PRÁCTICA 2
ANALIZADOR LÉXICO

MODELOS DE COMPUTACIÓN
GRUPO B

Juan Carlos Ruiz Fernández B2

Contenido

INTRODUCCIÓN **2**

INSTRUCCIONES **2**

OPERACIONES 2

Ejemplos de instrucciones *2*

CONSULTAS 3

Ejemplos de consulta *3*

PROGRAMA **3**

EXPRESIONES REGULARES 3

Variables *3*

Números *3*

Peticiones y ordenes *4*

FUNCIONALIDADES 4

Variables necesarias *4*

Funciones *5*

Macros 5

Funciones 5

MATCHINGS 6

Flags *6*

Peticiones *6*

Variables *6*

Números *7*

Defaults *8*

EJEMPLO DE FLUJO DE EJECUCIÓN **8**

COMPILACIÓN Y EJECUCIÓN **9**

Introducción

En esta segunda práctica se ha realizado un analizador léxico para ejecutar unos comandos básicos de calculadora con lenguaje natural, mayormente. En ella se pueden realizar operaciones de suma, resta, multiplicación y división usando tantos números enteros, reales, variables o resultados anteriores almacenados en memoria.

Instrucciones

Operaciones

Para ejecutar las operaciones hay que escribir el siguiente tipo de sentencia:

[operación] [operando_1] [operando_2]

Para la ***operación*** se puede aportar de forma natural, en cada caso se encuentran las siguientes opciones:

- **Suma:**
  - suma, SUMA, sum, SUM, add, ADD
- **Resta:**
  - resta, rest, sub, substract, RESTA, REST, SUBSTRACT, SUB
- **Multiplicación:**
  - multiplica, mult, multi, MULTIPLICA, MULT, MULTI
- **División:**
  - divide, div, DIVIDE, DIV

Los ***operandos*** se pueden aportar de tres maneras:

- **Números**
  - Reales o enteros, positivos o negativos. Ejemplo: *1, 2.1, -3.2…*
- **Memoria**
  - Entre corchetes indicar el índice de valor. Ejemplo: *[4]*
  - Para conocer esos índices se le puede pedir que muestre la memoria.
- **Variables**
  - Basta con ingresar el nombre de la variable
  - RESTRICCIONES
    - *Solo se puede usar la variable como "operando_2".*
    - *El nombre de las variables solo puede ser alfabética, empezando con una letra mayúscula.*
    - *Si no esta creada previamente, su valor es 0.*

Ejemplos de instrucciones

De forma simple, estas son unas posibles sentencias de ejecución:

sum 10 20	*mult 20.5 2.21*	*rest 2 3*	*divide 20 10*
suma 10.05 -20.41	*multi 3.14 Pi*	*sub 5 4.1*	*div [1] [5]*
add [0] [1]	*MULTIPLICA [2] Pi*	*Resta 0.1 0.001*	*DIV [0] Pi*

Consultas

El analizador almacena todas las operaciones realizadas y todas las variables creadas. Se pueden consultar de forma similar y se puede reducir a dos modos: con lenguaje natural o forma reducida.

La forma natural comprende palabras de petición como *dame, quiero, muestra, muéstrame…* y lo que se quiere pedir, *la memoria* o *las variables*.

La forma reducida es simplemente acotaciones o palabras clave como *mem, memoria*, *vars,* o *variables.*

Ejemplos de consulta

muestra la memoria
dame la memoria
mem

quiero las variables
enseña las variables
vars

Programa

Expresiones Regulares

Estas expresiones se dividen en tres tipos, unas para detección de variables y su asignación, otras para números y accesos a memoria, y otros para para operaciones y peticiones.

Variables

El nombre de las variables presenta las restricciones, anteriormente dichas, de que solo pueden contener letras, que la primera deber ser mayúscula. Al menos debe contener una letra:

```
variable ([A-Z][a-z]*)
```

Esa expresión se usará mas adelante tanto para consultar el valor de la variable como para usarla en una operación.

Para crear **y** asignar una variable hay que introducir el nombre seguido del símbolo *=.*

```
asigVariable {variable}"="
```

Esta expresión se usa tal cual, se activa un booleano que servirá de indicador cuando se lea un número y almacenar el valor en esa variable.

Números

Para los números se desglosará en dos: los números enteros *ent*(cualquier combinación de dígitos del 0 al 9) y los decimales *numdec*(enteros seperados por un “.”) para formar un numero “útil”. La expresión *num* engloba ese número útil que además puede ser negativo gracias a *“-”?*.

```
ent [0-9]*
numdec {ent}"."{ent}
num “-”?{ent}|{numdec}
```

Para acceder a una posición de memoria de los resultados en una operación hay que envolver un numero entero entre corchetes.

```
memnum"["{ent}"]"
```

Peticiones y ordenes

Aquí se recogen todas las posibilidades para ejecutar la orden de la operación como anteriormente se ha descrito:

```
sum     "suma"|"SUMA"|"sum"|"SUM"|"add"|"ADD"
mul     "multiplica"|"mult"|"multi"|"MULTIPLICA"|"MULT"|"MULTI"
res     "resta"|"rest"|"sub"|"substract"|"RESTA"|"REST"|"SUBSTRACT"|"SUB"
divi    "divide"|"div"|"DIVIDE"|"DIV"
```

Además las peticiones en forma natural se unifican en la expresión llamada *petición:*

```
peticion      ["MUESTRA ""muestra ""MUESTRAME ""muestrame ""dame ""DAME
""quiero  ""QUIERO  ""Enseña  ""Enseñame  ""enseña  ""enseñame""ENSEÑAME
""ENSEÑA "]
```

Con esto se crea las expresiones para atender las peticiones de obtener memoria y variables, tanto en forma natural como reducida:

```
memoria      ({peticion}"la memoria")|"mem"|"memoria"
variables    ({peticion}"las variables")|"vars"|"variables"
```

Funcionalidades

Variables necesarias

Para poder reutilizar las expresiones creadas y no tener que crear adicionales para cada caso se han utilizado booleanos como flags:

`bool sum = 0;`	Operación suma
`bool rest = 0;`	Operación resta
`bool mult = 0;`	Operación multiplicar
`bool divi = 0;`	Operación dividir
`bool asignarVariable = 0;`	Asignación/creación de variable

Para el funcionamiento del programa se ha usado objetos de librerías de C++ como *stack, vector, map* y *string:*

```
stack<float> pila;
```

Los operandos de las operaciones se almacenan en una pila de números reales, o coma flotante.

```
vector<pair<string, float>> mem;
```

Los resultados de estas operaciones se guardan en un vector que almacena pares. El primero corresponde a una cadena de caracteres generada en el momento de la operación que almacena los operandos usados y su resultado. El segundo es el resultado en coma flotante para su posterior recuperación.

```
map<string, float> variables;
```

Las variables se almacenan en un diccionario de cadenas de caracteres(nombre de la variable) y números reales, lo cual facilita con creces su creación y posterior recuperación de su valor.

```
string variable;
```

Esta ultima variable se usa para almacenar de forma temporal el nombre de la variable cuando se crea y asigna un valor.

Funciones

Macros

El uso de macros en este caso simplemente es para facilitar la compresión del código y reducir su tamaño. Se han usado dos:

```
#define toFloat(cadena) ((float)strtod(cadena,NULL))
```

Esta macro devuelve un numero real de una cadena de caracteres del tipo *char*.*

```
#define printn(cadena) (printf("He leido el numero %s\n",cadena))
```

Printn imprime por pantalla la cadena aportada, integrada en otra para su mejor compresión.

Funciones

```
float returnSol();
```

Esta función es la base de la ejecución de las operaciones y su almacenaje en memoria:

```
    a = pila.top();pila.pop();
    b = pila.top();pila.pop();
    limpiarPila();
```

Primero extrae los operandos de la pila y la vacía al completo si hubiera, por error, algún elemento más.

```
if( sum ){
 sol = b + a;
 toMem = to_string(b)+" + "+to_string(a)+" = "+to_string(sol);
}
```

Ejecuta la operación dependiendo del flag activado, generando y almacenando el resultado y su cadena de caracteres correspondiente en memoria. Para terminar resetea todos los flags, imprime el resultado por pantalla con el mensaje *Resultado=X* donde X será la solución.

```
void limpiarPila();
```

Limpiar pila se ha creado para acortar el código, mejorar su lectura y evitar errores encadenados. Solo contiene la siguiente instrucción:

```
while(!pila.empty()) pila.pop();
```

```
void returnMemoria();
```

Con la expresión correspondiente a la petición de obtener memoria se ejecuta esta función.

```
int i = 0;
cout << "# Memoria" << endl;
for(auto sol : mem){
   cout << "# [" << i << "] " << sol.first << endl;
   i++;
}
```

La cual muestra y calcula los índices de las entradas a memoria para poder acceder a ella. Se hace uso de la palabra clave **auto** de C++ para facilitar la compresión del código.

```
void returnVariables();
```

De la misma manera que la anterior, esta función se ejecuta cuando se pide visualizar las variables existentes.

```
int memint(char* a);
```

Esta función esta creada para acortar la extracción del índice de memoria al que se quiere acceder. A la variable *a* contendrá una expresión de esta forma *[14]* y devuelve el numero entero entre los corchetes.

```
string s(a);
string num = s.substr(1,s.size()-2);
return stoi(num);
```

Hace uso de la funcion *substr* de la clase *String* para dejar fuera de un nuevo objeto *string* los corchetes y asi poder usar *stoi* para convertirlo a *int.*

```
string popback(char* cstr);
```

Su uso se restringe a convertir la funcion *pop_back()* de la clase *String* a usarse con *char** y devolver un *string* por comodidad. Se usa para eliminar el ultimo carácter, en ese caso cuando se asigna a una variable un valor. Se introduce *[Nombre_variable]=* y se elimina el *=.*

Matchings

Aquí se recogen las acciones ejecutadas por las expresiones regulares. Se pueden categorizar en *flags, peticiones, variables* y *números:*

Flags

```
{sum}           {sum  = 1; limpiarPila();}
{mul}           {mult = 1; limpiarPila();}
{res}           {rest = 1; limpiarPila();}
{divi}          {divi = 1; limpiarPila();}
```

Corresponden a las expresiones que engloban las posibilidades de llamada a las operaciones, además ejecutan la limpieza de la pila ya que se presupone que está vacía. Con esto ultimo se evitan posibles errores encadenados.

Peticiones

```
{memoria}       {returnMemoria();}
{variables}     {returnVariables();}
```

Las coincidencias correspondientes a los accesos a memoria y a las variables.

Variables

```
{variable}{
if( sum || rest || mult || divi )
        pila.push(variables[string(yytext)]);
else
       cout << "Variable <" << yytext << "> = " << variables[string(yytext)]<< endl;
if(pila.size() >= 2) returnSol();
}
```

Aquí se representa, de forma más visual que en el programa, las posibilidades de ejecución cuando se encuentra el nombre de una variable (palabra con la primera mayúscula). Se puede dividir en tres secciones, las dos primeras exclusivas:

El primer if detecta si se esta ejecutando una operación, en dicho caso se inserta en la pila como operando el valor de dicha variable consultado en el diccionario de variables.

Si no se está ejecutando una operación significa que se esta consultado el valor de la variable y se imprime por pantalla. La peculiaridad es que no hace falta que estén previamente creadas, gracias al diccionario.

Independientemente de lo que se haya ejecutado antes, se comprueba si la pila está llena (dos operandos dentro) y si es así se ejecuta la función que calcula, imprime y almacena la operación de esos dos comandos.

```
{asigVariable}  {asignarVariable = 1; variable = popback(yytext);}
```

Conviene recordar que esta coincidencia es del tipo *{variable}"="* donde se ingresa el nombre de una variable, el cual debe tener la primera letra mayúscula. Seguida de = para asignarle un valor.

Gracias a ese = se sabe que se quiere asignar un valor numérico a una variable, por lo que se setea el flag de asignación de variable y se almacena temporalmente su nombre.

Números

```
{num}{
if( sum || rest || mult || divi )
   pila.push( toFloat(yytext) );
else
   if(asignarVariable){
       variables[variable] = toFloat(yytext);
       asignarVariable=0;
    }
    else printn(yytext);
if(pila.size() >= 2) returnSol();
}
```

Igual que con las variables, se distinguen tres secciones donde las dos primeras son exclusivas.

Primeramente se detecta si se está ejecutando una operación, si es así se introduce el valor en la pila de operandos.

Si no es así, pero el flag de asignación de variable esta activo, se introduce ese numero en el diccionario de variables gracias a que el nombre de variable se guardó previamente de forma temporal. Si el flag no estuviera activado, se devolvería el numero por pantalla.

Como último paso, independiente a los superiores, se comprueba si la pila esta llena se ejecuta la función *returnSol.*

```
{memnum}{
if(memint(yytext)<0 || memint(yytext)>mem.size() || mem.size()==0)
        cout << "ERROR: indice de memoria fuera de rango" << endl;
    else
        if( sum || rest || mult || divi )
            pila.push(mem[memint(yytext)].second);
        else{
            if(asignarVariable) {
                variables[variable] = mem[memint(yytext)].second;
                asignarVariable=0;
            }
        }
```

```
    if(pila.size() >= 2) returnSol(); }
```

En este caso se consideran las lecturas de un acceso a memoria, el cual tiene una primera comprobación para no provocar un error de segmentación por culpa de un índice que este fuera del rango de ese momento.

Si fuera correcto, se hacen las mismas comprobaciones que en casos anteriores. Se permite accesos a memoria como operandos o como asignaciones a variables.

Además de la ultima sentencia que ejecuta la operación si la pila esta llena.

Defaults

En este caso se modifica el comportamiento del analizador para casos de lectura de espacios, retorno de líneas y un caso especial:

```
[ \t]           {}
\n              {}
\n\n            {sum=0;mult=0;rest=0;divi=0;asignarVariable=0; cout <<
"**RESETEO**"<<endl;}
```

Con los dos primeros no se imprime nada si se lee un espacio o si se ha pulsado enter, pero si se pulsa doblemente la tecla enter se resetearán todos los flags. Esto solo es útil en caso de debugging.

Ejemplo de flujo de ejecución

Se va a tomar una instrucción genérica con la cual se explicará como el programa evoluciona mientras lee la cadena de caracteres. Esa instrucción es `multiplica [5] Var:`

Steps	Matching	Ejecución
1	`multiplica [5] Var`	`{mul}{mult = 1; limpiarPila();}`
2	`Multiplica_[5] Var`	`[ \t]{}`

3	`multiplica [5] Var`	`{memnum}`
		Primeramente se comprobará que el índice entre corchetes está en el rango, para consultar el índice se usa ejecuta la función:
		`memint(yytext)`
		Se supone correcta, pues pasará a comprobar que operación hay activada. Como en el paso *1* se activo *mult* meterá en la pila el operando extrayéndolo de la memoria:
		`pila.push(mem[memint(yytext)].second);`
		Saltará a la ultima comprobación, si la pila esta llena, como no lo está acabará la ejecución de esta parte.

4	`multiplica [5]_Var`	`[ \t]{}`

5	`multiplica [5] Var`	`{variable}`
		Se comprobará si se esta ejecutando alguna operación, como es así desde el paso *1,* meterá el valor de la variable en la pila:
		`pila.push(variables[string(yytext)]);`
		Saltará a la última comprobación sobre el estado de la pila, la cual se ejecutará porque será cierta:
		`if(pila.size() >= 2) returnSol();`
		Y entrará en la funcion que ejecuta la operación.

Compilación y ejecución

Se ha creado un makefile con diferentes tipos de llamada para su compilación y ejecución:

- **make**
  Simple ejecución de makefile que genera el ejecutable *calculator.* No ejecuta nada
- **make clean**
  Limpia el ejecutable, ficheros de compilación lex y la salida de ejemplo.
- **make run**
  Compila y ejecuta el programa para su uso directo(sobre terminal).
- **make ejemplo**
  Compila y ejecuta el programa introduciéndole un archivo ejemplo de ejecución, que salvará en el archivo *salida_ejemplo.txt* y lo mostrará por pantalla automáticamente.