"LOS PROGRAMAS WHILE. Bases para una teoría de la Computabilidad"

Jesús Ibáñez; Arantza Irastorza; Ana Sánchez

UPV/EHU/ LSI / TR 5-96

LOS PROGRAMAS WHILE

Bases para una Teoría de la Computabilidad

Jesús Ibáñez; Arantza Irastorza; Ana Sánchez

Indice

1. Introducción 3

2. Los programas while 7

2.1. Sintaxis de los programas while 9

2.2. Semántica de los programas while 11

2.3. While-computabilidad 16

3. Macros 25

3.1. Macrovariables 26

3.2. Macroexpresiones 28

3.3. Macrocondiciones 32

3.4. Macroestructuras de control 36

3.4.1. Macro if_then_else 36
3.4.2. Macro case 37
3.4.3. Macro for 40

3.5. El uso de las macros 41

3.6. Funciones de código 44

4. Tipos de datos 51

4.1. Implementaciones 53

4.2. Implementación de tipos simples 57

4.2.1. El tipo BOOLEAN 57
4.2.2. El tipo NATURAL 59

4.3. Uso de las implementaciones 60

4.4. Implementación de tipos compuestos 66

4.4.1. El tipo PILA 66
4.4.2. El tipo VECTOR 69

4.5. Gödelización de los programas while 73

5. Conclusiones 83

1. Introducción

El objetivo de la Informática Teórica consiste en formular y demostrar las propiedades universales de los sistemas computacionales o leyes que gobiernan los mecanismos del procesamiento de la información. Y al decir universales nos estamos refiriendo a aquellas cuyo ámbito de validez se extiende más allá de nuestra experiencia científica: no interesa establecer leyes que sólo sean aplicables a la Informática *tal como la conocemos* (con sus condicionantes científicos, económicos y culturales), sino también a la Informática *tal como puede ser en el futuro*. Por ello se esfuerza en enunciar principios que sigan siendo válidos independientemente de que nuestro punto de vista de lo que es la computación se vaya ensanchando por mor de los progresos científicos y tecnológicos.

Dentro de la Informática Teórica, la Teoría de la Computabilidad pretende estudiar los *límites teóricos* de los sistemas computacionales. Su objetivo central consiste en clasificar los problemas en **computables** e **incomputables**, donde llamamos computable a un problema si admite *solución informática*. Este objetivo puede ser (y es) abordado en dos sentidos: *en positivo*, comprobando *hasta dónde se puede llegar*, mediante el hallazgo de clases cada vez más amplias de problemas computables, y *en negativo*, determinando *de dónde no se puede pasar*, mediante el diseño de técnicas para demostrar la incomputabilidad de otras clases de problemas.

En el primer caso el método es relativamente directo, y es el mismo que se utiliza en las múltiples ramas de la Informática Aplicada: si queremos probar que algún problema es resoluble con medios informáticos, deberemos determinar cuales son dichos medios. Por ejemplo, si queremos demostrar que el problema de acertar las quinielas es computable, deberemos encontrar un algoritmo que lo demuestre. Para ello podremos utilizar todo el conocimiento del problema que nos parezca relevante, podremos permitirnos utilizar cualquier lenguaje de programación susceptible de ser ejecutado en cualquier ordenador posible, podremos utilizar todos los elementos tecnológicos que precisemos y elegir el sistema informático que nos convenga más (cuanto más poderoso, mejor) para encontrar la solución. Incluso podremos desarrollar un computador específico para resolver el problema. En definitiva, lo único importante será poder demostrar que, efectivamente, el algoritmo encontrado acierta siempre los resultados de las quinielas.

Pero la Teoría de la Computabilidad ataca el lado difícil del asunto: encontrar los límites teóricos de la Informática, los problemas cuya solución no puede ser

programada. Supongamos que queremos probar que el acertar las quinielas es incomputable. ¿Cómo podemos hacerlo?. El que una, cien o mil personas pasen uno, cien o mil años intentando acertar las quinielas por ordenador sin conseguirlo lo único que prueba es que hacerse rico no es fácil. No basta con comprobar que *ninguno de los algoritmos que se han intentado utilizar* resuelve el problema, sino que deberemos demostrar que *ninguno de los algoritmos posibles* es capaz de acertar las quinielas. Además tenemos una dificultad adicional: ni siquiera bastará con comprobar que ninguno de los programas posibles *en un computador determinado* obtiene éxito, ya que el problema podría ser computable en un ordenador muy diferente, incluso quizá en un ordenador que sólo sería tecnológicamente viable en un futuro no cercano. Para resolver estas dificultades la estrategia que se ha seguido en Teoría de la Computabilidad es la siguiente:

- se elige un *modelo abstracto de computador*, que se toma como estándar en el que deben reflejarse los demás; históricamente el más utilizado es la Máquina de Turing
- se comprueba que ningún modelo existente es mejor que la Máquina de Turing, es decir, que todo cuanto es computable en un computador (real o abstracto) también lo es en aquella; dado que la Máquina de Turing es anterior a todos los sistemas electrónicos de cómputo, esta comprobación se ha hecho sobre la marcha: cada vez que se ha diseñado o construido un nuevo ordenador, lo primero que se ha hecho es compararlo con la Máquina de Turing
- usando técnicas en su mayoría derivadas del álgebra y de la lógica matemática, se demuestra que ninguna Máquina de Turing es capaz de resolver ciertas clases de problemas; ello nos lleva a concluir que ningún modelo de ordenador conocido lo es, y que por consiguiente dichos problemas son incomputables.

La puesta en práctica de esta estrategia con muchos paradigmas diferentes ha llevado a la Informática a tres conclusiones importantes:

- a pesar de lo extraordinariamente diferentes que son entre sí, todos los modelos de computador diseñados hasta la fecha tienen exactamente las mismas capacidades de computabilidad: las que en su día se demostraron para la Máquina de Turing
- dado que la Máquina de Turing y otros modelos equivalentes son dispositivos asombrosamente simples, es forzoso reconocer que las operaciones mínimas que un ordenador necesita para tener capacidades plenas son

poquísimas; se puede concluir que la mayor parte de las características de las máquinas o de los lenguajes de programación convencionales son prescindibles: permiten hacer mejor, más rápido o más cómodamente algunas tareas, pero no consiguen hacer tareas nuevas

- es extremadamente improbable que aparezca en el futuro un tipo de ordenador que supere a los existentes (y por tanto a la Máquina de Turing) en cuanto a capacidad de resolución de problemas, al menos mientras sigamos utilizando el actual concepto de información digital.

La simplicidad de la Máquina de Turing es un elemento importante a la hora de demostrar propiedades limitantes, ya que cuanto más complicado es un sistema computacional, más costoso es (aunque sea igual de cierto) demostrar que con él no se pueden realizar ciertas tareas. Pero tiene una desventaja fundamental: su sistema de programación es extraño a los usos informáticos y por tanto resulta muy arduo utilizarla en la práctica. Para resolver este problema se han definido otros sistemas con pretensión de servir de estándar alternativo. El presente texto presenta uno de ellos: los programas while. En realidad hemos adaptado la sintaxis clásica de los mismos pensando en un currículum de Ingeniería Informática en el que Pascal (base de los programas while tradicionales) haya sido desplazado por ADA como lenguaje de aprendizaje conceptual.

Los programas while permiten resolver los mismos problemas que las máquinas de Turing (aunque el demostrarlo quede fuera de los objetivos de este texto), pero en cambio son mucho más sencillos de utilizar, sobre todo para personas que tienen una experiencia previa en la informática real, pues toman la forma de lenguaje imperativo clásico.

En los siguientes capítulos explicaremos qué son los programas while y cómo se utilizan (capítulo **2**). Dado que constituyen un lenguaje de programación muy simple, trataremos asimismo de justificar por qué hemos desechado la incorporación de instrucciones más complejas (capítulo **3**) o de tipos de datos más ricos (capítulo **4**), comprobando que ello no era necesario porque dichas instrucciones o tipos de datos son simulables con los elementos del lenguaje.

2. Los programas while

Nuestro objetivo es definir un lenguaje de programación **completo** (queremos que contenga todas los elementos esenciales) y al mismo tiempo **minimal** (no queremos que contenga ningún elemento no esencial). Un elemento no es esencial cuando puede ser eliminado del lenguaje sin reducir su potencia porque sus funciones pueden ser cubiertas (con mayor o menor eficacia) por el resto de los elementos del lenguaje, y es por tanto esencial cuando hay al menos un algoritmo que puede expresarse utilizando dicho elemento, pero no sin él. Por ejemplo, en un lenguaje de programación no recursivo que cuente con una única estructura cíclica (de tipo **loop**) y las operaciones numéricas de suma y producto, esta última no es un elemento esencial, puesto que la multiplicación puede ser programada usando el bucle y la suma. Sin embargo la estructura cíclica sí es esencial, puesto que los bucles no pueden ser simulados con otro tipo de construcciones no iterativas, y hay gran número de algoritmos que no pueden expresarse sin estructuras cíclicas.

A primera vista puede parecer que es más difícil conseguir la completitud de un lenguaje que su minimalidad: por un lado, porque podemos olvidar alguna primitiva imprescindible (sobre todo si esta sólo se necesita en casos muy rebuscados), y por otro, porque el diseño de nuevos equipos o el planteamiento de nuevos problemas pueden dejar desfasado nuestro lenguaje. Pero en la práctica esta situación no se da: no sólo es asombrosamente pequeño el número de elementos esenciales (siendo todos ellos de uso corriente), sino que desde que se definieron (allá por los años 30) nadie ha podido ampliar la lista, y además existe el convencimiento generalizado de que nadie lo conseguirá jamás. Así pues el mayor problema es lograr que el lenguaje sea realmente minimal y no contenga nada que no sea absolutamente imprescindible.

El lenguaje que vamos a definir es extraordinariamente sencillo y su sintaxis está basada en la de ADA. Pero ADA es uno de los lenguajes más barrocos que existen e incorpora infinidad de elementos que no nos interesan para nuestro lenguaje minimal, por lo que habrá que decidir cuáles son los esenciales y prescindir del resto. Concretamente estas serán las líneas principales de simplificación de nuestro lenguaje con respecto a ADA (o a cualquier otro lenguaje de alto nivel):

- sólo se utilizará un único tipo de datos: el formado por cadenas de caracteres (strings) sobre un alfabeto prefijado

- dado que todas las variables usadas en cualquier programa pertenecerán obligatoriamente a dicho tipo de datos, resultará innecesario declararlas
- además no se permitirá la utilización de identificadores arbitrarios para nombrarlas: para simplificar, cada variable tendrá un número (índice) que la identificará de forma inequívoca
- sólo se podrá utilizar como constante la cadena vacía ε, por lo que para poder manejar cualquier otra en un programa será necesario construirla a partir de operaciones sucesivas sobre ε
- las únicas operaciones primitivas consistirán en añadir o eliminar un símbolo a una cadena (siempre por la izquierda de la misma), preguntar si una cadena es vacía o preguntar por su primer símbolo
- la única instrucción que existirá es la de asignación
- sólo habrá tres estructuras de control permitidas en nuestro lenguaje, es decir, tres maneras de combinar las instrucciones de un programa: secuenciación (ejecución consecutiva de varias instrucciones), ramificación condicional (ejecución alternativa de varias instrucciones) e iteración (ejecución repetitiva de varias instrucciones)
- en la parte derecha de una asignación no se admitirán expresiones anidadas; es decir que, por ejemplo, para añadir ocho símbolos a una cadena será necesario utilizar ocho instrucciones de asignación consecutivas
- las condiciones que gobiernan las estructuras de control condicional e iterativa tampoco admitirán expresiones anidadas; de hecho las iteraciones sólo podrán utilizar la comparación con la cadena vacía, y las condicionales sólo la comprobación del primer símbolo
- nuestros programas sólo podrán devolver un único resultado (si queremos producir más de uno necesitaremos ejecutar varios programas)
- no habrá ningún procedimiento de entrada/salida: por un lado se supondrá que los datos están ya cargados en unas variables específicas al iniciar una ejecución, y por otro que el resultado es el contenido de otra variable especial en el momento de terminar la ejecución
- no se podrá definir ninguna clase de objetos de usuario (constantes, tipos, subprogramas, ...), lo cual, unido a lo que se ha comentado más arriba sobre las variables, implica que nuestros programas no necesitarán ningún tipo de declaración

2.1. Sintaxis de los programas while

Suponemos previamente definido un alfabeto $\Sigma = \{\mathbf{a_1}, \dots, \mathbf{a_n}\}$ de n símbolos. Cuando nos refiramos a un elemento genérico del alfabeto lo haremos normalmente como s. Sobre Σ se define de forma usual el conjunto de palabras o cadenas finitas de sus símbolos, denotado por Σ^*. En Σ^* siempre tenemos la cadena vacía ε (que no contiene ningún símbolo), y dadas dos cadenas cualesquiera v y w de Σ^* escribimos $v \bullet w$ para indicar la concatenación de ambas. Del mismo modo, denotamos por $|x|$ el número de símbolos que contiene la cadena x, y por x^R la palabra que resulta de invertir la cadena x.

Los programas while tomarán como datos una serie de cadenas de Σ^* (posiblemente ninguna) las manipularán y devolverán como resultado otra cadena de Σ^*. Los elementos morfológicos que van a constituir el lenguaje de los programas while son los siguientes:

- Las **variables**, que tendrán por nombre siempre una X seguida de un número natural (**X0**, **X1**, **X2**, **X185**, **X59848** son ejemplos de variables).
- La **constante ε** que denota la cadena vacía.
- Las **funciones** primitivas $\mathbf{cons_s}$ y **cdr** que los programas pueden utilizar para manipular cadenas (s representa un símbolo del alfabeto, por lo que hay tantas funciones $cons_s$ como elementos en Σ). La operación $cons_s$: $\Sigma^* \to \Sigma^*$ añade el símbolo s a la izquierda de una cadena, mientras que cdr: $\Sigma^* \to \Sigma^*$ hace lo contrario, pues elimina el primer símbolo de cualquier palabra. Formalmente:

$$\text{cons}_{\mathbf{s}}(\text{w}) = \mathbf{s} \bullet \text{w}$$

$$\text{cdr}(\text{w}) = \begin{cases} \varepsilon & \text{si w} = \varepsilon \\ \text{v} & \text{si w} = \mathbf{s} \bullet \text{v} \end{cases}$$

  Definimos arbitrariamente $cdr(\varepsilon)$ para evitar tratar condiciones de error.

- Los **predicados** primitivos $\mathbf{car_s?}$ y **nonem?** que los programas pueden utilizar para evaluar condiciones sobre cadenas (s representa un símbolo del alfabeto, por lo que hay tantos predicados $car_s?$ como elementos en Σ). $car_s?$ es cierto exclusivamente para las palabras que empiezan por el símbolo s, y *nonem?* para todas las palabras distintas de la vacía.
- Los **símbolos de puntuación :=** , **;** , **(** y **)** , que separan los distintos elementos de un programa.

- Las **palabras reservadas** **while**, **loop**, **if**, **then** y **end**, que permiten dotar al programa de estructura.

Como no vamos a utilizar ninguna clase de encabezamientos o declaraciones, un programa while sólo va a contener el cuerpo de instrucciones ejecutables. Como hay que considerar el caso de los programas formados por una única instrucción, haremos una definición inductiva expresando cuáles son los programas básicos (instrucciones elementales) y cómo se construyen programas compuestos a partir de otros preexistentes (utilizando estructuras de control).

DEFINICIÓN 1: Dado un alfabeto $\Sigma = \{a_1, \ldots, a_n\}$, un **programa while** sobre Σ se define inductivamente como sigue:

I) Si $i \in \mathbf{N}$ entonces **XI := ε;** es un programa while.

II) Si $i, j \in \mathbf{N}$ y $\mathbf{s} \in \Sigma$ entonces **XI := cons$_S$(XJ);** es un programa while.

III) Si $i, j \in \mathbf{N}$ entonces **XI := cdr (XJ);** es un programa while.

IV) Si P_1 y P_2 son programas while, entonces $\mathbf{P_1}$ $\mathbf{P_2}$ también es un programa while

V) Si $i \in \mathbf{N}$, $\mathbf{s} \in \Sigma$ y Q es un programa while, entonces también es un programa while **if car$_s$?(XI) then Q end if;**

VI) Si $i \in \mathbf{N}$ y Q es un programa while entonces **while nonem?(XI) loop Q end loop;** también es un programa while

EJEMPLO 1: Dado el alfabeto Σ = {**a**, **b**, **c**}, los siguientes son ejemplos de programas while sobre Σ:

a)
```
X0 := consa(X0); X0 := consc(X0); X0 := consb(X0);
X0 := consa(X0); X0 := consb(X0);
```

b)
```
X0 := consa(X0);
while nonem?(X0) loop
        X0 := consc(X0);
end loop;
```

c)
```
X0 := ε;
while nonem?(X1) loop
        if carb?(X1) then
                X0 := consc(X0);
        end if;
        X1 := cdr(X1);
end loop;
```

NOTA: Si $\mathbf{P_1}$, $\mathbf{P_2}$ y $\mathbf{P_3}$ son programas while, tenemos un pequeño problema al tratar de identificar el programa $\mathbf{P_1}$ $\mathbf{P_2}$ $\mathbf{P_3}$, formado por dos aplicaciones sucesivas

de la cláusula **IV**. En rigor *no es lo mismo* el programa **Q** $\mathbf{P_3}$, donde **Q** = $\mathbf{P_1}$ $\mathbf{P_2}$, que el programa $\mathbf{P_1}$ **R**, donde **R** = $\mathbf{P_2}$ $\mathbf{P_3}$, puesto que son programas formados de diferente manera y por consiguiente *distintos*. Lo que sucede es que ambos programas while, como se justificará en el siguiente apartado, *tienen exactamente el mismo comportamiento*, por lo que tampoco nos importa demasiado saber cuál de ellos es en realidad $\mathbf{P_1}$ $\mathbf{P_2}$ $\mathbf{P_3}$. Aunque en realidad no vamos a necesitar entrar demasiado en estas distinciones, a partir de ahora tomaremos el convenio de que, salvo indicación contraria, la notación $\mathbf{P_1}$ $\mathbf{P_2}$ $\mathbf{P_3}$ se refiere al programa de la forma $\mathbf{P_1}$ **R**, siendo **R** = $\mathbf{P_2}$ $\mathbf{P_3}$. Así, el programa while del ejemplo **1a** es de la forma **X0:=cons**$_{\mathbf{a}}$**(X0); P**, donde **P** es a su vez de la forma **X0:=cons**$_{\mathbf{c}}$**(X0); Q**. Por su parte **Q** es un programa while de la forma **X0:=cons**$_{\mathbf{b}}$**(X0); R**, siendo por último **R** el programa composición **X0:=cons**$_{\mathbf{a}}$**(X0); X0:=cons**$_{\mathbf{b}}$**(X0);**.

2.2. Semántica de los programas while

Una vez definida la forma de construir los programas while es necesario explicar cómo han de interpretarse (por más que intuitivamente se pueda imaginar). En este caso optamos por utilizar una semántica de tipo operacional: suponemos que el programa se ejecuta en un entorno determinado (definido exclusivamente por el contenido de las distintas variables) y hemos de describir las operaciones puntuales que el programa va realizando, y cómo se va alterando el entorno en consecuencia.

Para ello es necesario primeramente describir ese entorno, para lo cual introducimos el concepto de estado de cómputo. Después debemos indicar la influencia que la ejecución de un programa tiene sobre tal estado, y con ese objetivo se introduce la idea de cómputo o sucesión de estados de cómputo por los que pasa el programa a medida que se van ejecutando sus instrucciones individuales. Es importante distinguir cómputos finitos (aquellos que terminan tras la ejecución de su última instrucción) de cómputos infinitos (aquellos en los que las instrucciones se suceden sin fin unas a otras porque no es posible salir de algún bucle).

DEFINICIÓN 2: Diremos que un programa while **usa k+1 variables** ($k \geq 0$) cuando las variables que aparecen en su texto están incluidas en el conjunto {X0, X1, X2, …, XK}.

El número de variables usadas es un margen de seguridad que nos permite tomar un segmento contiguo que incluye todas las que realmente aparecen en el programa while. Si sabemos que usa 5 variables también sabemos que *es suficiente*

controlar la evolución de 5 valores para estudiar su comportamiento (los contenidos de las variables X0, X1, X2, X3 y X4). Pero esto no quiere decir que *sea necesario* controlar todos esos valores. Por ejemplo, el programa while **P** = **if** **car_b?(X1)** **then** **X4:=$cons_a$(X3); X0:=cdr(X1);** **end** **if**; usa 5 variables de acuerdo con nuestra definición (aunque la variable X2 no parece ser necesaria para nada).

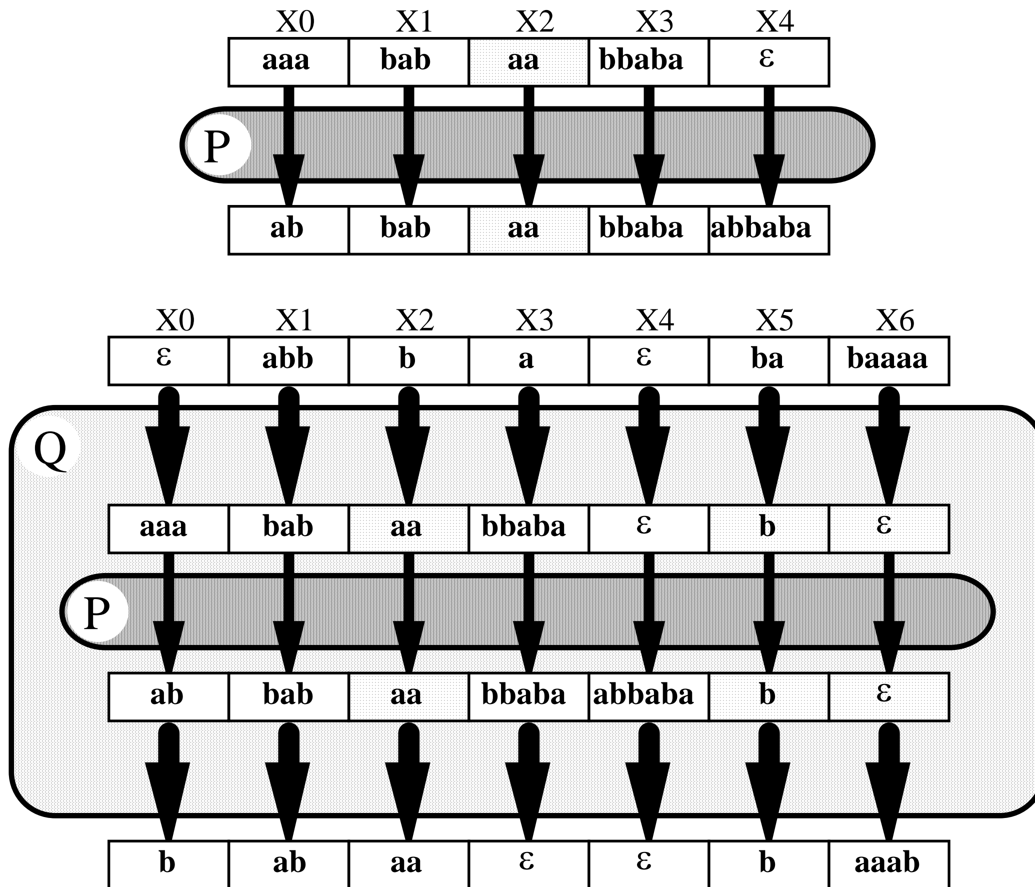


Figura 1: Se puede decir de un mismo programa while **P** que usa más o menos variables en función del contexto en el que se sitúa dicho programa. Si **P** va a ser considerado aisladamente, es suficiente con mencionar las que efectivamente aparecen en su texto (en este caso 5), pero si en realidad va a ser el subprograma de otro programa while **Q**, habrá de plegarse a las necesidades de este último, y entonces diremos que usa tantas variables como aparecen en el texto de **Q** (en este caso 7). Esto debe interpretarse como que las variables sobrantes (X5 y X6) son también variables de **P**, aunque P no tiene por misión consultar o alterar su contenido.

En este sentido es conveniente destacar que el número de variables usadas no está unívocamente determinado. Como se ha dicho, el programa while **P** citado en el párrafo anterior usa 5 variables (porque sus variables están en el conjunto {X0, X1, X2, X3, X4}), pero también es verdad, de acuerdo con la definición, que usa, por ejemplo, 7 variables (porque también están en el conjunto {X0, X1, X2, X3, X4, X5, X6}). Lo que no podemos decir es que **P** use 4 variables (porque en su texto aparece X4, que no está en la lista {X0, X1, X2, X3}). ¿Por qué permitimos esta

ambigüedad en lugar de hacer una definición más exacta que nos diga que **P** utiliza 5 variables, pero no 4 ni 6? La razón es que **P** quizá no sea más que un pequeño subprograma de otro programa while más grande **Q**, en cuyas instrucciones es posible que sí se mencionen las variables X5 y X6. Si ello es así, nos interesará expresar no sólo que **P** puede alterar el contenido de ciertas variables, sino también que *no puede alterar el contenido de las demás* (véase la figura 1).

DEFINICIÓN 3: Si Σ es un alfabeto y P un programa while sobre Σ que usa *k+1* variables, un **estado de cómputo** para P es cualquier vector de dimensión *k+1* sobre el conjunto Σ^*. El estado de cómputo indicará el contenido de las variables en un momento determinado de la ejecución de P.

DEFINICIÓN 4: Una **operación** puede ser de modificación (asignación de la forma **XI := ε**, **XI := cons$_s$(XJ)** o bien **XI := cdr (XJ)**) o de consulta (condición de la forma **nonem?(XI)** o bien **car$_s$?(XJ)**). Las operaciones de un programa while P son aquellas que aparecen en su texto, y constituyen las acciones elementales que el programa puede realizar. Por ello serán los elementos básicos de nuestra semántica operacional.

Ahora vamos a introducir una noción que nos permitirá describir el comportamiento de un programa while a lo largo del tiempo: el cómputo. Un **cómputo** es una secuencia de estados de cómputo y operaciones, que tiene las siguientes propiedades:

- su primer elemento es un estado de cómputo (lo llamamos **estado inicial**)
- los estados de cómputo y las operaciones se alternan rigurosamente
- todos los estados de la secuencia tienen la misma dimensión *k+1*
- todas las operaciones de la secuencia corresponden a un programa while concreto **P** que usa *k+1* variables
- la secuencia puede ser infinita, pero si es finita entonces su último elemento es un estado de cómputo, que solemos llamar **estado final**
- la secuencia de operaciones corresponde al orden en que son ejecutadas dentro de **P**, y la secuencia de estados de cómputo indica los consiguientes cambios en el contenido de las *k+1* variables

Es decir, que un cómputo es o bien una secuencia finita de la forma

$$V_0\ A_1\ V_1\ A_2\ V_2\ A_3\ V_3\ A_4\ \ldots\ A_n V_n$$

y entonces decimos que es **convergente**, o bien una infinita de la forma

$$V_0\ A_1\ V_1\ A_2\ V_2\ A_3\ V_3\ A_4\ ...\ A_nV_n\ A_{n+1}\ ...$$

y entonces decimos que es **divergente**. En ambos casos cada V_i representa un estado de cómputo y cada A_i una operación. Formalmente lo describimos a continuación:

DEFINICIÓN 5: Sea P un programa while y V_0 un estado de cómputo para P. El **cómputo** K asociado a P y V_0 se define inductivamente como sigue:

I) Si P es un programa de la forma **XI:=ε;**, entonces:

$$\mathbf{K = V_0}\ \text{XI:=}\varepsilon\ \mathbf{V_1}$$

donde V_1 es el vector que cumple

$$V_1[m] = \begin{cases} V_0[m] & \text{si } m \neq i \\ \varepsilon & \text{si } m = i \end{cases}$$

es decir, que coincide con V_0 en todas sus coordenadas salvo en la *i*-ésima, correspondiente a la variable XI, que ha de contener el valor ε.

II) Si P es un programa de la forma **XI:=cons$_{\mathbf{s}}$(XJ);**, entonces:

$$\mathbf{K = V_0}\ \text{XI:=cons}_{\mathbf{s}}\text{(XJ)}\ \mathbf{V_1}$$

donde V_1 es el vector que cumple

$$V_1[m] = \begin{cases} V_0[m] & \text{si } m \neq i \\ \text{cons}_{\mathbf{s}}(V_0[j]) & \text{si } m = i \end{cases}$$

de forma análoga al caso anterior.

III) Si P es un programa de la forma **XI:=cdr(XJ);**, entonces:

$$\mathbf{K = V_0}\ \text{XI:=cdr(XJ)}\ \mathbf{V_1}$$

donde V_1 es el vector definido de forma análoga a los casos anteriores:

$$V_1[m] = \begin{cases} V_0[m] & \text{si } m \neq i \\ \text{cdr}(V_0[j]) & \text{si } m = i \end{cases}$$

IV) Si P es un programa de la forma $\mathbf{P_1\ P_2}$, tenemos dos casos posibles:

1) Si $V_0\ K_1$ es el cómputo divergente asociado a P_1 y V_0 (donde K_1 es una secuencia infinita de operaciones y estados de cómputo), entonces:

$$\mathbf{K = V_0\ K_1}$$

indicando que el subprograma while P_1 cicla y P_2 no se llega a ejecutar, por lo que K resulta divergente.

2) Si $V_0\ K_1\ V_1$ es el cómputo convergente asociado a P_1 y V_0 (donde K_1 es una secuencia finita de operaciones y estados de cómputo), y $V_1\ K_2$ es el

cómputo asociado a P_2 y V_1 (donde K_2 es una secuencia finita o infinita de operaciones y estados de cómputo), entonces:

$$\mathbf{K = V_0\, K_1\, V_1\, K_2}$$

indicando que P_1 converge y P_2 es ejecutado, por lo que K resulta convergente exactamente cuando lo sea V_1 K_2.

V) Si P es un programa de la forma **if** **cars?(XI)** **then** **Q** **end** **if;**, tenemos dos casos posibles:

1) Si no es cierto car_s?(V_0[i]), entonces:

$$\mathbf{K = V_0}\ car_s?(XI)\ \mathbf{V_0}$$

indicando que la condición se evalúa sin consecuencias, pues al ser falsa en el estado V_0 el subprograma while Q no debe ejecutarse. En este caso K será convergente.

2) Si es cierto car_s?(V_0[i]), entonces sea V_0 K_1 el cómputo asociado a Q y V_0 (donde K_1 puede ser una secuencia finita o infinita de operaciones y estados de cómputo). En este caso:

$$\mathbf{K = V_0}\ car_s?(XI)\ \mathbf{V_0\, K_1}$$

indicando que, tras evaluar la condición en el estado V_0 y resultar cierta se ejecuta Q. K será divergente o no según lo sea V_0 K_1.

VI) Si P es un programa de la forma **while** **nonem?(XI)** **loop** **Q** **end** **loop;**, llamaremos V_{j+1} al estado de cómputo que satisface que V_j K_{j+1} V_{j+1} es el cómputo convergente asociado a Q y V_j. El estado de cómputo V_{j+1} puede no existir para algún valor de *j* (y por tanto tampoco para valores mayores) si el cómputo asociado es divergente. Distinguimos tres casos posibles:

1) Si existe un valor de *j* (que puede ser el propio 0) para el que no se verifica nonem?(V_j[i]), entonces sea *m* el menor valor que tiene tal propiedad, es decir, que $V_m[i]=\varepsilon \wedge \forall j\ (\ j<m \rightarrow V_j[i]\neq\varepsilon\ )$. En este caso tenemos que:

$$\mathbf{K = V_0}\ nonem?(XI)\ \mathbf{V_0\, K_1\, V_1}\ nonem?(XI)\ \mathbf{V_1\, K_2\, V_2} \ldots$$
$$\ldots\ \mathbf{V_m}\ nonem?(XI)\ \mathbf{V_m}$$

indicando que la variable XI no se anula hasta terminar la *m*-ésima iteración de Q, y que entonces se sale del bucle. K resulta convergente

2) Si ese valor no existe y el estado V_{j+1} está definido para todos los valores de *j*, entonces:

$$\mathbf{K = V_0}\ \text{nonem?(XI)}\ \mathbf{V_0\ K_1\ V_1}\ \text{nonem?(XI)}\ \mathbf{V_1\ K_2\ V_2 \dots}$$
$$\mathbf{\dots\ V_j}\ \text{nonem?(XI)}\ \mathbf{V_j\ K_{j+1}\ V_{j+1} \dots}$$

indicando que la ejecución no se termina porque ninguna de las infinitas iteraciones del subprograma while Q consigue que la condición de salida sea verdadera. Es obvio que K resulta divergente.

3) Si ese valor no existe y además el estado V_{j+1} está indefinido para algún valor de *j*, entonces sea *m* el menor valor que tiene tal propiedad. Si V_m existe y V_{m+1} no, ello se debe a que el cómputo asociado a Q y V_m es divergente, y por tanto tiene la forma V_m K_{m+1} , (donde K_{m+1} es una secuencia infinita de operaciones y estados de cómputo). En este caso:

$$\mathbf{K = V_0}\ \text{nonem?(XI)}\ \mathbf{V_0\ K_1\ V_1}\ \text{nonem?(XI)}\ \mathbf{V_1\ K_2\ V_2 \dots}$$
$$\mathbf{\dots\ V_m}\ \text{nonem?(XI)}\ \mathbf{V_m\ K_{m+1}}$$

indicando que la m+1-ésima iteración del subprograma while Q cicla. En este último caso también K resulta divergente.

EJEMPLO 2: Veamos un caso de cómputo convergente y otro de divergente. Utilizaremos el programa del ejemplo **1c** con vector de estado inicial (**ca**, **bab**):

(**ca**, **bab**) X0:=ε (ε, **bab**) nonem?(X1) (ε, **bab**) $car_{\mathbf{b}}$?(X1) (ε, **bab**)
X0:=$cons_{\mathbf{c}}$(X0) (**c**, **bab**) X1:=cdr(X1) (**c**, **ab**) nonem?(X1) (**c**, **ab**)
$car_{\mathbf{b}}$?(X1) (**c**, **ab**) X1:=cdr(X1) (**c**, **b**) nonem?(X1) (**c**, **b**) $car_{\mathbf{b}}$?(X1) (**c**, **b**)
X0:=$cons_{\mathbf{c}}$(X0) (**cc**, **b**) X1:=cdr(X1) (**cc**, ε) nonem?(X1) (**cc**, ε)

y el del ejemplo **1b** con el vector (ε, **cba**):

(ε, **cba**) X0:=$cons_{\mathbf{a}}$(X0) (**a**, **cba**) nonem?(X0) (**a**, **cba**) X0:=$cons_{\mathbf{c}}$(X0) (**ca**, **cba**)
nonem?(X0) (**ca**, **cba**) X0:=$cons_{\mathbf{c}}$(X0) (**cca**, **cba**) nonem?(X0) (**cca**, **cba**)
X0:=$cons_{\mathbf{c}}$(X0) (**ccca**, **cba**) nonem?(X0) (**ccca**, **cba**) ...

2.3. While-computabilidad

En el apartado anterior se ha explicado el comportamiento *interno* de un programa while, es decir, cómo manipula las cadenas *una vez están a su disposición* (en sus variables). Nos queda explicar el comportamiento *externo*, es decir, al empezar la computación ¿cómo se consigue introducir los datos para formar el estado de cómputo inicial?. Y, tras acabar ésta ¿cómo se interpreta el estado de cómputo final para saber el resultado obtenido?.

Antes ya se ha indicado que los programas while pueden tomar varios datos, que devuelven un único resultado y que la entrada y la salida se realizarán a

través de variables específicas. Concretamente, supondremos que cuando un programa while empieza a ejecutarse *tiene ya los valores de sus datos* cargados "automáticamente" en las variables X1, …, XN (según los que necesite), y que cuando termina deja su resultado en la variable X0. Un programa puede utilizar más variables que las mencionadas (variables de trabajo), y para evitar problemas con posibles errores de inicialización, supondremos que toda variable que no contiene un dato está inicializada con el valor ε.

Así pues, un programa while que usa *k+1* variables y toma *n* datos *actúa como si tuviera* al principio instrucciones de la forma **X0 := ε; get(X1); get(X2); ...; get(XN); X(N+1):= ε; ...; XK:= ε;**, y al final la instrucción **put(X0);**. Con este convenio estamos suponiendo que la entrada y la salida son *automáticas*, y por ello no necesitamos procedimientos de lectura o escritura.

Así, el comportamiento externo de un programa while puede describirse en términos de entrada/salida, expresando la relación que hay entre datos y resultado. Esto suele hacerse a través de una función del conjunto de posibles entradas al de posibles salidas. Como el programa while puede eventualmente ciclar (si hay cómputo divergente), esta función puede no estar definida.

NOTA: El concepto de función que vamos a utilizar es el de **función parcial**. Con ello queremos decir que puede estar definida sobre todos los datos posibles o no. Una función parcial ψ de *j* argumentos es notada como $\psi\colon \Sigma^{*j} \longrightarrow \Sigma^*$. Si ψ está definida (produce un resultado) sobre la *j*-tupla $(y_1,y_2,\ldots,y_j)$, decimos que **converge** y lo indicamos por $\psi(y_1,y_2,\ldots,y_j)\downarrow$. Si no está definida (no produce ningún valor), decimos que **diverge** y lo indicamos por $\psi(y_1,y_2,\ldots,y_j)\uparrow$. Una función parcial puede ser **total** (si converge para todos los argumentos posibles) o **no total** (si diverge para alguno).

El conjunto de valores sobre los que una función ψ converge constituye su dominio *dom(ψ)* y el conjunto de valores que produce como posibles resultados constituye su rango *ran(ψ)*. Usaremos las letras griegas φ, ψ, χ, etc., para nombrar las funciones, pero preferiremos las letras latinas f, g, h, etc., cuando *sepamos que son totales* y queramos insistir en ese hecho.

Si un programa while **P** usa *k+1* variables, el problema consiste en decidir cómo sabemos el número *n* de datos que puede recibir. De nuevo optamos por la solución más simple: el número *n* no está predeterminado por el programa **P**, que así puede ser usado de distintas formas según el número de datos que se le suministre. Así, *todo programa es en realidad varios*, que corresponden a los distintos usos que se pueden dar del mismo: un programa while no tendrá el mismo comportamiento cuando es usado con un sólo dato, con siete o simplemente sin

suministrarle entradas. Cada programa while puede ser interpretado como un evaluador de funciones, una distinta para cada posible número de argumentos.

DEFINICIÓN 6: Supongamos definido sobre el alfabeto Σ un programa while **P** que usa *k+1* variables, y sea *j≥0*. Llamamos **función j-aria computada por P** a la función parcial $\varphi_P^j : \Sigma^{*j} \longrightarrow \Sigma^*$ que describe la relación entre datos y resultado que establecen los cómputos de **P** cuando se usa con *j* entradas, y que se define de la siguiente manera sobre *j* argumentos cualesquiera $(y_1,y_2,\ldots,y_j)$:

a1) si $k \geq j$, tomamos el vector $V = (\varepsilon,y_1,y_2,\ldots,y_j, \varepsilon,\ldots,\varepsilon)$ como estado de cómputo inicial

a2) si $k \leq j$, tomamos el vector $V = (\varepsilon,y_1,y_2,\ldots,y_k)$ como estado de cómputo inicial, por lo que ignoramos los datos y_{k+1}, y_{k+2}, …, y_j que no pueden ser utilizados por **P** al no aparecer en él las variables X(K+1), X(K+2), ..., XJ

b1) si el cómputo asociado a **P** y V es convergente, y el estado de cómputo final es $V' = (z_0,z_1,\ldots,z_k)$, entonces, $\varphi_P^j(y_1,y_2,\ldots,y_j) = z_0$ (la función está definida para esos valores)

b2) si el cómputo asociado a **P** y V es divergente, entonces, $\varphi_P^j(y_1,y_2,\ldots,y_j) \uparrow$ (la función está indefinida para esos valores)

EJEMPLO 3: Sean el alfabeto Σ = {**a**, **b**, **c**} y el siguiente programa while P:

```
X0 := cons_b(X1);
X0 := cons_a(X0);
if car_c?(X2) then
    X0 := cdr(X2);
    while nonem?(X3) loop
        X0 := cons_a(X4);
        if car_c?(X4) then
            X3 := cdr(X3);
        end if;
    end loop;
end if;
```

Como cualquier otro programa while, P computa infinitas funciones. Las seis primeras (para los valores de *j* entre 0 y 5) las definimos a continuación:

$\varphi_P^0 = \mathbf{ab}$

$\varphi_P^1(u) = \mathbf{ab} \bullet u$

$$\varphi_P^2(u,v) = \begin{cases} cdr(v) & \text{si } car_{\mathbf{c}}?(v) \\ \mathbf{ab} \bullet u & \text{c.c.} \end{cases}$$

$$\varphi_P^3(u,v,x) \simeq \begin{cases} \mathbf{ab}\bullet u & \text{si } \neg car_{\mathbf{c}}?(v) \\ cdr(v) & \text{si } car_{\mathbf{c}}?(v) \square x = \varepsilon \\ \bot & \text{c.c.} \end{cases}$$

$$\varphi_P^4(u,v,x,y) \simeq \begin{cases} \mathbf{ab}\bullet u & \text{si } \neg car_{\mathbf{c}}?(v) \\ cdr(v) & \text{si } car_{\mathbf{c}}?(v) \square x = \varepsilon \\ \mathbf{a}\bullet y & \text{si } car_{\mathbf{c}}?(v) \square x \neq \varepsilon \square car_{\mathbf{c}}?(y) \\ \bot & \text{c.c.} \end{cases}$$

$$\varphi_P^5(u,v,x,y,z) \simeq \begin{cases} \mathbf{ab}\bullet u & \text{si } \neg car_{\mathbf{c}}?(v) \\ cdr(v) & \text{si } car_{\mathbf{c}}?(v) \square x = \varepsilon \\ \mathbf{a}\bullet y & \text{si } car_{\mathbf{c}}?(v) \square x \neq \varepsilon \square car_{\mathbf{c}}?(y) \\ \bot & \text{c.c.} \end{cases}$$

Todas las demás funciones devolverán el mismo resultado, aunque el número de argumentos de entrada sea diferente (los sobrantes son siempre ignorados).

NOTA: Indicaremos con el símbolo $\bot$ el resultado vacío o ausencia de resultado para una función. Es decir, que denotamos exactamente lo mismo mediante las expresiones $\psi(x)\uparrow$ y $\psi(x) \simeq \bot$ (pero sin embargo no es correcto decir que $\psi(x) \simeq \uparrow$, ya que $\uparrow$ e $\downarrow$ no actúan como valores, sino como predicados en notación posfija). Respecto al uso del símbolo de la **igualdad parcial** $\simeq$, viene motivado por el hecho de utilizar expresiones que pueden estar indefinidas. En efecto, si afirmamos que $\psi(x) = \chi(y)$, estamos diciendo *tres cosas a la vez*:

$$\psi(x) = \chi(y) \Leftrightarrow \begin{cases} \psi(x)\neg \\ \square \\ \chi(y)\neg \\ \square \\ \psi(x) \text{ y } \chi(y) \text{ son la misma palabra} \end{cases}$$

Por eso *siempre será falsa* la expresión $\psi(x) = \bot$, independientemente de que $\psi(x)$ converja o no. Para poder comparar valores posiblemente indefinidos

necesitamos la igualdad parcial, que también es cierta cuando se comparan dos expresiones indefinidas:

$$\psi(x) \simeq \chi(y) \Leftrightarrow \begin{cases} \psi(x)\neg \;\square\; \chi(y)\neg \;\square\; \psi(x) \text{ y } \chi(y) \text{ son la misma palabra} \\ \Delta \\ \psi(x) \;\square\; \chi(y) \end{cases}$$

Las igualdades también son usadas para comparar entre sí funciones completas, pero respetando los principios antes indicados: la igualdad sólo se verifica en la convergencia. Por consiguiente:

$$\psi = \chi \Leftrightarrow \forall x\ \psi(x) = \chi(x)$$

$$\psi \simeq \chi \Leftrightarrow \forall x\ \psi(x) \simeq \chi(x)$$

En el segundo caso estamos diciendo que ambas funciones coinciden cuando convergen y cuando divergen, por lo que tienen el mismo dominio y producen los mismos resultados sobre los elementos de dicho dominio. Pero en el primero estamos afirmando que ambas *convergen siempre* (son totales) y que además coinciden sobre todos los valores. Así, si ψ es una función no total, la expresión $\psi = \psi$ *no será cierta*, aunque $\psi \simeq \psi$ se cumplirá independientemente de la naturaleza de ψ.

Es de reseñar que lo dicho sobre la igualdad **=** también es cierto en general para todos los predicados que podamos definir, y en particular para la desigualdad $\neq$, que sólo se verifica cuando las dos expresiones comparadas *están definidas y corresponden a palabras distintas*:

$$\psi(x) \neq \chi(y) \Leftrightarrow \begin{cases} \psi(x)\neg \\ \square \\ \chi(y)\neg \\ \square \\ \psi(x) \text{ y } \chi(y) \text{ son palabras distintas} \end{cases}$$

Así, cuando comparamos dos expresiones indefinidas o una convergente y otra no, nos encontramos con que *no son ni iguales ni desiguales* (ni la expresión $\perp = \perp$ ni $\perp =$ **aba** son ciertas, pero tampoco lo serán sus "contrarias" $\perp \neq \perp$ y $\neg\perp \neq$ **aba**). Por ello *la desigualdad no es la negación de la igualdad*, Ahora bien, podemos usar la **desigualdad parcial**:

$$\psi(x) \not\simeq \chi(y) \Leftrightarrow \begin{cases} \psi(x)\neg \,\square\, \chi(y)\neg \,\square\, \psi(x) \text{ y } \chi(y) \text{ son palabras distintas} \\ \Delta \\ \psi(x)\neg \,\square\, \chi(y)^{-} \\ \Delta \\ \psi(x)^{-} \,\square\, \chi(y)\neg \end{cases}$$

que *sí* es la negación de la igualdad parcial ($\psi(x) \not\simeq \chi(y) \Leftrightarrow \neg\psi(x) \simeq \chi(y)$).

Para comparar funciones enteras sólo se usa esta última desigualdad, también como negación de la igualdad parcial:

$$\psi \not\simeq \chi \Leftrightarrow \exists x\, (\psi(x) \not\simeq \chi(x))$$

DEFINICIÓN 7: Una función parcial $\psi: \Sigma^{*j} \longrightarrow \Sigma^*$ decimos que es **while-computable** si existe algún programa while P que la computa, es decir, que verifica $\varphi_P^j \simeq \psi$.

EJEMPLO 4: Las siguientes funciones son while-computables:

a) Las funciones cdr y $cons_{\mathbf{s}}$ (para todo $\mathbf{s} \in \Sigma$) son while-computables. Los programas que lo prueban son, respectivamente, **X0:=cdr(X1);** y **X0:=cons$_{\mathbf{s}}$(X1);**.

b) La función vacía $\perp\!\!\perp(x_1,x_2,\ldots,x_j) \simeq \perp$, que está indefinida para cualesquiera argumentos. Un programa while para demostrar su computabilidad es el dado en el ejemplo **1b**.

c) Cualquier función constante de la forma $K_w(x_1,x_2,\ldots,x_j) = w$ para cualquier número de argumentos *j*. El programa que computa K_ε puede ser **X0:=ε;**, y si el programa **P** computa K_x, entonces el programa **P X0:=cons$_{\mathbf{s}}$(X0);** computará $K_{s\bullet x}$. El programa while del ejemplo **1a** computa la función constante $K_{\mathbf{babca}}$ para cualquier número de argumentos.

d) La función identidad, $id:\Sigma^* \to \Sigma^*$, $id(x) = x$, es computada por el siguiente programa while (donde *s* es cualquier símbolo de Σ)

```
X0 := conss(X1);
X0 := cdr(X0);
```

e) Las funciones de proyección $p_k^j:\Sigma^{*j} \to \Sigma^*$, $p_k^j(z_1, \ldots, z_j) = z_k$ con $1 \le k \le j$, que seleccionan un argumento específico, se computan de manera similar:

```
X0 := conss(XK);
X0 := cdr(X0);
```

f) La función *primero*: $\Sigma^* \to \Sigma^*$, que nos devuelve una palabra cuyo único símbolo es el primero de su argumento (por convención devuelve ε para el caso de la palabra vacía):

X0 := ε;
if $car_{\mathbf{a_1}}$(X1) **then** X0:= $cons_{\mathbf{a_1}}$(X0); **end if**;
…
if $car_{\mathbf{a_n}}$(X1) **then** X0:= $cons_{\mathbf{a_n}}$(X0); **end if**;

Dentro de la Teoría de la Computabilidad hay un subgrupo de funciones que tienen una importancia especial y suelen recibir un tratamiento específico. Son aquellas que, en lugar de estar asociadas a un *cálculo* (que pueden tener cualquier resultado o incluso la ausencia del mismo) corresponden más bien a una *decisión* (siempre producen un resultado, pero este sólo puede tomar dos valores, que se suelen asociar a una respuesta de tipo SI/NO). Son en realidad funciones booleanas o *predicados*, que devuelven el valor VERDADERO sobre unos datos y FALSO para los demás. Como los programas while no pueden producir un valor booleano porque no utilizan ese tipo de datos, hemos de acudir a un subterfugio para manejar estas funciones.

DEFINICIÓN 8: Sea R un predicado cualquiera de *j* argumentos. Llamamos **función característica del predicado R** a la función total $C_R:\Sigma^{*j} \to \Sigma^*$ definida como:

$$C_R(z_1,z_2,...,z_j) = \begin{cases} \mathbf{a_1} & \text{si } R(z_1, z_2, ..., z_j) \\ \varepsilon & \text{c.c.} \end{cases}$$

donde $\mathbf{a_1}$ es la palabra formada por el primer elemento de Σ. La función C_R sirve para detectar cuándo se cumple R sobre sus argumentos y cuándo no.

De modo similar se puede asociar a cualquier conjunto de tuplas de palabras de Σ^* un predicado que se cumple si y sólo si sus argumentos pertenecen a dicho conjunto, con lo que podemos tratarlos como predicados. De hecho no hay ninguna razón de peso para distinguir entre los conceptos de conjunto y predicado, por lo que utilizaremos ambos de forma indistinta.

DEFINICIÓN 9: Sea $A \subseteq \Sigma^{*j}$. Llamamos **función característica del conjunto A** a la función total $C_A:\Sigma^{*j} \to \Sigma^*$ definida como:

$$C_A(z_1,z_2,...,z_j) = \begin{cases} \mathbf{a_1} & \text{si } (z_1,z_2,...,z_j) \in A \\ \varepsilon & \text{c.c.} \end{cases}$$

DEFINICIÓN 10: Un conjunto o predicado *j*-ario P es **while-recursivo** si su función característica C_P: $\Sigma^{*j} \rightarrow \Sigma^*$es while-computable.

EJEMPLO 5: Los siguientes predicados son while-recursivos (damos su definición en notación de conjuntos):

- $car_{\mathbf{s}}?(x) \Leftrightarrow x \in \{ \mathbf{s} \bullet y \colon y \in \Sigma^* \}$

  X0 := ε; **if** $car_{\mathbf{s}}$?(X1) **then** X0 := $cons_{\mathbf{a_1}}$(X0); **end if**;

- $nonem?(x) \Leftrightarrow x \in \{ z \colon z \neq \varepsilon \}$

  X0 := ε; **while** nonem?(X1) **loop** X0 := $cons_{\mathbf{a_1}}$(X0); X1 := ε; **end loop**;

- $igual_comienzo?(x,y) \Leftrightarrow (x,y) \in \{ (x,y) \colon \exists \mathbf{s} \in \Sigma \ (car_{\mathbf{s}}?(x) \wedge car_{\mathbf{s}}?(y)) \}$

  X0 := ε;
  if $car_{\mathbf{a_1}}$?(X1) **then if** $car_{\mathbf{a_1}}$?(X2) **then** X0 := $cons_{\mathbf{a_1}}$(X0); **end if**; **end if**;
  if $car_{\mathbf{a_2}}$?(X1) **then if** $car_{\mathbf{a_2}}$?(X2) **then** X0 := $cons_{\mathbf{a_1}}$(X0); **end if**; **end if**;
  …
  if $car_{\mathbf{a_n}}$?(X1) **then if** $car_{\mathbf{a_n}}$?(X2) **then** X0 := $cons_{\mathbf{a_1}}$(X0); **end if**; **end if**;

3. Macros

A medida que aumenta la complejidad de los programas que hemos de construir la simplicidad del lenguaje de programación nos crea un inconveniente importante: la necesidad de programar todo desde cero. Por ejemplo, supongamos que escribimos un programa while **P** para demostrar que la función $f(x) = x^R$ es computable. **P** no tiene por qué ser excesivamente complicado: bastará con ir utilizando instrucciones **if** para averiguar los sucesivos símbolos de *x* e incorporarlos al resultado. Ahora supongamos que queremos demostrar la computabilidad de la función g(x,y) = x•y, para lo cuál escribimos otro programa while **Q**. Teniendo en cuenta que las funciones *cons* nos obligan a construir las palabras de atrás hacia adelante, habrá que empezar pasando los símbolos de *y* al resultado para después añadirle los de *x* uno a uno, del último al primero. Por tanto habremos de invertir primeramente *x*, así que es razonable suponer que **Q** incluirá como subprograma a **P**. A su vez, cualquier nuevo programa while que necesite concatenar palabras como operación intermedia deberá incluir todas las instrucciones de **Q**, y así sucesivamente.

Esta servidumbre de tener que expresar los algoritmos en términos de cinco operaciones tan elementales conlleva un aumento desproporcionado de la longitud de los programas, y consiguientemente el hastío de la persona que ha de programarlos. Naturalmente, este problema ha sido solucionado en los lenguajes de programación convencionales con la incorporación de mecanismos de modularización del código, que en su versión más sencilla se realiza mediante la definición de subrutinas y librerías.

Ahora bien, nuestro lenguaje de programación no parece necesitar ningún elemento nuevo en su sintaxis (téngase en cuenta que nos quejamos de que programar ciertas cosas puede ser pesado, no de que sea imposible hacerlo). Y si no es necesario, entonces es seguro que no nos interesa, pues nuestro lenguaje dejaría de ser mínimo. ¿Cómo podemos combinar este doble interés: por un lado querer reutilizar los programas ya diseñados sin tener que escribirlos otra vez, y por otro negarnos en redondo a ampliar el lenguaje de los programas while sin que sea estrictamente necesario?

La solución consiste en utilizar abreviaturas: mecanismos del estilo "y aquí vendría el código del programa P, que escribimos la semana pasada". Con ello no estaremos complicando la sintaxis de los programas while (las abreviaturas *no son* programas while), sino utilizando convenios que nos permitan describirlos de forma más cómoda (las abreviaturas se refieren a programas while que, aunque no

tengamos delante, sabemos *cómo podrían ser obtenidos en caso de necesidad*). Siempre deberemos tener en cuenta que lo único "real" es el programa while (escrito siguiendo estrictamente la sintaxis definida en el capítulo anterior), y por ello nunca nos arriesgaremos a introducir un programa abreviado para el que no tengamos la absoluta seguridad de la existencia de un programa while que lo respalde.

Una **macro** es un mecanismo de abreviatura para programas while. La **expansión** de una macro es el programa while que ha sido abreviado utilizando dicha macro. Un **macroprograma** es un programa escrito utilizando macros. Definiendo macros es posible escribir programas más cortos, ya que se evita la repetición de código: a medida que vamos escribiendo nuevos programas estos se van incorporando a una especie de librería que puede ser utilizada por otros programas en el futuro. Para definir una macro es necesario:

- razonar la necesidad de utilizar dicha macro
- describir su sintaxis (cómo se construye y dónde se utiliza la abreviatura)
- desarrollar su mecanismo de expansión (cómo se recuperaría el programa while auténtico que ha sido abreviado mediante la macro)

El describir la expansión de una macro en el momento de definirla es fundamental, pues es el mecanismo que nos asegura que podemos utilizar la abreviatura con tranquilidad porque siempre habrá un programa while auténtico detrás de la misma. A partir de ahora utilizaremos el término **programa** para designar indistintamente un programa while o un macroprograma. Vamos a definir cuatro tipos de macros que utilizaremos de forma extensiva en nuestros programas: macrovariables (apartado **3.1**), macroexpresiones (apartado **3.2**), macrocondiciones (apartado **3.3**) y macroestructuras de control (apartado **3.4**). Por último estableceremos una serie de convenios para una mejor utilización de las macros (apartado **3.5**).

3.1. Macrovariables

Aunque no se trate exactamente de un mecanismo de abreviatura, para no tener que controlar el número de variables usado, y también (como es costumbre en los diferentes lenguajes de programación) para utilizar nombres de variables que reflejen su cometido, querremos ocasionalmente escribir programas en los que aparezcan variables que no sean del tipo XI.

DEFINICIÓN 11: Una **macrovariable** es un identificador cualquiera según la convención de ADA (una letra seguida de una secuencia no vacía de letras, dígitos y caracteres de subrayado en la que estos últimos no pueden ir ni consecutivamente ni al final). Un macroprograma puede usar macrovariables en las mismas posiciones que las variables ordinarias, es decir, que si U y V son macrovariables, las siguientes construcciones pueden aparecer en los macroprogramas:

- U := ε;
- U := $cons_{\mathbf{s}}$(V);
- U := cdr(V);
- **if** $car_{\mathbf{s}}$?(U) **then** P **end if**;
- **while** nonem?(U) **loop** P **end loop**;

donde P también puede incluir macrovariables.

Para la entrada o la salida evitaremos usar macrovariables (seguiremos empleando las variables X1, X2, ..., XJ y X0, respectivamente, para evitar confusiones), y al mismo tiempo procuraremos extender su uso para el resto de los casos, quedando así claro en cada programa qué variables son puramente de trabajo.

La expansión correspondiente a un programa con macrovariables se obtendrá simplemente sustituyendo cada macrovariable por una nueva variable con nombre "legal". Sea **P** un macroprograma que utiliza *n+1* variables ordinarias X0, X1, ... , XN y *m* macrovariables V_1, V_2, ..., V_m. Obtenemos la expansión **Q** de **P** sustituyendo en el segundo todas las apariciones de cada macrovariable V_i por la nueva variable X(N+I). El resultado será un programa while.

EJEMPLO 6: Vamos a demostrar que la función $f(x) = x^R$ es computable, y para ello usaremos un macroprograma **P** que incorporará una variable auxiliar (la macrovariable AUX) donde copiamos la entrada para no destruirla. En este programa introduciremos tres hábitos de programación que, no siendo estrictamente necesarios, sí contribuyen a clarificar notablemente la lectura de los programas. El primero consiste en evitar alterar el contenido de las variables de entrada. El segundo, inicializar siempre las variables que no son de entrada, no utilizando el hecho de que el sistema les da por defecto el valor ε. El tercero consiste en incluir cuando sea conveniente los invariantes de bucle en forma de comentarios.

AUX := $cons_{\mathbf{a_1}}$(X1); AUX := cdr (AUX); X0 := ε;

```
- - AUX^R•X0=X1^R
    while nonem?(AUX) loop
        if car_a1?(AUX) then X0 := cons_a1(X0); end if;
        if car_a2?(AUX) then X0 := cons_a2(X0); end if;
        ...
        if car_an?(AUX) then X0 := cons_an(X0); end if;
        AUX := cdr (AUX);
    end loop;
```

Si quisiéramos encontrar la expansión de **P**, tendríamos en cuenta que usa 1+1 variables ordinarias, y que por tanto X2 puede emplearse en la sustitución. La expansión sería:

```
X2 := cons_a1(X1); X2 := cdr (X2); X0 := ε;
while nonem?(X2) loop
    if car_a1?(X2) then X0 := cons_a1(X0); end if;
    if car_a2?(X2) then X0 := cons_a2(X0); end if;
    ...
    if car_an?(X2) then X0 := cons_an(X0); end if;
    X2 := cdr (X2);
end loop;
```

Por supuesto que no es necesario expandir cada macroprograma que definimos: si así fuera no habríamos ganado nada. Lo que hacemos es, como ahora, describir de forma general el proceso de expansión de la macro en el momento de definirla, y con ello adquirimos el convencimiento que este proceso es posible en cada caso particular. Ello nos permitirá usar la macro con toda tranquilidad a partir de entonces, pues sabremos que detrás del macroprograma hay un programa while equivalente que podría ser obtenido mediante el procedimiento de expansión descrito. Si damos algunas expansiones de los ejemplos de este capítulo lo hacemos a título meramente ilustrativo, para que se comprenda mejor el mecanismo de expansión dado.

3.2. Macroexpresiones

Una vez que tenemos un programa que computa una función, es deseable para el programador o programadora no tener que repetir este mismo programa cada vez que necesita obtener el resultado proporcionado por dicha función para un cálculo intermedio. La solución es usar las funciones que ya sabemos que son **while**-computables como subrutinas que puedan ser llamadas cuando nos haga falta. Es la misma idea que cuando se usan llamadas a funciones definidas por el usuario en otros lenguajes de programación.

DEFINICIÓN 12: Sea $\psi: \Sigma^{*j} \longrightarrow \Sigma^{*}$ una función while-computable. Una **macroexpresión** es una expresión de la forma $\psi(V_1, ..., V_j)$, que se interpreta como el valor que ψ produce cuando sus argumentos son los contenidos de las variables $V_1, ..., V_j$ (que pueden ser variables ordinarias o macrovariables). Un macroprograma puede contener macroexpresiones en las asignaciones, es decir, que aparte de las asignaciones ordinarias puede contener otras del tipo:

- $U := \psi(V_1, ..., V_j)$;

La expansión del macroprograma deberá sustituir cada una de estas asignaciones por el programa while que calcula la función computable correspondiente, pero debemos tener cuidado con el uso de las variables: no podemos insertar directamente el programa que computa a ψ, pues entonces podríamos alterar datos necesarios en otros lugares del macroprograma.

Sea **P** un macroprograma que contiene macroexpresiones. La expansión de **P** se realiza en dos fases. En primer lugar cada asignación de la forma **U:=** $\boldsymbol{\psi(V_1, ..., V_j)}$**;** es sustituída por el siguiente macroprograma:

Z1 := $cons_s(V_1)$; Z1 := cdr(Z1);
…
ZJ := $cons_s(V_j)$; ZJ := cdr(ZJ);
Q_Z
U := $cons_s$(Z0); U := cdr(U);

donde **Q** es el *programa while* que computa la función ψ (que supondremos utiliza *n+1* variables), Z0, ..., ZJ son macrovariables que no aparecen en **P** ni se han usado en anteriores pasos de la expansión, y $\mathbf{Q_Z}$ es el *resultado de sustituir en **Q** cada variable XI por la macrovariable ZI* (con lo cuál su funcionamiento no interferirá con el del macroprograma en el cual está presente la macroexpresión). Para terminar, se deberán expandir las macrovariables de la forma ZI, así como cualesquiera otras que pudiera contener el programa **P** de antemano.

Obsérvese que en la expansión aparece primero la "carga" de los datos en las macrovariables (que desempeñan el papel de variables "locales"), después se ejecuta el programa propiamente dicho, y finalmente se "descarga" la salida en la variable deseada. Ha de hacerse así porque **Q** debe tomar sus datos de las variables $V_1, ..., V_j$ y no necesariamente de X1, ..., XJ, y, fundamentalmente, porque no se debe modificar el contenido de ninguna otra variable de **P**.

EJEMPLO 7: El siguiente programa computa la función f(x,y) = x•y utilizando dos asignaciones con macroexpresiones, **X0:=X2;** y **AUX:=X1$^{\mathbf{R}}$;**, correspondientes a las funciones computables identidad e inversa

```
    X0 := X2;
    AUX := X1^R;
- - AUX^R•X0=X1•X2
    while nonem?(AUX) loop
        if car_a1?(AUX) then X0 := cons_a1(X0) end if;
        if car_a2?(AUX) then X0 := cons_a2(X0) end if;
        …
        if car_an?(AUX) then X0 := cons_an(X0) end if;
        AUX := cdr (AUX);
    end loop;
```

NOTA: Adoptamos el convenio de no citar en los programas el nombre de la identidad (id), pues ello sería innecesariamente farragoso y antinatural.

El programa resultante tras la primera fase de la expansión de las macro-expresiones sería el siguiente

```
- - carga del argumento (desde X2) para calcular la identidad
    Z1 := cons_a1(X2); Z1 := cdr(Z1);
- - cálculo de la identidad
    Z0 := cons_a1(Z1); Z0 := cdr(Z0);
- - descarga del resultado (en X0) del cálculo de la identidad
    X0 := cons_a1(Z0); X0 := cdr(X0);
- - carga del argumento (desde X1) para calcular la inversa
    Y1 := cons_a1(X1); Y1 := cdr(Y1);
- - cálculo de la inversa
    Y2 := cons_a1(Y1); Y2 := cdr (Y2);
    while nonem?(Y2) loop
        if car_a1?(Y2) then Y0 := cons_a1(Y0); end if;
        ...
        if car_an?(Y2) then Y0 := cons_an(Y0); end if;
        Y2 := cdr (Y2);
    end loop;
- - descarga del resultado (en AUX) del cálculo de la inversa
    AUX := cons_a1(Y0); AUX := cdr(AUX);
- - resto del programa
    while nonem?(AUX) loop
        if car_a1?(AUX) then X0 := cons_a1(X0); end if;
        ...
        if car_an?(AUX) then X0 := cons_an(X0); end if;
        AUX := cdr (AUX);
    end loop;
```

Los programas que computan la identidad y la inversa utilizados han sido tomados de los ejemplos **4d)** y **6)** (en este último caso ha sido necesario expandirle primero su propia macrovariable).

NOTA: La función concatenación podrá a su vez ser utilizada en otras macroexpresiones, desde el momento en que hemos demostrado que es while-computable. Cuando lo hagamos, es decir, cuando aparezca en el texto de un macroprograma, utilizaremos la notación ADA habitual (símbolo **&**).

PROPOSICIÓN 1: (la **composición** de funciones while-computables es while-computable) Si las *n* funciones $\psi_1,...,\psi_n$: $\Sigma^{*j} \longrightarrow \Sigma^*$ y la función θ: $\Sigma^{*n} \longrightarrow \Sigma^*$ son todas while-computables, entonces la función composición χ: $\Sigma^{*j} \longrightarrow \Sigma^*$ definida a continuación también es while-computable:

$$\chi(y_1,...,y_j) = \theta(\psi_1(y_1,...,y_j), ... ,\psi_n(y_1,...,y_j)).$$

Demostración

El siguiente macroprograma computaría la función χ

```
Z1 := ψ1(X1, ..., XJ);
...
ZN := ψn(X1, ..., XJ);
X0 := θ(Z1, ..., ZN);
```

y haciendo la expansión de las macrovariables y las macroexpresiones se conseguiría el programa while equivalente.

EJEMPLO 8: La función $h(x,y) = x^R \bullet y \bullet \mathbf{bbb}$ es while-computable porque puede ponerse como la composición de las siguientes funciones while-computables.

$h(x,y) = \theta(\psi_1(x,y), \psi_2(x,y),\psi_3(x,y))$, donde

$\theta(u,v,w) = u \bullet v \bullet w$

$\psi_1(x,y) = x^R$

$\psi_2(x,y) = y = p_1^2(x,y)$ (while-computable)

$\psi_3(x,y) = \mathbf{bbb} = K_{\mathbf{bbb}}(x,y)$ (while-computable)

La while-computabilidad de ψ_2 y de ψ_3 ya ha sido demostrada, pero las de θ y ψ_1 no lo han sido, aunque nos pueda parecer que sí. Por ejemplo, se ha comprobado que la concatenación de *dos* palabras es while-computable, pero no que lo sea la de *tres*. Probarlo no es difícil, aunque requiere usar con tino las funciones de proyección:

$\theta(x, y, z) = x \bullet y \bullet z = \lambda(\xi_1(x,y,z), \xi_2(x,y,z))$, donde

$\lambda(u,v) = u \bullet v$ (while-computable)

$\xi_1(x,y,z) = x \bullet y$

$\xi_2(x,y,z) = z = p_3^3(x,y,z)$ (while-computable)

Por último, para ver que ξ_1 es while-computable podemos obtenerla a su vez por composición:

$\xi_1(x,y,z) = x \bullet y = \mu(\rho_1(x,y,z), \rho_2(x,y,z))$, donde

$\mu(u,v) = u \bullet v$ (while-computable)

$\rho_1(x,y,z) = x = p_1^3(x,y,z)$ (while-computable)

$\rho_2(x,y,z) = y = p_2^3(x,y,z)$ (while-computable)

Es también sencillo comprobar que ψ_1 es computable, pues se compone de la inversa y la primera proyección de entre dos componentes (p_1^2).

NOTA: El uso de macroexpresiones y macrovariables puede dar lugar a confusiones. Una macroexpresión de la forma **X2:=abb;** podría en principio querer decir dos cosas: o bien que existe una macrovariable de nombre abb cuyo contenido queremos traspasar a X2, o bien que lo que queremos almacenar en X2 es la palabra **abb** (puesto que la función $K_{\mathbf{abb}}(x) = \mathbf{abb}$ es computable, la macro sería correcta). Para evitarlo escribiremos siempre las constantes entre comillas (**X2:='abb';**) y además utilizaremos para las macrovariables sólo letras mayúsculas (**X2:=ABB;**).

3.3. Macrocondiciones

Es muy probable que para el diseño de programas más complejos resulte bastante incómoda la utilización únicamente de los predicados $car_{\mathbf{s}}$? y nonem? en las estructuras de control condicionales o iterativas. Resultaría razonable esperar que, al igual que hemos permitido abreviar el cálculo de expresiones complejas para calcular los valores de las variables, pudiéramos hacer lo mismo para evaluar condiciones complejas que guíen el control de nuestros programas.

La idea, de hecho, es parecida. Todo predicado while-recursivo R tiene asociada una función while-computable (su función característica C_R), que puede ser calculada en una variable de trabajo. Una vez hecho esto, el resultado puede ser examinado por un predicado ordinario de programa while ($car_{\mathbf{s}}$? o nonem?), y en la práctica es como si la condición hubiera sido evaluada.

DEFINICIÓN 13: Sea $R \subseteq \Sigma^{*j}$ un predicado while-recursivo. Una **macro-condición** es una expresión de la forma $R(V_1, ..., V_j)$, que se interpreta como el valor booleano que R produce cuando sus argumentos son los contenidos de las variables $V_1, ..., V_j$ (que pueden ser variables ordinarias o macrovariables). Un

macroprograma puede contener macrocondiciones en las instrucciones condicionales e iterativas, es decir, que aparte de las instrucciones ordinarias puede contener otras del tipo:

- **if** $R(V_1, ..., V_j)$ **then** Q **end if**;
- **while** $R(V_1, ..., V_j)$ **loop** Q **end loop**;

donde **Q** es otro programa, que puede incluir sus propias macrovariables, macroexpresiones y/o macrocondiciones.

Las macrocondiciones serán expandidas utilizando una macroexpresión que calcule la función característica del predicado R en una nueva macrovariable. Sea **P** un macroprograma que contiene macrocondiciones. La expansión de **P** se realiza en dos fases. Primero cada instrucción de la forma **if** $\mathbf{R(V_1, ..., V_j)}$ **then Q end if;** es sustituida por un macroprograma de la forma:

```
Z := C_R(V_1, ..., V_j);
if car_a1?(Z) then Q end if;
```

mientras que cada instrucción de la forma **while** $\mathbf{R(V_1, ..., V_k)}$ **loop Q end loop;** es sustituida por un macroprograma de la forma:

```
Z := C_R(V_1, ..., V_k);
while nonem?(Z) loop
        Q
        Z := C_R(V_1, ..., V_k);
end loop;
```

donde Z es una nueva macrovariable no utilizada en **P** (aunque puede usarse la misma Z para todas las macrocondiciones, incluso aunque estén anidadas: en cada caso se evalúa el predicado en Z justamente antes de utilizar su resultado en una condición).

Por último hay que expandir las macroexpresiones y la macrovariable así introducidas, y las que por su cuenta contuviera el propio programa **P**.

EJEMPLO 9: Como ya hemos visto, los predicados *igual_comienzo?* $\subseteq \Sigma^{*2}$ y *nonem?* $\subseteq \Sigma^*$son while-recursivos, por lo que pueden aparecer en macrocondiciones de un macroprograma. Así sucede con el siguiente, que computa la función característica del predicado *prefijo?* $\subseteq \Sigma^{*2}$, definido de la forma *prefijo?*(x,y) $\Leftrightarrow \exists w\ (y = x \bullet w)$.

```
    X0 := 'a1'; AUX1 := X1; AUX2 := X2;
- - ∃Z(Z•AUX1=X1 ∧ Z•AUX2=X2)
    while igual_comienzo?(AUX1, AUX2) loop
            AUX1 := cdr(AUX1);
```

```
        AUX2 := cdr(AUX2);
    end loop;
    if nonem?(AUX1) then X0 := ε; end if;
```

El programa resultante tras la primera fase de la expansión sería:

```
    X0 := 'a1';
    AUX1 := X1;
    AUX2 := X2;
- - evaluación de la condición
    Z := C_igual_comienzo?(AUX1, AUX2);
    while nonem?(Z) loop
        AUX1 := cdr(AUX1);
        AUX2 := cdr(AUX2);
- - reevaluación de la condición
        Z := C_igual_comienzo?(AUX1, AUX2);
    end loop;
    if nonem?(AUX1) then X0 := ε; end if;
```

La expansión continuaría con las macroexpresiones y posteriormente con las macrovariables.

Una vez definida la utilización de predicados recursivos en las macrocondiciones, podemos introducir mecanismos que nos permitan definir expresiones booleanas aún más complejas (mediante composición u operaciones lógicas). Para ello será suficiente comprobar que la aplicación de dichos mecanismos sobre predicados while-recursivos sigue produciendo predicados while-recursivos.

En cuanto a la composición se aplica un principio similar al caso de la composición de funciones de la proposición **1**, pero con predicados en lugar de funciones. La diferencia estriba en dos detalles. Por un lado un predicado puede tomar cualquier clase de datos, por lo que puede componerse con otro tipo de funciones, y no necesariamente con otros predicados. Pero al mismo tiempo un predicado es una función total, así que, para asegurarnos de que el resultado de la composición también lo es, deberemos exigir la totalidad de esas otras funciones empleadas en la composición.

PROPOSICIÓN 2: (la **composición** de un predicado recursivo con funciones computables *y totales* es un predicado recursivo) Si las funciones $f_1,\ldots,f_n\colon \Sigma^{*j} \to \Sigma^*$ son totales y while-computables, y el predicado $R \subseteq \Sigma^{*n}$ es while-recursivo, entonces el predicado $P \subseteq \Sigma^{*j}$ definido a continuación también es while-recursivo:

$$P(z_1, \ldots, z_j) \Leftrightarrow R(f_1(z_1, \ldots, z_j), \ldots, f_n(z_1, \ldots, z_j))$$

Demostración

Basta aplicar el resultado de la proposición **1** a las funciones computables C_R, f_1, …, f_n para ver que $C_P(z_1, ..., z_j) = C_R(f_1(z_1, ..., z_j), …, f_n(z_1, ..., z_j))$ es computable. Como dichas funciones C_R, f_1, …, f_n son todas totales, su composición C_P también es total. Y, por último, como C_R es una función característica, C_P también lo es.

EJEMPLO 10: El predicado *sufijo?*$\subseteq \Sigma^{*2}$, definido de la forma sufijo?(x, y) $\Leftrightarrow$ $\exists w\ (y = w \bullet x)$ es recursivo. En efecto, sufijo?(x, y) $\Leftrightarrow$ prefijo?(x^R, y^R), por lo que se cumplen las condiciones de la proposición, ya que componemos el predicado recursivo *prefijo?* con las funciones $f_1(x, y) = x^R$ y $f_2(x, y) = y^R$, cuya while-computabilidad se obtiene a su vez como composición de funciones while-computables (proyección e inversa).

PROPOSICIÓN 3: Sean $R,S \subseteq \Sigma^{*j}$ predicados while-recursivos. Entonces los predicados **conjunción** y **disyunción** de ambos, así como el predicado **negación** de uno de ellos, también lo son. Dicho de otra forma, los siguientes predicados son while-recursivos:

$T_1(y_1,...,y_j) \Leftrightarrow R(y_1,...,y_j) \wedge S(y_1,...,y_j)$

$T_2(y_1,...,y_j) \Leftrightarrow R(y_1,...,y_j) \vee S(y_1,...,y_j)$

$T_3(y_1,...,y_j) \Leftrightarrow \neg R(y_1,...,y_j)$

Demostración

El macroprograma que computaría la función característica del predicado conjunción (T_1) sería:

X0 := ε;
if nonem?(C_R(X1, ..., XJ)) **then** X0 := C_S(X1, ..., XJ); **end if**;

El correspondiente a la función característica del predicado disyunción (T_2) sería:

X0 := ε;
if nonem?(C_R(X1, ..., XJ) & C_S(X1, ..., XJ)) **then** X0 := $cons_{a_1}$(X0) **end if**;

Y, por último, para la negación (T_3):

X0 := ε; X0 := $cons_{a_1}$(X0);
if nonem?(C_R(X1, ..., XJ)) **then** X0 := ε **end if**;

NOTA: Usaremos los operadores lógicos con la terminología usual ADA: **and** para la conjunción, **or** para la disyunción y **not** para la negación

EJEMPLO 11: El predicado igualdad $= \subseteq \Sigma^{*2}$ es while-recursivo, dado que se puede escribir como la conjunción de dos predicados recursivos:

$$=(x,y) \Leftrightarrow \text{prefijo?}(x, y) \wedge \text{prefijo?}(y, x)$$

Escribiremos, como es usual, x = y en lugar de =(x, y). Y del mismo modo podemos concluir que es while-recursivo el predicado desigualdad $\neq \subseteq \Sigma^{*2}$:

$$\neq(x,y) \Leftrightarrow \neg\, x = y$$

el cual no solo escribiremos en notación infija, sino que usaremos la notación habitual ADA cuando aparezca en un programa (símbolo **/=**).

3.4. Macroestructuras de control

Una vez definidas las macroexpresiones y macrocondiciones, podemos comprobar que la tarea de programar se hace notablemente más sencilla. Sin embargo, quien esté acostumbrado a trabajar con lenguajes de programación de alto nivel como el propio ADA, echará seguramente de menos la existencia de algunas instrucciones útiles. Nuestra siguiente línea de actuación consistirá en permitirlas, pero, de nuevo, como macros o abreviaturas: los programas que contengan estas instrucciones no serán verdaderos programas while, y deberemos definir con precisión cuál es el programa while que abrevian.

3.4.1. Macro if_then_else

Probablemente la primera instrucción que se echa en falta es la condicional generalizada de la forma: **if B_1 then Q_1 elsif B_2 then Q_2 ... elsif B_n then Q_n else Q_{n+1} end if;**, donde B_1, B_2, ..., B_n son macrocondiciones y Q_1, Q_2, ..., Q_{n+1} son macroprogramas. Su semántica sería la usual: si B_1 es cierta, se ejecuta Q_1; en caso contrario, si B_2 es cierta se ejecuta Q_2, y así sucesivamente hasta Q_n; por último, si ninguna de las condiciones resulta cierta se ejecuta el programa Q_{n+1}. La manera de hacer la expansión de dicha macro es incluyendo una macrovariable Z para controlar la evaluación sucesiva de las condiciones.

Sea **P** un programa que contiene macroestructuras de control de tipo if_then_else. La expansión de **P** se realiza en dos fases. En primer lugar cada instrucción de la forma **if B_1 then Q_1 elsif B_2 then Q_2 ... elsif B_n then Q_n else Q_{n+1} end if;** es sustituida por un macroprograma de la forma:

Z := ε; Z := $cons_s$(Z);
if B_1 **then** Q_1 Z := ε; **end if**;

if B_2 **and** nonem?(Z) **then** Q_2 Z := ε; **end if**;
...
if B_n **and** nonem?(Z) **then** Q_n Z := ε; **end if**;
if nonem?(Z) **then** Q_{n+1} **end if**;

donde **s** $\in \Sigma^*$ es un símbolo cualquiera y Z es una nueva macrovariable no utilizada en **P** ni en la expansión de otra macroestructura de control.

Por último hay que expandir las macrocondiciones y la macrovariable introducidas, así como el resto de macros que tuviera el propio **P**.

PROPOSICIÓN 4: (las funciones **definidas por casos** mediante predicados recursivos y funciones computables son siempre computables) Si las *n+1* funciones $\psi_1, \psi_2, ..., \psi_{n+1}: \Sigma^{*j} \longrightarrow \Sigma^*$ son while-computables y los *n* predicados $P_1, P_2, ..., P_n \subseteq \Sigma^{*j}$ son while-recursivos e incompatibles, es decir que nunca se cumplen dos de ellos a la vez, entonces la función $\xi: \Sigma^{*j} \longrightarrow \Sigma^*$, definida por casos como sigue, también es while-computable:

$$\xi(x_1,...,x_j) = \begin{cases} \psi_1(x_1,...,x_j) & \text{si } P_1(x_1,...,x_j) \\ \psi_2(x_1,...,x_j) & \text{si } P_2(x_1,...,x_j) \\ \quad ... & \\ \psi_n(x_1,...,x_j) & \text{si } P_n(x_1,...,x_j) \\ \psi_{n+1}(x_1,...,x_j) & \text{si } \neg P_1(x_1,...,x_j) \square \neg P_2(x_1,...,x_j) \; ... \\ & \quad ... \square \neg P_n(x_1,...,x_j) \end{cases}$$

Demostración

Inmediata a partir de lo anterior. Nótese que es esencial que los predicados sean incompatibles para que la definición de ξ tenga sentido.

3.4.2. Macro case

Ocasionalmente resultará de utilidad una instrucción condicional de tipo **case**, en la que el control no va dirigido por condiciones arbitrarias, sino por los valores de una variable. Su sintaxis en ADA es **case V is when y_1 => Q_1 when y_2 => Q_2 ... when y_n => Q_n when others** => **Q_{n+1} end case;**, donde V es una variable o macrovariable, $y_1,\ldots,y_n \in \Sigma^*$ y **Q_1**, ..., **Q_{n+1}** son macroprogramas. Su semántica consiste en tomar el valor de la variable V y, según sea éste uno de los valores $y_1,\ldots,y_n$, ejecutar el programa **Q_1**, ..., **Q_n** correspondiente. En caso de no coincidir con ninguno de los valores preespecificados se ejecutaría **Q_{n+1}**. Podríamos enriquecer sin dificultad la sintaxis de la macro, permitiendo por ejemplo agrupar valores en las cláusulas *when*, pero no lo haremos porque no lo vamos a necesitar en lo sucesivo.

Para expandir un programa **P** que contenga apariciones de la macro **case** se sustituye cada una de ellas por un macroprograma de la forma:

```
if V = y1 then Q1
elsif V = y2 then Q2
...
elsif V = yn then Qn
else Qn+1
end if;
```

Posteriormente deberán expandirse el resto de macros (macroestructuras **if_then_else**, macrocondiciones, macroexpresiones y macrovariables) que pueda contener **P**.

EJEMPLO 12: Supongamos que en el alfabeto $\Sigma = \{\mathbf{a_1}, \ldots, \mathbf{a_n}\}$ los símbolos están previamente ordenados de la forma $\mathbf{a_1}<\mathbf{a_2}<\ldots<\mathbf{a_n}$. Consideramos entonces el siguiente orden inducido entre las palabras de Σ^*:

- $|x| < |y| \Rightarrow x < y$
- $\left.\begin{array}{l} |x| = |y| \\ i < j \end{array}\right\} \Rightarrow z \bullet \mathbf{a_i} \bullet x < z \bullet \mathbf{a_j} \bullet y$

Es decir, que las palabras están ordenadas primeramente de menor a mayor longitud, y dentro de la misma, por orden lexicográfico. Si, por ejemplo, el alfabeto es {**a**,**b**,**c**} y consideramos que el orden entre sus símbolos es **a**<**b**<**c**, tendremos que:

$$\varepsilon < \mathbf{a} < \mathbf{b} < \mathbf{c} < \mathbf{aa} < \mathbf{ab} < \mathbf{ac} < \mathbf{ba} < \mathbf{bb} < \mathbf{bc} < \mathbf{ca} < \mathbf{cb} < \mathbf{cc} < \mathbf{aaa} < \mathbf{aab} < \mathbf{aac} < \ldots$$

Dado que se trata de un orden total y discreto, definimos la función *sig*: $\Sigma^* \rightarrow \Sigma^*$, que dada una palabra nos devuelve la siguiente según el mismo. Su definición inductiva sería:

$$\mathrm{sig}(\varepsilon) = \text{'}\mathbf{a_1}\text{'}$$

$$\mathrm{sig}(w \bullet \mathbf{a_i}) = \begin{cases} w \bullet \mathbf{a_{i+1}} & \text{si } i < n \\ \mathrm{sig}(w) \bullet \mathbf{a_1} & \text{si } i = n \end{cases}$$

Vamos a demostrar que esta función es while-computable mediante el programa que damos a continuación:

```
    X0 := ε;
    AUX := X1^R;
 - - sig(AUX^R)•X0=sig(X1)
    while car_an?(AUX) loop
        AUX := cdr(AUX);
```

```
        X0 := cons_a1(X0);
    end loop;
    if AUX = ε then
- - sig(ε)•X0=sig(X1)
        X0 := cons_a1(X0);
    else
- - sig(AUX^R)•X0=sig(X1) ∧ nonem?(AUX) ∧ ¬car_an?(AUX)
        Z := primero(AUX);
        case Z is
        when 'a1' => X0 := cons_a2(X0);
        when 'a2' => X0 := cons_a3(X0);
        ...
        when 'an-1' => X0 := cons_an(X0);
        end case;
        X0 := (cdr(AUX))^R & X0;
    end if;
```

Además utilizando también la función *sig*, podemos demostrar un resultado más sobre funciones while-computables:

PROPOSICIÓN 5: Dada una función, f: $\Sigma^* \to \Sigma^*$ while-computable, total e inyectiva, su función inversa f^{-1}: $\Sigma^* \longrightarrow \Sigma^*$ es también while-computable.

Demostración

Es importante advertir que, aunque f^{-1} seguirá siendo inyectiva, *no tiene por qué seguir siendo total*. En efecto, si la palabra *w* no es imagen por *f* de ninguna otra (no pertenece a ran(f)), entonces $f^{-1}(w)\uparrow$. Nuestro algoritmo toma como dato la palabra *w* y va comprobando si es igual a f(ε), f(sig(ε)), f(sig(sig(ε)), etc. Si encuentra una palabra *y* que cumpla que f(y) = w, entonces la devuelve como resultado; en caso contrario, cicla en su búsqueda.

```
    AUX := ε;
- - ∀Z(Z<AUX → f(Z)≠X1)
    while f(AUX) /= X1 loop
        AUX := sig(AUX);
    end loop;
    X0 := AUX;
```

EJEMPLO 13: Ahora es fácil ver que la función *ant:* $\Sigma^* \to \Sigma^*$ que devuelve la palabra anterior según el orden citado es while-computable. Por convenio asumimos que *ant*(ε)=ε, por lo que no coincide exactamente con la inversa de la función sig, pero podemos definirla por casos utilizando un predicado recursivo y funciones computables:

$$\text{ant}(x) = \begin{cases} \varepsilon & \text{si } x = \varepsilon \\ \text{sig}^{-1}(x) & \text{c.c.} \end{cases}$$

EJEMPLO 14: También es sencillo demostrar que el predicado ≤ es while-recursivo. Para comprobarlo basta ver que su función característica es computada por el siguiente macroprograma:

```
    AUX1 := X1; AUX2 := X2;
- - X1≤AUX1 ∧ X2≤AUX2 ∧ (AUX1≤X2 ∨ AUX2≤X1)
    while AUX1 /= X2 and AUX2 /= X1 loop
        AUX1 := sig(AUX1);
        AUX2 := sig(AUX2);
    end loop;
    if AUX1 = X2 then X0 := 'a1'; else X0 := ε; end if;
```

Por último podemos ver que introducir como predicados while-recursivos el resto de comparadores usuales resulta trivial: **>** es la negación de **≤**, **≥** es la disyunción entre **>** e **=**, y **<** se obtiene como la negación de **≥**. En los programas escribiremos los comparadores en notación ADA, como de costumbre (símbolos **<=** y **>=**).

3.4.3. Macro for

Otra instrucción iterativa usual en lenguajes de programación es la sentencia *for*, que controla su comportamiento cíclico mediante el incremento progresivo del valor de una variable. Sintácticamente toma la forma **for V in A..B loop Q end loop;**, siendo V una variable o macrovariable *que no es modificada por* **Q**, A y B variables, macrovariables o macroexpresiones y **Q** un programa. El significado de la macro es ejecutar **Q** las veces que indique el rango de valores A..B. Para ello V empieza tomando el valor A y lo va incrementando hasta alcanzar el valor B (el valor que tenía al empezar a ejecutarse el *for*). Para cada valor tomado por V se ejecuta **Q** una vez. Se entiende que si el valor de A es mayor que el de B, la ejecución no tiene ningún efecto.

Utilizando la función *sig* y el predicado ≤ podemos definir la expansión de la macroestructura de control:

```
    V := A;
- - la siguiente asignación es necesaria por si B es una variable cuyo valor
    pudiera ser
- - alterado por la ejecución de Q
    Z := B;
    while V ≤ Z loop
        Q
```

```
        V := sig(V);
    end loop;
```

De manera análoga puede definirse la macro **for_reverse**, cuya sintaxis sería **for V in reverse A .. B loop Q end loop;** En este caso V tomaría inicialmente el valor B y lo iría disminuyendo hasta alcanzar el valor A (en caso de que A sea mayor que B, la macro no se ejecuta). La expansión es análoga a la del caso anterior.

3.5. El uso de las macros

Como hemos visto no se puede incluir en un programa ninguna macroexpresión que incluya una función cuya while-computabilidad no haya sido demostrada anteriormente, y análogamente no se puede utilizar ninguna macrocondición si previamente no se ha demostrado la while-recursividad del predicado que aparece en dicha condición. La apuesta es importante: si en un momento determinado hiciéramos uso de una función supuestamente computable sin comprobarlo y luego resultase incomputable, dicho programa y cuantos hiciésemos a partir de entonces basados directa o indirectamente en él serían incorrectos, y según el número, la calidad y el lugar de los errores podríamos invalidar una parte sustancial de la Teoría de la Computabilidad que estamos intentando construir.

Por ello es fundamental trabajar de modo jerárquico al utilizar los macroprogramas para demostrar la while-computabilidad de funciones y predicados. En particular es conveniente evitar una serie de prácticas de riesgo que suelen pagarse caras:

- Operar "a la inversa", usando primero la macro y dejando para después la demostración de la while-computabilidad de las funciones y/o predicados que en ella aparecen. La falta de orden puede hacer que "olvidemos" alguna demostración crucial, o lo que es peor, que hagamos demostraciones circulares: por ejemplo, provocando que la computabilidad de ψ se base en la de χ y viceversa porque cada una de ellas aparezca en una macroexpresión del programa de la otra.
- Dar por "obviamente computable" sin demostrarlo alguna función aparentemente sencilla: si es obviamente computable, seguro que no cuesta nada probarlo.

- Confundir funciones en apariencia similares pero que pueden ser muy diferentes en la práctica: hay casos en que una de ellas es while-computable y la otra no.
- Generalizar las funciones y con ellas su computabilidad. Supongamos que no hemos conseguido demostrar que la concatenación de dos palabras es while-computable, pero descubrimos un programa para probar que lo es la operación $f(x) = x \bullet x$ de concatenar una palabra consigo misma. Pues bien, seguimos sin saber absolutamente nada de la while-computabilidad de la concatenación. Es más, aunque encontráramos que para cada palabra w de Σ^* la función $g_w(x) = w \bullet x$ es while-computable, ello no demostraría *nada* de la while-computabilidad de la concatenación. El creer lo contrario es un despiste relativamente frecuente, ya que el razonamiento inverso sí se da: probando que la concatenación es while-computable automáticamente demostramos que lo son tanto f como las funciones g_w (sería un proceso de particularización justificado por la proposición **1**).
- Hacer complejas "composiciones" de funciones computables para formar expresiones anidadas sin comprobar la corrección del proceso.

Por ello es útil como buena documentación de un programa entresacar del mismo la lista de todas aquellas funciones y predicados que se van a utilizar en sus macros. Así resulta más sencillo razonar sobre su computabilidad o recursividad, bien recurriendo al programa, bien acudiendo a alguno de los resultados de las proposiciones **1** a **5** .

La estrategia que vamos a adoptar es la siguiente: imaginaremos que las funciones y predicados cuya while-computabilidad vamos demostrando con el tiempo van quedando integradas en un paquete (similar a los de ADA) que puede ser "importado" por nuestro programa, para así poder utilizar dichas funciones y predicados. La forma de hacer esto consiste en declarar al principio del programa una cabecera de dicho paquete con la especificación de las funciones y predicados (desde el punto de vista de la declaración no se distinguen) que se van a usar en sus macros, indicando el número de argumentos que implican. Esto no nos supondrá un alargamiento excesivo del macroprograma, y sin embargo será de gran utilidad para disciplinar el uso de macroexpresiones y macrocondiciones. El encabezamiento será de la forma:

package COMPUTABLES **is**
    CF_1 CF_2 ... CF_n
end COMPUTABLES;

donde CF_1, CF_2, ..., CF_n son cabeceras de función que hacen referencia a las funciones usadas en las macros, por lo que el paquete agrupa en cada caso a las funciones cuya while-computabilidad necesitamos para escribir el programa. Cada una de dichas cabeceras tiene la forma:

function fun (STRING, STRING, ..., STRING) **return** STRING;

siendo *fun*: $\Sigma^{*j} \longrightarrow \Sigma^*$ la función while-computable cuyo uso se reivindica, y repitiéndose STRING exactamente *j* veces en la lista de argumentos. Hay una serie de precisiones pertinentes al uso de cabeceras de macroprogramas:

- STRING es la forma que tenemos de aludir en los programas al conjunto Σ^*. Representa por tanto un *tipo de datos*, y su uso se justificará en el capítulo **4**.
- A diferencia de ADA, no citaremos los nombres de los parámetros formales en las cabeceras de función, sino sólo su número y tipo.
- Suele ser más adecuado citar en la cabecera las funciones atómicas que intervienen en las macros, antes de composición. Por ejemplo, si tenemos una macroexpresión de la forma **AUX:=φ(ψ(X1,M,X4),χ(X4));**, es mucho más adecuado tener tres cabeceras para las funciones φ (dos argumentos), ψ (tres argumentos), y χ (un argumento), además de otra para la proyección p_3^3 (necesaria para la composición) , que una sóla para la función resultante, que probablemente no tendrá ni nombre ni significado claro. Lo mismo se puede decir de los operadores lógicos.
- Incluimos también los predicados como funciones en la cabecera, pues al fin y al cabo eso son, funciones (totales) con rango $\{\mathbf{a_1},\varepsilon\}$ (aunque este aspecto lo modificaremos en parte en el capítulo **4**).
- Al igual que se hace en ADA para sobrecargar operadores, cuando la cabecera aluda a una función con notación especial (por ejemplo, si es infija, como la función *&* y los predicados de comparación, o posfija como la función ^{R}) se denotarán entre comillas.
- Las funciones constantes K_w y proyecciones p_k^j se denotan en cabecera como *const_w* y *proy_j_k* respectivamente.
- Si hemos de escribir varios macroprogramas que usan funciones comunes, podemos hacer una única cabecera para todos ellos.

EJEMPLO 15: Si quisiéramos escribir el siguiente macroprograma:

X0 := ε;
AUX := $X1^R$;
X0 := X3;

```
while cdr(AUX) <= F(X0) or P(X1, AUX) loop
        AUX := 'acc' & AUX;
        X0 := Ψ(X0);
end loop;
if F(X0) <= X0 then X0 := ⊥⊥ ; end if;
```

deberemos incluir la siguiente declaración:

```
package COMPUTABLES is
        function "R" (STRING) return STRING;
- - Esta cabecera corresponde a la identidad, que representamos sin símbolo
    alguno
        function "" (STRING) return STRING;
        function F (STRING) return STRING;
- - Las siguientes proyecciones son necesarias para construir la condición del
    while
        function proy_3_1 (STRING, STRING) return STRING;
        function proy_3_2 (STRING, STRING) return STRING;
        function proy_3_3 (STRING, STRING) return STRING;
        function "<=" (STRING, STRING) return STRING;
        function P (STRING, STRING) return STRING;
- - Ni la macroexpresión ni la condición del if necesitan proyecciones (sólo hay
    un argumento)
        function const_acc (STRING) return STRING;
        function "&" (STRING, STRING) return STRING;
        function Ψ (STRING) return STRING;
        function ⊥⊥ (STRING) return STRING;
end COMPUTABLES;
```

3.6. Funciones de código

A lo largo de los apartados anteriores hemos ido demostrando la while-computabilidad y while-recursividad de unas cuantas funciones y predicados que nos han sido útiles para avanzar en nuestro objetivo de ilustrar cómo la potencia expresiva de los programas while es la misma que la de otros lenguajes de programación más convencionales. Pero si bien hemos encontrado pocas dificultades para simular con nuestro lenguaje los mecanismos de programación más usuales, hay un terreno que aún no hemos tocado, a pesar de constituir una de las mayores diferencias entre los programas while y otros lenguajes: el de los tipos de objetos que manejan los programas, pues sólo permitimos cadenas de caracteres sobre un alfabeto. Este aparente vacío lo resolveremos en el capítulo **4**, pero antes demostraremos la computabilidad de algunas funciones que nos serán

de gran utilidad en el mismo. Veremos que es posible definir **funciones de codificación** que permiten "comprimir" o "agrupar" series de palabras en una sola.

Empezaremos definiendo una función de codificación para pares de palabras, $cod^2: \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$. Nuestro objetivo es tener un mecanismo de agregación para conseguir codificar cada posible par de palabras en una sóla llamada **código** del par, de forma que a partir del código podamos restituir el par original en caso de necesidad. Por ejemplo la concatenación no sería una elección adecuada como función de codificación, ya que el par (**abab**,**bb**) se codificaría en **ababbb**, pero a partir del código **ababbb** es imposible saber si el par original era (**abab**,**bb**), (**ab**,**abbb**) o cualquiera de las otras cinco posibilidades de trocear el código en dos subcadenas. De ello deducimos que es imprescindible que la función cod^2 sea *inyectiva*, es decir, que para pares diferentes produzca códigos diferentes. Además nos va a ser de utilidad que también sea sobreyectiva, porque así nos aseguramos de que *toda palabra es código de algún par* (en caso contrario se podrían producir errores al decodificar, cosa que en general tratamos de evitar).

	Σ = {**a**}	Σ = {**a**,**b**}	Σ = {**a**,**b**,**c**}
w_0	ε	ε	ε
w_1	**a**	**a**	**a**
w_2	**aa**	**b**	**b**
w_3	**aaa**	**aa**	**c**
w_4	**aaaa**	**ab**	**aa**
w_5	**aaaaa**	**ba**	**ab**
w_6	**aaaaaa**	**bb**	**ac**
w_7	**aaaaaaa**	**aaa**	**ba**
w_8	**aaaaaaaa**	**aab**	**bb**
w_9	**aaaaaaaaa**	**aba**	**bc**
w_{10}	**aaaaaaaaaa**	**abb**	**ca**

Tabla 1: Ordinales de las primeras 11 palabras para alfabetos de ejemplo con un, dos y tres símbolos.

Vista la conveniencia de que la función cod^2 sea biyectiva, vamos a emplear para definirla el método de Cantor. Consideramos de nuevo el orden de las palabras sobre cualquier alfabeto definido en el apartado **3.4.2**. Dado que es un orden total, podemos asociar a cualquier palabra un ordinal i y expresarla como w_i. Se entiende que el ordinal más bajo es el 0, y por tanto, inductivamente, $w_0 = \varepsilon$ y $w_{i+1} = sig(w_i)$ (véase la tabla **1**).

Suponemos ahora una tabla bidimensional infinita, en la que tanto las filas como las columnas están etiquetadas con las palabras ordenadas de Σ^*. Cada entrada de la tabla corresponde con un par de palabras, y por tanto una definición de la función *cod*^{2} equivale recorrer ordenadamente dicha tabla e ir asignando a cada entrada la palabra que ha de codificar el par correspondiente, de manera que cada entrada sea visitada exactamente una vez: primero colocaremos w_0 (indicando con ello qué par codifica), después w_1 y así sucesivamente con cada una de las palabras. De entre las muchas maneras satisfactorias que existen de realizar dicho recorrido elegiremos la siguiente: empezamos por la casilla superior izquierda, correspondiente al par (ε, ε), y despúes rellenaremos, de menor a mayor, las diagonales de la tabla en el sentido SO-NE, tal y como muestra la tabla **2**.

	$\mathbf{w_0}$	$\mathbf{w_1}$	$\mathbf{w_2}$	$\mathbf{w_3}$	$\mathbf{w_4}$	**...**	$\mathbf{w_j}$	**...**
$\mathbf{w_0}$	w_0	w_2	w_5	w_9	w_{14}	...		...
$\mathbf{w_1}$	w_1	w_4	w_8	w_{13}	w_{19}	...		...
$\mathbf{w_2}$	w_3	w_7	w_{12}	w_{18}	w_{25}	...		...
$\mathbf{w_3}$	w_6	w_{11}	w_{17}	w_{24}	w_{32}	...		...
$\mathbf{w_4}$	w_{10}	w_{16}	w_{23}	w_{31}	w_{40}	...		...
$\mathbf{w_5}$	w_{15}	w_{22}	w_{30}	w_{39}	w_{49}	...		...
$\mathbf{w_6}$	w_{21}	w_{29}	w_{38}	w_{48}	w_{59}	...		...
...	...	...	...	...	...	...		...
$\mathbf{w_i}$							$\mathbf{w_k}$	...
...	...	...	...	...	...	...	...	...

Tabla 2: Esquema de la definición de la función cod^2. La fila indica el primer elemento del par a codificar, la columna el segundo y la entrada de la tabla el código. Por ejemplo, el código del par (w_5,w_3) será w_{39}. En el caso particular del alfabeto {**a**,**b**} esto se traduce en que el código del par (**ba**,**aa**) es la palabra **aaaba**.

Siguiendo este esquema puede hacerse una definición inductiva de la función *cod*^{2}. Para ello hemos de tener en cuenta tres casos: el básico, el principio de una nueva diagonal (que se construye por referencia al fin de la anterior) y el del resto de situaciones:

- cod^2(ε, ε) = ε
- cod^2(sig(w), ε) = sig(cod^2(ε, w))

- $cod^2(v, sig(w)) = sig(cod^2(sig(v), w)$

Por el método seguido para rellenar la tabla podemos asegurar que la función *cod*^{2} es *total* y *biyectiva*: Total, porque toda casilla es alcanzada alguna vez (a todo par de palabras le corresponde un código); inyectiva, porque al hacerse de forma ordenada no hay posibilidad de asignar la misma palabra a dos casillas diferentes (pares distintos tienen códigos distintos); sobreyectiva, porque toda palabra caerá en su momento en alguna casilla (toda palabra es código de algún par).

Por último, la función es *while-computable*, ya que podemos construir un programa siguiendo la misma idea que utilizamos al rellenar la tabla: para calcular el código de un par dado (w_i, w_j) empezamos calculando el de (w_0, w_0), luego el de (w_1, w_0), (w_0, w_1), (w_2, w_0), (w_1, w_1), (w_0, w_2), y así sucesivamente, recorriendo las diagonales de la tabla de forma ordenada hasta llegar a generar el código del par de entrada. Como indicamos en el capítulo anterior, el macroprograma correspondiente requiere una cabecera con la declaración de todas las funciones computables que incorpora:

```
package COMPUTABLES is
        function "/=" (STRING, STRING) return STRING;
- - Las siguientes proyecciones son necesarias para construir la condición del
    while,
- - dado que el or combina predicados que actúan sobre los mismos
    argumentos.
- - Por tanto las desigualdades de la condición deben considerarse como
    predicados
- - de cuatro argumentos de los cuales cada una sólo "usa" dos
        function proy_4_1 (STRING, STRING) return STRING;
        function proy_4_2 (STRING, STRING) return STRING;
        function proy_4_3 (STRING, STRING) return STRING;
        function proy_4_4 (STRING, STRING) return STRING;
        function "=" (STRING, STRING) return STRING;
- - La función identidad es necesaria para componer la condición del if
        function ""(STRING) return STRING;
- - La siguiente función no es necesaria para las primeras asignaciones (dado
    que es
- - primitiva), pero sí para la condición del if, y por eso la tomamos de un
    argumento
        function const_ε (STRING) return STRING;
        function sig (STRING) return STRING;
        function ant (STRING) return STRING;
end COMPUTABLES ;
```

El macroprograma correspondiente sería el siguiente:

```
    LINEA := ε;
    COLUMNA := ε;
    X0 := ε;
- - X0 = cod2(LINEA, COLUMNA)
    while LINEA /= X1 or COLUMNA /= X2 loop
        if LINEA = ε then
- - En este caso hemos llegado al final de una diagonal y tenemos que empezar
    la siguiente
            LINEA := sig(COLUMNA);
            COLUMNA := ε;
        else
- - En este caso seguimos avanzando por la misma diagonal
            LINEA := ant(LINEA);
            COLUMNA := sig(COLUMNA);
        end if;
        X0 := sig(X0);
    end loop;
```

Si bien la while-computabilidad de cod^2 queda fuera de toda duda con este programa, el algoritmo utilizado no resulta muy práctico si queremos calcular algún código concreto, aunque sólo sea por curiosidad. Para encontrar otro modo más directo haremos lo siguiente:

- Numeramos las diagonales de menor a mayor, llamando 0-ésima a la que sólo contiene la casilla del par (w_0, w_0).
- La diagonal k-ésima contiene entonces *k+1* elementos.
- Los pares cuyas casillas están en la k-ésima diagonal son los de la forma (w_i, w_j) con $i+j = k$, siendo el primero de cada diagonal el par (w_k, w_0).
- Para calcular el subíndice del código w_k de un par (w_i, w_j) que se encuentra en la diagonal (i+j)-ésima, hemos de sumar primero el número de casillas que contienen las diagonales anteriores (de la 0-ésima a la (i+j-1)-ésima), y después le hemos de añadir las casillas de la diagonal del par anteriores a la suya (la suya propia no debe contarse, porque se ha contado la casilla correspondiente al código w_0).

Por todo ello, una forma (numérica) más rápida de calcular *k* sería:

$$k = \frac{(i+j) * (i+j+1)}{2} + j$$

Por otro lado, como la función cod^2 es biyectiva, podemos definir dos funciones inversas de **decodificación** $decod_{2,1}$, $decod_{2,2}$: $\Sigma^* \rightarrow \Sigma^*$ para recuperar ambos componentes del par codificado a partir de su código.

Lo que se pretende hacer cierta la ecuación:

$$cod^2 (decod_{2,1} (w), decod_{2,2} (w)) = w$$

Estas funciones también serán totales, sobreyectivas y while-computables. El programa que las computa se construye análogamente al de la función cod^2, pero teniendo en cuenta que esta vez partimos del código del par. La idea es ir construyendo pares sucesivos hasta dar con el bueno, en cuyo momento se devuelve el valor adecuado: línea (para $decod_{2,1}$) o columna (para $decod_{2,2}$):

```
package COMPUTABLES is
    function "/=" (STRING, STRING) return STRING;
    function "=" (STRING, STRING) return STRING;
    function const_ε (STRING) return STRING;
    function "" (STRING) return STRING;
    function sig (STRING) return STRING;
    function ant (STRING) return STRING;
end COMPUTABLES;
```

```
    LINEA := ε; COLUMNA := ε; AUX := ε;
- - X0 = cod2(LINEA, COLUMNA)
    while AUX /= X1 loop
        if LINEA = ε then
            LINEA := sig(COLUMNA);
            COLUMNA := ε;
        else LINEA := ant(LINEA );
            COLUMNA := sig(COLUMNA);
        end if;
        AUX := sig(AUX);
    end loop;
- - solo para decod2,1. Para decod2,2 esta última instrucción sería
    X0:=COLUMNA;
    X0 := LINEA;
```

Una vez que sabemos cómo codificar dos palabras en una sola podemos buscar la codificación de cualquier número de palabras aplicando cod^2 sucesivamente: solo hemos de acordar en qué orden. Definimos para cada $k \geq 1$ una función de código $cod^k\colon\ \Sigma^{*k} \rightarrow \Sigma^*$ de manera inductiva sobre el número de argumentos:

- $cod^1(z) = z$
- $cod^{k+1}(z_1, \ldots, z_{k+1}) = cod^2(z_1, cod^k(z_2, \ldots, z_{k+1}))$

Es decir, para codificar cualquier tupla vamos aplicando sucesivamente la función cod^2 empezando por las dos últimas palabras de la tupla:

$$cod^k(z_1, z_2, \ldots z_{k-1}, z_k) = cod^2(z_1, cod^2(z_2, \ldots cod^2(z_{k-1}, z_k)\ldots))$$

Toda función de codificación de la forma cod^{k+1} será total, biyectiva y while-computable siempre que lo sea la correspondiente cod^{k} (ya que para los otros componentes de la composición, cod^{2} e *id*, ya está demostrado). Como cod^{1} cumple las tres condiciones, podemos razonar inductivamente la totalidad, biyectividad y while-computabilidad de toda la familia de funciones.

También en este caso podemos definir funciones de decodificación $decod_{k,i}$: $\Sigma^* \rightarrow \Sigma^*$, que nos devuelvan el i-ésimo elemento de la k-tupla codificada por una cierta palabra, y que también serán totales, sobreyectivas y while-computables, ya que:

$$\text{decod}_{k,i}(w) = \begin{cases} w & \text{si } i=1 \;\square\; k=1 \\ \text{decod}_{2,1}(w) & \text{si } i=1 \;\square\; k>1 \\ \text{decod}_{k-1,i-1}(\text{decod}_{2,2}(w)) & \text{si } i>1 \;\square\; k>1 \end{cases}$$

y de nuevo se puede razonar inductivamente:

- todas las funciones de la forma $decod_{k,1}$ son while-computables, totales y sobreyectivas (por serlo *id* y $decod_{2,1}$)
- si para un cierto valor de j todas las funciones de la forma $decod_{k,j}$ son while-computables, totales y sobreyectivas, también lo son todas las de la forma $decod_{k,j+1}$ (por serlo $decod_{2,2}$)

EJEMPLO 16: Veamos como aplicar estas funciones a una tupla concreta. Supongamos que queremos calcular $\mathbf{cod^4(w_5,w_2,w_0,w_1)}$ y $\mathbf{decod_{4,3}(w_{178})}$. Siguiendo las definiciones:

$$\begin{aligned} cod^4(w_5,w_2,w_0,w_1) &= cod^2(w_5,cod^3(w_2,w_0,w_1)) = \\ &= cod^2(w_5,cod^2(w_2,cod^2(w_0,w_1))) = cod^2(w_5,cod^2(w_2, w_2)) = \\ &= cod^2(w_5,w_{12}) = w_{165} \end{aligned}$$

$$\begin{aligned} decod_{4,3}(w_{178}) &= decod_{3,2}(decod_{2,2}(w_{178})) = decod_{3,2}(w_7) = \\ &= decod_{2,1}(decod_{2,2}(w_7)) = decod_{2,1}(w_1) = w_1 \end{aligned}$$

4. Tipos de datos

Mediante el uso de macros hemos conseguido sortear las deficiencias que imponía la pobreza sintáctica de nuestro lenguaje: obligatoriedad de repetir código, restricciones en el uso de variables o carencia de un juego razonable de funciones y estructuras de control primitivas disponibles. Ello nos ha permitido comprobar que una buena parte de los elementos de control existentes en los lenguajes convencionales son simulables mediante programas while, y así los hemos incorporado como abreviaturas (macroprogramas) sin necesitar modificación alguna de nuestro lenguaje de partida. En el fondo lo que estamos haciendo no es muy distinto de lo que sucede en el mundo real. La UCP de un ordenador (que es quien en realidad "hace" las cosas) admite un juego de instrucciones relativamente pequeño, incluso diminuto en algunas arquitecturas, y luego se programa una interfaz que transmite al usuario o usuaria la sensación de que la máquina es mucho más compleja y cómoda. Pero en realidad todo lo que hacemos en términos de dicha máquina virtual se podría programar directamente sobre la UCP (y de hecho es traducido en ese sentido por la interfaz).

Hay sin embargo un aspecto complementario al control en el que no hemos avanzado nada: la draconiana limitación en los tipos de datos que pueden manejar nuestros programas while. Realmente resulta muy poco realista un lenguaje de programación que no pueda manejar objetos numéricos y lógicos, o agregaciones de objetos como vectores y ficheros. ¿Por qué nos hemos restringido al tipo de datos STRING? Fundamentalmente por dos razones:

- En primer lugar porque ello es lo que sucede en los computadores reales. De cara al usuario parece que existe una gran variedad de tipos de objetos, pero ello no es sino una ilusión. Externamente, porque cuando introducimos o extraemos datos de un ordenador lo hacemos a través de secuencias de caracteres sobre un alfabeto bien especificado (por ejemplo ASCII). Internamente, porque el computador convierte dicha información a su propio alfabeto (por ejemplo binario), y manipula las secuencias resultantes con operaciones propias de cadenas de caracteres.

- En segundo lugar, porque no nos vamos a quedar aquí. Como en el caso de las macros, vamos a ver que es posible simular la computación con **tipos de datos** que nuestro lenguaje no posee. Por ello la elección de STRING es tan válida como la de las funciones $cons_s$ y *cdr*. Comprobaremos que STRING es un buen trampolín sobre el que simular el resto de los tipos de datos y, por tanto el objetivo de minimalidad seguirá cumpliéndose.

La informática nos enseña que, en general, los mecanismos de abstracción y modularización funcional son bastante más sencillos que los correspondientes a nivel de datos. Por ello encontraremos que nuestro sistema de abreviaturas para simular tipos de datos tiene algunas diferencias técnicas con respecto a lo visto en el diseño de macros, pero sin embargo se basa exactamente en la misma filosofía:

- primero identificamos y definimos con precisión un cierto tipo de datos que los programas while no tienen, pero que nos gustaría que tuvieran
- después comprobamos que, aunque no forme parte del lenguaje, es siempre simulable con los elementos propios de los programas while
- a partir de ese momento empezamos a actuar *como si formara realmente parte* del lenguaje.

DEFINICIÓN 14: Un tipo de datos **T** es una estructura algebraica formada por un conjunto de **datos** o valores posibles del tipo (que de momento llamamos también **T** por extensión) y un conjunto de funciones $\{\psi_1, \psi_2, \ldots, \psi_n\}$ llamadas **operaciones** del tipo, siendo cada una de la forma ψ_i: $T_1 \times \ldots \times T_j \rightarrow T_{j+1}$, donde al menos uno de los tipos T_k es el propio **T**. Las operaciones que definen un tipo de datos tendrán en general argumentos y resultado de dicho tipo, pero también podrán ser funciones mixtas en las que algunos argumentos y/o el resultado pueden ser de otros tipos (como veremos que sucede en el caso de los predicados).

Por tanto, si nos interesa incorporar las características de un tipo de datos a nuestro lenguaje de programación, tenemos que resolver dos problemas: simular sus datos y simular sus operaciones. Por ejemplo, si se trata del tipo booleano, deberemos primeramente representar sus valores {true, false} mediante lo único que disponemos: palabras sobre un cierto alfabeto. Si el alfabeto es {**0**, **1**} podemos decidir que *false* sea la palabra '**0**' y *true* la palabra'**1**'. Si el alfabeto es {**a**, **b**, ..., **z**} podemos preferir representar *true* con la cadena '**cierto**' y *false* con '**falso**'. Entonces podremos trabajar con dichas palabras en las variables de los programas while *como si fueran* realmente los valores booleanos correspondientes, porque así las podemos *interpretar*. De hecho ya tuvimos un adelanto de esta idea cuando definimos predicados while-recursivos: ante la imposibilidad de usar valores booleanos decidimos arbitrariamente interpretar la palabra '$\mathbf{a_1}$' como *true* y la palabra vacía como *false*, lo cual nos ha permitido trabajar con operaciones lógicas para dirigir el control de las sentencias condicionales e iterativas sin necesidad de disponer de tipos lógicos.

Cuando hayamos encontrado la forma de simular los objetos tendremos que hacer lo propio con las operaciones que los manipulan, pero una vez dado el paso

anterior la solución es directa. Siguiendo con el ejemplo de los valores booleanos, si se trata de la operación disyunción, deberíamos comprobar que es posible realizarla dentro de nuestro lenguaje. Para ello es preciso comprobar que existe algún programa while que, cuando se le suministran dos palabras que representan valores booleanos (digamos '**falso**' y '**cierto**'), es capaz de devolver como resultado la palabra que representa su disyunción (en este caso '**cierto**').

Todo este proceso se denomina **implementación** del tipo de datos, y en la siguiente sección vamos a describirlo con más detalle y rigor.

4.1. Implementaciones

En general implementar un tipo de datos **T** significa decidir cómo representar cada dato del tipo que se desea implementar (**tipo abstracto**), en términos de los objetos del tipo que sirve de implementación (**tipo concreto**). En nuestro caso sólamente disponemos del tipo de datos STRING, y por tanto será éste el que utilicemos como tipo concreto para cualquier implementación (al menos de momento). La relación entre los datos de ambos tipos (palabras y objetos a implementar) se puede establecer de dos maneras:

- bien indicando un mecanismo de representación: dado un valor del nuevo tipo de datos, cuál será la palabra que lo va a representar
- bien indicando un mecanismo de interpretación: dada una palabra, qué valor del nuevo tipo de datos representa.

Vamos a concentrarnos en este segundo mecanismo, que es el que determina en lo sustancial la implementación.

DEFINICIÓN 15: Sea **T** un tipo de datos. Toda función de la forma:

$$\mathfrak{I}_{\mathbf{T}}: \Sigma^* \to \mathbf{T}$$

que sea total y sobreyectiva es llamada **función de interpretación** o función de abstracción. Esta función, como su propio nombre indica, debe establecer cómo se ha de interpretar cada palabra de Σ^* suponiendo que representa un elemento de **T**.

Conviene repasar los requisitos que $\mathfrak{I}_{\mathbf{T}}$ debe cumplir:

- Ser *sobreyectiva*, para que todo objeto del tipo de datos tenga al menos una palabra que lo represente. En caso contrario no estaríamos implementando el tipo **T**, sino en todo caso un subconjunto suyo.

- Ser *total* para que toda palabra de Σ^* pueda ser interpretada como un elemento de **T** en caso de necesidad. Esta exigencia es metodológica, ya que, aunque podríamos admitir como natural que no toda palabra represente necesariamente un dato, ello no sería conveniente por cuestiones de seguridad. No debemos olvidar que, aunque trabajaremos como si nuestros programas manipularan objetos del tipo **T**, en realidad lo único que estos procesarán son cadenas de símbolos que luego interpretaremos como tales objetos en virtud de $\Im_{\mathbf{T}}$. Si durante ese proceso se produjeran palabras que no tienen ningún significado en términos de **T** nos encontraríamos con situaciones de error, cosa que queremos evitar. En el caso de los booleanos el acuerdo de representar *true* con '**1**' y *false* con '**0**' no es válido si no lo completamos dotando de significado a todas las demás palabras de Σ^*. Por ejemplo, podemos acordar que todas las palabras que empiecen por **1** se interpreten como *true* y las demás (incluida ε) como *false*.
- La *inyectividad* no es necesaria, ya que para un mismo objeto podemos tener más de una representación. Así y todo la inyectividad (y por tanto biyectividad) opcional de $\Im_{\mathbf{T}}$ es una característica noticiable de la que estaremos pendientes.

De lo arriba indicado se deduce que las restricciones de cardinalidad son importantes a la hora de definir funciones de interpretación. Si el tipo **T** es finito es imposible que sea biyectiva, porque la cardinalidad de Σ^* es infinita. Si **T** es infinito numerable podemos elegir que lo sea o no. Y si **T** es no numerable la interpretación es sencillamente imposible. Alguna consecuencia de esto último es bien conocida en Informática: los números reales no pueden ser implementados en los lenguajes de programación. En su lugar se utilizan tipos como FLOAT, que sólo contienen aproximaciones de precisión fija, y cuyas propiedades son tan diferentes de las de los números reales que ha sido preciso crear toda una disciplina (el Análisis Numérico) para definirlas y estudiar sus consecuencias.

Al tiempo que se define la función que nos permite pasar de palabra a dato representado, a menudo resultará interesante contar también con el mecanismo inverso, llamado **función de representación**:

$$\Re_{\mathbf{T}}: \mathbf{T} \to \Sigma^*$$

que nos dice, dado un dato, una de las posibles palabras para representarlo (su **representación canónica**). En general suele resultar de ayuda el imaginar cómo queremos que sea $\Re_{\mathbf{T}}$ (que es una función "más natural") antes de diseñar $\Im_{\mathbf{T}}$.

Cuando la función de interpretación es biyectiva, se verifica simplemente que $\Re_{\mathbf{T}} = \Im_{\mathbf{T}}^{-1}$. En caso contrario, es preciso definir la función de representación de forma que $\Im_{\mathbf{T}}(\Re_{\mathbf{T}}(x)) = x$ para todo elemento x de **T**. Es obvio que la función de interpretación no es única, pero una vez la hayamos definido, todas las operaciones del tipo que estamos implementando deberán simularse ajustándose a ella.

El mecanismo de simulación de las operaciones es tal como se describió justo antes de la presente sección: si vamos a trabajar con la "ilusión óptica" de que ciertas palabras representan otra cosa diferente de lo que son, deberemos diseñar programas que obren en consecuencia y no rompan la ilusión: habrán de manipular dichas palabras de forma que los resultados sigan pareciendo eso que queremos ver tras ellas.

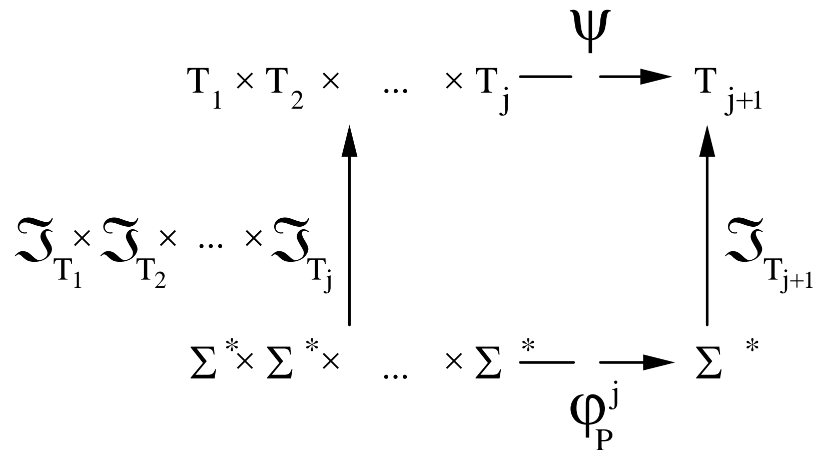
DEFINICIÓN 16: Sea $\psi: T_1 \times \ldots \times T_j \rightarrow T_{j+1}$ una operación sobre el tipo **T** (es decir, que al menos uno de los T_k es precisamente **T**). Sean $\Im_{\mathbf{T}_1}, \Im_{\mathbf{T}_2}, \ldots, \Im_{\mathbf{T}_{j+1}}$ funciones de interpretación para los tipos de datos que forman parte de la definición de ψ. Se dice que el programa while **P implementa** a ψ si verifica, para cualesquiera palabras $z_1, z_2, \ldots, z_j$:

a) si $\psi(\Im_{\mathbf{T}_1}(z_1), \Im_{\mathbf{T}_2}(z_2), \ldots, \Im_{\mathbf{T}_j}(z_j))\uparrow$ entonces $\varphi_{\mathbf{P}}^{j}(z_1, z_2, \ldots, z_j)\uparrow$

b) si $\psi(\Im_{\mathbf{T}_1}(z_1), \Im_{\mathbf{T}_2}(z_2), \ldots, \Im_{\mathbf{T}_j}(z_j))\downarrow$ entonces $\varphi_{\mathbf{P}}^{j}(z_1, z_2, \ldots, z_j)\downarrow$ y además

$$\Im_{\mathbf{T}_{j+1}}(\varphi_{\mathbf{P}}^{j}(z_1, z_2, \ldots, z_j)) \simeq \psi(\Im_{\mathbf{T}_1}(z_1), \Im_{\mathbf{T}_2}(z_2), \ldots, \Im_{\mathbf{T}_j}(z_j))$$

Es decir, el programa while **P** toma j palabras $z_1, z_2, \ldots, z_j$ como argumentos y devuelve otra palabra w como resultado, pero de forma que el valor representado por w sea exactamente el que se obtiene al aplicar ψ sobre los datos representados por $z_1, z_2, \ldots, z_j$. De forma análoga **P** diverge sobre las palabras que representan valores para los cuales ψ está indefinida. Se dice entonces que las interpretaciones son homomorfas con ψ, dado que se respeta el siguiente esquema de conmutatividad:



DEFINICIÓN 17: Sea (**T**, $\{\psi_1, \psi_2, \dots, \psi_n\}$) un tipo de datos. Una implementación de **T** es un par ($\mathfrak{I}_\mathbf{T}$, $\{P_1, P_2, ..., P_n\}$), donde $\mathfrak{I}_\mathbf{T}$ es una función de interpretación para los datos de **T** y cada programa P_i implementa la función ψ_i. Se dice que la implementación es **estricta** si la función $\mathfrak{I}_\mathbf{T}$ es biyectiva.

Para implementar un tipo de datos (**T**, $\{\psi_1, \psi_2, \dots, \psi_n\}$) hemos de dar por tanto los siguientes pasos:

- Encontrar una función de interpretación $\mathfrak{I}_\mathbf{T}$: $\Sigma^* \to \mathbf{T}$. En caso de que no sea biyectiva, es necesario definir también la correspondiente función de representación $\mathfrak{R}_\mathbf{T}$: $\mathbf{T} \to \Sigma^*$.
- Hacer una lista de los tipos de datos (aparte del propio **T** y de STRING) que intervienen en la definición de las funciones $\psi_1, \psi_2, \dots, \psi_n$. Es preciso que esos tipos estén previamente implementados antes de poder hacerlo con **T**.
- Construir los programas $\{P_1, P_2, ..., P_n\}$ que implementan las operaciones del tipo. El encabezamiento de dichos programas (en general común a todos) será de la forma:

  package COMPUTABLES **is**
  CT_1 CT_2 ... CT_m
  CF_1 CF_2 ... CF_p
  end COMPUTABLES;

  donde CT_1, CT_2, ..., CT_m son cabeceras de tipos que hacen referencia a los tipos de datos mencionados en el punto anterior, y cuya implementación previa es necesaria, y CF_1, CF_2, ..., CF_p son cabeceras de funciones tal como se describieron en el capítulo anterior. Cada una de esas cabeceras tendrá la forma:

  type T_i **is** STRING;

indicando así que los elementos del tipo se simulan directamente con palabras de Σ^*.

4.2. Implementación de tipos simples

A continuación veremos dos ejemplos de implementaciones de dos tipos simples que nos permitirán hacer computaciones lógicas y numéricas. En general procuraremos diseñar las funciones de interpretación de la forma más sencilla posible, pero siempre procurando que tengan dos características:

- que sean biyectivas: la elección de implementaciones estrictas se debe, por un lado, a que nos evita tener que definir la función de representación, y por otro, a que en general las operaciones resultan más sencillas de implementar cuando a cada valor del tipo implementado le corresponde una única representación
- que sean independientes del alfabeto: las implementaciones que se suelen utilizar en informática están pensadas para un alfabeto binario y para una manera determinada de organizar los datos; por nuestra parte preferiremos que sean aplicables a cualquier situación posible, aunque a veces las soluciones encontradas puedan parecer "antinaturales".

Por otro lado procuraremos elegir con cuidado las operaciones de cada tipo, procurando siempre tomar un conjunto lo más pequeño posible de las mismas. Ello se debe a que, como veremos más adelante, una vez implementado, el tipo podrá ser utilizado sin restricciones para acrecentar su juego de operaciones. Así, buscaremos definir inicialmente sólo las esenciales para posteriormente enriquecer el tipo a voluntad.

4.2.1. El tipo BOOLEAN

Este tipo de datos, concebido para implementar operaciones lógicas, está constituido por el conjunto **B** = {true, false} y por las operaciones de negación e igualdad (equivalencia lógica).

$$\neg: \mathbf{B} \to \mathbf{B}$$

$$<\!-\!>: \mathbf{B} \times \mathbf{B} \to \mathbf{B}$$

La función de interpretación que vamos a utilizar estará basada en nuestra práctica para computar funciones características: representaremos la constante *false* con la palabra vacía y la constante *true* con la palabra $\mathbf{a_1}$, donde $\mathbf{a_1} \in \Sigma$ es el

primer elemento del alfabeto. Cualquier otra palabra será interpretada también como *true*. La ventaja que tiene esta representación (muy diferente a la que se usa en general en la práctica informática, salvo quizá en el lenguaje de programación LISP) es que es independiente del alfabeto que se utilice.

$$\Im_{\mathbf{B}}(w) = \begin{cases} \text{false} & \text{si } w = \varepsilon \\ \text{true} & \text{c.c.} \end{cases}$$

Lógicamente no se trata de una interpretación biyectiva, por lo que tenemos que definir la función de representación:

$$\Re_{\mathbf{B}}(t) = \begin{cases} \varepsilon & \text{si } t = \text{false} \\ \mathbf{a_1} & \text{si } t = \text{true} \end{cases}$$

Ahora implementaremos en primer lugar la función unaria negación, que devuelve el valor contrario al de su argumento:

```
package COMPUTABLES is
      function "=" (STRING, STRING) return STRING ;
      function "" (STRING) return STRING;
      function const_ε (STRING) return STRING;
end COMPUTABLES;
```

```
X0 := ε;
if X1 = ε then X0 := cons_a1(X0); end if;
```

Y para terminar la igualdad. Dado que la interpretación no es biyectiva, la igualdad de valores no se refleja en la igualdad de palabras, ya que la primera se cumple siempre que sus dos argumentos *representen* el mismo valor. Esta es una característica típica de las implementaciones no estrictas, en las que la igualdad debe ser programada. Esta es también la razón por la que no podemos expresar la igualdad entre booleanos con el símbolo habitual "=", ya que no debe ser confundida con la igualdad entre palabras cuando la usemos en nuestros programas.

```
package COMPUTABLES is
      function "=" (STRING, STRING) return STRING ;
      function "/=" (STRING, STRING) return STRING ;
      function proy_2_1 (STRING, STRING) return STRING ;
      function proy_2_2 (STRING, STRING) return STRING ;
- - Porque el predicado de la macrocondición tiene dos argumentos
      function const_ε (STRING, STRING) return STRING;
end COMPUTABLES;
```

```
X0 := ε;
if (X1 = ε and X2 = ε) or (X1 /= ε and X2 /= ε) then X0 := cons_a1(X0); end if;
```

Una vez que tenemos implementados los booleanos podremos utilizarlos en la construcción de predicados, que definiremos a partir de este momento como funciones con resultado BOOLEAN. Lo mismo podemos decir de las funciones características de los conjuntos. De hecho reconvertiremos todos los predicados y funciones características definidas hasta ahora como funciones con resultado BOOLEAN.

4.2.2. El tipo NATURAL

Este tipo de datos está formado por el conjunto **N** = {0, 1, 2, 3, ... } de los números naturales y las operaciones sucesor (unaria) e igualdad (predicado binario):

$$\text{suc}: \mathbf{N} \rightarrow \mathbf{N}$$

$$=: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$$

Para definir la función de interpretación haremos corresponder el orden de los números naturales con el que definimos para las palabras sobre cualquier alfabeto en el apartado **3.4.2**, con lo que la palabra vacía representará al 0, y si *w* representa al número *k*, entonces *sig(w)* representará a *suc(k)*. En definitiva se cumplirá que $\Im_{\mathbf{N}}(w_k) = k$.

Si suponemos que $\Sigma = \{ \mathbf{a_1}, \mathbf{a_2}, ..., \mathbf{a_n} \}$, la función $\Im_{\mathbf{N}}: \Sigma^* \rightarrow \mathbf{N}$ se puede definir inductivamente como sigue:

$$\Im_{\mathbf{N}}(\varepsilon) = 0$$

$$\Im_{\mathbf{N}}(x \bullet \mathbf{a_i}) = n * \Im_{\mathbf{N}}(x) + i$$

Tal como hemos definido la interpretación, la función *suc* es idéntica a la función *sig*. Por tanto sería computada por un programa tan simple como el siguiente:

```
package COMPUTABLES is
    function sig (STRING) return STRING;
end COMPUTABLES;

X0 := sig(X1);
```

Algo parecido sucede con la igualdad. Al tratarse de una implementación estricta dos números naturales son iguales si y sólo si son representados por la misma palabra, por lo que el programa que computa el predicado igualdad entre números naturales quedaría reducido a:

```
package COMPUTABLES is
```

```
        type BOOLEAN is STRING;
        function "=" (STRING, STRING) return BOOLEAN;
    end COMPUTABLES;

    X0 := X1 = X2;
```

En este caso no hay inconveniente en expresar la igualdad entre naturales con el mismo símbolo que la igualdad entre palabras. Nótese que se ha utilizado por primera vez el predicado while-recursivo de igualdad entre palabras como función de resultado booleano (en una macroexpresión en lugar de en una macrocondición).

En general, cuando se den casos como los anteriores, en los que se pretende definir una nueva función y resulta equivalente a otra que ya ha sido programada previamente, nos limitaremos a citarlo y no nos molestaremos en escribir el programa. En particular nunca será necesario programar la igualdad de los tipos estrictamente implementados.

4.3. Uso de las implementaciones

Una vez construida la implementación de un tipo de datos **T** podremos cumplir nuestro objetivo de utilizarlo dentro de nuestros macroprogramas, aunque conviene detallar las modalidades y la justificación de este uso.

En primer lugar, podremos utilizar las constantes del tipo en las macro-expresiones y macrocondiciones. . El primer paso de la expansión de cualquier macro que contenga apariciones de una constante **t** $\in$ **T** consiste en sustituir dichas apariciones por '*w*', siendo $w = \Re_{\mathbf{T}}(\mathbf{t})$ (la representación canónica de **t**). Ahora bien, igual que hacíamos con las constantes STRING habremos de declarar la correspondiente función constante en el paquete COMPUTABLES.

La expansión es una vuelta a la dura realidad: nos habíamos creado la ilusión de que manejábamos objetos del tipo **T** y resulta que no son más que simples palabras "disfrazadas" por nuestra interpretación. De todas formas ello no es de preocupar: lo que nos interesa del tipo **T** no son sus elementos y operaciones, sino *las propiedades de dichos elementos y de sus operaciones*. Si, por ejemplo, queremos hacer computación numérica, no es porque nos apetezca especialmente trabajar con números, sino porque nos interesa poder contar, sumar, multiplicar etc..., y todo eso lo podremos hacer con nuestros programas while.

En segundo lugar, podremos utilizar las operaciones del tipo **T** en las macroexpresiones y macrocondiciones. Si tenemos un macroprograma con

apariciones de la operación ψ, el primer paso de su expansión consistirá en sustituir cada una de esas apariciones por φ_P^j (donde P es el programa que implementa a ψ), que es la correspondiente función entre palabras.

Existe un tercer posible uso de los elementos del tipo **T** en macroprogramas, una vez que hemos introducido el tipo BOOLEAN: sus predicados (es decir, aquellas de sus operaciones que además de ser totales producen resultado del tipo BOOLEAN) pueden ser utilizados en las macrocondiciones. La expansión sería idéntica a la del caso anterior.

Por último, existen una serie de funciones while-computables entre cadenas que pueden ser heredadas sin dificultad por cualquier tipo de datos: identidad, proyecciones, función vacía, constantes, o funciones de codificación. Existe un pequeño problema con estas funciones, pues la posibilidad de usarlas extensivamente con otros tipos puede engordar mucho los encabezamientos de nuestros macroprogramas. Por ejemplo, podemos usar en el mismo programa tres funciones identidad: una para cadenas, otra para booleanos y otra para naturales. Para evitar confusiones tomaremos el convenio de declarar siempre estas funciones únicamente dentro del tipo STRING, aunque luego puedan ser utilizadas por otros tipos en el programa.

Naturalmente, si empezamos a utilizar un tipo de datos ya implementado para construir programas, una de las consecuencias más inmediatas es que probablemente definamos nuevas funciones asociadas a dicho tipo. Solemos decir que dichas funciones creadas con posterioridad a la implementación constituyen **enriquecimientos** del tipo. Llegados a este punto conviene que ampliemos un poco nuestra noción de while-computabilidad.

DEFINICIÓN 18: Sea Ψ: $T_1 \times \ldots \times T_j \rightarrow T_{j+1}$ una función cualquiera que cumple que todos los tipos T_1, T_2, … , T_{j+1} están implementados. Decimos que Ψ es while-computable si existe un programa while P que la implementa. Si Ψ es total y T_{j+1} = **B**, entonces se trata de un predicado y podemos decir alternativamente que es while-recursivo. Cuando demostramos que Ψ es while-computable deben aparecer en su cabecera todos los tipos que intervienen en ella salvo STRING.

Hay una diferencia menor y otra sustancial entre las operaciones que se introducen en el momento de crear el tipo y los enriquecimientos. Cuando construimos las primeras el tipo aún no está implementado, por lo que *es necesario trabajar directamente sobre la representación*, con operaciones propias de palabras. Sin embargo, una vez implementado el tipo de datos disponemos de un juego de operaciones que permiten tratar a las variables como si en efecto contuvieran valores del mismo. Por ello en los enriquecimientos *evitaremos usar nada que*

dependa de la forma concreta como se ha diseñado la implementación: únicamente emplearemos las constantes y operaciones del tipo. Por ejemplo, cuando utilicemos una variable cuyo contenido va a ser manipulado como booleano, nunca utilizaremos el predicado *nonem?* sobre la misma, sino que en todo caso preguntaremos si su contenido es igual a *true*. La idea que subyace es la de la **no visibilidad** de la implementación, tanto de la representación de los datos como de la implementación de sus operaciones.

EJEMPLO 17: Vamos a comprobar que las siguientes funciones y predicados son while-computables, y por tanto son enriquecimientos de los tipos BOOLEAN y NATURAL.

a) Las funciones conjunción y disyunción, que se aplican sobre el tipo BOOLEAN con el significado usual:

$$\wedge: \mathbf{B} \times \mathbf{B} \rightarrow \mathbf{B}$$

$$\vee: \mathbf{B} \times \mathbf{B} \rightarrow \mathbf{B}$$

Podemos computar la conjunción basándonos en las operaciones con las que se construyó el tipo:

```
package COMPUTABLES is
    type BOOLEAN is STRING;
    function "<->" (BOOLEAN, BOOLEAN) return BOOLEAN;
    function proy_2_1 (BOOLEAN, BOOLEAN) return BOOLEAN;
    function proy_2_2 (BOOLEAN, BOOLEAN) return BOOLEAN;
    function const_true (BOOLEAN, BOOLEAN) return BOOLEAN;
    function const_false (BOOLEAN) return BOOLEAN;
end COMPUTABLES;
```

```
if (X1 <-> true and X2 <-> true) then X0 := true; else X0 := false; end if;
```

y la disyunción a partir de la anterior:

```
package COMPUTABLES is
    type BOOLEAN is STRING;
    function "¬" (BOOLEAN) return BOOLEAN;
    function "∧" (BOOLEAN, BOOLEAN) return BOOLEAN;
end COMPUTABLES;
```

```
X0 := ¬(¬X1 ∧ ¬X2);
```

b) La función predecesor y los distintos predicados de comparación entre números naturales:

$$\text{pred}: \mathbf{N} \rightarrow \mathbf{N}$$

$$<: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$$

$$<=: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$$

$$>: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$$

$$>=: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$$

$$/=: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$$

No necesitamos hacer programas para ninguna de estas operaciones, ya que todas ellas corresponden a operaciones ya definidas entre cadenas. En el caso de *pred* la función es *ant*, y en los demás se trata de los predicados del mismo nombre entre palabras

b) Las funciones aritméticas de suma, diferencia y producto:

$$+: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$$

$$-: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$$

$$*: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$$

Es fácil realizar el programa para la suma a partir de aplicaciones sucesivas de la función *suc*:

```
package COMPUTABLES is
    type NATURAL is STRING;
    type BOOLEAN is STRING;
    function "" (STRING) return STRING;
    function "/=" (NATURAL, NATURAL) return BOOLEAN;
    function const_0 (STRING) returns NATURAL;
    function suc (NATURAL) return NATURAL;
    function pred (NATURAL) return NATURAL;
end COMPUTABLES;
```

```
X0 := X1;
AUX := X2;
- - X0+AUX=X1+X2
while AUX /= 0 loop
    X0 := suc(X0);
    AUX := pred(AUX);
end loop;
```

Para la diferencia utilizaríamos el mismo encabezamiento que en el programa anterior. Hay que tener en cuenta que la diferencia entre números naturales es problemática, pues no se pueden expresar resultados negativos. En nuestro caso la definiremos de modo que si el minuendo es menor que el sustraendo, la resta se considera 0 (en consonancia con la definición de la función *pred* y siguiendo en nuestra línea de evitar los casos de error):

```
X0 := X1;
```

```
    AUX := X2;
- - X0–AUX=X1–X2
    while AUX /= 0 loop
        X0 := pred(X0);
        AUX := pred(AUX);
    end loop;
```

Para el producto haremos un nuevo encabezamiento, ya que nos interesa utilizar la recién definida suma en una de sus macros:

```
package COMPUTABLES is
    type NATURAL is STRING;
    type BOOLEAN is STRING;
    function const_0 (STRING) returns NATURAL;
    function "" (STRING) return STRING;
    function "/=" (NATURAL, NATURAL) return BOOLEAN;
    function "+" (NATURAL, NATURAL) return NATURAL;
    function pred (NATURAL) return NATURAL;
end COMPUTABLES;
```

```
    X0 := 0;
    AUX := X2;
- - X0+X1*AUX=X1*X2
    while AUX /= 0 loop
        X0 := X0 + X1;
        AUX := pred(AUX);
    end loop;
```

También podemos demostrar la while-computabilidad de funciones en cuyos argumentos intervienen naturales y palabras. En general estas funciones son enriquecimientos mixtos del tipo STRING.

Por ejemplo, es fácil ver que la función *longitud*, que dada una palabra nos devuelve su número de símbolos, es while-computable. Su definición inductiva sería:

- $|\varepsilon| = 0$
- $|x \bullet \mathbf{s}| = suc(|x|)$

Y el programa que la implementa podría ser el siguiente

```
package COMPUTABLES is
    type NATURAL is STRING;
    function suc (NATURAL) return NATURAL;
end COMPUTABLES;
```

```
X0 := 0;
AUX := X1;
```

```
while nonem?(AUX) loop
    X0 := suc(X0);
    AUX := cdr(AUX);
end loop;
```

Sabemos que toda palabra puede ser considerada como el código de una *k*-tupla. Conociendo el número *k* se pueden computar cada una de las distintas funciones de decodificación $decod_{k,j}$ que extraen el *j*-ésimo elemento de dicha *k*-tupla. Pero también podemos hacer algo más general: en lugar de definir una batería de funciones, podemos limitarnos a una sola que los englobe:

$$\text{decod: } \mathbf{N} \times \mathbf{N} \times \Sigma^{*} \rightarrow \Sigma^{*}$$

de forma que decod(k, j, w) = $decod_{k,j}(w)$ para toda palabra *w*, estando indefinida cuando $k=0$, $j=0$ ó $j>k$.

La ventaja de esta última es que no necesitamos saber, en el momento de escribir el programa, ni cuántas palabras codifica *w* ni cuál de ellas nos va a interesar, por lo que estos datos podrán determinarse en tiempo de ejecución. Otro detalle importante es que la computabilidad de *decod* no se deduce a partir de la while-computabilidad de todas las funciones $decod_{k,j}$ (de hecho veremos que en el programa apenas se usan estas últimas). Es un ejemplo de generalización tal como lo mencionábamos al principio del capítulo.

```
package COMPUTABLES is
    type NATURAL is STRING;
    function "=" (NATURAL, NATURAL) return BOOLEAN;
    function proy_2_1 (STRING, STRING) return STRING;
    function proy_2_2 (STRING, STRING) return STRING;
    function const_0 (STRING, STRING) return NATURAL;
    function "<" (NATURAL, NATURAL) return BOOLEAN;
    function ⫫ (STRING) return STRING;
    function pred (NATURAL) return NATURAL;
    function "" (STRING) return STRING;
    function decod_2_1 (STRING) return STRING;
    function decod_2_2 (STRING) return STRING;
end COMPUTABLES;
```

Se observará que hemos aplicado el convenio de declarar las funciones de aplicación universal (proyecciones, constantes, vacía e identidad) para el tipo STRING de forma sistemática, sin entrar en consideración de si en realidad se aplican sobre palabras que han de ser interpretadas como elementos de otro tipo.

```
if (X1 = 0) or (X2 = 0) or (X1 < X2) then
    X0 := ⫫;
else AUX := X3;
```

```
        if X1 = X2 then
                MAX := pred(X2);
        else    MAX := X2;
        end if;
        for IND in 1 .. MAX loop
                X0 := decod_2_1(AUX);
                AUX := decod_2_2(AUX);
        end loop;
        if X1 = X2 then
                X0 := AUX;
        end if;
    end if;
```

4.4. Implementación de tipos compuestos

Vamos a ver ahora que podemos utilizar los mecanismos descritos para implementar **tipos compuestos**, es decir, tipos que se forman por agregación de valores de otro tipo (llamado **tipo base**). Los dos ejemplos que vamos a ver corresponden a estructuras que luego revelan su utilidad en la construcción de algoritmos fundamentales en el desarrollo de la Teoría de la Computabilidad: pilas y vectores dinámicos.

En ambos casos usaremos como tipo base STRING, pero no hay ninguna dificultad en redefinir estos tipos compuestos si queremos, por ejemplo, construir vectores de pilas de números naturales. Lo único que hay que hacer es declarar las operaciones sobre los tipos correspondientes e interpretar los elementos del tipo base de la forma adecuada.

4.4.1. El tipo PILA

Las pilas son dispositivos de almacenamiento de datos. El tipo de datos PILA está constituido por el conjunto **P** de las pilas de palabras. Cada valor del tipo se forma por agregación de cero o más cadenas de caracteres (llamadas **componentes** de la pila) que se almacenan ordenadas en él. Representamos un objeto del tipo pila listando sus componentes entre los símbolos "<" y "]". Por ejemplo < **aba**, **bb**, ε, **a**] es una pila con cuatro componentes, y <] es una pila con cero componentes, denominada **pila vacía**. Al componente más cercano al símbolo "<" de una pila no vacía se le llama **cima** de la pila. Las operaciones básicas del tipo PILA son:

$$\text{empilar: } \Sigma^* \times \mathbf{P} \rightarrow \mathbf{P}$$

$$\text{es_p_vacía?}: \mathbf{P} \to \mathbf{B}$$

$$\text{cima}: \mathbf{P} \to \Sigma^*$$

$$\text{desempilar}: \mathbf{P} \to \mathbf{P}$$

Para definir la función de interpretación hemos de representar cada pila mediante una palabra, que debe codificar a todos sus componentes respetando el orden en que se han introducido. Naturalmente ha de ser posible recuperar esta información, por lo que parece razonable utilizar las funciones de código definidas en la sección **3.6**. Podría parecer que la solución más directa es codificar la pila $< x_1, x_2, \ldots, x_k]$ con la palabra $cod^k(x_1, x_2, \ldots, x_k)$, pero no sería una buena idea, porque ello no nos aseguraría una función de interpretación:

- En primer lugar la pila vacía $<]$ no tiene componentes que codificar, y sin embargo debe ser representada. Necesitamos reservar un código para ella.
- En segundo lugar, suponiendo que, por ejemplo, el alfabeto es {**a**, **b**, **c**} ¿cómo se debe interpretar la palabra **acba**, teniendo en cuenta que **acba** = cod^1(**acba**) = cod^2(**aa**, **ac**) = cod^3(**aa**, **c**, ε) = cod^4(**aa**, **c**, ε, ε) = ...? Necesitamos alguna forma de fijar cuántos elementos tiene la pila a partir de su código.

Una forma de resolver este segundo problema consiste en imaginar que podemos "sacar" elementos de una pila hasta encontrar ε, que actuaría como una marca de "fondo de pila". Tenemos la ventaja de que a fuerza de aplicar $decod_{2,2}$ se tiene que producir necesariamente ε (no es difícil probar que $decod_{2,2}(x) < x$). Con este sistema también resolvemos el primer problema: ε representaría la pila de la que no se puede sacar nada. Pero ahora tenemos una dificultad distinta, porque no podemos representar pilas que tengan la palabra vacía en el fondo (esa palabra sería confundida con la señal de que la pila ha quedado vaciada).

La solución definitiva pasa por un pequeño truco: codificar todos los elementos de la pila normalmente *salvo el del fondo*, porque en su lugar codificaremos la palabra siguiente. Es decir, pretendemos que $\Re_{\mathbf{P}}(< x_1, x_2, \ldots, x_k]) = cod^{k+1}(x_1, x_2, \ldots, sig(x_k), \varepsilon)$. De este modo, al interpretar una palabra e ir extrayendo las cadenas que codifica en su seno, tendremos siempre la seguridad de que un residuo ε indica el final de la pila.

Ahora bien, lo importante es la función de interpretación, pues sin ella no tenemos implementación. Siguiendo las ideas que hemos esbozado, su definición podría ser la siguiente:

$$\Im_{\mathbf{P}}(w) = \begin{cases} <\,] & \text{si } w = \varepsilon \\ <\text{ant}(\text{decod}_{2,1}(w))\,] & \text{si } w \neq \varepsilon \wedge \text{decod}_{2,2}(w) = \varepsilon \\ \text{empilar}\,(\text{decod}_{2,1}(w), \Upsilon_{\mathbf{P}}(\text{decod}_{2,2}(w))) & \text{c.c.} \end{cases}$$

La función es total por lo que hemos comentado antes: al cumplirse siempre que $\text{decod}_{2,2}(x) < x$ tenemos garantía de que la definición recursiva converge para cualquier w. Tampoco es difícil ver que es biyectiva. Vamos ahora a definir los programas que implementan sus operaciones.

En primer lugar tenemos la función *empilar*, que añade una palabra a una pila colocándola en su cima:

```
package COMPUTABLES is
    function cod_2 (STRING, STRING) return STRING;
    function sig  (STRING) return STRING;
    function const_ε ( STRING) return STRING;
end COMPUTABLES;
```

```
if nonem?(X2) then
    X0 := cod_2(X1, X2);
else  X0 := cod_2(sig(X1), ε);
end if;
```

El predicado *es_p_vacía?* indica si su argumento es la pila vacía o no:

```
package COMPUTABLES is
    type BOOLEAN is STRING;
    function const_false ( STRING) return BOOLEAN;
    function const_true ( STRING) return BOOLEAN;
end COMPUTABLES;
```

```
if nonem?(X1) then
    X0 := false;
else  X0 := true;
end if;
```

También hemos de implementar la función *cima*, que aplicada a una pila no vacía proporciona la palabra situada en su cima. Por el contrario, la cima de una pila vacía está indefinida.

```
package COMPUTABLES is
    function decod_2_2 (STRING) return STRING;
    function decod_2_1 (STRING) return STRING;
    function ant (STRING) return STRING;
    function ⊥⊥ (STRING) return STRING;
end COMPUTABLES;
```

```
if  nonem?(X1) then
        if nonem?(decod_2_2 (X1)) then
                X0 := decod_2_1 (X1);
        else   X0 := ant(decod_2_1 (X1));
        end if;
else  X0 := ⊥⊥ ;
end if;
```

Por último implementaremos la función *desempilar*, que elimina la cima de la pila que se le suministra como argumento, devolviendo la pila resultante. En caso de que la pila original sea la vacía, la función está indefinida. Para esta función podemos aprovechar el mismo encabezamiento que para la anterior, quedándonos el programa:

```
if nonem?(X2) then
        X0 := decod_2_2 (X1);
else  X0 := ⊥⊥ ;
end if;
```

4.4.2. El tipo VECTOR

Los vectores dinámicos también son estructuras de almacenamiento ordenado de datos que pueden variar de tamaño en tiempo de ejecución. La diferencia con las pilas es que cualquier componente de un vector es accesible en cualquier momento mediante su indice. Los índices cuentan siempre a partir de 0, y todo vector tiene al menos una componente. Denotamos por **V** el conjunto de valores del tipo de datos.

Representamos los vectores como listas de sus componentes entre paréntesis. Así, en el vector (**aa**, **b**, **aca**,) la 0-ésima componente es **aa**, y la segunda es **aca**. Un vector se crea de un determinado tamaño, y puede ser modificado localmente en cualquiera de sus componentes. En el momento en que se modifica localmente una componente inexistente porque el índice es demasiado grande, el vector aumenta de tamaño para dar cabida al nuevo valor, rellenandose los índices intermedios con la palabra vacía. En el ejemplo anterior, si se modifica el vector en séptima componente automáticamente pasa a tener ocho componentes, pasando a ser ε el contenido de la tercera a la sexta. Sin embargo, no está permitida la operación de acceso a componentes inexistentes.

Las operaciones que se definen para este tipo de datos son:

$$\text{ult_índice}: \mathbf{V} \rightarrow \mathbf{N}$$

$$\text{acceso}: \mathbf{V} \times \mathbf{N} \rightarrow \Sigma^*$$

$$\text{modifica}: \mathbf{V} \times \mathbf{N} \times \Sigma^* \rightarrow \mathbf{V}$$

Para definir la función de interpretación volvemos a tener el mismo problema que con las pilas: la solución de representar el vector $(x_0, x_2, ..., x_n)$ con la palabra $cod^{k+1}(x_0, x_1, ..., x_k)$ no es adecuada porque los componentes no son recuperables. Pero en este caso vamos a resolver la situación almacenando directamente el tamaño del vector al codificarlo, es decir que el vector arriba citado se representaría como $cod^{k+2}(k, x_0, x_1, x_2, \ldots, x_k)$. Por supuesto ,se entiende que el primer argumento de la función es la palabra que representa al número *k*. En definitiva, la función de interpretación que usaremos es la siguiente:

$$\Im_{\mathbf{V}}(w) = (x_0, x_1, x_2, \ldots, x_k) \Leftrightarrow \begin{cases} k = \Upsilon_{\mathbf{N}}(decod_{2,1}(w)) \\ \square \\ \forall i\ (0 \le i \le k \varnothing z_i = decod_{k+1,i+1}(decod_{2,2}(w))) \end{cases}$$

De las operaciones que tenemos que implementar, la función *ult_índice*, que nos proporciona el índice o posición correspondiente al último elemento del vector es simplemente $decod_{2,1}$, por lo que no ha de ser programada. La función *acceso*, dado un vector y un número (posición a la que queremos acceder), nos proporciona la componente del vector que se encuentra en dicha posición. Como se ha indicado antes, esta función está indefinida cuando la posición a la que se quiere acceder es superior a la última del vector.

```
package COMPUTABLES is
    type NATURAL is STRING;
    function "<=" (NATURAL, NATURAL) return BOOLEAN;
    function decod (NATURAL, NATURAL, STRING) return STRING;
    function suc (NATURAL) return NATURAL;
    function decod_2_2 (STRING) return STRING;
    function proy_2_1 (STRING, STRING) return STRING;
    function proy_2_2 (STRING, STRING) return STRING;
    function decod_2_1 (STRING) return STRING;
    function ⊥⊥ (STRING) return STRING;
end COMPUTABLES;
```

El programa distingue el caso de indefinición, y en los demás extrae el componente solicitado utilizando la función *decod*:

```
if X2 <= decod_2_1(X1) then
    X0 := decod(suc(decod_2_1(X1)), suc(X2), decod_2_2(X1));
else X0 := ⊥⊥ ;
end if;
```

Por último tenemos la función *modifica*, que toma un vector, una posición y una palabra de entrada, y devuelve el vector resultante tras incluir la palabra en la posición indicada del vector de entrada. La función *modifica* sustituye, si es preciso, la componente que estaba en dicha posición. Por el contrario, si la posición modificada es superior a la última del vector que sirve de argumento, tal posición y las que le preceden son automáticamente creadas, quedando estas últimas dotadas del valor ε. Esta es sin duda la función más complicada de implementar, puesto que para modificar cualquier componente hay que literalmente destripar el vector y rehacer su estructura por completo sin poder aprovechar nada de su organización anterior.

```
package COMPUTABLES is
	type PILA is STRING;
	type NATURAL is STRING;
	type BOOLEAN is STRING;
	function decod_2_1 (STRING) return STRING;
	function decod_2_2 (STRING) return STRING;
	function const_< ] (STRING) return PILA;
	function const_0 (STRING) return NATURAL;
	function pred (NATURAL) return NATURAL;
	function empilar (STRING, PILA) return PILA;
	function proy_2_1 (STRING, STRING) return STRING;
	function proy_2_2 (STRING, STRING) return STRING;
	function "<" (NATURAL, NATURAL) return BOOLEAN;
	function cima (PILA) return STRING;
	function suc (NATURAL) return NATURAL;
	function desempilar (PILA) return PILA;
	function cod_2 (STRING, STRING) return STRING;
	function "" (STRING) return STRING;
	function "=" (NATURAL, NATURAL) return BOOLEAN;
	function es_p_vacía? (PILA) return BOOLEAN;
end COMPUTABLES;
```

La función *modifica* distingue tres casos: que la posición sea anterior a la última, posterior a la última, o que sea exactamente la última. En todos los caso es necesario "deshacer" el vector y guardar sus elementos en una pila.

```
- - La variable ULT_IND contendrá el último índice del vector
    ULT_IND := decod_2_1(X1);
- - En primer lugar volcaremos las componentes del vector en la pila PILAUX
    AUX := decod_2_2(X1);
    PILAUX := < ];
- - En cada iteración se incorpora el i-ésimo componente del vector en la pila
    for IND in 0 .. pred(ULT_IND) loop
        PILAUX := empilar(decod_2_1(AUX), PILAUX);
```

```
            AUX := decod_2_2(AUX);
        end loop;
    - - En la cima queda el último componente
        PILAUX := empilar(AUX, PILAUX);
        if X2 < decod_2_1(X1) then
    - - Caso 1: se pretende modificar una posición existente en el vector diferente de
        la última
            X0 := cima(PILAUX);
    - - Se cargan en la solución las componentes de índice posterior al de la
        modificada
            for IND in reverse pred(ULT_IND) .. suc(X2) loop
                PILAUX := desempilar(PILAUX);
                X0 := cod_2(cima(PILAUX), X0);
            end loop;
    - - Se carga el nuevo valor y se retira el viejo de la pila (junto con otro
    - - que queda por retirar del bucle anterior)
            X0 := cod_2(X3, X0);
            PILAUX := desempilar(desempilar(PILAUX));
        else  X0 := X3;
    - - Caso 2: se pretende modificar la última posición del vector o quizá una
        incluso posterior
    - - El nuevo valor será en cualquier caso el último de la solución
    - - Si se han creado posiciones nuevas se rellenan con la palabra vacía
            for IND in reverse pred(X2) .. suc(ULT_IND) loop
                X0 := cod_2(ε, X0);
            end loop;
    - - Si la posición modificada es la última, su valor antiguo es retirado de la pila
            if X2 = ULT_IND then
                PILAUX := desempilar(PILAUX);
            end if;
    - - En cualquier caso la posición alterada tendrá el nuevo índice máximo
            ULT_IND := X2;
        end if;
    - - A partir de aquí se reunifican los casos: sólo queda volcar desde la pila los
        valores de índice
    - - anterior al del modificado y terminar de construir el vector solución
        while not es_p_vacía?(PILAUX) loop
            X0 := cod_2(cima(PILAUX), X0);
            PILAUX := desempilar(PILAUX);
        end loop;
        X0 := cod_2(ULT_IND, X0);
```

NOTA: Al igual que se hace en ADA, preferiremos usar en nuestros programas la macroexpresión **A(α)** en lugar de **acceso(A,α)**, donde **A** es una variable vector y α es una expresión natural. Asímismo, la mayor parte de las veces en que se aplica la función *modifica* sobre un cierto vector, se suele asignar el

resultado al propio vector, quedando la asignación de la forma **A:=modifica(A,α,β);**, donde β es una macroexpresión cualquiera. También en estos casos admitiremos la notación de la forma **A(α):=β;**, que denominamos **macroasignaciones a componentes de vectores**.

4.5. Gödelización de los programas while

Los últimos objetos que vamos a querer manipular con nuestros programas while van a ser los propios programas while, que constituyen el tipo WHILE. Al conjunto de valores de este tipo de datos lo denominamos **W**. Una vez los tengamos implementados podremos proceder a estudiar la while-computabilidad de numerosas funciones que tienen que ver con la programación. Algunas de estas operaciones aluden a propiedades estáticas porque se deducen del texto de los programas (evaluables en tiempo de compilación), y otras a propiedades dinámicas porque dependen del funcionamiento de los programas (evaluables en tiempo de ejecución).

Aunque en Informática el hecho de que los programas puedan trabajar sobre otros programas es un hecho cotidiano, su importancia desde el punto de vista teórico es enorme. De hecho, este mecanismo de inmersión por el cuál los elementos de un sistema formal pasan a ser manipulables dentro del propio sistema ha quedado bautizado con el nombre de la primera persona que lo describió: **gödelización**.

Como siempre, nos va a interesar obtener una implementación estricta. La idea que utilizaremos es muy similar a la que se usa en muchos lenguajes-máquina: cada programa se codificará como un par *(c_op, info)*, donde *c_op* es un código de operación que describe qué tipo de programa es, e *info* contiene información relevante adicional (los argumentos de la operación). Por tanto necesitamos seis códigos de operación para los seis tipos de programa while existentes. Pero la función de interpretación ha de ser total, y si sólo tenemos estos seis códigos, ¿cómo interpretamos una palabra cuyo código de operación no sea ninguno de estos seis? Podríamos decidir reasignar estas palabras arbitrariamente a otros programas que ya tienen su código, pero entonces no tendremos inyectividad.

La solución pasa por distribuir un número infinito de códigos entre programas de un tipo en particular. Si el alfabeto sobre el que están definidos los programas while consta de *n* símbolos podemos hacer la siguiente reserva de códigos de operación:

- **0** para los programas de la forma **XI:=Ɛ;**
- **1** ≤ k ≤ **n**, para los de la forma **XI:=cons**$_{a_k}$**(XJ);**
- **n+1** para los de la forma **XI:=cdr(XJ);**
- **n+2** para las composiciones P_1 P_2
- **n+3** ≤ k ≤ **2*n+2** para los condicionales **if car**$_{a_k}$**?(XI) then P end if;**
- valores mayores para los iterativos **while nonem?(XI) loop P end loop;**.

El hecho de incluir, aparte de la clase a la que pertenece, información adicional sobre el programa dentro del código de operación busca precisamente conseguir la biyectividad. Por ejemplo, si el tipo de símbolo que interviene en la función *cons* o en el predicado *car* se codificara como una característica independiente, tendríamos el problema de que sólo puede tomar los valores entre *1* y *k*, por lo que las palabras que tuvieran otro valor en esa componente no representarían en principio ningún programa (habría que interpretarlas arbitrariamente como algún programa que ya tendría su propia representación). El caso de los **while** es aún más claro: estos programas podrán repartirse entre sí todos los códigos de operación mayores que *2*n+2* porque el código contendrá en cada caso información particular del programa.

Para comprender mejor la interpretación elegida viene bien explicitar qué función de representación estamos pretendiendo producir, lo que nos permitirá visualizar cuál es esa "información" que codificamos mediante palabras (recordemos que *n* representa el cardinal del alfabeto $\Sigma = \{a_1, a_2, ..., a_n\}$):

$$\Re_W(\mathbf{XI:=\varepsilon;}) = cod^2(\mathbf{0}, i)$$

$$\Re_W(\mathbf{XI:=cons_{a_k}(XJ);}) = cod^2(\mathbf{k}, cod^2(i, j)) = cod^3(\mathbf{k}, i, j)$$

$$\Re_W(\mathbf{XI:=cdr(XJ);}) = cod^2(\mathbf{n+1}, cod^2(i, j)) = cod^3(\mathbf{n+1}, i, j)$$

$$\Re_W(\mathbf{P_1\ P_2}) = cod^2(\mathbf{n+2}, cod^2(\Re_W(\mathbf{P_1}), \Re_W(\mathbf{P_2})))$$

$$\Re_W(\underline{\mathbf{if}}\ \mathbf{car_{a_k}?(XI)}\ \underline{\mathbf{then}}\ \mathbf{P}\ \underline{\mathbf{end}}\ \underline{\mathbf{if}};) = cod^2(\mathbf{n+2+k}, cod^2(i, \Re_W(\mathbf{P})))$$

$$\Re_W(\underline{\mathbf{while}}\ \mathbf{nonem?(XI)}\ \underline{\mathbf{loop}}\ \mathbf{P}\ \underline{\mathbf{end}}\ \underline{\mathbf{loop}};) = cod^2(\mathbf{2*n+3+i}, \Re_W(\mathbf{P}))$$

Así es como queremos codificar los datos sobre las demás componentes del programa: para el caso de la asignación vacía necesitamos conocer el índice de la variable *i*, en los casos de las asignaciones de tipo *cons* y *cdr* necesitamos conocer los índices de las dos variables intervinientes *i* y *j*, y así sucesivamente. En definitiva, la función de interpretación resulta:

$$
\Im_W(w)=\begin{cases}
XI:=\varepsilon; & \text{si } \Im_N(\text{decod}_{2,1}(w))=0\wedge \\
 & \wedge i=\Im_N(\text{decod}_{2,2}(w)) \\
XI:=\text{cons}_s(XJ); & \text{si } 1\le\Im_N(\text{decod}_{2,1}(w))\le n\wedge \\
 & \wedge i=\Im_N(\text{decod}_{3,2}(w))\wedge \\
 & \wedge s=\text{decod}_{3,1}(w)\wedge \\
 & \wedge j=\Im_N(\text{decod}_{3,3}(w)) \\
XI:=\text{cdr}(XJ); & \text{si } \Im_N(\text{decod}_{2,1}(w))=n+1\wedge \\
 & \wedge\Im_N(\text{decod}_{3,2}(w))=i\wedge \\
 & \Im_N(\text{decod}_{3,3}(w))=j \\
P\ Q & \text{si } \Im_N(\text{decod}_{2,1}(w))=n+2\wedge \\
 & \wedge\Im_W(\text{decod}_{3,2}(w))=P\wedge \\
 & \wedge\Im_W(\text{decod}_{3,3}(w))=Q \\
\text{if car}_s?(XI)\text{ then } P \text{ end if}; & \text{si } n+3\le\Im_N(\text{decod}_{2,1}(w))\le 2*n+2\wedge \\
 & \wedge\Im_N(\text{decod}_{3,2}(w))=i\wedge \\
 & \wedge\Im_N(\text{decod}_{3,1}(w))-n-2=\Im_N(s)\wedge \\
 & \wedge\Im_W(\text{decod}_{3,3}(w))=P \\
\text{while nonem? }(XI)\text{ loop } P \text{ end loop}; & \text{si } \Im_N(\text{decod}_{2,1}(w))\ge 2*n+3\wedge \\
 & \wedge i=\Im_N(\text{decod}_{2,1}(w))-2*n-3\wedge \\
 & \wedge\Im_W(\text{decod}_{2,2}(w))=P
\end{cases}
$$

Entre las operaciones a implementar podemos distinguir tres grupos: en primer lugar están las funciones constructoras, que nos permiten formar inductivamente objetos del tipo WHILE a partir de los datos de las variables, símbolos y/o subprogramas que intervienen en su construcción. Tendremos entonces una operación constructora para cada uno de los siguientes tipos de programa:

$$\text{haz_asig_vacía}: \mathbf{N}\to\mathbf{W}$$

$$\text{haz_asig_cdr}: \mathbf{N}\times\mathbf{N}\to\mathbf{W}$$

$$\text{haz_composisición}: \mathbf{W}\times\mathbf{W}\to\mathbf{W}$$

haz_iteración: **N** × **W** → **W**

y una operación constructora para cada uno de los siguientes tipos de programa y por cada símbolo **s** del alfabeto:

haz_cons_**s**: **N** × **N** → **W**

haz_condición_**s**: **N** × **W** → **W**

Para todo este grupo de funciones los argumentos naturales representan índices de variables que intervienen en la construcción.

Por otro lado nos serán de utilidad los predicados de inspección, que se aplican a programas y dan resultado booleano indicándonos si el programa es de un tipo concreto o no. Tendremos uno para cada tipo de programa:

es_asig_vacía?: **W** → **B**

es_asig_cons?: **W** → **B**

es_asig_cdr?: **W** → **B**

es_composición?: **W** → **B**

es_condición?: **W** → **B**

es_iteración?: **W** → **B**

Por último implementaremos también operaciones de acceso, que nos indican qué elementos intervienen en el programa while; según el tipo de este último podemos saber, por ejemplo, a qué variable se hace la asignación, o qué variable controla en la condición, o qué subprograma interviene en la construcción, etc. A partir de ahora se entiende que los casos que no se citan para cada función corresponden a situaciones de indefinición. La función:

ind_var: **W** → **N**

nos indica, por ejemplo, el índice de la variable principal que forma parte del programa. Su significado puede ser diverso:

- ind_var(**XI:=Ɛ;**) = i
- ind_var(**XI:=cons**$_{a_k}$**(XJ);**) = i
- ind_var(**XI:=cdr(XJ);**) = i
- ind_var(**if car**$_{a_k}$**?(XI) then P end if;**) = i
- ind_var(**while nonem?(XI) loop P end loop;**) = i

No obstante hay algunas instrucciones para las que se utiliza una segunda variable.Para acceder a esta se utiliza la función:

ind_var_2: **W** → **N**

que por tanto cumplirá lo siguiente:

- ind_var_2(**XI:=cons$_{a_k}$(XJ);**) = j
- ind_var_2(**XI:=cdr(XJ);**) = j

También definiremos una función para seleccionar el símbolo (en forma de palabra) que interviene cada vez que actúan las funciones *cons* y los predicados *car*. Hay que advertir que, a diferencia de las constructoras, las funciones de inspección no utilizan este dato, y de ahí la necesidad de esta función de acceso:

$$\text{ind_simb}: \mathbf{W} \rightarrow \Sigma^*$$

donde:

- ind_simb(**XI :=cons$_{a_k}$(XJ);**) = $\mathbf{a_k}$
- ind_simb(**<u>if</u> car$_{a_k}$?(XI) <u>then</u> P <u>end</u> <u>if</u>;**) = $\mathbf{a_k}$

Llamamos instrucción interna de un programa no básico al subprograma sobre el que se construye. Cuando trabajemos con objetos del tipo WHILE, una de las acciones fundamentales consistirá precisamente en descomponer los programas en subprogramas, y por ello mismo necesitaremos la función:

$$\text{inst_int}: \mathbf{W} \rightarrow \mathbf{W}$$

que nos devuelve el subprograma contenido en un programa while compuesto, y que por tanto no será de aplicación en los programas básicos:

- inst_int($\mathbf{P_1}$ $\mathbf{P_2}$) = $\mathbf{P_1}$
- inst_int(**<u>if</u> car$_s$?(XI) <u>then</u> P <u>end</u> <u>if</u>;**) = **P**
- inst_int(**<u>while</u> nonem?(XI) <u>loop</u> P <u>end</u> <u>loop</u>;**) = **P**

Pero dado que la composición de programas incorpora dos subprogramas, es necesaria la operación:

$$\text{inst_int_2}: \mathbf{W} \rightarrow \mathbf{W}$$

que únicamente está definida para ese caso:

- inst_int_2($\mathbf{P_1}$ $\mathbf{P_2}$) = $\mathbf{P_2}$

Citaremos en los programas la constante NSIM, que corresponderá al cardinal del alfabeto. Empezaremos con las funciones constructoras, que no tienen ninguna dificultad a la luz de la función de representación. Su encabezamiento común será el siguiente:

```
package COMPUTABLES is
    type NATURAL is STRING;
    type BOOLEAN is STRING;
```

```
        function cod_2 (NATURAL, NATURAL) return STRING;
        function const_0 (STRING) return NATURAL;
        function "" (STRING) return STRING;
        function cod_3 (STRING, STRING, STRING) return STRING;
  - - Se entiende que la siguiente función se fijará para cada alfabeto particular
        function const_nsim (STRING) return NATURAL;
        function "+" (NATURAL, NATURAL) return NATURAL;
        function const_1 (STRING) return NATURAL;
        function proy_2_1 (STRING, STRING) return STRING;
        function proy_2_2 (STRING, STRING) return STRING;
    end COMPUTABLES;
```

haz_asig_vacía:

X0 := cod_2(0, X1);

haz_asig_cdr:

X0 := cod_3(NSIM+1, X1, X2);

haz_composición:

X0 := cod_3(NSIM+2, X1, X2);

haz_iteración:

X0 := cod_2(2*NSIM+3+X1, X2);

*haz_cons_*$\boldsymbol{a_k}$

X0 := cod_3(K, X1, X2);

y *haz_condición_*$\boldsymbol{a_k}$

X0 := cod_3(NSIM+2+K, X1, X2);

También las funciones de inspección dispondrán de un encabezamiento común, pues utilizan todas ellas macros similares (de hacho su while-recursividad podría justificarse por la proposición **2**):

```
    package COMPUTABLES is
        type BOOLEAN is STRING;
        type NATURAL is STRING;
        function "=" (NATURAL, NATURAL) return BOOLEAN;
        function decod_2_1 (STRING) return STRING;
        function const_0 (STRING) return NATURAL;
        function const_true (STRING) return BOOLEAN;
        function const_false (STRING) return BOOLEAN;
        function ">=" (NATURAL, NATURAL) return BOOLEAN;
        function "<= " (NATURAL, NATURAL) return BOOLEAN;
        function const_1 (STRING) return NATURAL;
```

```
        function const_nsim (STRING) return NATURAL;
        function "+" (NATURAL, NATURAL) return NATURAL;
        function const_2 (STRING) return NATURAL;
        function const_3 (STRING) return NATURAL;
        function "*" (NATURAL, NATURAL) return NATURAL;
    end COMPUTABLES;
```

es_asig_vacía?:

```
    if decod_2_1(X1) = 0 then X0 := true;
    else  X0 := false;
    end if;
```

es_asig_cons?:

```
    if decod_2_1(X1) >= 1 and decod_2_1(X1) <= NSIM then
        X0 := true;
    else  X0 := false;
    end if;
```

es_asig_cdr?:

```
    if decod_2_1(X1) = NSIM + 1 then X0 := true;
    else  X0 := false;
    end if;
```

es_composición?:

```
    if decod_2_1(X1) = NSIM + 2 then X0 := true;
    else  X0 := false;
    end if;
```

es_condición?:

```
    if decod_2_1(X1) >= NSIM + 3 and decod_2_1(X1) <= 2 * NSIM + 2 then
        X0 := true;
    else  X0 := false;
    end if ;
```

es_iteración?:

```
    if decod_2_1(X1) >= 2 * NSIM + 3 then
        X0 := true;
    else  X0 := false;
    end if;
```

Por último hay que implementar las funciones de acceso, teniendo cuidado en las definiciones por casos según sea el tipo de programacomo

```
    package COMPUTABLES is
        type BOOLEAN is STRING;
        type NATURAL is STRING;
```

```
        function decod_2_1 (STRING) return STRING;
        function const_0 (STRING) return NATURAL;
        function decod_2_2 (STRING) return STRING;
        function const_1 (STRING) return NATURAL;
        function const_nsim (STRING) return NATURAL;
        function "+" (NATURAL, NATURAL) return NATURAL;
        function decod_3_2 (STRING) return STRING;
        function const_2 (STRING) return NATURAL;
        function ⊥⊥ (STRING) return STRING;
        function const_3 (STRING) return NATURAL;
        function "*" (NATURAL, NATURAL) return NATURAL;
        function "–" (NATURAL, NATURAL) return NATURAL;
        function ">=" (NATURAL, NATURAL) return BOOLEAN;
        function "<= " (NATURAL, NATURAL) return BOOLEAN;
        function decod_3_3 (STRING) return STRING;
        function "=" (NATURAL, NATURAL) return BOOLEAN;
    end COMPUTABLES;
```

ind_var:

```
    case decod_2_1(X1) is
    when 0 => X0 := decod_2_2(X1);
    when 1 .. NSIM+1 => X0 := decod_3_2(X1);
    when NSIM+2 => X0 := ⊥⊥;
    when NSIM+3 .. 2*NSIM+2 => X0 := decod_3_2(X1);
    when others  => X0 := decod_2_1(X1) – 2 * NSIM – 3;
    end case;
```

ind_var_2:

```
    if decod_2_1(X1) >= 1 and decod_2_1(X1) <= NSIM + 1 then
        X0 := decod_3_3(X1);
    else  X0 := ⊥⊥;
    end if;
```

ind_simb:

```
    case decod_2_1(X1) is
    when 1 .. NSIM => X0 := decod_2_1(X1);
    when NSIM+3 .. 2*NSIM+2 => X0 := decod_2_1(X1) – NSIM – 2;
    when others  => X0 := ⊥⊥;
    end case;
```

inst_int:

```
    case decod_2_1(X1) is
    when 0 .. NSIM+1 => X0 := ⊥⊥;
    when NSIM+2 => X0 := decod_3_2(X1);
    when NSIM+3 .. 2*NSIM+2 => X0 := decod_3_3(X1);
```

```
when others  => X0 := decod_2_2(X1);
end case;
```

inst_int_2:

```
if decod_2_1(X1) = NSIM+2 then
        X0 := decod_3_3(X1);
else X0 := ⊥⊥ ;
end if;
```

Con estas operaciones realmente estaremos en condiciones de realizar operaciones muy complejas con los programas while, incluyendo su ejecución, colocación de trazas o incluso manipulación de sus resultados. De todas formas esto queda fuera del objetivo del presente informe, que no obstante tendrá continuación en otro trabajo dedicado a aplicar las bases que aquí hemos sentado para deducir los resultados más relevantes de la Teoría de la Computabilidad.

5. Conclusiones

Pretendemos que los capítulos anteriores sirvan como base a un estudio de la Teoría de la Computabilidad por parte de estudiantes de primer ciclo sin formación teórica previa, pero con alguna experiencia de programación en lenguajes de alto nivel. Aunque no se incluye ningún resultado específico, se incorporan los elementos fundamentales para permitir la utilización del modelo aquí presentado en su consecución. Por ejemplo, las demostraciones de los Teoremas de Enumeración, S-M-N ó de Recursión (fundamentalmente el segundo, conocido también como Teorema de Kleene), que normalmente ocupan gran extensión se pueden hacer en un par de hojas cada una si se da por sentado el material del presente texto (otra cosa es el esfuerzo de comprensión que puedan requerir dichas pruebas, que no pretendemos cuantificar). De alguna forma hemos tratado de reunir la "cara amable" de la Teoría de la Computabilidad, es decir, la parte en la que se desarrolla positivamente un modelo general de computación (en este caso los programas while), comprobando hasta dónde llegan sus potencialidades. Asímismo es la parte en la que la programación en su sentido práctico es más importante, por lo que al alumno o alumna los conceptos básicos no le tienen por qué sonar lejanos.

La Teoría de la Computabilidad es una disciplina muy antigua. Para cuando aparecieron los primeros ordenadores basados tecnológicamente en la electrónica digital (finales de los años 40) ya se habían establecido sus presupuestos fundamentales. Es decir, que se supo cuáles serían las principales limitaciones de los computadores antes incluso de que estos fueran construidos.

Esta antigüedad, siendo como es cosa de admirar, tiene también sus desventajas: los modelos y las notaciones que en principio se usaron para probar resultados de incomputabilidad se han quedado prácticamente sin alterar desde entonces. El modelo más utilizado de computador abstracto sigue siendo con mucho la Máquina de Turing, y el proceso de computación es visto en todos los textos clásicos como una manipulación de números, lo cual es un singular hábito que nos viene de los tiempos de Gödel.

Para el o la estudiante universitaria de Ingeniería Informática esto suele suponer un problema. Por un lado la Máquina de Turing le resulta un dispositivo ajeno en el que la programación resulta poco menos que un suplicio. Por si fuera poco, esta lejanía entre el modelo teórico y los computadores reales (con los que está empezando a familiarizarse) provocan un escepticismo con respecto a los resultados: "Aunque hayamos probado que una Máquina de Turing no es capaz

de resolver este problema, ¿como podemos tener la seguridad de que ningún otro computador puede hacerlo? ¿Qué tiene que ver una cosa con la otra?". Además, si su toma de contacto con la Informática Teórica incluye un curso básico sobre Teoría de Autómatas y Lenguajes Formales, se produce una fractura adicional: se empiezan estudiando modelos de computación (los autómatas finitos, con pila y/o lineales acotados) que operan sobre una cadena arbitraria de símbolos para producir un resultado booleano y se pasa sin solución de continuidad a dispositivos de respuesta arbitraria que realizan operaciones aritméticas, por lo que es muy difícil integrar los resultados de Teoría de Lenguajes siquiera como una avanzadilla de la Teoría de la Computabilidad.

Ha habido varios intentos para resolver el primer problema y acercar la computabilidad a las personas con una formación eminentemente práctica. En particular las Máquinas de Registros de Cutland o los Programas While de Kfoury, Arbib y Moll son buenos ejemplos de cómo se puede aprovechar la experiencia programadora en la enseñanza de los conceptos más abstractos de la Informática. Pero desgraciadamente el segundo problema (dar una continuidad entre la Teoría de Autómatas y la Teoría de la Computabilidad) ha recibido mucha menos atención. Ese es precisamente el propósito del presente texto: retomar los Programas While aprovechando sus ventajas y reformularlos de manera que la computación quede definida en términos de manipulación de símbolos arbitrarios, algo que está mucho más en concordancia con la realidad informática.