Universidad Carlos III
Grado en ingeniería informática
Curso Procesadores del Lenguaje 2024-25
Laboratorio 2
Curso 2024-25

Fecha: **05/02/2025** - ENTREGA: **2**

GRUPO: **403**

Alumnos: **Mario Ramos Salsón (100495849)**

Miguel Yubero Espinosa (100495984)

Cuestiones abiertas:

2.1 Observa el resultado de diversas expresiones como:

2*3+1
Expresion=8.000000
1+2*3
Expresion=7.000000
2+3*1
Expresion=5.000000
1*3+2
Expresion=5.000000
1-1-1
Expresion=1.000000
1-1-1-1
Expresion=0.000000
1-1-1-1-1
Expresion=1.000000
1-1-1-1-1-1
Expresion=0.000000
1-2-3-4-5
Expresion=3.000000

¿Sábes determinar a qué se deben los resultados obtenidos?

El resultado obtenido se debe a que la pila primero añade todos los elementos hasta llegar al final en vez de ir reduciendo cada vez que encuentra una expresión válida. Por la operación 2*3+1, la hace mal porque primero suma 3 + 1 = 4 y después lo multiplica por 2, por eso da 8 y no 7. En el caso de 1 + 2 * 3 = 7, da bien porque primero multiplica y después suma.

2.2. Existe un problema en la solución desarrollada hasta ahora: 1 2 3 + 2 1 Expresión=144.000000

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

En este caso se introducen números (dígitos) uno a uno separados por un espacio en blanco, y el problema ocurre porque la función *yylex()* ignora los espacios en blanco y lee los caracteres introducidos de forma continua. Como resultado, la entrada "*1 2 3 + 2 1*" se interpreta como una única secuencia de caracteres (123+21), por ello obtenemos 144.000 como resultado.

Como solución lo que haríamos es modificar *yylex()* de forma que sea capaz de reconocer números completos como tokens. Esto nos permite que "*1 2 3 + 2 1*" se interprete correctamente como números separados en lugar de una única cadena.

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

El fallo en la definición de expresión se debe a que no se maneja correctamente la asociatividad de los operadores. La gramática no define explícitamente la asociatividad de los operadores +, -, *, y /, lo que puede llevar a resultados incorrectos en expresiones con múltiples operadores del mismo tipo.

Por ejemplo la operación 1 - 1 - 1 se evalúa como 1 - (1 - 1) = 1, lo cual es incorrecto. Esto también ocurre en la expresión 2*3+1. Primero suma 3 + 1 = 4 y después lo multiplica por 2, por eso da 8 y no 7.

2.4. Se ha diseñado la definición de número como:
Usando una estructura recursiva similar a la que habíamos empleado inicialmente para axioma (Caso Base, Baso Base + Recursividad).
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?

La diferencia radica en cómo se maneja la recursividad en cada caso:
La recursividad en la definición de número funciona porque los dígitos se procesan de izquierda a derecha, y la acción semántica se aplica después de que se ha construido el número completo. Esto permite manejar números de múltiples dígitos correctamente.

Ejemplo: 123 se procesa como 1 * 10 + 2 = 12, después cuando entra el 3 hace 12 * 10 + 3 = 123.

En el caso del axioma, la recursividad por la izquierda causaba problemas porque la acción semántica se aplicaba antes de que se completará la evaluación de la expresión.

Ejemplo: Si se usa recursividad por la izquierda en axioma, la acción semántica se aplica en orden inverso, lo que genera resultados incorrectos.

Conclusión: La recursividad funciona en la definición de número porque los dígitos se procesan en el orden correcto y la acción semántica se aplica al final. En cambio, en axioma, la recursividad por la izquierda causa problemas porque la acción semántica se aplica antes de que se complete la evaluación, lo que lleva a ambigüedades y retrasos.