# Presentación

PM no es la primera asignatura de Programación que tienen que cursar los estudiantes de las Ingenierías en Informática. Por ello, suponemos que el alumno tiene un conocimiento suficiente del sistema operativo y del lenguaje de programación con los que va a trabajar y ya sabe resolver un amplio abanico de problemas con las técnicas mostradas en IP. Por otro lado, ya ha tenido que presentar otras prácticas con anterioridad y, por lo tanto, ha adquirido una idea general de cómo documentarlas.

Consecuentemente, lo que en la práctica de PM se evalúa por encima de todo es el dominio de las técnicas de diseño de programas que son propias de esta asignatura. Sin embargo, es necesario que los trabajos que se presenten (de los que, no lo olvidemos, depende un cierto porcentaje de la nota global) vengan documentados de manera que su contenido quede expuesto con la mayor claridad posible.

En esta asignatura, además, se introducen elementos nuevos (derivación de programas, diseño recursivo, uso de TADs, ...) que hay que incorporar adecuadamente al trabajo práctico. Por último, al diseñar los algoritmos se comete un deliberado olvido de la gestión de la entrada/salida, problema que se pospone hasta el momento de la codificación. Aquí es donde corresponde tratar estos aspectos. De esta forma, el alumno no sólo avanzará en su formación teórica en el campo de la Programación sino también en el aspecto práctico.

En este documento se presenta una práctica resuelta de ProMet para que sirva de guía al estudiante al realizar la suya. Se compone de los siguientes elementos:

- Enunciado.
- Especificación informal del problema.
- Primer nivel de diseño.
- Solución algorítmica formal de todas o parte de las funciones que surgen del punto anterior. Eventualmente, pueden aparecer soluciones intermedias que vayan siendo rechazadas en favor de otras obtenidas por mejora de las anteriores o bien por otros métodos.
- Traducción de la mejor solución conseguida al lenguaje de programación Java. En realidad, usaremos este ejemplo para la clase de laboratorio, de forma que aquí sólo incluiremos algunas directrices básicas de codificación.

Hemos utilizado los convenios y notación algorítmica que aparecen en el libro "Programación Metódica" (J.L. Balcázar, 1993, *McGraw-Hill*). Asimismo, esa ha sido nuestra referencia básica para los fundamentos teóricos empleados: diseño recursivo, inmersiones, derivación, etc.

Comencemos con una serie de principios generales que recomendamos tengáis en cuenta al presentar vuestros trabajos.

1. No es necesario repetir el enunciado del problema que se va a resolver. Por lo tanto, las secciones tituladas "Enunciado" se omitirían en un trabajo convencional.

2. Una buena elección de las estructuras de datos es fundamental en problemas de cierta envergadura. También lo es la correcta descomposición del código en funciones.

3. Debemos resaltar la importancia de la especificación de funciones, que debe ser rigurosa y formal. Además de ser imprescindible para la resolución de los mismos, resulta de gran ayuda para comprender mejor su enunciado.

4. El diseño de instrucciones y la corrección de las mismas deben ir de la mano. Esto es consustancial con las técnicas algorítmicas que se emplean en esta asignatura.

5. Hay que procurar que todos los identificadores que intervengan en la solución algorítmica sean descriptivos. Eso ayuda notablemente a la comprensión de tal solución.

   Si un identificador consta de varias palabras éstas pueden separarse mediante subrayado (_) o mezclando mayúsculas o minúsculas. Por ejemplo,

   obtener_sig, o bien ObtenerSig

   Eso sí, una vez adoptado un criterio, éste debe mantenerse.

   Otro convenio útil es emplear nombres parecidos para las operaciones ocultas y las funciones que las implementan, así como, en el diseño recursivo, para las inmersiones y sus funciones originales.

6. En la mayoría de los casos, el código de un programa escrito en un lenguaje de programación contiene dos tipos de instrucciones, la lectura y escritura de datos y resultados, y el cómputo propiamente dicho. El primer grupo puede obtenerse con técnicas informales, a diferencia del segundo, que debe proceder de una derivación formal.

7. En la fase de traducción de un diseño a un lenguaje de programación puede resultar conveniente introducir cambios respecto al diseño original, bien sea por limitaciones o prestaciones del propio lenguaje, razones de eficiencia, etc. En tal caso, la sección del documento destinada a exponer esta fase incluirá los comentarios oportunos.

# Satisfabilidad de expresiones booleanas

## Enunciado

Decimos que una expresión booleana sobre variables, con los operadores $\wedge$, $\vee$ y $\neg$, es satisfactible si existen valores para sus variables que la hacen cierta.

Ejemplo:

$$(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \neg x_1$$

es satisfactible pues si $x_1$ es falso, $x_2$ es cierto y $x_3$ es cierto la expresión se evalúa a cierto.

En cambio

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (x_2 \vee x_1) \wedge \neg x_1$$

es insatisfactible.

Por tanto, una manera de detectar si una expresión booleana es satisfactible es comprobar exhaustivamente todos los posibles valores para sus variables. Si para alguno se evalúa a cierto entonces es satisfactible y en otro caso no lo es.

Se pide, dada una expresión booleana contenida en una secuencia de enteros, determinar si tal expresión es satisfactible o no y, en caso de que lo sea, devolver unos valores para sus variables que la hagan cierta.

La codificación de los datos es la siguiente:

- la expresión aparece en notación prefija, es decir, la fórmula

  $$(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \neg x_1$$

  se representa por

  $$\wedge(\wedge(\vee(x_1, x_2), \vee(\neg(x_2), x_3)), \neg(x_1))$$

- las variables $x_1, x_2, \ldots, x_n$ se codifican con los valores $1, 2, \cdots, n$, mientras que las operaciones $\wedge$, $\vee$ y $\neg$ se codifican con $-1, -2$ y $-3$ respectivamente. Dado que las aridades de los operadores son conocidas podemos prescindir de los paréntesis y las comas.

Así, la secuencia que codifica la expresión

$$\wedge(\wedge(\vee(x_1, x_2), \vee(\neg(x_2), x_3)), \neg(x_1))$$

es

$$-1 \ -1 \ -2 \ 1 \ 2 \ -2 \ -3 \ 2 \ 3 \ -3 \ 1$$

Como suponemos que la secuencia siempre contiene una expresión correcta, no es necesario que la secuencia tenga marca.

El número máximo posible de variables está acotado por la constante $N$ (p.e. 20). Las variables que aparecen en una expresión están numeradas correlativamente.

Consejos prácticos (que es importante seguir):

- Después de leer el enunciado, respirad profundamente.
- Pensad las estructuras más adecuadas para almacenar tanto la expresión como las soluciones. Para la expresión, notad que los operadores booleanos son binarios ($\wedge$ y $\vee$) o unarios ($\neg$).
- Haced un primer diseño "informal" que os ayude a entender el problema, empleando los esquemas vistos en IniPro, que más tarde ya completaréis y formalizaréis.
- NO penséis en leer (y almacenar) la expresión de la secuencia hasta que tengáis todo el resto del problema resuelto. Este punto es especialmente importante.
- Para generar las soluciones tened en cuenta lo siguiente: si identificáis "falso" con el dígito 0 y "cierto" con el 1, entonces los posibles valores para $N$ variables son los números binarios de $N$ dígitos, es decir, todos los números entre 0 y $2^N - 1$. Por ejemplo, si tenemos 3 variables los posibles valores son:

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

Notad que el valor decimal de cada combinación es igual al de la anterior más uno.

# Especificación informal

Para especificar el problema usaremos una notación estilo IP, es decir, no totalmente formal pero con la suficiente precisión:

$S$ : secuencia de naturales;
$sat$ : booleano;
{$S$ contiene una expresión booleana sintácticamente correcta }
    Satisfabilidad
{$sat$ = la expresión contenida en $S$ es satisfactible }

# Primer nivel

Del enunciado se desprende que nuestro problema consiste en una búsqueda sobre la secuencia de posibles soluciones para la expresión dato, previamente leída y almacenada.

Proponemos un esquema básico

```
leer expresión
obtener primer candidato
{Inv: ninguno de los candidatos probados satisface la expresión }
{Cota: número de candidatos que quedan por probar }
mientras no solución y no último candidato hacer
    probar candidato
    obtener siguiente candidato
fmientras
```

La variable resultado, *sat*, tomará el valor cierto si algún candidato satisface la expresión.

Queda por determinar cómo se representan tanto la expresión como los candidatos a solución. Una razonable elección para éstos son los vectores de ceros y unos, con al menos tantas posiciones definidas como el número de variables tenga la expresión. Para ésta, dado que consiste en operadores, a lo sumo binarios, aplicados a su vez sobre otras expresiones, elegimos la estructura *árbol binario* y en ella almacenamos los operadores en los nodos internos y las variables en las hojas. Como convenio adicional, supondremos que la expresión sobre la que se aplica el operador "¬" se almacena en el hijo izquierdo de éste, quedando el derecho vacío.

Pasemos a refinar los asertos y las instrucciones. En la precondición aparecerá una operación oculta *correcta* que indica si una expresión es sintácticamente correcta. Para el invariante, surge otra operación oculta, *satisface*, que describe cuándo un candidato satisface una expresión.

En cuanto a las instrucciones, necesitamos sendas funciones que obtengan el primer candidato para una expresión de un determinado número de variables, y el candidato siguiente a uno dado. Las llamaremos *cero* y *siguiente*. También habrá que comprobar si de verdad un candidato es solución de una expresión. Denominaremos *fsatisface* a tal función, que implementará a la operación oculta *satisface*.

Por último introducimos dos valores naturales, el número de variables de la expresión

y el número de candidatos que llevamos probados. De ese modo, sabremos fácilmente cuando llegamos al último candidato. Para relacionar cada candidato con el número que representa emplearemos otras dos operaciones ocultas, duales, que llamaremos *binario* y *decimal*.

Así, el esquema anterior puede refinarse del siguiente modo

**var**
  $expr$ : arbin
  $c$ : vector $[1..N]$ de $\{0,1\}$
  $l, m, n$ : natural
**fvar**

{Pre }
leer_expresión$(S, expr, n)$;
{correcta$(expr) \wedge n =$ número de variables de $expr \wedge 1 \leq n \leq N$ }

$c :=$ cero$(n)$;
$m := 0$;
$l :=$ pot$(n, 2) - 1$;
$sat :=$ fsatisface$(expr, c)$;

{Inv: $sat = \exists \alpha : 0 \leq \alpha \leq m :$ satisface$(expr,$ binario$(\alpha, n)) \ \wedge \ m \leq 2^n - 1 \ \wedge \ c =$ binario$(m, n)$ }
{Cota: $2^n - 1 - m$ }
**mientras** $m < l$ **y no** $sat$ **hacer**
  $sat :=$ fsatisface$(expr,$ siguiente$(c))$;
  $m := m + 1$;
  $c :=$ siguiente$(c)$
**fmientras**;

{Post: $sat = \exists \alpha : 0 \leq \alpha \leq 2^n - 1 :$ satisface$(expr,$ binario$(\alpha, n))$ }

Notad que se puede reorganizar el cuerpo del bucle para calcular siguiente$(c)$ sólo una vez.

# Operaciones ocultas

A continuación especificamos las operaciones ocultas. En primer lugar, la que informa sobre la corrección de una expresión se define mediante el perfil

correcta : árbol $\longrightarrow$ booleano

y las ecuaciones

correcta(`a_vacío`) = falso
correcta(`plantar`$(x,$ `a_vacío`, `a_vacío`$)) = 1 \leq x \wedge x \leq N$
correcta(`plantar`$(x,$ `plantar`$(y, a_1, a_2),$ `a_vacío`$)) = (x = -3) \wedge$ correcta(`plantar`$(y, a_1, a_2)$)
correcta(`plantar`$(x,$ `a_vacío`, `plantar`$(z, b_1, b_2))) =$ falso
correcta(`plantar`$(x,$ `plantar`$(y, a_1, a_2),$ `plantar`$(z, b_1, b_2))) = (x = -1 \vee x = -2) \wedge$
  correcta(`plantar`$(y, a_1, a_2)) \wedge$ correcta(`plantar`$(z, b_1, b_2)$)

Del mismo modo, el perfil de *satisface* será

satisface : árbol, vector $\longrightarrow$ booleano

y sus ecuaciones

$\text{correcta}(\texttt{plantar}(x, a_1, a_2)) \wedge 1 \leq x \Rightarrow \text{satisface}(\texttt{plantar}(x, a_1, a_2), v) = (v[x] = 1)$
$\text{correcta}(\texttt{plantar}(-1, a_1, a_2)) \Rightarrow \text{satisface}(\texttt{plantar}(-1, a_1, a_2), v) = \text{satisface}(a_1, v) \wedge \text{satisface}(a_2, v)$
$\text{correcta}(\texttt{plantar}(-2, a_1, a_2)) \Rightarrow \text{satisface}(\texttt{plantar}(-2, a_1, a_2), v) = \text{satisface}(a_1, v) \vee \text{satisface}(a_2, v)$
$\text{correcta}(\texttt{plantar}(-3, a_1, a_2)) \Rightarrow \text{satisface}(\texttt{plantar}(-3, a_1, a_2), v) = \neg\text{satisface}(a_1, v)$

y además posee ecuaciones de error

$\text{satisface}(\texttt{a\_vacío}, v) = \text{error}$
$\neg\text{correcta}(\texttt{plantar}(x, a_1, a_2)) \Rightarrow \text{satisface}(\texttt{plantar}(x, a_1, a_2), v) = \text{error}$

Por último, para especificar *binario* y *decimal* como operaciones ocultas utilizamos cuantificadores. Sus perfiles son

binario : nat, nat $\longrightarrow$ vector
decimal : vector, nat $\longrightarrow$ nat

y sus ecuaciones

$[\text{binario}(m, n) = v] = [m = \sum_{\alpha=1}^{n} v[\alpha] 2^{\alpha-1}]$
$\text{decimal}(v, n) = \sum_{\alpha=1}^{n} v[\alpha] 2^{\alpha-1}$

# Segundo nivel

Salvo que se diga lo contrario, hay que derivar todas las funciones que involucren vectores, números o TADs. En este ejemplo podemos prescindir de *pot* pues la consideramos primitiva del lenguaje. Quedan pues *fsatisface, cero* y *siguiente.*

## La función *fsatisface*

La especificación pre/post de *fsatisface* es

**funcion** fsatisface ($a$ : arbin; $v$ : vector) **retorna** $b$ : booleano
{correcta($a$)}

{$b = \text{satisface}(a, v)$}

En primer lugar, de la precondición deducimos que el árbol dato $a$ no es vacío, con lo que $\texttt{raíz}(a), \texttt{hi}(a)$ y $\texttt{hd}(a)$ están bien definidos y podemos aplicar el *Teorema de la Descomposición Canónica de los Árboles Binarios.*

$$\neg\texttt{es\_vacío}(a) \Rightarrow a \equiv \texttt{plantar}(\texttt{raíz}(a), \texttt{hi}(a), \texttt{hd}(a))$$

Como $a$ no es vacío, la postcondición puede reescribirse así:

$b = \text{satisface}(a, v)$

$\Leftrightarrow$ ($\neg$`es_vacío`$(a)$, teorema de la descomposición canónica)

$b = \text{satisface}(\texttt{plantar}(\texttt{raíz}(a), \texttt{hi}(a), \texttt{hd}(a)), v)$

Aplicando nuevamente la definición de *correcta*, concluimos que hay cuatro posibilidades para el valor de `raíz`$(a)$, lo que nos brinda un excelente análisis por casos:

**funcion** fsatisface ($a$ : arbin; $v$ : vector) **retorna** $b$ : booleano
{correcta$(a)$}

**si**
$1 \leq \texttt{raíz}(a) \longrightarrow ?$
$[]$ `raíz`$(a) = -1 \longrightarrow ?$
$[]$ `raíz`$(a) = -2 \longrightarrow ?$
$[]$ `raíz`$(a) = -3 \longrightarrow ?$
**fsi**

{$b = \text{satisface}(a, v)$}

Ahora, podemos transformar la postcondición de una manera adecuada para cada rama

- Primera rama: $1 \leq \texttt{raíz}(a)$

$b = \text{satisface}(\texttt{plantar}(\texttt{raíz}(a), \texttt{hi}(a), \texttt{hd}(a)), v)$
$\Leftrightarrow$ (correcta$(a)$; protección )
$b = \text{satisface}(\texttt{plantar}(\texttt{raíz}(a), \texttt{vacío}, \texttt{vacío}), v)$
$\Leftrightarrow$ (protección; ec. 3 de satisface )
$b = (v[\texttt{raíz}(a)] = 1)$

Con lo que, en este caso, la postcondición se logra con la asignación

$$b := v[\texttt{raíz}(a)] = 1$$

- Segunda rama: `raíz`$(a) = -1$

$b = \text{satisface}(\texttt{plantar}(\texttt{raíz}(a), \texttt{hi}(a), \texttt{hd}(a)), v)$
$\Leftrightarrow$ (protección )
$b = \text{satisface}(\texttt{plantar}(-1, \texttt{hi}(a), \texttt{hd}(a)), v)$
$\Leftrightarrow$ (ec. 4 de satisface )
$b = \text{satisface}(\texttt{hi}(a), v) \wedge \text{satisface}(\texttt{hd}(a), v)$

Para llegar a ella, necesitamos primero obtener los valores de satisface($\texttt{hi}(a), v$) y satisface($\texttt{hd}(a), v$). Se consiguen mediante sendas llamadas a *fsatisface*, suponiendo que podemos aplicar su hipótesis de inducción. En ese caso, la postcondición se satisface con la asignación $b := b_1 \ \wedge \ b_2$.

$b_1 :=$ fsatisface($\texttt{hi}(a), v$);
$b_2 :=$ fsatisface($\texttt{hd}(a), v$);
{HI: $b_1 =$ satisface($\texttt{hi}(a), v$) $\wedge$ $b_2 =$ satisface($\texttt{hd}(a), v$)}
$b := b_1 \ \wedge \ b_2$

Para poder realizar tal suposición hay que comprobar que se cumplen las condiciones necesarias:

1. $\texttt{hi}(a)$ está bien definido. Cierto, porque $a$ no es nulo.
2. $\texttt{hi}(a)$ cumple la precondición de *fsatisface*. Cierto, porque al ser $a$ correcto, las condiciones de la protección aseguran que también se cumple correcto($\texttt{hi}(a)$).
3. Los parámetros decrecen. Como el preorden empleado para comprobar el decrecimiento ha de ser el mismo para todas las ramas, hay que esperar al final del diseño para demostrarlo.

El mismo estudio es válido para $\texttt{hd}(a)$).

- Tercera rama: $\texttt{raíz}(a) = -2$. Razonando simétricamente, obtenemos las instrucciones

$b_1 :=$ fsatisface($\texttt{hi}(a), v$);
$b_2 :=$ fsatisface($\texttt{hd}(a), v$);
{HI: $b_1 =$ satisface($\texttt{hi}(a), v$) $\wedge$ $b_2 =$ satisface($\texttt{hd}(a), v$)}
$b := b_1 \ \vee \ b_2$

- Cuarta rama: $\texttt{raíz}(a) = -3$. De manera análoga encontramos

$b_1 :=$ fsatisface($\texttt{hi}(a), v$);
{HI: $b_1 =$ satisface($\texttt{hi}(a), v$)}
$b := \neg b_1$

(Nuevamente, recordad que esto es un documento modelo y no deseamos extenderlo demasiado, pero en una práctica normal hay que desarrollar completamente **todos** los apartados)

Ahora debemos comprobar que hay un preorden en el que el tamaño de los parámetros decrece en todas las ramas tras las llamadas recursiva(s). Definimos, para todo par de árboles $a$ y $b$, y todo par de vectores $v$ y $w$

$$\langle a, v\rangle <_{\texttt{alt}} \langle b, w\rangle \Leftrightarrow \texttt{altura(a)} < \texttt{altura}(b)$$

Hay que demostrar que

$$\neg\texttt{es\_vacío}(a) \Rightarrow \langle \texttt{hi}(a), v\rangle <_{\texttt{alt}} \langle a, v\rangle$$

y

$$\neg\texttt{es\_vacío}(a) \Rightarrow \langle \texttt{hd}(a), v\rangle <_{\texttt{alt}} \langle a, v\rangle$$

Para ello contamos con los siguientes teoremas inductivos, que podemos llamar *Teoremas del decrecimiento de la altura de los árboles binarios.*

$$\neg\texttt{es\_vacío}(a) \Rightarrow \texttt{altura}(\texttt{hi}(a)) < \texttt{altura}(a)$$

y

$$\neg\texttt{es\_vacío}(a) \Rightarrow \texttt{altura}(\texttt{hd}(a)) < \texttt{altura}(a)$$

Demostremos, por ejemplo, el primero:

$\texttt{altura}(a)$
$\Leftrightarrow$ ($\neg\texttt{es\_vacío}(a)$, teorema de la descomposición canónica)
$\texttt{altura}(\texttt{plantar}(\texttt{raíz}(a), \texttt{hi}(a), \texttt{hd}(a)))$
$\Leftrightarrow$ (segunda ecuación de `altura`)
$1 + \max(\texttt{altura}(\texttt{hi}(a)), \texttt{altura}(\texttt{hi}(a)))$
$>$ (propiedad de max, aritmética)
$\texttt{altura}(\texttt{hi}(a))$

Ahora ya tenemos la desigualdad para $\texttt{hi}(a)$ (con $\texttt{hd}(a)$ la demostración es análoga)

$\langle \texttt{hi}(a), v\rangle <_{\texttt{alt}} \langle a, v\rangle$
$\Leftrightarrow$ (def. de $<_{\texttt{alt}}$)
$\texttt{altura}(\texttt{hi}(a)) < \texttt{altura}(a)$
$\Leftrightarrow$ (teorema del decrecimiento de la altura sobre $a$, que no es vacío )
cierto

La función adquiere, definitivamente, el siguiente aspecto

**funcion** fsatisface ($a$ : arbin; $v$ : vector) **retorna** $b$ : booleano
{correcta($a$)}

**var** $b_1, b_2$ : booleano
**si**
$1 \leq \texttt{raíz}(a) \ \wedge \ \texttt{raíz}(a) \leq N \ \longrightarrow \ b := v[\texttt{raíz}(a)] = 1$
[] $\texttt{raíz}(a) = -1 \ \longrightarrow \ b_1 :=$ fsatisface($\texttt{hi}(a), v$);
$b_2 :=$ fsatisface($\texttt{hd}(a), v$);
{HI: $b_1 =$ satisface($\texttt{hi}(a), v$) $\wedge$ $b_2 =$ satisface($\texttt{hd}(a), v$)}
$b := b_1 \ \wedge \ b_2$
[] $\texttt{raíz}(a) = -2 \ \longrightarrow \ b_1 :=$ fsatisface($\texttt{hi}(a), v$);
$b_2 :=$ fsatisface($\texttt{hd}(a), v$);
{HI: $b_1 =$ satisface($\texttt{hi}(a), v$) $\wedge$ $b_2 =$ satisface($\texttt{hd}(a), v$)}
$b := b_1 \ \vee \ b_2$
[] $\texttt{raíz}(a) = -3 \ \longrightarrow \ b_1 :=$ fsatisface($\texttt{hi}(a), v$);
{HI: $b_1 =$ satisface($\texttt{hi}(a), v$)}
$b := \neg b_1$
**fsi**

{$b =$ satisface($a, v$)}

## Las funciones *cero* y *siguiente*

Pasemos ahora a las funciones que obtienen los candidatos a solución. Planteemos su especificación:

**funcion** cero($n$ : natural) **retorna** $v$ : vector
{cierto}

{$v =$ binario($0, n$)}

**funcion** siguiente($w$ : vector; $n$ : natural) **retorna** $v$ : vector
{$1 \leq n \leq N \wedge$ decimal($w$) $< 2^n - 1$}

{$v =$ binario(decimal($w, n$) $+ 1, n$)}

Comencemos por *siguiente*. En primer lugar, es útil comprobar que

$$\text{decimal}(w) < 2^n - 1 \Leftrightarrow \exists \alpha : 1 \leq \alpha \leq n : w[\alpha] = 0$$

(El alumno debe incluir esta demostración en su trabajo). Este aserto garantiza que existe un resultado $v$ y lo aprovecharemos a su debido tiempo.

Emplearemos una inmersión para obtener *siguiente*. Si bien esto no es estrictamente necesario, podéis intentar un diseño recursivo directo y comprobaréis que la solución es ineficiente, con lo que de todas formas habría que realizar a continuación una inmersión de eficiencia.

Busquemos una versión débil de su postcondición. Si expandimos ésta queda:

$v = \text{binario}(\text{decimal}(w, n) + 1, n)$

$\Leftrightarrow$ (expansión)

$\sum_{\alpha=1}^{n} w[\alpha]2^{\alpha-1} + 1 = \sum_{\alpha=1}^{n} v[\alpha]2^{\alpha-1}$

$\Leftrightarrow$ (substitución de 1 por $i$)

$\sum_{\alpha=i}^{n} w[\alpha]2^{\alpha-i} + 1 = \sum_{\alpha=i}^{n} v[\alpha]2^{\alpha-i} \ \wedge \ i = 1$

$\Rightarrow$ (debilitando)

$\sum_{\alpha=i}^{n} w[\alpha]2^{\alpha-i} + 1 = \sum_{\alpha=i}^{n} v[\alpha]2^{\alpha-i} \ \wedge \ 1 \leq i \leq n$

La función de inmersión, que llamaremos *siguiente_aux*, tendrá un dato adicional, la variable $i$. Además, su precondición debe reforzarse puesto que, si originalmente el aserto que garantiza la existencia del resultado afecta a todos los dígitos de $w$, ahora sólo involucra a los que van del $i$-ésimo al $n$-ésimo. A partir de ahora emplearemos la versión alternativa de la precondición:

**funcion** siguiente_aux($w$ : vector; $n, i$ : natural) **retorna** $v$ : vector
$\{1 \leq n \leq N \wedge \exists \alpha : i \leq \alpha \leq n : w[\alpha] = 0 \wedge 1 \leq i \leq n\}$

$\{\sum_{\alpha=i}^{n} w[\alpha]2^{\alpha-i} + 1 = \sum_{\alpha=i}^{n} v[\alpha]2^{\alpha-i}\}$

En estas condiciones, es sencillo probar que si $i = 1$ entonces las dos precondiciones coinciden y también lo hacen las dos postcondiciones, lo que permite completar el diseño de *siguiente*:

**funcion** siguiente($w$ : vector; $n$ : natural) **retorna** $v$ : vector
$\{1 \leq n \leq N \wedge \exists \alpha : 1 \leq \alpha \leq n : w[\alpha] = 0\}$

$v :=$ siguiente_aux$(w, n, 1)$

$\{v = \text{binario}(\text{decimal}(w, n) + 1, n)\}$

Pasemos a *siguiente_aux*. Como $i \leq n$, podemos separar el $i$-ésimo elemento del sumatorio:

$\sum_{\alpha=i}^{n} w[\alpha]2^{\alpha-i} + 1 = \sum_{\alpha=i}^{n} v[\alpha]2^{\alpha-i}$

$\Leftrightarrow$ ($i \leq n$; sacar el $i$-ésimo sumando; asociatividad )

$(w[i] + 1) + \sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-i} = v[i] + \sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-i}$

Hay un caso claro: si $w[i] = 0$, construir un vector $v$ que satisfaga esa condición es sencillo, pues sirve el propio $w$, cambiando su posición $i$-ésima a 1. Podemos usar las asignaciones

$v := w;$
$v[i] := 1$

aunque en realidad es suficiente con asignar las posiciones $i + 1$-ésima hasta la $n$-ésima.

Si $w[i] = 1$ podemos escribir

$(w[i]+1)+\sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-i} = v[i]+\sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-i}$

$\Leftrightarrow$ $(w[i]=1)$

$2+\sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-i} = v[i]+\sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-i}$

$\Leftrightarrow$ (factor común 2; conmutatividad )

$2(\sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-(i+1)}+1) = v[i]+2(\sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-(i+1)})$

Ahora bien, esa igualdad se parece mucho a la postcondición de la llamada

$$v := \text{siguiente\_aux}(w, n, i+1)$$

Tal llamada es válida, porque

- $i+1$ es un número natural
- Su precondición viene implicada por la de la llamada anterior y el hecho de que $w[i]=1$. En efecto, si ha de haber alguna posición que valga 0 entre $i$ y $n$ y no es $i$, deberá ser alguna de la siguientes.
- Como veremos al final, los parámetros decrecen en un cierto preorden.

La postcondición obtenida, aplicando la hipótesis de inducción, es

$$\sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-(i+1)}+1 = \sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-(i+1)}$$

Podemos intentar resolver esta rama partiendo de dicha llamada. Queda la situación

$\{\text{Pre} \wedge w[i]=1\}$

$v := \text{siguiente\_aux}(w, n, i+1);$

$\{\sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-(i+1)}+1 = \sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-(i+1)}\}$

?

$\{2(\sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-(i+1)}+1) = v[i]+2(\sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-(i+1)})\}$

Ahora bien, si multiplicamos por 2 a ambos lados del aserto superior, observamos que éste coincide con el inferior, salvo por la presencia de $v[i]$. Para que coincidan del todo, $v[i]$ debe de valer 0, lo que se logra con una simple asignación. La función queda, pues,

**funcion** siguiente_aux($w$ : vector; $n, i$ : natural) **retorna** $v$ : vector
$\{1 \leq n \leq N \wedge \exists \alpha : i \leq \alpha \leq n : w[\alpha] = 0 \wedge 1 \leq i \leq n\}$

**si**
$w[i] = 0 \longrightarrow v := w;$
$\quad\quad\quad\quad\quad\quad\quad\quad v[i] := 1$
$[\!] \; w[i] = 1 \longrightarrow v :=$ siguiente_aux$(w, n, i+1);$
$\quad\quad\quad\quad\quad\quad\quad\quad v[i] := 0$
**fsi**

$\{\sum_{\alpha=i}^{n} w[\alpha]2^{\alpha-i} + 1 = \sum_{\alpha=i}^{n} v[\alpha]2^{\alpha-i}\}$

Para demostrar que los parámetros de las llamadas recursivas decrecen en algún preorden, definimos, con $v$ y $w$ vectores e $i, j, m$ y $n$ enteros

$$\langle v, m, i\rangle <_{2-3} \langle w, n, j\rangle \Leftrightarrow m - i < n - j$$

En nuestro problema hay que garantizar que

$$\langle w, n, i+1\rangle <_{2-3} \langle w, n, i\rangle$$

lo que resulta sencillo:

$\langle w, n, i+1\rangle <_{2-3} \langle w, n, i\rangle$
$\Leftrightarrow$ (definición de $<_{2-3}$)
$n - (i+1) < n - i$
$\Leftrightarrow$ (simplificación)
$0 < 1$
$\Leftrightarrow$
cierto

Queda por derivar la función *cero*. En este caso nada impide un diseño recursivo directo. Un caso que destaca por su sencillez es $n = 0$, pues entonces cualquier vector satisface la postcondición.

Si $n > 0$, podemos intentar una llamada recursiva con $n - 1$ (que es un parámetro válido, cumple la precondición y es menor que $n$) y seguramente será sencillo completar los cálculos necesarios. La situación es

**funcion** cero($n$ : natural) **retorna** $v$ : vector
{cierto}

**si** $n = 0 \longrightarrow$ continuar
$[\!]\ n > 0 \longrightarrow$ $v :=$ cero$(n-1)$;
{HI: $\sum_{\alpha=1}^{n-1} v[\alpha]2^{\alpha} = 0$}
?
**fsi**

$\{\sum_{\alpha=1}^{n} v[\alpha]2^{\alpha} = 0\}$

Ahora obtenemos las intrucciones que faltan a partir de la postcondición.

$\sum_{\alpha=1}^{n} v[\alpha]2^{\alpha} = 0$
$\Leftrightarrow$ ($n > 0$, separamos el $n$-simo sumando)
$\sum_{\alpha=1}^{n-1} v[\alpha]2^{\alpha} + v[n] = 0$
$\Leftrightarrow$ (si no modificamos las $n-1$ primeras posiciones del $v$ de HI)
$v[n] = 0$

que se logra con la correspondiente asignación.

**funcion** cero($n$ : natural) **retorna** $v$ : vector
{cierto}

**si** $n = 0 \longrightarrow$ continuar
$[\!]\ n > 0 \longrightarrow$ $v :=$ cero$(n-1)$;
{HI: $\sum_{\alpha=1}^{n-1} v[\alpha]2^{\alpha} = 0$}
$v[n] := 0$
**fsi**

$\{\sum_{\alpha=1}^{n} v[\alpha]2^{\alpha} = 0\}$

## La acción *leer_expresión*

Ya hemos comentado que no íbamos a exigir la derivación formal de *leer_expresión*, pero su diseño resulta interesante aún si lo realizamos informalmente.

Habíamos elegido representar las expresiones mediante árboles binarios ya que poseen estructuras similares, identificando los subárboles de un árbol con las subexpresiones de una expresión. De este modo, la lectura de una expresión se puede plantear recursivamente, con llamadas sobre sus subexpresiones.

Respecto al cálculo del número de variables de *expr*, éste se simplifica notablemente al aplicar el hecho de que si las variables de una expresión están numeradas correlativamente, entonces el número de éstas coincide con el máximo de sus índices. Como, además, en nuestro caso cada variable se representa por su índice, el valor que devolveremos será el máximo de las variables involucradas en la expresión.

Al leer un entero de la secuencia de entrada pueden darse cuatro casos, al contener ésta un expresión correcta

```
accion leer_expresion(ent/sal S : secuencia; sal expr : arbin; sal n : natural)
    {cierto }

    leer(c);
    si 1 ≤ c  ⟶  ?
    [] c = −1  ⟶  ?
    [] c = −2  ⟶  ?
    [] c = −3  ⟶  ?
    fsi

    {correcta(expr) ∧ n = máximo de los índices de las variables de expr }
```

Si $c = -1$, es que comienza una subexpresión del tipo $\wedge\ a\ b$, por tanto podemos leer las dos subexpresiones siguientes en sendos árboles y plantar éstos sobre $c$. El máximo de las variables de la expresión será el máximo de las variables de sus dos subexpresiones. Análogo razonamiento sirve si leemos un $-2$. Si el entero es el $-3$, leemos sólo la subexpresión siguiente y la plantamos junto a un árbol vacío sobre $c$. Las variables de la nueva expresión son las mismas que las de su subexpresión y, por tanto, también su máximo.

Si $1 \leq c$, es que hemos leído una variable o, lo que es lo mismo, una expresión formada por una variable. En ese caso, la subexpresión a devolver será $c$ plantada sobre dos árboles vacíos y el entero la propia variable.

Resulta el diseño

```
accion leer_expresion(ent/sal S : secuencia; sal expr : arbin; sal n : natural)
    {cierto }

    var
        expr1, expr2 : arbin
        n1, n2 : natural
    fvar

    leer(c);
    si
       1 ≤ c  ⟶  expr := plantar(c, a_vacío, a_vacío);
                  n := c
    [] c = −1 o c = −2  ⟶  leer_expresion(S, expr1, n1);
                           leer_expresion(S, expr2, n2);
                           expr := plantar(c, expr1, expr2);
                           n := max(n1, n2)
    [] c = −3  ⟶  leer_expresion(S, expr1, n);
                  expr := plantar(c, expr1, a_vacío)
    fsi

    {correcta(expr) ∧ n = máximo de los índices de las variables de expr }
```

# Codificación en Java

En la tercera sesión de laboratorio codificamos una parte de esta práctica en el lenguaje Java. Lo hacemos en dos fases, primero la función que evalúa la expresión sobre un candidato y, posteriormente, el bucle principal que lee la expresión y va generando y probando todos los candidatos. Se supone que todas las demás funciones ya están codificadas y se pueden importar.

En la codificación podéis incluir mejoras como convertir en alternativas las asignaciones de conjunciones y disyunciones y modificar el bucle principal para ejecutar una sóla vez la función siguiente. Para la función pot usad la biblioteca matemática del lenguaje,

La parte que ya está codificada incluye métodos para leer la expresión, obtener el primer candidato y pasar de un candidato al siguiente. Dicho código se aloja en los ficheros `Expresion.class` y `candidato.class`.

El primero define la clase `Expresion` y el método `leer_expresion`. Recordad que una expresión se representa mediante un árbol binario de enteros, pero necesitamos definir esa nueva clase porque queremos que `leer_expresion` también devuelva el número de variables de la expresión, para lo cual hemos de agrupar ambos valores en un solo objeto. También están los métodos para consultar ambos campos.

Para probar `leer_expresion` podéis ejecutar el método principal de la clase, que lee una expresión en preorden, la escribe en inorden e informa del número de variables distintas que contiene (en realidad, es el índice de la mayor variable, pero éste coincide con el número de variables si los índices son correlativos y comienzan por 1), cuántos candidatos hay y cuáles son éstos. Consultad los demás detalles en el fichero `Expresion.doc`.

Los candidatos se representan con arrays de naturales y los valores booleanos falso y cierto con los números 0 y 1. Para simular cómodamente los vectores [1..N] de la notación algorítmica, dimensionamos los arrays a [N+1] e ignoramos la posicion 0.

Por otro lado, `candidato.java` contiene los métodos `cero` y `siguiente` que obtienen el primer candidato y pasan de un candidato al siguiente, respectivamente. Las cabeceras de ambos están en el fichero `candidato.doc`.