Lenguaje Luma

Introducción

Luma es un lenguaje imperativo con sintaxis en español inspirada en Python, Ruby y C.

Definición

Un programa en Luma comienza en la primera línea de código de arriba a abajo que se encuentre fuera de cualquier definición de métodos o funciones.

Se recomienda que los ficheros de código fuente terminen en .luma pero no es estrictamente necesario.

Los comentarios que el compilador ignora se escriben tras dos barras en una línea o tras un /* hasta que se encuentre un */, estilo C:

```
// Esto es un comentario

/*   Esto es

     un bloque

     de comentarios */
```

Las instrucciones no se terminan con ‘;’, se delimitan por un salto de línea.

Los nombres de variables y funciones aceptan tildes. Las variables se declaran implícitamente en su asignación. Su tipo se decide en función del valor asignado. La asignación se realiza colocando el nombre de la nueva variable, el símbolo '=' y el nuevo valor o expresión. Existen cuatro tipos primitivos:

- Los **caracteres** se escriben entre comillas simples
- Los **enteros** se escriben como números sin punto
- Los **reales** se escriben como números con punto
- Los **booleanos** se escriben como verdadero o falso (también V o F)

```
tipo = 'A' // caracter

edad = 47 // entero

altura = 1.82 // real

conclusión = falso // booleano
```

Las cadenas, aceptadas en las llamadas a `escribe`, se escriben entre comillas dobles y aceptan los siguientes caracteres especiales:

- `\n` Salto de línea
- `\t` Tabulador

Conversión

Cuando se espere un tipo de dato y se encuentre otro, como en una asignación cuyo tipo de variable ya estuviera definido, o en una expresión, según los tipos de dato origen y destino implicados, se hará una conversión implícita o se lanzará un error indicando al usuario que no se puede convertir en ese caso particular. En la siguiente tabla se recoge la compatibilidad de tipos en las conversiones:

¿Permite conversión implícita?

Desde/Hasta	Caracteres	Enteros	Reales	Booleanos
Caracteres	-	Sí	No	No
Enteros	Sí	-	Sí	No
Reales	No	Sí	-	No
Booleanos	No	No	No	-

Expresiones

Los números enteros y reales se pueden operar con los símbolos '+', '-', '*' y '/' que equivalen a las operaciones aritméticas suma, resta, multiplicación y división respectivamente. Si se intenta operar un entero con un real, el entero se convertirá en un real antes de realizar la operación. Los caracteres y los booleanos no se pueden operar. Las cadenas se pueden concatenar mediante el operador '+'. Si se intenta concatenar cualquier otro tipo con una cadena, primero se convertirá en cadena.

Los booleanos se pueden operar con los símbolos 'y', 'o' y 'no' que equivalen a las operaciones lógicas and, or y not. Estas operaciones solo aceptan valores booleanos.

Las operaciones se pueden combinar en una expresión aritmética que se puede colocar para evaluarse en el lado derecho de una asignación o al pasar un parámetro a una función.

Las variables se pueden comparar entre ellas o directamente entre valores con los operadores de comparación ‘mayor’, ‘menor’, ‘igual’, ‘mayorigual’ y ‘menorigual’, que se evalúan a un valor booleano. No se pueden comparar tipos distintos salvo enteros y reales, convirtiendo previamente los enteros a reales. Las comparaciones no se pueden combinar como las operaciones aritmético-lógicas (no se puede hacer 3 menor x menor 4).

Precendencia de operadores

Todo lo que esté entre paréntesis tiene prioridad. Los operadores que estén en el mismo nivel de prioridad se leerán de izquierda a derecha. Más alto = Más prioritario

Aritméticos	Lógicos
* / modulo	no
+ -	y
	o

Estructuras

Las estructuras comienzan con una cabecera terminada en ‘:’, les sigue un bloque de código (un conjunto de instrucciones) y se finalizan con la palabra clave `fin`.

La palabra clave `si` va seguida de una expresión condicional terminada por dos puntos. La expresión no tiene por qué ir rodeada de paréntesis. Tras esta estructura se pueden escribir instrucciones que se ejecutarán solo si la condición se evalúa como verdadero. Si la condición se evalúa como falso, se ejecutarán las instrucciones que siguieran a la palabra clave `sino`, si la hubiera.

La palabra clave `mientras` también va seguida de una expresión condicional que no tiene por qué ir rodeada de paréntesis, dos puntos y un conjunto de instrucciones que se ejecutarán continuamente hasta que la expresión se evalúe como falso.

Métodos de biblioteca

La palabra reservada `escribe` muestra por pantalla el mensaje que le sigue. A diferencia de las funciones, no necesita paréntesis en su único argumento, que debe ser texto formateado estilo C o una expresión.

El símbolo `lee` recibe entrada del usuario mediante teclado. Debe ir seguido de la palabra clave `entero` o `decimal`, que indica el tipo de dato que se leerá.

No es posible cargar otras bibliotecas a falta de import o similares.

Funciones

Las funciones no declaran el tipo que devuelven. Si no llevan argumentos, se pueden escribir sin paréntesis. No es necesario escribir ninguna palabra clave antes del nombre de la función. A la cabecera de la función le sigue un conjunto de instrucciones que se ejecutarán tras su llamada.

```
nombreDeFunción:
```

```
nombreDeFunción(tipo primerArgumento, tipo segundoArgumento,
...):
```

Es posible devolver un único valor mediante la palabra clave `devuelve` seguida de un valor o una expresión. Si la función no contiene un devuelve, no devolverá nada. Si se intenta acceder al valor devuelto por una función que no devuelve ningún valor se genera un error. El tipo de dato devuelto por una función viene dado por el tipo del primer dato que devuelve.

Es necesario definir el tipo de dato de cada parámetro de la función antes de su nombre.

No es posible declarar funciones sin definirlas, y por tanto no es posible llamar funciones dentro de otras funciones si no están definidas. Se permiten funciones recursivas. Las variables declaradas en el ámbito superior a la función son accesibles por ella.

Para llamar a una función sin parámetros no hace falta incluir paréntesis. Una función y una variable no podrán tener el mismo nombre. Es posible evaluar una función directamente desde una expresión.

```
x = 3 + potencia(x, 2)
```

Pruebas

Sumador

```
suma(entero primerSumando, entero segundoSumando):
	devuelve primerSumando + segundoSumando
fin

si suma(2, 3) igual (2 + 3):
	escribe "Las sumas son iguales\n"
fin
```

Potencia

```
elevar(entero base, entero exponente):
	resultado = 1
	i = 0
	mientras i menor exponente:
		resultado = resultado * base
		i = i + 1
	fin
	devuelve resultado
fin

escribe elevar(elevar(3, 3), 3)
escribe "\n"
```

Años bisiestos

```
bisiesto(entero año):
	devuelve (año modulo 4 igual 0) y (año modulo 100 distinto
0) o (año modulo 400 igual 0)
fin

escribe "Introduzca un año después de Cristo: "
comando = lee entero
mientras comando distinto -1:
	si bisiesto(comando):
```

```
            escribe "Año bisiesto\n"
      sino:
            escribe "Año ordinario\n"
      fin
      comando = lee entero
fin
```

Recursividad

```
factorial(entero n):
    si (n igual 0) o (n igual 1):
        devuelve 1
    fin
    devuelve n*factorial(n-1)
fin

fibonacci(entero n):
    si (n igual 0) o (n igual 1):
        devuelve 1
    fin
    devuelve fibonacci(n-1) + fibonacci(n-2)
fin

i = 0
mientras i menor 10:
    escribe fibonacci(i) + factorial(i)
    escribe "\n"
    i = i + 1
fin
```

Aplicación de ejemplo: Simulación física sencilla

```
posX = 0.0
posY = 0.0
velX = 2.1
velY = -1.2
atractorX = -1.0
atractorY = -1.0

// Integración de Euler
actualizarEstado:
    fuerzaX = atractorX - posX
    fuerzaY = atractorY - posY
    velX = velX + fuerzaX
    velY = velY + fuerzaY
    posX = posX + velX
    posY = posY + velY
```

```
fin
mostrarEstado:
    escribe "Posición: "
    escribe posX
    escribe ", "
    escribe posY
    escribe "\n"
fin

escribe "Introduzca el número de pasos de la simulación: "
pasos = lee entero
escribe "\n"
pasosActuales = 0
mientras pasosActuales menor pasos:
    actualizarEstado
    mostrarEstado
    pasosActuales = pasosActuales + 1
fin
```

Analizador Léxico

El analizador léxico se ha implementado con la herramienta FLEX, con patrones en forma de expresión regular para cada símbolo que el compilador se pueda encontrar en el lenguaje.

Algunos apuntes sobre la implementación:

- Los operadores aritméticos y lógicos se reconocen como un mismo tipo de símbolo OPERATOR en esta fase
- Además de aceptar comillas dobles normales para las cadenas (") el analizador reconoce también comillas inglesas (“”)
- Las palabras clave si y sino se reconocen ambas como un mismo símbolo IF. Será tarea del analizador sintáctico diferenciar cuándo nos encontramos ante una situación u otra.
- Dentro del conjunto de caracteres que se acepta en un identificador se incluyen; además del conjunto de letras del inglés y los números; las vocales con tilde y la ñ, mayúsculas y minúsculas. Esto se debe a que este es un lenguaje enfocado a escribirse enteramente en español.

Analizador Sintáctico

Para la fase de análisis sintáctico se ha delegado a bison la codificación de los *tokens* correspondientes a cada símbolo, dando nombre a cada uno mediante su operador `%token`.

También se ha agregado el operador *distinto,* que faltaba en la implementación de la fase anterior.

Las expresiones se han implementado como un único subárbol para las operaciones tanto aritméticas como lógicas. A la hora de reducir un identificador (un acceso a una variable o llamada a función dentro de una expresión) la gramática no era capaz de distinguir si se trataba de una condición lógica o una operación aritmética, por eso se ha dado un nivel mayor de precedencia a las operaciones lógicas y menor a las aritméticas bajo el mismo subárbol.

Además, dado que las llamadas o declaraciones de funciones no tienen por qué incluir paréntesis, no se hace distinción entre el acceso a un dato y el nombre de una función.

Será tarea del analizador semántico identificar qué nombre corresponde a un dato y cuál a una función, así como asegurarse de que las operaciones realizadas entre distintos tipos de datos son válidas.

Control de errores

El analizador sintáctico resultante incluye también cierta detección de errores haciendo uso del token *error* e instrucciones en código C para notificar al programador. Los errores detectados pueden ser:

- Falta de paréntesis en expresiones o funciones
- Operadores sobrantes en expresiones
- Falta de dos puntos ‘:’
- Falta de la palabra clave “fin” al final de alguna estructura

Tabla de símbolos

Para apoyar la fase de análisis sintáctico y otorgar contexto a la gramática que nos proporciona Bison se implementó una tabla de símbolos. La estructura de datos, implementada como una pila de structs, lleva registro de las variables y funciones definidas en el script que se está analizando.

Cada struct nodo de la pila contiene el nombre de la variable, el tipo (si es un tipo primitivo, una función o un tipo desconocido provisionalmente) y el número de parámetros. En caso de no ser una función, el número de parámetros asignado es -1. Existe un nodo especial de tipo ContextSeparator que se utiliza para separar el contexto más externo del más interno (el contexto global del contexto de definición de una función, por ejemplo). Se han definido funciones pushContext y popContext que se aprovechan de estos nodos especiales para entrar y salir de un contexto concreto.

El analizador aprovecha el número de parámetros almacenado para comprobar si el número de parámetros pasado en una llamada es correcto y dar un error en caso contrario. También se comprueba si al invocar una variable esta ha sido definida.

Hubo que redefinir algunas reglas de la gramática de Bison para distinguir entre la cabecera de la definición de una función y la llamada a una función, que pueden llegar a ser prácticamente iguales salvo por un ‘:’ al final de la cabecera de la definición.

Generación de código

Se programó un compilador que traduce instrucciones del lenguaje luma a instrucciones para la máquina virtual Q. También se añadieron funciones en Qlib para mostrar números de coma flotante por consola y para obtener del usuario númerosenteros y flotantes.

Estructuras de datos

El compilador se apoya en varias estructuras de datos, fundamentalmente algunas pilas con ligeras variaciones entre sí y una lista de cadenas de caracteres, todas ellas programadas desde cero en C.

- La tabla de símbolos. Una pila donde se almacenan funciones y variables declaradas en el programa luma y valores de interés para las mismas, como direcciones de memoria, número de parámetros, tipo de la variable o tipo de retorno.
- Una pila de etiquetas para las condiciones si-sino y otra para los bucles mientras.
- Una lista de char* donde se almacenan las líneas que se escribirán en el fichero final en lenguaje Q. Tener el fichero primero en una lista aporta flexibilidad a la hora de colocar a conveniencia unas líneas antes o después de las ya insertadas.

Modificaciones del lenguaje

Para acomodar la traducción al conocimiento que se tenía sobre Q y el tiempo disponible hubo que hacer algunas modificaciones en Luma que pasarán a detallarse a continuación:

- Los parámetros de las funciones ahora deben declarar su tipo de dato (entero, decimal, caracter o booleano). Se añadió esta restricción porque en tiempo de compilación no es posible inferir qué tipo de dato se pasa al llamar a una función sin estar declarado. Si el programa fuera un intérprete probablemente se podría inferir al ejecutar las instrucciones paso por paso.
- Las cadenas o String ya no son un tipo de dato. Por tanto no se pueden operar en expresiones ni concatenar.
- Como consecuencia de lo anterior, las llamadas a “escribe” solo pueden ser de dos tipos: Con un parámetro de texto o con una expresión. Se pueden mostrar cadenas de caracteres, que se escriben durante la compilación en la zona de memoria estática del programa Q, pero no se pueden concatenar. Si se desea

mostrar texto y expresiones en la misma línea, habrá que utilizar varios "escribe" uno detrás de otro.

- Las llamadas a "lee" deben declarar también el tipo que se va a leer y solo pueden ser de dos tipos: entero o decimal (Integer o Float). Por razones similares a los anteriores dos puntos.

Los programas de prueba se modificaron acorde a los cambios y se corrigieron algunos errores que se hicieron evidentes al compilar. Además, se añadió un programa de prueba para probar la recursividad.