Programación Orientada a Objetos en el Micro mundo del Robot Karel.
Libro No 2

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

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.

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.

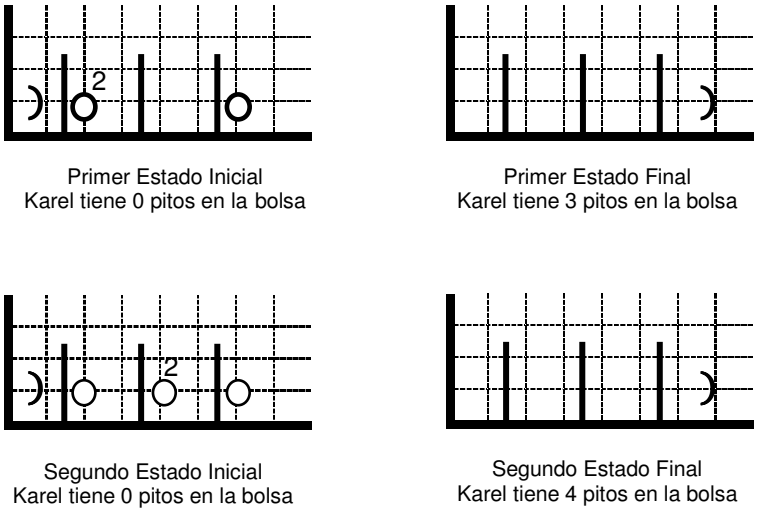


Figura 1. Estados Inicial y final de Karel

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:

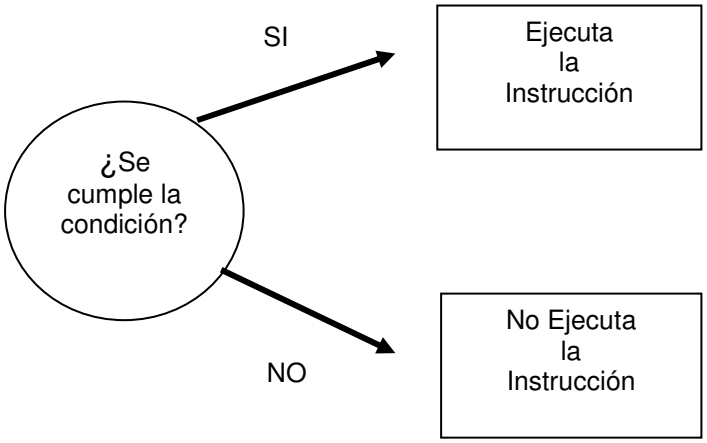
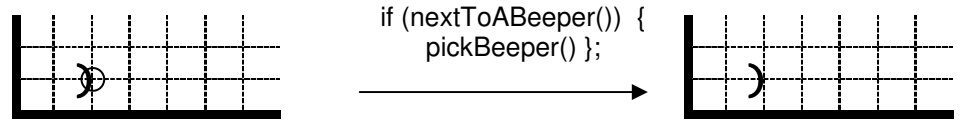


Figura 2. Estructura de la instrucción condicional simple

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.

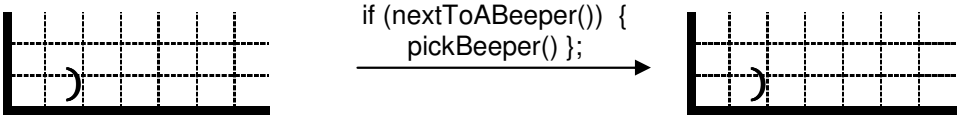


Estado inicial
Karel tiene 0 pitos en la bolsa

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 {…}.

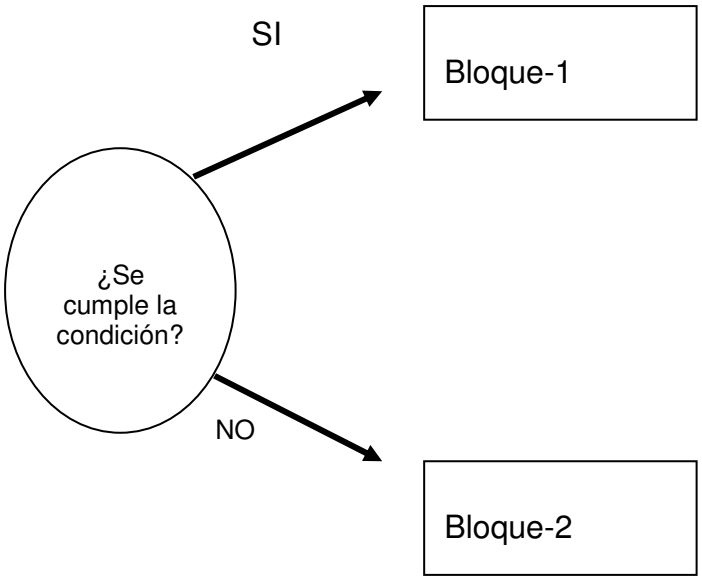


Figura 5. Estructura de la instrucción condicional compuesta

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.

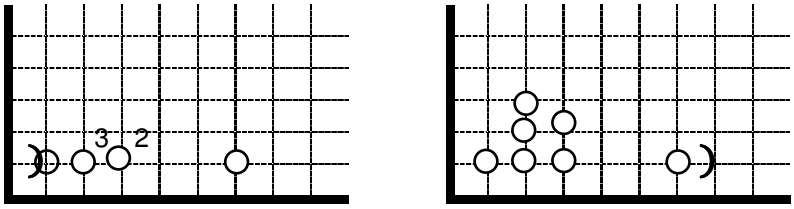


Figura 6. Estado Inicial y Final del Programa Distribuidor

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:

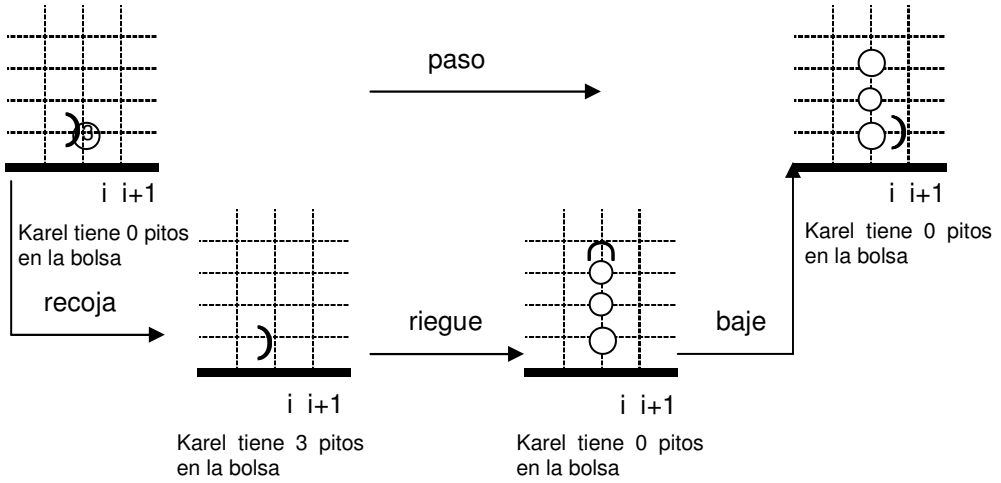


Figura 7. Ilustración de un Estado Invariante del Programa Distribuidor

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.

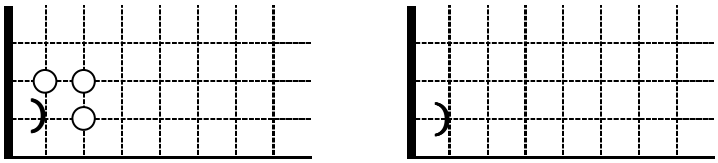


Figura 8. Estado inicial y final de un robot recogiendo pitos adyacentes a su posición

```
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

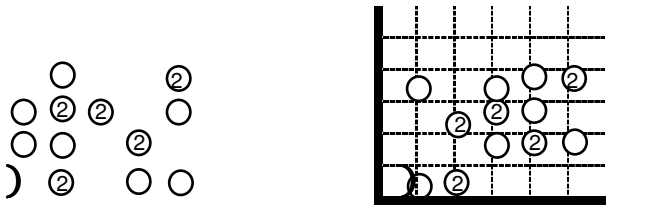
```
void saltador:: baje()
{
  move();
 if  ( frontIsClear() )
{
   move();
}
         if ( frontIsClear() )
          {
              move();
          }

     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.

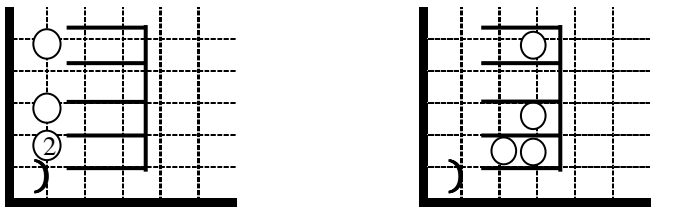


Estado Inicial
Karel tiene 10 pitos en la bolsa

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.



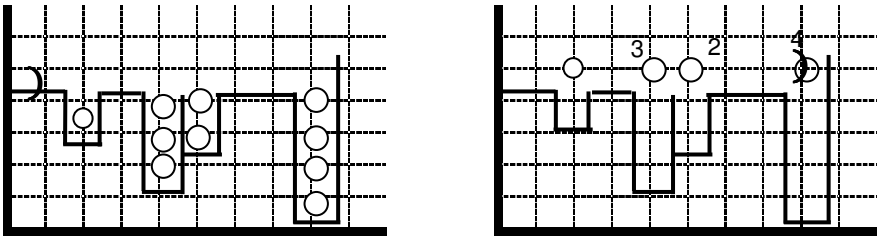
Estado Inicial
Karel tiene 0 pitos en la bolsa

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.

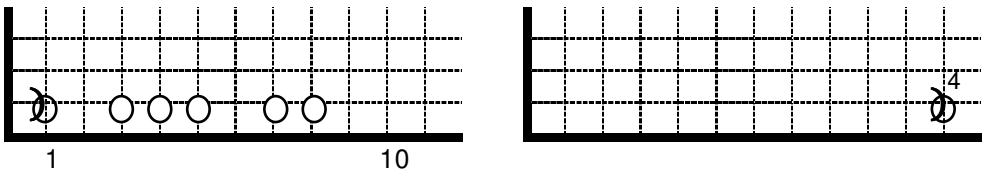


Estado Inicial
Karel tiene 0 pitos en la bolsa

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).

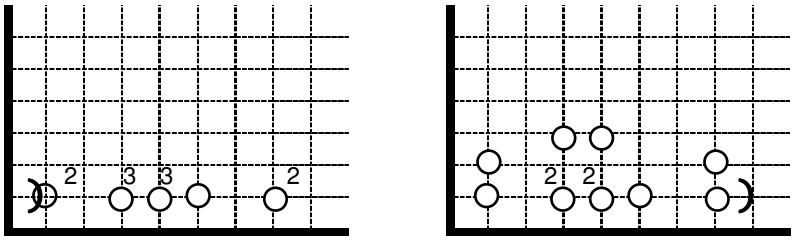


Estado Inicial
Karel tiene 10 pitos en la bolsa

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.

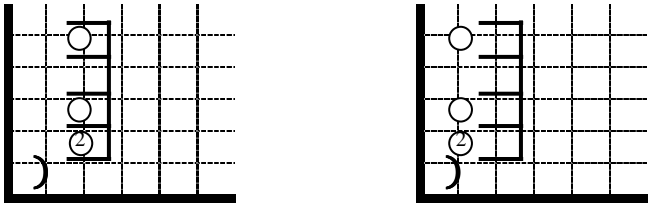


Estado Inicial
Karel tiene 0 pitos en la bolsa

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.



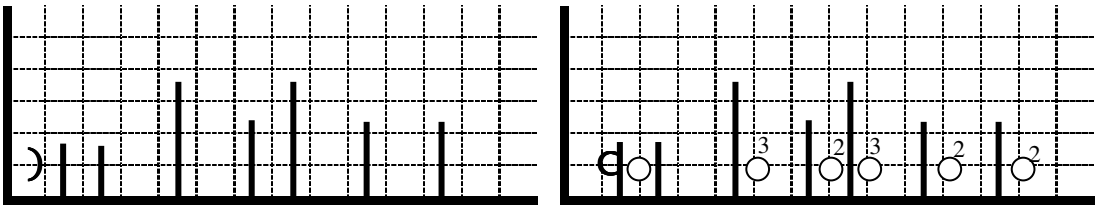
Estado Inicial
Karel tiene 0 pitos en la bolsa

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.

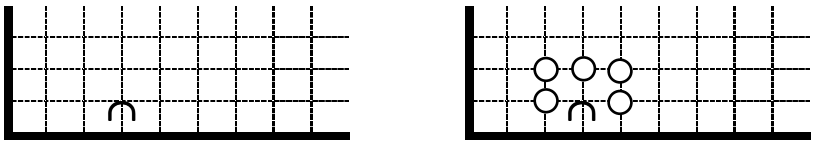


Estado Inicial
Karel tiene 30 pitos en la bolsa

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:

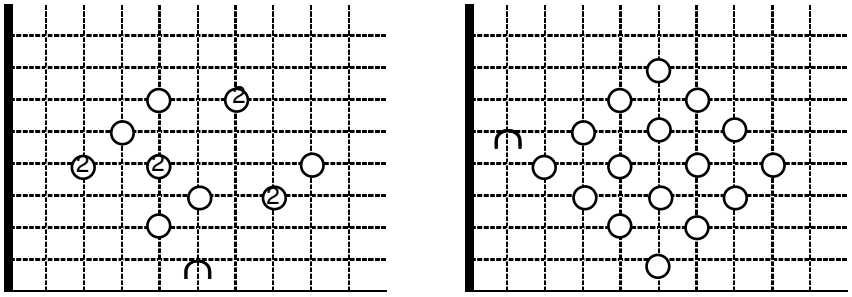
- si no hay pitos, Karel debe colocar cuatro pitos en diagonal

- 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.

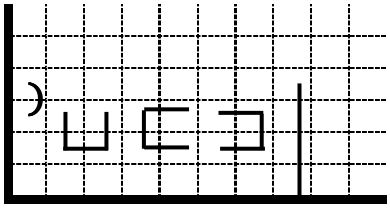


Estado Inicial
Karel tiene 20 pitos en la bolsa

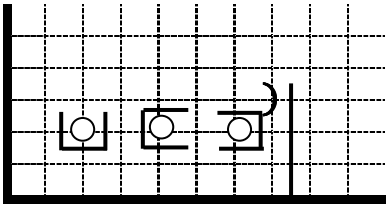
Estado Final
Karel tiene 17 pitos en la bolsa

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.



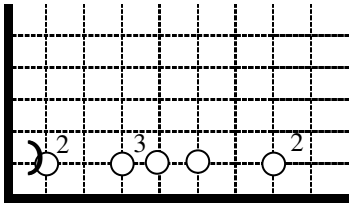
Estado Inicial
Karel tiene 20 pitos en la bolsa



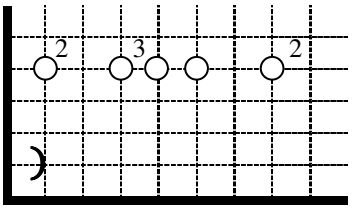
Estado Final
Karel tiene 17 pitos en la bolsa

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

2. INSTRUCCIONES DE REPETICION

Objetivo. Este capítulo termina la discusión de las instrucciones construidas en el vocabulario del lenguaje de programación del robot. Las dos nuevas instrucciones que se aprenderán son while{...}. Las instrucciones se pueden ejecutar en varias ocasiones, en cualquier instrucción que entienda el robot. Estás adiciones realzan ampliamente la concisión y energía del lenguaje de programación del robot.

Somos ya programadores experimentados del robot. Una vista abreviada del mundo del robot será utilizada para algunas figuras. Reducir alboroto visual, las etiquetas de la calle y de la avenida y, de vez en cuando, los muros meridionales u occidentales no serán mostrados. Como de costumbre, en nuestros ejemplos se utilizará un solo robot, a menudo nombrado Karel. Puesto que la mayoría de nuestros ejemplos implicarán la manipulación de pitos. Suponemos que seguimos enriqueciendo la clase Robot.

2.1 INSTRUCCIÓN WHILE

Karel está en el origen mirando al este. Entre dos avenidas n y n+1 (n no se conoce) hay un muro de altura 1 en la calle 1. Sobre la calle 1 y hay una fila de pitos que se extiende desde el origen hasta la avenida n (uno en cada esquina). Karel debe recoger estos pitos y terminar frente al muro.

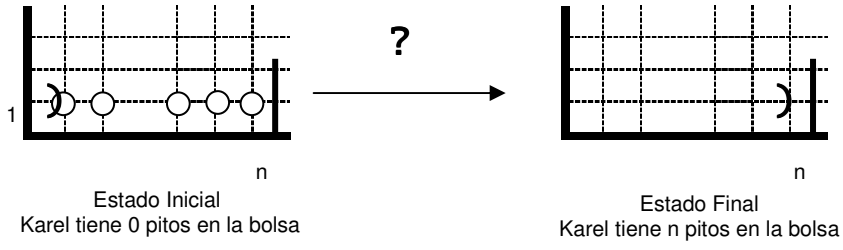


Figura 20. Posibles estados inicial y final del problema propuesto

Note que el número de esquinas con pito no es fijo como en el problema anterior. Hay una cantidad desconocida de esquinas con pito. No se puede usar loop(n) porque no se conoce el valor de n. Se sabe que el robot debe repetir el paso hasta que encuentre un muro al frente (frente bloqueado), es decir, debe hacerlo mientras su frente esté despejado.

El problema es que se debe repetir un paso "varias" veces, pero no se sabe exactamente cuántas. Sin embargo, hay una condición en el mundo que le indica a Karel cuando terminar la repetición. En estos casos se utiliza la instrucción repetitiva while.
Está instrucción tiene la siguiente estructura general:

```
While <condición>
{
  <instrucciones>
}
```

Karel ejecuta las <instrucciónes> mientras la <condición> sea verdadera. Esto es, evalúa la condición en el estado en que se encuentra el mundo y si se cumple ejecuta la instrucción (modificando el mundo) y vuelve a evaluarla en este nuevo estado. Esto se repite hasta que la condición se deje de cumplir (termina cuando se cumple lo contrario). Cada ejecución de las <instrucciónes> se denomina una iteración. La <condición> se llama la condición de entrada. Una instrucción while se llama un ciclo. Si el ciclo tiene una sola instrucción, no podemos suprimir el {...}.

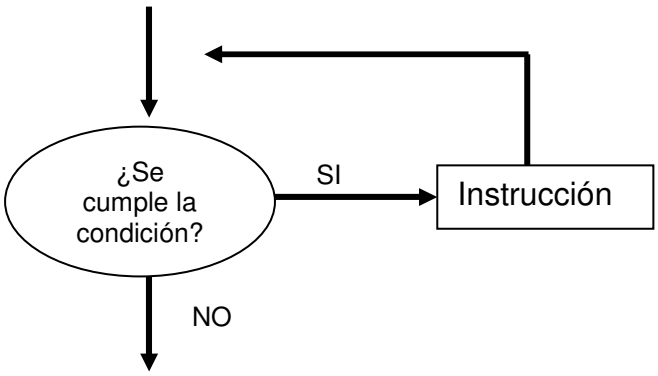


Figura 21. Estructura de la instrucción while

En el problema del ejemplo anterior, se debe repetir el paso hasta que Karel llegue al muro, luego la condición de entrada al ciclo es frontIsClear(), y se obtiene un segmento del programa:

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

2.2 DESARROLLO DE CICLOS while

En el momento cuando decidimos utilizar un while como parte de la solución de un problema se debe tratar de identificar varios factores:

1. El estado del mundo antes de comenzar a ejecutar el ciclo.
2. El estado del mundo en el momento de terminación de ejecutar el ciclo.
3. La transformación gradual que se desea aplicar al mundo: ¿Cómo acercarnos a la solución en cada paso?. ¿Cuál es la regularidad que existe en el problema que permite resolverlo repitiendo varias veces un conjunto de instrucciones?. Para determinar esto es necesario tener en cuenta que:

- Se debe garantizar que en algún momento se llegue a la condición de salida (la negación de la condición de entrada) para que while termine.
- La ejecución del cuerpo del ciclo debe llevar a un estado que se vaya "acercando" al estado final deseado.
- Cada vez que se entre al ciclo se debe encontrar un estado similar, ya que en el problema existe una regularidad.

Como se verá más adelante, una manera de formalizar la esencia de este proceso es mediante un invariante, y así es más fácil desarrollar después el cuerpo del while.

La condición de salida: cuando no interesa volver a ejecutar el ciclo es porque ya llegamos al estado deseado. La negación de está condición es siempre la condición de entrada (la guarda del while). En el ejemplo anterior, en el momento en que el frente esté bloqueado no interesa que Karel vuelva a repetir el cuerpo del while, luego la condición de entrada, será frente desbloqueado (o sea, la negación del frente bloqueado).

El Invariante

Una invariante es una condición que satisface cualquier estado intermedio ("después de i iteraciones"). Dado que existe una regularidad en el proceso (transformación gradual), el invariante dice cómo es el estado del mundo en todo instante durante la ejecución del ciclo: antes de entrar la primera vez, después de cada iteración y al finalizar. Un invariante es útil para el proceso de desarrollo, si muestra el estado de todos los elementos involucrados en el problema, en un punto intermedio de la solución.

Observe el problema anterior:

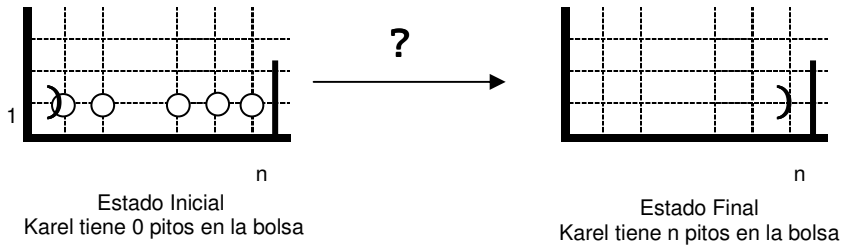


Figura 22. Posibles estados inicial y final de un variante del problema

Para solucionar el problema necesitamos que Karel vaya recogiendo uno a uno los pitos sobre la calle. El invariante que nos "explica" está transformación es el siguiente:
Karel ha recogido los primeros (i -1) pitos y se encuentra sobre la avenida i'ésima, donde $1 \leq i \leq n$. Gráficamente se tiene el mismo invariante así:

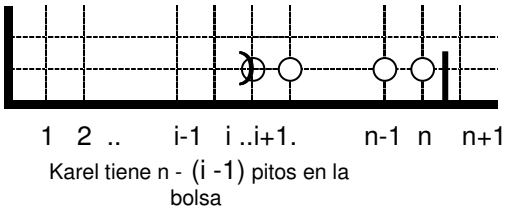


Figura 23. Estado invariante del problema

Lo cual describe claramente el proceso y muestra la forma cómo éste "avanza" hacia la solución del problema.

Se sabe que Karel no puede seguir avanzando cuando encuentra el muro, por lo tanto, la condición de entrada al ciclo debe ser:

```
frontIsClear()
```

Si se supone que el invariante se cumple en el estado i'ésimo para que se conserve en el siguiente estado, el cuerpo del ciclo debe ser tal que se llegue a:

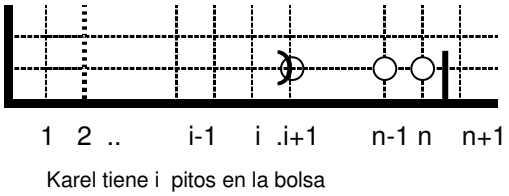


Figura 24. Estado invariante del problema

Es fácil ver que Karel debe recoger el pito y avanzar una avenida:

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

El ciclo completo es entonces:

```
while ( frontIsClear() )
{
  pickBeeper();
```

```
  move();
}
```

El estado final es:

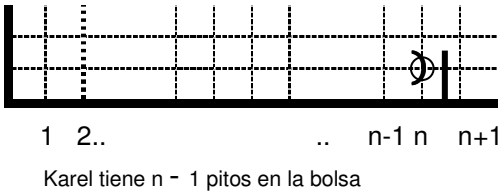


Figura 25. Estado final del problema

Como todavía falta el último pito, el programa completo es el siguiente:

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

Ejemplo Completo: Otra Cosecha

Karel debe recolectar en un campo de 5 surcos verticales de longitud variable. Los surcos empiezan siempre en la calle 1 y van a lo largo de las avenidas, a partir de las avenidas 2 hasta la 6. Todos los pitos en cada surco están juntos, y hay a lo sumo un pito por esquina. A continuación, se describe un estado inicial posible y el correspondiente estado final:

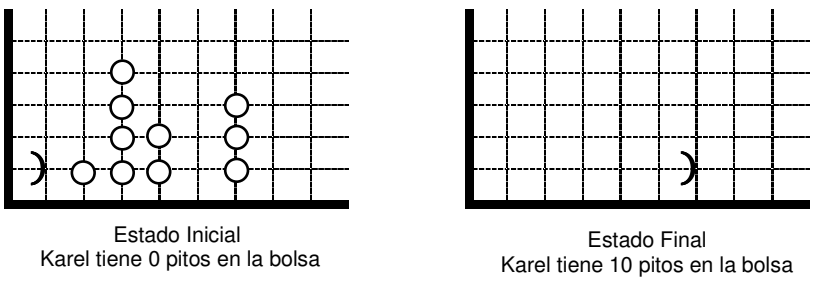


Figura 26. Posibles estados inicial y final del problema recolectar

Plan General:

El enunciado del problema habla de 5 surcos verticales, todos con las mismas características, que deben recibir el mismo tratamiento por parte de Karel, lo que sugiere dividir el problema en 5 subproblemas iguales que se llamarán: recoja-surco. Ahora bien, Karel comienza antes del primer surco y termina a la altura del último, por lo que cada recoja-surco debe comenzar con Karel una avenida antes del surco que va a recoger, mirando al este y terminar sobre la avenida del surco recogido (después de haberlo recogido).

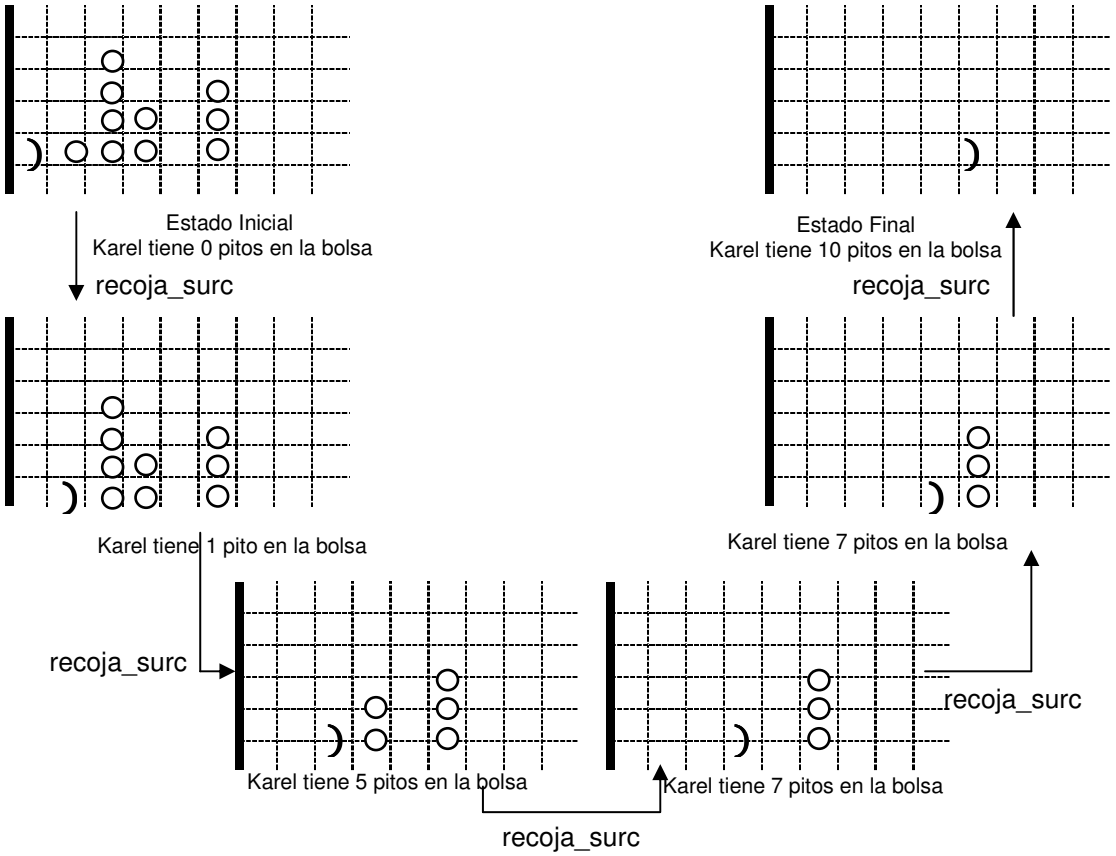


Figura 27. Estado invariante del problema

Bloque principal de ejecución:

```
task
{
 Recolector karel(1,1,East,o);
```

```
loop(5)
    {
       karel.recoja_surco();
       karel.turnOff();
    }
```

Solución del subproblema recoja-surco

La instrucción recoja-surco es todavía bastante compleja, por lo que se debe dividir en subproblemas: en uno que indique a Karel cómo moverse el surco y subir recogiendo los pitos (suba-recogiendo) y en otro para que Karel baje y se localice para el siguiente surco, mirando al este (baje):

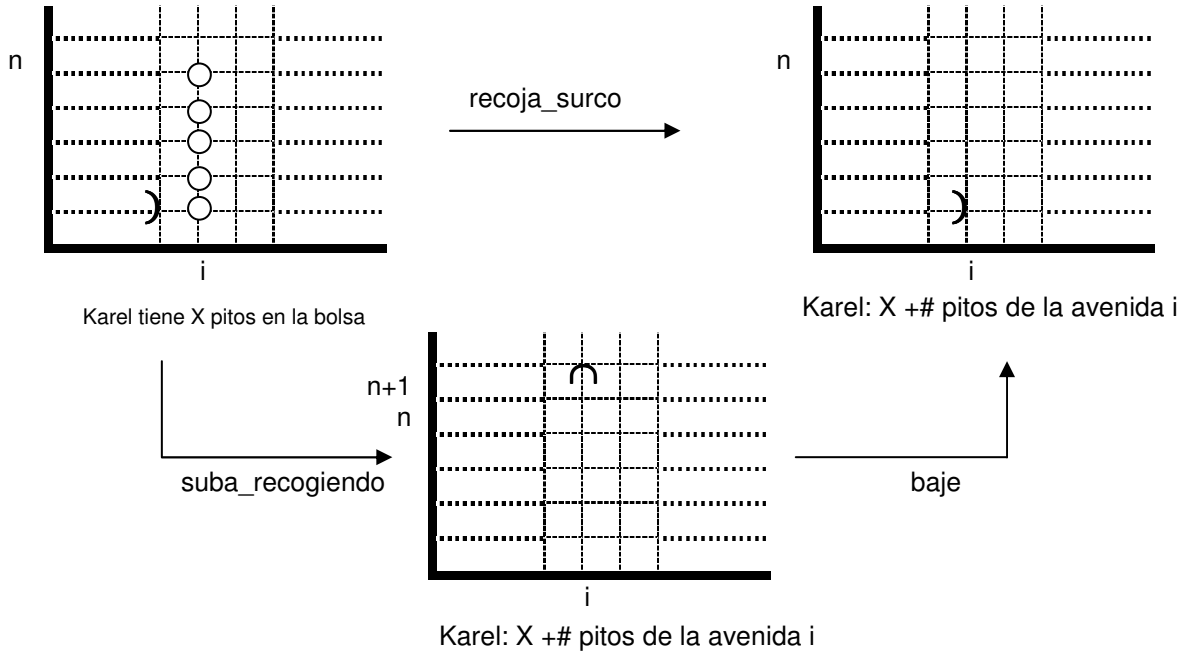


Figura 28. Estado invariante del problema

El bloque de definición de recoja-surco sería entonces:

```
void Recolector:: recoja_surco()
{
   suba_recogiendo();
```

```
   baje();
}
```

Solución del subproblema suba-recogiendo:

Después de un move() y un turnLeft() que localiza a Karel en el surco hacia arriba para comenzar a recoger se tiene:

Figura 29. Estado invariante de suba - recogiendo

No se sabe cuántos pitos hay sobre la calle y Karel debe recogerlos todos. Para poder resolver este problema se requiere un ciclo (while) para que Karel recoja un pito y se mueva, hasta que no haya más pitos. El siguiente estado (j'ésimo) muestra el invariante:

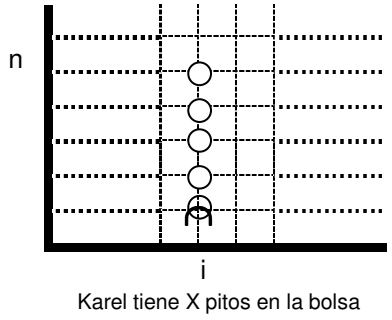


Figura 29. Estado invariante de suba - recogiendo

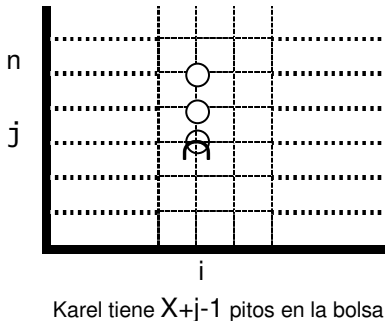


Figura 30. Estado invariante de subproblema

Para que se conserve el invariante en el estado j+1 el cuerpo del ciclo debe ser:

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

La solución completa es entonces:

```
void Recolector:: suba_recogiendo()
{
   move();
   turnLeft();
    While ( nextToABeeper() )
    {
      pickBeeper();
     move();
   }
}
```

Solución del subproblema baje:

Está instrucción debe orientar a Karel hacia el sur con dos turnLeft(), hacerlo que se mueva hasta un muro (con un while) y colocarlo hacia el este, con otro turnLeft():

```
void Recolector:: baje()
{
   turnLeft();
   turnLeft();

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

El programa completo tendría que tener la definición de estás instrucciones en el orden contrario al que se definieron en la solución del problema; es decir, debe definirse primero baje y suba-recogiendo y luego recoja-surco.

```
class Recolector: robot
{
  void recoja_surco();
  void baje();
  void  suba_recogiendo()
}
void Recolector:: baje()
{
   turnLeft();
turnLeft();

    While ( ( frontIsClear() )
    {
      move();
    }
     turnLeft();

}

void Recolector:: suba_recogiendo()
{
   move();
   turnLeft();
    While ( nextToABeeper() )
    {
      pickBeeper();
     move();
   }
}
}

void Recolector:: recoja_surco()
{
   suba_recogiendo();
```

```
   baje();
}

task
{
 Recolector karel(1,1,East,0);
   loop(5)
       {
          karel.recoja_surco();
          karel.turnOff();
       }
}
```

2.3 EJERCICIOS PROPUESTOS

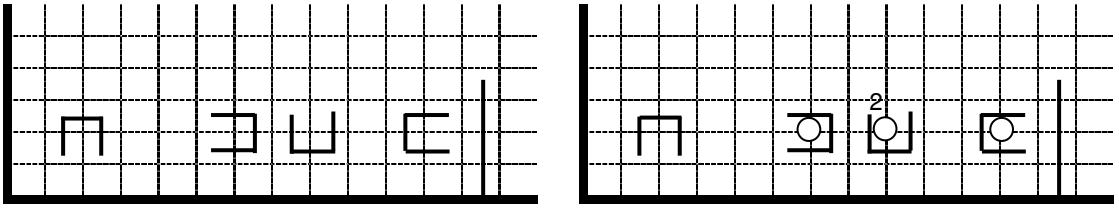
2.3.1. Karel se encuentra en un mundo lleno de cajas cuadradas (cajas formadas con tres muros de un bloque, que encierran una esquina y tiene una entrada por el lado norte). Estás cajas se encuentran todas sobre la calle 2 (encierran una esquina de la calle 2), separadas a una distancia irregular: el número de cajas es variable, pero al final de la última caja existe un muro vertical de altura 3 que corta las calles 1, 2 y 3 y que debe servirle a Karel para reconocer el final de la fila de cajas. Karel parte del origen, mirando al este, con suficientes pitos en su bolsa.

a) Prográmelo para que coloque un pito en la esquina interna de cada una de las cajas.

b) Modifique el programa para que funcione si las cajas están orientadas en cualquier dirección, es decir, si su entrada puede estar hacia el norte, el sur, el este o el oeste.

c) Modifique la solución anterior para que ahora Karel, en un mundo con las mismas especificaciones, coloque 0,1, o 2 pitos en cada caja así: si la caja tiene su entrada hacia el sur, Karel no debe colocar pitos (pues se saldrían), si la entrada está hacia el este o hacia el oeste debe colocar 1 pito y si está

hacia el norte debe colocar en ella dos pitos. A continuación, se muestra un posible estado inicial y el correspondiente estado final para este último caso:



Estado Inicial
Karel tiene 30 pitos en la bolsa

Estado Final
Karel tiene 26 pitos en la bolsa

Figura 31. Estado inicial y final problema de las cajas cuadradas

2.3.2 Revise las siguientes parejas de códigos y diga sí son o no equivalentes

Segmento A

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

Segmento B

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

2.3.3 Revise las siguientes parejas de códigos y diga sí son o no equivalentes

Segmento A

```
while (anyBeepersInBeeperBag() )
{
   putBeeper();
   if (frontIsClear())
   {
     move();
    }
}
```

Segmento B

```
while (frontIsClear()() )
{
   if (anyBeepersInBeeperBag())
   {
     putBeeper();
     move();
    }
}
```

2.3.4 Diga si los siguientes segmentos de programa son o no equivalentes:

Segmento A

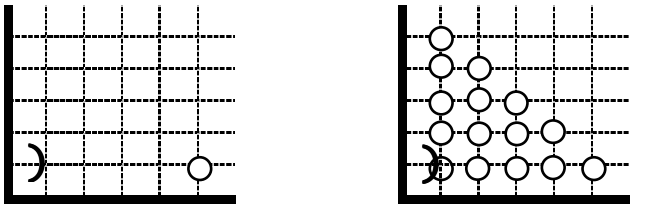
```
loop(3)
{
    move();
    while (facingNorth())
   {
     turnLeft();
     pickBeeper();
   }
      turnLeft();
```

```
}
```

Segmento B

```
while (facingNorth())
{
     move();
     loop(3)
     {
     turnLeft();
     pickBeeper();
    }
      turnLeft();
}
```

2.3.5 Karel se encuentra en un mundo en el que hay un pito en la esquina de la calle 1 con una avenida x. Programe a Karel para que construya un triángulo isósceles, de lado x, relleno de pitos, de forma que su vértice superior quede en el origen. Karel parte del origen, mirando al este y debe terminar en la misma posición. Suponga que el robot tiene suficientes pitos en su bolsa. A continuación, se presenta un posible estado inicial y el correspondiente estado final:



Estado Inicial
Karel tiene 16 pitos en la bolsa

Estado Final
Karel tiene 2 pitos en la bolsa

Figura 32. Condiciones iniciales y finales del problema construir isósceles

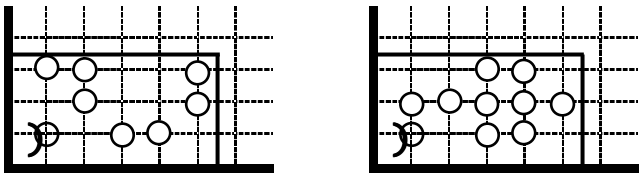
2.3.6 Modifique el programa anterior para que Karel construya únicamente el perímetro de un triángulo isósceles, sin rellenarlo.

2.3.7 Karel ha sido contratado por artesanías de Colombia para elaborar sus nuevos diseños de tapetes. Para construir el arte del tapete, Karel debe trabajar dentro de un cuarto rectangular, colocando un pito en algunas esquinas del cuarto. Karel parte de la esquina inferior izquierda del cuarto, mirando al este, con suficientes pitos en su bolsa. Para seguir el tejido, Karel debe obedecer las siguientes reglas:

a) El diseño (pitos) que está sobre la calle más al sur del cuarto, debe quedar intacto.
b) Para decidir el patrón del tejido para una calle del cuarto es necesario haber resuelto el patrón para las calles que están al sur de éstas.
c) Para decidir el patrón final en una calle dada, se hace lo siguiente:

- Si sobre la esquina en consideración no hay pito, pero en la inmediatamente al sur de ésta no hay pito, se deja tal como está.
- Si en la esquina en consideración no hay pito, pero en la inmediatamente al sur sí hay, Karel debe colocar uno en la esquina en consideración.
- Si en la esquina de Karel hay pito y en la de más al sur no hay, Karel deja la esquina intacta.
- Si tanto en la que está, como en la de más al sur hay pito, Karel recoje el de su esquina.

Programe a Karel para que resuelva está tarea. A continuación, se da un ejemplo:



Estado Inicial
Karel tiene 16 pitos en la bolsa

Estado Final
Karel tiene 0 pitos en la bolsa

Figura 33. Posibles estados iniciales del problema diseño de tapetes

2.3.8 Karel está sembrando flores (pitos) en las colinas de su mundo (una serie de montes de altura 4 pegados uno a otro M). Programe a Karel para que, partiendo del origen, mirando al este, coloque todas las flores que tiene en su

bolsa en las colinas. Karel debe terminar sobre la calle 1, al final de la última colina en la que colocó alguna flor. Note que a Karel se le pueden acabar las flores en la mitad de una colina, en tal caso debe terminar de recorrer la colina (sin colocar flores). A continuación, se ilustra un estado posible y el correspondiente estado final:

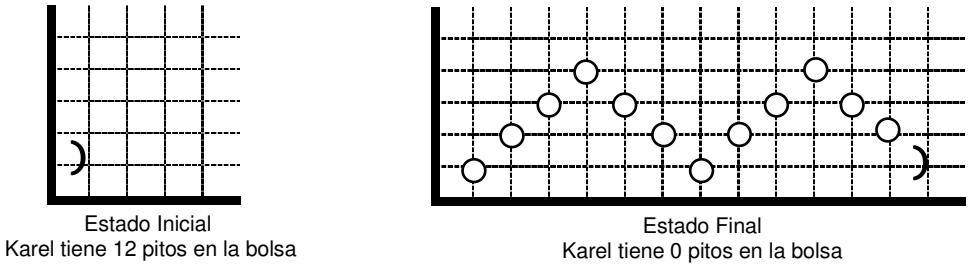


Figura 34. Posibles estados inicial y final del problema sembrando flores

2.3.9 Karel parte del origen (mirando al este) y debe recoger una serie de diagonales de pitos que parten de la calle 1 y se extienden hacia el nordeste, las diagonales pueden ser de cualquier tamaño; la primera diagonal parte del origen; a partir de ésta, el primer pito de cada diagonal se encuentra en la calle 1, sobre la avenida siguiente a la última avenida ocupada por la diagonal anterior. Por ejemplo, en el caso particular que se presenta a continuación, la primera diagonal termina en la avenida 4 y la segunda comienza en la 5:

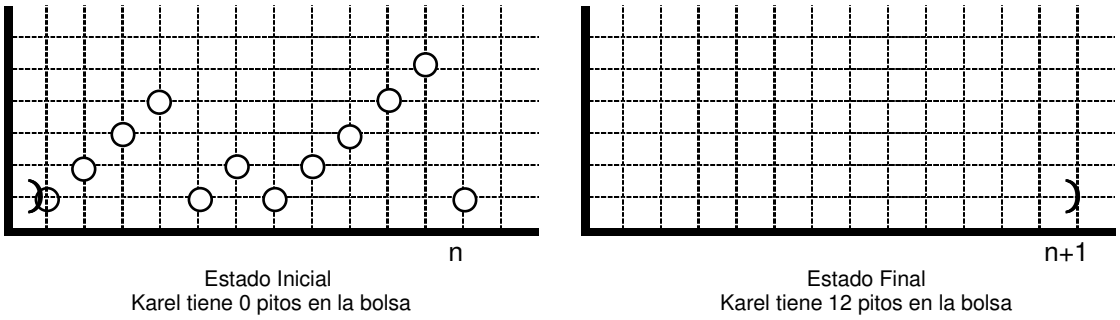


Figura 35. Estados inicial y final del problema haciendo diagonales

2.3.10 Karel se encuentra sobre la calle 1, mirando al este, frente a un obstáculo vertical de longitud x, Karel tiene en su bolsa n pitos ($n \geq x$).

Prográmelo para que coloque x de sus n pitos delante del obstáculo (al oeste del muro) y el resto (n-x) del lado este del obstáculo (ambos grupos sobre la calle 1). Karel debe terminar en su posición inicial. <<

A continuación, se da un ejemplo del problema:

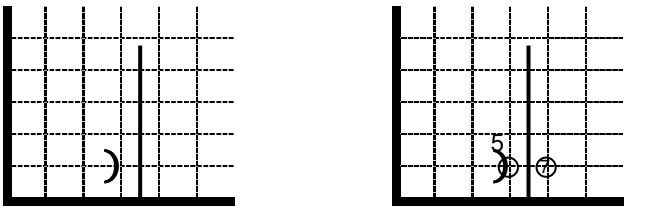


Figura 36. Estados inicial y final del problema en colocar pitos

2.3.11 En el mundo de Karel se tienen dos pitos: uno sobre la calle 1 en alguna avenida "x" y otro en la avenida 1 sobre la calle "z". Programe a Karel para que coloque un pito en la esquina de la calle "z", avenida "x". Karel comienza sin pitos en la bolsa; note que no se especifica la posición en la que debe terminar Karel (puede escoger la que más le sirva para su algoritmo). A continuación, se presenta un posible estado inicial y el correspondiente estado final.

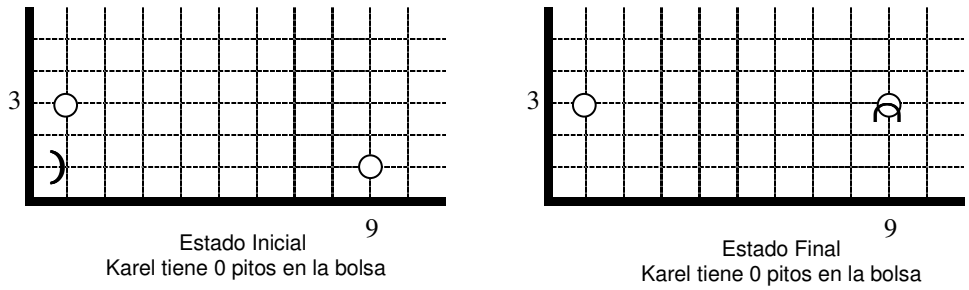
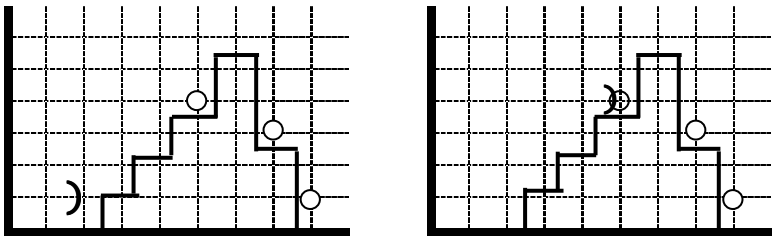


Figura 37. Estados inicial y final del problema "colocando pito calle z, avenida x"

2.3.12 Escriba un programa en lengua Karel para que el robot suba una montaña irregular (de cualquier forma, pegada al muro infinito del sur del mundo), hasta encontrar un pito (que se encuentra contra la montaña). Karel

parte del origen, mirando al este, frente a la montaña. A continuación, se muestra un caso posible.

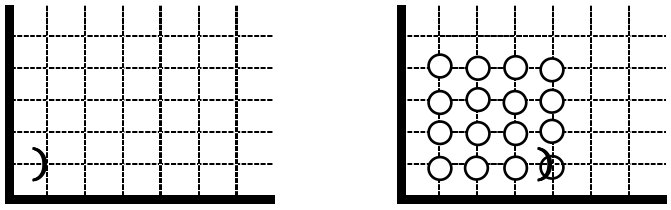


Estado Inicial
Karel tiene 0 pitos en la bolsa

Estado Final
Karel tiene 0 pitos en la bolsa

Figura 38. Estados inicial y final del problema suba montaña irregular

2.3.13 Karel se encuentra en el origen, mirando al este con N^2 pitos en su bolsa. Prográmelo para que coloque todos sus pitos en un cuadrado de lado N a partir del origen. A continuación, se da un posible estado inicial y su correspondiente estado final.

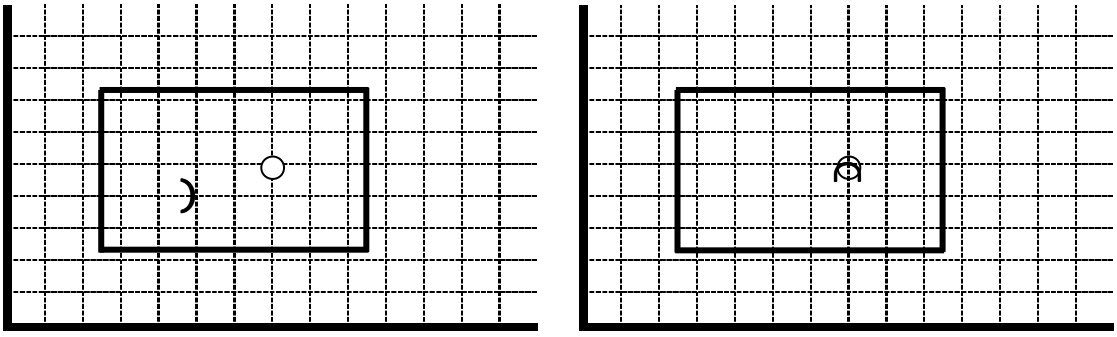


Estado Inicial
Karel tiene 16 pitos en la bolsa

Estado Final
Karel tiene 0 pitos en la bolsa

Figura 39. Estados inicial y final del problema colocar pitos en un cuadrado

2.3.14 Karel se encuentra en un cuarto rectangular cerrado (sin muros interiores), en el que se le ha perdido un pito. Programe a Karel para que localice el pito. (Nota: en el estado inicial tanto Karel como el pito pueden estar en cualquier posición dentro del rectángulo). A continuación, se presenta un posible estado inicial y el correspondiente estado final.



Estado Inicial
Karel tiene 0 pitos en la bolsa

Estado Final
Karel tiene 0 pitos en la bolsa

Figura 40. Estados inicial y final del problema localice el pito

2.3.15 Karel se encuentra en una oficina rectangular que tiene varias puertas y ventanas y ha perdido un documento importante. Ayude a Karel a encontrar el documento (pito), sin salirse de su oficina. Ayuda: dentro de la oficina no hay muros. A continuación, se presenta un posible estado inicial y el correspondiente estado final.

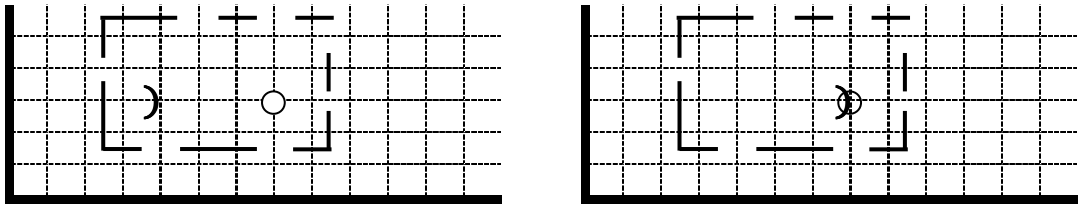


Figura 41. Estados inicial y final del problema encontrar documento

2.3.16 En el mundo de Karel hay una columna vertical (de longitud impar) de pitos sobre alguna avenida del mundo, a partir de la calle 1. Programe a Karel para que construya otra columna de pitos, de igual tamaño que se cruce con la original en el centro, formando una cruz perfecta. Suponga que la columna original está suficientemente separada del muro infinito vertical del mundo para que quepa la cruz y que Karel parte del origen con suficientes pitos en su bolsa. Por ejemplo:

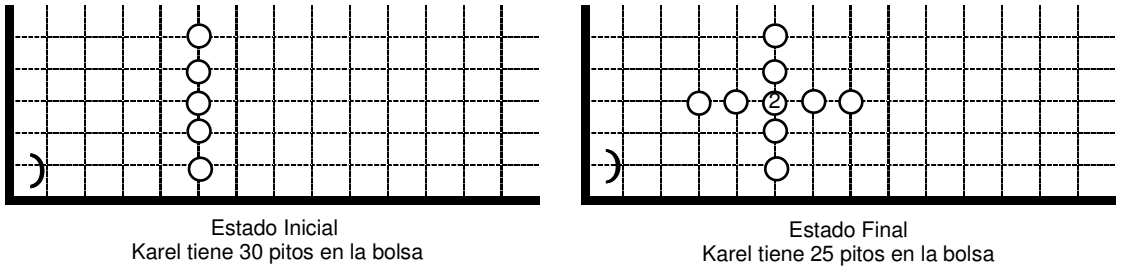
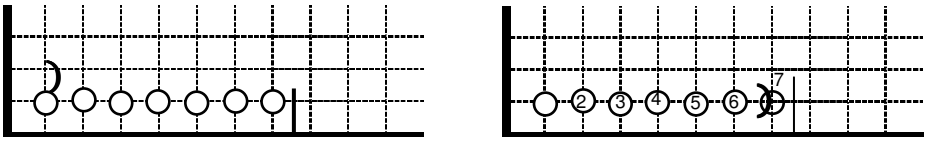


Figura 42. Estados inicial y final del problema “Construya columna de pitos”

2.3.17. En el mundo de Karel hay un muro que corta la calle 1 a una distancia desconocida. Programe a Karel para que genere los números naturales, desde 1 hasta el muro, es decir que coloque en el origen un pito, en la segunda avenida 2, en la siguiente 3, etc. Hasta llegar al muro. Suponga que Karel parte

del origen, mirando al este, con suficientes pitos en su bolsa. A continuación, se muestra un ejemplo posible:

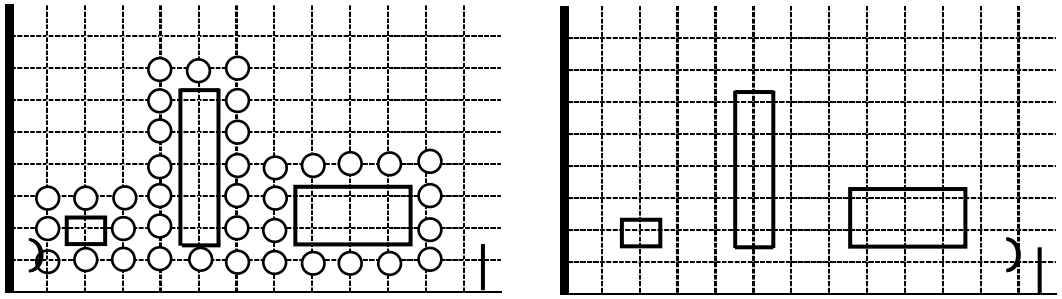


Estado Inicial
Karel tiene 30 pitos en la bolsa

Estado Final
Karel tiene 2 pitos en la bolsa

Figura 43. Estados inicial y final del problema "Genere los números naturales"

2.3.18. Karel está muy preocupado por el problema de las basuras del Mundo-Karel. En su candidatura para la alcaldía ofreció resolver este problema y ahora nos solicita que lo ayudemos. En el Mundo-Karel hay varias construcciones (rectángulos de segmentos de muro: ™ de cualquier tamaño) localizados encima de la calle 1, a partir de la avenida 1. Estás construcciones están separadas entre sí por 2 avenidas. Dos avenidas después de la última construcción, hay un muro (de altura 1 bloque), que corta la calle 1. En cada una de las esquinas que rodean una construcción hay una caneca de basura (pito). Haga un programa para que Karel, partiendo del origen, mirando al este, recoja todas las canecas de basura (pitos) que rodean las construcciones y termine frente al muro que corta la calle 1, mirando al este. A continuación, se muestra un posible estado inicial y el correspondiente estado final:



Estado Inicial
Karel tiene 0 pitos en la bolsa

Estado Final
Karel tiene 38 pitos en la bolsa

Figura 44. Estados inicial y final del problema "Basuras"

2.3.19 Karel parte del origen, mirando al este. A una distancia desconocida del origen, hay un muro vertical de una altura mayor que 6. Sobre la calle 1, entre el origen y este muro vertical hay "montones" de pitos (cualquier cantidad entre 0 y 6 en cada esquina). Karel debe ordenar estos montones de mayor a menor (de acuerdo con el número de pitos que haya en cada uno) a partir del origen, y terminar contra el muro vertical, mirando al este. A continuación, se presenta un estado inicial posible y el estado final correspondiente y se da un plan de solución para el problema general.

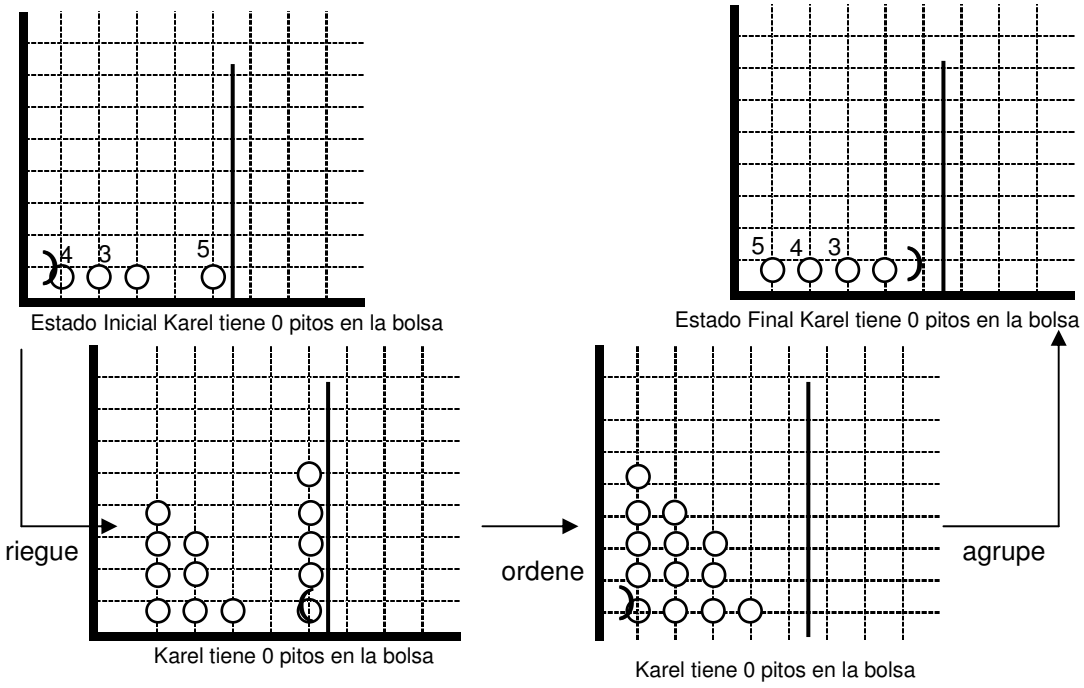


Figura 45. Invariante del problema propuesto

De acuerdo con el plan presentado anteriormente, el bloque de ejecución del programa resulta de la siguiente manera:

```
task
{   ur_Robot  karel(1,1,East,0);
      karel.riegue();
      karel.ordene();
      karel.agrupe();
      karel.turnOff();
}
```

La instrucción ordene consiste en repetir 6 veces (1 para cada calle i a partir de la 1): recoger todos los pitos de la calle i, colocarlos a partir de la avenida 1 y subir a la calle siguiente (i+1)

A continuación, se describen gráficamente estos tres pasos para la calle i:

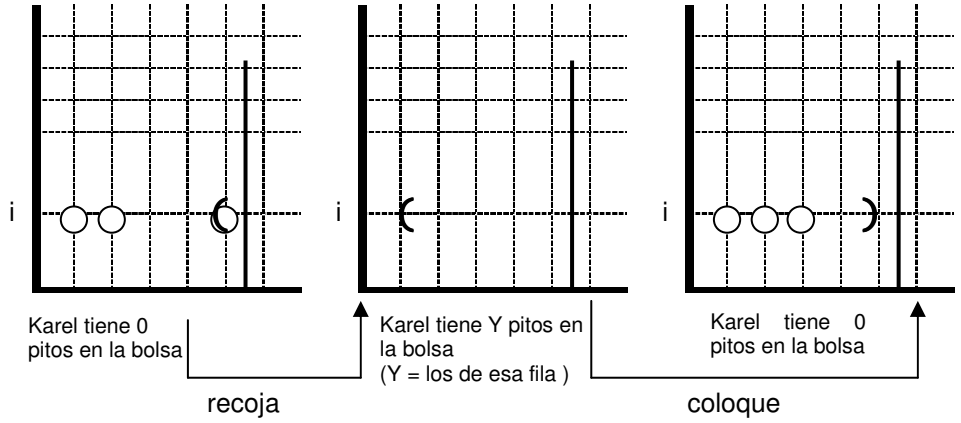


Figura 46. Invariante del problema enunciado

Después de repetir 6 veces estos pasos, la instrucción ordene debe hacer que karel vuelva al origen (vuelvaorigen) y quede mirando al este (para seguir con agrupe). La definición de ordene es:

```
void clase::ordene()
{
  loop(6)
   {
    recoja();
    coloque();
    suba();
   }
 vayaorigen();
}
```

Termine de escribir el programa de solución, definiendo las instrucciones que faltan (para definir estás instrucciones se puede requerir más instrucciones nuevas).

a) Defina la instrucción riegue

b) Defina la instrucción agrupe

c) Defina las instrucciones recoja, coloque, suba y vuelvaorigen

d) En el problema se supuso que el máximo número de pitos en cada esquina era 6. Modifique el programa para que resuelva correctamente el problema para cualquier cantidad de pitos en cada esquina, suponiendo que el muro vertical finito que limita los pitos es de una altura mayor que el máximo número de pitos que hay en una esquina.

2.3.20. Karel se encuentra en un campo de práctica de 1 milla (8 bloques) de largo, con obstáculos de 1, 2 o 3 bloques de alto colocados aleatoriamente entre dos esquinas de la pista. Para ayudar a un amigo principiante, Karel desea indicar el tamaño de cada obstáculo. Programe a Karel para que coloque delante de cada obstáculo (al oeste del muro), tantos pitos como bloques de alto tenga el obstáculo. Karel comienza en el origen mirando al este con el número exacto de pitos que necesita colocar en su recorrido y debe terminar nuevamente en el origen, mirando al oeste con 0 pitos en la bolsa. A continuación, se presenta un posible estado inicial y el correspondiente estado final:

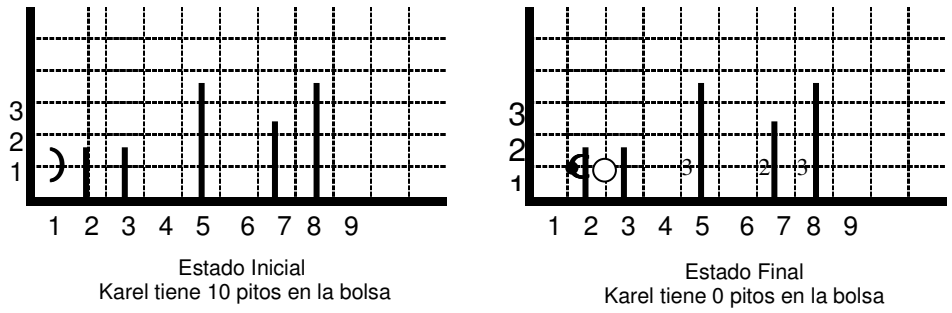


Figura 47. Estados inicial y final del problema de obstáculos

2.3.21. Karel es el portero de un edificio de oficinas. Entre sus funciones está la de lavar los parqueaderos que se encuentran en el sur de la calle principal del conjunto (la calle Alamos). Los parqueaderos se extienden verticalmente; cada

parqueadero tiene un bloque de ancho y cualquier longitud. Los parqueaderos están localizados a partir de la avenida 2. El final de los parqueaderos está marcado por un muro que bloquea la calle Alamos. Karel parte de la calle Alamos con avenida 1, mirando al este, con suficientes baldes de agua (pitos) en su bolsa. Prográmelo para que lave todos los parqueaderos (1 balde-pito por esquina de parqueadero) y termine frente al muro que rodea la calle Alamos, mirando al este. Una posible estado inicial y el correspondiente estado final se muestra a continuación.

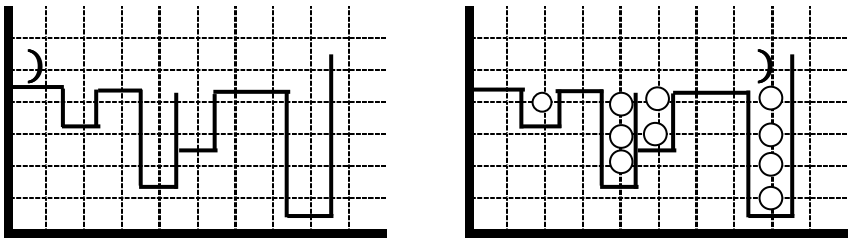


Figura 48. Estados inicial y final del problema "Lavar el parqueadero"

Note que no en todas las avenidas hay parqueaderos y que puede haber dos parqueaderos contiguos.

2.3.22. Programe a Karel para que recoja todas las flores de la montaña (escalera de muros). Karel comienza y termina en el origen, mirando al este con la bolsa vacía y debe terminar igual. A continuación, se presenta un posible estado inicial y su correspondiente estado final.

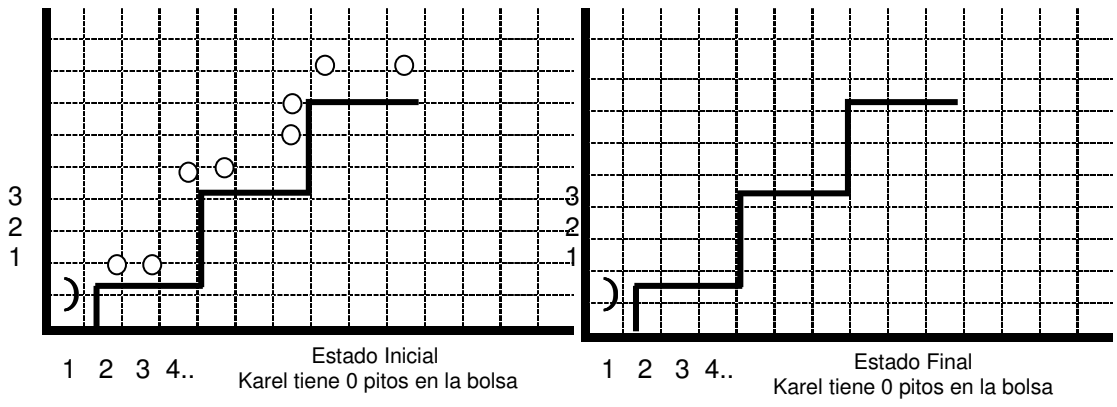


Figura 49. Estados inicial y final del problema "Recoger todas las flores"

2.3.23. En el mundo de Karel hay dos filas verticales de pitos, que parten de la calle 1 se extienden hacia el norte (1 pito por esquina). Karel parte del origen, mirando al este, con 1 pito en su bolsa y debe terminar sobre la base (calle) de la fila más larga de pitos, mirando al norte, con 1 pito en su bolsa. A continuación, se muestra un posible estado inicial y el correspondiente estado final.

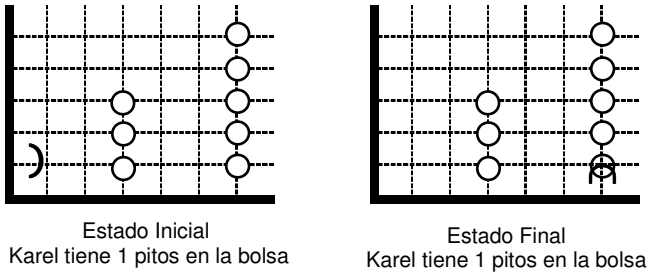


Figura 50. Estados inicial y final del problema "Identificar la fila más larga"

En los siguientes problemas se da la descripción de un estado inicial para Karel y un programa. Usted debe describir el estado final al que llega Karel después de ejecutar ese programa, si parte del estado inicial dado (recuerde que describe el estado final e incluye la descripción de todos los elementos del mundo). Si Karel termina por un error de ejecución, explique claramente la causa de ese error.

2.3.24. Karel se encuentra en la calle 1 con avenida 7 mirando al este con 0 pitos en la bolsa, frente a una montaña. A continuación, se presenta el estado inicial:

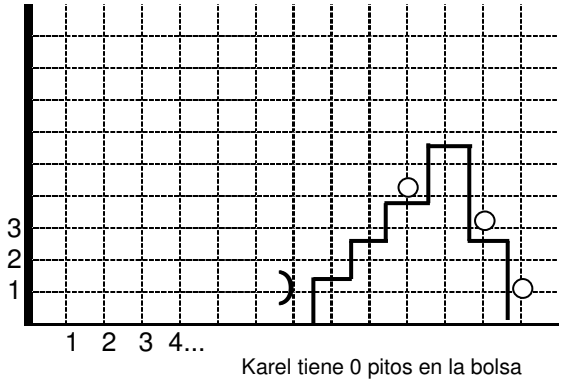


Figura 51. Estado inicial del problema propuesto

Karel debe ejecutar el siguiente programa:

```
class alpinista : Robot
{
  void turnRight();
}
void alpinista::turnRight()
{
  loop(3)
  {  turnLeft();
  }
}

task
{  alpinista karel(1,7,East,0);
  while (!nextToABeeper())
  {
    if (rightIsBlocked())
     {
        if (frontIsBlocked())
        {
             turnLeft();
        }
         else
              {
                   move();
              }
     }
       else
          {
            turnRight();
            move();
          }
  }
    turnOff();
}
```

Describa claramente el estado final al que llega Karel:

2.3.25. Karel se encuentra en el origen, mirando al este, con 15 pitos en su bolsa y ejecuta el siguiente programa:

```
class Desconocido: Robot
{
       void turnRight();
       void ponga();
       void escala();
}
void Desconocido::turnRight()
{
  loop(3)
  {  turnLeft();
  }
}
void Desconocido::ponga()
{
     if(anyBeepersInBeeperBag())
     {
          putBeeper();
      }
}
void Desconocido::escala()
{
     loop(3)
     {
          move();
          turnLeft();
          move();
          turnRight();
          ponga();
      }
}

task
```

```
{
    Desconocido karel(1,7,East,15);
   while(anyBeepersInBeeperBag())
   {
      karel.escala();
      karel.turnRight();
      karel.escala();
      karel.turnLeft();
   }
   karel.turnOff();
}
```

Describa claramente el estado final al que llega Karel:

2.3.26. En el mundo de Karel se tiene un sembrado rectangular de pitos formado por un número par de calles y cualquier cantidad de avenidas. El sembrado está cercado por muros y tiene una entrada de un bloque en la esquina más al este de la pared sur. En cada esquina del sembrado puede haber una mata pequeña (1 pitos), grande (2 pitos) o puede estar sin cultivar (0 pitos). Programe a Karel para que, partiendo de la esquina que queda frente a la entrada de sembrado, mirando al norte y con suficientes pitos en la bolsa, cultive el sembrado y vuelva a su posición inicial. Cultivar el sembrado significa regar las matas pequeñas (donde hay 1 pito deben quedar 2), talar las grandes (donde hay 2 pitos quitarlos) y sembrar (donde no hay pitos debe colocar uno). Fíjese que Karel no puede pasar dos veces por la misma esquina cultivando. Un estado inicial posible y el correspondiente estado final se muestran a continuación:

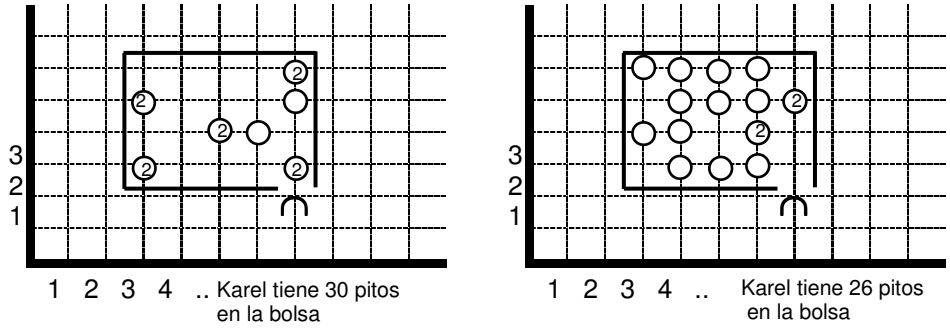


Figura 52. Condiciones iniciales y finales del problema “Sembrado”

2.3.27. Karel está de visita en una gran ciudad y se le ha encargado que ponga bombillos (pitos) en el techo de los 4 edificios de la ciudad. Un edificio está construido con tres muros: dos verticales y uno horizontal; puede ser de 1, 2 o 3 pisos y cubrir 1, 2 o 3 manzanas; los edificios están separados entre sí por una avenida (una sola esquina). Karel se encuentra en el origen mirando al este con un número suficiente de pitos. Prográmelo para que ponga un pito en cada esquina de la parte superior de cada edificio y termine después del último edificio, mirando al este. A continuación, se presenta un posible estado inicial y el correspondiente estado final.

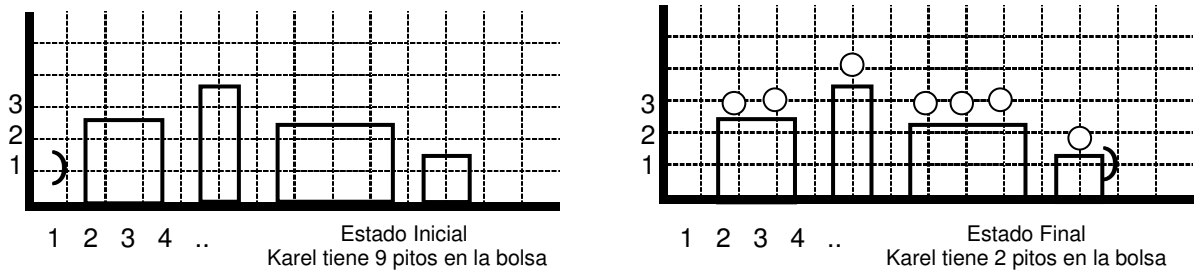


Figura 53. Estados inicial y final del problema “Ponga bombillos”

2.3.28. Karel encontró la pista que lo conduce al famoso tesoro de Morgan (varios pitos, más de uno); el pirata, al enterrar el tesoro, dejó un rastro de monedas (pitos) que parte del origen del mundo. Programe a Karel para que, partiendo del origen, mirando al este, siga el rastro de monedas y recoja el tesoro. El rastro de monedas es un camino; entre un pito del camino y el siguiente hay una calle o una avenida de diferencia; recorriendo el camino hacia delante, Karel sólo encuentra en cada paso un movimiento posible. A continuación, se presenta un posible estado inicial y el correspondiente estado final.

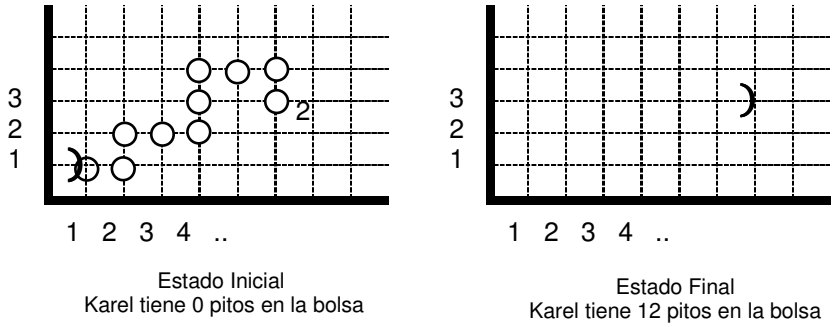


Figura 54. Condiciones iniciales y finales del problema "Tesoro de morgan"

2.3.29. Karel se encuentra en un cuarto cerrado (sin muros interiores), en el que se le ha perdido un pito. Programe a Karel para que localice el pito y lo recoja. El pito perdido se encuentra contra una pared del cuarto. A continuación se presenta un posible estado inicial y el correspondiente estado final.

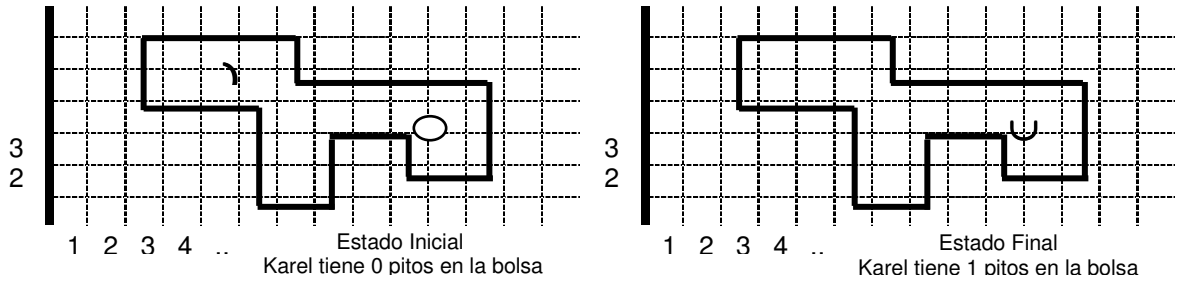


Figura 55. Condiciones iniciales y finales del problema "Cuarto cerrado"

2.3.30. Programe a Karel para que busque un pito (en un mundo sin muros) y coloque alrededor de éste, más pitos. Tenga en cuenta que el pito puede estar pegado a una de las dos paredes infinitas del mundo; Karel comienza con 8 pitos en su bolsa. Fíjese que al final Karel puede quedar en cualquier posición.

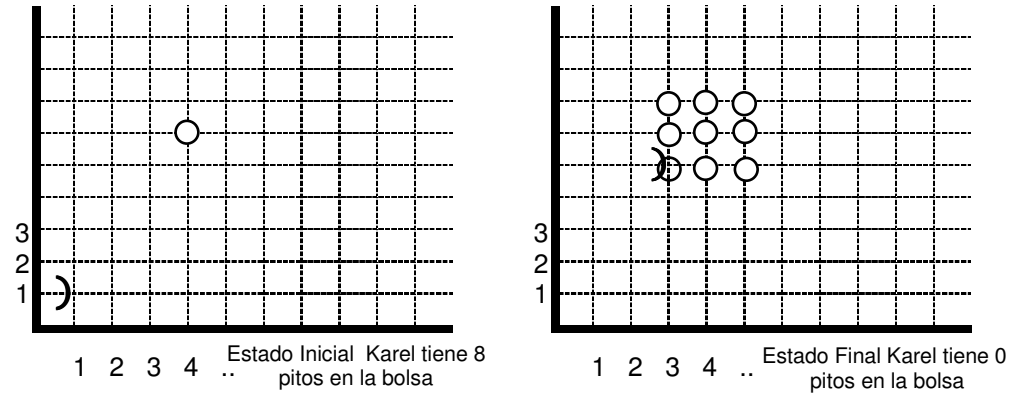


Figura 56. Condiciones iniciales y finales del problema "Buscar y rodearse"

2.3.31. Karel se encuentra en un túnel en forma de cruz. Los brazos del túnel son de un bloque de ancho y de longitud mayor que uno. Karel se encuentra en el centro del túnel mirando en cualquier dirección y desea colocar una marca, en este sitio (el centro) que indique por dónde es la salida y luego ir hasta la salida. Programe a Karel para que coloque en el centro del túnel un pito a la salida si está en el brazo oeste, dos si está en el brazo norte, tres si está en el

brazo este y cuatro si está por el lado norte y para que termine en la salida. A continuación, se presenta un posible estado inicial y el correspondiente estado final.

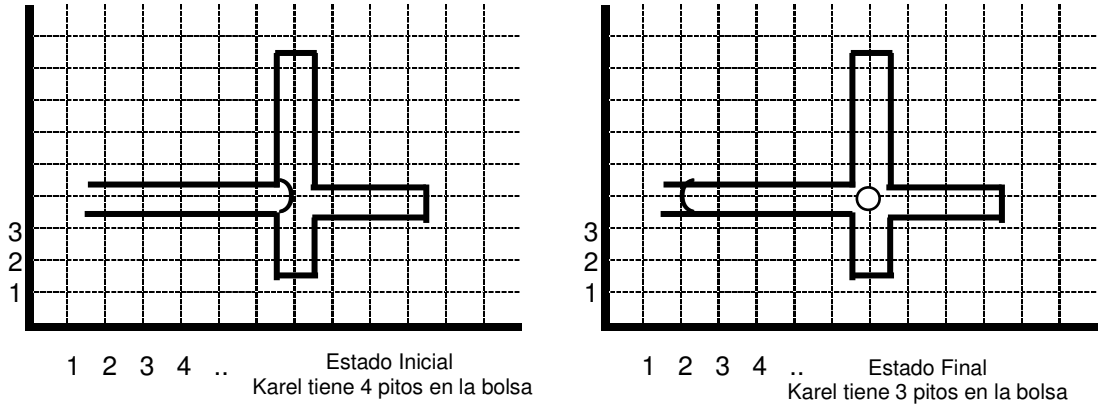


Figura 57. Condiciones iniciales y finales del problema "Túnel"

2.3.32. En el mundo de Karel hay una figura formada por calles de pitos. Las calles de pitos que conforman la figura comienzan a partir de la calle 1 hasta la calle F (arbitraria) y en cada calle los pitos están colocados uno detrás de otro empezando en la avenida 1 (mínimo un pito, puede estar totalmente llena). Entre las avenidas E (arbitraria) y E+1 hay un espejo (muro) que parte de la calle 1 y tiene longitud L (arbitraria) (donde L>=F). Haga un programa para que Karel refleje la imagen de la figura sobre el espejo. A continuación, se muestra un posible estado inicial con su correspondiente estado final. Karel parte del origen mirando al este y también termina en el origen mirando al este.

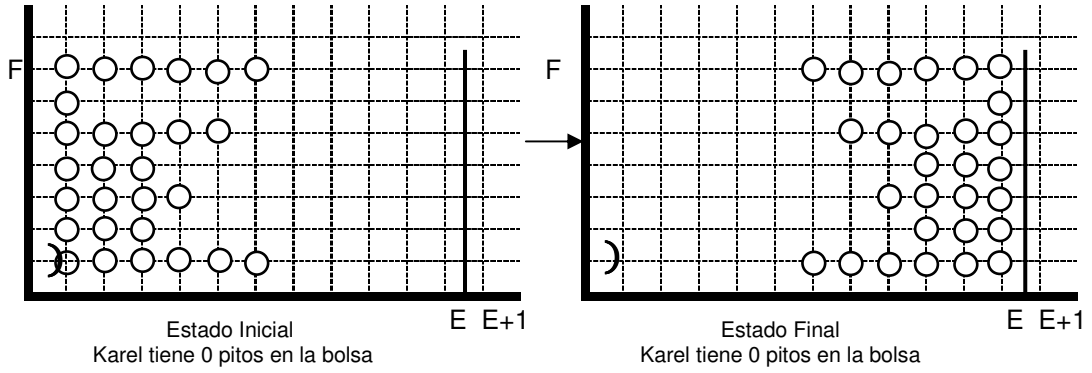


Figura 58. Condiciones iniciales y finales del problema "Imagen"

Plan general:

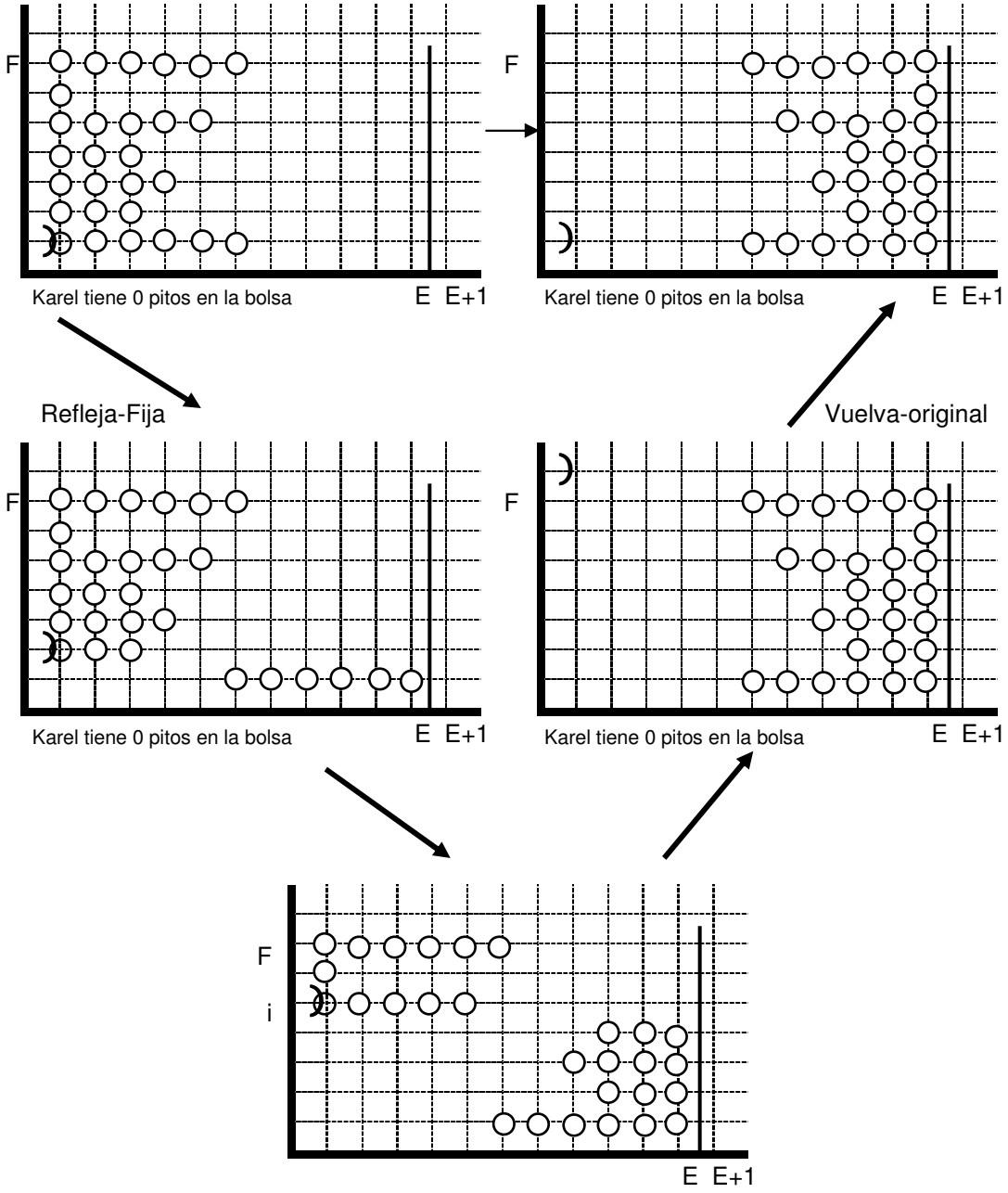


Figura 59. Invariante del problema "Imagen"

Resuelva el problema siguiendo el plan de solución presentado. Este plan muestra que en cada paso Karel debe reflejar una fila de la figura en el espejo (empezando con la fila que está sobre la calle 1). Después de reflejar la última fila que tenía la figura, Karel debe volver al origen.

2.3.33. Karel es un coleccionista de carros antiguos. Cada vez que llueve tiene el problema de guardar todos sus carros en los parqueaderos cubiertos de su

casa. Ayúdele a encontrar la mejor manera de guardar los carros dentro de los parqueaderos. Tenga en cuenta que no necesariamente hay parqueaderos para todos los carros (puede haber más carros que parqueaderos). Un carro lo vamos a representar como un pito y un parqueadero cubierto como una avenida del mundo que tiene muros en sus dos lados verticales. Note que la longitud de los dos muros determina la capacidad de los parqueaderos. A continuación, se da un posible estado inicial con sus correspondientes estados final. Karel siempre comienza en el origen mirando al este

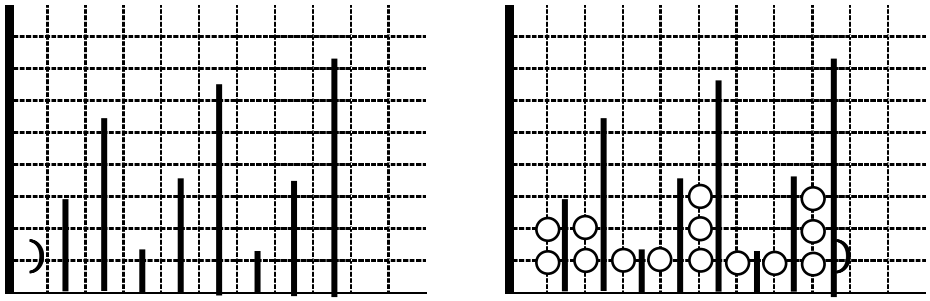


Figura 60. Condiciones iniciales y finales del problema "Coleccionista de carros"

3. TECNICAS AVANZADAS PARA ROBOTS

Objetivo: En este capítulo se presentará un número de tópicos bastante innovadores. Primero se presentará el concepto de recursión. Recursión vs procesos repetitivos que, permitirán que los robots ejecuten una secuencia de instrucciones más de una vez. También se establecerá la relación formal entre recursión y repetición. Se estudiarán dos nuevas instrucciones que darán a los robots la capacidad de solucionar algunos problemas de manipulación de pitos, incluyendo computación numérica. Se aplicará la nueva programación orientada a objetos y se verán algunas aplicaciones que serán aprendidas en este tema. Lo anterior puede parecer extraño al comienzo, pero después de explicar varios ejemplos se modificará la situación.

3.1. INTRODUCCIÓN A LA RECURSIÓN

Haciendo un análisis a los procesos repetitivos, ahora se examinará una nueva forma de que un robot repita una acción. La técnica se llama recursión. La palabra significa simplemente recurrir o repetir. Cuando un lenguaje de programación permite definiciones recursivas o recursión significa que un método puede llamarse a sí mismo durante la ejecución del robot. Lo anterior puede parecer extraño al comienzo, pero después de explicar varios ejemplos la confianza retornará como si fuera la ejecución de un robot en proceso repetitivo while. Se notará que la recursión es exactamente otra estructura de control como puede parecer inicialmente, se entenderá fácilmente como una repetición. Está es otra herramienta para adicionar a la colección. Existe el ejemplo de un robot que se mueve de un punto a otro, conociendo exactamente qué distancia se tiene que mover.

Hay muchas situaciones, sin embargo en la cual esto no es verdadero, por ejemplo: La condición inicial considera un robot llamado Kristin colocado en el origen (1,1) mirando al este y hay un pito en cualquier esquina a lo largo de la calle 1. El método deseable debe hacer que Kristin busque el pito, lo recoja y retorne al origen.

La instrucción de llamado al procedimiento sería:

```
Kristin.BuscaPito();
```

Si se sabe que el pito está entre la calle 1 y la calle 23 se escribiría un programa con procesos repetitivos. Se sabe que no hay más qué decir, ¿En un viaje de 15 bloques (bloques de 5 esquinas por ejemplo), se puede encontrar una solución? ¿Cómo? ¿Pero si no conocemos la distancia?. Pensemos inicialmente en la clase encontrar_pito().

```
class Encontrar_Pito: Robot
{
  void encuentraPito( );
  void regresa( );
  void Buscapito( );
}
 void Encontrar_Pito::BuscarPito ( ){
            EncuentraPito( );
                        pickBeeper( );
                        turnLeft( );
                        turnLeft( );
                        Retornar( );
            }
```

Esto efectivamente hace la tarea, se tendrá que escribir otros dos nuevos métodos EncuentraPito() y Retornar(). Ahora se analizará el procedimiento EncuentraPito( ). Se sabe que hay un pito en alguna de las esquinas sobre la calle 1. También se sabe que Kristin está sobre la calle 1 mirando al Este.

Puede ser que Kristin esté ya sobre el pito en una esquina, en este caso no hay nada qué hacer. Sin embargo, se puede comenzar a escribir:

```
void EncuentraPito()
{       if (! nextToABeeper())
        {
                ...
        }
}
```

Terminará correctamente en el caso de que Kristin esté sobre un pito en la esquina. Supóngase que el pito está sobre alguna otra esquina, entonces Kristin necesitará moverse rumbo a otra y comprobar otras esquinas.

```
  void EncuentraPito()
{       if (! NextToABeeper())
        {       move();
                ???
        }
}
```

Bien, ¿Qué necesita hacer Kristin después de moverse? Note que Kristin está en una esquina diferente a la esquina original de donde partió, comprobó y encontró ausencia de pitos. Por consiguiente, se puede concluir que un pito está ahora (después de moverse) ya sea en la localización actual de Kristin o más hacia el este de él. Pero ésta es exactamente la misma situación (relativamente) en la que Kristin está cuando comenzó está tarea (condición inicial). Por consiguiente, se concluye que lo que Kristin necesita hacer ahora exactamente es EncontrarPito() una vez más.

```
void EncuentrePito()
{       if (! nextToABeeper())
        {       move();
                EncuentrePito();
        }
}
```

Obsérvese que EncuentrePito() ha sido definido completamente, pero ha sido definido llamando además es EncuentrePito() dentro de su procedimiento. ¿Qué puede estar haciendo? Bien, supóngase que necesitamos vaciar un océano con un balde. Al ejecutar el procedimiento OceanoVacío(), inicialmente se pregunta si el océano está vacío. Si es así está hecho el trabajo, en caso contrario, se saca un balde de agua justamente del océano y llamamos el procedimiento Oceanovacío().

Es importante pensar que semejante definición recibe el nombre de definición recursiva, y se define un proceso en sus propios términos cada vez menos complejo. Se define un proceso en términos de algo más simple o una versión más pequeña de sí misma. Aquí se define EncuentraPito() en cualquier caso sea que Kristin esté ya en la esquina del pito o no; se usa move(); y EncuentraPito() así Kristin este en cualquier lugar. Es necesario conocer también dónde se ejecuta tal instrucción y dónde efectivamente se encuentra un pito en el camino de karel, de otra manera después de todo movimiento la reejecución de EncuentraPito() encontrará que la prueba es verdadera, generando sin embargo otra reejecución de EncuentraPito() a menos que termine.

El otro método retorna el procedimiento regresar() en forma similar, excepto para la prueba. Aquí sin embargo se conoce que hay un muro al oeste. Si se garantiza que el robot está mirando al Oeste, el siguiente método es válido.

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

La programación con recursión es muy potente y algunas veces está sujeta a error.

3.2. AMPLIACIÓN A LA RECURSIÓN

Dado el problema, un robot tiene que limpiar todos los pitos de cualquier esquina, se puede fácilmente escribir un proceso repetitivo como aparece a continuación:

```
class Aspiradora: Robot
{
  void LimpiaEsquina();
}
  void  LimpiaEsquina()
            {           while ( nextToABeeper() )
                        {           pickBeeper();
                        }
            }
            ...
}
```

Esto soluciona correctamente el problema y se conoce como solución iterativa ("iterativa" significa una repetición de algún tipo, while or loop, explicada en el capitulo anterior). Contrasta la anterior solución con la siguiente solución recursiva.

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

La diferencia entre estos dos métodos es muy sutil. El primer método, que usa el repetitivo while, es llamado una vez y nunca el foco del robot sale del proceso repetitivo hasta que haya finalizado, recogiendo cero o más pitos (depende del número inicial en la esquina).¿Qué pasa en la función recursiva, LimpiaEsquina()? Se analizará cuidadosamente.

Si el robot está inicialmente en una esquina vacía cuando el mensaje es enviado, no pasa nada (similar al repetitivo con while). Si es una esquina con un pito cuando el mensaje es enviado, La condición del if es verdadera, así que el robot ejecuta una instrucción pickBeeper() (vacía la esquina). Entonces hace un segundo llamado de LimpiaEsquina(), teniendo en cuenta donde fue la primera llamada. Cuando LimpiaEsquina() es llamado la segunda vez, la condición de if es falsa, no ocurre nada y el robot retorna al primer llamado.

[Invocación Inicial]

```
void LimpiaEsquina()
{       if ( nextToABeeper() )
        {       pickBeeper();
                LimpiaEsquina(); // <--  Este es la segunda llamada de LimpiaEsquina.
        }                                       /. El robot retorna a este punto cuando
}                                               // el llamado finaliza la ejecución.
```

Segunda Invocación

```
void LimpiaEsquina()
{       if ( nextToABeeper() )          // <-- Esto es ahora falso
        {       pickBeeper();
                LimpiaEsquina();
        }
}
```

Cada llamado resulta en invocaciones separadas (Ejemplarizacion) de la instrucción LimpiaEsquina(). El robot puede completamente ejecutar cada ejemplo, siempre recordando dónde fue el ejemplo previo, así que retorna allí cuando finaliza.

El proceso para escribir instrucciones del robot recursivas, es muy similar a escribir procesos repetitivos.

Fase 1: Consideremos la condición de parada (Llamado también caso base)-- ¿Cuál es el caso más simple en que el problema puede ser solucionado? En el problema de LimpiarEsquina(), el caso más simple, o base, es el caso cuando ya está la esquina vacía.

Fase 2: ¿Qué tiene que hacer el robot en el caso base? En este ejemplo no tiene qué hacer nada.

Fase 3: Encontrar la forma de solucionar una parte sencilla del problema completo que no es el caso base. Esto es llamado "reducción del problema en el caso general ". En el problema de LimpiaEsquina(), El caso general es cuando el robot está en la esquina con uno o más pitos y la reducción es recoger un pito.

Fase 4: Estar seguro que la reducción guía es caso base. Una vez más en el ejemplo anterior del LimpiaEsquina(), por elegir un pito a la vez, el robot puede eventualmente limpiar la esquina de pitos, independiente del número originalmente presente.

Comparación y contraste entre iteración y recursión.

Un proceso repetitivo iterativo completa cada iteración antes de comenzar la siguiente.

Un método tipo recursivo comienza una nueva instancia antes de completar el que rige en el momento. En el momento que ocurre la instancia actual es temporalmente suspendida, en espera de completar una nueva instancia.

Desde luego, está nueva instancia puede no completarse antes de generar propiamente otra. Cada instancia sucesiva tiene que ser completa y a su vez la última debe ser la primera instancia.

Aunque cada instancia recursiva se supone hace algo (frecuentemente mínimo) hacia adelante en el caso base, no se usan repeticiones para controlar los

llamados recursivos. Así, generalmente se ve un if o un if/else en el cuerpo del nuevo método recursivo, pero no un while.

Se propone utilizar recursión para mover un robot denominado Karel hacia un pito. ¿Cómo puede hacerse esto? Sigamos las fases presentadas anteriormente:

¿Cuál es el caso base?. Karel está sobre el pito.
¿Qué tiene que hacer el Robot en el caso base? nada.
¿Cuál es el caso general? El robot no está sobre el pito.
¿Cuál es la reducción? Moverse hacia el pito y hacer un llamado recursivo.
¿La reducción conduce hacia la terminación? Sí, se asume que el pito está directamente enfrente del robot, la distancia es mínima de un bloque en cada llamada recursiva.

La implementación final será:

```
void EncuentraPito()
{       if ( ! nextToABeeper() )
        {       move();
                EncuentraPito();
        }
}
```

Note que este problema puede también ser solucionado fácilmente, usando la instrucción repetitiva while. Analicemos un problema que no es fácil de solucionar, usando la instrucción repetitiva while. Consideremos una mina de pitos perdidos, ¿La esquina con un número grande de pitos? Imagínese escribir el siguiente método en una búsqueda por la mina. Un robot llamado karel camina hacia el oriente, desde una localización actual hasta encontrar los pitos. La mina de pitos perdidos está justamente al norte de la intersección, a una distancia igual al número de movimientos que karel hace para llegar a una posición actual de un pito. Se escribirá el nuevo método EncontrarMina().

No es fácil solucionar el problema con la instrucción repetitiva porque no tenemos alguna forma apropiada de recordar cuántas intersecciones han sido recorridas. Puede aparecer probablemente un esquema de rastreo de pitos muy complejo, se analizará una solución recursiva bastante directa. Una vez más las respuestas a las preguntas.

¿Cuál es el caso base? Karel está sobre el pito.
¿Qué tiene que hacer Karel en el caso base? TurnLeft() (Ahora karel estará orientado hacia el Norte).
¿Cuál es el caso general? Karel no está sobre un pito.
¿Cuál es la reducción? Mover hacia adelante un bloque, haciendo un llamado recursivo y karel ejecuta un segundo movimiento después de un llamado recursivo. Este segundo movimiento será ejecutado en todas las instancias, pero en el caso base, motivará que karel haga muchos movimientos hacia el norte, después del caso base hasta llegar nuevamente a obtener el caso base.

¿La reducción produce la terminación? Si, se asume que el pito está directamente enfrente de Karel.

El método completo queda así:

```
void  EncontrarMina()
{           if ( nextToABeeper() )
            {           turnLeft();
            }
            else
            {           move();
                        EncontrarMina();
                        move();
            }
}
```

¿Cuántos turnLeft() son ejecutados? ¿Cuántos movimientos? ¿Cuántos llamados a EncuentraMina()?

Una forma de pensar en una solución recursiva y sus respectivos llamados, en términos de la especificación del propio método se explicrá a continuación: En este caso, la especificación es que cuando un robot ejecuta está instrucción caminará determinado números de etapas, hasta encontrar un pito. Supongase k etapas, giro a la izquierda y caminará k etapas adicionales. Adicionalmente se supone que inicialmente el robot viaja N etapas hasta llegar al pito (k = N). Cuando examinamos el método anterior, la cláusula else tiene primero un mensaje de **move()**. Esto significa que el robot está ahora a N-1 pasos del pito, por consiguiente para la especificación, el llamado recursivo (k = N-1) camina N-1 etapas propicia a la izquierda (turnLeft()), y camina N-1 paso más allá. Por consiguiente, se necesita proveer un mensaje de movimiento adicional (**move()**) después la recursión completa los N pasos.

Se tratará de solucionar un número de problemas y un buen paso de mirar la recursión que viene a ser un uso confortable de la iteración. Gran parte es debido a que la recursión requiere alguna intuición para ver la reducción correctamente, especialmente en problemas difíciles. Está inducción viene con la práctica, porque los problemas sencillos están diseñados en forma fácil.

3.3. RECURSIÓN DE COLA Y REPETICIÓN

Supóngase que se tiene un proceso repetitivo con while y deseamos reescribir el proceso no repetitivo, pero que haga la misma tarea. Consideremos el caso más sencillo, supóngase que el mecanismo de control más extremo en un método repetitivo. Tomemos como ejemplo el siguiente método:

```
void EncuentraPito()
{           while ( ! nextToABeeper())
            {           move();
            }
```

```
}
```

Por definición de la instrucción while lo anterior es lo mismo que:

```
void EncuentraPito()
{       if ( ! nextToABeeper())
        {       move();
                while ( ! nextToABeeper())
                {       move();          } EncuentraPito( )
                }
        }
}
```

Sin embargo, la instrucción while anidada en la última forma es exactamente el cuerpo de EncuentraPito() usando la definición inicial de EncuentraPito(), así que reescribiendo la segunda forma quedará:

```
void EncuentraPito()
{       if ( ! nextToABeeper())
        {       move();
                EncuentraPito();
        }
}
```

Así, que la primera forma, un while, es equivalente a la última forma en un programa recursivo. También notemos que se puede exactamente transformar el segundo en el primero, dado que son instrucciones equivalentes.

Se percibe finalmente que está es una forma especial de recursión, aunque después del paso de la recursión (EncuentraPito() anidado) no hay nada más que hacer en este método y por consiguiente hay que observar que las instrucciones del while son las mismas a las instrucciones del if. Está forma de recursión es llamada recursión de cola, puesto que el paso recursivo viene propiamente de la cola de la computadora.

Es posible probar formalmente que la recursión de cola es equivalente a un proceso repetitivo con while. Por lo tanto, se puede ver que un proceso repetitivo con while es exactamente una forma especial de recursión. Así nada puede hacer con un proceso repetitivo, usted, puede hacer un programa recursivo. Existen algunas teorías profundas en ciencia computacional (en matemática) que han sido desarrolladas para está observación especialmente en los lenguajes funcionales tales como CAML[1].

Existe una forma de contraste en el método de solución de EncuentraMina(), discutida anteriormente y que ciertamente fue recursiva, pero no recursiva de cola, porque el mensaje final **move()** antecede un mensaje recursivo.

3.4. BÚSQUEDA

Está sección introduce dos nuevos métodos llamados zigIzqUp() and zagDerDown(), que mueve un robot diagonalmente al noroeste y sureste respectivamente. Ambos de estos métodos están definidos usando sólo los métodos de ur_robot() como turnLeft(), pero se deriva un inmenso potencial conceptual con capacidad para pensar en términos de moverlo diagonalmente. Por razones que no llegan a ser claras en el ejercicio, la clase dentro de la cual colocamos estos métodos es matemático. Tenemos la clase robot como la clase padre.

La siguiente definición introduce clave de está sección zigIzqUp() and zagDerDown(). Este par de direcciones no son arbitrarias; si un robot se mueve a la izquierda y hacia arriba bastante tiempo, esto eventualmente alcanza el límite de la pared occidental. El mismo argumento posee el viaje hacia abajo y hacia la derecha, excepto que en este caso el robot eventualmente busca el límite de la pared sur.

[1] *www.inria.fr*

Las otras dos posibles direcciones carecen de propiedades útiles, un robot nunca encuentra paredes límites al viajar arriba y hacia la derecha, y no está seguro a cuál de las dos paredes límites llega primero, cuando viaja hacia abajo y a la izquierda.

El siguiente método define zigIzqUp() and zagDerDown().

```
class Matemático: Robot
{
void zigIzqUp( );
void zagDerDown( );

}
void Matemático:: zigIzqUp ( )
{           // Precondición: Orientado al oeste y el frente despejado
// Postcondición: Orientado al oeste
                        move( );
                        turnRight( );
                        move( );
                        turnLeft( );
            }

void Matemático:: zagDerDown( )
{           // Precondición: Orientado al sur y el frente despejado
// Postcondición: Orientado al sur
                        move( );
                        turnLeft( );
                        move( );
                        turnRight( );
            }
            ...
}
```

Se asume que se tiene un karel llamado Matemático. Obsérvese que no se parte de esos métodos de Heurísticos para moverse en una dirección dada. Para ejecutar zigIzqUp() correctamente, karel debe estar mirando al oeste; para ejecutar zagDerDown() correctamente, Karel tiene que estar mirando al sur.

Estos requisitos son llamados las precondiciones de los métodos. Se recuerda que una precondición de un método tiene que ser verdadera antes que el robot pueda correctamente ejecutar el método.

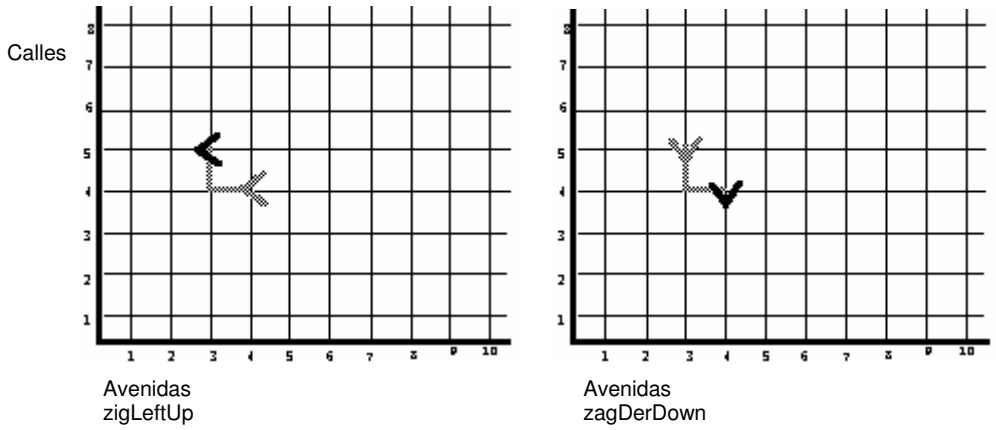


Figura 61. Precondicones para los Métodos ZipLeftUp() y zagDerDown()

Para este ejemplo la precondición direccional de zigIzqUp() es que karel esté mirando al Oeste; igualmente la precondición de zagDerDown() es que karel esté mirando al sur. Karel ejecuta estos métodos, cuando su precondición es satisfecha, como se ilustra en la figura 61.

Aquí hay una instrucción que es llena de terminología: La precondición direccional de zigIzqUp() y zagDerDown() es un invariante de cada instrucción ejecutada. Esto exactamente significa que si karel está mirando al oeste y ejecuta zigIzqUp(), el robot está aún mirando al oeste después que la instrucción ha terminado de ejecutarse. Está propiedad permite a karel ejecutar una secuencia de zigIzqUp() sin tener que restablecer su precondición direccional. Una instrucción similar mantiene a karel casi mirando al este para zagDerDown(). También observe que cada instrucción tiene que ser ejecutada sólo cuando karel encuentra el frente despejado. Está precondición no es un invariante de la instrucción, porque Karel tiene que estar a un bloque de la esquina donde el frente está bloqueado (e.g., Karel puede ejecutar zigIzqUp() mientras mira al oeste en la calle 4 y avenida 2).

El primer método principal que se escribe soluciona el problema de encontrar un pito que puede ser localizado en cualquier parte del micromundo, la tarea es escribir un método llamado EncuentraPito() en la que karel se posiciona en la misma esquina del pito. Existe una versión de este problema donde tanto karel como el pito están encerrados en un cuarto. Está nueva formulación tiene menos restricciones rigurosas. El pito está localizado en alguna esquina de la calle del micromundo de karel, y no hay muros en el micromundo. Por supuesto, los muros limitantes están siempre presentes.

Una sencilla solución puede emanar de su mente. En este ensayo, karel inicialmente está en el origen y mirando al Este. El robot entonces se mueve hacia el este por la calle primera ubicando un pito. Si karel encuentra un pito en la calle 1º, él ha culminado la tarea; si el pito no es encontrado en la calle 1º, karel se mueve para atrás hacia la pared del oeste, conmutando sobre la calle 2º, y continúa buscando a partir de allí. Karel repite está estrategia hasta encontrar el pito. Desafortunadamente existe una confusión que está implícita en la instrucción de búsqueda. No hay forma para que karel conozca que un pito no está en la calle 1º. No tiene importancia cuánto explora karel en la calle 1º, el robot nunca puede estar seguro que un pito no está más de un bloque hacia el este.

La percepción es como si karel y nosotros estuviéramos atrapados en una trampa imposible, pero existe una solución ingeniosa de nuestro problema.

Se puede prever la implicación de movimientos de zig-zag. Se necesita programar a karel para ejecutar radicalmente un tipo diferente de patrón de búsqueda; el procedimientos EncuentraPito() muestra un patrón de búsqueda.

```
void  EncuentraPito( )
{           VayalOrigen( );
            MirandoOeste( );
            while (  ! nextToABeeper( ) )
            {           if ( facingWest( ) )
                        {           zigMove( );
                        }
```

```
                    else
                    {       zagMove( );
                    }
            }
}
```

Este método de búsqueda aumenta la frontera de la búsqueda, similar a la forma como el agua puede fluir sobre el mundo de karel como si fuera una fuente desbordante en el origen. A grandes rasgos se puede ver a karel viajar de acá para allá diagonalmente en el margen de está ola de agua. Se debe estar seguro que es un patrón de búsqueda garantizado para encontrar un pito eventualmente, independiente de las localizaciones de los pitos en nuestra analogía, se necesita convencernos que el pito eventualmente se encuentra húmedo (ver figura 62). Se puede utilizar refinamiento paso a paso para escribir el método EncuentraPito() utilizando la estrategia de búsqueda con los métodos zigIzqUp() and zagDerDown().

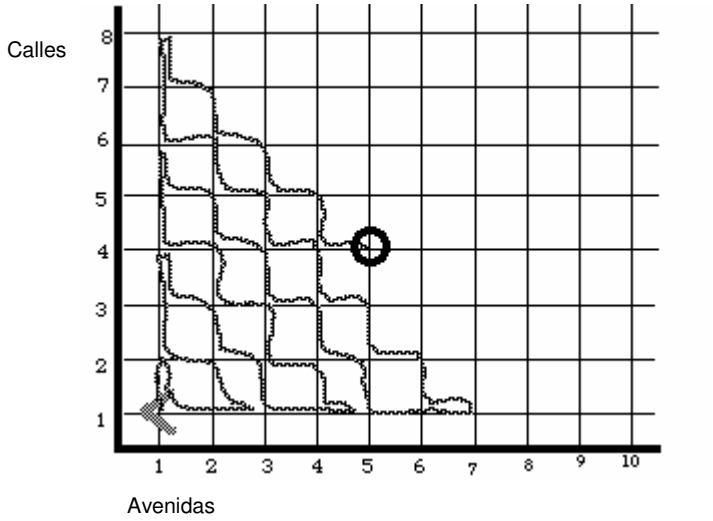


Figura 62. Aumento de la frontera de la búsqueda

El método EncuentraPito() comienza moviendo a karel al origen y mirando al oeste. (Decimos cómo escribir la instrucción mirando al oeste). Estos mensajes establecen la precondición direccional para zigIzqUp(). El ciclo while propuesto propone mover a Karel hasta encontrar un pito, y es correcto si el ciclo repetitivo eventualmente termina. La condición if, el cual es anidado en el cuerpo del ciclo, determinando en qué dirección karel ha girado y continúa

moviéndose a lo largo de la diagonal en la misma dirección. Se continúa refinando paso a paso, escribiendo los métodos zigMove() y zagMove().

```
void  zigMove()
{           // Precondición: Orientado al Oeste
            if ( frontIsClear( ) )
            {           zigIzqUp( );
            }
            else
            {           avanceProxDiagonal( );
            }
}
```

```
void  zagMove( )
{           // Precondición: Orientado al sur
            if ( frontIsClear( ) )
            {           zagDerDown( );
            }
            else
            {           avanceProxDiagonal( );
            }
}
```

Los métodos de desplazamiento zigMove() y zagMove() operan similarmente; sin embargo, se discutirá sólo zigMove(). El método zigMove() mueble robot diagonalmente hasta próxima esquina. De otra forma, el robot es bloqueado por el muro límite occidental y puede avanzar hacia el norte hasta la próxima diagonal. Ahora se escribirá el método que avanza karel hasta la próxima diagonal.

```
void  avanceProxDiagonal( )
 {          if ( facingWest( ) )
             {          facingNorth( );
             }
            else
             {          facingEast( );
```

```
            }
            move( );
            turnAround( );
 }
```

El método avanceProxDiagonal() comienza con Karel orientado hacia fuera del origen; girando en una dirección diferente, dependiendo tanto del robot como del zig o zag. En ambos casos, Karel se mueve en una esquina en forma más rápida del origen y gira al rededor. Si karel ha hecho zig en la diagonal actual, después de ejecutar avanceProxDiagonal(), el robot es posicionado para continuar con zag en la próxima diagonal, y viceversa.

Obsérvese que cuando karel ejecuta un método de zigIzqUp() o un método de zagDerDown(), tiene que visitar dos esquinas; la primera es visitada temporalmente, y la segunda es una esquina diagonal a la esquina del inicio de karel. Cuando se piensa acerca de estos métodos, se debe ignorar la esquina intermedia y exactamente recordar que está instrucción mueve diagonalmente a karel. También se advierte que una visita temporal a la esquina es garantía de no tener un pito, porque es parte de una oleada frontal que karel visita mientras estuvo pasando por la anterior diagonal.

La ejecución de EncuentraPito() por Karel, ilustra en la situación presentada anteriormente y lo conecta con la operación. Intente obtener una retribución de cómo todas estás instrucciones, se ajustan en conjunto para acoplar la tarea particularmente y así estará cerrada la atención al método avanceProxDiagonal(). Implemente EncuentraPito() en la situación donde el pito está en el origen y la situación donde el pito está próximo al límite del muro.

3.6 HACIENDO ARITMÉTICA

Una de las cosas que los computadores hacen bien es manipular los números. Los robots pueden estar enseñando aritmética se notará a continuación: Una forma de representar números en el mundo del robot es el uso de los pitos. Se

puede representar el número 32 colocando 32 pitos en una esquina, pero puede presentarse más sofisticado. Supóngase que se representan los diferentes dígitos de un número multidígitos separadamente. Consecuentemente para representar el número 5132 se puede utilizar la 2º calle como un ejemplo de un lugar para colocar el número, se colocan 5 pitos en la calle 2º , avenida 1º, 1 pito en la 2º con 2º , 3 pitos en la 2º con 3º , y 2 pitos en la 2º con 4º. Se puede escribir en la columna completa el número como se ilustra a continuación:

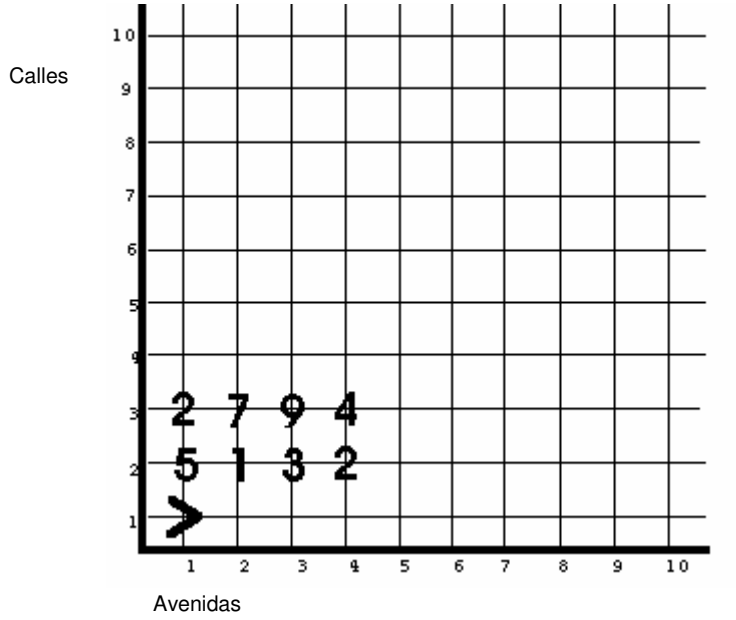


Figura 63 Abstracción de sumar en el mundo de Karel

Se comienza con un simple caso, y exactamente se adiciona una columna de números de simples dígitos. Veamos la figura siguiente como un ejemplo. Supóngase que iniciamos con un robot sumador en la calle 1º con una columna de números representados por pitos al norte de este. En cada esquina habrá entre 1 y 9 pitos. Esperamos escribir en la calle 1º el número decimal que representa la suma de la columna. Entonces se puede "llevar" si el número total de pitos es más de 9 se asume que no inicia en la avenida 1º.

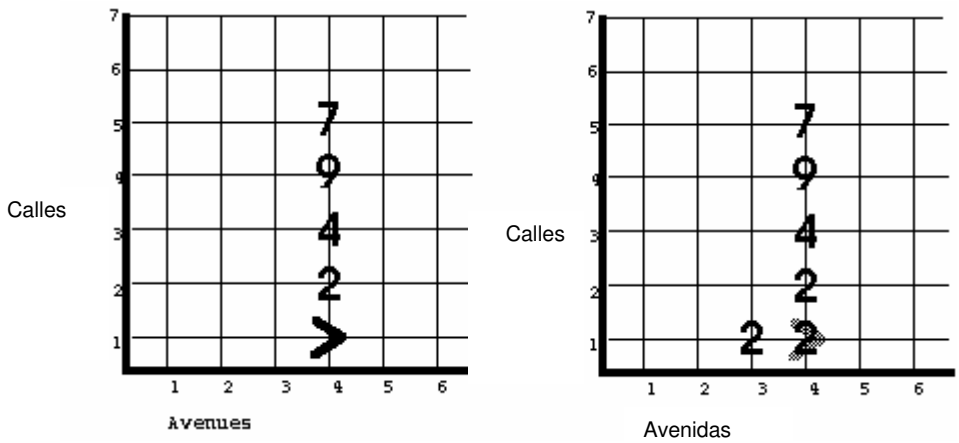


Figura 64 Invariante del problema sumar.

Nuestro robot sumador utiliza dos robots asistentes. El primero de estos chequea para ver si una lleva es necesaria y el segundo actualmente hace el lleva si es necesario. Estos robots necesitan ser creados y encontrar el robot sumador que los crea a ellos, se iniciará con una super clase(padre) que provee este método de encuentro. De todas las otras clases para este problema se derivan subclases de está clase base; Descubridor. Actualmente no creamos cualquier robot descubridor, sin embargo está clase es exactamente un método de lugar apropiado que puede ser común a otras clases.

```
class Finder : Robot
{
        void turnAround( ){...}
        void turnRight( ){...}

        void moveToRobot()   // Robot directamente  a la cabeza.
        {       while( ! nextToaRobot() )
                {       move();
                }
        }
        ...
}
```

```
class Carrier extends Finder
{
        void carryOne(){...}   // lleva"uno" para la próxima columna
}

class Checker extends Finder
{
        boolean enoughToCarry(){...}   // Requiere carrear significativamente
        . . .
}

class Adder extends Finder   // Siempre crea en la calle 1 mirando al norte
{
        Carrier Carry = new Carrier(1,1, East, infinity);
        Checker Check = new Checker(1,1, East, 0);

        void gatherHelpers(){...}
        void addColumn(){...}
        . . .
}

void gatherHelpers() // Adder class
{       Carry.findRobot( );
        Carry.turnLeft( );
        Check.findRobot( );
        Check.turnLeft();
}
```

Una vez que el robot sumador ha creado sus asistentes, se puede ejecutar el procedimiento addColumn(). Hacer esto es simplemente recoger todos los pitos nortes hasta encontrar una esquina vacía, retorna a la calle 1º, depositar todos los pitos allí y luego los asistentes terminarán la tarea.

```
void addColumn( ) // Orientado al  norte sobre la primera calle
{       move( );
        while(nextToABeeper( )
        {       pickBeeper( );
                if ( ! nextToABeeper( ) )
```

```
                    {       move( );
                    }
            }
            turnAround( );
            while( frontIsClear( ) )
            {       move( );
            }
            turnAround( );
            while ( anyBeepersInBeeperBag( ) )
            {       putbeeper( );
            }
            // Computa el cociente y el residuo.
            while( Check.enoughToCarry( )
            {       Carry.carryOne( );
            }
}
```

El acarreador es considerablemente simple. Advierte que administrar al acarreador, un número infinito de pitos en la bolsa así que no es posible que salga corriendo.

```
void carryOne( )  // Facing north on 1st Street. -- Carrier class
{       turnLeft( );
        move( );    // Nota:  Error: intento de salirse si trata de acarrear.
                            // desde la calle 1
        putBeeper( );
        turnAround( );
        move( );
        turnLeft( );
}
```

El robot verificador hace todo el trabajo interesante. Se tiene que determinar si hay 10 o más pitos en la esquina actual. Si hay, retorna verdadero, de otra manera falso. Se puede tantear para recoger 10 pitos para chequear esto. Sin embargo, se tiene que hacer más desde tener que llamar repetitivamente para ver cuántos múltiplos de 10 realmente hay. Sin embargo, se tiene que encontrar una forma de disponer de cada grupo de 1 pito antes de intentar para comprobar el próximo grupo de 10. Otra importante tarea adicional es encontrar al menos 10 pitos en un grupo que puedan salir de una esquina actual. Esto es llevar la cuenta para el primer dígito de la respuesta; una que no se acarrea.

```
boolean enoughToCarry( ) // Orientado al Norte sobre la primera calle. – Clase Checker
{       loop(10)
        {       if (nextToABeeper( ) )
                {       pickBeeper( );
                }
                else
                {       emptyBag( );
                        return false;
                }
        }
        // Se tienen que encontrar 10 pitos.
        move( );
        emptyBag( ); // lSalirse en la calle 2.
        turnAround( );
        move( );
        turnAround( );
        return true;
}
```

Para finalizar se necesita sólo adicionar el procedimiento emptyBag() en la clase Checker :

```
void emptyBag( )
{       while( anyBeepersInBeeperBag( ) )
        {       putBeeper( );
        }
}
```

Ahora estamos listos para afrontar el problema de la suma multicolumna. Observe que si se inicia a la derecha al final del valor de fila y exactamente en la trasparencia del sumador y de sus asistentes a la izquierda, después de adicionar una columna, los tres robots se posesionan en la próxima columna. Entonces se trabaja de derecha izquierda, computando la suma correcta porque acarreamos en la columna antes de que está columna sea sumada.

Se necesitan dos métodos adicionales en la clase sumador: slideLeft() y addAll( ). El método slideLeft( ) es fácil.

```
void slideLeft( )

{       turnLeft( );
```

```
        move( );
        turnright( );
        carrier.turnLeft( );
        carrier.move( );
        carrier.turnRight( );
        checker.turnLeft( );
        checker.move( );
        checker.turnRight( );
}
```

¿Cómo addAll() conoce cuando se hace adición de columnas? Una forma es tener el número más a la izquierda iniciando él la segunda avenida, así que hay un salón para llevar cualquier valor de la columna más a la izquierda. El sumador entonces necesita comprobar para ver si en la segunda avenida antes de la adición. Esto requiere un nuevo predicado.

```
boolean onSecondAve( )  // Precondición:  Orientado al norte y no sobre la primera avenida.
{       turnLeft( );
        move( );
        if (frontIsClear)
        {       turnAround( );
                move( );
                turnLeft( );
                return False;
        }
        turnAround( );
        move( );
        turnLeft( );
        return True;
}
```

Se está ahora listo para escribir el método addAll() en la clase sumador.

```
void addAll( )
{       while( ! onSecondAve( ) )
        {       addColumn( );
                slideLeft( );
        }
```

```
        addColumn( );
}
```

3.7 POLIMORFISMO- ¿POR QUÉ ESCRIBIR MUCHOS PROGRAMAS CUANDO HAY QUE HACER UNO?

Polimorfismo significa literalmente en griego "muchas formas". En programación orientada objetos se refiere al hecho de que los mensajes enviados por objetos (robots) pueden ser interpretados diferente, y dependiendo de la clase de objeto (robot) que recibe el mensaje. La mejor forma de pensar en esto es recordar que el robot es autónomo en el micromundo. Se envía un mensaje y éste responde. Esto no llega a ser una unidad de control remoto para que el usuario directamente accione. Preferentemente, éste "Escucha" el mensaje enviado y éste responde de acuerdo con un diccionario interno. Recuerde que cada robot consulta su propio diccionario de instrucciones para seleccionar el método que utiliza para responder cualquier mensaje. La nueva versión de cualquier método, entonces, cambia el significado del mensaje para robots de la nueva clase.

Para ilustrar la consecuencia de esto, nos apropiamos de algo dramático, sin embargo, no es un ejemplo útil. Supóngase que tenemos las siguientes dos clases:

```
class Putter : Robot
{
        void move( )
        {       super.move( );
                if (anyBeepersInBeeperBag())
                {       putBeeper( );
                }
        }
}

class Getter : Robot
{
```

```
        void move( )
        {       super.move( );
                while (nextToABeeper( ))
                {       pickBeeper( );
                }
        }
}
```

Ambas clases sustituyen el método move(). De está manera el robot Putter pone pitos en las esquinas en que se mueve y el robot Getter recoge por las que se mueve. Supóngase ahora que utiliza estás dos clases en la siguiente tarea:

```
task
{       Putter  Lisa = new Putter(1, 1, East, infinity);
        Getter  Tony = new Getter(2, 1, East, 0);

        loop (10)
        {       Lisa.move( );
        }
        loop (10)
        {       Tony.move( );
        }
}
```

Si el micromundo contiene un pito en cada una de las primeras diez esquinas a lo largo de la calle 1º y también cada una de las primeras diez esquinas de la 2º calle entonces cuando la tarea es realizada habrá 2 pitos en cada uno de los primeros 10 bloques a lo largo de la 1º calle, y ninguno de los correspondientes bloques de la calle 2º. No hay nada sorpresivo acerca de esto, pero note que tanto Lisa y Tony responden al mismo mensaje en idénticas (relativas) situaciones.

El significado de polimorfismo es siempre profundo para esto, pero sin embargo. En efecto, los nombres que se utilicen para referirse al robot no están "Lanzamiento anterior" para que los robots propiamente, pero sólo por comodidad del usuario. Un robot dado puede ser referido por diferentes nombres, llamados alias.

Primero, se declara el nombre que se usa como alias. No se hace referencia a un nuevo robot aún.

Robot Karel; // "Karel" Se refiere a algún robot.

En segundo lugar, se requiere "asignar" un valor al nombre karel. En otras palabras se necesita especificar que el robot de nombre "Karel" es al que nos referimos. Se hace esto con una instrucción de asignación. Esto asume que el nombre Tony ya se refiere a un robot como si ya fuera parte del citado bloque de tarea principal.

Karel = Tony;

Esto establece a karel como un nombre alternativo(alias) para el robot también conocido como Tony. Se puede exactamente acomodarse al nombre de "Karel" refiriéndose al robot Lisa. Es muy importante notar, sin embargo, que un alias no se refiere a cualquier robot hasta que asignemos un valor al nombre.

Entonces enviando un mensaje de turnLeft() usando el nombre de karel envía un mensaje al mismo robot que es nombrado como Tony aludiéndolo, aunque ellos son el mismo robot.

Karel.turnLeft( );

Asumamos ahora que se considera la tarea revisada lentamente a continuación: Se usa la misma estructura y que el micromundo es como antes. La única diferencia es que nos referimos al robot usando el nombre de Karel en cada caso. Note que mientras tenemos tres nombres aquí, sólo tenemos dos robots.

```
task
{       Robot  Karel;
        Putter Lisa = new Putter(1, 1, East, 100);
        Getter Tony = new Getter(2, 1, East, 0);
```

```
        Karel = Lisa;                   // "Karel" refiere a  Lisa;
        loop (10)
        {       Karel.move( );          // Lisa pone abajo 10 pitos.
        }
        Karel = Tony;                   // "Karel" refiere a  Tony
        loop (10)
        {       Karel.move( );          // Tony sweeps ten blocks
        }
}
```

Así que no sólo puede tener mensajes idénticos (move()) refiriéndose a diferentes acciones; aún si la instrucción del mensaje es idéntica Karel.move(). Se pueden tener diferentes acciones. Advierta sin embargo, que, aún se están enviando mensajes de los dos diferentes robots, y que estos robots son de diferentes clases. Es aún posible formar diferentes cosas que sucedan en diferentes ejecuciones de la misma instrucción.

Consideremos lo siguiente:

```
task
{       Robot  Karel;
        Putter Lisa(1, 1, East, 100);
        Getter Tony(2, 1, East, 0);

        loop (10)
        {       Karel = Lisa;                   // "Karel" refiere a Lisa;
                loop (2)
                {       Karel.move( );          // ??
                        Karel = Tony;           // "Karel" refiere a Tony

                }
        }
}
```

Note que el mensaje move() es enviado 20 veces, pero 10 veces es enviado a Lisa, y 10 veces a Tony, alternando nuevamente. La calle 1º alcanza pitos extras y la calle 2º los atrapa rápidamente.

3.8 CONCLUSIÓN

Finalmente, se espera hacer la siguiente pregunta: ¿Cuándo es apropiado diseñar una nueva clase y cuándo se modificará o adicionará una clase de robot existente?.

Una completa respuesta a la pregunta está más allá del alcance de este documento, porque hay muchas cosas que deben considerarse en la decisión, pero se pueden dar algunas directrices aquí si en su juicio una clase de robot contiene errores u omisiones, para que todos entiendan las modificaciones. Aquí las omisiones significan que hay algún método (acciones o predicados) que es necesario completar la funcionalidad básica de la clase o hacer que el robot lo haga en la clase o en la clase que fue diseñado.

De otra manera, si tenemos una clase útil, y necesitamos una funcionalidad adicional, especialmente más funcional que lo previsto por la clase donde ya se tiene construida una nueva clase como una subclase proporcionada apropiadamente. De tal forma, cuando se necesita un robot con capacidades originales puede usarse una clase original y cuando se necesita una nueva funcionalidad puede usarse una nueva. Algunas veces la escogencia es hecha porque encontramos que muchos de los métodos de algunas clases son la elección, es hecha porque encontramos que muchos de los métodos de alguna clase son exactamente las que esperamos, pero uno o dos métodos pueden ser más útiles en el nuevo problema si ellos son modificados o ampliados.

3.9 PROBLEMAS PROPUESTOS

Los siguientes problemas usan métodos de recursión, búsqueda y aritmética discutidos en el capítulo. Algunos de los problemas a continuación usan combinación de los métodos zigIzqUp() y zagDerDown(), o variantes sencillos de estos. Cada problema es difícil de solucionar pero una vez que el plan es descubierto (a través probablemente de un "aha de experiencia"), el programa

que implementa la solución no será asimismo difícil de escribir. Se puede asumir también que no hay sección de muros en el micromundo.

Finalmente podrá asumirse que el robot, karel comienza sin pitos en su bolsa a menos que se diga otra cosa. No hacer cualquier suposición acerca de la esquina de inicio de karel u orientación dada, a menos que se especifique en el problema.

1. Karel se ha graduado en alfombrar por etapas. Karel tiene que alfombrar un salón completamente encerrado sólo un pito puede ser localizado en cada esquina. El salón puede ser de cualquier tamaño y de cualquier forma. La figura muestra un posible piso. El área gris no debe ser alfombrada por karel. Karel puede comenzar en cualquier lugar del salón y estar en cualquier orientación.

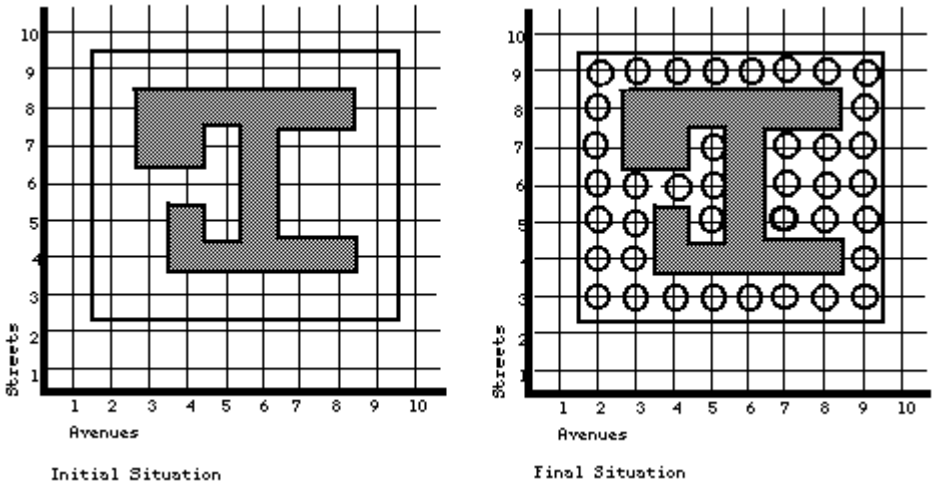


Figura 65 Estado inicial y final del problema “Alfombrar”

2. Reescriba tanto zigIzqUp() y zagDerDown() así que automáticamente satisfagan las precondiciones de orientación.

3. Asuma que hay un pito en la calle 1º y avenida Nº. Programe a karel para encontrarlo y llevarlo a la calle Nº y avenida 1º.

4. Asuma que hay un pito en la calle Sº y avenida Aº y que karel tiene dos pitos en la bolsa. Programe a karel para poner los pitos de la bolsa en la calle 1º y avenida Aº y calle Sº y avenida 1º.

El pito original tiene que quedarse en la esquina en que comenzó.

5. Si usted disfruta de la ciencia computacional y eventualmente ha tomado un curso en teoría de la computación, gratamente evoque los días que usted, gastó programando karel y trate de solucionar el siguiente problema, pruebe que karel con la ayuda de algunos pitos es equivalente a la máquina de Turing. Hint. Use la equivalencia entre la máquina de Turing y un autómata 2-contador.

LISTA DE FIGURAS

Pág.