Programación Orientada a Objetos en el Micro mundo del Robot Karel.
Libro No 1

Wladimir Rodríguez Gratérol
Doctorado en Ciencias Aplicadas

Hernando Castañeda Marín
Estudiante de doctorado en Ciencias Aplicadas

Universidad de los Andes
Facultad de Ingeniería
Mérida, abril 2006-03-20

INTRODUCCION

El dramaturgo checo Karel Capek publicó su libro "*Amaneceres solo*", que en uno de sus poemas menciona la palabra "Robota", para significar "trabajo forzado".

Richard E. Pattis, en California, inventó un robot y lo bautizó Karel; para él creó un mundo: El mundo de Karel.

El Mundo de Karel, es un mundo imaginario cuadriculado, plano e infinito, habitado por robots.

Antes de explicar las instrucciones primitivas del lenguaje de programación del robot, primero se deben definir los términos técnicos de ejecución que debe realizar el robot: Un robot ejecuta una instrucción realizando una acción o varias acciones asociadas de instrucciones. El robot corre un programa ejecutando una secuencia de instrucciones que le son dadas por el piloto del helicóptero (Programador).

Cada instrucción en secuencia le entrega al robot un mensaje, que le ordena realizar una o varias instrucciones en el programa.

Este módulo esta dividido en cuatro capítulos, en el primer capítulo se describe el Mundo de Karel, se definen las capacidades del robot y se estudian las aplicaciones del robot. En el segundo capítulo se estudian las instrucciones primitivas de Karel (su vocabulario primitivo), como son su posición, sus movimientos, sus giros (izquierda y derecha), el manejo de los pitos y el lenguaje de programación del robot, comienza con una explicación detallada de

las instrucciones primitivas de programación que construyen el vocabulario de cada robot. Usando estas instrucciones, podemos ordenar a cualquier robot moverse a través de su mundo y enseñarle a recoger los pitos.

En el capítulo tercero se extiende el vocabulario de Karel con el uso de nuevas instrucciones de programación y la técnica de refinamiento paso a paso para resolver problemas y a manera de ejemplos se muestran algunos programas completos del robot, donde se discuten las reglas elementales de la sintaxis del lenguaje de programación del robot. En el cuarto capítulo se explica el mecanismo de especificar una nueva clase de robots e igualmente se adicionan nuevas instrucciones al vocabulario del robot. Al final del capítulo se formulan algunos programas sencillos, donde el robot realiza tareas simples de superar obstáculos, recoge y transporta los pitos.

Los Autores

TABLA DE CONTENIDO

Pág.

1. EL MUNDO DE KAREL

1.1 GEOGRAFÍA DEL MUNDO DE KAREL[1]

Inicialmente, Pattis[2] diseñó para Karel un mundo cuadriculado, plano e infinito. Llamó calles a las líneas horizontales (de oeste a este) y avenidas a las verticales (de sur a norte), y quiso que Karel transitara por unas y otras:

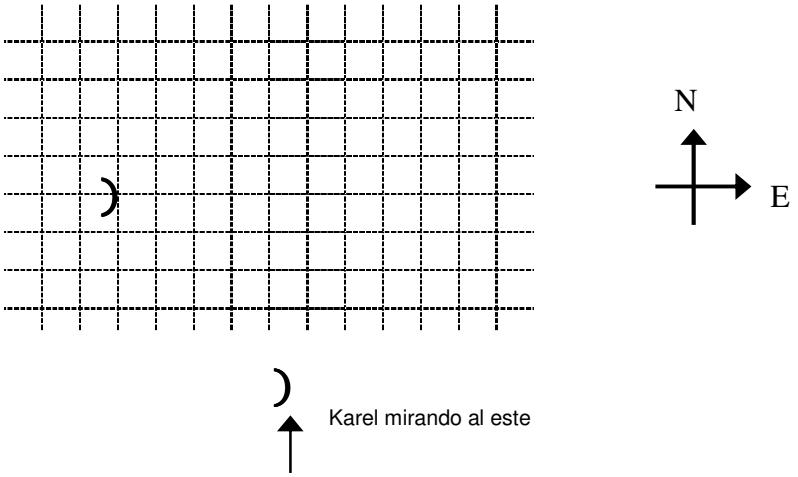


Figura. 1 Diseño del mundo de Karel

Este primer diseño presentó dos problemas: el primero fue que el mundo de Karel resultaba demasiado grande y por lo tanto complejo y el segundo era la ubicación de karel, no se sabía donde estaba exactamente o en que otro mundo se encontraba, ya que, todos los sitios del mundo eran iguales para él robot. Para resolver el primer problema, Pattis[3] decidió limitar el mundo por el oeste y por el sur, así, evitó que Karel saliera inadvertidamente de su mundo,

[1] ***CAPEK, Karel*** *Amaneceres, solo. Checoslovaquia. 1862*
[2] ***PATTIS Richard E.*** *El mundo de karel. California. 1890*
[3] ***PATTIS Richard E.*** *Op. cit.*

colocó dos muros infinitos construidos en Neutrino (materia impenetrable que ningún robot puede atravesar, romper o mover) en los bordes del mundo.

Los muros sirvieron también como referencia para identificar calles y avenidas, numerándose en orden ascendente (a partir de 1) desde el cruce de los muros; una esquina (cruce de calle y avenida) quedaba entonces identificada por el número de su calle y su avenida. Por ejemplo, (3,17) identifica el cruce de la calle 3 con avenida 17. La primera esquina del mundo (1,1), en el extremo sur-oeste, que correspondía al cruce de la calle 1 con la avenida 1 recibió el nombre de origen.

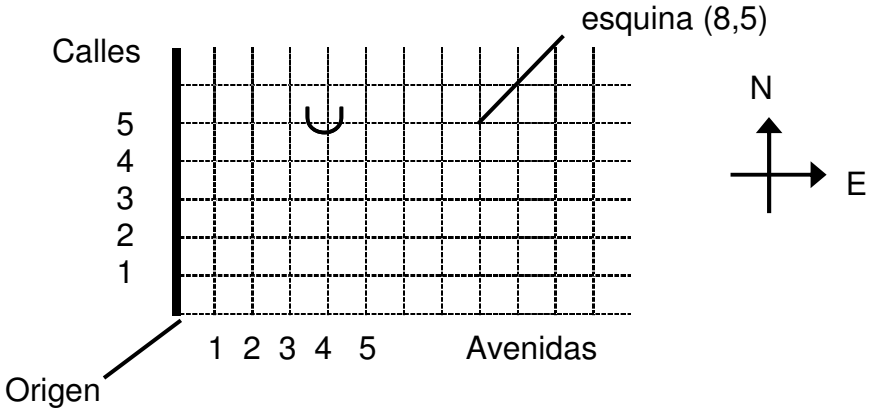


Figura. 2 Estructura del mundo de Karel

Karel puede localizarse únicamente en las esquinas del mundo, orientado hacia uno de los cuatro puntos cardinales (norte, sur, este, oeste).

En el figura 2 Karel se encuentra en la esquina (4,5): calle 5 con avenida 4, mirando al sur.

El segundo problema fue resuelto limitando su mundo[4]. Para hacer más interesante su mundo, Pattis diseñó dos nuevos elementos: los segmentos de los muros y los pitos[5].

Los segmentos de los muros son de longitud finita, horizontal o vertical y van por la mitad de las manzanas. Para que Karel no pueda atravesarlos ni moverlos están construidos en Neutrino. En la figura 3 se muestra la

[4] ***PATTIS Richard E.*** *Op cit.*
[5] ***PATTIS Richard E.*** *Op. cit.*

representación gráfica de los muros del mundo de Karel, los que se denotan con segmentos de recta:

Figura. 3 Barreras horizontal y vertical (muros)

Los pitos son esferas de plástico que emiten un leve sonido. Al igual que Karel, pueden estar localizados solamente en las esquinas; en una esquina se puede encontrar uno o varios pitos. Karel puede recoger, transportar y colocar pitos según las instrucciones del programa.

En la figura 4 se muestra la ubicación de los pitos en las esquinas.

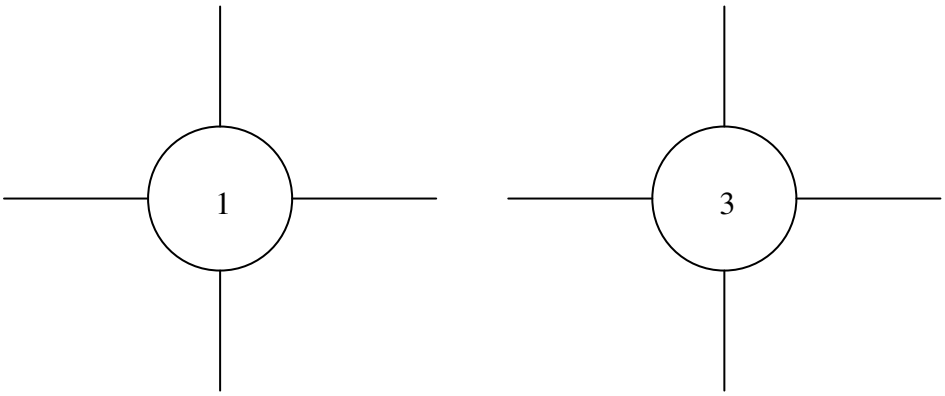
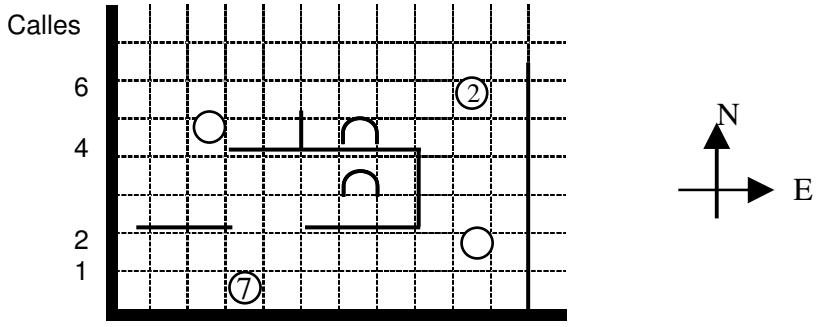


Figura. 4 Número de pitos en una esquina

En la figura 5 se muestra una situación del mundo de Karel, en la cual el robot se encuentra en la esquina (5,7) mirando al norte, hay 12 pitos regados por el mundo y 6 muros que bloquean el movimiento de Karel.



1 2 3 4 5 6 7 Avenidas

Figura. 5 Elementos del mundo de Karel

1.2 CAPACIDAD DE KAREL

Pattis, debía dotar a Karel de capacidades que le permitiera aprovechar su mundo y desenvolverse en él. Estas capacidades se clasificaron en: propiedades de movimiento, de percepción y de manipulación[6].

1.2.1 Capacidad de movimiento: Karel puede moverse una cuadra hacia adelante en la dirección en la cual está orientado y puede rotar en su posición.

1.2.2 Capacidad de percepción: Karel tiene sentidos rudimentarios de vista, oído, tacto y orientación: Posee tres cámaras de televisión, dirigidas hacia el frente, hacia su derecha y hacia su izquierda. Con ellas puede ver un muro a media cuadra de distancia al frente, a la derecha y a la izquierda. Tiene además un oído que le permite percibir pitos localizados en su misma esquina (Karel no puede saber cuantos pitos hay en la esquina, solamente percibe que hay pitos). Consultando su brújula interna el robot puede saber la dirección en la que está mirando (norte, sur, este u oeste).

1.2.3 Capacidad de manipulación: Finalmente, Karel posee un brazo mecánico con el que puede recoger y poner pitos en una bolsa a prueba de ruido para transportarlos, puede determinar si lleva algún pito en su bolsa, tocándola con su brazo mecánico, es capaz de recibir, memorizar y seguir una serie de instrucciones dadas en un lenguaje de programación que él siempre entienda.

1.3 ESPECIFICACIÓN DE PROBLEMAS EN EL MUNDO DE KAREL

Las instrucciones que Karel debe realizar en su mundo, están definidas por las especificaciones de un problema determinado. Por ejemplo, para salir de un

---

[6] ***PATTIS Richard E.*** *Op. cit.*

laberinto de muros, recoger un sembrado de mazorcas (pitos), recoger y colocar pitos en determinada direcciones, forman un estado en el mundo de Karel, que es una descripción exacta y completa de todos los componentes del mundo en un momento dado.

Un estado lo podemos ver, entonces, como una "aerofotografía[7]" del mundo de Karel que muestre los muros del mundo, la localización y orientación de Karel y el estado de los pitos (tanto los que están en las esquinas del mundo como los que están en la bolsa de Karel). En la figura 6 se muestra un estado posible del mundo de Karel.

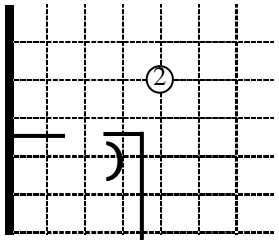


Figura. 6 Estado posible en un mundo de Karel

En el mundo de Karel un problema se especifica dando una condición inicial que debe cumplirse en su mundo y una condición final que debe cumplirse para resolver el problema.

Resolver un problema en este contexto es encontrar una secuencia de pasos dentro de una instrucción que le indique al robot la manera de "transformar" cualquier estado que cumpla la condición inicial en un correspondiente estado que cumpla la condición final. Los muros del mundo, no pueden variar del estado inicial al final, ya que Karel no tiene la capacidad de moverlos. Los pitos pueden sufrir modificaciones en su localización, pero el número total de pitos en el mundo no pueden variar en una tarea, pues Karel no los puede crear ni destruir.

[7] ***PATTIS Richard E.*** *Op. cit.*

1.4 APLICACIONES

1.4.1 Ejemplos

Problema 1. Karel debe recoger los 5 pitos que se encuentran en la calle 1 del mundo:

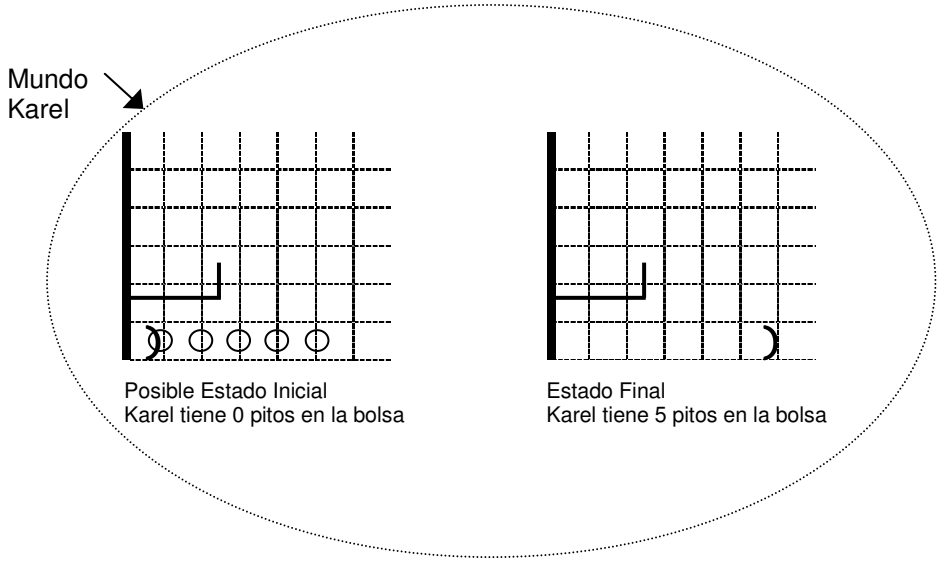


Figura. 7 Estado inicial y final del problema 1

Problema 2. Karel se encuentra en alguna esquina del mundo, con 8 o más pitos en su bolsa y queremos que deje pitos en las esquinas a su alrededor:

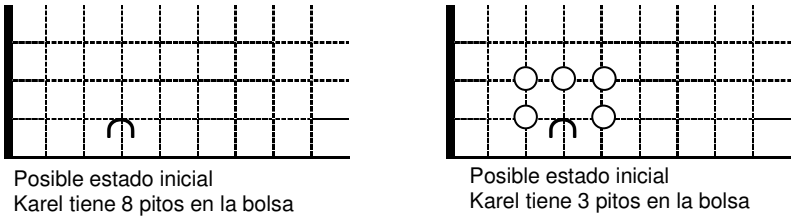


Figura. 8 Estado inicial y final del problema 2

Problema 3. Karel se encuentra en el origen mirando al este, en el mundo hay montañas escalonadas que parten del límite horizontal del mundo y alcanzan una altura de 3 calles, las dos montañas están separadas por una calle. Karel

debe recoger los pitos que se encuentran en cada una de las esquinas que rodean las montañas y regresar al origen:

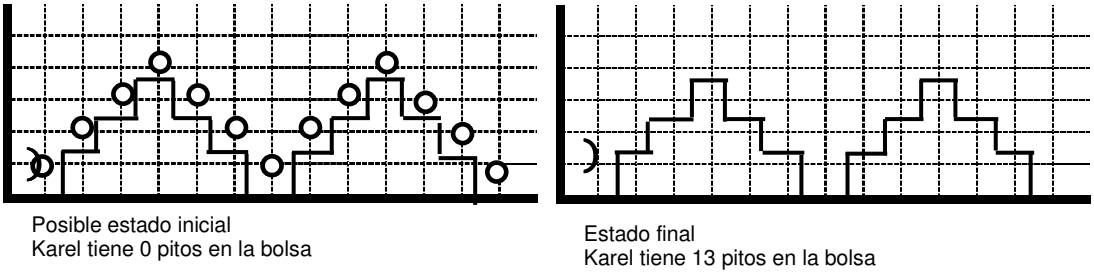


Figura. 9 Estado inicial y final del problema 3

Problema 4. Karel debe recoger los pitos que se encuentran en su mundo. Parte del estado inicial y debe llegar al estado final como se muestra en la figura 10.

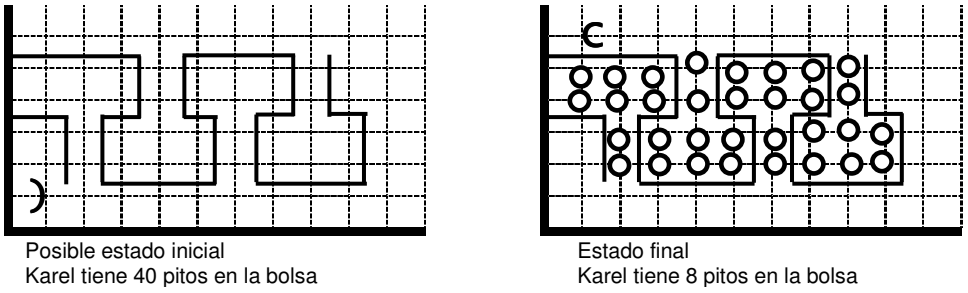


Figura. 10 Estado inicial y final del problema 4

Problema 5. Karel se encuentra en alguna calle de la avenida uno. Al frente de él hay un muro de tres calles de altura y dos calles más adelante hay otro muro igual. Sobre la avenida uno, a la izquierda del segundo muro, hay un pito que Karel debe recoger. Estado inicial: $j > 0$, para indicar el número de la calle donde Karel se encuentra.

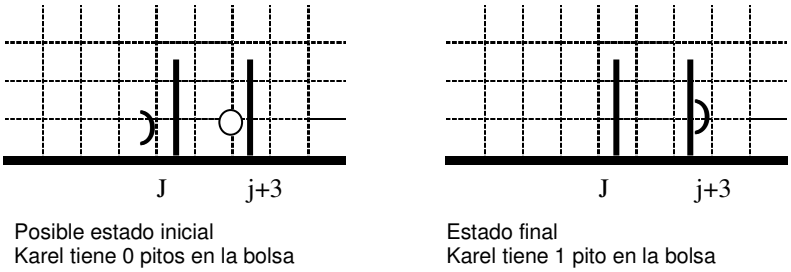


Figura. 11 Estado inicial y final del problema 5

1.5 EJERCICIOS PROPUESTOS

Ejercicio. 1 Identifique los errores de definición de los elementos que se presentan en el siguiente mundo de Karel.

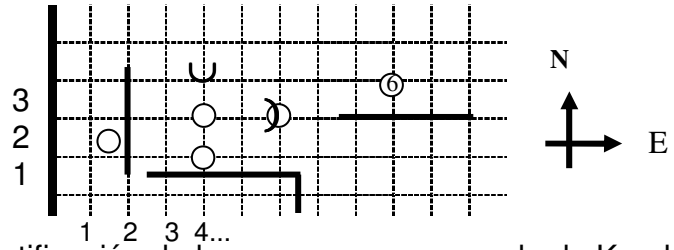


Figura. 12 Identificación de los errores en un mundo de Karel para el ejercicio

Ejercicio. 2 Realice las siguientes modificaciones en el mundo de Karel:

a) Coloque tres pitos en el origen
b) Coloque un pito en la esquina (4,2)
c) Ponga un pito en la calle 4 entre avenidas 1 y 2
d) Ponga a Karel en la esquina (6,1) mirando hacia el sur-este
e) Coloque a Karel en la esquina (2,2) mirando al sur

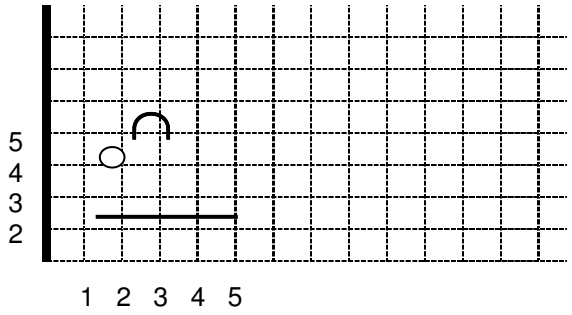
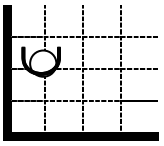
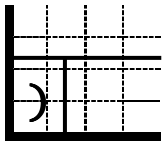
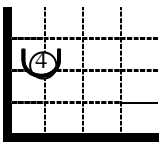


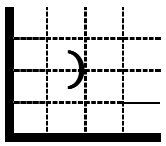
Figura. 13 Modificaciones en un mundo de Karel para el ejercicio 2

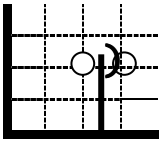
Ejercicio. 3 ¿Qué percibe Karel en cada una de los estados siguientes?

A. 

b 

C. 

d. 

e. 

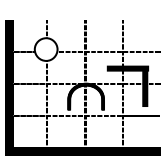
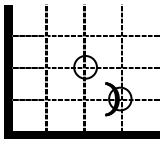
f. 

Figura. 14 Percepción de estados en un mundo de Karel para el ejercicio 3

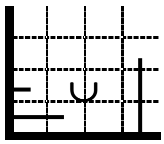
Ejercicio. 4 ¿Cuáles de los siguientes casos especifican correctamente un problema?

a)



Karel tiene 0 pitos en la bolsa

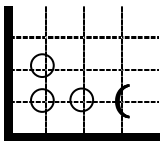
b)



Karel tiene 1 pito en la bolsa

Figura. 15 Especificación de estados en un mundo de Karel para los Ejercicios 4 a y 4 b

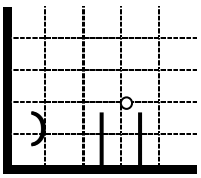
c) Karel debe determinar el siguiente estado:



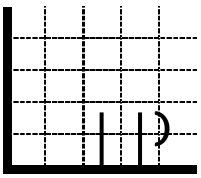
Karel tiene 0 pitos en la bolsa

Figura. 16 Determinación de un estado en un mundo de Karel para el ejercicio 4 c

d) Se tiene el siguiente estado inicial con su correspondiente estado final:



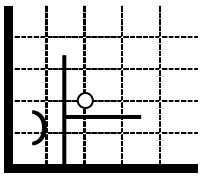
Karel tiene 0 pitos en la bolsa



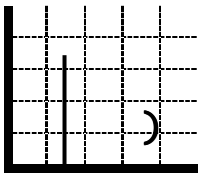
Karel tiene 1 pito en la bolsa

Figura. 17 Estado inicial y final en un mundo de Karel para el ejercicio 4 d

e) Se tiene el siguiente estado inicial con su correspondiente estado final:



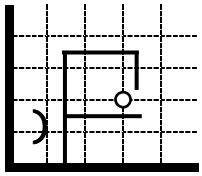
Karel tiene 0 pitos en la bolsa



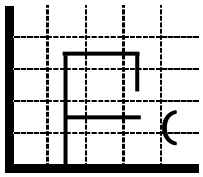
Karel tiene 1 pito en la bolsa

Figura. 18 Estado inicial y final en un mundo de Karel para el ejercicio 4 e

f) Se tiene el siguiente estado inicial con su correspondiente estado final:



Karel tiene 0 pitos en la bolsa



Karel tiene 0 pitos en la bolsa

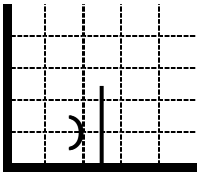
Figura. 19 Estado inicial y final en un mundo de Karel para el ejercicio 4 f

g) Karel debe saltar un muro y tomar el pito que se encuentra detrás.

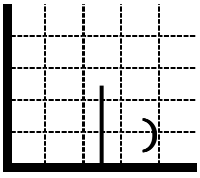
h) Karel debe partir del origen, y terminar en la esquina (3,1), mirando al norte.

Ejercicio. 5. ¿Cuáles de los siguientes problemas están bien especificados y debería poderlos resolver Karel? En la figura 20 se ilustran los estados iniciales y finales.

a)



Karel tiene 1 pito en la bolsa



Karel tiene 1 pito en la bolsa

Figura. 20 Estado inicial y final en un mundo de Karel para el ejercicio 5 a

b)

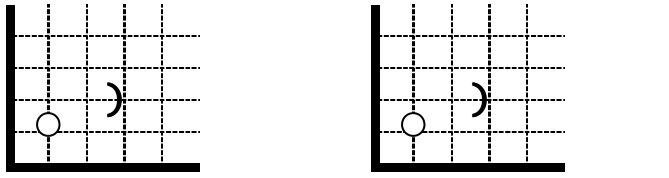


Figura. 21 Estado inicial y final en un mundo de Karel para el ejercicio 5 b

c)

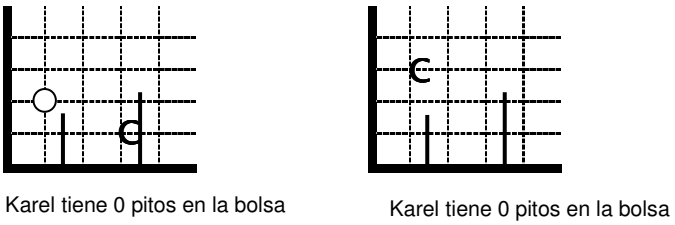


Figura. 22 Estado inicial y final en un mundo de Karel para el ejercicio 5 c

d)

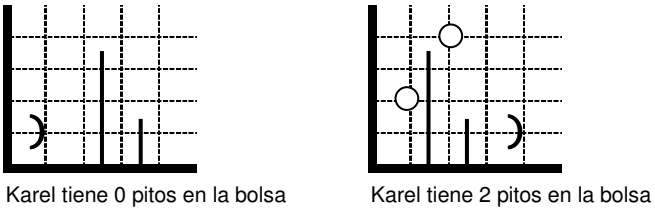


Figura. 23 Estado inicial y final en un mundo de Karel para el ejercicio 5 d

e)

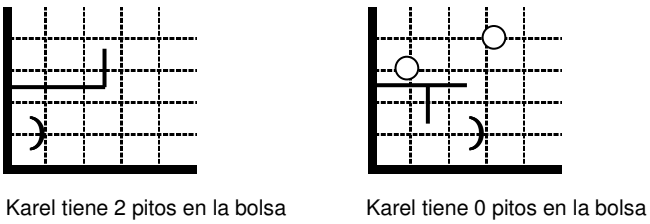


Figura. 24 Estado inicial y final en un mundo de Karel para el ejercicio 5 d

f)

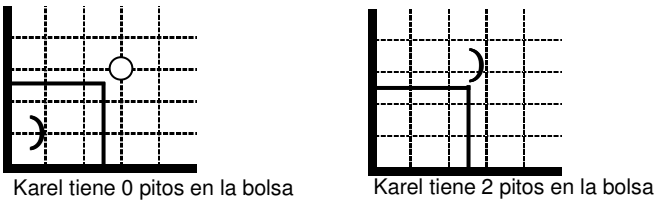


Figura. 25 Estado inicial y final en un mundo de Karel para el ejercicio 5 f

Ejercicio. 6 Defina el estado inicial y el estado final de cada uno de los siguientes enunciados:

a) Karel debe recoger todos los pitos que se encuentran en su mundo.

b) Karel debe recoger todos los pitos de la calle 1.

c) Karel se encuentra en el origen, no existen muros y hay pitos en las esquinas (3,4), (2,3), los cuales debe recoger el robot.

d) Karel se encuentra en la esquina (1,5) y debe recoger todos los pitos que los rodean.

e) Karel se encuentra en la siguiente situación, y debe llenar el cuarto con pitos:

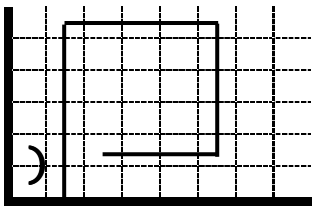


Figura. 26 Estado especificado para Karel para el ejercicio 6 e

f) En la figura 27 se ilustra el mundo de Karel, quién tiene que recoger un pito en las esquinas que tengan más de un pito y regresar al origen:

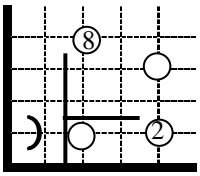


Figura. 27 Estado especificado para un mundo de Karel para el ejercicio 6 f

g) En la figura 28 se define el estado del mundo de karel, quién debe recoger todos los pitos y formar tres torres del mismo tamaño en las avenidas rodeadas por muros:

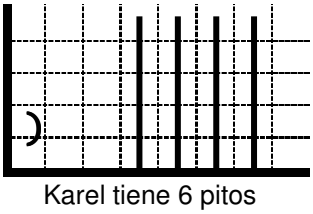


Figura. 28 Estado especificado en un mundo de Karel para el ejercicio 6 g

Ejercicio. 7 Para cada uno de los siguientes problemas dibuje otro posible estado inicial con su correspondiente estado final:

a) Karel trabaja en una librería y debe colocar todos los libros en los cuatro anaqueles de la biblioteca, en cada anaquel caben hasta dos libros; Karel ya colocó los libros (pitos) frente a cada anaquel, Karel debe guardarlos. A continuación se muestra un posible estado inicial y su correspondiente estado final:

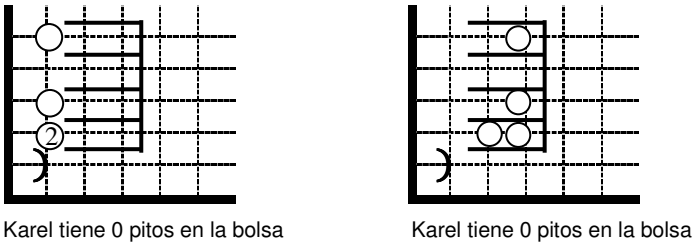


Figura. 29 Estado inicial y final definidos para Karel para el ejercicio 7 a

b) Karel se está entrenando para una carrera de obstáculos en una pista de 12 millas de largo, en la que hay (entre las esquinas (1,1) y (1,12)) obstáculos de 0, 1, 2 ó 3 metros de altura. Karel debe saltar los obstáculos colocando detrás de cada una, a la altura de la calle 1 tantos pitos como metros midan el obstáculo y regresar al origen. Suponga que Karel parte con suficientes pitos. A continuación se da un ejemplo de un posible estado inicial y con su correspondiente estado final.

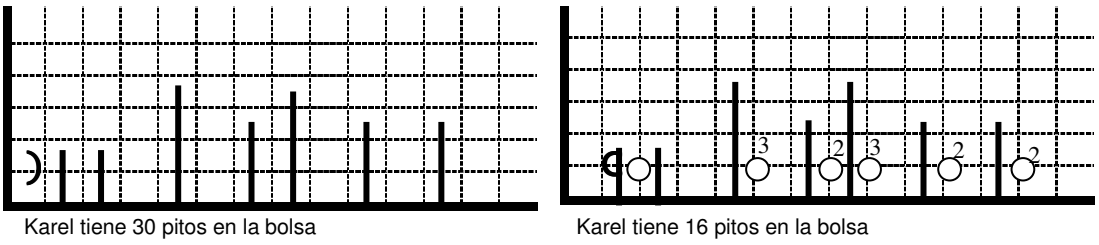


Figura. 30 Estado inicial y final definidos para Karel para el ejercicio 7 b

c) Karel se encuentra en el origen mirando al Este. En las primeras 7 esquinas del mundo sobre la calle 1 (de la avenida 1 a la 7) hay una serie de pitos (0,1, 2 ó 3 en cada esquina). Haga un programa para que Karel “mueva” este

“patrón” de pitos 3 calles más arriba (en la calle 4). Karel debe terminar en el origen, mirando al este. A continuación se muestra un posible estado inicial y su correspondiente estado final.

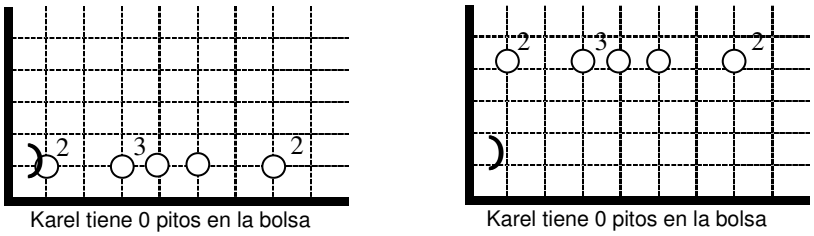


Figura 31. Estado inicial y final definidos para Karel para el ejercicio 7 c

d) Karel tiene un criadero de conejos formado por cuatro corrales. Todos los días Karel saca los conejos de su corral y los pone frente a la entrada para que coman. En cada corral puede haber hasta 3 conejos. Karel debe sacar los conejos de cada corral y ponerlos frente a la entrada (una esquina al oeste). Karel parte del origen mirando al Este y termina igual. A continuación se muestra un posible estado inicial y su correspondiente estado final

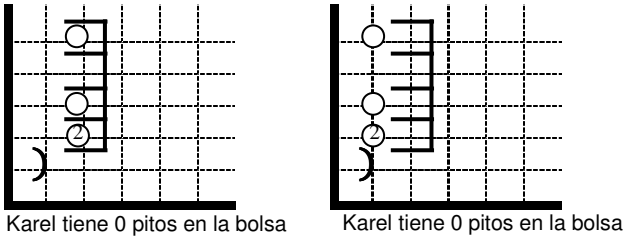


Figura. 32 Estado inicial y final definidos para Karel para el ejercicio 7 d

2 INTRUCCIONES PRIMITIVAS DE KAREL

Para que Karel realice una tarea en su mundo, debe dársele un conjunto detallado de instrucciones que le indiquen cómo realizarla. Karel es capaz de recibir, memorizar y ejecutar ese conjunto de instrucciones o programas.

En este capítulo se explican las cinco instrucciones primitivas (vocabulario básico) del lenguaje de Karel, con las que se ordena al robot moverse a través de su mundo y manejar los elementos que allí se encuentran.

2.1. EL VOCABULARIO PRIMITIVO DE KAREL

Las instrucciones primitivas son órdenes que le permite a Karel resolver tareas.

2.1.1 Primitiva cambio de posición: Karel tiene dos instrucciones para cambiar de posición: "**move** ()" cambia su localización y "**turnLeft ()**" su orientación.

Mensaje **move ()**: si el frente está despejado (no hay un muro a media cuadra), se mueve una cuadra en esa dirección. Si hay un muro al frente, a media cuadra, se apaga después de informar el error.
En la figura 33 se observa como karel ejecuta esta instrucción en dos situaciones posibles:

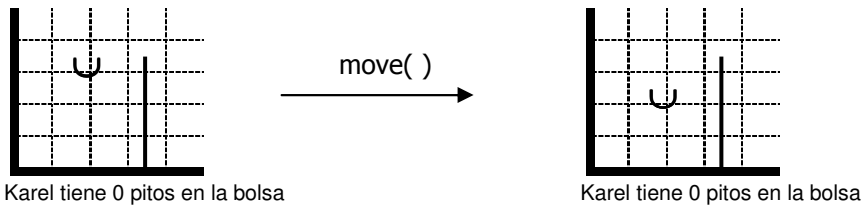


Figura 33 Estado inicial y final de Karel después de ejecutar una instrucción

Puesto que al frente (a media cuadra de distancia) no hay muro, Karel ve el camino despejado y avanza una cuadra en la dirección en que está mirando, en este caso, hacia el sur.

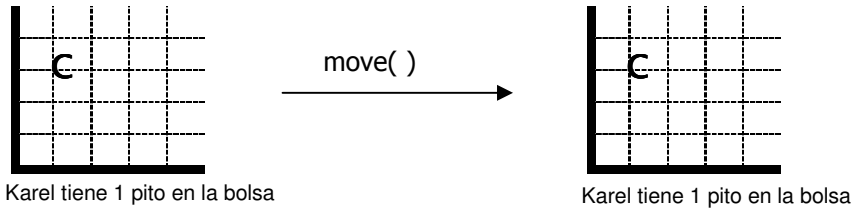


Figura 34 Estado inicial y final de Karel después de ejecutar una instrucción

Con su cámara delantera Karel ve un muro a media cuadra de distancia y como medida de seguridad no ejecuta el movimiento que lo haría estrellarse contra éste, sino que, se apaga después de informar el error. Esta acción de apagarse debido a un error del programa la llamaremos apagarse por error (error shutOff)

Mensaje turnLeft () (Giro a la izquierda): Karel ejecuta esta instrucción rotando 90 grados hacia su izquierda. Los otros elementos del mundo (muros y pitos) y la localización de Karel no se modifican. Lo único que cambia es su orientación. Karel nunca se apaga por error al ejecutar la instrucción girar hacia la izquierda (siempre puede realizar el giro). En la figura 35 se muestran cuatro situaciones posibles en las que Karel puede ejecutar esta instrucción y sus respectivos resultados:

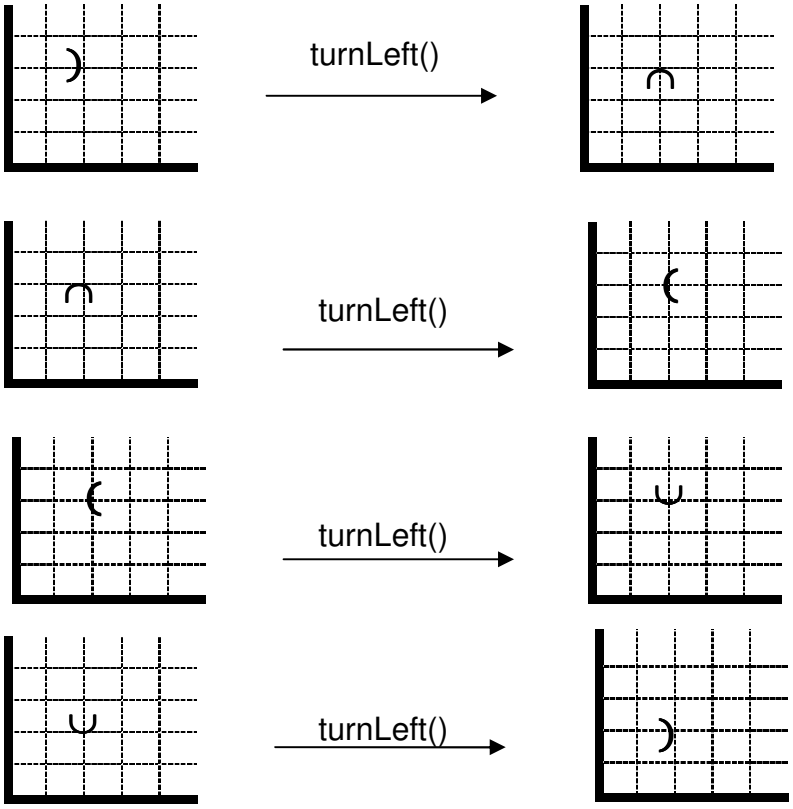


Figura. 35 Estados posibles de karel después de ejecutar la instrucción turnLeft

2.1.2 Manejo de Pitos: Karel puede recoger los pitos de su esquina, llevarlos en su bolsa y ponerlos en otra esquina. Para recoger y guardar un pito en la bolsa tenemos la instrucción "**pickBeeper** ()" y para tomar un pito de la bolsa y ponerlo en la esquina usamos "**putBeeper** ()"

Mensaje pickBeeper (): si hay pitos en su esquina, Karel toma uno y lo guarda en su bolsa. Si no hay pitos en su esquina, el robot se apaga por error. Estos dos casos se ilustran en la figura 36:

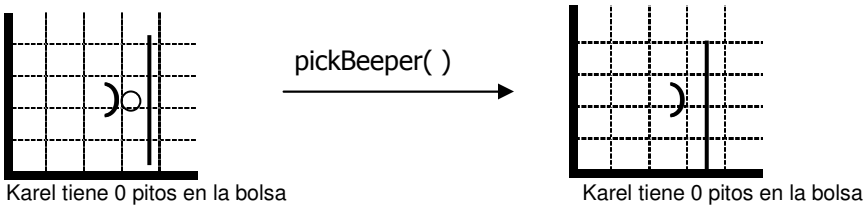


Figura. 36 Estados posibles después de ejecutar la instrucción pickBeeper

Karel oye el sonido de los pitos en su esquina, recoge uno de ellos y lo guarda en la bolsa. En la figura 37 se observa que el número total de pitos sigue siendo uno, pues ahora hay cero pitos en la esquina y uno en la bolsa de Karel. Como no hay pitos en la esquina de Karel, éste se apaga por error sin realizar otra acción.

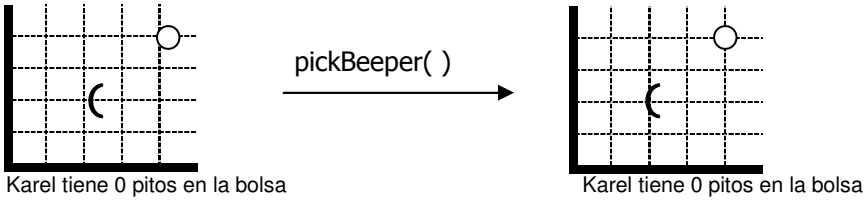


Figura. 37 Estados posibles de Karel después de ejecutar la instrucción pickBeeper

Mensaje putBeeper (): si tiene pitos en la bolsa, saca uno y lo pone en su esquina y si tienen la bolsa vacía se apaga por error.

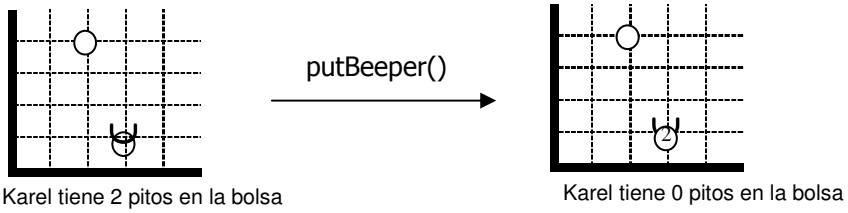


Figura. 38 Estados posibles de Karel después de ejecutar la instrucción putBeeper

Karel realiza el putBeeper () revisando en su bolsa si tiene pitos, coge uno y lo coloca en la esquina en la que está; en la situación final quedan dos pitos en la esquina y uno en la bolsa. Como no tiene pitos en su bolsa, Karel se apaga por error.

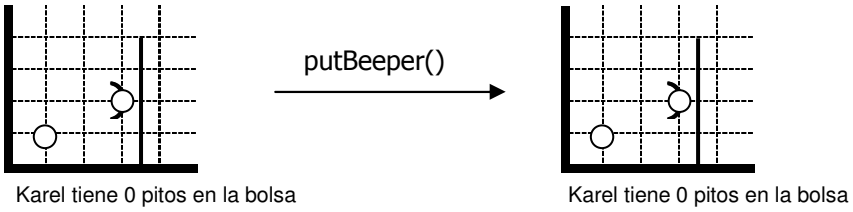


Figura. 39 Estados posibles de Karel después de ejecutar la instrucción putBeeper

2.1.3 Terminación de instrucciones

Mensaje turnOff (): Karel se apaga normalmente, sin errores. Esta instrucción debe ser siempre la última que ejecute, cuando ya haya llegado a la situación final del problema a resolver, se debe colocar dentro de un programa como la última instrucción.

2.2 SOLUCIÓN DE PROBLEMAS

Las instrucciones primitivas resuelven problemas. Para resolver esta clase de problema basta con saber qué instrucción le permitirá a Karel transformar el estado inicial en el estado final.

Problema 1. En las figuras 40 y 41. ¿Qué primitivas utilizaría karel? Para resolver los problemas

Primer caso:

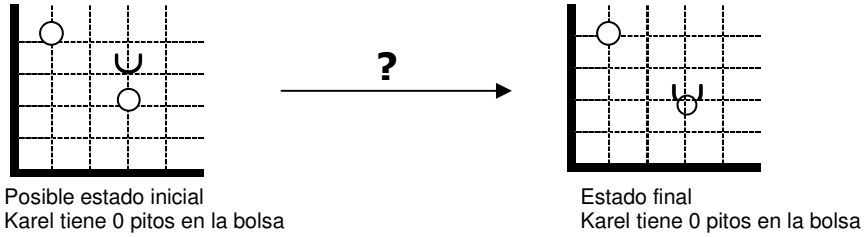


Figura. 40 Tipo de instrucción que le permite a Karel transformar el estado inicial en estado final

Segundo caso:

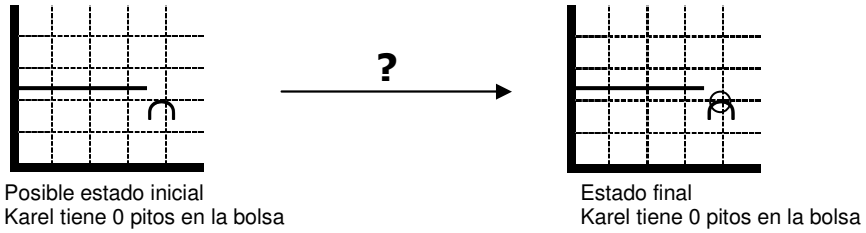


Figura. 41 Tipo de instrucción que le permite a Karel transformar el estado inicial en estado final

Soluciones:

Primer caso: En el primer caso, lo único que cambia es la posición de Karel. Sólo se necesita una instrucción move ( ), figura 42.

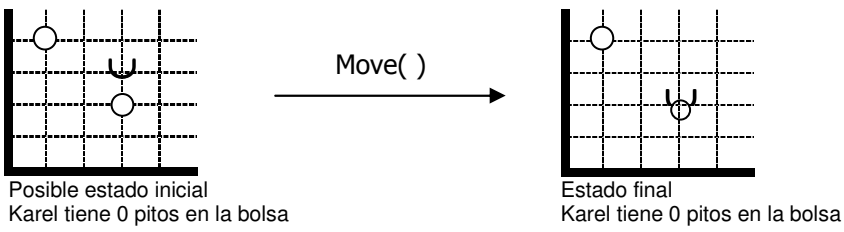


Figura. 42 Uso de la instrucción move para la solución del primer caso

Segundo caso:

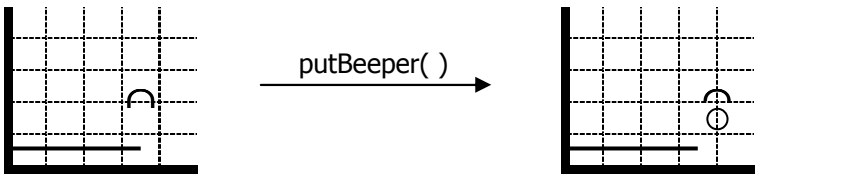


Figura. 43 Uso de la instrucción putBeeper para la solución del segundo caso

Cuando existen tareas más complejas que karel no puede resolver con una sola instrucción primitiva, se debe llevar a cabo el siguiente proceso:

1. Identificar uno o más estados intermedios para partir el problema en problemas más simples (subproblemas).

2. Resolver cada uno de los subproblemas simples. Para los complicados es necesario aplicar de nuevo todo el proceso.

3. Unir las soluciones de los subproblemas para conformar la solución global del problema. En el lenguaje de Karel se utiliza el signo **';'** (punto y coma) para componer (unir) las soluciones de dos subproblemas.

Problema 2: En la figura 44 se le ordena a karel resolver la situación presentada:

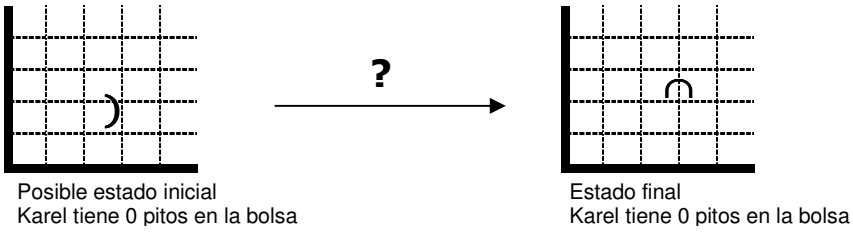


Figura. 44 Estados posibles para el problema 2

Solución:

Se deben identificar los estados intermedios del problema para obtener así dos subproblemas:

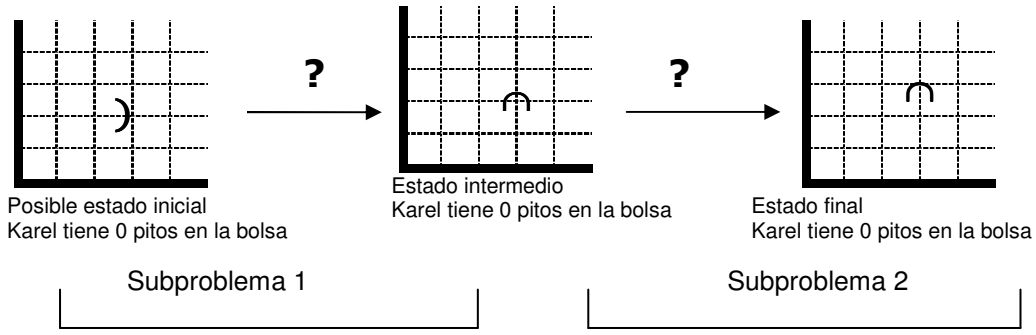


Figura 45. Estados Intermedios para el problema 2

Ahora tenemos dos subproblemas simples que se pueden resolver directamente utilizando instrucciones primitivas. Finalmente se deben unir las soluciones mediante un ';'(punto y coma), obteniendo un programa: El cual representa la solución del problema planteado.

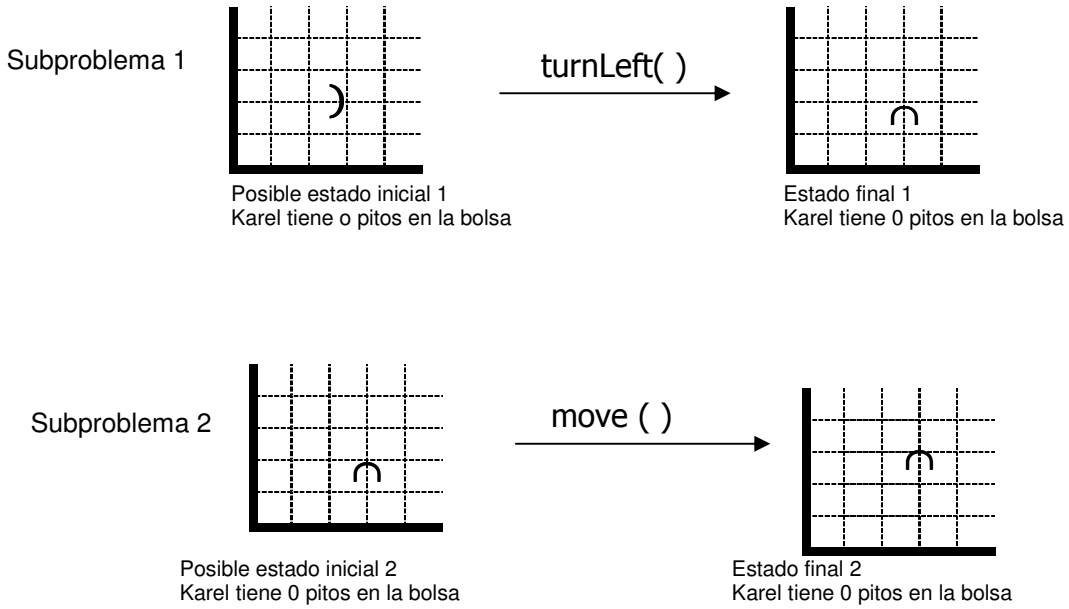


Figura 46 Estados inicial y final de los subproblemas 1 y 2 del problema 2

En algunos casos es necesario identificar varios estados intermedios tal como se ilustra en la figura 47:

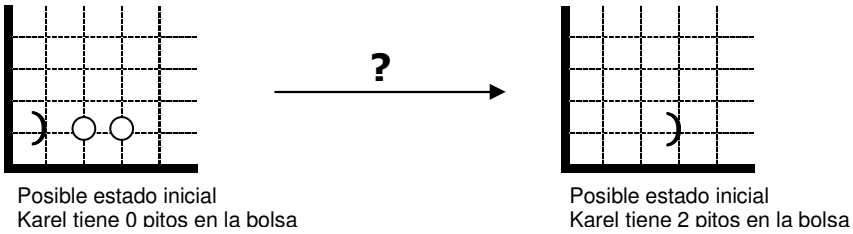


Figura. 47 Posibles estados iniciales y finales

Solución:

Se divide el problema en cuatro subproblemas y se resuelve cada uno independientemente, como se observa en la figura 48.

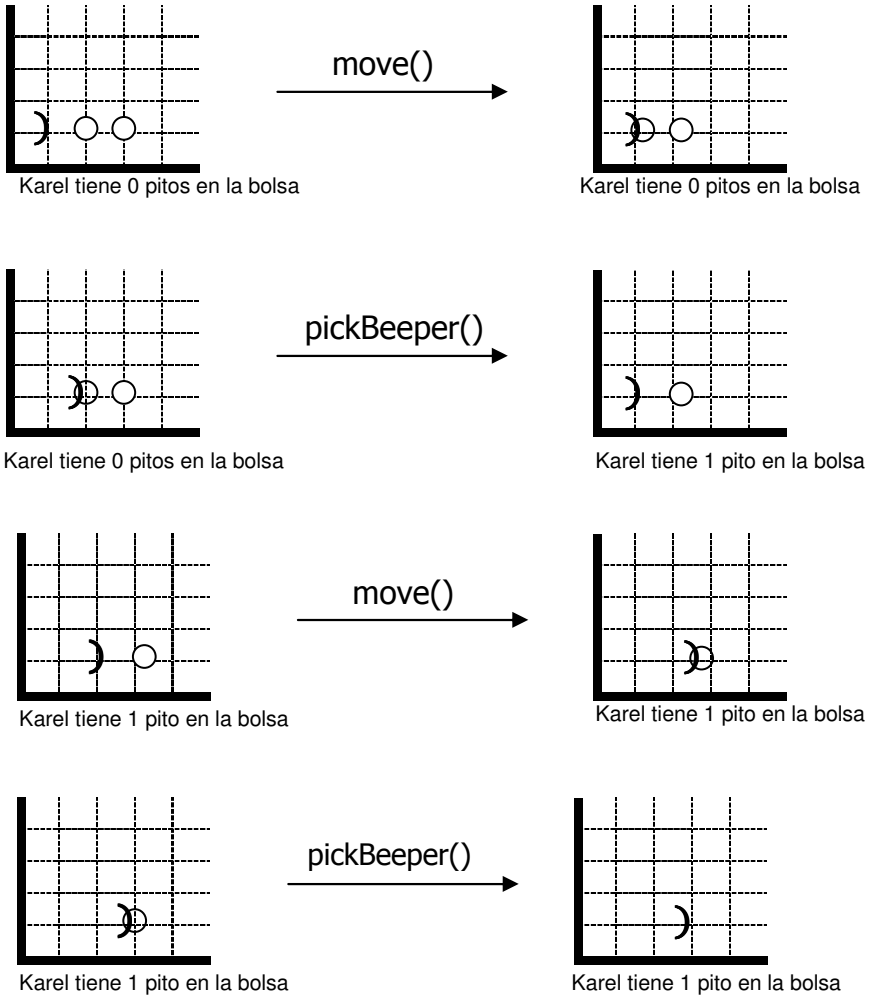


Figura. 48 Posibles estados iniciales y finales

2.3 ESTRUCTURA DE UN PROGRAMA COMPLETO

Hasta el momento se han explicado las instrucciones primitivas del lenguaje de Karel y su uso para resolver problemas, A continuación se le define la forma de comunicarle a Karel las instrucciones que conforman la solución a un problema, de manera que las entienda y pueda ejecutar el programa.

El lenguaje de Karel emplea el **';'** (punto y coma) y un conjunto de palabras reservadas que sirven para estructurar el programa, convencionalmente estas palabras se escriben en mayúsculas y las instrucciones en minúsculas.

2.3.1 Codificación: La codificación de los programas consiste en asignar la sintaxis correcta a las instrucciones para que karel obedezca las órdenes.

Ejemplo de la codificación de un programa:

```
    task
{
    ur_robot karel(2,3,East,0);
    karel.turnLeft();
    karel.move();
    karel.turnOff();
}
```

```
task
{
    ur_robot karel(1,1,East,0);
    karel.move();
    karel.pickBeeper();
    karel.move();
    karel.pickBeeper();
    karel.turnOff()
   }
```

2.3.2 Ejecución de un programa: Para que Karel ejecute un programa son necesarios los siguientes pasos:

Prenderlo: Karel queda en modo lectura, listo para "oír" el programa mediante el mensaje ur_robot karel(n,m,o,k) donde n es el número de la calle, m es el número de la avenida, o es la orientación del robot y k el número de pitos en la bolsa.

Bloque del programa {…}: Encierra todo el programa que le vamos a dar a Karel.

Dentro del bloque {…}: Colocamos, una tras otra, todas las instrucciones del programa, incluyendo al final la instrucción turnOff() para que Karel se apague al terminar.

Pares de símbolos { }: Limitan el comienzo y final de un grupo de instrucciones y se llaman delimitadores.

2.4 ERRORES DE PROGRAMACIÓN

Los errores de programación se clasifican en cuatro categorías:

2.4.1 Errores Léxicos: Ocurren cuando Karel encuentra una palabra que no está en su vocabulario.

2.4.2 Sintácticos: tienen que ver con la "gramática" y la "puntuación" del programa, es decir, con las palabras reservadas y la localización del ';' (punto y coma). Karel detecta estos dos tipos de errores en el modo de lectura; cuando encuentra un error lo notifica y se apaga.

2.4.3 De ejecución: Si Karel entiende el programa (no hay errores léxicos ni sintácticos) se prende el botón de ejecución y comienza la ejecución del programa; en este modo pueden presentarse errores cuando Karel no puede realizar una primitiva correctamente y se apaga por error (move() con el frente bloqueado, sin pickBeeper() cuando no hay pitos en la esquina, putBeeper() sin pitos en la bolsa). Estos son los errores de ejecución.

2.4.4 De intención: Los errores de intención, que son los más difíciles de identificar, ocurren cuando Karel ejecuta correctamente el programa, pero éste no resuelve la tarea especificada (partiendo del estado inicial dado, no se llega al estado final deseado).

2.5 CASO PRÁCTICO

A continuación se especifica una tarea en términos de un estado inicial y su estado final y se presenta un programa escrito para solucionarlo.

Revise el programa, identifique y corrija los errores léxicos y sintácticos que encuentre. Indique los errores de ejecución y modifique el programa para que resuelva la tarea dada (hemos enumerado las líneas del programa únicamente para facilitar el ejercicio).

Problema Caso Práctico: Karel se encuentra en el origen mirando al Este y debe recoger el pito que se encuentra en el origen, el que se encuentra dos calles adelante y regresar al origen: Véase figura 49

Figura 49 Estado inicial y final del problema del caso práctico

Solución:

```
task;
{
3    ur_robot karel (1, 1, East, 0);
4     karel.pickBeeper ();
5     karel.move (); karel.pickBeeper ();
6     karel.move (); karel.pickBeeper ()
7     karel.turnLeft ();
8     karel.move ();
9     move ();
10   }
```

Errores:

- **Léxicos**

  línea 5 karel.pickBeeper();

  línea 7 karel.turnLeft();

- **Sintácticos:**

  línea 1 **;**

  línea 6 falta ;

  línea 8 **; ;**

línea 9 Falta referencia objeto karel

- **De ejecución.**

línea 4 al ejecutar karel.pickBeeper();

línea 10 falta antes de } karel.turnOff()

2.6 EJERCICIOS PROPUESTOS

Ejercicio. 1 ¿Cómo ejecuta Karel la instrucción move en los siguientes estados (dibuje el estado final)?

a)

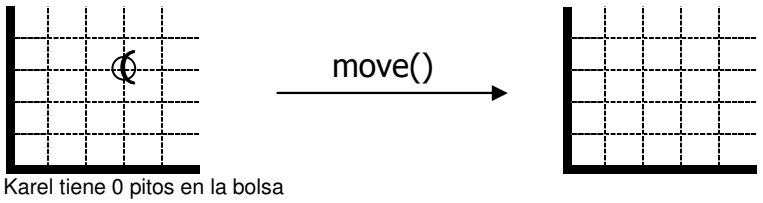


Figura. 50 Posibles estados inicial y final de la instrucción move()

b)

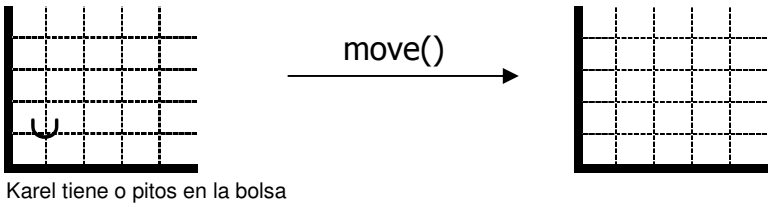


Figura. 51 Posibles estados inicial y final de la instrucción move()

c)

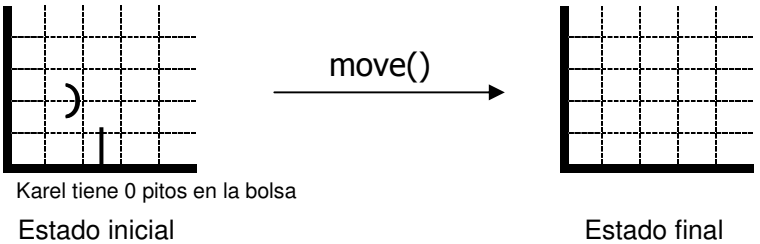


Figura. 52 Posibles estados inicial y final de la instrucción move()

d)

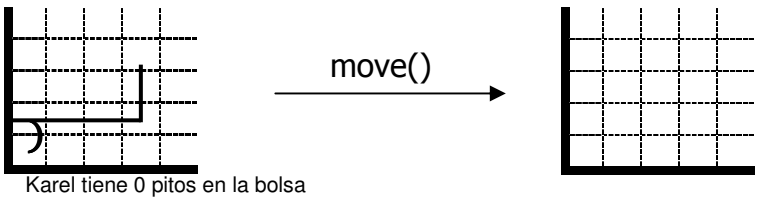


Figura. 53 Posibles estados inicial y final de la instrucción move()

Ejercicio. 2 ¿Cómo quedaría el mundo de Karel después de realizar un turnLeft()?

a)

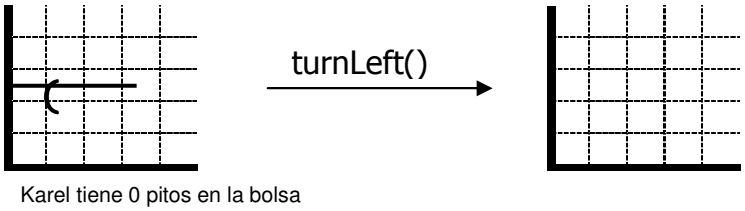


Figura. 54 Posibles estados inicial y final de la instrucción turnLeft()

b)

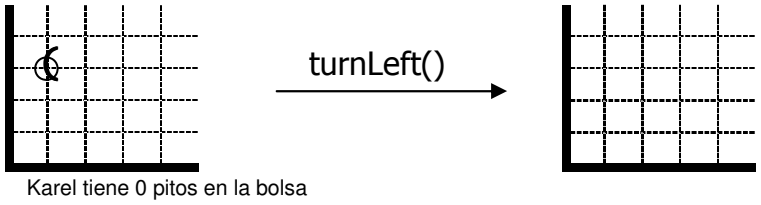


Figura. 55 Posibles estados inicial y final de la instrucción turnLeft()

c)

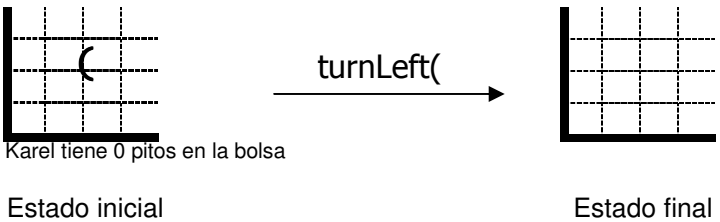


Figura. 56 Posibles estados inicial y final de la instrucción turnLeft()

d)

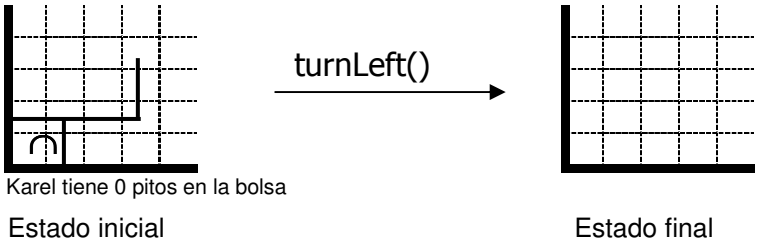


Figura. 57 Posibles estados inicial y final de la instrucción turnLeft()

Ejercicio. 3 ¿Cómo ejecuta Karel pickBeeper() en los siguientes estados?

a)

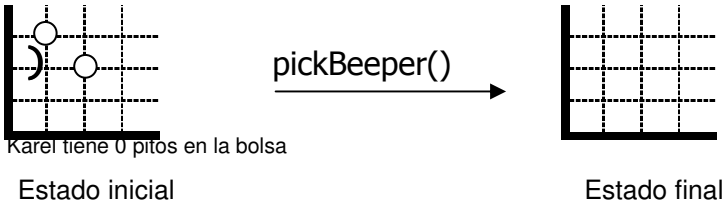


Figura. 58 Posibles estados inicial y final de la instrucción pickBeeper()

b)

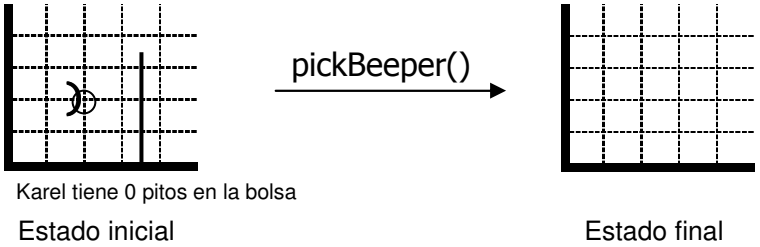


Figura. 59 Posibles estados inicial y final de la instrucción pickBeeper()

c)

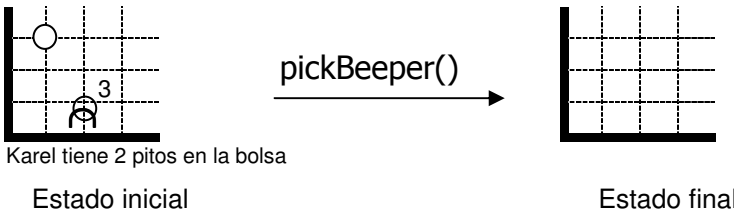


Figura. 60 Posibles estados inicial y final de la instrucción pickBeeper()

d)

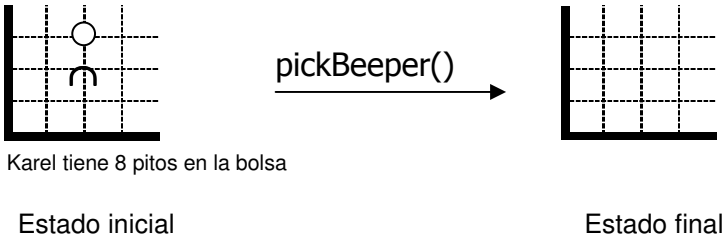


Figura. 61 Posibles estados inicial y final de la instrucción pickBeeper()

Ejercicio. 4 ¿Cómo ejecuta Karel putBeeper() en los siguientes estados?

a)

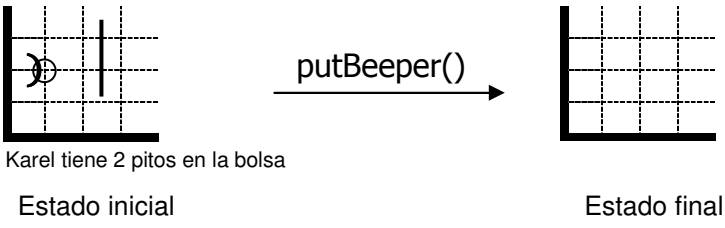


Figura. 62 Posibles estados inicial y final de la instrucción putBeeper()

b)

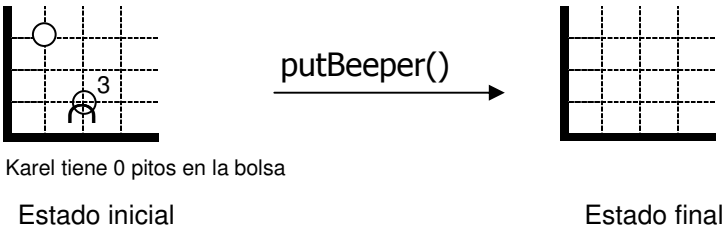


Figura. 63 Posibles estados inicial y final de la instrucción putBeeper()

c)

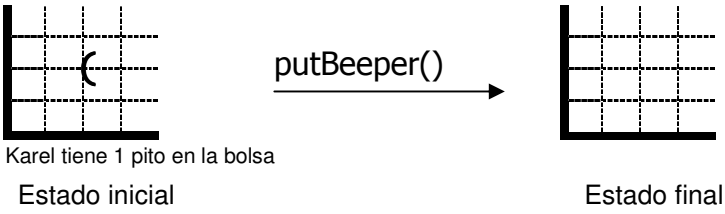


Figura. 64 Posibles estados inicial y final de la instrucción putBeeper()

d)

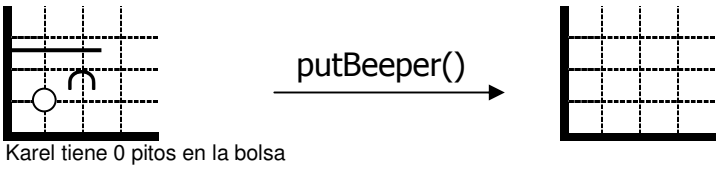


Figura. 65 Posibles estados inicial y final de la instrucción putBeeper()

Ejercicio. 5 ¿Determine las primitivas que se deben utilizar para resolver los siguientes problemas?.

a)

Figura. 66 Posibles estados inicial y final para el uso de primitivas

b)

Figura. 67 Posibles estados inicial y final para el uso de primitivas

c)

Figura. 68 Posibles estados inicial y final para el uso de primitivas

Ejercicio. 6 Resuelva los siguientes problemas identificando sus estados intermedios.

a)

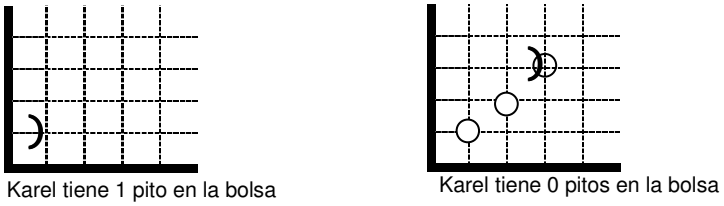


Figura. 69 Posibles estados inicial y final para identificar los intermedios

b)

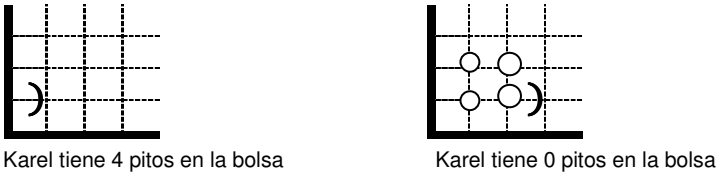


Figura. 70 Posibles estados inicial y final para identificar los intermedios

c)

Figura. 71 Posibles estados inicial y final para identificar los intermedios

d)

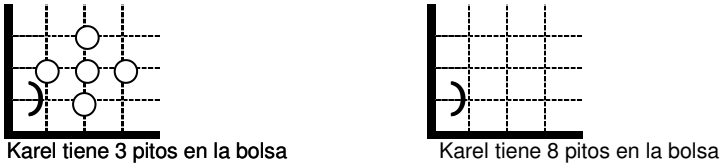
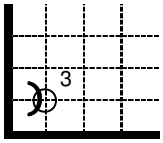


Figura. 72 Posibles estados inicial y final para identificar los intermedios

Ejercicio. 7 Identifique los errores de programación que aparecen en cada uno de los siguientes programas.

a)



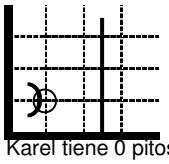
Karel tiene 0 pitos en la bolsa

Estado inicial

Figura. 73 Posible estado inicial para identificar los errores de programación

```
task
{
ur_robot karel(1,1,East,0);
karel.pickBeeper();
karel.pickBeeper();
2 karel.move();
karel.turnLeft();
karel.turnOff();
}
```

b)



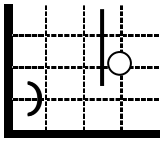
Karel tiene 0 pitos en la bolsa

Estado inicial

Figura. 74 Estado inicial para identificar los errores de programación

```
task
{
ur_robot karel(1,1,East,0);
karel.pickBeeper();
karel.putBeeper();
karel.move();
karel.move();
karel.turnRight();
karel.turnOff() ;
}
```

c)



Karel tiene 0 pitos en la bolsa

Figura. 75 Estado inicial para identificar los errores de programación

```
task
{
ur_robot karel(1,1,East,0);
karel.move();
karel.move();
karel.turnLeft();
karel.move();
karel.pickBeeper();
karel.turnLeft();
karel.move();
karel.turnOff() ;
}
```

3. EXTENSIÓN DEL VOCABULARIO BASICO

En este capítulo se define la forma de enseñarle a Karel nuevas palabras de programación y la técnica de refinamiento paso a paso para resolver problemas.

3.1 INSTRUCCIÓN LOOP

Algunas veces, cuando programamos a Karel, es necesario hacerle repetir una instrucción un número de veces. Anteriormente, la solución de los problemas se efectuó escribiendo una instrucción tantas veces como fuera necesario.

La instrucción loop nos permite simplificar el programa escribiendo esta instrucción una sola vez e indicando el número de veces que se debe repetir, es una ayuda sintáctica para comprimir el código.

La instrucción loop se escribe así:

```
loop (número positivo)
{
  <instrucción>
}
```

Karel repite las instrucciones que están escritas dentro del bloque {…} <número positivo> veces. Si lo que queremos es que repita sólo una instrucción, no podemos omitir el {…}.

Por ejemplo, si queremos que Karel gire a la derecha tres veces, esto lo podemos abreviar utilizando la instrucción loop así:

```
loop(3)
 {
   turnLeft();
 }
```

Veamos el siguiente problema y su solución utilizando la instrucción loop: Karel está en el origen, mirando al este. En todas las esquinas de la calle 1 desde la avenida 1 hasta la 23 hay un pito que Karel debe recoger. Karel termina en la esquina (1,24) con los 23 pitos en su bolsa.

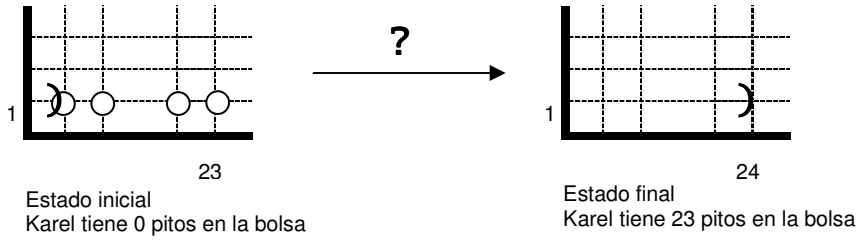


Figura. 76 Posibles estados inicial y final de Karel para la solución del problema utilizando la instrucción loop

Solución: Karel debe repetir el paso básico de recoger un pito y moverse, ya que, se conoce exactamente el número de pitos del mundo y el número de veces que se debe repetir este paso (23). Utilizando la instrucción loop se tiene la siguiente solución:

```
ur_robot karel(1,1,East,0);
loop(23)
 {
   karel.pickBeeper();
   karel.move();
 }
```

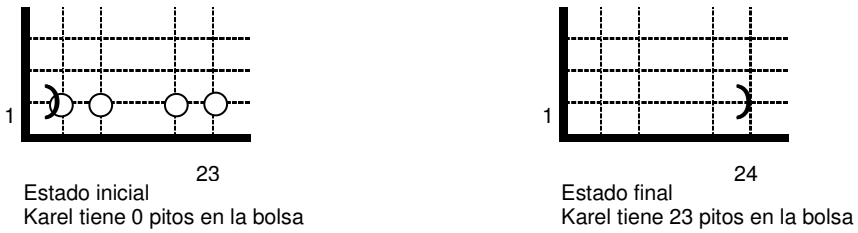


Figura. 77 Posibles estados inicial y final de Karel en la solución del problema utilizando la instrucción loop

3.2 DEFINICIÓN DE NUEVAS INSTRUCCIONES

Karel tiene un mecanismo que le permite aprender, esto nos da la posibilidad de enseñarle nuevas instrucciones adicionales a las primitivas, para que realice acciones más complejas y así ampliarle su vocabulario. Cada instrucción o palabra nueva del vocabulario tiene que estar definida en términos de primitivas o de instrucciones ya aprendidas por Karel. Una instrucción nueva se define dando su nombre y su significado así:

```
class <nombre del robot> : ur_robot
{
    <instrucción>
};
void <nombre de la clase> ::<nuevo método>
{
    <instrucción>
}
```

Donde <instrucción> puede ser una sola declaración o un bloque {…} de instrucciones separadas por ';' (punto y coma).

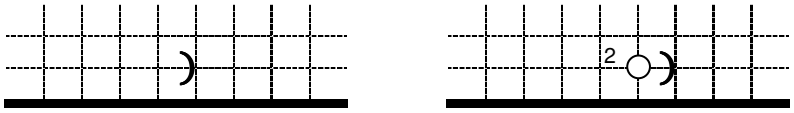
Cuando una nueva instrucción es única, no se puede suprimir los corchetes {...} que delimitan la definición de la instrucción.

Ejemplo, podemos definir la instrucción Gire izquierda como:

```
class zurdo : ur_robot
{
Gire Izquierda ();
};
void Zurdo() :: Gire Izquierda ()
{
    turnLeft();
}
```

Podemos definir una instrucción Pongaysiga para que Karel coloque dos pitos en su esquina y se mueva una cuadra a la derecha, dirección hacia la cual está mirando inicialmente:

```
void Pongaysiga()
      {
           putBeeper();
           putBeeper();
          move();
}
```



Estado inicial
Karek tiene X pitos en la bolsa

Estado final
Karel tiene X-2 pitos en la bolsa

Figura. 78 Posibles estados de Karel en la instrucción Pongaysiga

En un programa la declaración de nuevas instrucciones se coloca entre el comienzo de la clase y el final del bloque de declaraciones.

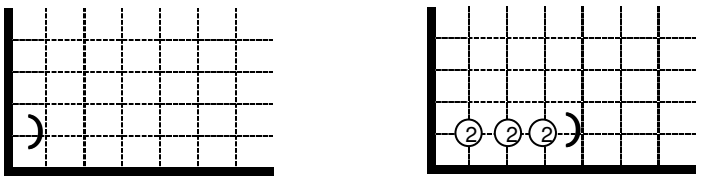
Por ejemplo: entre {después digite class <nombre del robot>: ur_robot y el final del bloque delimitado por};

En un programa la definición de nuevas instrucciones se coloca entre el final del bloque de declaraciones y el comienzo del bloque de ejecución task.

Por ejemplo entre {…}; se escriben primero las instrucciones que sólo utilizan primitivas para su definición, luego las que hacen referencia a las primeras y así sucesivamente hasta llegar al bloque principal.

Entre una definición y la siguiente y entre una definición y el bloque de ejecución no debe colocarse el símbolo ‘;’ (punto y coma).

Problema: Uso de la instrucción Pongaysiga. Para el problema del repartidor



Estado inicial
Karel tiene 7 pitos en la bolsa

Estado final
Karel tiene 1 pito en la bolsa

Figura. 79 Posibles estados inicial y final para la solución del problema del repartidor

```
class Repartidor: robot
{
void Pongaysiga();
};
Vid Repartidor: Pongaysiga()
{
              putBeeper();
              putBeeper();
             move();
}
task
{
  Repartidor karel(1,1,East,7);
       loop(3)
        {
          karel.Pongaysiga();
        }
     karel.turnOff();
}
```

Karel comienza a ejecutar el bloque de ejecución. La primera instrucción es Pongaysiga, como ésta no es una primitiva, Karel busca en el diccionario su definición y la ejecuta, obedeciendo una tras otra, cada una de las instrucciones del bloque de definición de Pongaysiga; cuando termina de ejecutar este bloque continúa la ejecución del programa a partir del sitio donde encontró dicha instrucción.

3.3 TÉCNICA DE PROGRAMACIÓN POR REFINAMIENTO PASO A PASO

La técnica por refinamiento paso a paso consiste en dividir un problema grande en problemas más pequeños o subproblemas y éstos a su vez en otros más pequeños, hasta llegar a un nivel en que los problemas sean sencillos y fácilmente solucionables. La solución de un problema es, entonces, la unión de las soluciones de los subproblemas que a su vez pueden estar dadas como la unión de problemas más pequeños.

En el mundo de Karel para resolver problemas complejos utilizando esta técnica se usan las capacidades de definición de nuevas instrucciones para ir dando nombre a los subproblemas que se van identificando. De esta forma se tiene desde un principio, el bloque principal (de ejecución) del programa.

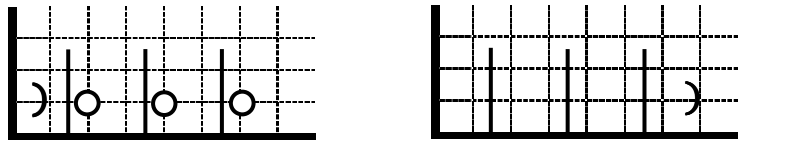
Cada subproblema complejo se resuelve de la misma forma y su solución se convierte en el cuerpo de la instrucción con que se “bautizó” el paso a paso.

La técnica por refinamiento paso a paso se describe así:

1. Hacer un plan general de solución, dividir el problema en subproblemas o pasos.
2. Darle nombre a los diferentes subproblemas que se van identificando, definir las nuevas instrucciones que se van a utilizar
3. Escribir el bloque principal del programa, definiendo los diferentes pasos y asignándoles nombres.
4. Resolver cada subproblema. Si el subproblema es todavía muy complejo, puede ser nuevamente dividido.

Ejemplos de la Técnica de Refinamiento Paso a Paso

Primer caso: Salto de Obstáculos. Karel está en el origen mirando al este. Frente a él hay tres obstáculos cada dos cuadras. Los obstáculos son de dos cuadras de altura; a la derecha de cada obstáculo, sobre la calle 1, se encuentra un pito. Programe a Karel para que salte los tres obstáculos, recoja los tres pitos y termine en la esquina (1,7), mirando al este.



Estado inicial
Karel tiene 0 pitos en la bolsa

Estado final
Karel tiene 3 pitos en la bolsa

Figura. 80 Posibles estados inicial y final de Karel para la solución del problema de obstáculos

Solución: Paso 1: Plan General

En la figura 80 se observa que existen tres obstáculos iguales y para cada uno, Karel debe saltarlo y recoger el pito que está a la derecha. Es lógico, entonces, partir el problema en tres subproblemas iguales:

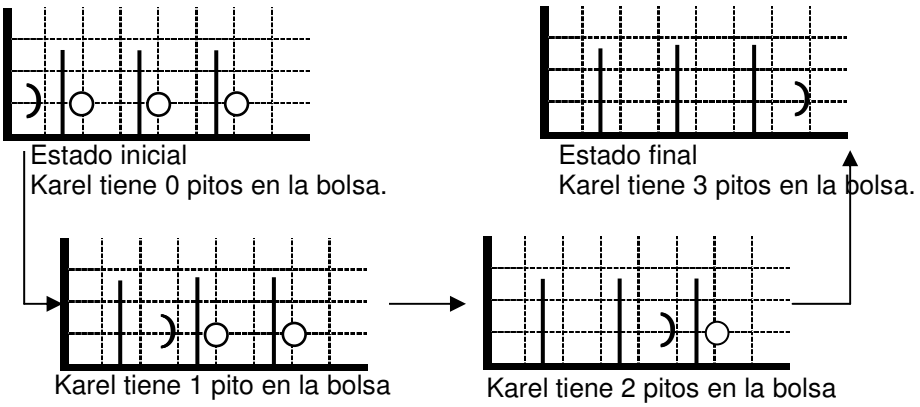


Figura. 81 Posibles estados

Paso 2: Darle nombre a los pasos

Como los tres pasos son iguales, sólo necesitamos definir una nueva instrucción, que vamos a llamar salte-y-recoja, que resuelva el subproblema de saltar un obstáculo y recoger el pito a su derecha.

Paso 3: Escribir el bloque principal de ejecución

El bloque de ejecución se construye pegando las soluciones de los subproblemas en el orden apropiado, en este caso obtenemos lo siguiente:

```
task
{
/* salto es el nombre de la clase */
 salto karel(1,1,East,0);
 loop(3)
  {
    karel.salte_y_recoja;
  }
 karel.turnOff();
}
```

Paso 4: Resolver los subproblemas

En este caso, sólo hay un subproblema, llamado salte-y-recoja, el cual podemos especificar gráficamente así:

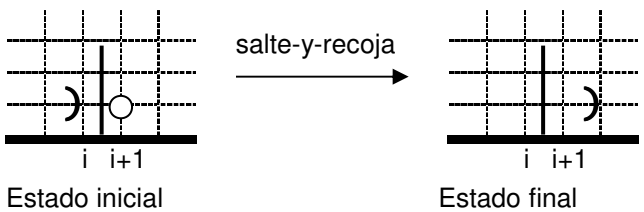


Figura. 82 Posibles estados

Como este problema todavía es un poco complicado para resolverlo con primitivas, podemos volverlo a partir, repitiendo los pasos seguidos para el problema original:

Paso 4.1. Plan general: el problema lo podemos partir en dos subproblemas de la siguiente forma (aunque esta no es la única posibilidad):

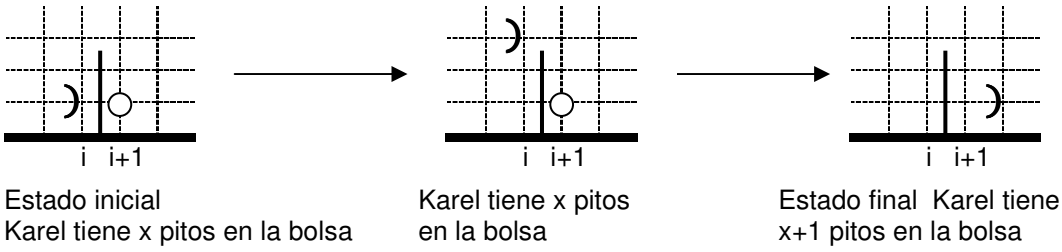


Figura. 83 Posibles estados

Paso 4.2. Nombrar los subproblemas: Como en este caso los dos subproblemas obtenidos son diferentes, debemos darles nombres distintos, por ejemplo, suba y baje respectivamente.

Paso 4.3. Bloque principal de salte-y-recoja: La solución (el cuerpo de su definición) es la unión de suba y baje:

```
void salto::salte_y_recoja()
 {
   suba();
   baje();
 }
```

Paso 4.4. Resolver los subproblemas: Los problemas suba y baje se pueden resolver directamente con primitivas y con la instrucción turnRight() cuya definición se reduce a hacer tres turnLeft()

```
void salto::suba()
 {
   turnLeft();
   move();
   move();
   turnRight();
}


void salto::baje()
 {
   move();
   turnRight();
   move();
   move();
   pickBeeper();
   turnLeft();
   move();
}
```

Para que karel pueda ejecutar correctamente una instrucción no primitiva, debe haberla aprendido antes, teniendo en cuenta esta condición, el programa completo sería el siguiente:

Programa Completo

```
class salto:robot
{
  void turnRight();
  void suba();
  void baje();
  void salte_y_recoja();
};

void salto:turnRight()
{
      turnLeft();
      turnLeft();
      turnLeft();
}

void salto::suba()
{
   turnLeft();
   move();
   move();
   turnRight();
}

void salto::baje()
 {
   move();
   turnRight();
   move();
   move();
```

```
   pickBeeper();
   turnLeft();
   move();
}

void salto::salte_y_recoja()
 {
  suba();
  baje();
}
task
{
/* salto es el nombre de la clase */
  salto karel(1,1,East,0);
  loop(3)
   {
     karel.salte_y_recoja();
   }
   karel.turnOff();
}
```

Segundo caso: Pista de Aterrizaje. En su primer trabajo en la base interestelar, Karel debía iluminar (con pitos) las pistas asignadas a las naves de los marcianos (las diagonales); por descuido iluminó las de los visitantes de Júpiter (verticales y horizontales). Ayude a Karel a cambiar la iluminación:

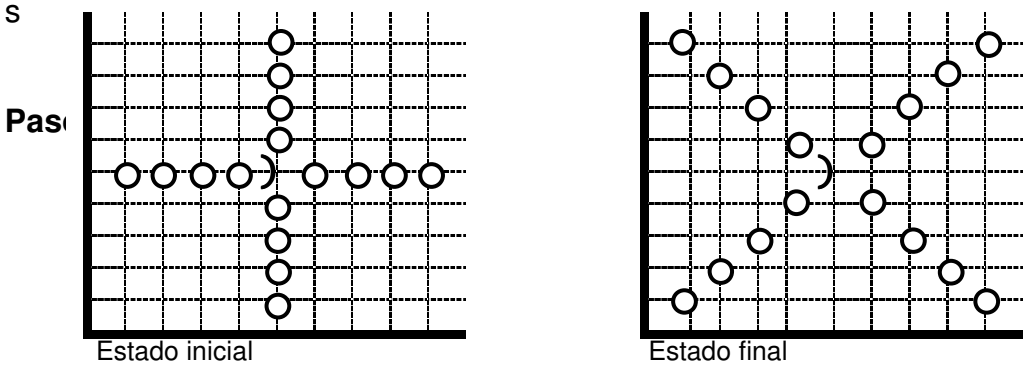


Figura 84 Posibles estados

La idea es dividir el problema en cuatro subproblemas iguales: rotar una fila de

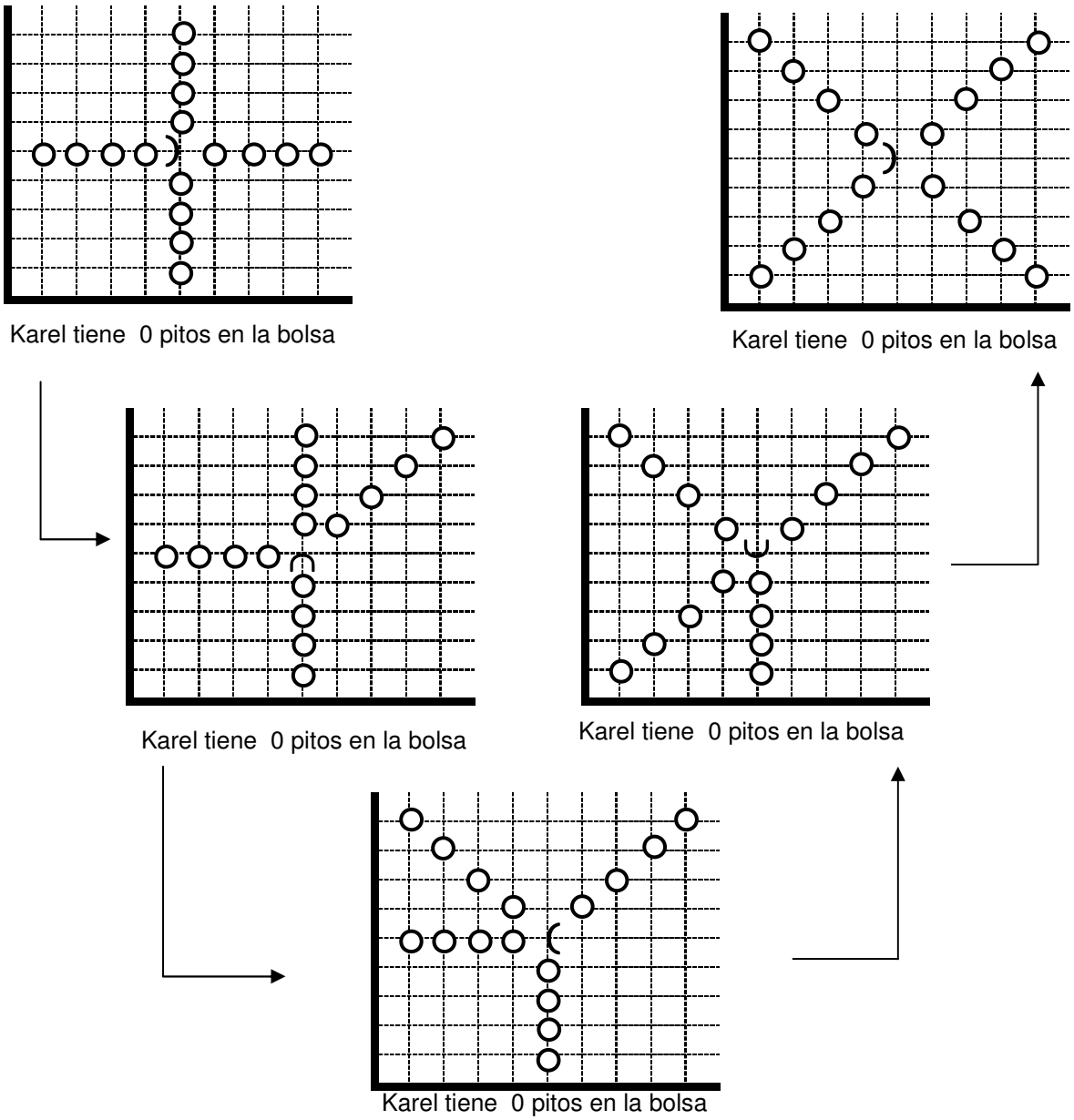


Figura 85 Especificación de los posibles estados

Paso 2: Darle nombre a los pasos

Definimos la instrucción rotar, en la cual Karel comienza mirando hacia la fila que va a recoger y termina mirando hacia la siguiente, después de haber colocado los pitos diagonalmente.

Paso 3: Escribir el bloque principal de ejecución

```
task
{
  /* Pista es el nombre de la clase */
  Pista karel(4,4,East,0);
  loop(4)
  {
    karel.rotar();
  }
  karel.turnOff();
}
```

Paso 4: Resolver los subproblemas

Puesto que rotar es todavía demasiado complicado para resolverlo con primitivas, lo vamos a partir de nuevo:

Paso 4.1. Plan general: El problema se puede dividir en tres subproblemas de la siguiente forma:

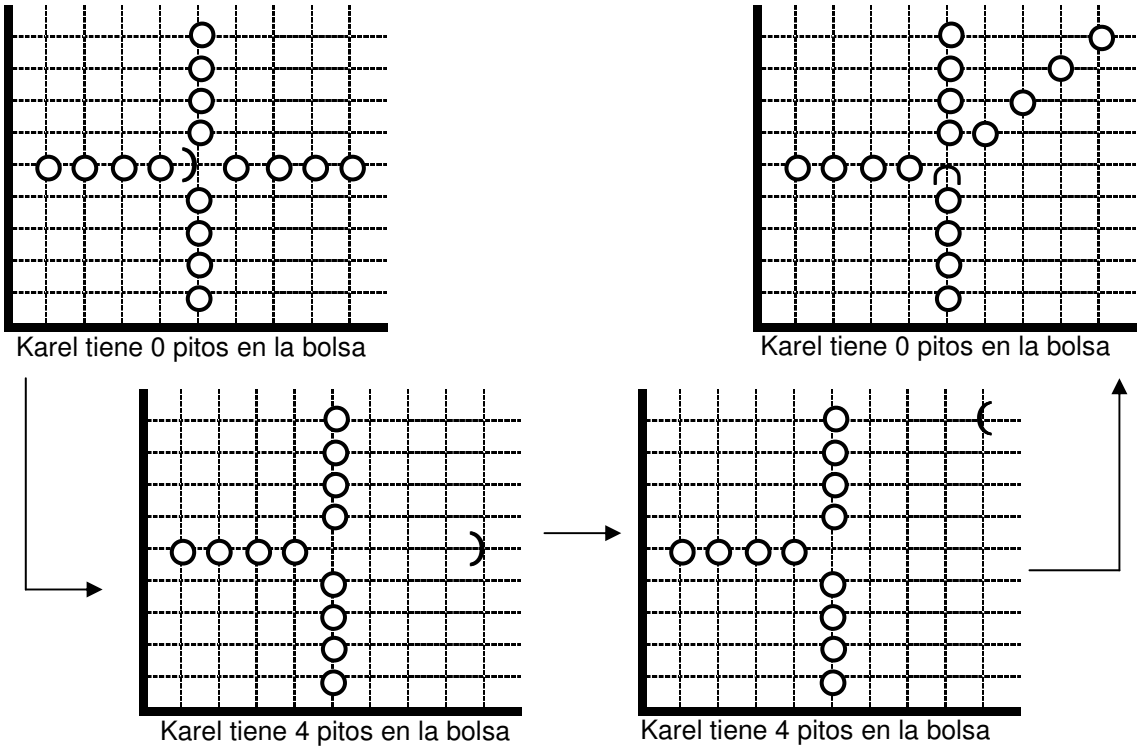


Figura. 86 Estados posibles de los subproblemas

Paso 4.2. Nombrar los subproblemas: cada uno de los tres subproblemas planteados lo vamos a resolver con una nueva instrucción: recoja-recta, vaya - esquina y ponga-diag.

Paso 4.3. Bloque principal de rotar:

```
Void pista::rotar()
{
recoja_recta();
vaya_esquina();
ponga_diag();
}
```

Paso 4.4. Resolver los subproblemas: para resolver la subtarea que colocar la diagonal de pitos, vamos a definir una instrucción adicional:

```
Void pista::diag1()
 {
    putBeeper();
    move();
    turnLeft();
    move();
    turnRight();
 }
```

Coloca uno de los pitos de la diagonal y queda preparado para colocar el siguiente. De esta forma ponga-diag se reduce a:

```
Void pista::ponga_diag()
 {
    loop(4)
     {
        diag1();
         turnRight();
     }
 }
```

Las instrucciones recoja-recta y vaya-esquina se pueden resolver directamente de la siguiente manera:

```
Void pista::recoja_recta()
 {
   loop(4)
    {
```

```
      move();
      pickBeeper();
    }
 }
```

```
Void pista::vaya_esquina()
{
  turnLeft();
    loop(4)
     {
       move();
       turnLeft();
    }
}
```

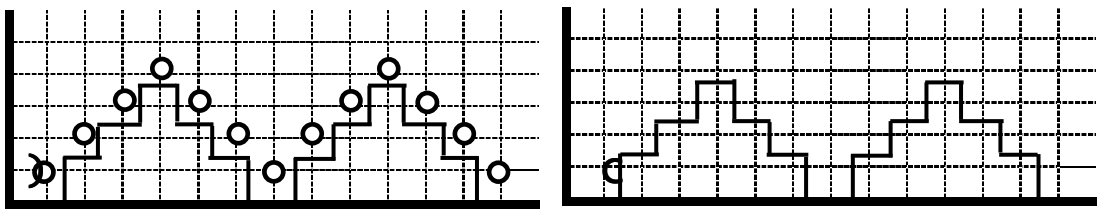
Programa Completo

```
Class pista : robot
  {
     void turnRight();
     void diag1();
     void ponga_diag();
     void recoja_recta();
     void vaya_esquina();
     void rotar()
 };

void pista::turnRight()
 {
    turnLeft();
    turnLeft();
    turnLeft();
 }

void pista::diag1()
 {
   putBeeper();
   move();
   turnLeft();
   move();
   turnRight();
}

void pista::ponga_diag()
 {
    loop(4)
     {
        diag1();
         turnRight();
     }
 }
void pista::recoja_recta()
 {
     loop(4)
```

```
    {
      move();
      pickBeeper();
    }
}

void pista::vaya_esquina()
{
    turnLeft();
    loop(4)
     {
       move();
       turnLeft();
     }
}

void pista::rotar()
{
  recoja_recta();
  vaya_esquina();
  ponga_diag();
}

task
{
    /* Pista es el nombre de la clase */
  Pista karel(4,4,East,0);
  loop(4)
   {
     karel.rotar();
   }
     karel.turnOff(); }
```

3.4. EJERCICIOS PROPUESTOS

3.4.1. El pastor Karel ve que se avecina una tormenta y desea recoger a sus ovejas (pitos) que se encuentran en las dos colinas de la región y volver a la casa (origen).

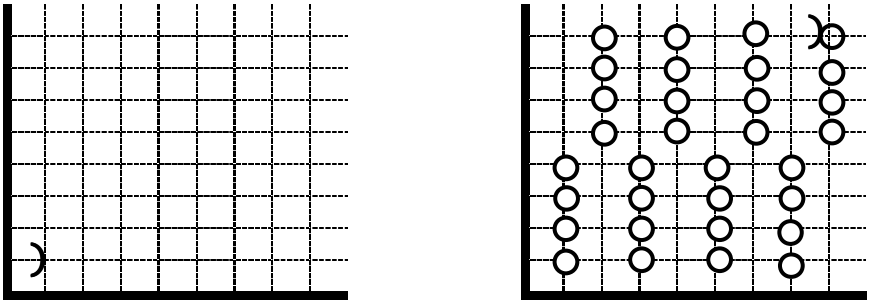


Posible estado inicial
Karel tiene 0 pitos en la bolsa

Estado final
Karel tiene 13 pitos en la bolsa

Figura. 87 Estados inicial y final del ejercicio recogiendo ovejas

3.4.2. El pobre Karel tuvo que ir a la guerra y se encuentra en el frente de batalla; su jefe le acaba de encargar la difícil tarea de poner minas (pitos) en el campo. Programe a Karel para que, partiendo del origen, coloque las 32 minas en la forma que se describe a continuación (Nota: obviamente Karel no debe volver a pasar por una esquina ya minada porque explotaría).

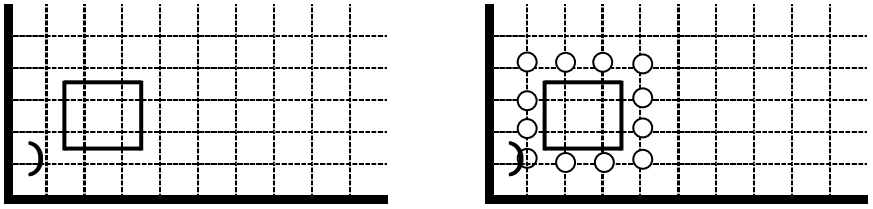


Estado inicial
Karel tiene 32 pitos en la bolsa

Estado final
Karel tiene 0 pitos en la bolsa

Figura. 88 Estados inicial y final del ejercicio colocar minas

3.4.3. Programe a Karel para que coloque baldosines (pitos) alrededor de su piscina cuadrada. Debe partir y terminar en el origen, mirando hacia el este.

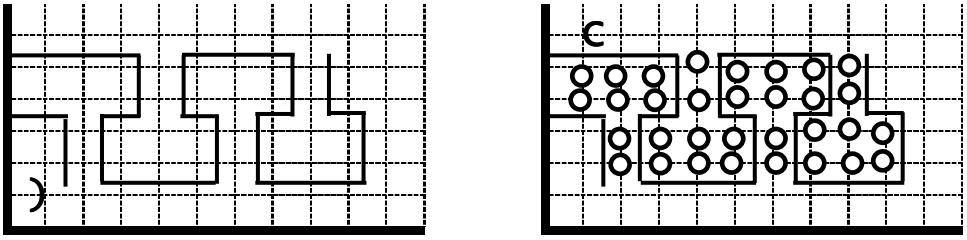


Estado inicial
Karel tiene 12 pitos en la bolsa

Estado final
Karel tiene 0 pitos en la bolsa

Figura. 89 Estados inicial y final del ejercicio colocando baldosines

3.4.4. Karel trabaja en un hotel y debe lavar los tapetes de las 4 habitaciones del primer piso; para cada metro cuadrado (esquina) gasta un balde de agua con jabón (pito). Programe a Karel para que lave los tapetes. A continuación se describe el problema con la planta del piso. (Nota: observe que todos los cuartos son iguales, tanto los que dan hacia el sur como los que tienen la entrada por el norte).

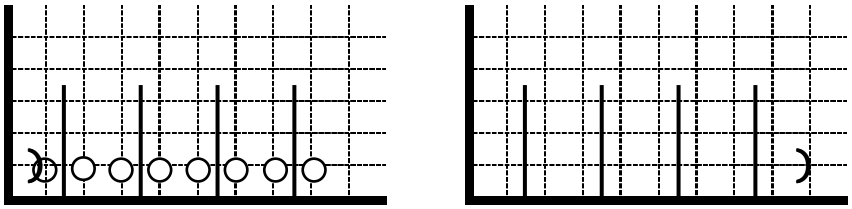


Estado inicial
Karel tiene 40 pitos en la bolsa

Estado final
Karel tiene 8 pitos en la bolsa

Figura. 90 Estados inicial y final del ejercicio lavar tapetes

3.4.5 Modifique el programa de los obstáculos presentados en la teoría, para que resuelva el siguiente problema:

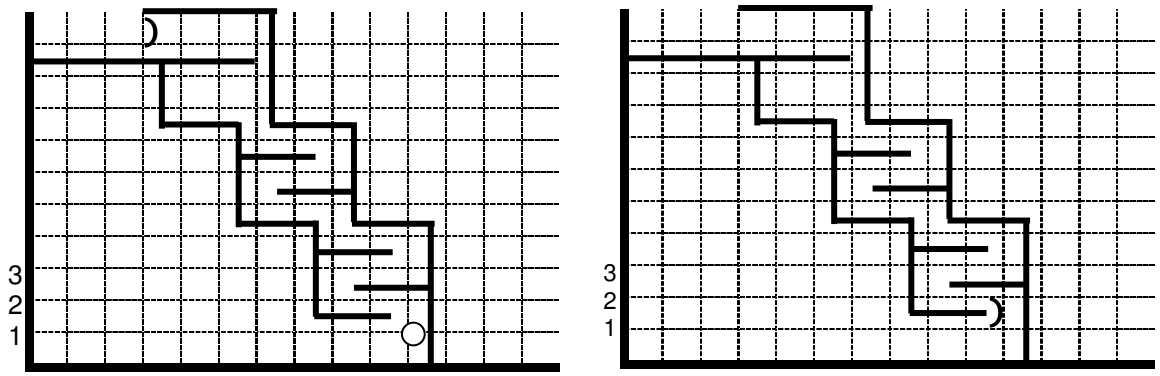


Estado inicial
Karel tiene 0 pitos en la bolsa

Estado final
Karel tiene 8 pitos en la bolsa

Figura. 91 Estados inicial y final del ejercicio obstáculos

3.4.6 Programe a Karel para que recoja el pito que se encuentra al final del laberinto que se muestra a continuación (Ayuda: note que el laberinto no es completamente irregular).



Estado inicial
Karel tiene 0 pitos en la bolsa

Estado final
Karel tiene 1 pito en la bolsa

Figura. 92 Estados inicial y final del ejercicio laberinto

3.4.7 Karel, el jardinero del parque VERDE debe abrir un camino en forma de cruz para que los peatones pueden atravesar el parque por dentro. Para esto debe talar algunos árboles de la manzana del parque. Programe a Karel para

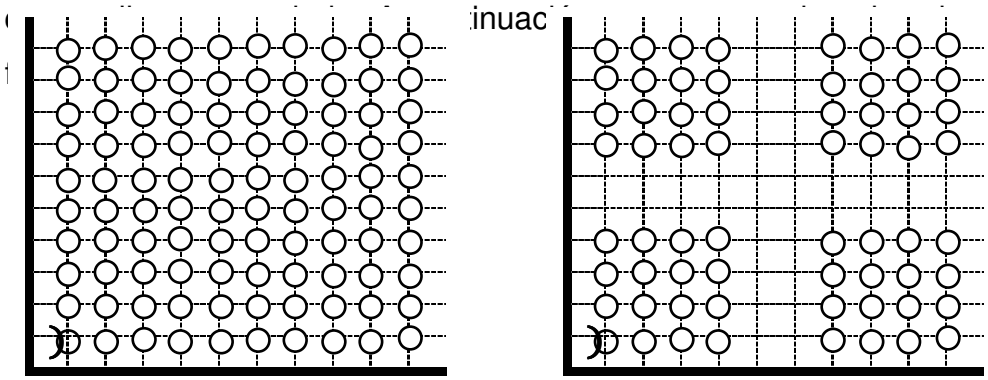


Figura. 93 Estados inicial y final del ejercicio del jardinero

3.4.8 Programe a Karel para que recoja la diagonal de 5 pitos que va de sureste a noroeste y construya con estos pitos una diagonal que parta del origen y se extienda hacia el nordeste.

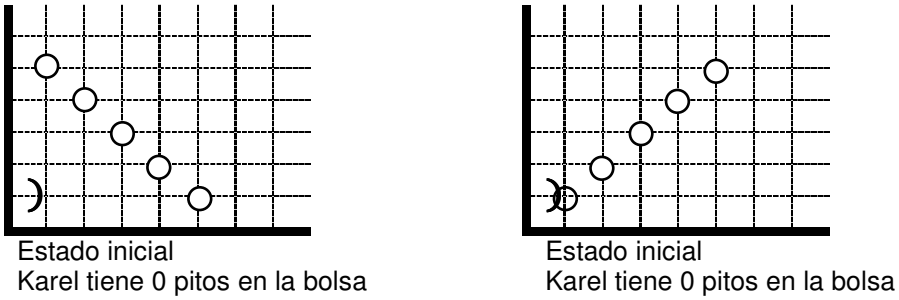


Figura. 94 Estados inicial y final del ejercicio diagonal sureste a noreste

3.4.9 Programe a Karel para que, comenzando en el origen y mirando al este con 6 pitos en su bolsa, coloque todos sus pitos en una línea recta de pendiente 1/2 a partir del origen. Resuelva el problema usando refinamiento paso a paso.

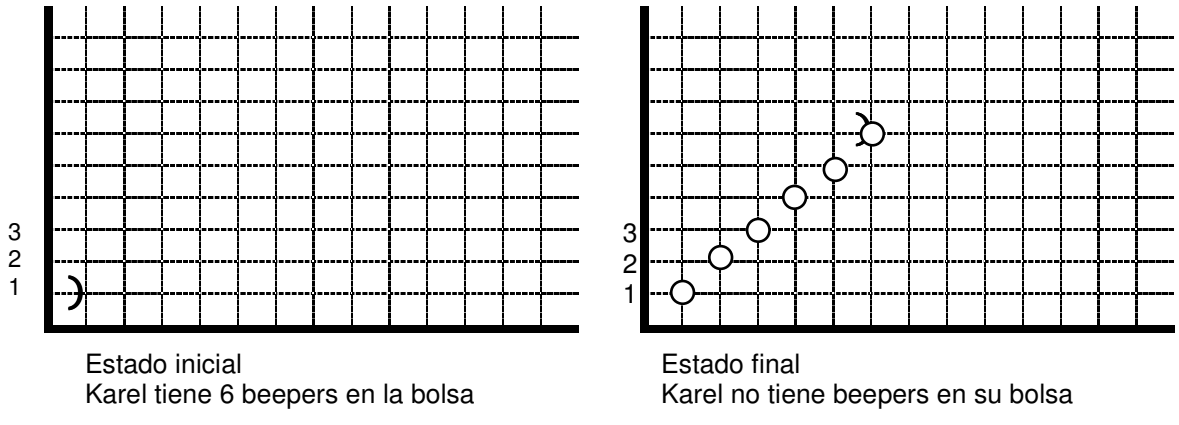


Figura. 95 Estados inicial y final del ejercicio línea recta de pendiente

3.4.10 Haga un programa para que Karel recoja todos los pitos de su mundo. Estos se encuentran cubriendo una escalera, como se muestra a continuación:

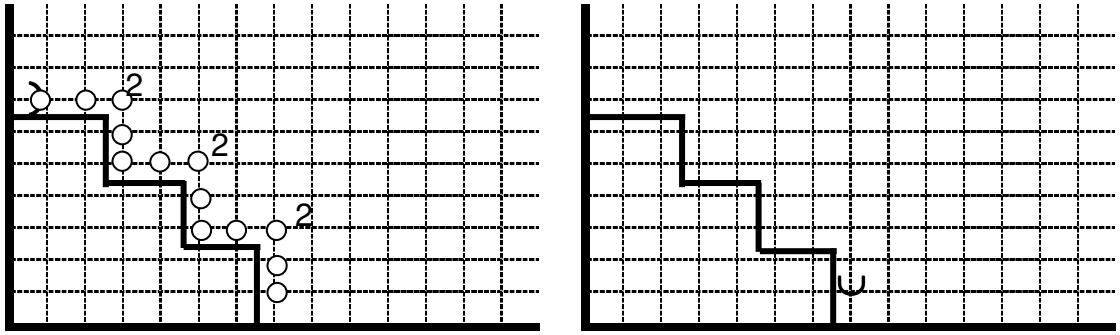


Figura. 96 Estados inicial y final del ejercicio recoger pitos de la escalera

Fíjese que en las esquinas de los escalones hay dos pitos y en las otras esquinas que rodean la escalera sólo hay uno. Karel parte de la calle 7 avenida 1 mirando al este y debe terminar en la calle 1 avenida 7 mirando al sur. Resuelva el problema usando refinamiento paso a paso.

3.4.11 Programe a Karel que comenzando en el origen y mirando al norte no tiene pitos en su bolsa, recoja todos los frutos del sembrado de lulo que se muestra en la siguiente figura (cada lulo es un pito) y regrese al origen. Resuelva el problema usando refinamiento paso a paso.

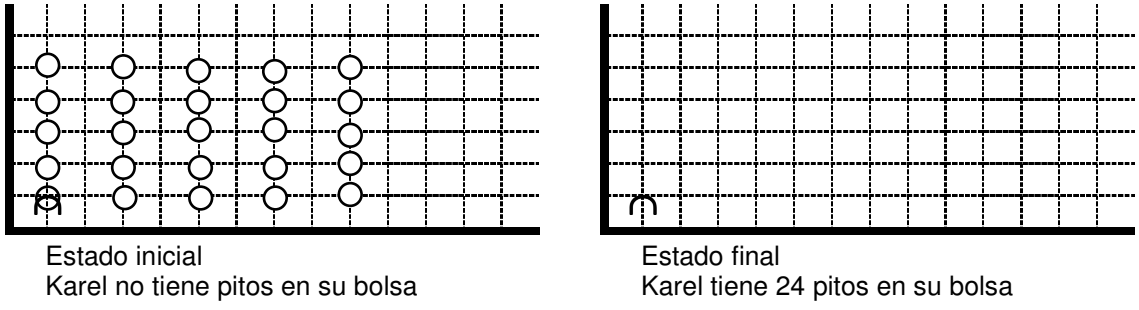
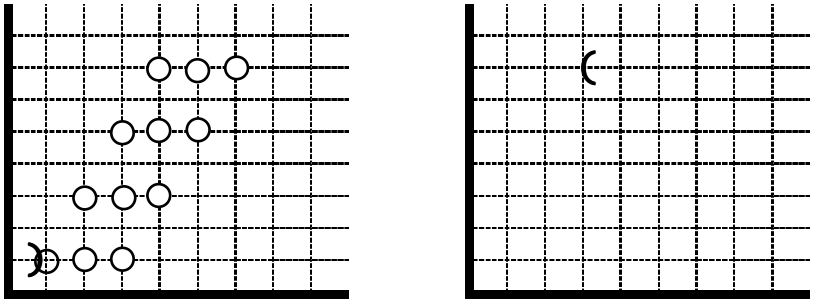


Figura. 97 Estados inicial y final del ejercicio recoger frutos

3.4.12 Programe a Karel que partiendo del origen y mirando al este, recoja las 12 alcachofas (pitos) de su sembrado y termine en la esquina (7,3), mirando al oeste. Resuelva el problema usando refinamiento paso a paso.

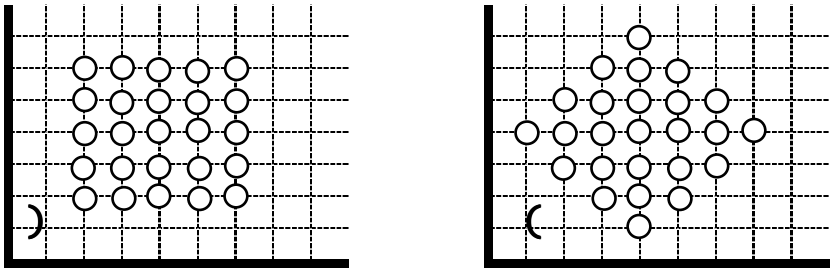


Estado inicial
Karel tiene 0 pitos en la bolsa

Estado final
Karel tiene 12 pitos en la bolsa

Figura. 98 Estados inicial y final del ejercicio recoger alcachofas

3.4.13 Karel tiene una huerta de lechugas (pitos) cuadrada (de 5 por 5), que va de la esquina (2,2) a la (6,6). Por problemas de riego quiere distribuir las lechugas formando un diamante. Programe a Karel para que resuelva su tarea partiendo del origen en dirección este y terminando en esa misma posición. Ayuda: mueva el mínimo número de lechugas posibles.



Estado inicial
Karel tiene 0 pitos en la bolsa

Estado final
Karel tiene 0 pitos en la bolsa

Figura. 99 Estados inicial final del ejercicio huerta de lechugas

3.4.14. Programe a Karel que comenzando en la esquina (2,1) y mirando al este, con 27 pitos en su bolsa, siembre 27 matas de lulo como se indica en la figura (cada semilla de lulo es un pito) y luego regrese al origen. Resuelva el problema usando refinamiento paso a paso.

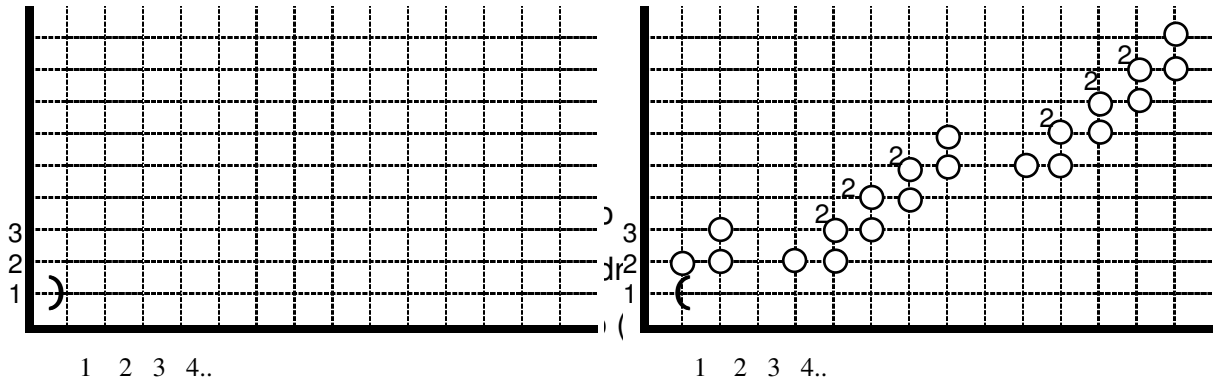


Figura. 100 Estados inicial y final del ejercicio matas de lulo

3.4.15 Karel se encuentra en un campo de trigo, montando a caballo (este caballo se mueve como el caballo de ajedrez). Programe Karel para que, partiendo del origen, recoja todos los atados de trigo (pitos) que se encuentran en el rectángulo de 12 pitos que se muestra en la figura.

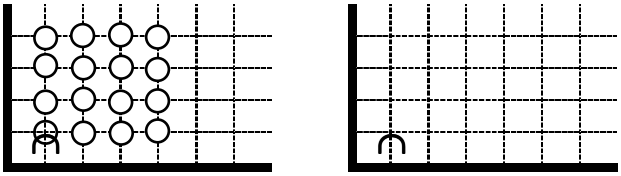


Figura. 101 Estados inicial y final del ejercicio campo de trigo

4. AMPLIANDO LA PROGRAMACION DEL ROBOT

Este capítulo explica el mecanismo de especificar una nueva clase de robots y adicionar nuevas instrucciones al vocabulario del robot. También, se discutirán algunos métodos de programación para poner en práctica y probar los programas.

4.1. CREANDO UN LENGUAJE DE PROGRAMACIÓN MÁS NATURAL

Se dice que un robot realiza una tarea compleja, también que recibe muchos mensajes para ejecutar tareas.

El lenguaje de programación de robots permite al programador de robots especificar nuevas clases de robots. Esta descripción de clases suministra especificación de nuevas instrucciones de robot. Karel-Werke intentará entonces usar la descripción de clases para crear robots capaces de interpretar los nuevos mensajes.

La disponibilidad para que aprenda un robot es bastante limitada. Karel-Werke construyó cada robot con un *diccionario* de nombres de métodos útiles y sus definiciones, pero cada definición tiene que ser construida con instrucciones más simples que los robots ya entienden.

Para suministrar a los robots con un diccionario de instrucciones que ejecute acciones complejas, se puede construir un vocabulario robot que corresponda a su propietario. Concedido este mecanismo se pueden solucionar nuestros problemas de programación usando instrucciones que sean naturales para nuestra forma de pensar y entonces podemos proveer robot con la definición de estas instrucciones.

Se puede definir una instrucción **moveMile()** con 8 mensajes de **move()**, entonces, cuando un robot es avisado para una instrucción moveMile() en un programa, éste busca la definición asociada con este nombre de mensaje y lo ejecuta. Ahora nuestro inmanejable programa de movimiento de pitos puede ser escrito con una definición **moveMile()**, conteniendo 8 mensajes de **move()**, y otros 15 mensajes de **moveMile().** Este programa contiene estos 23 mensajes, que serán más comprensibles con el programa original, que necesita más de 120 mensajes para culminar la tarea.

En problemas complicados la disposición de ampliar el vocabulario del robot permite la comprensión de los programas

4.2. UN MECANISMO QUE DEFINE NUEVAS CLASES DE ROBOTS

La declaración de la clase primitiva **ur_robot().** Los usuarios del lenguaje robot pueden también declarar nuevas clases de robot y la instalación capaz de enviarlos, exactamente en el estándar de los robots. Para especificar una nueva clase de robots, se incluye una especificación de clase en la sección de declaración al comienzo de cada uno de los programas. Aislada de un programa, la forma general de especificación se muestra a continuación:

class <Nuevo nombre de clase> extends <nombre clase vieja>
{
<lista -de-nuevos-métodos>
}

La clase de especificación usa la palabra reservada **class** and **extends(:)**, y los símbolos especiales entre paréntesis, separan varias partes de la declaración. Esta forma general incluye elementos delimitados por paréntesis angulares, < >, que pueden ser remplazados con una substitución apropiada cuando se incluye una especificación en un programa robot.

Los paréntesis angulares no son parte de los lenguajes de programación de robots, son una estructura de programa donde los elementos del lenguaje pueden aparecer. En este caso, <nueva-clase-nombre> tiene que ser remplazada por un nuevo nombre, todavía no usada en el programa. Este nombre puede ser construido con letras minúsculas y mayúsculas, dígitos y caracteres subrayados pero no enfrentado con cualquier otro nombre en el programa, ni escribir correctamente cualquier palabra reservada.

Los nombres también deben comenzar con una letra. La reposición para r <anterior-clase-nombre> es el nombre de la clase de robot existente, ya sea ur_Robot() o una declarada previamente en el siguiente programa.

Se puede hacer así con la especificación de una nueva clase a continuación:

```
class Caminante_Millas : ur_Robot
{
void moveMile();
};
void Caminante_Millas  :: moveMile()
        {       ...     // instrucciones excluidas por ahora.
        }
}
```

El nombre de la nueva clase de robot es **Caminante_Millas()**. Se indica también, por el nombre de la clase **ur_Robot()** seguida de (**:)** que significa **ampliado** que la clase **Caminante_Millas()** seguido de ampliado (:), ese **Caminante_Millas()** tiene todas las capacidades de los miembros de la clase **ur_Robot()**.

La especificación **ur_Robot()** es la clase padre de Caminante_Millas que **Caminante_Millas()** es una subclase de **ur_Robot()**. También se dice que los robots de la nueva clase heredan todas las habilidades de la clase padre. Por consiguiente, Caminante_Millas sabe como moverse y girar a la izquierda, al

igual que los miembros de la clase **ur_Robot()**. Ellos pueden también recoger y poner pitos y dar vueltas apropiadamente.

Cada método en la lista es escrito con su definición en paréntesis. Esta especificación dice que cuando un robot en la clase es inicialmente cambiada será capaz de ejecutar instrucciones de Caminante_Millas correctamente como todos los métodos heredados de la clase **ur_Robot()**.

La declaración de clase introduce los nombres de nuevos métodos de robots.

4.3. DEFINICIÓN DE NUEVOS MÉTODOS

Tal como se declara una nueva clase de robot, es necesario definir todas las instrucciones introducidas en éste. Estas definiciones son parte de la declaración de clase en la parte de declaración del programa robot. La forma de una definición de instrucción se ilustra a continuación:

```
void <instruccion_nombre> ()
{
        <lista_de_instrucciones>
}
```

Siempre se debe comenzar con la palabra reservada **void**. Se tiene que dar el nombre de la instrucción que se define entre los paréntesis delimitadores, dando una lista de instrucciones similar al bloque de tarea principal que llama a un robot de esta clase, para continuar con la nueva instrucción. Esta lista de instrucciones delimitada por paréntesis recibe el nombre de bloque en el vocabulario de la programación del robot.

Toda instrucción en la lista debe terminar con un punto y coma. Muchas de las instrucciones en la lista de instrucciones son mensajes, la instrucción **moveMile()** en la clase **CaminadorMile()** puede ser escrita así:

```
class CaminadorMile: ur_Robot
{
 void moveMile();
```

```
};
void CaminadorMile ::moveMile()
        {
                move();
                move();
                move();
                move();
                move();
                move();
                move();
        move();
        }
```

El lenguaje tiene que ser diseñado de tal forma que el robot se refiera a él mismo, como una palabra reservada, en este caso la instrucción move, en el caso citado puede ser remplazada por **this.move()**, pero no se requiere. Esto hace que el lenguaje sea más conciso. "this" significa "this robot."

Si se tiene un CaminadorMile() llamado **Lisa**, puede llegar a caminar una milla así: Lisa.moveMile();
o así:

```
Lisa.move();
Lisa.move();
Lisa.move();
Lisa.move();
Lisa.move();
Lisa.move();
Lisa.move();
Lisa.move();
```

En este caso, **Lisa** se mueve 8 veces recibiendo un mensaje de **moveMile()**. El programa completo del robot para el anterior caso es:

```
class CaminadorMile :ur_Robot
{
  void movemile();
}
void  CaminadorMile::moveMile()
```

```
        {
                move();
                move();
                move();
                move();
                move();
                move();
                move();
                move();
        }

task
{       CaminadorMile Lisa  (3, 2, East, 0); // Declaración de un nuevo CaminadorMile Lisa.

        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.pickBeeper();
        Lisa.turnLeft();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.turnOff();
}
```

4.4. EL ENTENDER Y EXACTITUD DEL NUEVO MÉTODO

Un robot es una máquina, un dispositivo completamente desprovisto de inteligencia. El robot hace exactamente lo que le “decimos”, no hay lugar para interpretación. Una declaración de clase de robot es una descripción de la

factoría de robot que dice como construir robot de esta clase. En la fábrica de robot una declaración entera parte de cualquier programa de robot que es leído y examinado para errores. Como parte de la fabricación y proceso de entrega de los robots está dada la definición de cada una de las nuevas instrucciones y sus clases.

Cada robot almacena la definición de las instrucciones en su propio diccionario de instrucciones. Así, cuando llamamos al robot de la clase **CaminadorMile()** para hacer moveMile(), este recibe el mensaje, consulta el diccionario para ver como tiene que responder y entonces ejecuta la requerida acción. El piloto del helicóptero no tiene que leer esta parte del programa cuando asigna el robot para repartos.

Según las reglas de gramática de la programación de robots, esta es una definición perfectamente legal por esto contiene errores lexicográficos o sintácticos. Sin embargo, la instrucción moveMile le dice al robot que ejecute la instrucción moveMile(), equivalente a ejecutar 6 instrucciones de move( );.

Cuando simulamos la ejecución de un robot de una instrucción definida, tenemos que adherirnos a las reglas que el robot usa para ejecutar estas instrucciones. El Robot ejecuta una determinada instrucción definida por desempeño de acciones asociadas con esta definición. El robot no conoce cual instrucción definida denota, el robot solo conoce como está definida. Se puede reconocer el significado de esta distinción y aprender a interpretar el programa robot como literalmente lo hace el robot. El significado de nombres es supuesto para ayudar al lector humano a entender el programa.

Si la instrucción actual define el significado del nombre este desacuerdo con el significado del nombre, es fácil estar desorientado.

4.5. DEFINIENDO NUEVOS MÉTODOS EN UN PROGRAMA

En esta sección se expone un programa completo de robot que usa el método de definición de mecanismo. Primero rastreamos la ejecución del programa.

Se discutirá la forma general del programa que usa el nuevo método de definición de mecanismo. La tarea se ilustra en la figura 58 “Limpiar la escalera”, este recoge cada pito en el micromundo mientras asciende por la escalera.

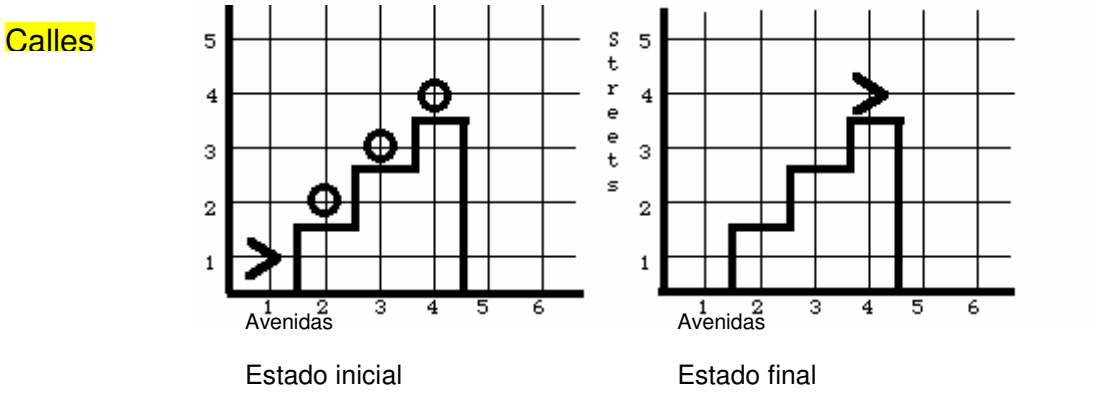


Figura. 102 Programa “Limpiar la escalera”

```
class Limpia escalera : ur_Robot
{
void turnRight();
void climbstair();
};
        void Limpiaescalera ::turnRight()
        {
                turnLeft();
                turnLeft();
                turnLeft();
        }
                void Limpiaescalera ::climbstair()
        {
                turnLeft();
                move();
                turnRight();
                move();
        }

task
{       Limpiaescalera  Alex(1, 1, East, 0);
```

```
        Alex.climbstair();
        Alex.pickBeeper();
        Alex.climbStair();
        Alex.pickBeeper();
        Alex.climbStair();
        Alex.pickBeeper();
        Alex.turnOff();
}
```

A continuación, se suministra una versión comentada del mismo programa con números en cada instrucción y mensajes en el orden en que se están ejecutando, comenzando con la especificación de la instrucción de reparto así #n.

```
class Limpia escalera : ur_Robot
{
}
        void  Limpiaescalera::turnRight()
        {// to here from      #4,        #13 or     #22
                turnLeft();  #5 or      #14 or     #23
                turnLeft();  #6 or      #15 or     #24
                turnLeft();  #7 or      #16 or     #25
        }//   return to      #4         #13        #22

        void  Limpiaescalera::climbStair()
        {// to here from      #1,        #10, or #19
                turnLeft();  #2         #11        #20
                move();                 #3         #12        #21
                turnRight();            #4         #13        #22
                move();                 #8         #17        #26
        }//    return to     #1         #10        #19

task
{        Limpiaescalera  Alex(1, 1, East, 0);          #0

        Alex.climbStair();                                          #1
        Alex.pickBeeper();                                          #9
        Alex.climbStair();                                          #10
```

```
        Alex.pickBeeper();                                    #18
        Alex.climbStair();                                    #19
        Alex.pickBeeper();                                    #27
        Alex. turnOff();                                      #28
}
```

Cuando un programa se está ejecutando, llamamos a la instrucción actual el "foco de la ejecución". Cuando el piloto del helicóptero comienza a ejecutar un programa, el foco está inicialmente en la primera instrucción dentro del bloque de la tarea principal. En este programa de muestra el foco inicial es el mensaje del climbstair, que se anota como # 1. El mensaje del climbStair() se envía a través del satélite al robot Alex. Cuando Alex recibe este mensaje, el foco de la ejecución pasa del piloto a Alex. El piloto, que debe esperar hasta que se haya terminado la instrucción, tiene que ser muy cuidadoso de recordar en que parte del programa estaba cuando recibió el mensaje del climbStair().

En el programa de muestra el nuevo punto de la ejecución se anota como "de aquí # 1". Alex se centra en la lista de las instrucciones que definen la nueva instrucción, climbStair(), y encuentra un mensaje del turnleft (marcado como # 2). Alex entrena al foco de la ejecución en el mensaje del turnLeft() lo ejecuta, y después se centra en # 3, movimiento. Alex ejecuta este movimiento y focos en # 4, turnRight(). Puesto que esto no es una instrucción primitiva, Alex debe recuperar su definición de su diccionario. Después se centra en la lista de la instrucción de la definición del turnRight() y ejecuta las tres instrucciones del turnLeft(), # 5, # 6, y # 7. Terminar la ejecución de la instrucción del turnRight(), Alex ahora vuelve su foco al lugar en el cual el mensaje del turnright ocurrió dentro de climbStair(). Alex cambia de puesto el foco a #8 y ejecuta el movimiento.

Después de que Alex realice este movimiento, se acaba de ejecutar la instrucción climbStair() como las ejecuciones de las producciones de nuevo al piloto puesto que ha realizado totalmente la tarea requerida por el mensaje climbstair. El foco de la ejecución vuelve al lugar en el programa marcado # 1, y el piloto envía Alex el mensaje del pickBeeper() que está marcado # 9.

Con este mensaje, el foco de la ejecución se pasa otra vez de piloto al robot. Alex también interpreta y realiza esta instrucción de pickBeeper() y rinde el foco de nuevo al piloto en # 10. El piloto entonces envía a Alex otro mensaje climbStair().

Alex repite esta misma secuencia de pasos una segunda vez para subir-escalera marcado instrucción #10 y el pickBeeper() que sigue y otra vez para los terceros mensajes de subir-escalera y del pickBeeper(). Alex finalmente ejecuta la instrucción de salida, entonces la ejecución de programa es completa.

El piloto y el robot deben recordar siempre la posición exacta en el programa donde estaban cuando el foco cambia. Esto permite que la ejecución vuelva a su lugar correcto y continúe ejecutando el programa. Es importante que entendamos que no hay reglas complejas para ejecutar un programa que contiene nuevas instrucciones.

Se debe entender como el piloto del helicóptero y el robot trabajan juntos para ejecutar un programa que incluya el mecanismo de la definición de la instrucción.

Anteriormente, se menciono que las declaraciones están escritas siempre antes del bloque de la tarea principal. En nuestro ejemplo de programación, vimos que la declaración de la nueva clase y la definición de las nuevas instrucciones para la clase están consignadas aquí. Debemos escribir siempre nuestras nuevas definiciones de la instrucción en esta área.

Los nombres definidos dentro de una clase del robot, incluyendo los nombres de la clase del padre y del padre del padre, así sucesivamente (llamados los antepasados), se llama el diccionario de la clase. La declaración del ur_Robot() no necesita ser incluido en los programas del robot, puesto que es un "estándar de la fábrica."

Cualquier otra clase que usted necesite definir se debe escribir totalmente en la pieza de la declaración, junto con las definiciones de todas sus instrucciones. Pronto veremos un método de taquigrafía para hacer la escritura de este diccionario mucho más fácil. Cada instrucción en un bloque es terminada por un punto y coma.

Las palabras de la clase no son permanentes y el mundo no recuerda ninguna definición del programa al programa. Cada vez que escribimos un programa del robot, debemos incluir un sistema completo de todas las palabras requeridas en ese programa.

4.6. LOS MÉTODOS HEREDADOS DE MODIFICACIÓN

La construcción de la clase CaminadorMile() le dio al robot la capacidad de caminar una milla a la vez, se conservó su capacidad de caminar un bloque a la vez. Construir una clase en la cual se redefine una instrucción previamente definida debe tener un nuevo significado. Por ejemplo, suponga que teníamos un problema en el cual un robot necesitó siempre moverse por millas, pero nunca al lado de los bloques.

En este caso sería una ventaja crear una clase con una nueva definición de instrucciones de movimiento, de modo que, cuando un robot de esta clase reciba una orden para moverse, éste se mueva una milla.

```
class Mile_Mover extends ur_Robot
{       void move()
        {       super.move();
                super.move();
                super.move();
                super.move();
                super.move();
                super.move();
                super.move();
                super.move();
        }
```

En la nueva definición de movimiento **move()** esta clase elimina la definición original heredada de la clase ur_Robot(). Ahora tenemos un problema, puesto que para mover una milla, necesitamos poder mover ocho bloques, pero estamos definiendo un movimiento que equivale a una milla. Por lo tanto, no podemos decir movimiento ocho veces, necesitamos indicar que deseemos utilizar la original, o eliminar, la instrucción **move()**, desde la clase ur_Robot().

Podemos hacer esto puesto que ur_Robot() es la clase del padre. Apenas necesitamos introducir el mensaje del movimiento **move()** con la palabra clave estupenda y un período. Nos da una manera de especificar un método particular de la clase del padre. Ahora terminamos el programa anterior con:

```
task
{       Mile_Mover Karel(5, 2, North, 0);
        Karel.move();
        Karel.pickBeeper();
        Karel.move();
        Karel.putBeeper();
        Karel.turnOff();
}
```

Karel encontrará el pito en (13, 2) y lo dejarán en (21, 2).

Aviso ahora que si teníamos robots en el mismo programa y enviar a cada uno de ellos los mismos mensajes, puede suceder que cada uno de ellos responda en forma diferente a estos mensajes.

En detalle, un CaminadorMile() mueve solamente un bloque cuando está **move()**, mientras que un Mover_Mile(), mueve una milla.

4.7. UN PROGRAMA GRAMATICALMENTE ERRÓNEO

En el ejemplo que se ilustra a continuación se observa un error de programación común, omitir apoyos necesarios alrededor de un bloque. La definición del longMove() define la instrucción correctamente, pero se ha

omitido el apoyo de la abertura del par que debe incluir las tres instrucciones del movimiento. ¿Usted debe encontrar el error? ¿Está usted confundido por el otro error en este ejemplo?

Encontrar el error de la sintaxis no es fácil, porque las identaciones ayudan a que sea correcto para nosotros.

```
class Big_Stepper : ur_Robot
{
void longMove();
};
        void Big_Stepper:: longMove()
                move();
                move();
                move();
        }

task
{       Big_Stepper Tony(5, 2, North, 0);
        Tony.longMove();
        Tony.turnLeft();
        Tony.turnOff();
}
```

La instalación lee la parte de la declaración de un programa y del bloque de la tarea principal del programa para comprobar si hay errores léxicos y de la sintaxis.

Un lector descubre errores de sintaxis comprobando los componentes "significativos" del programa y comprobando si la gramática y la puntuación son apropiadas. Los ejemplos de componentes significativos son declaraciones de la clase, definiciones del método, y el bloque de la tarea principal. En efecto verificamos los componentes significativos por separado.

Ilustremos cómo la instalación encuentra el error en el programa haciendo uso de esta técnica. La instalación lee sólo las palabras del programa y no es

influenciada por nuestra identación. La fábrica examina la nueva declaración de la clase del robot. Tiene un nombre, una clase del padre, y una lista correcta de características.

Compruebe la puntuación, verifique el nombre de la clase y el nombre del método en la definición de la instrucción. Determine un bloque para incluir la definición. No encuentra el apoyo de la abertura, en su lugar encuentra un nombre. La instalación nos dice que existe un error de sintaxis.

En resumen, el olvidarse de utilizar apoyos necesarios alrededor de un bloque puede conducir a errores de sintaxis. Debemos ser expertos que rápidamente nos convierta en analista de programas. Dado un programa de robot se debe verificar la gramática y errores de puntuación.

4.8. HERRAMIENTAS PARA DISEÑAR Y ESCRIBIR LOS PROGRAMAS DE KAREL

Diseñar las soluciones para los problemas y escribir programas del robot implica como solucionar un problema.

Un modelo describe como solucionar un problema. Un proceso consta de cuatro actividades:

Definición del problema
Planeación del problema
Solución del problema
Ejecución y análisis de la solución del problema

La definición inicial del problema se presenta por medio de figuras proporcionadas de las situaciones iniciales y finales. Una vez que examinemos estas situaciones y entendamos qué tarea debe realizar un robot, comenzamos a planear, a poner, y a analizar una solución en ejecución.

Esta sección examina las herramientas para diseñar y escribir programas poniendo y analizando programas del robot ejecución. Combinando estas

técnicas con el nuevo mecanismo de la clase y de la instrucción, podemos desarrollar las soluciones que son fáciles de leer y de entender.

Para desarrollar y escribir los programas que solucionan los problemas del robot, debe seguirse estas tres pautas:

- Los programas deben ser fáciles de leer y de entender,
- Los programas deben ser fáciles de eliminar errores.
- Los programas deben ser fáciles de modificarse para tomar variadas formas de solución respecto de la tarea original.

4.9. TÉCNICA DE REFINAMIENTO PASO A PASO PARA EL USO DE LAS HERRAMIENTAS DE DISEÑO Y ESCRITURA DE PROGRAMAS DE KAREL

En esta sección, se describe el refinamiento paso a paso de un método que podemos utilizar para diseñar y escribir los programas del robot. Este método permite resolver el problema de una forma más simple que sean fáciles de leer y de entender, para llegar a una solución más entendible y correcta.

Es natural definir todas las nuevas clases y métodos que necesitamos para una tarea y después escribir el programa usando las instrucciones correctas.

Lo importante es definir que robot se va a utilizar, igualmente que instrucciones nuevas se necesitan para escribir el programa solución.

La técnica de refinamiento paso a paso determina escribir primero el programa usando un robot cualquiera y nombre de las instrucciones que se necesita, seguidamente definir el nuevo robot y sus instrucciones. Es decir, escribir la secuencia de mensajes en el bloque de la tarea principal para escribir las definiciones y los nuevos nombres de las instrucciones que se usan dentro de este bloque. Finalmente, se agrupan estos segmentos del programa en un programa completo.

En la figura 59 “Tarea del cosechador”, presenta una tarea de cosecha que requiere un robot para recoger un campo rectangular de pitos. El cuadro ilustra

la tarea de la cosecha, nuestro primer paso es desarrollar un plan total para dirigirnos a escribir un programa del robot que permite que Karel realice la tarea. El planeamiento es probablemente el mejor hecho como actividad del grupo.

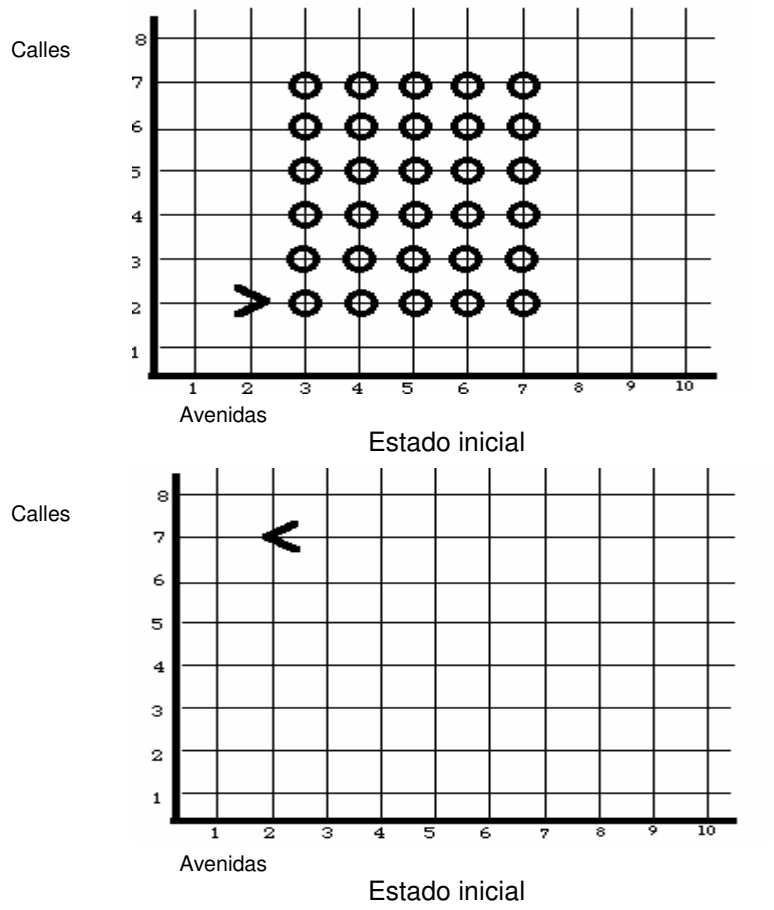


Figura 103 Estados inicial y final "Tarea del cosechador"

Compartir las ideas de solución de los problemas en grupo permite que los miembros presenten diversas formas de solución que se puedan discutir cuidadosamente para establecer verdadera solución al problema. Se pueden usar patrones de preguntas y de respuestas, tal como se relacionan a continuación:

PREGUNTA: ¿Cuántos robots se necesitan para el desarrollo de esta tarea?

RESPUESTA: Podríamos hacerla con un robot que camina hacia adelante y hacia atrás sobre todas las filas que se cosecharán, o podríamos hacerla con un equipo de robots.

PREGUNTA: ¿Cuántos robots se van a utilizar?

RESPUESTA: Vamos a intentarlo con uno y el nombre del robot es Marcos.

PREGUNTA: ¿Cómo puede Marcos recoger una fila?

RESPUESTA: Marcos podría moverse del oeste al este a través de la fila de pitos, recogiendo un pito cada vez que se mueva.

PREGUNTA: ¿Cómo puede Marcos recoger el campo entero?

RESPUESTA: Marcos podía dar vuelta alrededor y moverse de nuevo al lado occidental del campo, moviendo al norte un bloque, cara al este, y repitiendo las ordenes de la instrucción. Marcos podía hacer esto para cada fila de pitos en el campo.
Marcos no está parado en un pito, lo movemos al primer pito antes de comenzar a cosechar la primera fila. El siguiente paso es escribir el bloque de la instrucción de la tarea principal del programa usando en inglés las primitivas de los nuevos nombres del mensaje.

```
task
{       Cosechador  Marco(2, 2, East, 0);
        Marco.move();
        Marco.harvestOneRow();
        Marco.returnToStart();
        Marco.moveNorthOneBlock();
        Marco.harvestOneRow();
        Marco.returnToStart();
        Marco.moveNorthOneBlock();
        Marco.harvestOneRow();
        Marco.returnToStart;
        Marco.moveNorthOneBlock();
        Marco.harvestOneRow();
        Marco.returnToStart();
        Marco.moveNorthOneBlock();
```

```
        Marco.harvestOneRow();
        Marco.returnToStart();
        Marco.moveNorthOneBlock();
        Marco.harvestOneRow();
        Marco.returnToStart();
        Marco.turnOff();
}
```

Obsérvese como se ha diseñado una nueva clase del robot que pueda realizar tres nuevos mensajes. Se puede pensar en una clase de robot como mecanismo para crear abastecedores de servicio.
Estos robots son un ejemplo de objetos en la programación orientada a objetos, proporcionan servicios específicos cuando los mensajes enviados solicitan su servicio más allá de los servicios básicos que todos los ur_Robots() pueden proporcionar, como son servicios de: harvestOneRow(), returnToStart(), y moveNorthOneBlock().

PREGUNTA: ¿Cuáles son las fortalezas de este plan?

RESPUESTA: El plan se aprovecha del nuevo mecanismo de la instrucción y permite que Marcos coseche los pitos.

PREGUNTA: ¿Cuáles son las debilidades del plan?

RESPUESTA: Marcos hace algunos viajes "para vaciar".

PREGUNTA: ¿Cuáles son estos viajes vacíos?

RESPUESTA: Marcos vuelve al punto de partida en la fila que acaba de ser cosechada.

PREGUNTA: ¿Por qué esto es malo?

RESPUESTA: Porque el robot es un recurso valioso que se debe utilizar eficientemente.

PREGUNTA: ¿Puede Marcos recoger más pitos?

RESPUESTA: En vez de cosechar solamente una fila y entonces dar vuelta alrededor y de volver al comienzo, Marcos puede cosechar una fila, mover al norte una calle y volverse al oeste que coseche una segunda fila. Marcos entonces mueve una calle al norte para comenzar el proceso entero encima para las dos filas siguientes.

PREGUNTA: ¿Qué ventaja ofrece el primer plan?

RESPUESTA: Marcos hace solamente seis viajes a través del campo en vez de doce. No hay viajes vacíos.

PREGUNTA: ¿Cuáles son las debilidades de este nuevo plan?

RESPUESTA: Ninguna que podemos ver, ya que existe un número par de filas. Cuando planteamos soluciones del planeamiento, debemos ser muy críticos y no apenas aceptar el primer plan como el mejor. Ahora tenemos dos diversos planes y usted puede pensar probablemente en otras soluciones. Lo importante es evitar los viajes vacíos y poner el segundo plan en ejecución.

```
task
{       cosechador Marco(2, 2, East, 0);
        Marco.move();
        Marco.harvestTwoRows();
        Marco.positionForNextHarvest();
        Marco.harvestTwoRows();
        Marco.positionForNextHarvest();
        Marco.harvestTwoRows();
        Marco.move();
        Marco.turnOff();
}
```

4.10. PLANEAR CON HARVESTTWOROWS Y LA POSICIÓN DE FORNEXTHARVEST

La segunda etapa del plan contiene dos subtareas: Una cosechar dos filas y la otra posicionar a Marcos para cosechar dos filas más. El planteamiento de estas dos subtareas debe ser tan cuidadoso y exacto como debe ser para la tarea total.

4.10.1..Planear con la instrucción HarvestTwoRows

PREGUNTA: ¿Qué se hace en HarvestTwoRows?

RESPUESTA: El plan HarvestTwoRows consiste en cosechar dos filas de pitos. Una será cosechada como recorrido de Marcos al este y la segunda será cosechada como vuelta de Marco al oeste.

PREGUNTA: ¿Qué tiene que hacer Marcos?

RESPUESTA: Marcos debe recoger los pitos y moverse mientras que viaja al este. En el final de la fila de pitos, Marcos debe mover al norte un bloque, hacer frente al oeste, y volver al borde occidental de los pitos de la cosecha del campo mientras que viaja al oeste.

Modelo para un plan general con la instrucción HarvestTwoRows().

```
class Harvester extends ur_Robot
{
        void  harvestTwoRows()
        {
                harvestOneRowMovingEast();
                goNorthToNextRow();
                harvestOneRowMovingWest();
        }
        ...
}
```

Analicemos el plan HarvestTwoRows() buscando sus fortalezas y sus debilidades.

PREGUNTA: ¿Cuáles son las fortalezas del plan HarvestTwoRows()?

RESPUESTA: Solución de problemas.

PREGUNTA: ¿Cuáles son las debilidades de este plan?

RESPUESTA: Una, tener dos posibles instrucciones para cosechar una sola fila de pitos.

PREGUNTA: ¿Realmente se necesitan dos posibles instrucciones para cosechar una sola fila?

RESPUESTA: Necesitamos una instrucción para ir al este y otra para regresar al oeste.

PREGUNTA: ¿Realmente necesitamos una instrucción separada para cada dirección?

RESPUESTA: En la dirección que se mueva Marcos no importa. Si planeamos el goToNextRow cuidadosamente, podemos utilizar una instrucción de cosechar una fila de pitos cuando va Marcos al este y la misma instrucción para que vaya al oeste. Nuestro análisis nos demuestra que podemos reutilizar una sola palabra (harvestOneRow) en vez de definir dos instrucciones similares, haciendo nuestro programa más pequeño.

A continuación se muestra un modelo para el uso de la instrucción harvestTwoRows()

```
void harvestTwoRows()
{
```

```
                    // Antes de realizar la instrucción el robot debe estar orientado al este
                    // Sobre el primer pito que se encuentra en la fila
                     harvestOneRow();
                     goToNextRow();
                     harvestOneRow();
          }
```

4.10.2 Planear con la instrucción Fornextharvest()

PREGUNTA: ¿Qué hace la instrucción ForNextHarvest()?

RESPUESTA: Se utiliza esta instrucción cuando Marcos está en el lado occidental del campo del pito. Se mueve el robot al norte un bloque y hace giro al este en la posición para cosechar dos más filas de pitos.

PREGUNTA: ¿Qué tiene que hacer Marcos?

RESPUESTA: Marcos debe dar vuelta a la derecha girando al norte, se mueve un bloque y da vuelta a la derecha girando a este. Ponemos esta instrucción en ejecución como sigue.

```
class Cosechador  : ur_Robot
  {
      void positionForNextHarvest();
      void  turnRight();
  }
          void Cosechador ::positionForNextHarvest()
          {
               // Antes de realizar la instrucción el robot debe estar orientado al oeste
              //  Sobre la primera esquina de la fila
                    turnRight();
                    move();
                    turnRight();
          }

          void Cosechador :: turnRight()
          {
```

```
        turnLeft();
        turnLeft();
        turnLeft();
    }
```

4.10.3 Planear con Harvestonerow() y Gotonextrow()

Ahora centramos nuestros esfuerzos en harvestOneRow() y finalmente goToNextRow().

PREGUNTA: ¿Qué hace harvestOneRow()?

RESPUESTA: Comenzando en el primer pito y haciendo frente a la dirección correcta, Marcos debe cosechar cada uno de las esquinas que encuentra, parando en la localización del último pito en la fila.

PREGUNTA: ¿Qué tiene que hacer Marcos?

RESPUESTA: Marcos debe ejecutar una secuencia de harvestCorner() y mover instrucciones de recoger los cinco pitos en la fila.

PREGUNTA: ¿Cómo Marcos cosecha una sola esquina?

RESPUESTA: Marcos debe ejecutar una instrucción del recogerpitos. Podemos poner el harvestOneRow() y el harvestCorner() en ejecución como sigue.

```
class Cosechador : ur_Robot
 {
   void harvestOneRow();
   void harvestCorner();
 };
        ...
        void Cosechador:: harvestOneRow()
        {
```

```
        harvestCorner();
        move();
        harvestCorner();
        move();
        harvestCorner();
        move();
        harvestCorner();
        move();
        harvestCorner();
}

void Cosechador:: harvestCorner()
{
        pickBeeper();
}
```

4.10.4 Planear con la instrucción goToNextRow

PREGUNTA: ¿Qué hace goToNextRow()?

RESPUESTA: Esta instrucción mueve a Marcos hacia el norte un bloque, a la fila siguiente.

PREGUNTA: Por que no se puede utilizar la posición ForNextHarvest()?

RESPUESTA: Porque no trabajará correctamente. Cuando utilizamos la posición ForNextHarvest(), Marcos debe estar mirando al oeste. Marcos ahora está mirando al este, en está posición la instrucción ForNextHarvest() no trabaja.

PREGUNTA: ¿Qué tiene que hacer Marcos?

RESPUESTA: Marcos debe girar a la izquierda girando al norte, mover un bloque, y dar vuelta a la izquierda girando al oeste. Finalmente pone en práctica la nueva instrucción.

Se debe simular la instrucción posición ForNextHarvest() en el papel. Comience ubicando a Marcos al oeste y observe donde está el robot cuando usted acabe de simular la instrucción.

Uso de la instrucción goToNextRow()

```
class Cosechador: ur_Robot
{
void goToNextRow();
};          ...
            void Cosechador goToNextRow()
            {
                        // Antes de realizar la instrucción el robot debe estar orientado al este
                        //   Sobre la ultima esquina de la fila
                        turnLeft();
                        move();
                        turnLeft();
            }
```

Se puede utilizar la simulación para analizar esta instrucción e igualmente demostrar que es correcta y el programa está realizado.

Paso final: Verifica que el programa completo es correcto

La verificación del programa nos permite comprobar una vez más que la solución del problema es correcta.

```
class Cosechador : ur_Robot
 {
   void harvestTwoRows();
   void positionForNextHarvest();
   void turnRight();
   void harvestOneRow();
   void harvestCorner();
   void goToNextRow();
 };
            void  Cosechador::harvestTwoRows()
            {
                        // Antes de realizar la instrucción el robot debe estar orientado al este
```

```
            //  Sobre el primer pito encontrado en la fila
            harvestOneRow();
            goToNextRow();
            harvestOneRow();
}
void  Cosechador::positionForNextHarvest()
{
            // Antes de realizar la instrucción el robot debe estar orientado al oeste
            //   Sobre la ultima esquina de la fila
            turnRight();
            move();
            turnRight();
}

void Cosechador:: turnRight()
{           turnLeft();
            turnLeft();
            turnLeft();
}

void   Cosechador::harvestOneRow()
{           harvestCorner();
            move();
            harvestCorner();
            move();
            harvestCorner();
            move();
            harvestCorner();
            move();
            harvestCorner();
}

void   Cosechador::harvestCorner()
{           pickBeeper();
}
void   Cosechador::goToNextRow()
{
            // Antes de realizar la instrucción el robot debe estar orientado al este
            // Sobre la ultima esquina de la fila
            turnLeft();
```

```
            move();
            turnLeft();
        }

task
{       Cosechador Marco(2, 2, East, 0);
        Marco.move();
        Marco.harvestTwoRows();
        Marco.positionForNextHarvest();
        Marco.harvestTwoRows();
        Marco.positionForNextHarvest();
        Marco.harvestTwoRows();
        Marco.move();
        Marco.turnOff();
}
```

Se debe simular la ejecución del programa de Marcos completamente para demostrar que todos los subprogramas trabajan correctamente y para estar seguros que el programa es correcto.

4.11 LAS VENTAJAS DE USAR NUEVAS INSTRUCCIONES

Una ventaja de dividir un programa en subprogramas utilizando nuevas instrucciones, así estas instrucciones se ejecuten solamente una vez. Las nuevas instrucciones estructuran programas, y las palabras y las frases inglesas hacen programas más comprensibles.

Los programas apenas se han leído y escrito y vemos con el fin de encontrar instrucciones que estén confusas o difíciles de entender.

En el ejemplo de la cosechadora es típico para utilizar la instrucción harvestField() como servicio. La utilización de este servicio nos permite cosechar un campo entero. Podemos agregar fácilmente esta característica a la clase Cosechadora. Su uso puede esstar en todas las declaraciones del bloque de la tarea principal, exceptuando la primera y la ultima.

```
class Field_Harvester : Cosechador
{
        void harvestField()
        {       move();
                harvestTwoRows();
                positionForNextHarvest();
                harvestTwoRows();
                positionForNextHarvest();
                harvestTwoRows();
                move();
        }
}
```

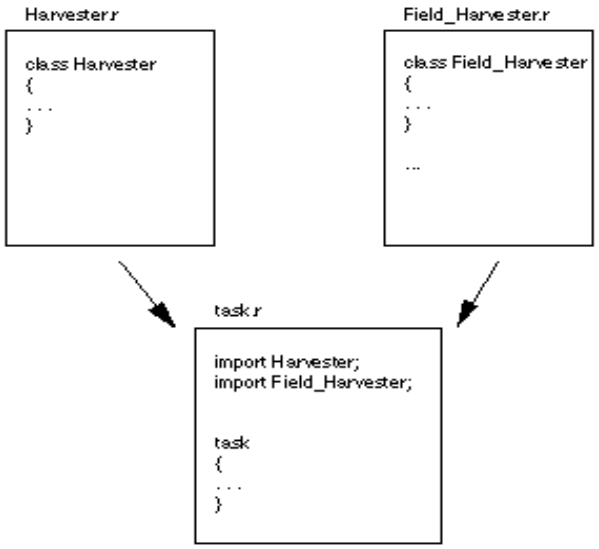


Figura. 104 Definición de la clase cosechador

Para utilizar esta nueva clase se debe incluir su definición, igualmente la definición de la clase Cosechador y el bloque de la tarea principal en un sólo archivo. La forma más eficiente para hacer esto es aprovechar la característica de la importación de la lengua del robot .

Si con antelación se conoce la definición del archivo nombrado "Field_Harvester.r" y pusimos las definiciones de la clase Cosechador() en otro diverso archivo "Harvester.r." Ninguno de los dos archivos contiene un bloque de la tarea principal.

Se puede especificar una tarea dentro de un archivo.

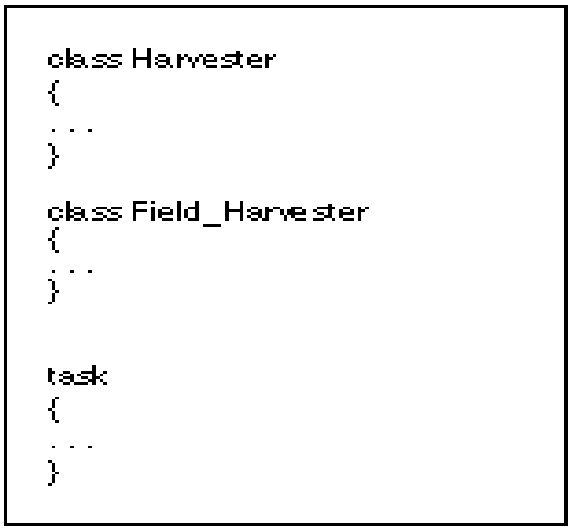


Figura 105 Definición estructural de las clases Harvester() y field_Harvester()

```
import cosechador;
import Field_Harvester;

task
{       Field_Harvester Tony (2, 2, East, 0);
        Tony.harvestField();
        Tony.turnOff();
}
```

Las primeras dos líneas dicen que la instalación incluirá el contenido entero de Harvester.r y de Field_Harvester.r en este programa en lugar de las instrucciones de la importación.

Usar la importación le dice a la instalación que lea otra descripción, tal como Harvest.r, en cierto lugar de especificación.

4.12 EVITANDO ERRORES

Muchos principiantes piensan que todo esto de planear, analizar, ejecutar y simular los programas toma demasiado tiempo. Lo que realmente toma tiempo es estar corrigiendo errores.

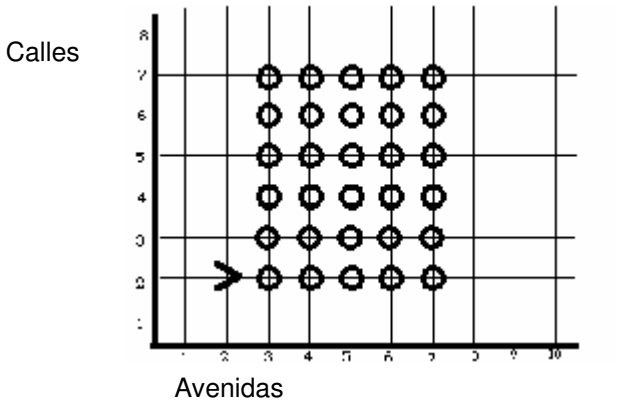
Estos errores se clasifican en dos amplias categorías: Los errores del planeamiento de ejecución y de intento, suceden cuando escribimos un programa sin un plan de bien pensado y podemos perder mucho tiempo de programación. Son generalmente difíciles de encontrar porque los segmentos grandes del programa pueden ser modificados o desechados. El planeamiento y el análisis cuidadoso del plan pueden ayudarnos a evitar errores del planeamiento. Los errores de programación léxicos y de sintaxis, suceden cuando escribimos realmente el programa. Si escribimos el programa entero sin la prueba de él, tendremos indudablemente muchos errores a corregir, algunos de los cuales pueden ser casos múltiples del mismo error.

Al escribir el programa en bloques se reduce el número total de errores introducidos en cualquier instante y pueden prevenir ocurrencias múltiples del mismo error. La técnica de refinamiento paso a paso es una herramienta que permite la planeación y el análisis, para poner los planes en ejecución de tal manera que permita conducir a un programa del robot que contenga un mínimo de errores.

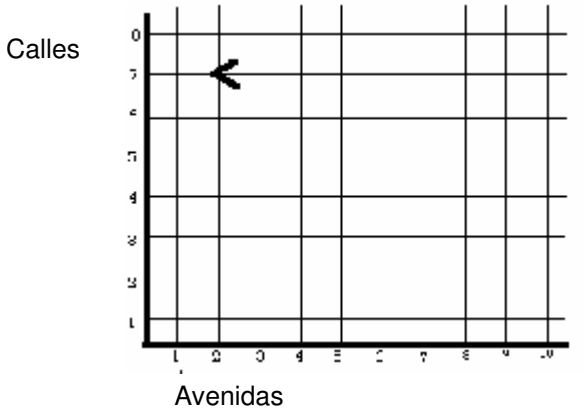
4.13 MODIFICACIONES FUTURAS

En el capítulo anterior, se dijo que es necesario escribir programas que sean fáciles de leer, de entender, de eliminar errores y de modificar.

El mundo del robot puede ser modificado fácilmente para que los programas existentes se mantengan en el robot. Puede ser mucho más simple y toma menos tiempo para modificar un programa existente, para realizar una tarea, para escribir totalmente un nuevo programa. A continuación se presentan dos situaciones que diferencian la tarea del Cosechador().



Estado inicial



Estado final

Figura 106 Estados inicial y final tarea del cosechador

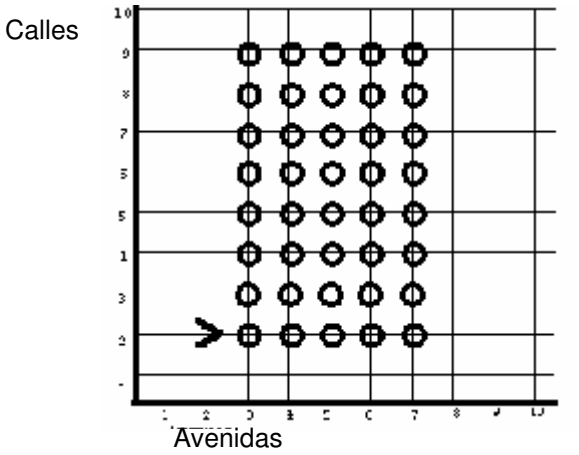


Figura. 107 Tarea del cosechador con más filas

Es difícil modificar nuestra clase del Cosechador() y el programa que la contiene. El problema es fácil de solucionar. ¿Qué pasa si agregamos dos nuevas líneas al bloque original de la tarea principal para solucionar la nueva tarea? No necesitamos ningún cambio a la clase del Cosechador() por sí mismo.

El uso de nuevas instrucciones permite encontrar rápidamente dónde se necesita realizar el cambio.

Realizamos un cambio simple al harvestOneRow como sigue,

```
void harvestOneRow()
{
        harvestCorner();
        move();
        harvestCorner();
        move();
        harvestCorner();
        move();
        harvestCorner();
        move();
        harvestCorner();
        move();                 // Adiciones estas dos
        harvestCorner();        // Nuevos mensajes
}
```

Este cambio a la clase del Cosechador() está muy bien a condición de que no necesitamos solucionar el problema original en el futuro. Es ciertamente ventajoso dejar la clase Cosechador() sin cambios y crear una nueva clase, Long_Harvester() que contenga esta instrucción modificada del harvestOneRow().

```
class Long_Harvester extends Harvester
{
        void  harvestOneRow()
        {
                super.harvestOneRow();          // Ejecute la instrucción
```

```
                move();                 // Adiciones estas dos
                harvestCorner();        // Nuevos mensajes
        }
}
```

El uso de nuevas instrucciones simplifica la tarea de encontrar y fijar errores. Esto es cierto si las instrucciones son cortas y pueden ser fácilmente entendidas.

Si el robot marca una vuelta incorrecta e intenta recoger un pito en el lugar incorrecto. Utilizando nuevas instrucciones para escribir el programa, y que cada nueva instrucción realice una tarea específica. La posición ForNextHarvest() y un sistema de controles de las tareas relacionadas con harvestTwoRows() nos permite determinar generalmente la localización probable del error.

4.14 PROGRAMAR SIN NUEVAS INSTRUCCIONES

A continuación se presenta un programa que procura solucionar el problema de los pitos, planteado inicialmente con sólo instrucciones primitivas.

Examine el programa y haga las mismas preguntas que acabamos de exponer:

¿Dónde cambiaríamos el programa para solucionar la primera situación modificada? ¿ dónde cambiaríamos el programa para solucionar la segunda situación modificada? - suponga que Marcos da una vuelta incorrecta mientras que recoge los pitos. ¿Dónde miraríamos para corregir el error?

```
task
{       ur_Robot Marco = new ur_Robot(2, 2, East, 0);
        Marco.move();
        Marco.pickBeeper();
        Marco.move();
        Marco.pickBeeper();
        Marco.move();
        Marco.pickBeeper();
```

```
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.turnLeft();
Marco.move();
Marco.turnLeft();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.turnRight();
Marco.move();
Marco.turnRight();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.turnRight();
Marco.move();
Marco.turnLeft ();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
```

```
        Marco.turnLeft();
        Marco.move();
        Marco.turnLeft();
        Marco.pickBeeper();
        Marco.move();
        Marco.pickBeeper();
        Marco.move();
        Marco.pickBeeper();
        Marco.move();
        Marco.pickBeeper();
        Marco.move();
        Marco.pickBeeper();
        Marco.move();
        Marco.turnOff();
}
```

Las listas largas de mensajes como éstas, pueden solucionar correctamente un problema pero son muy difíciles de leer y de entender, además, son muy difíciles para eliminar errores y para modificarse.

4.15 ESCRIBIR PROGRAMAS COMPRENSIBLES

En la programación de Karel, el éxito está en su correcta escritura de los programas para lograr su comprensibilidad, ya que de ésta depende la claridad para buscar la solución de los problemas. Si un programa es comprensible, es fácil eliminar sus errores. ¿Qué se necesita para que un programa sea fácil de entender?

Para que un programa sea fácil de entender se necesitan dos criterios:

Primero, que el programa este compuesto por bloques simples, fáciles y comprensibles. Que cada parte del programa sea entendible.

Segundo, que el programa este dividido en bloques pequeños y fáciles de entender. Debemos cerciorarnos de nombrar las nuevas instrucciones

correctamente. Estos nombre proporcionan una descripción (posiblemente la única descripción) de las ordenes que ejecuta la instrucción.

El lenguaje de programación del robot permite elegir cualquier nombre de la instrucción, pero con esta libertad viene la responsabilidad de seleccionar nombres exactos y descriptivos. Porque es más fácil verificar o eliminar errores de un programa que contenga nuevas instrucciones.

Las nuevas instrucciones pueden ser probadas independientemente. Al escribir un programa debemos simular y verificar cada instrucción inmediatamente hasta que sea correcta. Entonces podemos olvidarnos cómo trabaja la instrucción y sólo recordar que hace la instrucción. El recordar debe ser fácil, si nombramos la instrucción correctamente. Esto es más fácil, si la instrucción ejecuta solamente una orden.

Las nuevas instrucciones imponen una nueva estructura a nuestros programas, entonces, podemos utilizar esta estructura para encontrar la solución del problema. Para eliminar los errores de un programa, primero debemos encontrar las nuevas instrucciones que están funcionando incorrectamente.

Un fenómeno psicológico interesante, está relacionado con el mecanismo de definición de la instrucción del robot, con el razonamiento humano. El cerebro humano puede centrarse en una cantidad de información limitada en cualquier momento. La capacidad que tiene el robot de no hacer caso de los detalles que no son relevantes es una gran ayuda para programar la escritura y eliminar errores.

La mayoría de los programadores principiantes tienen la tendencia a escribir las definiciones demasiado grandes en la instrucción. Lo correcto es escribir pequeñas definiciones e ir nombrando las instrucciones, a cambio de escribir grandes definiciones.
Escribir programas comprensibles con nuevas instrucciones y usar la técnica del refinamiento paso a paso, reduce el número de errores y la cantidad de tiempo para la programación del robot de escritura.

4.16 UNA TAREA PARA DOS ROBOTS

En esta sección comencemos con una tarea para dos robots que no nos restringe a usar un solo robot. Podemos tener tantos robots como tantas tareas se quieran programar.

Los robots pueden comunicarse de maneras, como órdenes reciba en su programación. Por ejemplo está es una tarea simple para dos robots. Karel se encuentra en la calle 3ª con avenida 1ª, tiene un pito, y está orientado al este. Carlos está en los revestimientos del origen del este. Karel debe llevar el pito a Carlos y ponerlo abajo. Carlos debe entonces tomarlo y llevarlo a la calle 1ª con avenida 3ª. El pito se debe colocar en esta esquina.

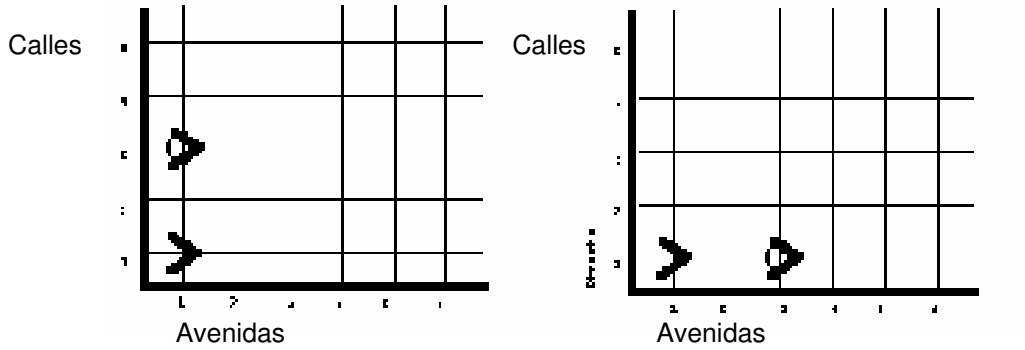


Figura. 108 Posibles estados inicial y final para una tarea con dos robots

```
task
{ur_Robot Karel ur_Robot(3, 1, East, 0);
ur_Robot Carl  ur_Robot(1, 1, East, 0);

Karel.pickBeeper();
Karel.turnLeft();
Karel.turnLeft();
Karel.turnLeft();
Karel.move();
Karel.move();
Karel.putBeeper();
```

```
Carl.pickBeeper();
Carl.move();
Carl.move();
Carl.putBeeper();
Karel.turnOff();
Carl.turnOff();
}
```

Resulta fácil solucionar problemas con la ayuda de un equipo o grupo de robots, lo importante es el uso de las nuevas instrucciones que se deben emplear, por ejemplo un mensaje tipo HarvestTwoRows() sería el apropiado para este tipo de tareas (Véase el problema del cosechador).

La solución del programa será la siguiente:

```
import Harvester;

task
{       Harvester Karel = new Harvester(2, 2, East, 0);
        Harvester Kristin = new Harvester(4, 2, East, 0);
        Harvester Matt = new Harvester(6, 2, East, 0);
        Karel.move();
        Karel.harvestTwoRows();
        Karel.turnOff();
        Kristin.move();
        Kristin.harvestTwoRows();
        Kristin.turnOff();
        Matt.move();
        Matt.harvestTwoRows();
        Matt.turnOff();
}
```

El problema del cosechador también se puede solucionar con seis robots, una manera aún más interesante de hacer esto, es dejar a un robot que coordine las acciones de los otros. Para que este proceso funcione, necesitamos dos clases de robot diferentes. Una clase de robots será llamada un Coreógrafo, porque dirige los otros, que son los robots corrientes y los otros robots son los

estándares de la edición ur_Robot(). El truco aquí es que el Coreógrafo instalará a los otros robots y después garantizando que atenderán las acciones del Coreógrafo.

El Coreógrafo necesita saber los nombres de los otros dos robots, así que damos nombres a estos robots privados del Coreógrafo. Pero como no se ha visto esta característica en el lenguaje de programación del robot. Entonces declaramos el robot momentos antes del bloque de la tarea principal, es permitido definir nombres del robot dentro de una nueva clase. Los robots declarados están disponibles, los ayudantes de la clase de robot ya están declarados, pero no se pueden utilizar por otros robots o en el bloque de la tarea principal.

Esto es, porque los nombres del ayudante del robot serán privados en la nueva clase de robot. El Coreógrafo también necesitará eliminar todos los métodos del robot. Si ordenamos al Coreógrafo que se mueva, éste puede ordenar a los otros robots que se muevan también.

```
class Choreographer : ur_Robot
{
        ur_Robot Lisa(4,2,East,0);      // El primer robot ayudante
        ur_Robot Tony(6,2,East,0);      // El Segundo robot ayudante
        void harvest();
        void harvestARow();
        void harvestCorner();
        void move();
        void pickBeeper();
        void turnLeft();
        void turnOff();
}
```

Aquí es el bloque de tarea principal de nuestro programa.

```
task
{       Choreographer Karel(2, 2, East, 0);
        Karel.harvest();
```

```
        Karel.turnOff();
}
```

Aquí está la clase completa del Coreógrafo, con algunas anotaciones.

```
class Choreographer extends ur_Robot
{
        ur_Robot Lisa = new ur_Robot(4,2,East,0);          // El primer robot ayudante
        ur_Robot Tony = new ur_Robot(6,2,East,0);          // El Segundo robot ayudante
```

Harvest y harvestARow son similares como se ha visto anteriormente.

```
        void harvest()
        {
                harvestARow();
                turnLeft();
                move();
                turnLeft();
                harvestARow();
        }

        void harvestARow()
        {
                move();
                harvestCorner();
                move();
                harvestCorner();
                move();
                harvestCorner();
                move();
                harvestCorner();
                move();
                harvestCorner();
                move();
        }

        void harvestCorner()
        {
                pickBeeper();
```

```
}
```

La clave está en definir un robot Coreógrafo y el grupo de robots que trabará con él en la redefinición de métodos heredados. A manera de ejemplo se muestra un caso de movimiento único:

```
void  move()
{           super.move();
            Lisa.move();
            Tony.move();
}
```

En cada uno de los otros métodos, primero se ejecuta la instrucción heredada del mismo nombre y enseguida se envía el mensaje a los dos robot del ayudante.

```
        ... // similar al método move()
}
```

Cuando ordenamos que se mueva el robot del Coreógrafo (aquí Karel) primero ejecuta la instrucción heredada del movimiento y enseguida envía mensajes de movimiento a los dos robots ayudantes, que pertenecen a la clase ur_Robots(), sin afectar a ningún otro robot. Esto significa que siempre que Karel se mueva, cada uno de sus ayudantes también lo hará "automáticamente." Igualmente para las instrucciones pickBeeper() y turnleft(). Cuando un robot envía un mensaje a otro robot, el mensaje pasa del remitente a través del satélite al otro robot. El robot remitente debe entonces esperar que termine la instrucción enviada antes de reasumir su propia ejecución.

- **Diseño orientado a objetos.**

Anteriormente se aprendió una técnica útil para diseñar una sola clase, preguntando qué tareas son necesarias para esta y de diseñar las tareas complejas en tareas más simples. Ahora vamos a mirar este diseño desde una perspectiva más amplia. Se debe prever que podemos necesitar varias clases

de robot para realizar cierta tarea y estos robots pueden cooperar de cierta manera. Aquí se discute solamente las ediciones del diseño

En el mundo real, la construcción de casa es una tarea moderadamente compleja, ya que, es hecha generalmente por un equipo de constructores, cada uno con su propia especialidad. Quizás sea igual en el mundo del robot.

Si se mira algunos de nuestros ejemplos anteriores, se encontrará que hay realmente dos clases de instrucciones. La primera clase de instrucción, es el harvestTwoRows() en la clase Cosechadora que significa ser utilizada en el bloque de la tarea principal, la segunda clase es goToNextRow(), que tiene significado para ser utilizada internamente como parte de la descomposición del problema. Por ejemplo, la instrucción goToNextRow() es poco probable para ser utilizada excepto dentro de otras instrucciones.

La primera clase de instrucción tiene significado para ser pública y define de una cierta manera qué se piensa hacer con el robot. Si pensamos en un robot como servidor, entonces su cliente (el bloque de la tarea principal, o quizás otro robot) le enviará un mensaje con una de sus instrucciones "públicas". El cliente es quién solicita un servicio de un servidor. El robot Cosechador proporciona un servicio de cosecha. Su cliente solicita ese servicio. El lugar para comenzar el diseño de las instrucciones está con los servicios públicos y los servidores que los proporcionan. El robot propiamente servidor, ejecutará otras instrucciones para proporcionar el servicio. Podemos utilizar el refinamiento sucesivo para ayudar a diseñar las otras instrucciones que ayudan al servidor a realizar su servicio.

La manera más fácil de construir una casa es contratar los servicios de un especialista, que montará un equipo apropiado para construir nuestra casa.

Consideremos como robot contratista para construir la casa, el trabajo se considera de alguna manera hecho. El cliente no se preocupa cómo el contratista realiza la construcción mientras el resultado sea aceptable.

Note que nosotros hemos dado el diseño preliminar para una clase, la clase del contratista, con un método público: buildHouse(). Con esta metodología se ve la necesidad de descubrir nuevos métodos y otras clases de instrucciones.

También, sabemos que necesitaremos probablemente un solo robot de la clase contratista. Vamos a nombrarlo Kristin, para tener un nombre de referencia. La idea es examinar la tarea desde el punto de vista de Kristin. ¿Qué necesita hacer Kristin para construir una casa?. Una posibilidad de colocar todos los pitos apropiadamente, la otra forma es utilizar robots especialistas para construir cada una de las partes de la casa, por ejemplo: las paredes, la azotea, las puertas y las ventanas. Si quisiéramos que las paredes fueran hechas de ladrillo, debemos contar con los servicios de uno o más robots especializados para tal fin.

Las puertas y las ventanas se podían construir por un par de robots de carpinteria, y la azotea construirla por un robot techador. El contratista, Kristin necesita reunir a este equipo. Nosotros necesitamos otro método para esto, el equipo podría ser llamado por el cliente o comenzar la construcción. Kristin también necesita poder conseguir el equipo para empezar la obra.

Necesitamos un nuevo método gatherTeam() del contratista. ocupándose, de los trabajos más pequeños. El robot albañil debe responder a un mensaje de buildWall(). El contratista debe indicar al albañil donde deben ser construidas las paredes. De igual forma, el robot techador debe responder al mensaje makeRoof(), y el robot carpintero debe conocer los mensajes de makeDoor() y de makeWindow(). Puede ser que vayamos un paso más lejos con el techador y decidamos que sería provechoso hacer los dos aguilones de la azotea por separado. También quisiéramos que un techador pudiera hacer makeLeftGable() y a makeRightGable().

```
class Mason : ur_Robot
 {
   void buildWall();
 }
```

```
            void Mason :: buildWall()
            {

            }

class Carpintero : ur_Robot
  {
     void makeWindow();
     void makeDoor();
  }
            void Carpintero:: makeWindow()
            {

            }

            void Carpintero::makeDoor()
            {

            }


class Roofer : ur_Robot
 {
     void makeRoof();
     void makeLeftGable();
     void makeRightGable();
 }
            void  Roofer::makeRoof()
             {
                        ...
             }
            void Roofer:: makeLeftGable()
            {
                        ...
            }
            void Roofer:: makeRightGable()
            {
            }
```

El equipo de constructores está definido por el contratista, quien conoce sus nombres para evitar equivocaciones en las órdenes y sea él quien diga a los constructores que tiene que hacer cada uno y no el cliente decir que se tiene que hacer.

En la clase del contratista se debe declarar a estos robots con nombres privados y no con nombres globales, ya que, esto podría crear confusión.

Una declaración de la clase contratista sería la siguiente:

```
class Contratista :  ur_Robot
{
        Mason Ken_the_Mason = new Mason(1,1,E,??);
        Roofer  Sue_the_Roofer = new Roofer(...);
        Carpenter Linda_the_Carpenter = new Carpenter(...);
        Carpenter Steve_the_Carpenter = new Carpenter(...);

        void gatherTeam()// Call prior to first move.
        {
                ...// mensajes aquí en la posicion inicial del grupo.
        }

        void buildHouse()
        {
                ...// mensajes aquí para los 4 trabajadores..

        }
}

task
{       Contractor Kristin = new Contractor(1, 1, East, 0);
        ...
        Kristin.buildHouse();
        ...
        Kristin.turnOff();
}
```

Observe que el contratista es un servidor. Su cliente es el bloque de la tarea principal. Pero note también, que Kristin es un cliente de los cuatro ayudantes puesto que proporcionan los servicios (servicios de la pared a Kristin).

Este hecho es relativamente común en el mundo verdadero para que el cliente y el contratista estén de acuerdo

4.17 PROBLEMAS PROPUESTOS

Los problemas propuestos en esta sección, están definidos con nuevos métodos para Karel. Estos métodos están basados en las técnicas de la programación orientada a objetos, donde es necesario cumplir las nuevas técnicas de solución, como es la de refinamiento paso a paso para la definición de nuevas instrucciones. Además, se debe definir las secuencias dentro de una nueva instrucción.

Detectar y corregir los errores de sintaxis en los programas garantiza la solución correcta de los problemas. Simule la ejecución de karel en cada programa para verificar que la solución es correcta. Las tareas complejas se pueden solucionar con los programas que se dividen en pequeños problemas con instrucciones potentes.

1 Escriba las definiciones apropiadas para los nuevos métodos:

- MoveMile, recordando que las millas son 8 bloques de largo.
- El move_backward, que mueve Karel un bloque al revés, pero lo deja que hace frente a la misma dirección.
- Move_kilo_mile, que mueve Karel 1000 millas adelante.

2 Karel trabaja a veces como perno-pin-setter en un callejón del bowling. Escriba un programa que mande a Karel para transformar la situación inicial en el cuadro 3-5 en la situación final. Karel comienza esta tarea con diez pitos en su beeper-bolso. Cuadro Tarea Perno-Que fija

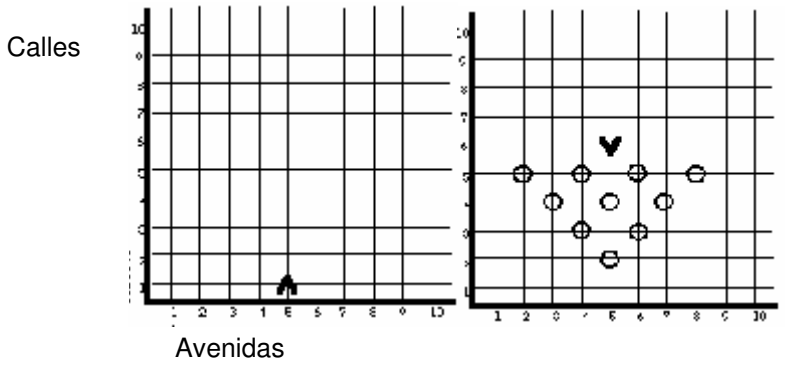


Figura. 109 Estados inicial y final del ejercicio 2

4.17.3 Escriba el programa del cosechador usando la técnica de refinamiento paso a paso.

4.17.4 La figura 110 ilustra un campo de pitos que Karel plantó una noche después de un juego del béisbol. Escriba un programa que coseche todos los pitos.

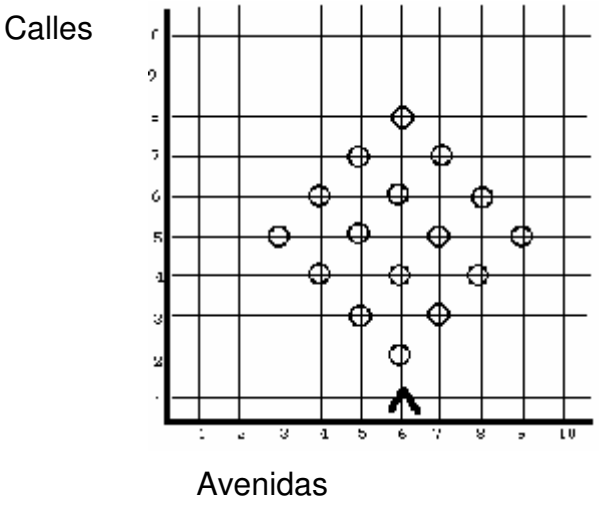


Figura. 110 Juego de Béisbol

TABLA DE CONTENIDO

Pág.

LISTA DE FIGURAS

Pág.