Desarrollo de un pequeño Intérprete Descendente Recursivo

En esta práctica guiada vamos a abordar el diseño de un Intérprete muy elemental y con recursos básicos para repasar los principales conceptos de un Analizador Sintáctico Descendente Recursivo. Para no tener que lidiar con una gramática grande y complicada, vamos a restringir el dominio a la típica calculadora de expresiones aritméticas. De esta forma podremos ver resultados con pocas producciones.

Empezaremos con un planteamiento muy elemental, para ir complicándolo en sucesivos pasos:

1. Parser de operaciones muy simples.
2. Calculadora de operaciones muy simples (Parser + Acciones Semánticas).
3. Inclusión de expresiones con paréntesis.
4. Inclusión de precedencia de operador, y signos unarios.

1. Parser de operaciones muy simples.

El planteamiento más sencillo corresponde a una secuencia de operandos enteros y operadores aritméticos (+, -, * /) terminada por un salto de línea, sin tener en cuenta precedencias, ni asociatividad. Tampoco buscaremos obtener el resultado numérico de la expresión. Sólo queremos comprobar la corrección sintáctica de la entrada.
La gramática que podemos plantear inicialmente sería:

```
Expression ::=   Expression + Expression |
                 Expression - Expression |
                 Expression * Expression |
                 Expression / Expression |
                 Number
```

Donde *Number* es un *token* que representa un valor entero.

Sabemos que en la gramática de un parser descendente recursivo no se permite la recursividad por la izquierda. Podemos aplicar el algoritmo para eliminarla, pero optamos por rediseñarla:

```
Expression ::=   Term + Expression |
                 Term - Expression |
                 Term * Expression |
                 Term / Expression |
                 Term

Term ::=         Number
```

Tampoco se permite para un mismo No Terminal derivar en varias producciones con el mismo comienzo (que conlleva una indeterminación[1]). Por eso factorizamos las producciones aritméticas:

```
Expression ::=        Term ExpressionRest

ExpressionRest ::=    + Expression |
                      - Expression |
                      * Expression |
                      / Expression |
                      Lambda

Term ::=              Number
```

Vemos que ExpressionRest tiene varias producciones alternativas, y además, deriva lambda. En el primer caso quiere decir que para decidir qué producción aplicar tendremos que decidir en base al token/lexema leído en el momento dado. Debe pertenecer al conjunto PRIMERO del símbolo ExpressionRest. En este caso es fácil ver qué Terminales pertenecen a éste. PRIMERO(ExpressionRest) = { + , - , * , / , ␣ } puesto que son los primeros símbolos de cada una de las cinco producciones alternativas.

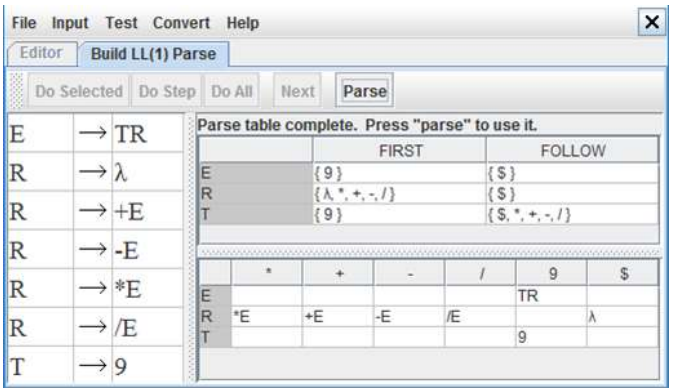
Puesto que ExpressionRest deriva *lambda*, se trata de un símbolo anulable. Concretamente sirve para generar una secuencia finita de pares operador-operando. Para detectar cuándo termina sta secuencia necesitaremos saber cuál es el conjunto SIGUIENTE(ExpressionRest). Este conjunto contendrá únicamente el símbolo final de la secuencia. No se suele representar en la gramática inicial, pero se suele incluir luego representado por $. En este caso habíamos definido que la secuencia aritmética termina con un salto de línea. Eso es lo que representará el símbolo $.

Podemos recurrir a herramientas para calcular los conjuntos PRIMERO y SIGUIENTE. Por ejemplo, podemos introducir la gramática en JFLAP. Representamos:

```
Expression      □    E
ExpressionRest  □    R
Term            □    T
Number          □    9
```

Seleccionamos <Input>→<Build LL(1) Table> y obtenemos entre otros los conjuntos PRIMERO (FIRST) y SIGUIENTE (FOLLOW).

[1] Varias producciones que comienzan con los mismos símbolos implicará que sus respectivos conjuntos Primero se solapan, cosa que incumple una de las condiciones de LL(1).



Ahora podemos abordar la codificación del *parser*.

Empezamos con las definiciones iniciales y el Analizador Léxico:

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

#define T_NUMBER 1001
#define T_OP     1002

int token ;                     // Here we store the current token/literal
int number ;                    // and the value of the number

int line_counter = 1 ;

int rd_lex ()
{
        int c ;

        do {
                c = getchar () ;
        } while (c == ' ' || c == '\t') ;

        if (isdigit (c)) {
                ungetc (c, stdin) ;
                scanf ("%d", &number) ;
                token = T_NUMBER ;
                return (token) ;        // returns the Token for Number
        }

        if (c == '\n')
                line_counter++ ;      // info for rd_syntax_error()

        token = c ;
        return (token) ;        // returns a literal
}
```

El Analizador Léxico rd_lex() lee la entrada carácter a carácter, saltando los espacios en blanco y los tabuladores. Si un carácter dado es numérico, lo devuelve al flujo de entrada, y lee un número entero completo con *scanf*. En tal caso deja el número almacenado en la variable global *number* (no se usa en este apartado). Y devuelve el *token* correspondiente a número *T_Number*, que queda también almacenado en la variable *token.*

En cualquier otro caso devuelve el carácter leído como literal (se almacena también en la variable *token*).

En caso de salto de línea se incrementa el contador *line_counter* para poder dar información aproximada en caso de error.

La variable *token* tiene la función de almacenar en cada instante el lexema leído, y *number* el valor del entero leído (cuando proceda).

Los errores sintácticos se tratan con la función *rd_syntax_error()* (muy mejorable). En el caso de invocar esta función, se imprime un mensaje y el programa termina:

```
void rd_syntax_error (int expected, int token, char *output)
{
	fprintf (stderr, "ERROR in line %d ", line_counter) ;
	fprintf (stderr, output, token, expected) ;

	exit (0) ;
}
```

El resto lo vemos de forma desordenada:
La función main se limita a llamar al *parser* dentro de un bucle infinito. Se entiende que el *parser* terminará cada vez que se introduzca un salto de línea. Si hubiese un error, el *parser* da el fallo y el programa termina. Si todo va bien, el bucle en *main* imprime OK y repite el proceso.

```
int main (void)
{
	while (1) {
		ParseExpression () ;
		printf ("OK\n") ;
	}

	system ("PAUSE") ;
}
```

La función *ParseExpression()* es la principal del Analizador. Vemos que transcribe la producción gramatical para Expression:

```
Expression ::= Term ExpressionRest
```

```
void ParseExpression ()			// E ::= TE'
{
	ParseTerm () ;
	ParseExpressionRest () ;
}
```

Cada No Terminal de la Gramática tendrá asociado una función que se encarga de analizar la entrada suponiendo que corresponde al No Terminal esperado.

Por ejemplo, *ParseTerm()* verifica que se lee lo que indica su producción.

```
Term ::= Number
```

Como *Number* es un *Token*, y no otro No Terminal, tendremos que leer algo concreto de la entrada (llamada a *rd_lex()*). La verificación se realiza llamando a *ParseNumber()*.

```
void ParseTerm ()		// T ::= N
{
	rd_lex () ;
	ParseNumber () ;
}
```

ParseNumber() usa la función genérica *MatchSymbol(token)* para comprobar que el lexema leído es del tipo *T_NUMBER*.

```
void ParseNumber ()
{
	MatchSymbol (T_NUMBER) ;
}
```

MatchSymbol() comprueba que el *token* actual corresponde con el pasado por argumento. En caso contrario, llama al mensaje de error para terminar.

```
void MatchSymbol (int expected_token)
{
	if (token != expected_token) {
		rd_syntax_error (expected_token, token, "token %d expected, but %d was read") ;
	}
}
```

Se entiende que cuando una función *ParseXXX()* termina de forma normal, es que el proceso de análisis ha sido correcto.

Nos queda por comentar la función más elaborada, que es *ParseExpressionRest()*, dado que en esta tenemos que lidiar con producciones alternativas y con la generación de *lambda*.

Dado que hay recursividad mutua entre *ParseExpression()* y *ParseExpressionRest()* vamos a necesitar incluir al menos el prototipo de una, para resolver la referencia en avance.

ParseExpressionRest() espera o bien un operador (seguido de un operando) o bien lambda. Esto quiere decir que baja a nivel de símbolos terminales, por lo que tiene que realizar una lectura con *rd_lex()* .

```
void ParseExpression () ;	// required prototype for forward reference in mutual recursion

void ParseExpressionRest ()	// E' ::= lambda | Op E
{
	rd_lex () ;				// ExpressionRest is a nullable Non Terminal
	if (token == '\n') {		// Therefore, we check FOLLOW(ExpressionRest)
		return ;			// This means that lambda has been derived
	}

	switch (token) {			// ExpressionRest derives in alternatives
		case '+' :			// requires checking FIRST(ExpressionRest))
		case '-' :
		case '*' :
		case '/' :
			break ;
		default :	rd_syntax_error (token, 0, "Token %d was read, but an Operator was expected");
			break ;
	}

	ParseExpression () ;// Forward reference in mutual recursion requires a previous prototye
}
```

Lo primero que necesitamos es asegurar que de verdad va a haber un par operador-operando. Para ello comprobamos que el *token* actual no está en el conjunto SIGUIENTE(ExpressionRest), en este caso sólo el símbolo '\n' (salto de línea==$).

Si hubiésemos leído el salto de línea, se entiende que ExpressionRest deriva en *lambda*, y habrá que retornar de la función actual.

Si no se cumple lo anterior, necesitamos comprobar que el token actual corresponde con alguno de los síbolos de PRIMERO(ExpressionRest) distintos de lambda. Estos pueden ser uno de los cuatro operadores aritméticos (+, -, * y /). Usamos un switch/case para verificarlo. Obviamente, si se ha leído un símbolo distinto (por ejemplo, %) habrá un error sintáctico. No hay más código en este bloque puesto que con comprobar los cuatro casos nos basta para verificar la corrección sintáctica. Además, el final de las cuatro producciones es igual

```
ExpressionRest ::=    + Expression |
                      - Expression |
                      * Expression |
                      / Expression | …
```

Por lo que podemos analizar el símbolo Expression de forma independiente al operador leído. Usamos una llamada (recursiva) a *ParseExpression()*.

Con esto, tendríamos el Analizador Sintáctico descrito, y en condiciones de operar sobre las siguientes secuencias:

```
1+2*3
OK
3*2+1
OK
3
OK

ERROR in line 4 token 10 expected, but 1001 was read
```

dr_calc1.c

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

#define T_NUMBER 1001
#define T_OP     1002

int token ;                             // Here we store the current token/literal
int number ;                            // and the value of the number

int line_counter = 1 ;

int rd_lex ()
{
        int c ;

        do {
                c = getchar () ;
        } while (c == ' ' || c == '\t') ;

        if (isdigit (c)) {
                ungetc (c, stdin) ;
                scanf ("%d", &number) ;
                token = T_NUMBER ;
                return (token) ;        // returns the Token for Number
        }

        if (c == '\n')
                line_counter++ ;        // info for rd_syntax_error()

        token = c ;
        return (token) ;        // returns a literal
}

void rd_syntax_error (int expected, int token, char *output)
{
        fprintf (stderr, "ERROR in line %d ", line_counter) ;
        fprintf (stderr, output, token, expected) ;

        exit (0) ;
}

void MatchSymbol (int expected_token)
{
        if (token != expected_token) {
                rd_syntax_error (expected_token, token, "token %d expected, but %d was read") ;
        }
}

void ParseNumber ()
{
        MatchSymbol (T_NUMBER) ;
}

void ParseTerm ()               // T ::= N
{
        rd_lex () ;
        ParseNumber () ;
}
```

```
void ParseExpression () ;       // required prototype for forward reference in mutual recursion

void ParseExpressionRest ()   // E' ::= lambda | OpE
{                             // ExpressionRest is a nullable Non Terminal
        rd_lex () ;
        if (token == '\n') {                    // Therefore, we check FOLLOW(ExpressionRest)
                return ;                        // This means that lambda has been derived
        }

        switch (token) {                        // ExpressionRest derives in alternatives
                case '+' :                      // requires checking FIRST(ExpressionRest))
                case '-' :
                case '*' :
                case '/' :
                        break ;
                default :    rd_syntax_error (token, 0, "Token %d was read, but an Operator was expected");
                        break ;
        }

        ParseExpression () ;// Forward reference in mutual recursion requires a previous prototye
}


void ParseExpression ()                 // E ::= TE'
{
        ParseTerm () ;
        ParseExpressionRest () ;
}


int main (void) {

        while (1) {
                ParseExpression () ;
                printf ("OK\n") ;
        }

        system ("PAUSE") ;
}
```

.

2. Calculadora de operaciones muy simples (Parser + Acciones Semánticas).

¿Cómo conseguimos que el que el Analizador Sintáctico además de comprobar la corrección sintáctica de la entrada, evalúe la secuencia y devuelva el valor numérico de la expresión?

Vamos a abordar una solución sencilla, evitando complicaciones. Implica que algunas funciones devuelvan un valor, por ejemplo *ParseNumber*, *ParseTerm*, *ParseExpression*, y que ésta última sea capaz de operar con los valores. Una complicación es que las Expresiones están partidas en dos (Expression y ExpressionRest), y hay que operar con valores obtenidos en cada una. Así que necesitaríamos un mecanismo para transmitir un valor de una a otra (por ejemplo, pasando argumentos). Esto lo evitaremos con una decisión de diseño.

No va a haber ningún cambio en el código inicial ya comentado. Los cambios comienzan en *ParseNumber()*. En concreto, que se cambia su tipo de *void* a *int* y se devuelve el valor almancenado en la variable global *number.* Se entiende que esto se puede realizar si *ParseNumber()* verifica que el *token* actual es *T_Number*. De *ParseTerm()* se puede comentar lo mismo.

```
int ParseNumber ()
{
        MatchSymbol (T_NUMBER) ;
        return number ;
}

int ParseTerm () {              // T ::= N      returns the numeric value of the Term
        int val ;

        rd_lex () ;
        val = ParseNumber () ;

        return val ;
}
```

Para ParseExpression() y ParseExpressionRest() tomamos la decisión de integrar el código de la segunda dentro de la primera (en vez de realizar la llamada a la segunda). Los cambios se refieren a la gestión de los valores devueltos por *ParseTerm()* y *ParseExpression()*. En el caso de derivar *lambda*, hay que devolver el valor del último Termino leído. También hará falta almacenar el operador leído para poder usarlo más tarde para operar. Para ello definimos las variables *val*, *val2* y *operator*.

```
int ParseExpression ()                          // E ::= TE'    U    E' ::= lambda | E
{                                               // returns the numeric value of the Expression
        int val , val2, operator ;

        val = ParseTerm () ;

//    ParseExpressionRes t() ;          // we expand this function into ParseExpression()

        rd_lex () ;                             // ExpressionRest is a nullable Non Terminal
        if (token == '\n') {                    // Therefore, we check FOLLOW(Expression)
                return val ;                    // This means that lambda has been derived
        }
        switch (token) {                // ExpressionRest derives in alternatives
                case '+' :              // requires checking FIRST(ExpressionRest))
                case '-' :
                case '*' :
```

```
                    case '/' :  operator = token ;  // remember the operator for later
                                break ;
                    default :   rd_syntax_error (token, 0, "Token %d was read, but an Operator was expected");
                                break ;
          }

          val2 = ParseExpression () ;                  // At this point the input has been parsed correctly
. . .
```

En el punto posterior a

```
          val2 = ParseExpression () ;                  // At this point the input has been parsed correctly
```

sabemos que el análisis sintáctico ha ido bien, con lo cuál podemos proceder a calcular el valor de la expresión. Por ello tenemos que repetir el switch/case realizando las operaciones correspondientes. Se ha incluido la opción *default* para detectar algún error "inverosímil".

```
. . .
          switch (operator) {                          // This part is for the Semantic actions
                    case '+' :  val += val2 ;
                                break ;
                    case '-' :  val -= val2 ;
                                break ;
                    case '*' :  val *= val2 ;
                                break ;
                    case '/' :  val /= val2 ;
                                break ;
                    default :   rd_syntax_error (operator, 0, "Unexpected error in ParseExpressionRest for
operator %c\n") ;
                                break ;
          }

          return val ;
}
```

Un último cambio mínimo corresponde a la función main() que ahora pasará a imprimir el valor de la expresión evaluada:

```
int main (void) {

          while (1) {
                    printf ("Valor %d OK\n", ParseExpression ()) ;
          }

          system ("PAUSE") ;
}
```

Ahora podemos probar el programa sobre las siguientes secuencias:

```
1+2*3
Valor 7 OK
3*2+1
Valor 9 OK
3
Valor 3 OK

ERROR in line 5 token 10 expected, but 1001 was read
```

dr_calc2.c

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

#define T_NUMBER 1001
#define T_OP     1002

int token ;                         // Here we store the current token/literal
int number ;                        // and the value of the number

int line_counter = 1 ;


int rd_lex ()
{
        int c ;

        do {
                c = getchar () ;
        } while (c == ' ' || c == '\t') ;

        if (isdigit (c)) {
                ungetc (c, stdin) ;
                scanf ("%d", &number) ;
                token = T_NUMBER ;
                return (token) ;    // returns the Token for Number
        }

        if (c == '\n')
                line_counter++ ;    // info for rd_syntax_error()

        token = c ;
        return (token) ;    // returns a Literal
}


void rd_syntax_error (int expected, int token, char *output)
{
        fprintf (stderr, "ERROR in line %d ", line_counter) ;
        fprintf (stderr, output, token, expected) ;

        exit (0) ;
}


void MatchSymbol (int expected_token)
{
        if (token != expected_token) {
                rd_syntax_error (expected_token, token, "token %d expected, but %d was read") ;
        }
}


int ParseNumber ()
{
        MatchSymbol (T_NUMBER) ;
        return number ;
}
```

```
int ParseTerm ()                // T ::= N      returns the numeric value of the Term
{
        int val ;

        rd_lex () ;
        val = ParseNumber () ;

        return val ;
}

int ParseExpression ()          // E ::= TE' + E' ::= lambda | E
{                               // returns the numeric value of the Expression
        int val ;
        int val2 ;
        int operator ;

        val = ParseTerm () ;

//      ParseExpressionRes t() ; // we expand this function into ParseExpression()

        rd_lex () ;                     // ExpressionRest is a nullable Non Terminal
        if (token == '\n') {            // Therefore, we check FOLLOW(ExpressionRest)
                return val ;            // This means that lambda has been derived
        }

        switch (token) {                // ExpressionRest derives in alternatives
                case '+' :              // requires checking FIRST(ExpressionRest))
                case '-' :
                case '*' :
                case '/' :  operator = token ;   // remember the operator for later
                            break ;
                default :   rd_syntax_error (token, 0, "Token %d was read, but an Operator was expected");
                            break ;
        }

        val2 = ParseExpression () ;
                                                // At this point the input has been parsed correctly
        switch (operator) {                     // This part is for the Semantic actions
                case '+' :  val += val2 ;
                            break ;
                case '-' :  val -= val2 ;
                            break ;
                case '*' :  val *= val2 ;
                            break ;
                case '/' :  val /= val2 ;
                            break ;
                default :   rd_syntax_error (operator, 0, "Unexpected error in ParseExpressionRest for
operator %c\n") ;
                             break ;
        }

        return val ;
}


int main (void)
{
        while (1) {
                printf ("Valor %d OK\n", ParseExpression ()) ;
        }

        system ("PAUSE") ;
}
```

3. Inclusión de expresiones con paréntesis.

El siguiente cambio se propone para ver cómo modificar el código a la hora de ampliar la gramática. Nos vamos a limitar a ampliar la gramática para permitir y operar con expresiones entre paréntesis.

Esto se traduce en la siguiente gramática, marcando en rojo la ampliación:

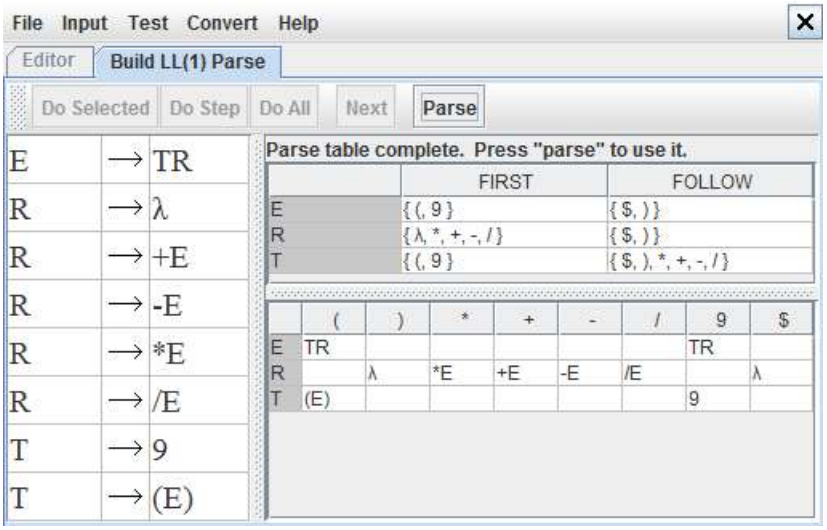
```
Expression ::=          Term ExpressionRest

ExpressionRest ::=      + Expression |
                        - Expression |
                        * Expression |
                        / Expression |
                        lambda

Term ::=                Number |
                        ( Expression )
```

Los cambios solo afectarían a la función *ParseTerm()*. Vemos que la gramática cumple con las exigencias para un Analizador Descendente Recursivo (sin recursividad a izquierdas, sin producciones que generen No Determinismo). Pero ahora Term genera dos producciones alternativas, lo cuál nos obliga a calcular su conjunto PRIMERO. En este caso, el cálculo es sencillo, pues ambas alternativas empiezan por Terminal en la primera derivación: PRIMERO(Term) = { *Number*, (}

Term no es anulable, por lo que SIGUIENTE(Term) no nos hará falta. **¡Pero siempre hay que ser consciente de que cualquier modificación en la gramática puede influir en el cálculo de todos los conjuntos!**



Podemos ver que cambian casi todos los conjuntos, aunque sólo va a influir SIGUIENTE(ExpressionRest) dado que en los demás No Terminales no hay producciones alternativas ni anulables.

Los cambios necesarios se indican en los siguientes fragmentos. *ParseTerm()* se amplía para procesar la alternativa de una expresión entre paréntesis. Primero consultamos si el token es alguno de los de PRIMERO(Term). Si es un *token* de tipo numérico, se llama a *ParseNumber()*, en caso contrario se supone que podría ser una expresión entre paréntesis y se llama a *ParseLParen()* seguido de *ParseExpression()* y *ParseRParen()*. ParseLParen() encarga de dar un error si el token no fuese el paréntesis de apertura (contenido en PRIMERO(Term)). La referencia recursiva a *ParseExpression()* en avance requiere un prototipo.

```
int ParseExpression () ;                    // Prototype for forward reference

int ParseTerm ()                // T ::= N | ( E )     returns the numeric value of the Term
{
        int val ;

        rd_lex () ;
        if (token == T_NUMBER) {                // T derives alternatives, requires checking FIRST(Expression)
                val = ParseNumber () ;
        } else {
                ParseLParen () ;
                val = ParseExpression () ;
                ParseRParen () ;
        }

        return val ;
}
```

ParseLParen() y *ParseRParen()* tendrían sus correspondientes funciones, pero en este caso las hemos redefinido como macros a *MatchSymbol(XXX)*. Esta es una práctica habitual cuando se quiere generar código un poco más eficiente, ya que cada llamada a una función (en este caso a *ParseLParen()* y *ParseRParen()*) supone un perder tiempo (*overhead*) en generar espacio para parámetros, variables locales, etc. Lo incluimos a modo ilustrativo.

```
#define ParseLParen()       MatchSymbol ('(') ; // More concise and efficient definitions
#define ParseRParen()       MatchSymbol (')') ; // rather than using functions
                                                // This is only useful for matching Literals
```

El único cambio adicional corresponde a *ParseExpression()* donde tenemos que ampliar la comprobación de si se está derivando *lambda*.

```
int ParseExpression () {                    // E ::= TE'      U     E' ::= lambda | E
{
. . .
        rd_lex () ;                                 // ExpressionRest is a nullable Non Terminal
        if (token == '\n' || token == ')') {        // Therefore, we check FOLLOW(ExpressionRest)
                return val ;                        // This means that lambda has been derived
        }

        switch (token) {                // ExpressionRest derives in alternatives
. . .
}
```

Ahora podemos probar el programa sobre las siguientes secuencias:

```
1+2*3
Valor 7
3*2+1
Valor 9
3
Valor 3
1+(2*3)
Valor 7
3*(2+1)
Valor 9
(3)*(2)+(1)
Valor 9
(3)
Valor 3

ERROR in line 9 token 10 expected, but 40 was read
```

Vemos que no se cumplen las precedencias de operador de * respecto a +. 3*2+1 debería dar 7 igual que 1+2*3.

En el siguiente ejemplo también se puede apreciar que no se cumple la asociatividad por la izquierda (el resultado debería ser -1). Aunque la teoría dice que un parser descendente recursivo aplica asociatividad de izquierda a derecha (la correcta en este caso), en esta implementación sucede que las evaluaciones se realizan en reverso al volver de las llamadas recursivas. De ahí que se realicen de derecha a izquierda.

```
1-1-1
Valor 1

ERROR in line 3 token 10 expected, but 40 was read
```

dr_calc3.c

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

#define T_NUMBER 1001
#define T_OP     1002

int ParseExpression () ;                    // Prototype for forward reference

int token ;                                 // Here we store the current token/literal
int number ;                                // and the value of the number

int line_counter = 1 ;

int rd_lex ()
{
        int c ;

        do {
                c = getchar () ;
        } while (c == ' ' || c == '\t') ;

        if (isdigit (c)) {
                ungetc (c, stdin) ;
                scanf ("%d", &number) ;
                token = T_NUMBER ;
                return (token) ;        // returns the Token for Number
        }

        if (c == '\n')
                line_counter++ ;        // info for rd_syntax_error()

        token = c ;
        return (token) ;        // returns a literal
}

void rd_syntax_error (int expected, int token, char *output)
{
        fprintf (stderr, "ERROR in line %d ", line_counter) ;
        fprintf (stderr, output, token, expected) ;

        exit (0) ;
}

void MatchSymbol (int expected_token)
{
        if (token != expected_token) {
                rd_syntax_error (expected_token, token, "token %d expected, but %d was read") ;
        }
}


#define ParseLParen()           MatchSymbol ('(') ; // More concise and efficient definitions
#define ParseRParen()           MatchSymbol (')') ; // rather than using functions
                                                    // This is only useful for matching Literals
int ParseNumber ()                          // Parsing Non Terminals and some Tokens require more
{                                                   // complex functions
        MatchSymbol (T_NUMBER) ;
        return number ;
}
```

```
int ParseTerm ()                    // T ::= N | ( E )     returns the numeric value of the Term
{
        int val ;

        rd_lex () ;
        if (token == T_NUMBER) {                    // T derives alternatives, requires checking FIRST( E )
                val = ParseNumber () ;
        } else {
                ParseLParen () ;
                val = ParseExpression () ;
                ParseRParen () ;
        }
        return val ;
}

int ParseExpression ()                      // E ::= TE' + E' ::= lambda | E
{                                           // returns the numeric value of the Expression
        int val, val2, operator ;

        val = ParseTerm () ;

//      ParseExpressionRes t() ;                        // we expand this function into ParseExpression()

        rd_lex () ;                                     // ExpressionRest is a nullable Non Terminal
        if (token == '\n' || token == ')') {            // Therefore, we check FOLLOW(ExpressionRest)
                return val ;                            // This means that lambda has been derived
        }

        switch (token) {                    // ExpressionRest derives in alternatives
                case '+' :                  // requires checking FIRST(ExpressionRest))
                case '-' :
                case '*' :
                case '/' :  operator = token ;
                            break ;
                default :   rd_syntax_error (token, 0, "Token %d was read, but an Operator was expected");
                            break ;
        }
        val2 = ParseExpression () ;
                                                        // At this point the input has been parsed correctly
        switch (operator) {                             // This part is for the Semantic actions
                case '+' :  val += val2 ;
                            break ;
                case '-' :  val -= val2 ;
                            break ;
                case '*' :  val *= val2 ;
                            break ;
                case '/' :  val /= val2 ;
                            break ;
                default :   rd_syntax_error (operator, 0, "Unexpected error in ParseExpressionRest for
operator %c\n") ;
                            break ;
        }
        return val ;
}

int main (void)
{
        while (1) {
                printf ("Valor %d\n", ParseExpression ()) ;
        }
        system ("PAUSE") ;
}
```

4. Inclusión de precedencia de operador, y signos unarios.

Como último apartado se propone la ampliación de la calculadora para que acepte signos unarios y potencias. Y que sea capaz de procesar con mayor precedencia el operador '^' (potencia) sobre los operadores '*' y '/', y estos con respecto a '+' y '-' (binarios). No es necesario tener en cuenta la asociatividad correcta.

Esto se propone como trabajo propio para los alumnos.

Como sugerencias:

- Los signos unarios '+' y '–' inicialmente se pueden limitar al *token Number*.

  Ejemplos:
  -1 + -1
  -1 - 1

- Para asignar mayor precedencia a un operador (*) respecto a otro (+), se propone incluir un nuevo nivel en la gramática. Se propone seguir un esquema como el que sigue (mantener Term como en el original, e introducir uno nuevo, como F).

  ```
  E ::= F + E | …

  F ::= T * F | …

  T ::= ( E ) | …
  ```

- Posteriormente, se recomienda ampliar la definición de los signos unarios para:
  - reconocer expresiones entre paréntesis con signos unarios;
  - admitir signos unarios encadenados.

  Ejemplos:
  ---1
  -(-1)+(-2)