Karel y Recursión

I. Entendiendo la Recursión

"**Recursión** es la forma en la cual se especifica un proceso basado en su propia definición. Siendo un poco más precisos, y para evitar el aparente círculo sin fin en esta definición, las instancias *complejas* de un proceso se definen en términos de instancias más *simples*, estando las **finales** más simples definidas de forma explícita."

Wikipedia

La recursión es una herramienta muy poderosa en el uso de Karel.
La forma de aplicar la recursión en Karel es definiendo una instrucción en términos de sí misma. Si aún no queda del todo claro este concepto, muy posiblemente este ejemplo lo aclarará:

Código 1 – Ejemplo de Recursión

```
1 define nueva-instruccion coge-zumbadores como
2 inicio
3       si junto-a-zumbador entonces
4       inicio
5               coge-zumbador;
6               coge-zumbadores;
7       fin;
8 fin;
```

No es muy dificil ver que lo que hace esta instrucción es tomar todos los zumbadores que hay en una casilla. En la línea 3 comprueba si hay zumbadores, si los hay, toma un zumbador (línea 5) y la función **se llama a si misma** (línea 6).
Volviendo a la definición formal de recursión, se dice que se definen las instancias complejas en términos de instancias mas simples, y las más simples de todas, definidas de forma explícita.
En el programa anterior se puede ver sin mucho problema que cada que se llama a la instrucción "coge-zumbadores" hay un zumbador menos que la ultima vez que se llamó a la instrucción "coge-zumbadores", es decir, la instancia compleja (coge-zumbadores con x zumbadores en la casilla), se está definiendo en términos de una instancia mas simple(coge-zumbadores con x-1 zumbadores en la casilla). Y finalmente, si hay 0 zumbadores en la casilla, no realizará acción alguna, esto es, la instancia mas simple, definida explícitamente.
Podría estarse preguntado, ¿Qué sucede si alguna de estas condiciones no se cumplen pero la instruccion sigue definiendose en términos de sí misma?.
En caso de que eso suceda, la recursión pasaría a ser un bucle interminable.

Código 2 – Bucle infinito (falsa idea de recursión)

```
1 define nueva-instruccion bucle como
2 inicio
3       bucle;
4 fin;
```

Si fuera posible seguir las especificaciones del lenguaje de Karel al pie de la letra, una vez llamada la función bucle el programa jamas terminaría.
Sin embargo, por cuestiones de memoria el programa en la practica si termina, pero no con una terminación normal, sino con un error llamado “desbordamiento de pila”.
Para entender a que se refiere con ésto, es necesario imaginar la memoria de la computadora donde Karel se está ejecutando.
Al llamar a una instrucción, obviamente Karel debe de “recordar” desde que línea fue llamado.
He aquí un ejemplo muy familiar:

Codigo 3 – Ejemplo de llamada a instrucción

```
1 iniciar-programa
2    define-nueva-instruccion gira-derecha como
3    inicio
4        repetir 3 veces
5            gira-izquierda;
6    fin;
7    inicia-ejecucion
8          deja-zumbador;
9          gira-derecha;
10         deja-zumbador;
11         apagate;
12   termina-ejecucion
13 finalizar-programa
```

Aquí, Karel inicia en la línea 7, deja un zumbador al llegar a la línea 8, y al llegar a la línea 9, se mueve súbitamente a la línea 4, repite 3 veces la línea 3. ¿Y luego qué?, salta a la línea 10, ¿por qué motivo?, obviamente porque la instrucción fue llamada desde la línea 9.
Para que Karel supiera a qué linea debía de ir despues de terminar de ejecutar “gira-derecha” necesitó recordarla, y eso implica ir guardando en algún lugar de la memoria la línea desde la cual se llamó a la instrucción actual.
Ahora, la pregunta obligatoria es ¿qué es lo que sucede cuando se llama a una instrucción dentro de otra instrucción?, obviamente no se puede olvidar desde dónde fue llamada la instrucción actual, y tambien es necesario recordar desde dónde se está llamando a la nueva instrucción, por lo tanto se tienen que guardar las 2 posiciones en memoria, cada que se llame a una instrucción dentro de otra, una nueva posición en memoria debe de guardar la línea desde la cual se este llamando a la instrucción.
Sin embargo, cuando una instrucción termina su ejecución, ya no es necesario seguir “recordando” desde donde se llamó a la instrucción.
De esa forma se puede saber que Karel guarda en memoria una especie de “lista” indicando desde donde fueron llamadas cada una de las instrucciones en el respectivo órden que fueron llamadas, cada que una instrucción nueva comienza su ejecución, un nuevo número se agrega al final de la “lista”, y cada que una instrucción termina su ejecución, un número se quita del final de la “lista”.
Bueno, pues a esta “lista” se le denomina pila o stack, la pila tiene un tamaño máximo, y si el programa sobrepasar este límite de memoria destinado a la pila, el programa súbitamente terminará.
Esto lleva a la pregunta: ¿Cuál es el tamaño máximo de pila de Karel? ¿es fijo o varía dependiendo de la computadora donde se ejecute?.

El tamaño máximo de la pila de Karel es fijo, y es exactamente de 5000 llamadas a instrucción anidadas, si alguien lo quiere comprobar, el siguiente código le será útil:

Código 4 – Soporte de Karel antes del desbordamiento de pila

```
1 define nueva-instruccion bucle como
2 inicio
3       deja-zumabdor;
4       bucle;
5 fin;
```

Al ejecutarlo asegurese de tener suficientes zumbadores en la mochila, y verá como después de colocar 5000 zumbadores el programa termina con el error de desbordamiento de pila.

II. Usando la Recursión en Karel para recuento

Si eres observador, seguramente te habrás dado cuenta que la recursión que se muestra en el código 1 es innecesaria e incluso puede causar un desbordamiento de pila si hay mas de 5000 zumbadores en la casilla que se está llamando.
Bien podría ser sustituido por esto:

Código 5 – Lo mismo que Código 1 pero eliminando la recursión

```
1 define nueva-instruccion coge-zumbadores como
2 inicio
3       mientras junto-a-zumbador hacer
4               coge-zumbador;
5 fin;
```

Sin embargo, hay ocaciones en que la recursión resulta muy útil.
La principal utilidad de la recursión en Karel es la de realizar una operación luego de efectuar la llamada recursiva; un patrón general sería esta forma:

Pseudocódigo 1

```
define-nueva-instruccion (nombre-de-la-instruccion) como
inicio
        (bloque-de-instrucciones-1)
        (llamada-recursiva)
        (bloque-de-instrucciones-2)
fin;
```

Aquí se entiende que en “llamada recursiva” ya está definida una condición que indica cuando dejará de llamarse la función a sí misma.
Si la función se llama a sí misma x veces, el bloque de instrucciones 1 se ejecutará x veces, y el bloque de instrucciones 2 tambien se ejecutará x veces después.
Nótese que este “patrón” NO tiene los mismos resultados que este otro:

Pseudocódigo 2

```
define-nueva-instruccion (nombre-de-la-instruccion) como
inicio
        (bloque-de-instrucciones-1)
        (bloque-de-instrucciones-2)
        (llamada-recursiva)
fin;
```

En el pseudocódigo 2, tanto el bloque de instrucciones 1 como el bloque de instrucciones 2 se ejecutan mientras la condición para que la instrucción se siga llamando a sí misma sea verdadera.
Sin embargo, en el pseudocódigo 1, el bloque de instrucciones 1 se ejecuta mientras la condición para que la instruccion se llame a si misma sea verdadera, sin embargo, el bloque de instrucciones 2 se ejecuta por primera vez cuando dicha condición es falsa, y las siguientes veces se ejecuta independientemente de si la condición es verdadera o falsa. En esta propiedad radica el poder que tiene la recursión en Karel.

Problema de Ejemplo 1

Datos

Originalmente Karel se encuentra en la casilla (1, 1) orientado hacia el norte.
Karel tendrá infinitos zumbadores en la mochila.
En la posición 1, 1 habrá 0<=x<=99 zumbadores.
No habrá mas muros de los predeterminados en un mundo nuevo de Karel.

Problema

Escribe un programa que cuando termine su ejecución, Karel se encuentre en la casilla (1, x+1).

Solución

Una manera iterativa de hacerlo podría ser tener una columna de zumbadores en la avenida 1, e ir quintando un zumbadores de la casilla (1, 1) e irlos poniendo en la parte superior de la columna de uno por uno de tal forma que al final haya una columna de x zumbadores y Karel simplemente se deba colocar en la parte superior de la columna.
Sin embargo hay una solución mucho mas sencilla.
Karel puede tomar todos los zumbadores que haya en la casilla y luego por cada zumbador que tomó, avanzar un paso hacia delante.
Para hacer esto es necesario usar recursión.
El “bloque de instrucciones 1”, en este caso sería “coge-zumbador”, y el “bloque de instrucciones 2” sería “avanza”, Es decir, primero tomará todos los zumbadores y posteriormente avanzará tantas veces como zumbadores tomados.
El siguiente código resuelve este problema con la solución recursiva descrita

Código 6 – Solución al problema de ejemplo 1

```
1  iniciar-programa
2    define-nueva-instruccion funcion como
3    inicio
4        si junto-a-zumbador entonces
5        inicio
6            coge-zumbador;
```

```
7           funcion;
8           avanza;
9       fin;
10  fin;
11  inicia-ejecucion
12      funcion;
13      apagate;
14  termina-ejecucion
15 finalizar-programa
```

Si no te queda claro el funcionamiento del codigo anterior, es recomendable implementarlo y observar detenidamente cómo actua, pues es el problema mas básico y sencillo que se aborda en este tutorial.

Problema de Ejemplo 2

Datos

Originalmente Karel se encuentra en la casilla (1, 1) orientado hacia el este.
Hay un solo zumbador en el mundo, en la casilla (2x+1, 1).
No habrá mas muros de los predeterminados en un mundo nuevo de Karel.

Problema

Escribe un programa que cuando termine su ejecución, Karel se encuentre en la casilla (x+1, 1).

Solución

Aquí por cada par de pasos que Karel de al frente, deberá de dar un paso hacia atrás.
Una forma de lograr esto, sería asegurandose de que al dar un paso atrás, Karel siempre se encuentre orientado hacia el oeste y al dar 2 pasos para el frente, Karel siempre se encuentre orientado al este.
Sin embargo, una forma más practica de hacerlo puede ser avanzar en línea recta, y luego de que la instruccion se llame a si misma por última vez, dar media vuelta y los pasos posteriores siempre serán hacia el oeste.
El siguiente código ilustra la solución recursiva que acaba de ser descrita:

Código 7 – Solución al problema de Ejemplo 2

```
1       iniciar-programa
2           define-nueva-instruccion funcion como
3           inicio
4               si no-junto-a-zumbador entonces
5               inicio
6                   avanza;
7                   avanza;
8                   funcion;
9               fin
10              sino
11              inicio
12                  gira-izquierda;
13                  gira-izquierda;
14              fin;
15              avanza;
```

```
16        fin;
17        inicia-ejecucion
18           funcion;
19           apagate;
20        termina-ejecucion
21     finalizar-programa
```

Problema de Ejemplo 3

Datos

Inicialmente Karel tendrá 0 zumbadores en la mochila y se encontrará en la posición (x, y).
Habrá un solo zumbador en el mundo y se encontrará en la posición (1, 1)
Los unicos muros que habrán, serán los que delimitan el perimetro del mundo.

Problema

Karel deberá colocar el zumbador en la posición (x, y) y posteriormente terminar su ejecución.

Solución

Aquí lo unico que hay que hacer es avanzar hacia el sur hasta encontrar una pared, luego al oeste hasta encontrar una pared, tomar el zumbador y por cada paso que se dio hacia el oeste, dar un paso hacia el este, por cada paso que se dió hacia el sur, dar un paso hacia el norte, de esa manera Karel podrá ir a la posición (1, 1) y regresar a la posición inicial.
Como aquí hay dos tipos de movimientos(norte-sur y este-oeste) sería conveniente implementar dos instrucciones, donde la ultima llamada recursiva de la primera inicie con la ejecución de la segunda.
Nuevamente, si quedó alguna duda de la solución, el codigo tal vez pueda aclararla:

Código 7 – Solución al Problema de Ejemplo 3

```
1     iniciar-programa
2        define-nueva-instruccion horizontal como
3        inicio
4           mientras no-orientado-al-oeste hacer
5              gira-izquierda;
6           si frente-libre entonces
7           inicio
8              avanza;
9              horizontal;
10             mientras no-orientado-al-este hacer
11                gira-izquierda;
12             avanza;
13          fin
14          sino
15          inicio
16             coge-zumbador;
17          fin;
18       fin;
19       define-nueva-instruccion vertical como
20       inicio
21          mientras no-orientado-al-sur hacer
```

```
22              gira-izquierda;
23          si frente-libre entonces
24          inicio
25              avanza;
26              vertical;
27              mientras no-orientado-al-norte hacer
28                  gira-izquierda;
29              avanza;
30          fin
31          sino
32          inicio
33              horizontal;
34          fin;
35      fin;
36      inicia-ejecucion
37          vertical;
38          deja-zumbador;
39          apagate;
40      termina-ejecucion
41  finalizar-programa
```

Problema de Ejemplo 4

Datos

En la posición (1, 1) habrá *x* zumbadores, y en la posición (2, 1) habrá *y* zumbadores.
No habrá muro entre dichas posiciones.
Inicialmente Karel se encontrará en la posición (1, 1) orientado al este y no tendrá zumbadores en la mochila.

Problema

Escribe un programa que situe a Karel en la posición (2, 1) si $y>x$, de otra forma situe a Karel en la posición (1, 1). Además, al terminar la ejecución, deberá haber exactamente *x* zumbadores en la posición (1, 1) e *y* zumbadores en la posición (2, 1).

Solución

Es posible tomar todos los zumbadores en la posición (1, 1) y volverlos a poner en su lugar, luego por cada zumbador que se tomó de la posición (1, 1), tomar otro zumbador de la posición (2, 1). Y si en algun momento no quedan zumbadores en la posición (2, 1), entonces se puede concluir que (1, 1) x>=y, de otra forma x<y.

Código 9 – Solución al problema de ejemplo 4

```
1   iniciar-programa
2       define-nueva-instruccion compara como
3       inicio
4           si junto-a-zumbador entonces
5           inicio
6               coge-zumbador;
```

```
7               compara;
8               si junto-a-zumbador entonces
9                  coge-zumbador;
10           fin
11           sino
12           inicio
13              mientras algun-zumbador-en-la-mochila hacer
14                 deja-zumbador;
15              avanza;
16           fin;
17        fin;
18        inicia-ejecucion
19           compara;
20           si junto-a-zumbador entonces
21           inicio
22              mientras algun-zumbador-en-la-mochila hacer
23                 deja-zumbador;
24           fin
25           sino
26           inicio
27              mientras algun-zumbador-en-la-mochila hacer
28                 deja-zumbador;
29              gira-izquierda;
30              gira-izquierda;
31              avanza;
32           fin;
33           apagate;
34        termina-ejecucion
35     finalizar-programa
```

III. Aprovechando la Información en la Pila de Karel

Como te habrás dado cuenta, el tamaño de la pila en Karel cuando se deja de cumplir una condición puede ser muy útil para repetir una operación varias veces.
Sin embargo, hasta ahora no hemos aprovechando el contenido de la pila, es decir, la pila contiene la información que dice desde que línea fue llamada la ultima función. ¡Esa información puede ser muy útil!, ya que esa es la unica forma que tiene Karel de “recordar” una cantidad vectorial (una cantidad vectorial es aquella formada por varias cantidades escalares, en este caso las cantidades escalares serían las líneas desde las cuales se realizaron las llamadas a función).

Problema de Ejemplo 5
Prob 3 del X concurso nacional de la OMI

Historia

Enojado por el saqueo de sus tesoros, Karel Sparrow, ha decidido crear un nuevo sistema de

codificación para sus mapas. El nuevo sistema es una secuencia de montones de zumbadores que indican la dirección en la que se debe dar cada paso. Un zumbador significa un paso al norte, dos zumbadores un paso al este, tres un paso al sur y cuatro uno al oeste.

Pasado el tiempo, ya viejo, Sparrow ha regresado a la isla y te ha pedido que lo ayudes a seguir el recorrido indicado por la secuencia de montones. Ayuda a Karel a encontrar su tesoro y ganarás el 10% de su botín.

Problema

Escribe un programa que permita a Karel seguir las instrucciones del mapa. Las instrucciones (secuencia de montones) se encuentran de manera consecutiva a lo largo de la primera fila, comenzando en la columna 1 y siguiendo hacia la derecha.

El punto donde inicia el recorrido siempre es la esquina de la fila 1 con la columna 1.

Las instrucciones del recorrido son tales que nunca te llevarán de vuelta a la primera fila, y el camino jamás se cruzará sobre si mismo.

Consideraciones

- Karel inicia en la esquina de la primera fila con la primera columna mirando al este.
- Las instrucciones terminan cuando encuentras el primer espacio vacío sobre la primera fila o llegas a la pared, un mapa puede tener hasta 100 instrucciones.
- No llevas ningún zumbador en la mochila.
- No hay paredes dentro del mundo ni zumbadores aparte de los de la secuencia de instrucciones.
- No importa la orientación final de Karel.
- Karel debe terminar en la posición final del camino indicado por el mapa.
- No importa si dejas zumbadores en algún lugar del mapa.

Solución

Una solución que puede llegarle a la mente a alguien es trazar el camino y regresar siempre a la fila 1 para ver el siguiente paso, otra posible solución sería ejectuar los “comandos” del mapa, de uno por uno y luego de ejecutar un comando, volver a la fila 1, buscar el siguiente y volver recursivamente al lugar en donde Karel se había quedado en la búsqueda del mapa utilizando la solución al problema 4.

Sin embargo, existe una solución mucho mas sencilla en la que no se tiene que ir y volver de la fila 1, ya que Karel puede “recordar” las instrucciones del mapa guardandolas en la pila,

Como ya habíamos visto antes, al terminar la ejecución de una función, Karel “saltará” a la línea donde se hizo la ultima llamada a función, al terminar la ejecución de esa función, “saltará” a la linea donde se hizo la penultima llamada a función, etc.

Por algo a la pila le llaman estuctura de datos LIFO(last in, first out), el ultimo que entra es el primero que sale.

Con todo esto lo que se pretende decir es que si Karel va a “leer” el mapa en la pila, hay que meter el mapa a la pila al revés, es decir, se mete a la pila la ultima instrucción del mapa, luego la penultima, despues la antepenultima y asi sucesivamente...

Ahora la pregunta obligatoria: ¿Cómo meter las instrucciones a la pila?.

No es muy dificil, lo primero que se tiene que hacer es obviamente posicionarse al final de las

instrucciones del mapa.

Luego de eso, cada casilla puede tener de 1 a 4 zumbadores, es posible hacer la llamada recursiva desde una línea A si se encontró un solo zumbador, desde otra línea B si se encontraron 2 zumbadores, desde otra línea C si se encontraron 3, y desde otra línea D si se encontraron 4. Y cuando Karel "regrese" de la recursividad, si se llamó desde A avanzar al norte, desde B avanzar al este, desde C avanzar al sur y desde D avanzar al oeste. Si aún no te queda claro cómo implementar esto, puedes checar el código:

Código 10 - Solución al Problema de Ejemplo 5

```
1       iniciar-programa
2           define-nueva-instruccion sigue como
3           inicio
4               coge-zumbador;
5               si no-junto-a-zumbador entonces (*Norte*)
6               inicio
7                   si frente-libre entonces
8                   inicio
9                       avanza;
10                      sigue;
11                  fin;
12                  mientras no-orientado-al-norte hacer gira-izquierda;
13                  avanza;
14              fin
15              sino
16              inicio
17                  coge-zumbador;
18                  si no-junto-a-zumbador entonces (*Este*)
19                  inicio
20                      si frente-libre entonces
21                      inicio
22                          avanza;
23                          sigue;
24                      fin;
25                      mientras no-orientado-al-este hacer gira-izquierda;
26                      avanza;
27                  fin
28                  sino
29                  inicio
30                      coge-zumbador;
31                      si no-junto-a-zumbador entonces (*Sur*)
32                      inicio
33                          si frente-libre entonces
34                          inicio
35                              avanza;
36                              sigue;
37                          fin;
38                          mientras no-orientado-al-sur hacer gira-izquierda;
```

```
39                  avanza;
40               fin
41               sino
42               inicio (*Oeste*)
43                  si frente-libre entonces
44                  inicio
45                     avanza;
46                     sigue;
47                  fin;
48                  mientras no-orientado-al-oeste hacer gira-izquierda;
49                  avanza;
50               fin;
51            fin;
52         fin;
53      fin
54      inicia-ejecucion
55         mientras frente-libre y junto-a-zumbador hacer
56            avanza;
57         gira-izquierda;
58         gira-izquierda;
59         si no-junto-a-zumbador entonces
60            avanza;
61         sigue;
62         apagate;
63      termina-ejecucion
64   finalizar-programa
```

IV. Búsqueda en Profundidad

Hasta ahora solamente hemos usado la recursión en Karel de manera que Karel realiza una sola llamada recursiva antes de “regresar”. Sin embargo, es posible hacer mas de una llamada recursiva cada vez que se ejecuta una función.

La búsqueda en profundidad es un algoritmo de naturaleza recursiva basado en visitar todos los estados de un sistema lógico, por ejemplo, hacer que Karel recorra todos los cuadros a los que puede llegar en un mundo.

Problema de Ejemplo 6

Datos

Karel se encuentra en una casa.
La casa en el mundo de Karel es un área delimitada por paredes.
La casa puede tener cualquier forma(incluso muros interiores), pero Karel puede llegar a cualquier parte de la casa.
No hay zumbadores en el mundo de Karel

Karel tiene infinitos zumbadores en la mochila

Problema
Karel quiere ponerle alfombra a la casa, ayuda a Karel a colocar un zumbador en cada espacio vacio de la casa.

Solución

Para tener una idea de cómo hacer eso, vamos poniendonos en el lugar de Karel, si nos encontramos parados en una cuadricula similar a la del mundo de Karel, ¿a qué lugares podemos ir?. La respuesta obvia es: norte, sur, este y oeste; a menos que haya una pared que nos lo impida.

Asi que la unica manera de estar seguros que recorrimos todo es, primero caminar hacia el frente, visitar todo lo que podamos, y volver a donde empezamos.

Luego caminar a la izquierda, visitar todo lo que se pueda visitar desde ahí, y al final volver al inicio.

Y hacer lo mismo para atras y para la derecha.

Sin embargo, necesitamos algo para marcar por donde ya pasamos, la leyenda del minotauro cuenta que Teseo utilizó una cuerda para marcar por dónde ya había pasado una vez que se encontraba perdido en un laberinto.

Ahora, ¿qué puede utilizar Karel? ¡zumbadores!

Así que la idea es, para cada lugar que visite Karel, regrese si ya hay un zumbador ahí, si no lo hay entonces deje un zumbador e intente ir a cada una de las 4 direcciones y en cada dirección que pueda ir llamar a esta funcion recursivamente.

Y nuevamente se muestra aquí el código de una posible implementación del algoritmo mencionado:

Código 11 – Solución al Problema de Ejemplo 6

```
1       iniciar-programa
2           define-nueva-instruccion busca como
3           inicio
4               si no-junto-a-zumbador entonces
5               inicio
6                   deja-zumbador;
7                   repetir 4 veces
8                   inicio
9                       si frente-libre entonces
10                      inicio
11                          avanza;
12                          busca;
13                          repetir 2 veces gira-izquierda;
14                          avanza;
15                          repetir 2 veces gira-izquierda;
16                      fin;
17                      gira-izquierda;
18                  fin;
19              fin;
20          fin;
```

```
21          inicia-ejecucion
22              busca;
23              apagate;
24          termina-ejecucion
25      finalizar-programa
```

Esta solución es una implementación de la búsqueda en profundidad, sin embargo, los problemas de Karel que requieren de búsqueda en profundidad no siempre se presentan de la misma manera, existen problemas de Karel que se resuelven utilizando modificaciones de esta solución, algunos ejemplos de esos problemas son encontrar un camino para ir de la fila 1 a la fila *n* sin pasar por cuadros marcados y escribir un "mapa" del camino en la fila 1, quitar los zumbadores que forman una figura en la que Karel se encuentra sin quitar los zumbadores que forman otras figuras alrededor ó encontrar en un cuarto donde esta Karel una casilla rodeada por 3 paredes.

Sin embargo, se considera que el ejemplo anterior da bases suficientes para resolver los otros problemas de busqueda en profundidad, por ello no se tratarán aquí pero si se invita al lector a que intente resolverlos.

V. Variables en Karel ¿Mito o Realidad?

Tal vez te sorprenda esto, pero Karel tiene soporte para variables; sin embargo, dicho soporte esta bastante limitado; ya que solo se puede utilizar una variable por función, dicha variable se pasa como parámetro.

Aunque no es posible hacer asignaciones directamente a una variable, existen 3 funciones de Karel que operan con variables:

- precede(x) devuelve x-1.
- sucede(x) devuelve x+1
- si-es-cero(x) devuelve verdadero si x=0 y falso en caso contrario

Al declarar una función en Karel es bastante facil permitirle que use una variable como parametro:

define-nueva-instruccion funcion(*nombre-de-la-variable*) como

Para llamar a una función con un parámetro declarado, se puede hacer de la siguiente manera:

funcion(*valor-del-parametro*);
Ahora es posible utilizar la variable en la expresión repetir de la siguiente manera:

repetir *nombre-de-la-variable* veces

En la expresión si de la siguiente manera:

si si-es-cero(*nombre-de-la-variable*) entonces

o bien

si no(si-es-cero(*nombre-de-la-variable*)) entonces

Estos usos, aunque pueden ahorrar código, no ayudan de manera significativa para resolver problemas. La principal utilidad de las variables en Karel, es la capacidad que tienen para incrementarse o

decrementarse al pasarlas como parámetros en una recursión. Para entender mejor esto, es conveniente ver un ejemplo práctico.

Problema de Ejemplo 7

Datos

Karel se encuentra en la casilla (1, 1) orientado al norte.
En la casilla (1, 1) hay *a* zumbadores.
En la casilla (1, 2) hay *b* zumbadores.
Karel tiene infinitos zumbadores en la mochila.
No hay muros en el interior del mundo.

Problema

Karel debe colocar *ab* zumbadores en la casilla (1, 3).

Existen varias maneras de hacer esto, pero la que quizá sea la mas sencilla es tomar recursivamente los *a* zumbadores mientras se incrementa la variable que se pasa como parámetro, luego tomar los *b* zumbadores recursivamente pasando *a* como parámetro pero sin incrementarlo, y cuando "regrese" de la recursión utilizada para tomar *b* zumbadores, colocar en la casilla (1, 3) *a* zumbadores por cada llamada recursiva hecha para tomar los *b* zumbadores, es decir, colocará *a* zumbadores *b* veces, osea *ab* zumbadores.

Codigo 12 – Solución al Problema de Ejemplo 7

```
1       iniciar-programa
2           define-nueva-instruccion agarra-b(a) como
3           inicio
4               si junto-a-zumbador entonces
5               inicio
6                       coge-zumbador;
7                       agarra-b(a);
8                       repetir a veces
9                           deja-zumbador;
10              fin
11              sino
12              inicio
13                  avanza;
14              fin;
15          fin
16          define-nueva-instruccion agarra-a(a) como
17          inicio
18              si junto-a-zumbador entonces
19              inicio
20                  coge-zumbador;
21                  agarra-a(sucede(a));
22              fin
23              sino
24              inicio
```

```
25              avanza;
26              agarra-b(a);
27            fin;
28        fin
29        inicia-ejecucion
30            agarra-a(0);
31            apagate;
32        termina-ejecucion
33    finalizar-programa
```

Como se puede ver, las variables en Karel son una buena alternativa cuando se requiere llevar 2 conteos simulteaneamente. Otras aplicaciones es llevar un conteo y hacer una búsqueda en profundidad a la vez, o llevar un conteo mientras se hacen llamadas recursivas que no necesariamente tienen que ver con el conteo.

VI. Agradecimientos y otras cosas...

Aunque yo escribí este tutorial, no habría sido posible si no existiera la versión de Karel programada por Cesar Cepeda, la cual, a diferencia de las tradicionales, soporta recursividad.

Los problemas de ejemplo 5 y 6 fueron ideados por Marte Alejandro Ramírez Ortegón(Mars).

Y para el lector:

Si ya lograste dominar todo lo que se menciona en este tutorial, ya deberías ser capaz de resolver cualquier problema de Karel que aparezca a nivel estatal o nacional. No quiero decir con esto que la recursividad es la respuesta para todo, sino que para lograr dominar los algoritmos recursivos, ya anteriormente se debió de haber dominado los algoritmos iterativos.

Sin embargo, hay que buscar la sencillez, la cual no siempre es fácil de encontrar(aún a estas alturas), un algoritmo de sencilla implementación tiene mucha ventaja en Karel, ya que es menos propenso a errores.

Tambien hay que tener muy claro en qué momento el algoritmo que se esta desarrollando en la mente ya está listo para ser implementado, y en qué momento hay que aclarar detalles que pueden causar grandes pérdidas de tiempo después si no se aclaran antes de implementarlo.

Tampoco hay que olvidar ser paranoico en cuanto a la revisón del programa.

En la Olimpiada de Informática no solo se necesita poder resolver los problemas, sino que tambien importa el tiempo que se requiere para resolverlos, el cuidado de detalles que son fáciles de pasar por alto y sobre todo leer bien los problemas para entenderlos correctamente; por ello se recomienda altamente que el lector siga resolviendo problemas de Karel en la preparación para una competencia estatal o nacional, cuantos más, mejor.

Tutorial escrito por Luis Enrique Vargas Azcona(Lobishomen)