1.1 **INSTRUCCIONES CONDICIONALES**

En la Programación Orientada a Objetos con fundamento en Karel, inicialmente la posición exacta de un robot era conocida en el comienzo de una tarea específica. Hoy cuando se escriben programas, en esta información no se considera las posibilidades de como Karel encuentra los pitos y evita los choques con los muros.

Sin embargo, estos programas trabajan solamente en situaciones iniciales específicas. Si un robot intentara ejecutar uno de estos programas en una situación inicial con muros, el robot realizaría un cierre del error. El robot necesita la capacidad de examinar su ambiente local y después decidir con base en la información qué hacer.

Existen dos versiones de la declaración if y el if/else, éstas determinan en el robot su capacidad de decisión. Ambas versiones permiten que un robot pruebe su ambiente y, dependiendo del resultado de la prueba, decida qué instrucciones debe ejecutar.

La instrucción if{..} permite escribir programas más generales para robots que logran la misma tarea, en situaciones iguales o diferentes.
Los programas del robot contienen varias clases de instrucciones. La primera y más importante, es el mensaje a un robot. Estos mensajes son enviados al robot, por el piloto (cuando aparecen en el bloque de la tarea principal) o por otro robot (cuando ocurren en un método de una cierta clase).

La acción asociada a esta clase de instrucción es definida por la clase del robot a la cual se dirige el mensaje. Otra clase de instrucción es la especificación de la entrega, que se envía a la fábrica para construir un robot nuevo y para hacer que el piloto del helicóptero la entregue.

A continuación se presentan las dos posibilidades de estado inicial y estado final, figura 1.

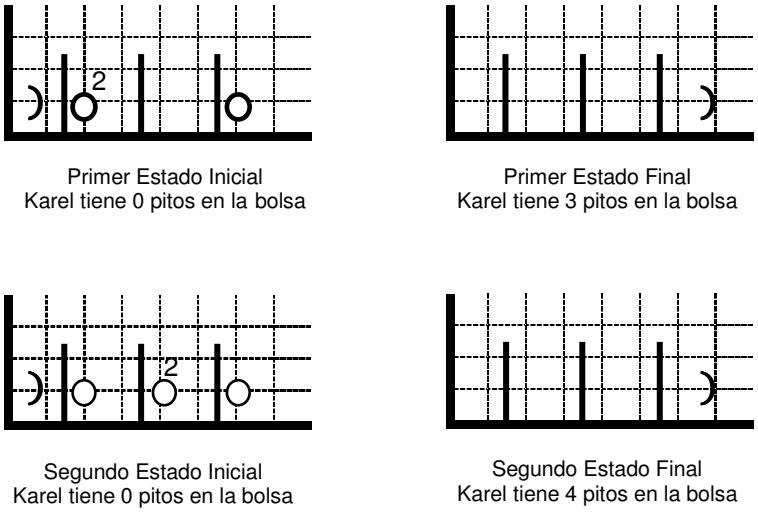


Figura 1. Estados Inicial y final de Karel

En este caso, Karel tiene la capacidad de reconocer su entorno y decidir con base en la información obtenida, qué acción tomar.

Las instrucciones que permiten ejecutar una acción u otra, dependen de las condiciones del entorno en un momento dado, éstas se llaman instrucciones condicionales.

El lenguaje de Karel ofrece dos instrucciones condicionales if/then e if/then/else, las que se presentan a continuación.

1.2 INSTRUCCIÓN if

Está instrucción es de la forma:

```
If <condición >
{
   < Instrucciones >
}
```

Karel evalúa la <condición> utilizando sus capacidades de percepción y determina si es verdadera o es falsa. Si la <condición> es verdadera, entonces Karel ejecuta las <instrucciones> que están dentro del bloque {…}.

Si la <condición> no se cumple, es falsa, entonces Karel no ejecuta las instrucciones dentro del bloque {…}.

Si sólo hay una instrucción dentro del if, se tiene que dejar los corchetes { }. Su uso se puede observar en la figura 2:

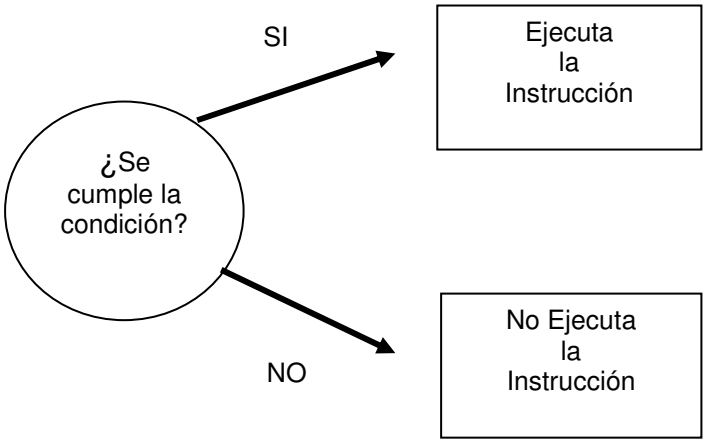
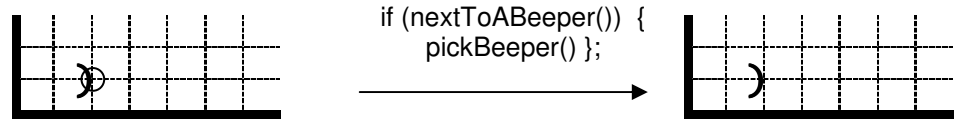


Figura 2. Estructura de la instrucción condicional simple

Toda instrucción **if{..}** sirve para indicar a Karel cómo cambiar el estado del mundo en el que se encuentra. A diferencia de las instrucciones primitivas, la instrucción condicional, le indica a Karel que debe revisar su mundo antes de realizar una acción. Así, el estado final al que deba llegar Karel depende del estado inicial y de la condición planteada en el **if**{..}

- Si la condición es verdadera en el estado inicial, el estado final es el resultado de realizar las instrucciones de la condición {…}, a partir del estado inicial.
- Si la condición es falsa en el estado inicial, el estado final es el mismo estado inicial.

Se tiene la instrucción que le indica a Karel que si hay pitos en su esquina debe recoger uno, esta instrucción se ejecuta de la siguiente manera, dependiendo del estado inicial. Cuando Karel oye un pito, concluye que la condición planteada es verdadera y realiza la instrucción dentro del {…}, cogiendo un pito. El estado final tiene un pito menos en la esquina y uno más en la bolsa de Karel.

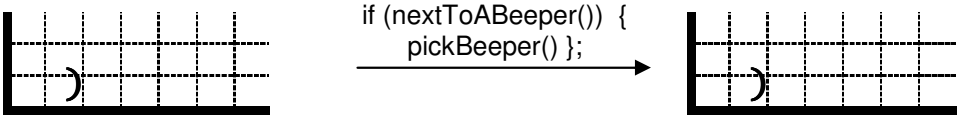


Estado inicial
Karel tiene 0 pitos en la bolsa

Estado final
Karel tiene 1 pito en la bolsa

Figura 3. Estado inicial y final cuando la condición es verdadera

En el estado inicial Karel no oye ningún pito en su esquina, la condición es entonces falsa y el estado es igual al inicial porque Karel no realiza ninguna acción.



Estado inicial
Karel tiene 0 pitos en la bolsa

Estado final
Karel tiene 0 pitos en la bolsa

Figura 4. Estado inicial y final cuando la condición es falsa

1.3 . CONDICIONES

Una condición es una característica del mundo que Karel puede evaluar para saber si es cierta o falsa en un momento dado.

Las capacidades que tiene Karel, le permiten evaluar las condiciones, cuando se refiere a los pitos del mundo y de la bolsa, orientación y la presencia de los muros al frente o a los lados.

1.3.1 Orientación: Con su brújula, Karel puede determinar su orientación y evaluar las situaciones de orientación así:

boolean facingNorth(){...}	(está mirando al norte)
boolean facingSouth(){...}	(está mirando al sur)
boolean facingEast(){...}	(está mirando al este)
boolean facingWest(){...}	(está mirando al oeste)

1.3.2 Presencia de Muros: Con las tres cámaras de visión que tiene, Karel puede saber si hay o no muros al frente, a su derecha o a su izquierda.

boolean frontIsClear(){...}	(el frente está despejado)

1.3.3 Reconocimiento de pitos: Karel puede oír si hay o no pitos en su esquina y con su brazo mecánico puede recoger y revisar si tiene pitos en su bolsa:

boolean nextToABeeper(){...}	(hay pitos en su esquina)
boolean nextToaRobot(){...}	(hay robots en la esquina)
boolean anyBeepersinBeeperBag(){...}	(hay algún pito en la bolsa)

1.3.4 Escribiendo Nuevos Predicados: Los ocho predicados definidos anteriormente están construidos dentro del lenguaje de Karel. Los predicados toman valores boléanos; como son: **true** (verdadero) y **false** (falso). Por consiguiente, se pueden definir nuevos predicados dentro de la condición. Para esto es necesario una nueva instrucción donde se definen los predicados de la instrucción return.

La forma de la instrucción return es la palabra reservada **return**, seguida de una expresión. En el método booleano el valor de la expresión tiene que ser true (verdadera) o false (falsa).

La instrucción return sólo se usa en los predicados. Ella no puede ser usada en métodos (void) ordinarios, tampoco en la condición de la tarea principal. Por ejemplo: un robot de clase inspector puede realizar las siguientes instrucciones:

```
class robot_inspector:  Robot
{           boolean frontIsBlocked();
};
boolean robot_inspector ::frontIsBlocked()
            {           return ! frontIsClear();
}
```

Cuando un robot inspector es interrogado: ¿si el frente está bloqueado?, el robot ejecuta la instrucción de return. Para hacer esto, el robot primero evalúa el predicado frontIsBlocked, éste recibe como respuesta un true (verdadero) o un false (falso).

Si el frontIsClear toma el valor **false**, y si éste es negado, entonces frontIsBlocked toma el valor **true**. De igual forma se puede escribir !nextToABeeper (), como se muestra en el siguiente predicado:

```
boolean not_nextToABeeper()
{           return (!nextToABeeper())
;
}
```

También, se puede ampliar la visión del robot, provisto de una prueba de rightIsClear(). Está instrucción es mucho más compleja porque los robots no tienen sensores por su derecha. Una solución es orientarlo hacia la derecha, así el sensor puede ser utilizado. Sin embargo, no se puede sacar al robot mirando hacia una nueva dirección. Por consiguiente, se debe estar seguro de orientarlo a la dirección original antes del retorno.

En consecuencia, rightIsClear() debe ejecutar una instrucción de giro, además de tomar un valor true o false.

```
boolean rightIsClear()
{           turnRight();
            if ( frontIsClear() )
            {           turnLeft();
                        return true;
            }

              turnLeft();
              return false;

}
```

Por lo tanto, si frontIsClear() es verdadero entonces el robot gira a la izquierda y toma el valor de verdadero, de esta forma se termina la función boolean rightIsClear().

.

Si no se cumple o se ejecuta la segunda instrucción turnLeft(), se toma el valor de falso, si la prueba de frontIsClear() toma el valor de falso, entonces el robot salta el bloque {…}, y ejecuta la segunda instrucción turnLeft() y toma el valor de falso.

En consecuencia, el programador que usa rightIsClear() puede ignorar el hecho de que el robot ejecute giro para evaluar este predicado, porque cualquier giro es cancelado. Se puede decir que el giro es "transparente" para el usuario.

El método rightIsClear(), reserva la orden de los dos últimos mensajes en el cuerpo, la instrucción de return será ejecutada antes de turnLeft() (sólo cuando frontIsClear() es falso). Aún cuando la instrucción return termina el predicado, nunca se ejecutará el turnLeft(). Esto es un error, porque no se sale con el robot mirando en la dirección original como se pretende.

1.4 . INSTRUCCIÓN if/else

Se utiliza la instrucción if/else cuando queremos que Karel ejecute una instrucción (o bloque de instrucciones), si se cumple la condición se ejecuta las instrucciones del bloque-1 y sino se ejecutará las instrucciones del bloque-2.

Las condiciones que se utilizan en esta instrucción condicional son las mismas que se usan en el if. La estructura de está instrucción es:

```
If <condición>
{
   <bloque-1>
}
  else
{
    <bloque-2>
}
```

Karel evalúa la <condición> y determina si en ese momento es cierta. Si se cumple entonces Karel ejecuta la <instrucción1> (el bloque de instrucciones del {…}); si la <condición> no se cumple, Karel ejecuta la <instrucción2> (el bloque del else {…}). Si es una sola instrucción, se puede suprimir el {…}.

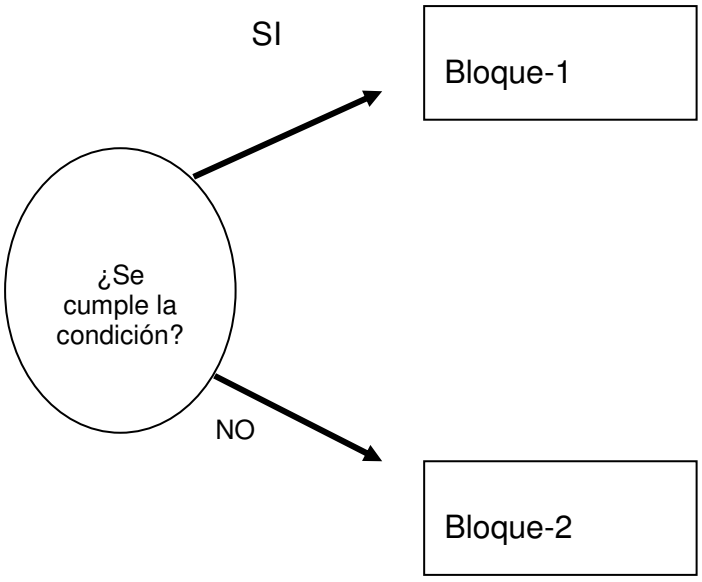


Figura 5. Estructura de la instrucción condicional compuesta

1.5 . REGANDO PITOS

Problema: En cada una de las esquinas del mundo de Karel, sobre la calle 1 entre las avenidas 1 y 6 hay 0, 1, 2 ó 3 pitos. Programe a Karel para que, partiendo del origen, mirando al este, sin pitos en la bolsa, riegue un pito en cada una de estás esquinas sobre la avenida en la que se encuentra. Karel debe terminar en la esquina (1,7) mirando al este.

Especificación: Se da una idea del problema estableciendo uno de los posibles estados iniciales y el correspondiente estado final.

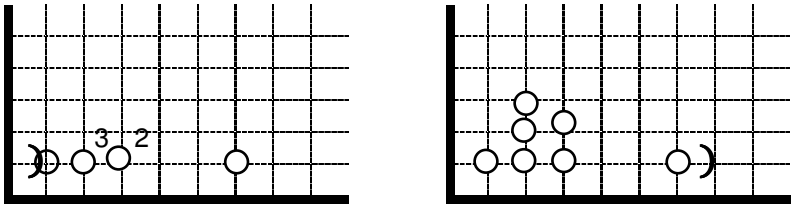


Figura 6. Estado Inicial y Final del Programa Distribuidor

Solución: La idea es repetir seis veces el paso de recoger los pitos de una esquina, enviarlos hacia arriba y pasar a la esquina siguiente. De está forma el bloque principal de ejecución será:

```
task
{
ur_Robot karel(1,1,East,0); // Inicializar el robot en su posición inicial, orientación y el número
                            // de pitos en su bolsa.
loop(6)
{
   karel.paso;
   karel.turnOff();
```

```
}
}
```

Para desarrollar la instrucción paso, descrita anteriormente, se divide la tarea en tres subproblemas de la forma ilustrada con el siguiente posible estado:

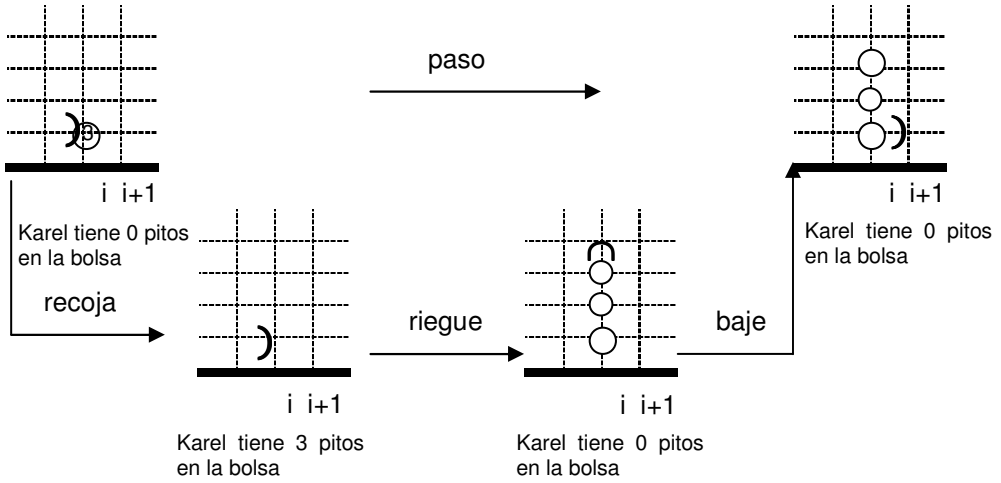


Figura 7. Ilustración de un Estado Invariante del Programa Distribuidor

Solución a la instrucción paso:

```
void  Distribuidor :: paso()
{
   recoja();
   riegue();
   baje();
}
```

Solución de las instrucciones recoja, riegue y baje:

```
void distribuidor:: recoja()
{
```

```
  if (nextToABeeper())
 {
              pickBeeper();
                 if (nextToABeeper())
                 {
                   pickBeeper();
                    if (nextToABeeper())
                   {
                            pickBeeper();
                    }
                 }
 }

void  Distribuidor :: riegue()
{
  turnLeft();
    loop(3)
    {
     ponga_y_siga();
    }

}
void  Distribuidor :: ponga_y_siga()
{
      if (anyBeepersInBeeperBag())
       {
           putBeeper();
           move();
        }
}

void  Distribuidor :: baje()
{
     turnRight();
     move();
     turnRight();
      if (frontIsClear())
       {
           move();
           if (frontIsClear())
```

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

Programa Completo

```
class distribuidor : robot
{
void turnRight();
void baje();
void ponga_y_siga();
void riegue();
void recoja();
void paso();
};
void distribuidor:: turnRight()
{
  loop(3){
  turnLeft(); }
}
void Distribuidor:: recoja()
{
  if (nextToABeeper())
 {
          pickBeeper();
             if (nextToABeeper())
             {
                pickBeeper();
                 if (nextToABeeper())
                 {
                        pickBeeper();
                 }
```

```
                }
 }
void  Distribuidor :: ponga_y_siga()
{
       if (anyBeepersInBeeperBag())
       {
            putBeeper();
            move();
        }
}
void  Distribuidor :: riegue()
{
  turnLeft();
    loop(3)
    {
     coloque_y_siga();
    }

}
void  Distribuidor :: baje()
{
    turnRight();
    move();
    turnRight();
      if (frontIsClear())
       {
            move();
            if (frontIsClear())
           {
              move();
               if (frontIsClear())
                {
                    move();
                }
           }
       }
}
void  Distribuidor :: paso()
{
  recoja();
```

```
  riegue();
  baje();
}
task
{
Distribuidor karel(1,1,East,0);
    loop(6)
  {
      karel.paso;
      karel.turnOff();
  }
}
```

1.6 . EQUIVALENCIAS DE CÓDIGO

Dos programas son equivalentes si al ejecutarse en el mismo estado inicial del mundo, conducen a estados finales idénticos. Por ejemplo, para resolver el siguiente problema se puede utilizar cualquiera de los dos programas porque son equivalentes.

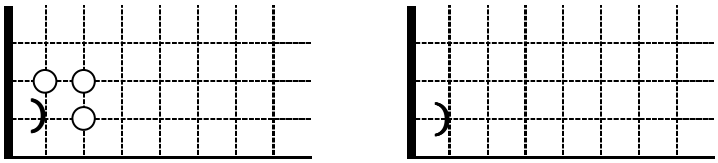


Figura 8. Estado inicial y final de un robot recogiendo pitos adyacentes a su posición

```
task
{
   ur_Robot karel(1,1,East,0);
   loop(3)
   {
     karel.move();
     karel.pickBeeper();
     karel.turnLeft();
    }
    karel.move();
    karel.turnLeft();
    karel.turnOff();
}

task
{
   ur_Robot karel(1,1,East,0);
    karel.turnLeft();
 loop(3)
   {
     karel.move();
     karel.pickBeeper();
     karel.turnRight();
    }
    karel.move();
    kael.turnLeft();
    karel.turnLeft();
    karel.turnOff();
}
```

1.7 . SIMPLIFICACIÓN DE CÓDIGOS

Existen cuatro transformaciones que van a permitir simplificar programas que contienen instrucciones if. Esto nos da la posibilidad de obtener programas, equivalentes al inicial, con una estructura más simple de entender. Para cada una de ellas mostraremos la estructura general de la transformación y un ejemplo.

1.7.1 Condición Inversa: Por comodidad, es conveniente poder intercambiar las instrucciones asociadas con la condición verdadera y con la condición falsa. Para esto basta con negar la condición del if.

Forma general:

```
If <condicion>
{
    <instruccion1>
}
 else
    {
      <instruccion2>
    }
  if <no condición>
  {
        <instruccion2>
  }
     else
        {
          <instruccion1>

        }
```

Ejemplo:

```
If (nextToABeeper())
{
  pickBeeper();
 }
  else
   { move();
   }
     if (!nextToABeeper()) {
  move();
 }
else
  {
   pickBeeper();
  }
```

1.7.2 Factorización hacia abajo: Si la última instrucción de la condición verdadera es igual a la última instrucción de la condición falsa, es posible factorizarla colocándola al final de todo el IF. Esto también es cierto para segmentos de programa.

Forma general:

Caso A

```
If <condición>
{
   <instrucción 1>
   <instrucción 3>
}
else
    {
     <instrucción 2>
     <instrucción 3>
   }
```

Caso B

```
if <condición>
{
  <instrucción 1>
}
 else
 {
   <instrucción 2>
   <instrucción 3>
  }
```

Ejemplo:

Caso A

```
   If (nextToABeeper())
   {
     pickBeeper();
     move();
    }
     else
         {
           putBeeper();
           move();
         }
```

Caso B

```
if (nextToABeeper())
{
   pickBeeper();
 }
else
   {
      putBeeper();
   }
move();
```

1.7.3 Factorización hacia arriba: Si la primera instrucción de la condición verdadera es igual a la primera instrucción de la condición falsa, es posible factorizarla colocándola antes de todo el if, sólo si esto no afecta la evaluación de la condición.

Forma general:

Caso A

```
If <condición>
{
   <instrucción 1>
   <instrucción 2>
}
else
    {
     <instrucción 1>
     <instrucción 3>
   }
```

Caso B

```
   <instrucción 1>
if <condición>
{
  <instrucción 2>
}
 else
 {
   <instrucción 3>
  }
```

Ejemplo (aplicación incorrecta de la factorización hacia arriba)

Caso A

```
If (nextToABeeper())
{
   move();
   turnLeft();
}
   else
       {
          move();
           turnRigth();
        }
```

Caso B

```
          move();
 if (nextToABeeper())
  {
    turnLeft();
   }
  else
  {
    turnRigth();
  }
```

Ejemplo (aplicación correcta de la factorización hacia arriba):

Caso A

```
if (anyBeepersInBeeperBag())
{
     move();
     turnLeft();
 }
   else
```

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

Caso B

```
move();
if (anyBeepersInBeeperBag() )
{
  turnLeft();
}
else
{
 turnRight();
}
```

1.7.4 Factorización de condiciones redundantes: Si en algún punto del programa se pregunta por una condición que ya fue establecida con anterioridad, puede suprimirse.

Ejemplo:

Caso A

```
If (facingNorth())
{
   move();
   if (facingNorth())
   {
       turnLeft();
    }
}
```

Caso B

```
if (facingNorth())
```

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

1.8 EJERCICIOS PROPUESTOS

1.8.1 En las siguientes parejas de segmentos de programa determine cuales son equivalentes.

Segmento A

```
If (nexttoABeeper())
{
        turnLeft();
 }
  else
 {
     if (!nextToABeeper())
     {
        putBeeper();
        turnRight();
      }
    else
       {
          turnRight();
       }
  }
```

Segmento B

```
If (nextToABeeper())
{
        turnLeft();
 }
else
{
  turnRight();
```

```
}
```

1.8.2 Determine cuáles de las siguientes parejas de segmentos de programa son equivalentes.

Segmento A

```
if (anyBeepersInBeeperBag() )
{
  pickBeeper();
  turnRight();
}
else
{
  if (anyBeepersInBeeperBag() )
  {
     pickBeeper();
     turnRight();
  }
    else
        {
            turnLeft();
        }
}
```

Segmento B

```
if (!anyBeepersInBeeperBag() )
{
  turnLeft();
}
else
{
  pickBeeper();
  turnRight();
}
```

1.8.3 Determine si las siguientes parejas de código son o no equivalentes:

Segmento A

```
void saltador::baje()
{
turnRight();
move();
}
if ( frontIsClear() )
{
  move();
     if ( frontIsClear() )
     {
           move();
     }
     turnLeft();
}
```

Segmento B

```
Void saltador::baje()
{
  if ( frontIsClear() )
  {
     turnRight();
     move();
     move();
       if ( frontIsClear() )
       {
          move();
          turnLeft();
        }
       else
           {
             turnRight();
             move();
             turnLeft();
           }
  }
}
```

1.8.4 Determine si las siguientes parejas de segmentos de programa son equivalentes.

Segmento A

```
If (!nextToABeeper())
{
     If (anyBeepersInBeeperBag() )
      {
         putBeeper();
      }
      else
           {
              pickBeeper();
            }
 }
```

Segmento B

```
If (nextToABeeper())
{
    pickBeeper();
 }
else
{
     If (anyBeepersInBeeperBag() )
      {
         putBeeper();
      }
  }
```

1.8.5 Determine si las siguientes parejas de código son o no equivalentes:

Segmento A

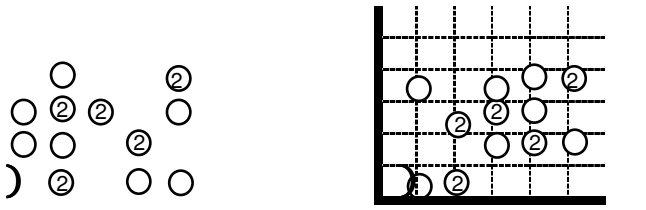
```
void saltador:: baje()
{
  move();
 if  ( frontIsClear() )
{
   move();
}
         if ( frontIsClear() )
          {
              move();
          }

     turnLeft();
}
```

Segmento B

```
void saltador:: baje()
{
   if  ( frontIsClear() )
   {
        turnRight();
        move();
        move();
         if  (frontIsClear() )
         {
             move();
             turnLeft();
         }
   }
   else
   {
      turnRight();
      move();
      turnLeft();
   }
}
```

1.8.6 En la granja de Karel existe un pequeño bosque (de 5 por 4) de pinos, contra el extremo sur-oeste del mundo. En él hay pinos de 1 metro de alto (1 pito), de 2 metros (2 pitos) y hay puntos (esquinas) sin pinos (0 pitos). Karel debe talar los pinos de 1 metro y debe sembrar pinos de 1 metro donde no hay. Donde encuentra pinos de 2 metros no realiza ninguna acción. Karel parte del origen mirando al este, con suficientes pitos en la bolsa.

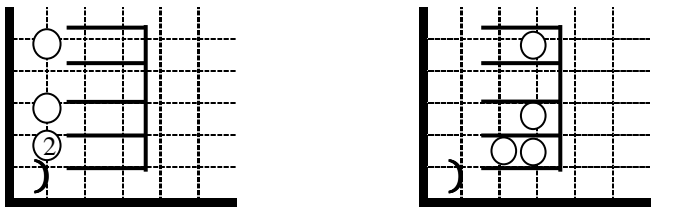


Estado Inicial
Karel tiene 10 pitos en la bolsa

Estado Final
Karel tiene 9 pitos en la bolsa

Figura 9. Posible estado inicial y final del problema de la granja

1.8.7 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, prográmelo para que los guarde.



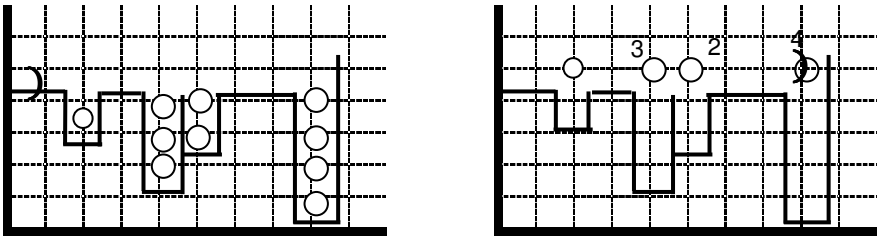
Estado Inicial
Karel tiene 0 pitos en la bolsa

Estado Final
Karel tiene 0 pitos en la bolsa

Figura 10. Posible estado inicial y final del problema de la librería

1.8.8 Haga un programa para que Karel se oriente de acuerdo con los pitos que tiene en su bolsa, de la siguiente manera: si no tiene pitos en la bolsa, debe quedar mirando al sur, si tiene un pito debe mirar hacia el este, si tiene 2 hacia el norte y con 3 hacia el oeste. Karel parte mirando en cualquier dirección y al terminar debe haber colocado en su esquina todos los pitos de su bolsa.

1.8.9 Karel va a hacer una fiesta y desea preparar los pasabocas. Para esto va a su despensa de galletas (pitos). La despensa consta de cajas de diferentes tamaños que se extienden hacia el sur desde la calle 5 entre las avenidas 2 y 8. En cada una de estás avenidas puede haber cajas de tamaño 0, 1, 2, ó 4. Las cajas están llenas de galletas (una en cada esquina). Programe a Karel para que saque estás galletas y las coloque "encima" de cada caja, es decir, sobre la calle 5 a la altura de la avenida de donde las tomó.

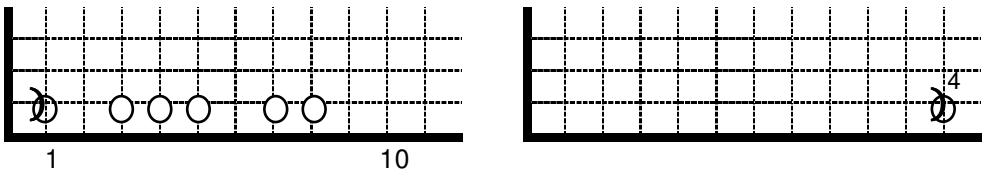


Estado Inicial
Karel tiene 0 pitos en la bolsa

Estado Final
Karel tiene 0 pitos en la bolsa

Figura 11. Posible estado inicial y final del problema de la fiesta

1.8.10 Karel se encuentra en el origen con 10 pitos en su bolsa; en cada una de las esquinas entre las avenidas 1 y 10 (inclusive), sobre la calle 1 puede haber 0 o 1 pito. Karel debe colocar en la esquina (1,11) de su mundo tantos pitos como posibles en estado inicial y el correspondiente estado final (Ayuda: no importa cuántos pitos queden al final en las otras esquinas).

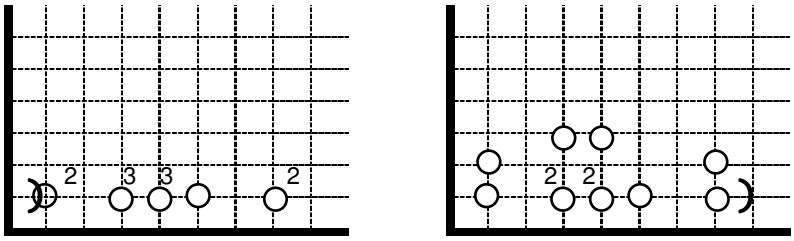


Estado Inicial
Karel tiene 10 pitos en la bolsa

Estado Final

Figura 12. Posible estado inicial y final del problema sin pitos

1.8.11 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). Para cada avenida Karel debe tomar uno de los pitos que hay en esa avenida sobre la calle 1 y colocarlo en la misma avenida sobre la calle cuyo número coincide con el número de pitos que había inicialmente en la avenida. Karel debe terminar en la esquina (1,8), mirando al este.

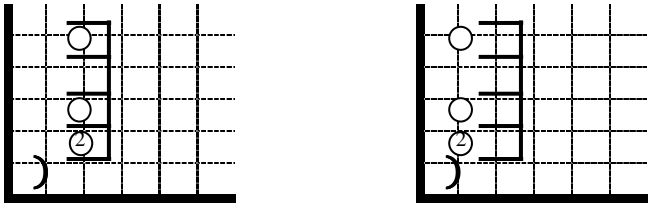


Estado Inicial
Karel tiene 0 pitos en la bolsa

Estado Final
Karel tiene 0 pitos en la bolsa

Figura 13. Posible estado inicial y final del problema subir un pito

1.8.12 Karel tiene un criadero de conejos, formados por cuatro corrales. Todos los días Karel saca a los conejos de su corral y los pone frente a la entrada para que coman. En cada corral puede haber hasta 3 conejos. Haga un programa para que Karel saque los conejos de cada corral y los ponga frente a la entrada (una esquina al oeste). Karel parte del origen mirando al este y termina igual. Ayuda: antes de comenzar a resolver el problema asegúrese de entender cuáles son las condiciones que deben estar presentes en todos los estados iniciales posibles.



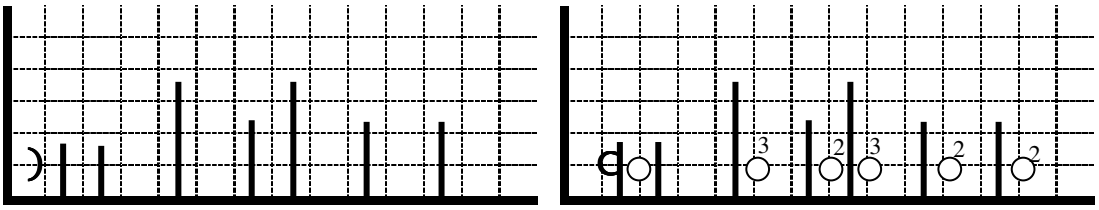
Estado Inicial
Karel tiene 0 pitos en la bolsa

Estado Final
Karel tiene 0 pitos en la bolsa

Figura 14. Posible estado inicial y final del problema criadero de conejos

1.8.13 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. Programe a Karel para que salte los obstáculos colocando detrás de cada una, a la altura de la calle 1 tantos pitos como metros midan el obstáculo y regrese al origen. Suponga que Karel parte con suficientes pitos.

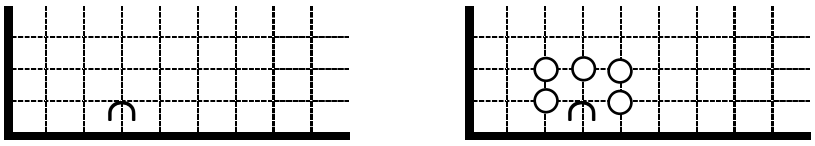


Estado Inicial
Karel tiene 30 pitos en la bolsa

Estado Final
Karel tiene 16 pitos en la bolsa

Figura 15. Posible estado inicial y final del problema carrera de obstáculos

1.8.14 Karel está probando diferentes técnicas de camuflaje. Haga un programa para que Karel se rodee de pitos, es decir, para que coloque un pito en cada una de las esquinas adyacentes a la esquina en la que se encuentra. Karel debe terminar en la misma posición y orientación en que comienza. Karel comienza con 8 pitos en su bolsa en cualquier esquina del mundo (puede estar contra uno de los muros infinitos del mundo).



Estado Inicial
Karel tiene 8 pitos en la bolsa

Estado Final
Karel tiene 3 pitos en la bolsa

Figura 16. Estado inicial y final del problema de camuflaje

1.8.15 En el origen del mundo de Karel, hay una clave que le va a indicar a Karel qué tarea debe realizar. La clave está dada por el número de pitos que hay en esa esquina así:

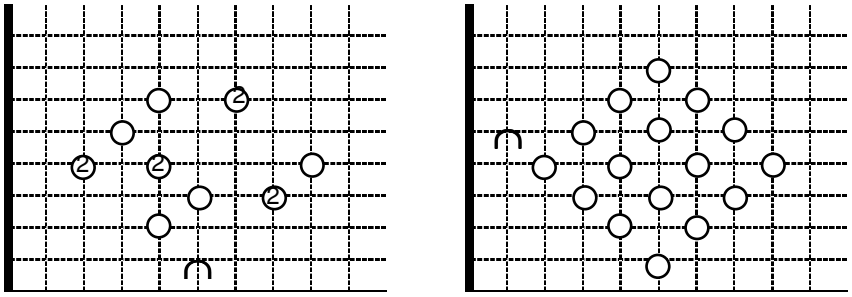
- si no hay pitos, Karel debe colocar cuatro pitos en diagonal

- si hay un pito, Karel debe colocar cuatro pitos verticalmente

- si hay dos pitos, Karel debe colocar cuatro pitos horizontalmente

La posición inicial de Karel es el origen, pero la orientación inicial es desconocida. Al final de la tarea que realice, Karel debe quedar en el sitio donde colocó el último pito. Haga un programa para que el robot, de acuerdo con la clave, realice la tarea correspondiente.

1.8.16 Karel, que hace poco se volvió vegetariano decidió plantar unas lechugas en su huerta (que tiene forma de diamante de lado 4). Karel es un poco descuidado y plantó en algunas de las esquinas 2 lechugas (pitos), dejando otras esquinas sin lechuga. Ayúdelo a redistribuir las lechugas, quitando una en las esquinas donde hay dos y colocando una en las esquinas que no tienen nada.

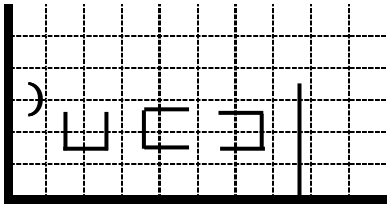


Estado Inicial
Karel tiene 20 pitos en la bolsa

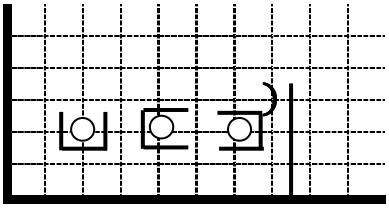
Estado Final
Karel tiene 17 pitos en la bolsa

Figura 17. Posible estado inicial y final del problema del vegetariano

1.8.17 Karel parte de la esquina (3,1) mirando al este. Sobre la calle 2 hay una serie de cajas de un bloque de ancho y un bloque de alto, que tienen una entrada hacia alguno de sus lados. Las cajas están separadas por una avenida. El final de las cajas está marcado por un bloque vertical de altura 3. Karel parte con suficientes pitos en su bolsa (más que el número de cajas). Prográmelo para que coloque un pito dentro de cada una de las cajas (en la esquina interior de la caja) y termine sobre la avenida 3 frente al muro de 3 bloques de alto.



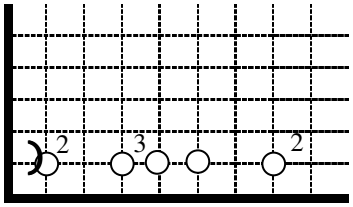
Estado Inicial
Karel tiene 20 pitos en la bolsa



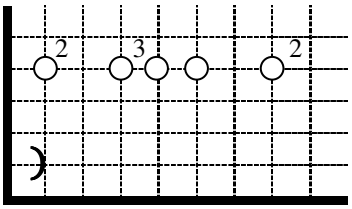
Estado Final
Karel tiene 17 pitos en la bolsa

Figura 18. Posible estado inicial y final del problema de las cajas

1.8.18 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.



Estado Inicial
Karel tiene 0 pitos en la bolsa



Estado Final
Karel tiene 0 pitos en la bolsa

Figura 19. Posible estado inicial y final del problema de subir pitos