uc3m | Universidad Carlos III de Madrid

Ingenieria Informatica

Procesadores de Lenguaje 25/26

Grupo 81

Práctica 2

«Calculadora»

121:

Denis Loren Moldovan — 100522240@alumnos.uc3m.es

Jorge Adrian Saghin Dudulea — 100522257@alumnos.uc3m.es

Profesor

Maria Paz

Tabla de contenidos

1. Questiones

1.1. ¿A que se deben los resultados de las siguientes expresiones?

```
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
```

Los resultados obtenidos son los siguientes:

```
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
```

Esto se debe a que el programa actual no respeta el orden de los operandos, y además, el orden de las operaciones siempre es de derecha a izquierda. Por ejemplo, el orden lógico de la primera operación es primero resolver el producto entre 2 y 3 y luego sumarle al resultado final 1, por lo que el resultado final es 7. Sin embargo, en nuestro programa, la primera operación que se hace es la suma entre 3 y 1 y luego se realiza el producto entre esta solución y 2.

1.2. ¿Que soluciones se te ocurren al siguente problema?

```
1 2 3 + 2 1 <intro>
Expresion=144.000000
```

Este caso ocurre principalmente debido a que en la función de `yylex`, el bucle lo que hace es eliminar los caracteres en blanco (los huecos que hay dentro de la expresión) y devolver los caracteres limpios al lexer. Para solucionar esto, habría que devolver directamente el caracter en plano, y modificar la gramática para permitir o no los espacios, y tratar las expresiones de forma distinta.

Para esto, los espacios dentro de los caracteres numéricos no deberían aparecer (dejando la gramatica como está definida ahora mismo), mientras que en el resto de la gramática, hay que añadir más reglas para cada no terminal.

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

El fallo aparece cuando se intenta realizar una operación después de que se haya realizado una operación dentro del paréntesis. Por ejemplo, al realizar la suma 3+(2+1), se realiza correctamente, dando como resultado 6. Sin embargo, al realizar (3+2)+1, hay un syntax error. Esto se debe a que el no terminal expresión no tiene ningún tipo de recursividad cuando hay una operación con paréntesis, por lo que cuando se añade una expresión a la derecha, el programa no es capaz de procesarlo y por lo tanto lanza un error. Para arreglar esto, hemos decidido incluir el paréntesis dentro de los operandos, no dentro de las expresiones. Con esto arreglamos el error de las operaciones no recursivas.

```
expresion:   operando                { $$ = $1 ; }
           | operando '+' expresion  { $$ = $1 + $3 ; }
           | operando '-' expresion  { $$ = $1 - $3 ; }
           | operando '*' expresion  { $$ = $1 * $3 ; }
           | operando '/' expresion  { $$ = $1 / $3 ; }
           ;

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

1.4. ¿Por que funciona en el primer caso y no en el segundo?

```
numero: digito { $$ = $1 ; pot = 1 ; }
| digito numero { pot *= 10 ; $$ = $1 * pot + $2 ; }
;
```

—

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

La primera es válida, a diferencia del segundo porque se genera una gramática no determinista. Cuando añadimos un no terminal después del sintagma, tenemos que añadir el siguiente caracter para dar por hecho que ramificación seguir, y entonces se ejecuta la expresión. Lo que produce es que, si usamos la segunda, se produciria un retardo a la hora de ejecutar expresiones, y no se seguiría un orden natural de ejecución.