == '\n')
           n_linea++ ;

    return c ;
}
//*/

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