Tutorial de Karel

Olimpiada de Informática del Estado de Guanajuato

http://www.cimat.mx/oieg

Elaborado por:

Edgar Alfredo Duéñez Guzmán

Marte Alejandro Ramírez Ortegón

Colaboradores en correcciones:

Carlos Francisco Ramírez Gloria

Limo Wan Kenobi

Versión 1.1, Agosto 2005

Índice de contenido

1 El robot y su mundo

En esta primera parte pretendemos introducir varios conceptos básicos de programación. Trataremos de brindar la posibilidad de adquirir los principios de un lenguaje de programación estructurado, tales como la *creación de procedimientos*, *condicionales*, *iteraciones*, etc.

1.1 El mundo de Karel

Karel es un robot que podemos controlar por medio de un programa para que realice cierto trabajo. El mundo de Karel consta de los siguientes elementos:

1.
    - *Calles* (horizontales) y *avenidas* (verticales) que se cruzan en *esquinas*.
    - *Paredes* impenetrables colocadas entre dos esquinas.
    - *Beepers* removibles colocados en las esquinas que emiten un sonido (su grosor es irrelevante).
    - *Bolsa de beepers* (beeper_bag) que Karel lleva consigo.
2.
    - *Karel* nuestro protagonista siempre está en una esquina y mirando al norte, sur, este u oeste. A través de tres cámaras puede ver si se encuentra una pared entre él y las esquinas más cercanas (enfrente, a su derecha y a su izquierda). Su oído le permite detectar el sonido de beepers en la esquina donde se encuentra.

La manera de comunicarse con Karel es por medio de un programa. El problema principal es que lo único que Karel puede hacer es seguir lo que le indiquemos "al pié de la letra". Karel no piensa y no puede darse cuenta de lo que queremos que haga si no sabemos cómo decírselo, para eso es necesario aprender su lenguaje.

2 Primeras instrucciones y programas

Realizar un trabajo o tarea específica con Karel consiste en llevarlo de una situación *original* a una *final* a través de la ejecución de *instrucciones*. Comenzaremos con las instrucciones básicas.

- Cambio de posición
  - **move**: hace un paso en la dirección que está apuntando (puede causar error "apagón" si hay una pared enfrente).
  - **turnleft**: gira a su izquierda 90º (siempre se puede).
- Manipulando beepers
  - **pickbeeper**: recoge un beeper de la esquina donde está parado (puede causar error "apagón" si no hay ningún beeper en la esquina).
  - **putbeeper**: deposita un beeper en la esquina (puede causar error "apagón" si la bolsa de beepers de Karel está vacía).
- Terminando
  - **turnoff**: es el comando que finaliza y apaga a Karel.

IMPORTANTE: Las nuevas funciones deben tener diferentes nombres y NO llamarse igual a una de las funciones básicas.

2.1 Primer programa

Para realizar un programa es necesario escribir con un formato que Karel pueda entender. El código de un programa para Karel debe tener el siguiente formato:

```
class program
        {
        [<Definiciones de funciones>]
        program()
                {
                <Definiciones de las instrucciones a ejecutar>
                turnoff();
                }
        }
```

Código 2.1.1 Estructura básica de un programa.

En la sección de "definiciones de funciones" declaramos los diferentes procedimientos (funciones) que emplearemos en la ejecución del programa. Por otro lado, Toda función va seguida de de paréntesis "("

2.1 Primer programa

")" y las llaves "{" y "}", estas llaves contienen el código a ejecutar por la función. En particular, la función principal (y la única que ejecuta Karel al activarse) se llama "program" y dentro de esta función se ponen las instrucciones que Karel ejecutará al momento de echar a correr el programa. Las palabras reservadas (esto significa que no puedes emplearlas para nombrar una función) de Karel son:

- **class program**: con esta palabra comienza el programa
- **program**: después de esta palabra comenzaran los comandos e instrucciones del programa

Un ejemplo: queremos que Karel, siendo que se encuentra apuntando al este en la calle 2, avenida 2 (Ver figura 2.1.1), realice la tarea de llevar el beeper que se encuentra en la calle 2, avenida 4, a la calle 4, avenida 5, y debe moverse una cuadra más al norte antes de apagarse (Como en la figura 2.1.2).

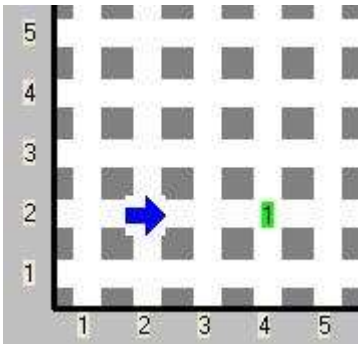

Figura 2.1.1 Karel frente a un beeper

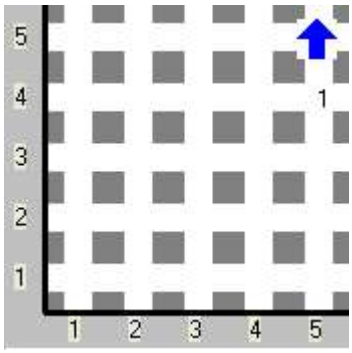

Figura 2.1.2 Karel después de seguir las instrucciones

```
class program
        {
        program()
                {
                move();
                move();
                pickbeeper();
                turnleft();
                move();
                move();
                putbeeper();
                move();
                turnoff();
                }
        }
```

Código 2.1.2 Ejemplo de primer programa.

2.1 Primer programa

Al ejecutar el programa, Karel revisa que el programa no tenga errores y luego procede a llevar a cabo cada una de las siguientes instrucciones entre los *paréntesis*: **{** y **}** de la función **program()** hasta que encuentra un **turnoff()** o hasta que se produce un error "apagón".

2.1.1 Gramática

Es muy importante respetar las reglas de puntuación y gramática para no confundir a Karel. Hay cuatro tipos de palabras o símbolos que Karel entiende:

- *Puntuación* (sólo el punto y coma " ; " después de cada comando).
- *instrucciones nativas* de Karel.
    - move
    - turnleft
    - putbeeper
    - pickbeeper
    - turnoff
- *Palabras reservadas*
    - if, else
    - iterate
    - while
    - define
    - {
    - }
    - &&
    - ||
    - !
- *Pruebas*
    - frontIsClear, frontIsBlocked, leftIsClear, leftIsBlocked, rightIsClear, rightIsBlocked
    - nextToABeeper, notNextToABeeper
    - anyBeepersInBeeperBag, noBeepersInBeeperBag
    - facingNorth, facingSouth, facingEast, facingWest, notFacingNorth, notFacingSouth, notFacingEast, notFacingWest

2.2 Errores

Hay varios tipos de dificultades con las que podemos toparnos al escribir un programa. Una de las partes más complicadas de la programación es el detectar dónde están las fallas de nuestros programas. Afortunadamente, Karel puede mostrarnos algunos errores en nuestro programa.

2.2.1 Apagón

Si Karel se ve impedido de realizar alguna de sus instrucciones básicas, entonces se apaga por si mismo. Esto es un error imputable al que escribió el programa y es mejor apagarse que comenzar a hacer más tonterías. Las instrucciones que puede ocasionar un apagón son:

- **move**: genera un apagón si se le ordena a Karel que avance y existe una pared frente a él
- **putbeeper**: genera un apagón cuando se le pide a Karel que ponga un beeper y su beeper_bag se encuentra vacía
- **pickbeeper**: genera un apagón cuando se le pide a Karel que recoja un beeper pero no existe alguno en tal posición

3 Instrucciones que se repiten

Las instrucciones que realizan ciclos permiten que Karel realice un cierto número de veces una determinada instrucción o un **bloque** de instrucciones delimitado por "{" y "}".

3.1 La instrucción iterate

En lugar de escribir una instrucción repetidas veces, podemos decirle a Karel exactamente el número de veces que queremos que repita cierta instrucción. Esto se hace con la siguiente sintaxis:

```
iterate(<número>)
        <instrucción>[{Bloque}]
```

Código 3.1.1 Sintaxis de la instrucción iterate.

Donde el *número* determina la cantidad de veces que se repetirá. El comando que queremos que se repita debe estar en la línea de abajo (*instrucción*). En caso de haber más de una instrucción ponemos un bloque. Por ejemplo:

```
iterate(3)
        turnleft();
iterate(4)
        {
        turnleft();
        move();
        }
```

Código 3.1.2 Ejemplo de la instrucción iterate con una y dos instrucciones respectivamente.

Las ventajas de usar la instrucción **iterate** son muchas, y entre ellas se cuenta el hecho de que si uno requiere cambiar el código de la instrucción o bloque ejecutado varias veces, en este caso sólo se debe cambiar el bloque una vez. Por ejemplo, si tenemos el programa que recoge 5 beepers en fila, tendríamos:

```
program()
        {
        pickbeeper();
        move();
        pickbeeper();
        move();
```

3.1 La instrucción iterate

```
        pickbeeper();
        move();
        pickbeeper();
        move();
        pickbeeper();
        move();
        turnoff();
        }
```

Código 3.1.3 Programa sin instrucción iterate.

Pero si ahora quisiéramos que recogiera 2 beeper en vez de uno, tendríamos que insertar una instrucción **pickbeeper** antes de cada paso. Este cambio se reduce aumentar un **pickbeeper** dentro del bloque del iterate:

```
iterate(4)
        {
        pickbeeper();
        pickbeeper();
        move();
        }
pickbeeper();
pickbeeper();
```

Código 3.1.4 Utilizando un iterate para simplificar nuestro código.

O incluso, en pos de la claridad, agregar un último paso:

```
iterate(5)
        {
        pickbeeper();
        pickbeeper();
        move();
        }
```

Código 3.1.5 Otra forma de ejecutar nuestra tarea.

4 Extendiendo el lenguaje de Karel

Tenemos la posibilidad de extender el limitado lenguaje de Karel definiendo instrucciones nuevas. La sintaxis es:

```
define <nombre_nueva_instrucción>()
        <instrucción> [{bloque}]
```

Código 4.1 Sintaxis para definir una nueva instrucción en Karel

Donde la *instrucción* puede ser una sola instrucción o un bloque de instrucciones creado con { y }. Esto crea lo que se conoce como una entrada del diccionario de Karel, que es donde está todo lo que Karel puede y sabe ejecutar. Es bueno darles nombres a las instrucciones de acuerdo a la acción que realizan, para simplificarnos el poder entender los programas que escribimos. No se puede usar nombres de instrucciones nativas ni palabras reservadas. Karel hace distinción entre mayúsculas y minúsculas. No se puede usar ningún caracter especial.

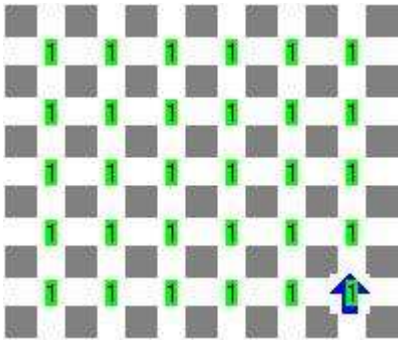

Figura 4.1 Cosecha de beepers.

Por ejemplo, Karel quiere cosechar unas zanahorias que plantó hace algún tiempo. Para esto usamos unas nuevas instrucciones, como:

```
define cosecha2Filas()
        {
        cosecha1Fila();
        siguienteFilaIzquierda();
        cosecha1Fila();
        siguienteFilaDerecha();
        }
```

Código 4.2 Instrucción que cosecha dos filas.

4 Extendiendo el lenguaje de Karel

Continuando con las definiciones sucesivas.

```
define cosecha1Fila()
        {
        iterate(4)
                {
                pickbeeper();
                move();
                }
        pickbeeper();
        }
define siguienteFilaIzquierda()
        {
        turnleft();
        move();
        turnleft();
        }
define siguienteFilaDerecha()
        {
        turnright();
        move();
        turnright();
        }
define turnright()
        {
        iterate(3)
        turnleft();
        }
```

Código 4.3 Definición de nuevas instrucciones que emplearemos en nuestro programa principal.

El programa principal quedaría simplemente como:

```
program()
        {
        iterate(3)
                cosecha2Filas();
        turnoff();
        }
```

Código 4.4 Programa principal para cosechar las zanahorias.

4 Extendiendo el lenguaje de Karel

Siempre se puede escribir un programa sin instrucciones nuevas, pero resulta una larga secuencia de comandos básicos de Karel, y es muy complicado de entender, corregir, modificar, etc. Es muy buen hábito de programación el tener las nuevas instrucciones definidas en la forma más clara y concisa posible para poder reutilizarlas en futuros programas escritos. Incluso es posible tener una librería de múltiples funciones de uso variado, para agregarlas dentro de los nuevos programas escritos. Si se llama a la misma función varias veces, se debe verificar que las condiciones en las que se empezará cada vez son similares. Por ejemplo, en el caso de la cosecha de 2 filas, Karel debe terminar de cosecharlas y estar sobre el primer beeper de la siguiente fila de 2 beepers. En la vida cotidiana usamos este mismo proceso de partir una tarea complicada en tareas más pequeñas que a su vez las dividimos en tareas más simples hasta llegar al punto en que las tareas que efectuamos son triviales.

5 Condicionales

En muchas ocasiones no queremos que Karel realice una serie de instrucciones ya definida, sino que queremos que Karel sea capaz de discernir qué hacer dependiendo del estado actual del mundo. Esto amplía enormemente el poder de Karel, ya que puede interaccionar con el mundo en el que se encuentra.

En particular, si queremos que Karel evite un apagón, podemos hacer que cada vez que intente recoger un beeper, observe primero si puede hacerlo, o que cada vez que queremos que de un paso, revisar que no haya una pared enfrente de él.

Las condiciones posibles son las siguientes 18:

frontIsClear	es verdadero si el frente de Karel esta libre de pared
frontIsBlocked	es verdadero si existe una pared frente a Karel
leftIsClear	es verdadero si en el lado izquierdo de Karel no existe una pared
leftIsBlocked	es verdadero si en el lado izquierdo de Karel existe una pared
rightIsClear	es verdadero si en el lado derecho de Karel no existe una pared
rightIsBlocked	es verdadera si en el lado derecho de Karel existe una pared
nextToABeeper	es verdadero si existe un beeper donde Karel esta parado
notNextToABeeper	es verdadero si no existe un beeper donde Karel esta parado
facingNorth	es verdadero si Karel esta volteando hacia al Norte
notFacingNorth	es verdadero si Karel no esta volteando hacia al Norte
facingSouth	es verdadero si Karel esta volteando hacia el Sur
notFacingSouth	es verdadero si Karel no esta volteando hacia el Sur
facingEast	es verdadero si Karel esta volteando hacia el Este
notFacingEast	es verdadero si Karel no esta volteando hacia el Este
facingWest	es verdadero si Karel esta volteando hacia al Oeste
notFacingWest	es verdadero si Karel no esta volteando hacia al Oeste
anyBeepersInBeeperBag	es verdadero si existe algún beeper en a beeper_bag
noBeepersInBeeperBag	es verdadero si no existes algún beepers en a beeper_bag

5.1 La instrucción if

Para realizar una acción (instrucción) dependiendo del estado actual del mundo, se utiliza la instrucción **if**. Su sintaxis es la siguiente:

```
if( <condición> [bloque condicional] )
        <instrucción> [{bloque}]
```

Código 5.1.1 Sintaxis de la instrucción if.

Por ejemplo:

5.1 La instrucción if

```
if( frontIsClear )
        move();
if( leftIsClear )
        {
        turnleft();
        move();
        }
```

Código 5.1.2 Ejemplo del uso de la instrucción if.

Donde, en cada caso, *condición* es una de las condiciones antes mencionadas. La instrucción (o bloque de instrucciones) que se encuentra inmediatamente después de la instrucción **if** sólo se ejecuta cuando la condición especificada es cierta. De lo contrario ignora a la instrucción (o bloque de instrucciones) y continua con la ejecución normalmente, es decir, no se apaga.

5.2 La instrucción else

Cuando, además de revisar una condición y actuar cuando se satisfaga, queremos realizar una prueba que puede tener dos resultados distintos y actuar de acuerdo con ello, podemos usar la instrucción **else**. Su sintaxis es:

```
if( <condición> [bloque condicional] )
        <instrucción1> [{bloque1}]
else
        <instrucción2> [{bloque2}]
```

Código 5.2.1 Sintaxis de la instrucción else.

En este caso, lo que ocurre es que si la condición es cierta, se ejecuta la instrucción1 (o el bloque1), pero si no lo es se ejecuta la instrucción2 (o el bloque2), e independientemente del resultado, se continua la ejecución del programa.

```
if( frontIsClear )
        move();
else
        turnleft();

if( nextToABeeper )
        {
        pickbeeper();
        move();
        }
```

5.2 La instrucción else

```
else
        putbeeper();
```

Código 5.2.2 Usando la instrucciones if-else.

Es importante notar que en este caso, siempre se ejecuta alguno de las dos instrucciones, pero nunca ambas. Nota la diferencia entre

```
if( frontIsClear )
        move();
else
        turnleft();
```

Código 5.2.3 Código del caso 1

y

```
if( frontIsClear )
        move();
if( frontIsBlocked )
        turnleft();
```

Código 5.2.4 Código del caso 2.

Si se tiene a Karel en la posición de la primera imagen (Figura 5.2.1), resulta que en el primer caso, Karel termina como en la segunda imagen (Figura 5.2.2), pero en el segundo caso, termina como en la tercera imagen (Figura 5.2.3).

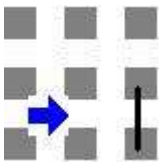

Figura 5.2.1 Caso inicial.

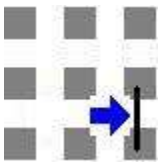

Figura 5.2.2 Final del primer caso.

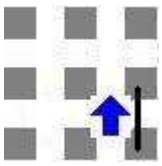

Figura 5.2.3 Final del segundo caso.

La razón por la cual ocurre esto es que, en el segundo caso, claramente Karel tiene el frente libre por, lo que ejecuta la instrucción **move**, pero al dar un paso, resulta que también tiene el frente obstruido, por lo que finalmente también ejecuta la **turnleft**. Observa que esto no pasa en el primer caso, dado que independientemente del resultado de la condicional, sólo se ejecuta una de las partes del **if else**.

5.3 La instrucción while

Existen instrucciones que deben de ejecutar mientras una *condición* sea verdadera. Para ello existe el comando **while** y diferentes estados o condiciones de entorno. Esto se hace con la siguiente sintaxis:

```
while( <condición> [bloque condicional] )
        <instrucción> [{bloque}]
```

Código 5.3.1 Sintaxis de la instrucción while.

Por ejemplo:

```
while( frontIsClear )
        move();

while( leftIsClear )
        {
        turnleft();
        move();
        }
```

Código 5.3.2 Usando la instrucción while.

Al igual que con la instrucción **if**, en el **while** se pueden usar las condiciones antes mencionadas. La versatilidad del **while** es amplia. En el siguiente ejemplo queremos mirar al norte y no sabemos a que dirección estamos mirando. Por lo tanto, la solución es girar hasta mirar al norte. El código quedaría como:

```
while( notFacingNorth )
        turnleft();
```

Código 5.3.3 Utilizando una sola instrucción en el while.

5.4 Bloques condicionales

Los bloques condicionales, son un grupo de condiciones que se piden en una instrucción **if** o **while**. Las condiciones deben ir "ligadas" con un operador lógico "and" u "or". La sintaxis del and y or son respectivamente:

- And: <*Condición 1*> && <*Condición 2*>
- Or: <*Condición 1*> || <*Condición 2*>

5.4 Bloques condicionales

Si enlazamos un par de condiciones (llamémoslas condición 1 y condición 2) a través de un operador lógico, es como haber creado una condición 3. La condición 3 es verdadera dependiente del resultado de cada condición y del operador lógico empleado para unirlas. Así pues, cuando empleamos el operador "and", la condición 3 será verdadera sólo si, las condiciones 1 y 2 son también verdaderas, en caso contrario será falsa. Cuando empleamos el operador "or", la condición 3 es verdadera si al menos una de las dos condiciones 1 y 2 son verdaderas.

Por ejemplo: queremos avanzar mientras haya beepers en el suelo. Por otro lado, debemos evitar chocar con una pared. Por tanto, tenemos dos condiciones, la primera es preguntar si hay beepers y la otra es preguntar si no hay pared. Empleando condicionales sencillas:

```
if ( nextToABeeper )
        {
        if ( frontIsBlocked )
                move();
        }
```

Código 5.4.1 Anidando condicionales.

Como queremos que ambas condiciones sean verdaderas para ejecutar la instrucción **move**, entonces empleamos el operador and (&&) para ligar a ambas condicionales. De este modo, la instrucción **move** sólo se ejecutará si estamos sobre un beeper y tenemos el frente despejado.

```
if ( nextToABeeper && frontIsClear )
        move();
```

Código 5.4.2 Instrucción que se ejecuta sólo si las dos condiciones son verdaderas.

Por el contrario, si quisiéramos detenernos en cuanto encontremos un beeper o topemos con pared, emplearíamos el operador "or" de la siguiente manera.

```
if ( nextToABeeper || frontIsBlocked )
        turnoff();
```

Código 5.4.3 Instrucción que se ejecuta si al menos una de las condiciones es verdadera.

Si hubiéramos deseado hacerlos a través de condicionales sencillas, nuestro código hubiera sido un poco más largo ya que tenemos que repetir instrucciones:

```
if ( nextToABeeper )
        turnoff();
if ( frontIsBlocked )
```

```
        turnoff();
```

Código 5.4.4 Apagando a Karel por dos condiciones diferentes.

En general, un bloque puede tener una cantidad arbitraria de condicionales. De este modo, podemos colocar múltiples restricciones, enlazadas con diferentes operadores, para un mismo código. Imaginemos que deseamos detectar un muro. Es decir, que haya pared hacia enfrente o lados. Por tanto, podemos emplear una triple condicional enlazada con el operador "OR".

Empleando el operador OR:

```
if ( frontIsBlocked || leftIsBlocked || rightIsBlocked )
        { Procesa(); }
```

Código 5.4.5 Apagando a Karel por diferentes condiciones.