Computación I

Ramón Reyes   Johan Van Horebeek
ramon@cimat.mx   horebeek@cimat.mx

Licenciatura en Matemáticas y Computación
CIMAT Guanajuato

agosto 1999

Parte 1: Karel el Robot

1 El robot y su mundo

1.1 Introducción

En esta primera parte pretendemos introducir varios conceptos básicos de programación. La meta no es aprender un lenguaje en particular (aunque en la práctica eso es lo que haremos), más bien, se trata de adquirir la habilidad de formular algoritmos corectos y aprender conceptos comunes a la mayoría de los lenguajes, tales como: procedimientos, condicionales, iteraciones, etc.

Con ese fin haremos uso de un robot imaginario al que le daremos las órdenes para verificar si hemos resuelto adecuadamente los problemas que se nos presentan.

1.2 El mundo de Karel

Karel[1] es un robot al cual podemos controlar por medio de un programa para que realice cierto trabajo.

El mundo de Karel consta de los siguientes elementos:

1. - *Calles* (horizontales) y *avenidas* (verticales) que se cruzan en *esquinas* .
   - *Paredes* inpenetrables colocadas entre dos esquinas.
   - *Bipers* removibles colocados en las esquinas que emiten un sonido (su grosor es irrelevante)

[1]El nombre Karel refiere a Karel Capek, dramaturgo checoslovaco que popularizó la palabra robot en unas de sus obras de teatro. La palabra robot viene de *robotica*, que significa "trabajo forzado" en checo.

- *Bolsa de bipers* que Karel lleva consigo.

2. Karel, el protagonista, siempre está en alguna esquina mirando al norte, este, sur u oeste. A través de tres cámaras puede ver si hay una pared entre el y las esquinas más cercanas. Su oido le permite detectar la presencia de bipers en la esquina donde se encuentra parado.

El manejo y limitaciones de cada uno de estos elementos es lo que pretendemos entender a lo largo de esta parte del curso.

La manera de comunicarse con Karel es por medio de un *programa*. El problema principal es que lo único que Karel puede hacer es seguir lo que le indiquemos "al pie de la letra". Karel no piensa y no puede darse cuenta de lo que queremos que haga si no sabemos como decírselo, para eso aprenderemos a comunicarnos adecuadamente con Karel usando su lenguaje, es decir recurrimos a un *lenguaje de programación*.

Lo que puede hacer Karel es, en primera instancia, muy limitado, de hecho comenzaremos con un conjunto muy reducido de instrucciones que podemos darle. Parte de la moraleja es ver cómo con un número reducido de instrucciones podemos llegar a hacer trabajos bastante sofisticados.

2 Primeras instrucciones y programas

Para describir en cada momento la situación en la que se encuentra Karel, necesitamos detallar todos los elementos que conforman su mundo, así, una tarea o trabajo específico de Karel consistirá en llevarlo de una situación *original* a una *final* através de la ejecución de *instrucciones*. Comenzaremos con las instrucciones básicas

- Cambio de posición:
  - `paso` hacer un paso en la dirección que está apuntando (puede causar error "apagón" si hay una pared en frente).
  - `vueltaIzquierda` girar a la izquierda (siempre se puede).

  Observa que no necesitamos `vueltaDerecha`, ¿Por qué?

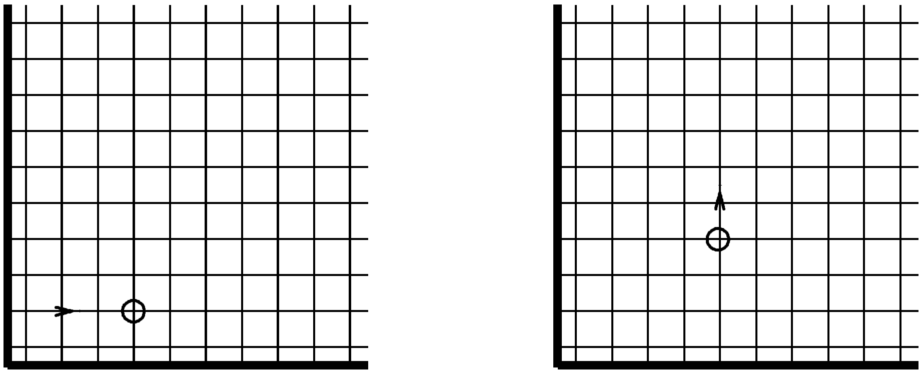
- Manipulando bipers
  - `recogeBiper` recoge un biper de la esquina dónde está parado.
  - `ponBiper` deposita un biper.

  Pueden ocasionar apagones.

- Terminado `termina` (debe ser siempre el último comando)

2.1 Primer programa completo

Queremos que Karel siendo que se encuentra apuntando al este en la calle 2 avenida 2, realice la tarea de llevar el biper que se encuentra en la calle 2 avenida 4 a la calle 4 avenida 5, y debe moverse una cuadra más al norte antes de apagarse.



```
beginProgram
  beginMain
    paso;
    paso;
    recogeBiper;
    paso;
    vueltaIzquierda;
    paso;
    paso;
    ponBiper;
    paso;
    termina;
  endMain
endProgram
```

Al ejecutar el programa, Karel revisa que no tenga errores y luego procede a llevar a cabo cada una de las instrucciones entre las *palabras reservadas*: `beginMain` y `endMain`, hasta que se produce un apagón o encuentra un `termina`.

No basta con escribir el programa, hay que simularlo, para verificar que realiza lo que queremos.

2.1.1 Gramática

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

- puntuación (solo el “punto y coma” después de cada comando)
- instrucciones

– `paso`, `vueltaIzquierda`, `recogeBiper`, `ponBiper` y `termina`

- palabras reservadas
  – `beginProgram`, `endProgram`
  – `beginMain`, `endMain`
  – `{` y `}`
  – `if`, `else`
  – `iterate`, `times`
  – `while`
  – `defineInstruction`
- pruebas
  – `Frentelibre`, `Frenteobstruido`, `Izquierdalibre`, `Izquierdaobstruida`, `Derechalibre`, `Derechaobstruida`
  – `sobreBiper`, `nosobreBiper`
  – `alNorte`, `noalNorte`, `alSur`, `noalSur`, `alOeste`, `alOeste`, `alEste`, `noalEste`
  – `Bipersdisponibles`, `noBipersdisponibles`

2.2 Errores

Hay varios tipos de dificultades con las que que podemos toparnos al escribir un programa. Una de las partes más complicadas de la programación es el detectar donde están las fallas de nuestro programa.

2.2.1 Apagón

Karel se ve impedido de realizar alguna de sus instrucciones básicas y se apaga por sí mismo. Esto es un error imputable al que escribió el programa y es mejor apagarse que comenzar a hacer más tonterías. Las instrucciones que pueden ocasionar un apagón son: `paso`, `recogeBiper` y `ponBiper`.

2.2.2 Programación

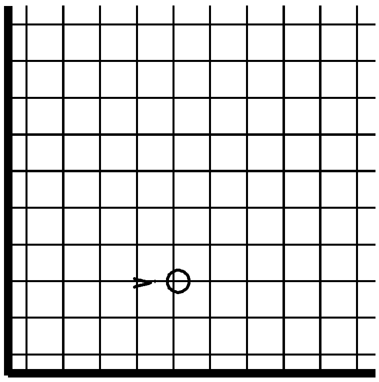
Hay varios tipos de errores de programación, estos los clasificaremos en:

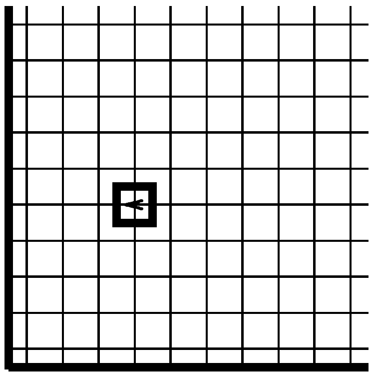
- léxicos (palabras que no están en el vocabulario)

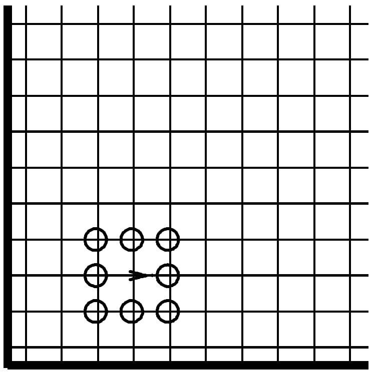
- sintácticos (se entiende, pero no está bien escrito)
- de ejecución (no puede hacer lo que dijimos, apagones)
- de intento (hace correctamente otra cosa)

2.3 Problemas

1. ¿En que direcciones puede Karel apuntar?
   - noreste
   - este
   - 164 grados
   - vertical
   - para abajo
2. ¿Qué objetos además de Karel forman parte de su mundo? ¿Cuáles de estos objetos puede Karel manipular o cambiar?
3. Da la posición exacta y relativa de Karel en los siguientes mundos:





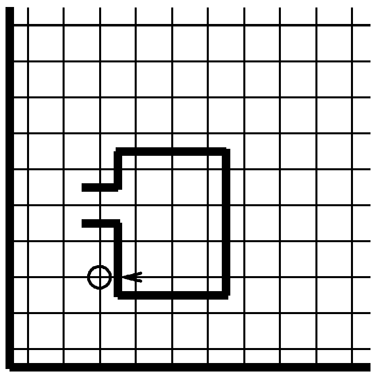


4. Encuentra y corrige los errores en el siguiente programa

```
beginProgram
beginmain
paso;          vueltaDerecha;
paso;
recogeBipers;
vueltaIzquierda ;
termina
endmain
endProgram
```

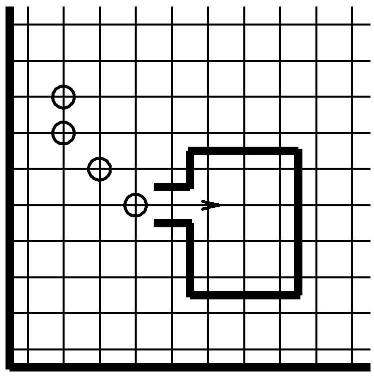
5. ¿Cuál es el programa más pequeño que es correcto en su sintaxis?

6. Cada mañana Karel tiene que salir a recoger el periodico y regresar a la cama a leerlo. Escribe el código correspondiente.



7. Karel tiene que escalar una montaña y plantar su bandera de alpinista. Escribe el código correspondiente.

8. A Karel se le caen algunos víveres cuando se rompe su bolsa del mandado al llegar a la casa y tiene que regresar a recogerlos. Escribe el código correspondiente.



3 Extendiendo el lenguaje de Karel

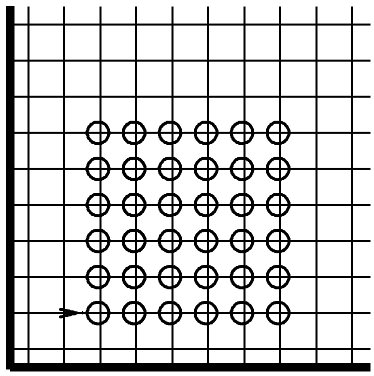
Tenemos la posiblidad de extender el limitado lenguaje de Karel definiendo instrucciones nuevas, la sintaxis es:

```
defineInstruction <nombre-nueva-instruccion>
<instruccion>
```

donde `<instruccion>` puede ser una sola instrucción o un bloque de instruciones creado con `{` y `}`. Esto crea lo que se conoce como una entrada del diccionario de Karel, que es donde está todo lo que Karel puede ejecutar.

Es bueno darles nombre a las instrucciones de acuerdo a la acción que realizan, para simplificarnos el poder entender los programas que escribimos. No se puede usar nombres de comandos básicos ni palabras reservadas. Karel hace distinción entre mayúsculas y minúsculas. No se puede usar carácteres especiales.

Por ejemplo, para cosechar las zanahorias que Karel plantó hace tiempo



usamos definiciones de nuevas instrucciones tales como:

```
defineInstruction cosecha2Filas
  {
    cosecha1Fila;
    siguienteFilaIzquierda;
    cosecha1Fila;
    siguienteFilaDerecha;
  }
```

sucesivamente definimos:

```
defineInstruction cosecha1Fila
  {
    recogeBiper;
    paso;
    recogeBiper;
    paso;
```

```
    recogeBiper;
    paso;
    recogeBiper;
    paso;
    recogeBiper;
    paso;
    recogeBiper;
  }
```

y

```
defineInstruction  siguienteFilaIzquierda
  {
   vueltaIzquierda;
   paso;
   vueltaIzquierda;
  }

defineInstruction  siguienteFilaDerecha
  {
   vueltaDerecha;
   paso;
   vueltaDerecha;
  }

defineInstruction  vueltaDerecha
  {
   vueltaIzquierda;
   vueltaIzquierda;
   vueltaIzquierda;
  }
```

el programa principal será:

```
beginMain
 paso;
 cosecha2Filas;
 cosecha2Filas;
 cosecha2Filas;
 termina;
endMain
```

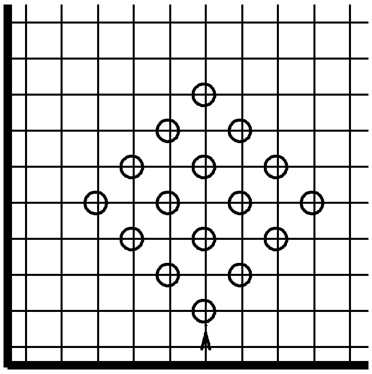
Siempre se puede escribir un programa sin instrucciones nuevas pero resulta una larga secuencia de comandos básicos de Karel, y es muy complicado de entender, corrijir, modificar, etc.

Si se llama una misma función en varias ocasiones se debe verificar bien que se empieza siempre desde situaciones similares (por ejemplo la segunda vez cuando se llama `cosecha2Filas`, Karel debe estar sobre el primer biper de la nueva fila como es el caso para la primera llamada).

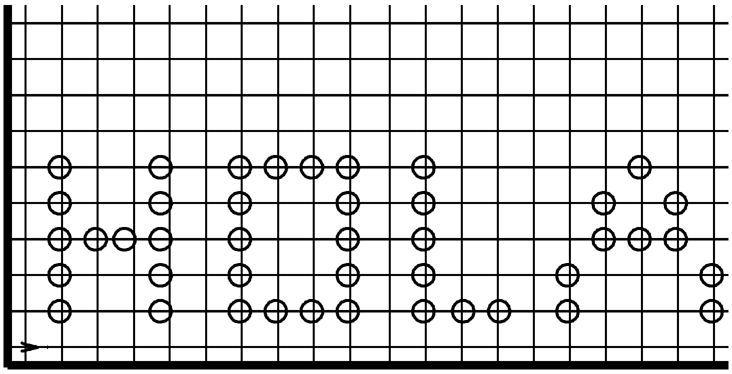
En la vida cotidiana usamos algo similar: para resolver un problema lo dividimos en problemas más chiquitos que a su vez vamos refinando hasta tareas simples. Se llama esta estrategia comunmente *top–down*.

3.1 Problemas

1. El problema de Karel es ahora cosechar el siguiente campo que plantó después de un partido de baseball.



2. Karel quiere mandar un mensaje a astrónomos de otros mundos para lo cual es necesario que arregle bipers de la siguiente manera:



4 Ejecutando instrucciones condicionalmente

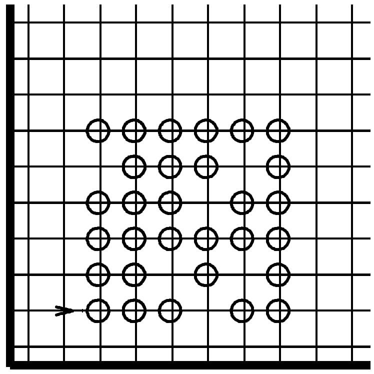
Karel necesita tener la posiblidad de decidir si va a realizar una instrucción dependiendo de una cierta condición. Para eso hay dos tipos de condicionales, `if` y `if/else`.

4.1 La instrucción `if`

Por ejemplo para evitar la muerte Karel, puede checar si cada vez que intenta recoger un biper, realmente hay uno, y así definir `recogesihay`. La sintaxis del `if` es:

```
if <prueba>
  <instruccion>
```

Si en el problema de la cosecha de Karel, resulta que algunas de las zanahorias que plantó, no se dieron



necesitamos modificar la definición de `cosecha1fila` sustituyendo `recogeBiper` por `recogesihay`:

```
defineInstruction recogesihay
{
 if sobreBiper
   recogeBiper;
}
```

A continuación tenemos dos ejemplos de como definir la instrucción `volteaalNorte`

```
defineInstruction volteaalNorte
  {
    if alEste
      vueltaIzquierda;
    if alSur
      {
        vueltaIzquierda;
        vueltaIzquierda;
      }
    if alOeste
      {
        vueltaIzquierda;
        vueltaIzquierda;
        vueltaIzquierda;
      }
  }
```

En este caso respondemos a la pregunta ¿qué necesita hacer Karel dependiendo de en qué dirección está apuntando? Sin embargo, si comenzamos planteando la situación de otro modo, respondiendo a la pregunta ¿Necesita Karel girar a la izquierda? obtenemos:

```
defineInstruction volteaalNorte
  {
    if noalNorte
      vueltaIzquierda;
    if noalNorte
      vueltaIzquierda;
    if  noalNorte
      vueltaIzquierda;
  }
```

Las dos son correctas.
Finalmente, observa la diferencia entre

```
if alOeste
vueltaIzquierda;
paso;
```

y

```
if alOeste
{
 vueltaIzquierda;
 paso;
 }
```

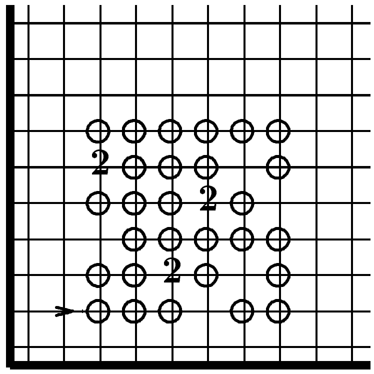
4.2 La instrucción `if/else`

Esta otra modalidad de condicional sirve para poder decidir entre dos posiblidades. La sintaxis es:

```
if <prueba>
  <instruccion1>
else
  <instruccion2>
```

4.3 Instrucciones `if` anidadas

Supongamos ahora que el campo a cosechar de Karel tiene el problema de que en algunas esquinas hay dos, en otras una y en otras ninguna zanahoria , y Karel quiere replantar el campo de tal manera que quede una sola zanahoria por esquina comenzando con una dotación suficiente de zanahorias (bipers).



La instrucción a ejecutar para la cosecha puede ser

```
defineInstruction replantaUno
  {
    if nosobreBiper
      ponBiper;
    else
      {
        recogeBiper;
        if  nosobreBiper
          ponBiper;
      }
  }
```

En un `if`, `if/else` podemos invertir el orden de las instrucciones siempre y cuando la prueba que hagamos sea la opuesta. Además, si al final de ambos bloques de instrucciones tenemos una(s) instruccione(s) común(es) podemos *factorizar por abajo* i.e. sacar estas instrucciones de las 2 partes y ponerlas después del `if/else`. Para factorizar por arriba debemos verificar que lo que factorizamos no afecte la prueba que estamos haciendo.

4.4 Problemas

1. Karel debe apuntar al norte si está en una esquina con un biper y apuntar al sur si en la esquina hay dos bipers (y después dejar los bipers tal y como estaban).

2. Karel debe poder decidir si está completamente rodeado de paredes y si así es, debe apagarse, en caso contrario, debe quedar funcionando y apuntando en la misma dirección que lo hacía originalmente.

3. ¿Qué es lo que hace la siguiente instrucción `misteriosa`? Simplifícala.

```
defineInstruction misteriosa
{
```

```
        if alOeste
          {
            paso;
            vueltaDerecha;
            if alNorte
              paso;
            vueltaIzquierda;
            vueltaIzquierda;
          }
        else
          {
            paso;
            vueltaIzquierda;
            paso;
            vueltaIzquierda;
            vueltaIzquierda;
          }
      }
```

5 Instrucciones que se repiten

Con las instrucciones que producen ciclos o "loops" completamos todo el lenguaje de Karel, éstas permiten que Karel realice un cierto número de veces determinada instrucción (o bloque de instrucciones).

5.1 Las instrucciones `iterate` y `while`

En lugar de escribir una instrucción repetidas veces podemos decirle a Karel exactamente el número de veces que queremos que repita la instrucción con la siguiente sintaxis:

```
  iterate <numero> times
    <instruccion>
```

Por ejemplo:

```
defineInstruction vueltaDerecha
  {
    iterate 3 times
      vueltaIzquierda;
  }
```

Sin embargo, hay muchas situaciones en las que no sabemos el número de veces que necesitaremos ejecutar una instrucción, pero sabemos que debemos ejecutarla en tanto se cumple cierta *prueba*. Para esto tenemos:

```
while <prueba>
  <instruccion>
```

Por ejemplo para recoger todos los bipers que se encuentran entre la posición de Karel y una pared que está más adelante en frente de el:

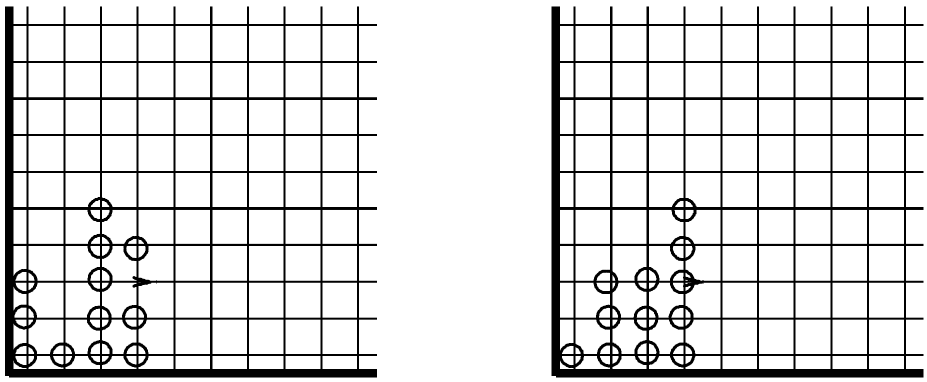
```
while  Frentelibre
  {
  if sobreBiper
        recogeBiper;
  paso;
  }
```

5.2 Problemas

1. Karel debe ordenar un conjunto de pilas de bipers. Un cierto número de avenidas consecutivas, empezando con el origen, contienen una pila vertical de bipers (empezando en el primer calle). Una avendina con ningún biper indica que ya no hay más.

   Karel debe reordenar las pilas según el número de bipers que contienen (de menor a mayor).

   Karel no sabe donde se encuentra al inicio. No importa donde el se encuentra al final.

   Ejemplo de una situación inicial y final.

2. Supongamos que Karel está en la calle 1, avenida 1; se sabe que en algún lugar hay un biper. '? Cómo encontrarlo?

6 Recursión

Tomamos la siguiente instrucción:

```
defineInstruction volteaalNorte
  {
    while noalNorte
      vueltaIzquierda;
  }
```

otra posiblidad será

```
defineInstruction  volteaalNorte
  {
    if  noalNorte
      {
        vueltaIzquierda;
        volteaalNorte;
      }
  }
```

Este es un ejemplo de recursión: usar en la definición de un instrucción lo que estás definiendo (pero en un contexto diferente para evitar un círculo visioso).

Consideramos el siguiente problema: Karel sabe que en la dirección donde está mirando, hay una mina. Lo que debe hacer es recogerla y luego regresar a su lugar. Se puede suponer que no hay paredes en el camino.

```
defineInstruction buscaMina
  {
    if sobreBiper
     {
      recogeBiper;
      vueltaIzquierda;
      vueltaIzquierda;
      }
    else
      {
        paso;
        buscaMina;
        paso;
      }
  }
```

6.1 Problemas

1. Karel está en frente de una pila de bipers. Si la cantidad es igual a la que el lleva consigo debe terminar mirando al norte, en el caso contrario al sur (puedes suponer que no hay cerca paredes ni otros bipers).

2. Supongamos que Karel se encuentra en calle n. Escribe un programa que lo mueva a calle $2n$.

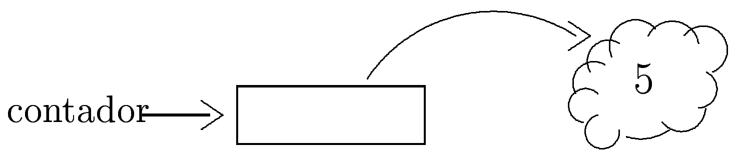
7 Extensiones de Karel

Antes de poder usar un lenguaje profesional de programación, necesitamos introducir dos conceptos más : una *variable* y un *objeto*.

7.1 Variables

Tomamos como punto de partida el biperbag de Karel. En cualquier momento se encuentra ahí un número determinado de bipers. Será eficiente que Karel lleve un *registro* de cuantos hay. Con ese fin va a *apuntar* en algun lugar un número entero (positivo).

Podemos llamar este registro la *variable* contador cuyo valor es un número entero. Lo podríamos representar de la siguiente manera:



La caja indica el lugar físico donde Karel va a apuntar el número (puede ser un papel, algunos bits en una memoria electrónica, etc. para nosotros no importa la implementación específica). Llamamos la nube el *valor* de la variable.

En general restringimos los posibles valores que una variable puede tomar. En el ejemplo del biperbag, es evidente que el valor nunca puede ser 3.24, $\sqrt{23}$, "AE", etc. Llamamos *integer* la familia de todos los numeros naturales. Indicar a que familia partenece una variable (por ejemplo contador), lo hacemos indicando el *tipo* de la variable de la siguiente manera:

```
int contador;
```

El uso de *tipos* permite saber con anticipación el espacio necesario para guardar los valores y al mismo tiempo facilita la detectión de posibles errores.

Otras familias son `boolean` (dos valores: `false` (falso) y `true` (verdadero)), `real` (el valor es un número real), `char` (carácteres, por ejemplo 'a', '2', etc.) y `string` (el valor es una cadena de carácteres, por ejemplo "tgfgwefw", donde usamos " " para indicar que no nos referimos a la variable tgfgwefw). El siguiente ejemplo muestra el uso:

```
int contador; int doble;
contador=7;
doble=contador*2;
```

Observa que '=' significa asignar el *valor* del lado derecho a la*variable* de lado izquierdo.

No olvides asignar un valor a una variable antes de usarla! No olvides declarar una variable antes de usarla!

El concepto *variable* permite definir operaciones que dependen de un parámetro. Por ejemplo, para ponBiper, podemos añadir una variable que indica cuantos bipers poner.

```
int cantidad;
cantidad=7;
ponBiper(cantidad);
ponBiper(cantidad+3);
```

En este contexto observa que pruebas como `sobreBiper`, `alNorte` son en realidad instruciones que regresan un valor tipo `boolean`.

7.2 Objetos

Hasta ahorita la manera de construir un programa fue siempre a través de la especificación de todas las instruciones que Karel va a tener que ejecutar para resolver una cierta tarea (eventualente agrupándolas en funciones para tener más claridad).

Existe otro enfoque. Con ese fin considera que el mundo de Karel consiste de *objetos* y cada uno partenece a una cierta familia (*class*): las paredes verticales y horizontales son miembros de la familia `Pared`, los bipers de la familia `Beeper`, karel es miembro de la familia `Robot`, etc. Para crear un *miembro* de una familia de objetos particular, usamos la siguiente estructura:

```
Karel = new Robot();
Clodomiro = new Robot();
pared1= new Pared();
```

A cada familia de objetos podemos atribuir ciertas características y operaciones (métodos). Por ejemplo, todos los miembros de la familia `Robot` pueden dar un paso adelante a través del comando `paso` (igualmente una `vueltaIzquierda`, etc.). Lo denotamos así:

```
Karel = new Robot();
Clodomiro = new Robot();
Karel.paso();
Karel.vueltaIzquierda();
Clodomiro.paso();
```

Como se puede ver, contrario a variables, en la notación de funciones se usa siempre '()' al final.

Es muy frecuente que ordenemos las familias según los métodos que tienen en común. `Pared` y `Beeper` (junto con `Fuente`, `Foco`, etc.) son todos `Articulo`. Tienen en común un método para colocarlos en un lugar específico. Por otro lado, la clase `Beeper` tendrá un método para indicar el volumen del tono, `Pared` algo para cambiar su grosor u orientación, etc. El siguiente ejemplo muestra un caso hipotético:

```
class Robot extends Maquinas  {
    void paso() {
      /* aqui el codigo para establecer un paso adelante */
   };
    void ponBipers(int x) {
      /* aqui el codigo para que ponga n bipers */
   };
    boolean sobreBiper()
    {/* aqui el codigo para que verificar si esta sobre un biper */
    }

   /* etc. etc. */
}
```

donde `void` indica que la instrución no regresa ningún resultado y `/* */` sirve para incluir comentarios que Karel no debe considerar.

8 Bibliografía

1. Richard Pattis, *Karel the Robot, a gentle introduction to the art of programing*, John Wiley, 1990

2. Rafel Gutierrez, *JavaKarel*, Tesis de licenciatura, CIMAT, 1999.