Estructuras de Datos y Algoritmos

Año 2022

Descripción Informática de Conjuntos

1. Introducción

Repasemos algunas nociones matemáticas:

Usualmente se dice que los conjuntos se pueden definir por comprensión o por extensión. Si hablamos de "el conjunto de los números pares entre 7 y 2.000 " tenemos un conjunto definido por comprensión. El largo de la definición es pequeño frente a la cantidad de números definidos. Si hablamos de "el conjunto de los números 137, 426, 7.208, 8.34" tenemos una definición por extensión. Si queremos saber si 1.000 está en el primer conjunto verificamos:

si es par y
si está comprendido entre 7 y 2.000

y ambas pruebas son satisfactorias. Números como 1.001, 3.000, 5.555 fracasan en una, otra o ambas pruebas.

En cambio si queremos saber si 7.208 está en el segundo conjunto debemos recorrer la lista que lo define hasta encontrarlo, en este caso en el tercer lugar. Para saber que 5.000 no está debemos recorrer la totalidad de la lista, que en este ejemplo es pequeña, pero podría haber sido mucho más larga.

Si lo quisiéramos hacer en una computadora nos ayudaríamos con una estructura de datos, pero esto no es por ahora nuestra principal preocupación.

Los matemáticos prefieren las definiciones por comprensión. Si necesitan afirmar una propiedad de los elementos del conjunto la demuestran con un teorema. Si la definición es por extensión la deben comprobar en cada uno de sus elementos.

Cuando se debe computar algo con los elementos del conjunto, nuevamente es superior la definición por comprensión.

Por ejemplo, hallar el conjunto de los duplos, en el primer caso se expresa como "el conjunto de los múltiplos de cuatro entre 14 y 4.000" para lo cual hice tres productos. En el segundo caso debo hacer tantos productos como elementos haya, en el ejemplo son 4 para obtener "el conjunto de los números 274, 852, 14.416 y 16.686" Otro ejemplo sería sumar entre sí los elementos del conjunto. En el primer caso reconozco que tengo una progresión aritmética y haría (primero+último) $\times$ cantidad/2 = (8+2.000) $\times$ 997/2 = 1.000.988 y habré sumado 997 números. En el segundo caso tendré que hacer suma tras suma, 137 + 426 + 7.208 + 8.343 = 16.114.

Para que pueda haber una definición por comprensión debe haber una cierta regularidad. Una definición por comprensión de un conjunto finito se puede transformar, a veces laboriosamente, en una definición por extensión. En cambio una definición por extensión no se puede transformar en una definición por comprensión, salvo casos aislados.

Otra posibilidad de definir conjuntos es el uso de una recurrencia. Se da un primer elemento y una regla que permite construir a partir de los elementos ya conocidos el siguiente. Usualmente esta

regla sólo cita al anterior. Si el conjunto es finito debe estar acompañado de algún criterio que permita reconocer que se ha llegado al último elemento.

Esta descripción de conjuntos comparte características con cada una de las formas vistas.

Para consultar pertenencia, en similitud con la definición por extensión, debe generar elemento tras elemento y el tiempo crece con la cantidad de elementos.

Para describir el conjunto, en similitud con la definición por comprensión, el espacio necesario depende muy poco de la cantidad de elementos.

Las definiciones por recurrencia son las que mejor se adaptan al trabajo computacional.

Como los conjuntos computacionales deben ser finitos es obligatorio disponer de un criterio −o sea una función booleana − que nos indique si hemos llegado al último elemento.

Hablar de detectar el último elemento implica que no aceptamos conjuntos vacíos, pues en ellos no hay último elemento. Si queremos cubrir este caso debiéramos tener una función que nos permita detectar el fin de la generación por la aparición de un elemento ajeno al conjunto.

Contaremos entonces con los siguientes elementos:

- El valor explícito del eventual primer elemento, o de varios iniciales si la recurrencia fuera más compleja.
- Una función que permite a partir del elemento corriente construir el candidato a ser el siguiente.
- Una función booleana que permita juzgar si el valor a usar pertenece o no al conjunto.

Con ellos se tiene la siguiente estructura básica para un programa que quiere procesar todos los elementos del conjunto:

```
x := Primer-Elemento;
while válido(x)
do begin Procesar(x);
   x := sig(x);
end ;
```

o en C o Java aprovechando la capacidad expresiva de su **for**:

```
for( x = Primer-Elemento; válido(x); x = sig(x))
  Procesar(x);
```

Si el conjunto fuera vacío todos estos trozos no hacen ninguna acción útil. Alguien podrá objetar que los cambios hechos para incluir conjuntos vacíos eran innecesarios. Pues con no escribir nada los hubiéramos procesado. Hay que tener presente que para una mentalidad imperativa de programación los nombres designan entidades dinámicas, que cambian sus valores a lo largo del cómputo. Incluir el conjunto vacío no es para procesar vacíos permanentes sino vacíos circunstanciales. La variación del conjunto puede surgir de un cambio de su valor inicial o de un cambio de las funciones de válido y

de siguiente. Si el lenguaje de programación no aceptara que éstos se cambien se puede introducir la necesaria flexibilidad usando funciones parametrizadas.

El matemático que se acerca a la programación imperativa choca con el hecho de que en su tema las letras representan valores arbitrarios pero fijos y aquí valores concretos y móviles.

Las otras definiciones de conjunto no engendran directamente algoritmos que los recorran.

Una definición por extensión como la que pusimos al comienzo si podría engendrar el siguiente programa:

```
Procesar(137);
Procesar(426);
Procesar(7.208);
Procesar(8.343);
```

Pagando el precio de hacer el conjunto estático. Una solución flexible obligaría a almacenar estos valores a fin de recorrerlos cuando se necesite el proceso de los elementos. Ese almacenamiento podría registrar los cambios que sufra el conjunto.

Pero almacenarlos obliga a asignarles un conjunto de celdas, pero a este conjunto hay que definirlo, con lo cual la solución se pospone hasta haber resuelto este nuevo problema de definición.

Las definiciones por comprensión bien estructuradas constan de dos partes: género próximo y diferencia específica.

Así nuestra primera definición “el conjunto de los números pares entre 7 y 2.000” tiene el género próximo “números pares” y la diferencia específica “que estén entre 7 y 2.000” . El género próximo de un conjunto es también un conjunto.

Por ejemplo:
Si llamamos X al conjunto que estamos definiendo y P al conjunto de los números pares, en símbolos sería:

$$X = \{x | x \in \underbrace{P}_{\text{género próximo}} \wedge \underbrace{7 \leq x \leq 2000}_{\text{diferencia específica}}$$

Si el conjunto es finito y sabemos recorrer el género próximo podríamos recorrer la estructura con el siguiente esquema

Una estructura de control (while o for) que recorra el género próximo con la variable x:

```
if Satisface−la−diferencia−específica(x)
  then Proceso(x)
fi
```

Pero esto tiene inconvenientes, el género próximo podría ser mucho mayor que el conjunto de interés y esto alargaría el proceso. Si el género próximo fuera infinito podríamos no poder reconocer que ya hemos obtenido la totalidad de los elementos que satisfacen la diferencia específica y continuar indefinidamente en el ciclo. Justamente buena parte del esfuerzo de programación consiste en transformar definiciones por comprensión en algoritmos eficientes.

2. Información

Información encierra siempre una idea de irregularidad, imprevisibilidad, o sea ausencia de ley que lo genere. Por lo tanto conjuntos que contienen información necesitan una descripción por extensión. Para poder almacenar un conjunto en una computadora para procesarlo es necesario disponer de un conjunto de celdas y este conjunto sí se puede describir con alguna de las técnicas enunciadas.

3. Convenciones gráficas para describir estados de memoria

Frecuentemente un dato ocupa más de un lugar de memoria. Tanto los intérpretes de los lenguajes de programación como la misma arquitectura de la máquina invaden sucesivas celdas de memoria para almacenarlos. Al conjunto le asignan una única dirección que suele ser la menor de las direcciones de las celdas involucradas. De este modo podemos decir que hay *celdas lógicas* que agrupan a varias *celdas físicas*.

Una *celda lógica* se grafica con un rectángulo. Su dirección, si se la conoce, ya sea en su real valor, ya sea con un nombre que un intérprete transformará en un valor, se escribe yuxtapuesto al rectángulo.

Si se conoce el contenido se lo escribe adentro. Si es un número, carácter o texto se recurre a un sistema de representación útil para su rápida interpretación.

Si el contenido es la dirección de otra celda, es de poca utilidad conocer su valor ya que obliga a localizar que otro rectángulo del dibujo tiene ese valor como rótulo. Hay un acuerdo unánime de representar estos contenidos con flechas cuyo inicio sea un redondel lleno (Figura 1(a)). Este redondel se ubica en el rectángulo que describe a la celda poseedora de la dirección y la flecha se termina en la cercanía del rectángulo poseedor de esa dirección. Lo habitual es que el dibujo no aclare en ningún lugar el valor de esa dirección.

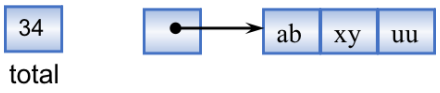
Direcciones, al igual que caracteres, son elementos que básicamente sólo se someten a operaciones de copia y comparación.

(a) Dirección (b) Null

Figura 1: Representación de direcciones.

Tratándose de direcciones es de utilidad tener una, adoptada convencionalmente, reservada para indicar la ausencia de un valor. En los lenguajes de programación se lo suele indicar con **NULL**, **nil**, **nihil**, ocultándose al programador cual es el valor utilizado para este fin. Se lo grafica con el símbolo eléctrico de una conexión a tierra(Figura 1(b)) . Finalmente si varias celdas constituyen una estructura sus rectángulos se dibujan con un lado contiguo. Si la estructura es un arreglo los rectángulos constituyentes deben ser del mismo tamaño.

Ejemplos:

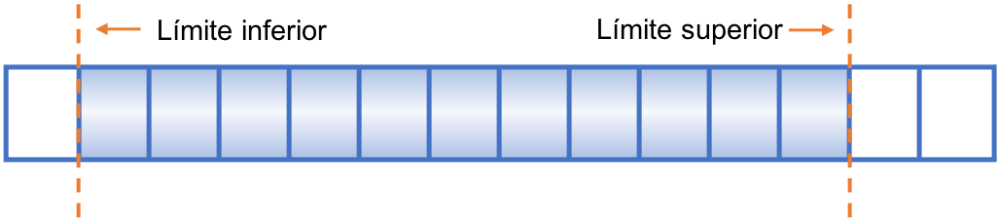


4. Descripción de conjuntos de celdas

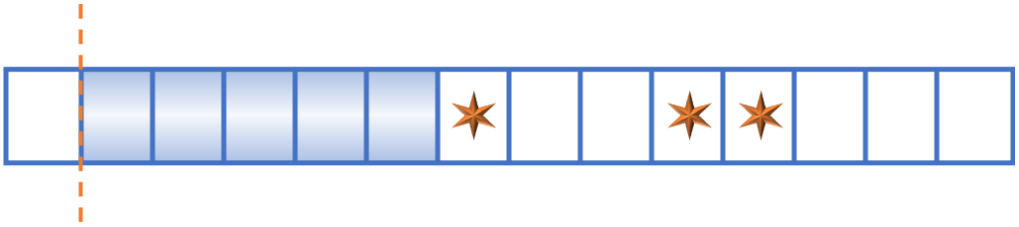
La dirección de una celda de memoria la describe unívocamente. Los circuitos de la máquina están orientados además a que su dirección conmuta compuertas y dé acceso a la misma. Describir un conjunto de celdas se puede reducir a describir un conjunto de direcciones.

El uso de lenguajes de programación permite utilizar nombres técnicamente identificadores en la descripción de las direcciones. Los compiladores y técnicas de direccionamiento de la máquina harán parte del trabajo de construir la dirección de una celda.

Podemos considerar que un trozo de un vector descripto por sus límites es un ejemplo de definición por comprensión. Para precisarla sólo necesitamos aportar el nombre del vector, suponiendo que usamos un lenguaje de programación, y los límites inferior y superior en el mismo. La sencillez de la definición permite transformarla inmediatamente en una definición por recurrencia apta para generar las direcciones de las celdas por medio de un programa.



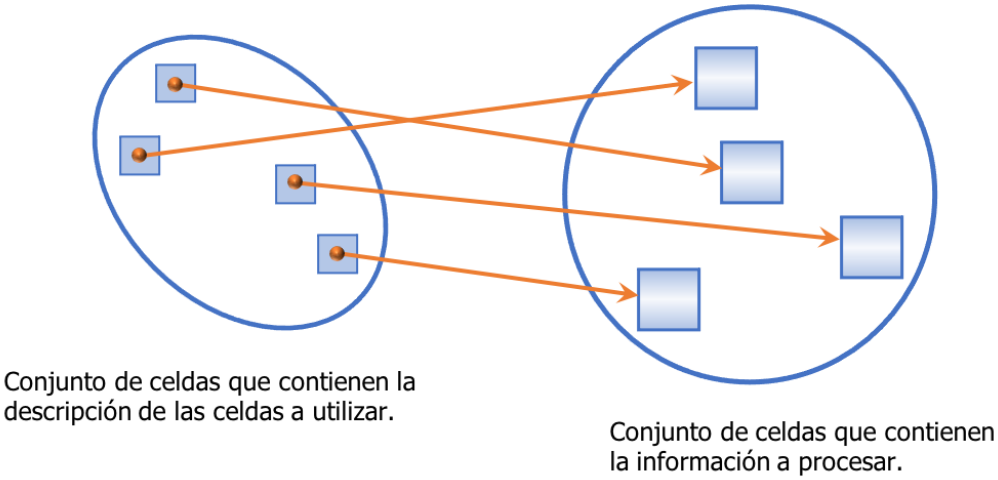
Un trozo de vector cuyo fin depende de un contenido particular tiene una definición por comprensión compleja pero admite una definición por recurrencia no significativamente más compleja que en el caso anterior.



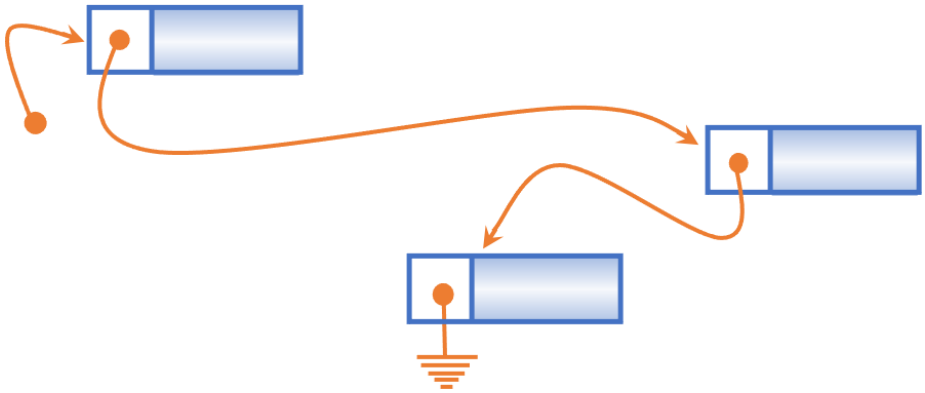
Una lista invertida es un ejemplo de definición por extensión

Evidentemente esta técnica sola no resuelve el problema sino que lo pospone. La pregunta será dónde almacenar el conjunto de direcciones.

Como toda definición por extensión es muy flexible para cambiar los elementos en uso. Finalmente tenemos una definición que comparte características de la definición por recurrencia y de la definición por extensión: la lista vinculada. Sólo después de determinar la dirección de un elemento estamos en



condiciones de determinar la ubicación del siguiente y en esto se parece a una recurrencia. Pero cada paso de avance está parametrizado por una nueva dirección y en esto se parece a una definición por extensión.



Localización de elementos

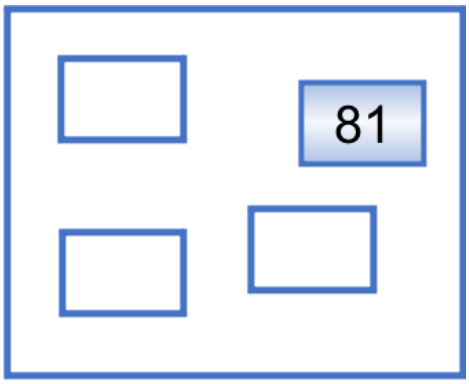
La operación más sencilla que podemos imaginar sobre un conjunto es la pregunta por pertenencia. La respuesta a esta pregunta es de naturaleza booleana, un valor de verdad, un sí o un no. Es por lo tanto un problema de decisión.

Ya aclaramos que si procesamos información ésta estará representada por un conjunto definido por extensión y sólo podremos contestar que está si lo hemos localizado. La localización aparece aquí como un subproducto no buscado pero veremos después que ésta será tan importante como saber la pertenencia.

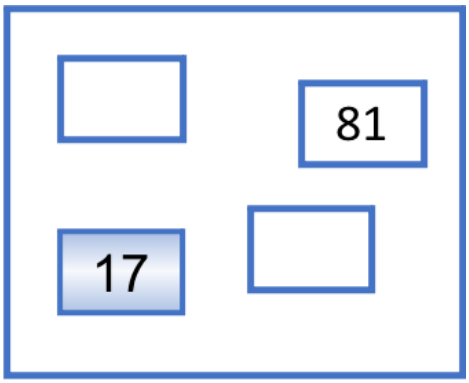
Si no poseemos ninguna información adicional fuera de la descripción de las celdas utilizadas para guardar el conjunto la única acción posible será examinar celda tras celda.

Veamos el siguiente ejemplo:

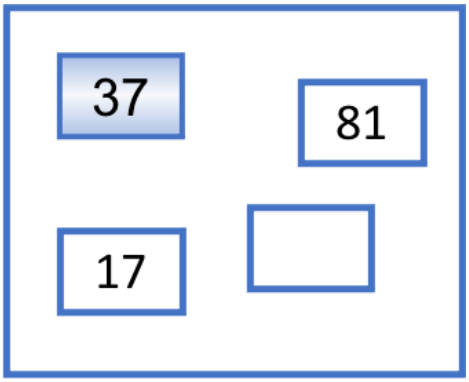
Disponemos de 4 celdas y buscamos el valor 37 (1)



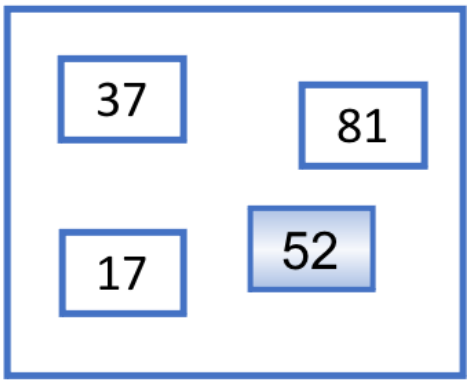
(a) **Escena 1**: miramos una celda y econtramos un 81, como no es lo buscado, seguimos.



(b) **Escena 2**: miramos otra celda y encontramos un 17, como no es lo buscado, seguimos.



(c) **Escena 3**: miramos otra celda y encontramos el 37, como es lo buscado estamos en condiciones de contestar que 37 pertenece al conjunto.



(d) Si por el contrario hubiéramos buscado el valor 50 hubiéramos continuado hasta el final para poder contestar que no pertenece al conjunto.

Cambiar el orden de búsqueda no introduce ninguna ventaja, tanto podemos encontrar antes lo buscado como postergar su encuentro. Por ello lo más razonable es recorrer el conjunto de celdas en un orden sencillo de programar.

4.1. Lista de alojamiento secuencial de largo conocido (conjunto guardado en un trozo de vector)

La primera línea del cuadro siguiente ilustra posibles contenidos y la segunda los subíndices que permiten direccionar las celdas en el vector V.

V :	184	238	107	193	74	523	234	171	297	84
	51	52	53	54	55	56	57	58	59	60

La siguiente función Pascal resolvería la pregunta de pertenencia de un valor x a este conjunto:

```
function pertenece (x : integer): boolean;
  var i: byte;
  begin
    i := 51;
    while (i < 61) and (v[i] <> x) do Inc(i);
    pertenece := i < 61;
  end;
```

o en C:

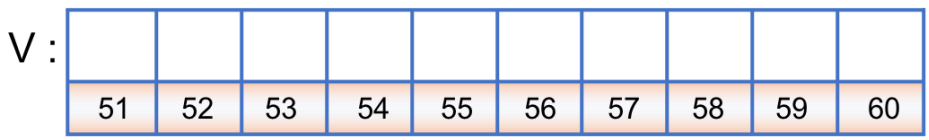
```
Bool pertenece(int x);
short i;
{
   for (i = 51; i< 61 && v[i]!= x ; i++);
   return i < 61;
};
```

o en Java:

```
Boolean pertenece(int x);
short i;
{
   for (i =51; i< 61 && v[i]!= x ; i++);
   return i < 61;
};
```

Comentarios:
La escritura C y Java que han utilizado && dejan en claro algo que en muchos compiladores Pascal es una opción: el modo de evaluar el and. Debe fijarse en modalidad de evaluación condicional del segundo operando. Si así no fuera se podría incursionar en una celda que no pertenece al trozo de interés del vector, es más podría no haber elementos adicionales en el vector y en algunas arquitecturas de máquina invadir otra partición o un lugar de memoria inexistente con el consiguiente error de ejecución.

Observe que la situación atada a un mecanismo concreto de descripción de celdas refleja lo que habíamos ilustrado más arriba en forma más abstracta. Ésta es la situación inicial. Todos los números sombreados describen celdas de cuyo contenido no conocemos.



Al recorrer el control del ciclo por primera vez hacemos este análisis: Si **184** fuera el valor buscado podríamos concluir que podemos contestar positivamente la pertenencia y abandonaríamos el ciclo con i = 51. Si 184 no fuera el valor buscado avanzaríamos i lo que equivale gráficamente a la siguiente situación:

El casillero de subíndice 51 no describe más una celda a examinar. De ahora en más la pertenencia

V :	184									
	~~51~~	52	53	54	55	56	57	58	59	60

o no al conjunto depende de que lo encontremos en las celdas de subíndice sombreado. Esto repite la situación inicial, lo que justifica el carácter iterativo del método. Después de algunas iteraciones suponiendo que x valga **200** tendríamos:

V :	184									
	~~51~~	52	53	54	55	56	57	58	59	60

La terminación queda asegurada por cuanto el conjunto a examinar tiene menos elementos y en un número finito de pasos llegaremos a un conjunto vacío. En un conjunto vacío podemos asegurar la no pertenencia. Esto queda ilustrado así:

V :	184	238	107	193	74					
	~~51~~	~~52~~	~~53~~	~~54~~	~~55~~	56	57	58	59	60

Por eso la decisión de pertenencia se resuelve preguntando si el ciclo se cortó por conjunto a examinar vacío, o sea i $\geqslant$ 61 (el único valor posible es i = 61).

Como esta estructura de programa es fundamental para la materia veremos un par de formas más para razonar el mismo programa.

4.2. A partir de una definición recursiva de pertenencia

Se puede razonar el programa a partir de la siguiente afirmación

$$x \in A \Leftrightarrow \begin{cases} \text{Si } A \neq \emptyset & \text{entonces } x = z \ \vee \ x \in A - \{z\} \\ \text{Si } A = \emptyset & \text{entonces } \bot \end{cases}$$

z es un elemento cualquiera; eligiéndolo convenientemente como el primero de la lista da lugar al siguiente programa recursivo:

```
pertenece(x, v[i..60]) =
   if  if i = 61 then ⊥
   else x = v[i] ∨ pertenece(x, v[i+1..60])
```

La llamada del programa principal será:

. . . pertenece(x, v[51..60]) . . .

Si la disyunción es de evaluación condicional el programa es más eficiente. Finalmente habría que usar alguna técnica de transformación de recursivo en iterativo por cuanto se trata de un programa ”tail-recursive”.

Si la disyunción no es de evaluación condicional nos da el siguiente programa:

```
Resultado := ⊥;
   for i variando de 51 a 60
    Resultado := (Resultado ⩒ x = v[i])
```

Si la disyunción es de evaluación condicional equivale a un **if**.

```
pertenece(x, v[i..60]) =
   if  i = 61 then falso
   else if  x = v[i] then verdadero
     else pertenece(x, v[i+1..60])
```

Quitando la recursión nos da el siguiente programa iterativo

```
i = 51
while ( i ≠ 61 and x ≠ v[i]) i := i + 1 ;
Resultado := i ≠ 61
```

4.3. Viéndolo como un problema de decisión con cuantificadores.

Hay una estructura genérica para problemas de decisión que involucran cuantificadores sobre un conjunto. La estructura general es: avanzo por el conjunto hasta poder tomar una decisión.

```
x := X-INICIAL
while no puedo decidir por verdadero and no puedo decidir por falso
  do
     x := sig(x)
  od
```

Decidir sobre la verdad de $(\forall x \in G)\ \varphi(x)$ se rige por las siguientes pautas:
Si los miré a todos y no hubo excepciones darlo por verdadero
En cuanto encuentro una excepción puedo darlo por falso

La generación de los $x \in G$ tiene una pregunta de control para saber si he llegado al fin y por lo tanto si los he mirado a todos. Puedo integrarlo con la pregunta por $\varphi(x)$.

Decidir por falso es encontrar un $\neg\varphi(x)$.

Decidir por verdadero es haber llegado hasta el fin sin que se haya cortado antes por un $\neg\varphi(x)$.

Si `fin(x)` es una función que devuelve el valor verdadero si alcanzó el final del conjunto y falso en otro caso, podríamos tener el siguiente código para decidir sobre la verdad de $(\forall x \in G)\ \varphi(x)$:

```
x := X-INICIAL
while ¬fin(x) and φ(x)
  do
     x := sig(x)
  od
Verdad := fin(x)
```

Si se tratara de un existencial $(\exists x \in G)\ \varphi(x)$, puedo decidir por verdadero en cuanto encuentro un x que satisface $\varphi(x)$. Sólo puedo decidir por falso si los he mirado a todos y nunca se dio $\varphi(x)$.

```
x := X-INICIAL
while ¬fin(x) and ¬φ(x)
  do
     x := sig(x)
  od
Verdad := ¬fin(x)
```

La pregunta por pertenencia de un elemento x' puede escribirse como el problema de decisión: $(\exists x \in G)\ (x = x')$ y el programa que lo resuelve es:

```
x := X-INICIAL
while ¬fin(x) and x ≠ x'
  do
     x := sig(x)
  od
Verdad := ¬fin(x)
```

Con este mismo enfoque pueden encararse programas que comparan dos hileras, que deciden si una lista está ordenada, si un número es primo, etc. Primero hay que expresarlo en términos lógicos como un problema de decisión. Los programas tendrán algunas de las estructuras indicadas.

4.4. Lista vinculada

Cambiar la forma de definir el conjunto no cambia la estructura básica del programa; sólo cambiarán:

- la forma de preguntar por conjunto remanente vacío,
- la naturaleza (tipo) de los elementos que describen la celda a examinar,
- el modo de darle el primer valor,
- la forma de referirse sintácticamente a los contenidos de las celdas y
- la forma de pasar de una celda a otra.

La siguiente función Pascal resolvería la pregunta de pertenencia de un valor x a este conjunto:

```
function pertenece(x : integer): boolean;
var p : Pnodo;
begin
   p := inicio;
   while (p <> nil) and (p^.valor <> x)
    do p = p^.sig;
   pertenece := p <> nil;
end;
```

o en C:

```
Bool pertenece(int x)
Nodo * p ;
{
  for (p = inicio ; p != NULL && p->valor != x ; p = p->sig)
  return p != NULL;
} ;
```

o en Java:

```
Boolean pertenece(int x)
Nodo p ;
{
  for (p = inicio ; p != NULL && p.valor != x ; p = p.sig)
  return p != NULL;
} ;
```

4.5. Listas ordenadas

Conservando constante la forma de pasar de una celda a otra, o sea la definición recursiva del conjunto de celdas, se puede decir que las celdas tienen un orden. Si los contenidos también admiten un orden, depositándolos convenientemente en las celdas se puede conseguir que al recorrer las celdas los contenidos sean monótonos crecientes (o decrecientes). En tales circunstancias se dice que la lista está ordenada.

El orden ayuda a una localización más eficiente.

Cuando se hace una exploración secuencial como las programadas hasta ahora, se puede decidir que un elemento no está cuando se ve uno mayor que el buscado. Un sencillo razonamiento por propiedad transitiva de la desigualdad permite razonar que todos los elementos siguientes serán mayores al buscado y por lo tanto distintos de él.

Reconocimientos

El presente apunte se realizó tomando como base notas de clases, de Estructuras de la Información y de Estructuras de Datos y Algoritmos, del Ing. Hugo Ryckeboer en la Universidad Nacional de San Luis.