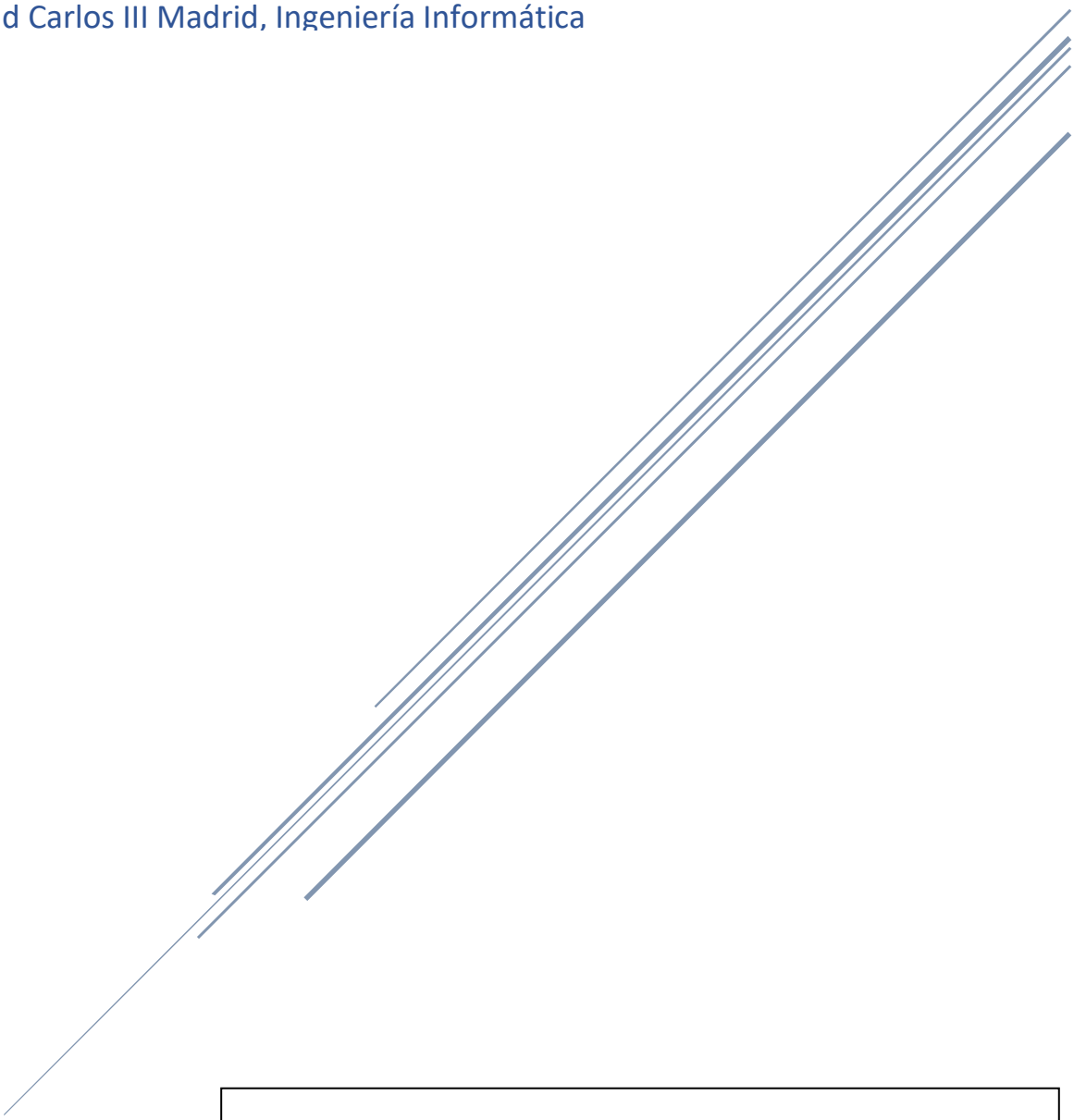
PROCESADORES DEL LENGUAJE

PRACTICA 2

Universidad Carlos III Madrid, Ingeniería Informática



IVAN AGUADO PERULERO – 100405871
JORGE SERRANO PÉREZ – 100405987

Contenido

1.- INTRODUCCIÓN

En el presente documento se van a describir brevemente las decisiones tomadas frente a los problemas abordados en la práctica número dos.

El objetivo de esta práctica es combinar los analizadores sintáctico y semántico para crear una calculadora con algunas funcionalidades extra. Para ello deberemos partir de la calculadora básica de los proyecto CUP, modificando las especificaciones de los lenguajes utilizados en el escáner y el *parser*. Usaremos las herramientas JFlex para el primero y CUP para el segundo.

En primer lugar, se nos pide aceptar e ignorar comentarios en el análisis léxico, y también incorporar número reales y hexadecimales. A continuación, deberemos incorporar nuevos símbolos en la gramática para calcular distintas funciones en el análisis sintáctico. Tendremos que eliminar las reglas de precedencia y modificar la gramática para evitar la necesidad de hacer uso de dichas reglas. Por último, incorporaremos un array de variables en memoria con sentencias de expresiones matemáticas y asignaciones, y los operadores de preincremento, postincremento, predecremento y postdecremento.

A la entrega se adjuntará un fichero de prueba con una entrada compuesta por código bien formado que producirá una salida correcta en el programa.

2.- PASOS SEGUIDOS

En este apartado se describirán las modificaciones realizadas para mejorar la calculadora básica. Se dividirán los cambios en el archivo *lexer.jflex* y el *parser.cup*.

RECONOCEDOR LÉXICO

En primer lugar, para aceptar los comentarios tan solo hemos añadido la expresión regular que los reconoce (que ya teníamos disponible de la calculadora básica) a la sección de las reglas de las expresiones regulares, y que como acción no haga nada, para de esta manera ignorarlos.

Por otro lado, para aceptar números tanto reales como hexadecimales, hemos usado las expresiones regulares que utilizamos en la práctica anterior, que son las siguiente:

- RealNumber = [0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?
- HexNumber = "0X"[0-9A-F]+|"0x"[0-9A-F]+

Los números reales incluyen números con decimales (Ej: 5.3 o también .1 (Equivalente a 0.1)) y números en notación científica (1e-1). Los números hexadecimales deben empezar por la secuencia “0X” o “0x”, seguidos de uno o más caracteres. Estos caracteres pueden ser números del 1 al 9 o letras de la A a la F, siendo la A el número 10, B el 11, etc.

A continuación, hemos añadido dichas expresiones a las reglas, pero en este caso hemos añadido una acción, que consiste en crear un símbolo en vez de ignorarlo como en el caso de los comentarios. Lo que hemos hecho ha sido asignar cada tipo de número al mismo símbolo terminal NUMBER, pero dándole el valor correspondiente en cada caso. Queremos destacar que en esta parte hemos modificado el código para convertir el valor reconocido a Double en vez de a Integer, ya que esto nos permite poder hacer sumas de números reales y enteros sin ningún problema, como se explicará más adelante. Además, para convertir el número hexadecimal primero hemos tenido que usar el método *Integer.decode* para obtener el valor entero, y luego transformarlo a Double.

Para completar esta parte, hemos incorporado una expresión regular que nos permitirá detectar cuando se quiere hacer uso de variables en memoria. La expresión en cuestión es esta:

- ArrayMEM = "MEM["[0-9][0-9]*"]"

Acepta variables denominadas como el literal “MEM” en mayúsculas, y luego entre corchetes la posición de memoria. Esta solo puede ser un número comprendido entre 0 y 99.

Siguiendo el procedimiento anterior, añadimos la regla que reconoce esta expresión, pero con un pequeño cambio. Hemos decidido crear una función para extraer el valor de la posición de memoria concreta a la que se quiere acceder, y ese será el valor del símbolo. Esto se debe a que una vez que hemos reconocido que se trata de una posición de memoria, solo es necesario conocer a qué posición corresponde. La función en cuestión recibe un *string* con la cadena reconocida (Ej: *MEM[0]*) y primero lo convierte a un array de caracteres con la función de java *toCharArray()*. A continuación recorre ese array en cada posición con un bucle, y comprueba mediante un *if* si el caracter es un dígito o no (función *isDigit()* de Java). Si lo es, lo añade a una variable *string*, que será la que más tarde devolverá. Por último, convertimos ese string a *Integer* para trabajar con él en el *parser*.

Finalmente, hemos añadido también los símbolos terminales correspondientes con división (/), exp, log, ln, cos y sin para incorporar en la gramática las respectivas funciones científicas, el símbolo “=” para la asignación de memoria y los operadores de preincremento (++x), postincremento (x++), predecremento(--x) y postdecremento (x--). Para ello tan solo hemos añadido la expresión en el lexer, y la acción será crear el símbolo con el método *symbolFactory.newSymbol()*, como en los casos anteriores.

RECONOCEDOR SINTÁCTICO

Tras lo añadido en el analizador léxico hemos procedido a vincularlo con el analizador sintáctico. El primer paso ha sido añadir los nuevos símbolos a la lista de terminales “*DIVISION, LOG, EXP, LN, SIN, COS, EQUAL, PLUSPLUS, MINUSMINUS*”. Los números hexadecimales y reales han sido declarados como terminales del tipo Double junto con los números naturales bajo el símbolo *NUMBER.* Esto nos facilitará las operaciones entre los mismos, dando resultados incluso más precisos al haber sido transformados todos ellos en números *Double* por el analizador léxico. Por otro lado, se ha agregado a los símbolos terminales *ARRAYMEM,* en este caso como entero pues como se ha mencionado anteriormente este es el valor del índice del array en memoria en sí.

A la hora de añadir nuevas reglas a la gramática se ha aprovechado la clase *Math* de Java, para realizar el logaritmo en base diez, el neperiano, exponencial, seno y coseno de un número.

A continuación, pasamos a reconocer y trabajar con el array en memoria. En primer lugar, tal y como se pedía en el enunciado, hemos inicializado todas sus posiciones a 0.0. El array es del tipo Double facilitando las operaciones con el resto de símbolos terminales. Esta declaración e inicialización del array se ha realizado dentro del *parser code* entre corchetes para que sea correctamente legible.

Tras esto, se ha procedido a añadir todas las funcionalidades pedidas con el array en la gramática. Si se escribe el nombre del array en el input se mostrará el valor que tiene almacenado en la posición indicada. Si este array está seguido de un = entonces se realizará una tarea de asignación, asignando el valor que resulte de la expresión de la derecha de la igualdad. Para ello, cuando se detecta, aparte de mostrarse por pantalla el resultado este se guarda en el array en la posición indicada.

Como parte opcional se ha implementado la funcionalidad de pre/post incremento y decremento de las variables del array. Para ello, cuando se detectan dos "+" o dos "-" seguidos sin espacio entre ellos delante de un array, se incrementa o decrementa en una unidad el valor de esa posición, y a continuación se muestra el resultado por pantalla. En caso de que la adición o resta sea posterior es al revés, primero se muestra el resultado y luego se cambia el valor. De esta manera no se tendrá en cuenta en la operación que esté involucrada, pero internamente si se cambiará.

Por último, se han eliminado las precedencias borrando todos los *precedence left* y hemos apreciado cómo la gramática detectaba varios conflictos entre reglas. Para solucionar esto se han creado nuevos símbolos no terminales e implementado una gramática escalonada. Esta va conectando no terminales entre sí. De esta manera podemos poner las operaciones de menor a mayor prioridad, es decir, de arriba abajo. Por lo tanto, se han puesto en la parte alta de la gramática las sumas y restas, seguidas en el siguiente nivel de las multiplicaciones y divisiones, para acabar con el resto de operaciones en el último escalón, además de los paréntesis. Estas operaciones serán las más prioritarias, y se trata de las operaciones de asignación, reconocimiento, incrementos, decrementos, modificadores de signo, seno, cos, logaritmos y exponenciales.

3.- CONSIDERACIONES PARA EL CORRECTOR

A continuación, se van a exponer diferentes cuestiones a tener en cuenta a la hora de pasar un corrector automático a la práctica.

- Todo lo que se quiera asignar a una posición del array de memoria debe ir entre paréntesis.
- Cuando se quieran poner dos signos seguidos debe haber un espacio entre ellos, o que estar separados por un paréntesis (para el caso de los - y +).
- Si se quiere hacer uso de los operadores de pre/post incremento y decremento en las variables de memoria, estos signos se escribirán juntos, sin espacios entre ellos.
- La expresión *log* calcula logaritmos en base 10.
- La expresión Ln calcula logaritmos en base e.
- Las expresiones log, Ln, exp, sin y cos deben escribirse de la manera expuesta para ser reconocidas, téngase en cuenta la L mayúscula del logaritmo neperiano.
- La operación solo se reconocerá si finaliza con el símbolo ";".
- Para referirse al array en memoria se escribirá MEM[i] con i perteneciente a [0,99].
- Una posición de array no modificada contiene el valor 0.0.

Se pueden ver las pruebas realizadas en el txt entregado. Entre ellas se incluyen el uso de comentarios, operaciones con números enteros, reales y hexadecimales, logaritmos, exponenciales, senos y cosenos, asignaciones de memoria y operaciones de preincremento y postincremento.

4.- CONCLUSIONES Y PROBLEMAS ABORDADOS

Esta práctica nos ha servido para fijar los contenidos de las anteriores y entender la importancia del funcionamiento simultáneo de un analizador léxico y sintáctico. Además, al tener que crear una gramática funcional y trabajar con ella esta práctica consigue ampliar los contenidos teóricos de las clases magistrales de la asignatura.

Los principales problemas abordados han sido los siguientes:

En primer lugar, conseguir operar entre símbolos terminales diferentes, lo cual se ha resuelto transformando todos a *Double*. Leer la posición del array escrito en el input para trabajar con ella también frenó nuestro desarrollo, lo cual como se ha mencionado lo soluciona nuestra función *arrayPos,* esta ignora el resto para quedarse únicamente con este índice. Por otro lado, inicializar todas las posiciones del array a 0 no fue fácil, pues aunque poníamos el código en el lugar correcto no se leía correctamente y daba error. Más tarde nos dimos cuenta de que poniéndolo entre corchetes funcionaba correctamente. Por último, diferenciar entre pre/post incremento y decremento nos atascó un poco, pero acabamos solucionándolo como se ha expuesto, mostrando y luego modificando en caso de ser post y modificando y luego mostrando en caso de ser pre.

Además, también nos confundió que algunos ejemplos que se proporcionaban en el enunciado no estaban correctamente escritos, faltando por ejemplo un paréntesis, y el resultado expuesto en el comentario a su derecha no era correcto. Creemos conveniente que se corrija este error del enunciado para futuros alumnos.

En conclusión, una práctica útil, la cual te obliga a replantearte el conocimiento adquirido con los problemas que se pueden encontrar. De esta manera se consigue un aprendizaje de los contenidos más profundo.