UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Conceptos básicos de desarrollo de aplicaciones con MetaTalk

Apellidos, nombre	Agustí i Melchor, Manuel (magusti@disca.upv.es)
Departamento	Departamento de Informática de Sistemas y Computadores
Centro	Escola Tècnica Superior d'Enginyeria Informàtica Universitat Politècnica de València

1 Resumen de las ideas clave

Este artículo es una breve revisión a cuestiones relacionadas con los aspectos más básicos del lenguaje *MetaTalk, [1]*, de forma práctica. Este lenguaje es propio de la aplicación *MetaCard* enfocada al desarrollo de aplicaciones portables entre diferentes plataformas.

En este trabajo se mostrarán los elementos básicos del lenguaje de programación *MetaTalk. Este* acompaña y complementa a la herramienta de autor *MetaCard* en la realización de aplicaciones, bajo el paradigma de desarrollo de aplicaciones multimedia de "tarjetas y guiones" (*cards & script*).

2 Objetivos

Fuera del rigor propio de la descripción exhaustiva de un lenguaje de programación, me ceñiré a lo que la experiencia en el desarrollo de aplicaciones con este lenguaje me sugiere. Se ofrecen retazos de código sin una explicación profusa para obligar al lector a tomar un papel activo y de exploración que acompañe la lectura de este documento.

Se exponen pequeños listados de código relacionados con las diferentes órdenes que se exponen para dar pie al usuario a que investigue su uso, ejecutando la aplicación, probando el código y consultando la ayuda incluida dentro de la propia herramienta. No espere largas y detalladas explicaciones del código: sea activo, pruébelo.

El objetivo global es presentar, de forma práctica, algunas de las construcciones e instrucciones que más sorprenden al desarrollador acostumbrado a un lenguaje imperativo o al que viene usando las metodologías orientadas a objetos. Para ello, nos vamos a centrar en:

- Introducir el concepto de contenedor.
- Exponer las sentencias de control de la secuencia de la ejecución.
- Introducir las estructuras de datos relacionadas con las listas y los vectores.

3 Introducción

No espere una iniciación a la programación, si no puntos de conexión con sus referencias anteriores a algún lenguaje de programación. Si necesita empezar con lo básico, como referencia le sugiero la serie de pilas *MetaTalk Programmer* de MetaCard Corp. Esta "serie" está compuesta por dos pilas, que se muestran en la fig. 1: *`mtp.mc`* para el aprendizaje autónomo de cuestiones de programación y el *`mtpguide.mc`* para ayudar a planificar y dirigir esta actividad de forma externa al estudiante. Las puede encontrar en el sitio *FTP de MetaCard*[1]. Entre bromas y veras, *MetaTalk Programmer*, le obligará a seguir

[1] ftp://ftp.metacard.com/MetaCard

una trayectoria de aprendizaje en la que avanzará conforme vaya completando etapas.

El contenido de este documento versará sobre ejemplos de uso de variables, funciones e instrucciones de control de flujo y vectores, Atendiendo a presentar así las similitudes de *MetaTalk* respecto a los usos típicos de lenguajes imperativos y de los orientados a objetos. Puesto que *MetaTalk* se puede considerar situado en un punto intermedio de estos dos enfoques.

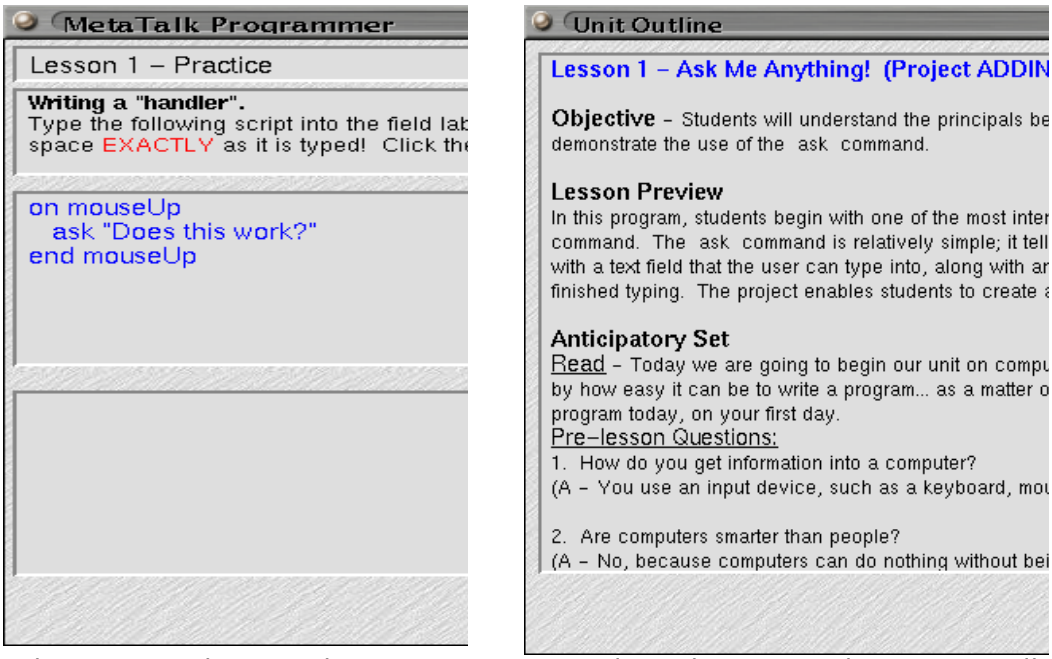

Figura 1: Puede aprender a programar con el propio MetaCard y su MetaTalk Programmer (izquierda) o dirigir este proceso con la ayuda de las lecciones del (derecha) MetaTalk Programmer Teacher's Manual.

Antes de empezar, hay que puntualizar algunos detalles de sintaxis peculiares de este lenguaje (aunque no únicos, ciertamente) que son aplicables a cualquier orden que se escriba:

- No existe un carácter de terminación de una sentencia o instrucción, el salto de línea es el que lo indica en un *script* (el código asociado a cualquier objeto). La única excepción es en la "*Message Box*"; allí, en una sola línea, se pueden introducir varias órdenes seguidas. En ese caso, para separarlas se utiliza el punto y coma (";").
- Por cuestiones de legibilidad, uno puede decidir escribir una orden en más de una línea, para evitar que se interprete por el sistema de ejecución como una nueva orden se utilizará el carácter barra invertida ("\").
- Los comentarios, tan importantes en cualquier programa, se denotan con el carácter almohadilla ("#") o con dos guiones seguidos ("--"); aunque existen otras posibilidades como la doble barra (“//”) al estilo del

lenguaje C. Lo más cómodo es, posiblemente, utilizar las opciones a este respecto que tiene el editor nativo (si es el que utiliza) de código.

En MetaTalk sólo hay nueve estructuras de control: **exit**, **if**, **next**, **pass**, **repeat**, **return**, **switch**, **throw** y **try**. Estas son las que dirigen el flujo de ejecución de nuestros programas. Habitualmente no las utilizo todas, así que no se preocupe por tener que memorizar demasiado. Aquí haré poco más que unos breves comentarios con las dos que utilizo día a día y que siempre he de ir a revisar la sintaxis por que tienen varias variantes, sólo copiaré aquí la sintaxis que puede encontrar en la ayuda integrada.

- La sentencia condicional **if** se puede reescribir como resume el listado 1, variando esta en función del número de instrucciones u órdenes de cada rama de la condición.

```
if <condición> then <orden> else <orden>

if <condición>
   then <orden>
   else <orden>


if <condición> then <órdenes> end if

if <condición> then <órdenes> else <orden>

if <condición> then <órdenes> else <órdenes> end if
```

Listado 1. Posibles formatos de escritura de la orden **if**.

- La orden **repeat** se utiliza para hacer bucles y se puede escribir de diferentes formas como se puede ver en el listado 2, Veremos qué son los "*chunk*" en el punto 5.

```
repeat [forever] end repeat

repeat [for] <contado> [times] end repeat

repeat until <condición>

repeat while <condición> end repeat

repeat with <variable> = <valorInicio> to <valorFinal> end repeat

repeat with <variable> = <valorInicio> down to <valorFinal> end repeat

repeat for each <chunk> <variable> in <expresión> end repeat
```

Listado 2: Posibles formatos de la orden **repeat**.

- El **switch** es interesante cuando una expresión se puede evaluar a varios valores y así agruparlos por rangos o intervalos.
- Para reaccionar ante errores en ejecución (*exception handling*) están **throw** y **try** que, respectivamente, genera una condición de error o ejecuta una orden en modo "protegido" por si sucede una condición de error. Para recoger estas situaciones se acompañan del inseparable **catch**. Las utilizo poco, lo admito, intento hacer las comprobaciones pertinentes antes, con la sentencia condicional **if**.

- La orden **return**, se puede utilizar para devolver un resultado en una función. También se puede utilizar como una salida rápida del flujo del programa, aunque no es recomendable para mantener la estructura del código. Con **exit** ocurre algo similar, pero permite la salida del programa en ejecución.
- La orden **pass**, se utiliza cuando se quiere actuar sobre la jerarquía de mensajes. Esta jerarquía se refiere el orden por defecto en que los objetos reciben un evento dado. Cuando un objeto ejecuta alguna instrucción como respuesta a un evento, por defecto, ya no es propagado. Se utiliza tanto para romper la secuencia natural, como para dar paso al siguiente nivel después de haber procesado un mensaje . Lo admito, esta la he utilizado bastante.

4 Contenedores

Los contenedores (**containers**), pueden almacenar tanto valores alfabéticos como númericos y en MetaCard/MetaTalk se definen cinco clases: campos de texto, botones, imágenes, direcciones *URL* y variables.

Ejecute las siguientes sentencias desde la "*Message Box*", por simplicidad en la exposición e ir experimentando con las características y diferencias de MetaTalk sin necesidad de crear una aplicación nueva en cada caso. En general, los contenedores admiten ser:

- Destinatarios de un flujo de datos, por ejemplo:

  ```
  put "hola" into varAux
  put 3 into varAux
  put "hola" && 3 into varAux
  put URL "http://www.w3c.org" into varAux
  ```

- El origen de un flujo de datos, por ejemplo para ver el resultado de cada sentencia vista anteriormente habrá de ejecutarse la orden

  ```
  put varAux
  ```

- Manipulados “a trozos” (*chunks*). Hasta ahora sólo he utilizado el modificador **into** que hace que se borre cualquier contenido anterior. Pero, como seguramente ya habrá intuido, la ayuda de la orden **put** le confirmará que también puede escribir al principio o al final de lo que haya en el contenedor. Este a su vez puede ser direccionado por líneas, items, palabras o caracteres. Por lo tanto se admiten desde procesos globales a otros de detalle fino.

  ```
  put 33 into varAux;  put "Antes del que n'hi ha" before varAux; put varAux
  ```

En otros lenguajes de programación se utiliza la orden **write** para escribir un texto. Aquí se puede utilizar también para escribir en ficheros o a un *socket*. Aquí utilizaremos la orden **put** por su mayor versatilidad y así reduciremos el número de órdenes a aprender.

Habrá que tener presente que existen diferencias particulares asociadas a cada tipo de contenedor:

- Los campos de texto, los botones y las imágenes son capaces de mostrar la información que contienen, cada uno con sus peculiaridades. En particular, en el caso de los botones se utiliza para crear menús.
- Los campos de texto, además pueden permitir al usuario la edición de su contenido y recuperar su contenido con o sin formato, usando las propiedades *htmlText* o *text* respectivamente.
- Las imágenes son contenedores de datos binarios y, si el formato es soportado por la herramienta, se muestran en pantalla sin más. Es posible procesar los datos con la propiedad *imageData* y obtener así los bytes que componen cada punto de una imagen y, por supuesto, modificarlos si es el caso.
- El contenedor de tipo URL sirve para acceder a ficheros en la máquina local y también a máquinas remotas. Para ello se especificará el protocolo adecuado. Por ejemplo:

  ```
  put varAux into URL("file:tmp.txt")
  ```

  volcará sobre el fichero *tmp.txt*, en el directorio actual, el contenido de la variable *varAux* La operación contraria se realiza simplemente con:

  ```
  put URL("file:tmp.txt") into varAux
  ```

  que leerá el contenido del fichero, en el directorio actual, y lo dejará en la variable indicada.

No es obligatorio declarar las variables, pero puede ayudar para detectar erratas al escribir el nombre de una variable. En caso de no estar declarada, el comportamiento por defecto es crear un variable local con el nombre encontrado y que, inicialmente, tiene contenido vacío. Al pedir al sistema que compruebe si están declaradas las variables que se usan. el sistema mostrará un error al encontrar una variable no declarada y permitirá detectar errores habituales de ortografía en la escritura del código.

Los contenedores en general y las variables en particular, en *MetaTalk*, no se definen con un tipo de datos que, en los lenguajes fuertemente tipados, sería comprobado estrictamente durante su uso. Lo único que se puede declarar de ellas es su ámbito de validez, esto es: si sólo es visible y modificable su contenido en un *script* (**local**) o en cualquier otro *script* en que se repita la declaración(utilizando la orden **global**) de la variable.

Existe otra acepción de variables, las de entorno. Estas hacen referencia a las que el sistema operativo ofrece a una aplicación en ejecución. Su número y nombre depende, por tanto, de su plataforma de ejecución. Entre otros usos, sirven para acceder a parámetros que se le pasen en la línea de órdenes al programa.

5 Vectores

Los vectores (*arrays*), junto con las listas, son dos elementos estructurados habituales de un lenguaje de programación. En *MetaTalk* se pueden utilizar

tanto vectores indexados con valores numéricos, como con valores textuales (también llamados diccionarios).

Se trabaja, por defecto, a modo de **listas**: puesto que las variables no tienen un tamaño predefinido y pueden crecer o disminuir con los contenidos que se les asignan, el número de elementos de una variable depende del uso que se le de en tiempo de ejecución. En este caso el siguiente o el antecesor se han de traducir a operaciones con los trozos (***chunks***) que permite direccionar el lenguaje.

```
local codics


on mouseUp
 repeat with i = 1  to 3
   put (i*10+1) into codics[ i ]
 end repeat


 put the keys of codics


  repeat with i = 1  to the number of lines of (the keys of codics)
    answer i & return & codics[ i ]
  end repeat
end mouseUp
```

Listado 3. Ejemplo de uso de vectores con índices numéricos.

Los elementos de un vector se pueden **indexar a partir de valores numéricos** escritos como es usual entre corchetes. Es la opción más habitual. El listado 3 muestra un ejemplo de escritura en un vector, una sencilla asignación de valores numéricos en este caso. Cómo acceder, para lectura, a los contenidos del vector. completa el código. Se realiza mediante la orden *keys*, que describiremos al final de este apartado y compararemos su uso en los dos tipos de vectores.

El número de dimensiones de los vectores no está limitado, así que se pueden construir matrices (*matrix*) y operar con ellas con las funciones que incorpora el lenguaje, véase listado 4 que muestra ejemplos de acceso a los elementos de la matriz para escritura y lectura (utilizando índices numéricos en este caso, entre corchetes), así como operaciones multiplicación entre matrices (***matrixMultiply***) o de consulta de sus dimensiones (***extents***).

La órden **extents,** permite obtener el número de componentes del vector; en realidad el índice menor y el mayor utilizados. Si tuviese más de una dimensión, cada rango de valores aparece separado del siguiente por un salto de línea.

```
local matriuA, matriuB, matriuC, dimensio

on mouseUp
  repeat with i = 1  to 3
    repeat with j = 1  to 3
      put (i&j) into matriuA[i,j]
      put 2 into matriuB[i,j]
    end repeat
  end repeat

  put matrixMultiply(matriuA, matriuB) into matriuC
  put extents(matriuC) into dimensio
  put "a" && extents(matriuA) && return &\
        "b" && extents(matriuB) && return &\
        "c" && dimensio

  repeat with i = 1  to the second item of line 1 of dimensio
   repeat with j = 1  to the second item of line 2 of dimensio
    answer i & "," & j && matriuC[ i,j ]
   end repeat
  end repeat
end mouseUp
```

Listado 4. Ejemplo de operaciones con matrices.

También es posible el uso de vectores como diccionarios o **vectores asociativos** que se indexan a partir de valores textuales. Son una variante de los vectores enunciados previamente, pero ahora los elementos índice no son numéricos sino alfabéticos. El listado 5, muestra cómo se puede generar un vector asociativo a partir de una cadena de caracteres, cómo se pueden ordenar los elementos de índice (el criterio no tiene por qué ser alfabético, pero era el más sencillo) y cómo acceder a los valores almacenados.

Por si lo está pensado: sí, existe una orden **combine** que hace la operación inversa a **split**. Sencillo, ¿verdad?

```
local codicsTecles="codic_a=97&codic_e=101&codic_i=105&codic_o=111&codic_u=117"

on mouseUp
```

```
put "Inicialment:" && codicsTecles
answer "Antes" & return & keys( codicsTecles ) titled "Antes"
split codicsTecles by "&" and "="
put keys( codicsTecles ) into varAux
sort lines of varAux
answer "Despues" & return & varAux titled "Despues"
repeat with i = 1  to the number of lines of varAux
 answer codicsTecles[ (line i of varAux)  ] && return & (line i of varAux)
end repeat
end mouseUp
```

Listado 5. Ejemplo de vectores asociativos.

La función **keys** también obtiene el número de componentes, además de los valores de los índices. Estos, en principio, pueden ser cualesquiera valores textuales que defina el usuario. Se ha introducido aquí para comparar el tratamiento con los vectores asociativos. El interés de estos radica en la facilidad con que se implementan los diccionarios y otros objetos que consisten en indexar los elementos por una clave no numérica.

6 Paso de mensajes y funciones

En *MetaTalk*, las cosas no suceden hasta que llega un evento que las desencadene, por ejemplo, un evento del sistema como una pulsación de un botón del ratón, una tecla que ha sido pulsada o liberada, el redimensionar una ventana, que haya pasado una cantidad de tiempo, etc.

El objeto al que le llega este evento, reaccionará ante él si en su código está hacerlo, esto es, si dispone de un manejador para ese evento. Por ejemplo el listado 6, muestra el manejador del habitual evento "cuando se pulse un botón del ratón" (***mouseUp***).

Como no es obligatorio y generalmente no lo usamos, igual no se había dado cuenta el lector de este hecho: se pueden pasar parámetros a un evento. En el ejemplo mostrado (listado 6) se puede ver que se introduce el uso de parámetros, al haber dado nombre a uno junto a la definición del manejador.

```
on mouseUp quinPulsat
 put "S'ha pulsat el botó" && quinPulsat
end mouseUp
```

Listado 6. Ejemplo de uso de manejadores.

A diferencia de los eventos, una función (**function**) tiene una sintaxis ligeramente diferente, pero, sobre todo, se diferencia en su semántica: se ejecuta y devuelve un valor; que se recoge si se quiere, claro. Así que aparecerán en una evaluación o expresión, aunque podrían aparecer como una

orden. El listado 7 muestra un código que puede ser asociado a un botón y que ilustra el uso de funciones y pase de parámetros. Se puede diferenciar el código de un manejador del evento **mouseUp** que llama a una función. Esta, recibe dos parámetros numéricos y devuelve uno alfabético. Aunque no sea muy útil ese código de "calculaLaMedia" quiere ilustrar las posibilidades de una función; y ahí tiene algunas.

```
function calculaLaMedia x y
  return "La media de (“&& x & ”,” & y && “ se calcula como  ((" & x & "*" & y & ") / 2)"
end  calculaLaMedia


on mouseUp
 put calculaLaMedia(2, 4)
end mouseUp
```

Listado 7. Ejemplo de uso de funciones.

7 Conclusiones

En este trabajo se han mostrado los elementos básicos para la realización de aplicaciones bajo el paradigma de desarrollo de aplicaciones multimedia de "tarjetas y guiones" (*cards & script*).

Las explicaciones se han acompañado de código para poder experimentar de forma práctica con la exposición. El lector interesado puede hacerse así una idea concreta de estas herramientas.

Hay algunas cosas que se nos han quedado por el camino, le sugiero investigar estas dos, por brevedad y por lo útiles que me han sido a mi:

- La orden **do** permite ejecutar una orden que se haya construido como resultado de la propia ejecución: por ejemplo, concatenando cadenas de caracteres y contenidos de variables.
- Es posible, y en determinadas ocasiones más interesante, realizar un bucle sólo con eventos programados en el tiempo o mientras se den ciertas condiciones.

8 Bibliografía

[1] MetaCard <http://www.metacard.com/>