Procesadores del Lenguaje: Sesión 2

Autores: José María Solinís Escolar, Pablo Martín Muñoz

1. Se debe a que la expresión se lee de izquierda a derecha de manera que no tiene en cuenta la prioridad de operaciones en matemáticas. En “expresión” están todas las operaciones metidas como si fueran iguales de manera que la multiplicación y la división no estén tratadas de forma distinta como debería ser. De esta manera 2*3+1 devuelve 8 ya que hace primero 3+1= 4 y luego se multiplica por 2 dando 8 en vez de 7 que sería la solución correcta.

2. Ésto ocurre porque no está tratado de ninguna manera la introducción de espacios en la introducción de expresiones. Por lo que además de permitir espacios después de signos de operación como “+” o “-” también lo permite entre números. La solución que hemos propuesto está incompleta debido a que permite números con espacios en la izquierda. La forma de resolverlo requiere devolver los números enteros en vez de los números cifra a cifra, creando un token que introducimos en nuestra gramática.

3. El fallo en expresión es como lo mencionado anteriormente que todas las operaciones sean tratadas por igual por lo que a la hora de resolver las operaciones no sepa qué operación debe hacer antes. Nosotros lo hemos resuelto separando las operaciones con prioridad (multiplicación, división y paréntesis) del resto de operaciones sin prioridad, de modo que se ejecutarán primero las operaciones con prioridad.

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

r_expr:                 /* lambda */
           | axioma
           ;

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

expresion:  prioridad                   { $$ = $1; }
           | expresion '+' prioridad     { $$ = $1 + $3; }
           | expresion '-' prioridad     { $$ = $1 - $3; }
           ;

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

4. Se debe a que en la primera implementación de axioma se usaba “expresión” que es un no terminal lo que genera ambigüedad al poder generar una misma sentencia de más de una manera, generando retardo en la salida. En cambio con “número” la recursividad se hace con terminales de manera que no genera la ambigüedad que generaba la anterior.