Computabilidad

Este documento es un resumen de las ideas principales
que se recogen en el capítulo "Computabilidad" del libro:
"Teoría de la Computación: Lenguajes Formales, Autómatas y Complejidad",
J. Glenn Brookshear,
Addison-Wesley Iberoamericana, 1993.

Hasta ahora hemos visto que el poder de las máquinas de Turing como aceptadores de lenguajes corresponde a los poderes generativos de las gramáticas. En este tema investigaremos otros casos en los cuales el poder computacional de las máquinas de Turing corresponde a las capacidades de otros sistemas computacionales.

Comenzaremos nuestra investigación considerando el poder expresivo de un determinado lenguaje de programación. Recordemos que existen funciones que no pueden calcularse con ningún programa escrito en ese lenguaje. La pregunta que debemos plantearnos ahora es si esta limitación es debida al diseño del lenguaje, o si, por el contrario, es un reflejo de las limitaciones de los procesos algorítmicos en general. Es decir, nos preguntamos si la imposibilidad de calcular una función por un programa en un determinado lenguaje significa que el lenguaje es incapaz de expresar un algoritmo, o más bien que no existe ningún algoritmo que pueda calcular esa función.

Para responder esta pregunta, primero identificaremos una clase de funciones que contiene la totalidad de las funciones computables, es decir, todas aquellas funciones que pueden calcularse por medios algorítmicos, sin importar cómo pueda expresarse o implementarse ese algoritmo. Después mostraremos que las funciones de esta clase pueden calcularse por medio de las máquinas de Turing, y también mediante algoritmos expresados en un subconjunto muy sencillo de la mayoría de los lenguajes de programación. A partir de esto llegaremos a la conclusión de que los límites detectados en las máquinas de Turing y en la mayor parte de los lenguajes de programación son el reflejo de las limitaciones de los procesos computacionales, no del diseño de la máquina o del lenguaje que se emplee.

1 Fundamentos de la teoría de funciones recursivas

Hasta ahora, nuestro estudio de los procesos computacionales y sus capacidades se ha basado en un enfoque operativo, no funcional. Nos hemos centrado en cómo se lleva a cabo un cálculo, no en lo que este cálculo logra. Sin embargo, nuestro objetivo es identificar las funciones que pueden calcularse con al menos un sistema computacional, así que debemos alejarnos de cualquier método específico para expresar o ejecutar los cálculos. En otras palabras, queremos utilizar el término *computable* para decir que una función puede calcularse mediante algún algoritmo, más que por un algoritmo que pueda implementarse en algún sistema específico.

Para obtener esta generalidad, adoptaremos un enfoque funcional que nos permita estudiar la *computabilidad*, centrándonos en determinar qué funciones pueden ser calculadas, y no en la forma de calcularlas. Más concretamente, utilizaremos el método que han seguido los matemáticos para el estudio de las funciones recursivas. Por tanto, comenzaremos con un conjunto de funciones,

llamadas funciones iniciales, cuya sencillez es tal que no hay dudas sobre su computabilidad, para luego mostrar que estas funciones se pueden combinar formando otras cuya computabilidad se deduce de las funciones originales. De esta manera, obtendremos una colección de funciones que contiene todas las funciones computables. Así pues, si un sistema computacional, como un lenguaje de programación o un ordenador, abarca todas estas funciones, concluiremos que el sistema tiene todo el poder posible. De lo contrario, dicho sistema estará necesariamente restringido.

Funciones parciales. Las funciones pueden abarcar una diversidad muy amplia de dominios y codominios. Para enfrentarnos a esta diversidad, es necesario observar que cualquier dato se puede codificar mediante una cadena de ceros y unos y, en el contexto de un sistema de codificación adecuado, es posible considerar entonces cualquier función computable como una función cuyas entradas sean tuplas de enteros no negativos. No obstante, esto no quiere decir que toda función computable tenga la forma

$$f : \mathbb{N}^m \to \mathbb{N}^n$$

donde m y n son enteros de $\mathbb{N}$. De hecho la función *división* definida por

$$división(x, y) = \text{la parte entera de } x/y, \text{ donde } x, y \in \mathbb{N} \text{ y } y \neq 0$$

cuyo dominio contiene pares de enteros, debería estar en nuestra colección de funciones computables. Sin embargo, *división* no es una función de la forma

$$f : \mathbb{N}^2 \to \mathbb{N}$$

ya que no está definida para los pares donde la segunda componente es cero.

Para tener en cuenta las funciones cuyos dominios no incluyen todo $\mathbb{N}^m$ para un m dado, presentaremos el concepto de *función parcial*. Una función parcial de un conjunto X es aquélla cuyo dominio constituye un subconjunto de X. Para indicar que el subconjunto es propio, es decir, que una función parcial de X efectivamente se encuentra indefinida por lo menos para un elemento de X, nos referiremos a ella como *estrictamente parcial*. Por otra parte, se llamará *función total* de X a una función parcial de X cuyo dominio es todo el conjunto X. De esta forma, tanto la función *división* definida anteriormente, como la función *suma* definida por

$$suma(x, y) = x + y$$

son funciones parciales de $\mathbb{N}^2$ en $\mathbb{N}$. Para mayor precisión, *suma* es una función total en $\mathbb{N}^2$, mientras que *división* es una función parcial en $\mathbb{N}^2$.

Por tanto, aplicando los sistemas de codificación adecuados, es posible identificar cualquier función computable como una función parcial de la forma $f : \mathbb{N}^m \to \mathbb{N}^n$, para m y n en $\mathbb{N}$. Así pues, nuestra búsqueda de todas las funciones computables se puede restringir a las funciones parciales de $\mathbb{N}^m$ en $\mathbb{N}^n$, donde m y n están en $\mathbb{N}$.

Por último, dado que con frecuencia haremos referencia a n-tuplas arbitrarias de enteros, es conveniente adoptar la sencilla notación $\bar{x}$ para representar una tupla de la forma $(x_1, x_2, \ldots, x_n)$ en aquellos casos donde no se requieren los detalles de la forma expandida de dicha tupla.

Funciones iniciales. La base de la jerarquía de las funciones computables que veremos está constituida por el siguiente conjunto de funciones, que llamaremos *funciones iniciales*:

- Una de estas funciones es la *función cero*, representada por ζ. Esta función establece una correspondencia entre la cero-tupla o tupla vacía y el entero cero, y se escribe $\zeta() = 0$.

  Así pues, ζ corresponde al proceso de escribir un cero en un trozo de papel en blanco o en una celda de memoria de un ordenador. Ambas tareas están de acuerdo con nuestro concepto intuitivo de proceso computacional, por lo que resulta sencillo aceptar que ζ es una función que debe clasificarse como computable.

- Otra de las funciones iniciales es la *función sucesor*, representada por σ. Esta función establece una correspondencia entre tuplas de una sola componente, de tal manera que $\sigma(x) = x + 1$, para cada entero no negativo x.

  Dado que existe un proceso computacional para la suma de enteros, también debemos considerar a σ como una función computable.

- La clase de funciones iniciales se completa con una colección de funciones conocidas como *funciones de proyección*. Cada una de estas funciones extrae como salida una componente específica de su tupla de entrada. Para representar una función de proyección, utilizamos el símbolo π junto con un superíndice que indica el tamaño de la tupla de entrada y un subíndice que indica la componente extraída. Por ejemplo, $\pi_2^3(7, 6, 4) = 6$. Como caso especial, consideraremos π_0^n, que establece una correspondencia entre tuplas de tamaño n y la tupla vacía, por lo que, por ejemplo, $\pi_0^2(6, 5) = ()$.

  Cualquier función π_m^n podría calcularse aplicando el procedimiento de recorrer la n-tupla de entrada hasta localizar el elemento m-ésimo y después extraer el valor entero allí encontrado. Así pues, al igual que con el resto de las funciones iniciales, es sencillo asumir que las proyecciones deben estar también en la clase de las funciones computables.

Las funciones iniciales forman la base de la jerarquía que existe en la teoría de funciones recursivas. Por supuesto, se trata de funciones que por sí solas no pueden lograr mucho. Por lo tanto, nuestro siguiente objetivo será estudiar cómo pueden emplearse estas funciones para construir otras más complejas.

Funciones recursivas primitivas. Existen varias formas de construir funciones más complejas a partir de las iniciales, de las cuales consideraremos las siguientes:

- La primera técnica es la denominada *combinación*. La combinación de dos funciones $f : \mathbb{N}^k \to \mathbb{N}^m$ y $g : \mathbb{N}^k \to \mathbb{N}^n$ es la función $f \times g : \mathbb{N}^k \to \mathbb{N}^{m+n}$, la cual viene definida por $f \times g\,(\bar{x}) = (f(\bar{x}), g(\bar{x}))$, donde $\bar{x}$ es una k-tupla. Es decir, la función $f \times g$ toma como entrada tuplas de tamaño k y produce como salida tuplas de tamaño $m + n$, donde las primeras m componentes se obtienen a partir de la salida de f y las n restantes a partir de la salida de g. Por ejemplo, $\pi_1^3 \times \pi_3^3\,(4, 6, 8) = (4, 8)$.

  Suponiendo que existe una forma de calcular las funciones f y g, podemos calcular $f \times g$ calculando primero por separado f y g, y después combinando sus salidas. Llegamos entonces a la conclusión de que la combinación de funciones computables también es computable.

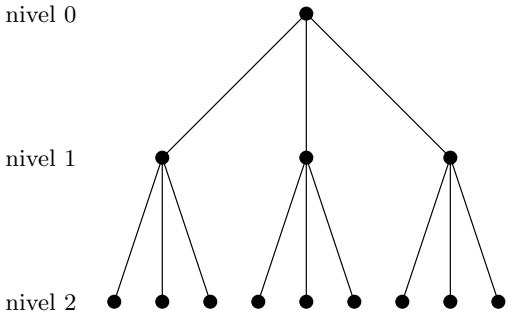
- La *composición* representa otro método para formar funciones más complejas. La composición de dos funciones $f : \mathbb{N}^k \to \mathbb{N}^m$ y $g : \mathbb{N}^m \to \mathbb{N}^n$ es la función $g \circ f : \mathbb{N}^k \to \mathbb{N}^n$ definida por $g \circ f\,(\bar{x}) = g(f(\bar{x}))$, donde $\bar{x}$ es una k-tupla. Por ejemplo, $\sigma \circ \zeta\,() = \sigma(\zeta()) = \sigma(0) = 1$.

  Calculando primero f y utilizando su salida como entrada para g, si f y g son computables, se deduce que la composición de ambas es también computable.

- La última técnica de construcción que veremos en esta etapa se denomina *recursividad primitiva*. Supongamos que queremos definir una función $f : \mathbb{N}^2 \to \mathbb{N}$ tal que $f(x, y)$ sea el número de nodos en un árbol completo y equilibrado en el cual cada nodo que no es una hoja tiene exactamente x hijos, y cada camino desde la raíz a un nodo hoja contiene y arcos (se dice entonces que este árbol tiene profundidad y). El primer paso es observar que cada nivel del árbol tiene exactamente x^n nodos, donde n es el número de nivel. De esta forma, con un valor fijo de x, para determinar el número de nodos de un árbol con profundidad $y + 1$, basta con sumar $x^{y+1} = x^y x$ al número de nodos del árbol de profundidad y. Esto, unido al hecho de que un árbol con un solo nodo raíz contiene x^0 nodos, permite definir f recursivamente con el siguiente par de fórmulas:

$$f(x, 0) = x^0$$
$$f(x, y + 1) = f(x, y) + x^y x$$

Por ejemplo, podemos calcular $f(3, 2)$ como sigue:



$$\begin{aligned} f(3,2) &= f(3,1) + 3^1 3 \\ &= f(3,1) + 9 \\ &= f(3,0) + 3^0 3 + 9 \\ &= f(3,0) + 12 \\ &= 3^0 + 12 \\ &= 13 \end{aligned}$$

En un contexto más general, lo que hemos hecho es definir f en términos de otras dos funciones. Una de ellas es $g : \mathbb{N} \to \mathbb{N}$ tal que $g(x) = 1, \forall x \in \mathbb{N}$. Y la otra es $h : \mathbb{N}^3 \to \mathbb{N}$ tal que $h(x, y, z) = z + x^y x$. Al emplear estas dos funciones, f está definida recursivamente por las fórmulas:

$$f(x, 0) = g(x)$$
$$f(x, y + 1) = h(x, y, f(x, y))$$

En este caso diremos entonces que f está construida a partir de g y de h por medio de recursividad primitiva. De manera aún más general, la recursividad primitiva es una técnica que permite construir una función f que establece la correspondencia entre $\mathbb{N}^{k+1}$ y $\mathbb{N}^m$ a partir de otras dos funciones que relacionan $\mathbb{N}^k$ con $\mathbb{N}^m$ y $\mathbb{N}^{k+m+1}$ con $\mathbb{N}^m$, respectivamente, tal y como se muestra en las siguientes ecuaciones:

$$f(\bar{x}, 0) = g(\bar{x})$$
$$f(\bar{x}, y + 1) = h(\bar{x}, y, f(\bar{x}, y))$$

donde $\bar{x}$ representa una k-tupla.

Empleando la recursividad primitiva junto con las funciones y operaciones que hemos presentado anteriormente, tenemos por ejemplo que la función *suma* se puede definir como:

$$suma(x, 0) = x$$
$$suma(x, y + 1) = 1 + suma(x, y)$$

es decir:

$$suma(x, 0) = \pi_1^1(x)$$
$$suma(x, y + 1) = \sigma[\pi_3^3(x, y, suma(x, y))]$$

donde $g = \pi_1^1$ y $h = \sigma \circ \pi_3^3$.

Una función construida mediante recursividad primitiva a partir de funciones computables es necesariamente computable. Más concretamente, si f está definida a partir de g y h, el papel de estas dos últimas funciones es el de permitir que $f(\bar{x}, y)$ se calcule en función de $f(\bar{x}, 0)$, $f(\bar{x}, 1)$, $f(\bar{x}, 2)$, hasta llegar a $f(\bar{x}, y)$. Por tanto, si g y h son computables, f también lo será.

En este momento estamos en condiciones de definir el siguiente escalón de la clasificación, el de la clase de las *funciones recursivas primitivas*. Esta clase está formada por las funciones que pueden construirse a partir de las funciones iniciales aplicando un número finito de combinaciones, composiciones y recursividades primitivas. Esta clase es muy extensa e incluye, si no todas, la mayoría de las funciones totales que se requieren en las aplicaciones de ordenador tradicionales.

Por último, debemos señalar que si f es una función recursiva primitiva, entonces f debe ser efectivamente total. De hecho, las funciones iniciales son totales, y las técnicas de construcción (combinación, composición y recursividad primitiva) producen funciones totales cuando se aplican a funciones totales. Por tanto, la clase de las funciones recursivas primitivas no completa nuestra jerarquía de funciones computables. Por ejemplo, *división* es computable, pero no es recursiva primitiva ya que no es total. Además, veremos también que existen funciones totales computables que no son recursivas primitivas. Por consiguiente, en la próxima sección ampliaremos el repertorio de funciones recursivas primitivas, y después continuaremos la búsqueda de funciones computables más allá de esta clase.

2 Alcance de las funciones recursivas primitivas

Como hemos dicho, el primer objetivo de esta sección es ampliar el repertorio de las funciones que sabemos que son recursivas primitivas, proporcionando una serie de ejemplos útiles que suelen aparecer en las aplicaciones de ordenador tradicionales. Posteriormente, el segundo objetivo será el de esclarecer la diferencia entre la clase de las funciones recursivas primitivas y la clase de las funciones totales computables.

Ejemplos de funciones recursivas primitivas. Comenzaremos nuestro estudio de las funciones recursivas primitivas considerando la colección de las *funciones constantes*, las cuales producen una salida fija predeterminada, sin importar cuál sea la entrada. Las funciones constantes cuyas salidas son tuplas de un solo elemento están representadas por K_m^n, donde n indica la dimensión del dominio y m es el valor de salida de la función. Así, K_3^2 establece la correspondencia entre cualquier tupla de dos elementos y el entero 3. Cualquier función constante de la forma K_m^n es recursiva primitiva, ya que puede calcularse mediante la composición de las siguientes funciones iniciales: la proyección π_0^n, para la obtención de la tupla vacía a partir de cualquier n-tupla; la función ζ para la obtención de 0; y m aplicaciones de la función σ para la obtención del valor m. Por ejemplo:

$$K_3^2(x_1, x_2) = \sigma \circ \sigma \circ \sigma \circ \zeta \circ \pi_0^2 \ (x_1, x_2) = 3$$

También son recursivas primitivas las funciones constantes cuyas salidas tienen más de una componente, ya que no son más que combinaciones de funciones de la forma K_m^n. Por ejemplo, la función que establece la correspondencia entre cualquier tupla de tamaño 3 y el par $(2, 5)$ es $K_2^3 \times K_5^3$. Para evitar notaciones complicadas, en aquellos casos en los que la dimensión del dominio sea evidente o no tenga importancia para el análisis, representaremos las funciones

constantes directamente con su valor de salida, es decir, 3 en lugar de K_3^2 o $(2,5)$ en lugar de $K_2^3 \times K_5^3$.

Utilizando las funciones constantes y la función *suma* definida en la sección anterior, podemos mostrar de la manera siguiente que la función *producto* es también recursiva primitiva:

$$\begin{aligned} &producto(x,0) = 0 \\ &producto(x,y+1) = x + producto(x,y) \end{aligned}$$

es decir:

$$\begin{aligned} &producto(x,0) = K_0^1(x) \\ &producto(x,y+1) = suma[\pi_1^3 \times \pi_3^3(x,y,producto(x,y))] \end{aligned}$$

donde $g = K_0^1$ y $h = suma \circ (\pi_1^3 \times \pi_3^3)$.

De forma similar, podemos mostrar que la función *potencia* es también recursiva primitiva:

$$\begin{aligned} &potencia(x,0) = 1 \\ &potencia(x,y+1) = x * potencia(x,y) \end{aligned}$$

es decir:

$$\begin{aligned} &potencia(x,0) = K_1^1(x) \\ &potencia(x,y+1) = producto(x, potencia(x,y)) \end{aligned}$$

Obsérvese que $g = K_0^1$ y que realmente $h = producto \circ (\pi_1^3 \times \pi_3^3)$. Sin embargo, a veces usaremos notaciones menos formales, como $producto(x, potencia(x,y))$ en este caso, con el fin de que determinadas expresiones sean más legibles.

La función *predecesor* establece la correspondencia entre tuplas de un solo elemento de tal forma que 0 corresponde a 0, y los números mayores que 0 se hacen corresponder con su predecesor en el orden natural de $\mathbb{N}$. Es decir, $predecesor(1) = 0$, $predecesor(2) = 1$, $predecesor(3) = 2$, etc. Para ver que *predecesor* es recursiva primitiva, observamos que puede definirse como:

$$\begin{aligned} &predecesor(0) = \zeta() \\ &predecesor(y+1) = \pi_1^2(y, predecesor(y)) \end{aligned}$$

En cierta forma, *predecesor* es la inversa de σ y, de hecho, puede utilizarse para definir la función de substracción propia, que llamaremos *monus*, de la misma manera que usamos σ para desarrollar la suma:

$$\begin{aligned} &monus(x,0) = x \\ &monus(x,y+1) = predecesor(monus(x,y)) \end{aligned}$$

Así pues, $monus(x,y)$ es $x - y$ si $x \geq y$, y 0 en caso contrario. Con frecuencia $monus(x,y)$ se representa como $x \dot{-} y$.

Otra tarea computacional muy común es determinar si dos valores son iguales. Este proceso está modelado por la función *igual*, definida como:

$$igual(x,y) = \begin{cases} 1 \text{ si } x = y \\ 0 \text{ si } x \neq y \end{cases}$$

Se observa que la función *igual* es recursiva primitiva, ya que:

$$igual(x,y) = 1 \dot{-} ((y \dot{-} x) + (x \dot{-} y))$$

Por ejemplo,

$$\begin{aligned} igual(5,3) &= 1 \dot{-} ((3 \dot{-} 5) + (5 \dot{-} 3)) \\ &= 1 \dot{-} (0 + 2) \\ &= 1 \dot{-} 2 \\ &= 0 \end{aligned}$$

y

$$\begin{aligned} igual(5,5) &= 1 \dot{-} ((5 \dot{-} 5) + (5 \dot{-} 5)) \\ &= 1 \dot{-} (0 + 0) \\ &= 1 \dot{-} 0 \\ &= 1 \end{aligned}$$

También podemos "negar" cualquier tipo de función $f : \mathbb{N}^n \to \mathbb{N}$ para producir otra función $\neg f : \mathbb{N}^n \to \mathbb{N}$ que sea 1 cuando f es 0 y 0 cuando f es 1, definiendo $\neg f$ como $monus \circ (K_1^n \times f)$, es decir, $\neg f(x) = 1 \dot{-} f(x)$. Por ejemplo:

$$\neg igual(x,y) = \begin{cases} 1 \text{ si } x \neq y \\ 0 \text{ si } x = y \end{cases}$$

Otra colección de funciones recursivas primitivas es aquélla que comprende las que pueden definirse en una tabla en la cual se presenta explícitamente un número finito de entradas posibles junto con sus valores de salida correspondientes, y todas las demás entradas se asocian a un único valor común. A estas funciones las llamaremos *funciones tabulares*. Un ejemplo sería la función f definida por:

$$f(x) = \begin{cases} 3 \text{ cuando } x = 0 \\ 5 \text{ cuando } x = 4 \\ 2 \text{ en los demás casos} \end{cases}$$

Cualquier función tabular es recursiva primitiva ya que puede expresarse como una suma finita de: una serie de productos de funciones constantes y funciones de igualdad, correspondientes a los casos de los valores de entrada citados explícitamente; y un último producto de una función constante y de una serie de funciones de igualdad negadas, correspondiente al caso de todas las demás entradas. Por ejemplo, una definición equivalente de la función f anterior es la siguiente:

$$\begin{aligned} f(x) &= producto(3, igual(x,0)) \\ &+ producto(5, igual(x,4)) \\ &+ producto(2, producto(\neg igual(x,0), \neg igual(x,4))) \end{aligned}$$

Como último ejemplo, mostramos que la función $cociente : \mathbb{N}^2 \to \mathbb{N}$ definida por:

$$cociente(x,y) = \begin{cases} \text{la parte entera de } x/y \text{ si } y \neq 0 \\ 0 \text{ si } y = 0 \end{cases}$$

es recursiva primitiva. Efectivamente, se trata de una versión total de la función parcial *división*. Básicamente, este ejemplo ilustra que, aunque la recursividad primitiva se define formalmente empleando como índice la última componente de la tupla de entrada, el uso de proyecciones y combinaciones nos permite aplicar la recursividad en base a otros índices y permanecer en la clase de las funciones recursivas primitivas. Así pues, *cociente* es recursiva primitiva ya que puede definirse como sigue:

$$\begin{aligned} &cociente(0,y) = 0 \\ &cociente(x+1,y) = cociente(x,y) + igual(x+1, producto(cociente(x,y), y) + y) \end{aligned}$$

Más allá de las funciones recursivas primitivas. Regresando ahora al objetivo principal de este tema, repasamos la jerarquía de funciones computables que hemos presentado hasta ahora, la cual se muestra en la figura 1. Hemos visto las funciones iniciales y cómo se pueden emplear estas funciones elementales para construir las funciones recursivas primitivas, todas las cuales son funciones totales computables. Sin embargo, tal y como se muestra en dicha figura, hemos dicho también que la clase de las funciones recursivas primitivas no abarca toda la colección de funciones computables, ya que:

- por una parte, existen funciones computables que no son totales, como por ejemplo la función *división*,
- y, por otra parte, existen funciones computables totales que no son recursivas primitivas, como por ejemplo la *función de Ackermann*.

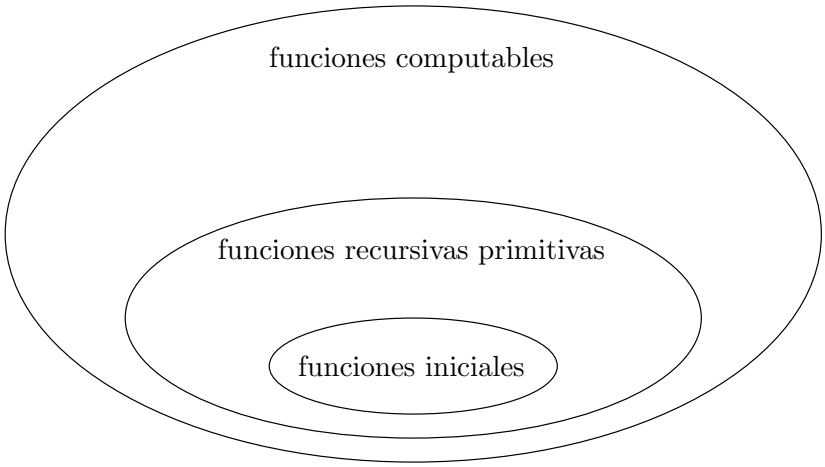


Figura 1: Jerarquía provisional de las funciones computables

Efectivamente, Ackermann presentó en 1928 un ejemplo de función $A : \mathbb{N}^2 \to \mathbb{N}$ definida por las ecuaciones:

$$\begin{aligned} &A(0, y) = y + 1 \\ &A(x + 1, 0) = A(x, 1) \\ &A(x + 1, y + 1) = A(x, A(x + 1, y)) \end{aligned}$$

y demostró que dicha función es computable y total, pero no recursiva primitiva. Nosotros no incluiremos esta demostración aquí, ya que lo importante es que en nuestro estudio no necesitamos un ejemplo específico de función computable que no sea recursiva primitiva. Lo único que necesitamos saber es que existe este tipo de funciones, tal y como enuncia el siguiente teorema:

> El conjunto de funciones recursivas primitivas es un subconjunto propio del conjunto de funciones totales computables.

Para demostrar este teorema, es suficiente con demostrar el siguiente enunciado equivalente:

> Existe una función total computable de $\mathbb{N}$ a $\mathbb{N}$ que no es recursiva primitiva.

La demostración parte de la idea de que, con una codificación adecuada, es posible representar mediante una cadena finita de símbolos a toda función primitiva recursiva de $\mathbb{N}$ a $\mathbb{N}$ que se

construya a partir de las funciones iniciales por medio de un número finito de combinaciones, composiciones y recursividades primitivas. Por tanto, es posible también asignar un orden a las funciones recursivas primitivas, colocando las codificaciones más cortas precediendo a las más largas y, dentro de las de igual longitud, respetando el orden alfabético de dichas codificaciones. En términos de este ordenamiento, podemos entonces hablar de la primera función recursiva primitiva (representada por f_0), de la segunda función (representada por f_1), de la tercera (representada por f_2), etc. Ahora definimos la función $g : \mathbb{N} \to \mathbb{N}$ tal que $g(n) = f_n(n) + 1$, para todo $n \in \mathbb{N}$. Entonces g es total y computable, ya que puede calcularse como sigue: primero localizamos la función recursiva primitiva f_n, después la aplicamos sobre n y finalmente sumamos 1. Sin embargo, aplicando la técnica de diagonalización, deducimos que g no puede ser recursiva primitiva. Si lo fuera, g tendría que ser igual a f_m para algún $m \in \mathbb{N}$, y entonces $g(m)$ sería $f_m(m)$, y no $f_m(m) + 1$ como indica la propia definición de g. En consecuencia, g es total y computable, pero no recursiva primitiva.

Este teorema indica que la figura 1 se puede refinar para obtener la figura 2. En la siguiente sección examinaremos la parte de la figura 2 que cae fuera de las funciones computables totales.

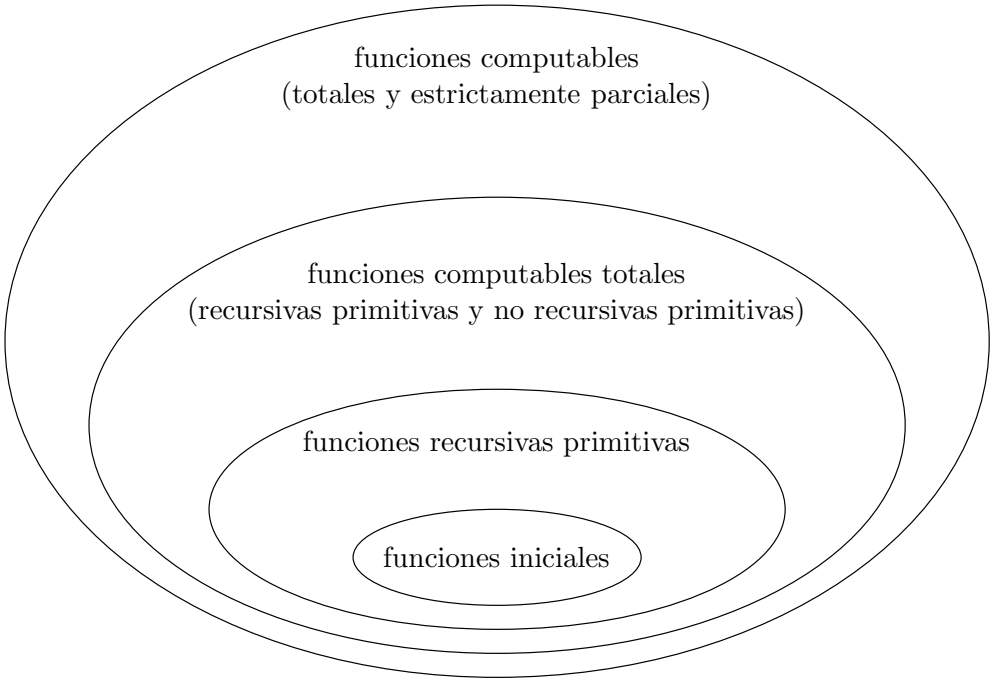


Figura 2: Revisión de la jerarquía provisional mostrada en la figura 1

3 Funciones recursivas parciales

Nuestro estudio nos ha llevado hasta ahora a las clases de las funciones iniciales, las funciones recursivas primitivas y las funciones totales computables, todas las cuales son funciones totales. Ahora ampliaremos nuestro estudio para incluir las funciones computables estrictamente parciales.

Definición de las funciones recursivas parciales. Para ampliar nuestro estudio de las funciones computables más allá de las funciones totales computables, aplicamos la técnica de construcción conocida como *minimalización*. Esta técnica nos permite construir una función

$f : \mathbb{N}^n \to \mathbb{N}$ a partir de otra función $g : \mathbb{N}^{n+1} \to \mathbb{N}$ declarando a $f(\bar{x})$ como la menor y en $\mathbb{N}$ tal que $g(\bar{x}, y) = 0$ y $g(\bar{x}, z)$ está definida para todos los enteros no negativos z menores que y. Esta construcción se representa con la notación:

$$f(\bar{x}) = \mu y[g(\bar{x}, y) = 0]$$

Como ejemplo, supongamos que $g(x, y)$ se define de acuerdo con los valores que aparecen en la tabla 1, y que $f(x)$ se define como $\mu y[g(x, y) = 0]$. Entonces, $f(0) = 4$, $f(1) = 2$, y $f(2)$ no está definido ya que, aunque 4 es el menor valor de y para el cual $g(2, y) = 0$, $g(2, z)$ no está definido para todos y cada uno de los valores de z menores que 4. En lo que se refiere al valor de $f(3)$, la tabla no nos proporciona suficiente información para determinar si se encuentra definido o no.

$g(0,0) = 2$	$g(1,0) = 3$	$g(2,0) = 8$	$g(3,0) = 2$	...
$g(0,1) = 3$	$g(1,1) = 4$	$g(2,1) = 3$	$g(3,1) = 6$	...
$g(0,2) = 1$	$g(1,2) = 0$	$g(2,2) =$ no definido	$g(3,2) = 7$	...
$g(0,3) = 5$	$g(1,3) = 2$	$g(2,3) = 6$	$g(3,3) = 2$	...
$g(0,4) = 0$	$g(1,4) = 0$	$g(2,4) = 0$	$g(3,4) = 8$	...
$g(0,5) = 1$	$g(1,5) = 0$	$g(2,5) = 1$	$g(3,5) = 4$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Tabla 1: Algunos valores de una cierta función $g(x, y)$

Es importante recalcar que, como sucede en el ejemplo anterior, la minimalización puede producir funciones que no están definidas para ciertas entradas. Otro ejemplo es la función $f : \mathbb{N} \to \mathbb{N}$ definida por $f(x) = \mu y[suma(x, y) = 0]$. En este caso, $f(0) = 0$, pero no está definida para las demás entradas ya que, para $x > 0$, no existe ningún y en $\mathbb{N}$ tal que $x + y = 0$.

Otro ejemplo que ya conocemos es la función $división : \mathbb{N}^2 \to \mathbb{N}$ definida por:

$$división(x, y) = \begin{cases} \text{la parte entera de } x/y \text{ si } y \neq 0 \\ \text{no definida si } y = 0 \end{cases}$$

que con la minimalización puede construirse como:

$$división(x, y) = \mu t[(x + 1) \dot{-} (producto(t, y) + y) = 0]$$

Por otra parte, en algunos casos la minimalización produce funciones totales, como por ejemplo $f(x) = \mu y[monus(x, y) = 0]$, que no es otra cosa que la *función identidad*: el menor y tal que $x \dot{-} y = 0$ es el propio x.

Ahora consideraremos la cuestión de si una función definida mediante la técnica de minimalización es o no computable. Si la función parcial g es computable, entonces se puede calcular $f(\bar{x}) = \mu y[g(\bar{x}, y) = 0]$ calculando los valores de $g(\bar{x}, 0)$, $g(\bar{x}, 1)$, $g(\bar{x}, 2)$, etc., hasta obtener el valor 0 o hasta llegar a un valor z para el cual $g(\bar{x}, z)$ no esté definido. En el primer caso, el valor de $f(\bar{x})$ es el valor de y para el cual se encontró que $g(\bar{x}, y)$ era 0. En el segundo caso, $f(\bar{x})$ no está definido. Por tanto, el proceso de minimalización aplicado a una función computable produce otra función computable. Y además, aplicado a una función computable estrictamente parcial produce otra función computable también estrictamente parcial.

La jerarquía de las clases de funciones. La anterior observación nos permite entonces rebautizar la clase de las funciones computables mostrada en la figura 2. Esta clase será denominada la clase de las *funciones μ-recursivas* o *funciones recursivas parciales*, tal y como

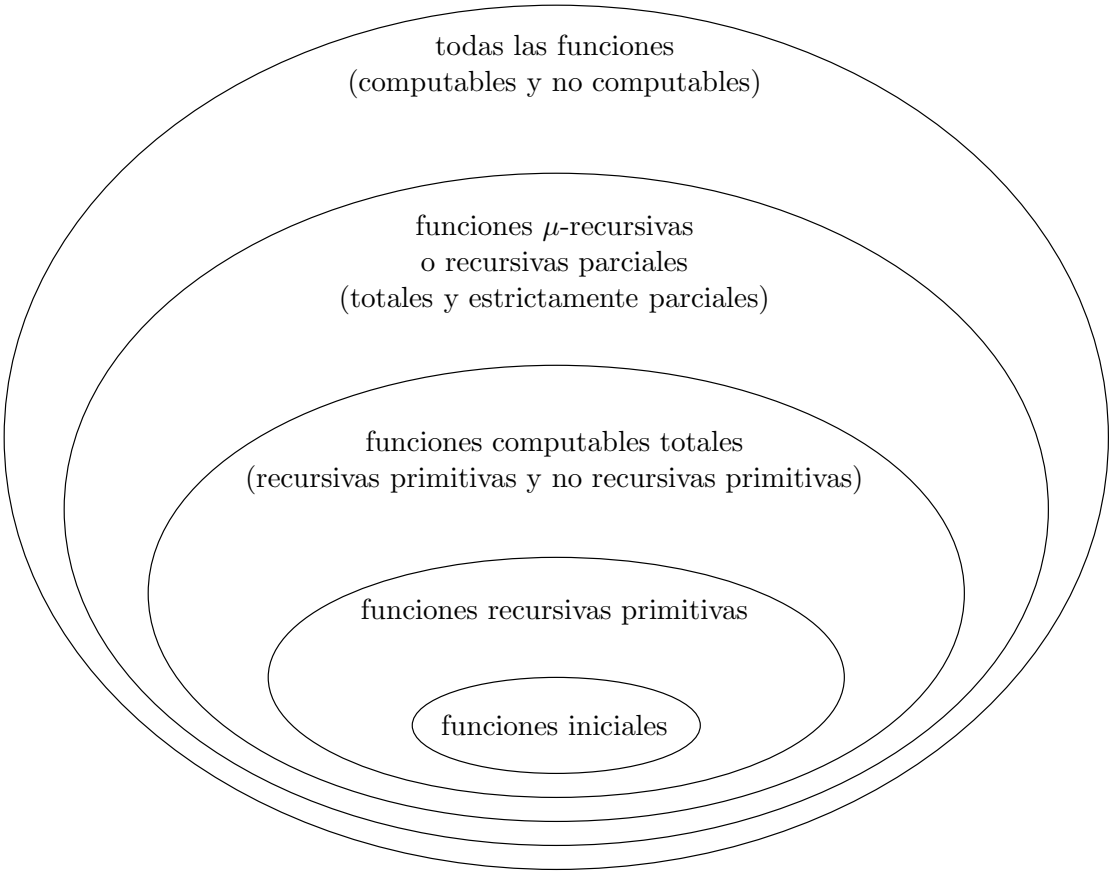


Figura 3: Jerarquía de las clases de funciones según la teoría de funciones recursivas

vemos en la figura 3, la cual muestra ya la jerarquía completa de las clases de funciones según la teoría de funciones recursivas.

Para ser más precisos, diremos entonces que la clase de funciones recursivas parciales es la clase de las funciones que pueden construirse a partir de las funciones iniciales aplicando un número finito de combinaciones, composiciones, recursividades primitivas y minimalizaciones.

Una vez más subrayamos que el término parcial no quiere decir que todas las funciones de la clase de las funciones recursivas parciales sean estrictamente parciales: hemos visto que el propio proceso de minimalización puede a veces producir funciones totales y, de hecho, la clase de las funciones recursivas parciales incluye a la clase de todas las funciones computables totales. Eso sí, en este contexto, funciones tales como la función *división*, por ser estrictamente parciales, están dentro de la clase de las funciones recursivas parciales, pero fuera de clase la de las funciones computables totales.

Equivalencia de las tesis de Turing y de Church. Con la presentación de las funciones μ-recursivas o recursivas parciales llegamos al final de nuestra jerarquía de funciones computables. En este punto, comentaremos que, al igual que la *tesis de Turing* propone que la clase de las máquinas de Turing posee el poder computacional de cualquier sistema de computación, se plantea también la hipótesis de que la clase de las funciones recursivas parciales contiene todas las funciones computables. Este último argumento se conoce como la *tesis de Church*, aunque Church la propuso originalmente en un contexto algo distinto.

Un hecho que apoya la tesis de Church es que nadie ha podido demostrar que sea falsa, es

decir, nadie ha encontrado una función parcial que sea computable pero no recursiva parcial. No obstante, un razonamiento aún más convincente es que la tesis de Church y la tesis de Turing representan lo mismo, tal y como enuncian los dos siguientes teoremas:

> Toda función recursiva parcial es computable por una máquina de Turing.
>
> Todo proceso computacional realizado por una máquina de Turing es en realidad el cálculo de una función recursiva parcial.

En resumen, hemos visto que las funciones μ-recursivas o recursivas parciales son computables por máquinas de Turing y que cualquier máquina de Turing no hace más que calcular una función recursiva parcial. Esto significa que las tesis de Turing y de Church son equivalentes aunque se muestren en contextos distintos. Con frecuencia, el concepto común que subyace en ambas se conoce con el nombre de *tesis de Church-Turing* en lugar de tesis de Church o tesis de Turing. Sin embargo, la equivalencia de estas hipótesis es más importante que la mezcla de la terminología, pues implica que hemos llegado al mismo límite aparente para el poder de los procesos computacionales siguiendo dos enfoques distintos, uno operacional y otro funcional, lo cual refuerza nuestra confianza en estas conjeturas.

4 El poder de los lenguajes de programación

Como aplicación práctica de la teoría presentada en las secciones anteriores, regresamos a nuestra pregunta con respecto al poder expresivo de los lenguajes de programación. Lo que nos concierne ahora será la cuestión de qué aspectos deben incluirse en un lenguaje de programación, para garantizar que una vez diseñado e implementado no descubramos que existen problemas computables cuyas soluciones no pueden especificarse con dicho lenguaje.

Nuestra estrategia consistirá en desarrollar un lenguaje de programación esencial, con el cual se pueda expresar un programa para calcular cualquier función recursiva parcial. Esto asegurará (suponiendo que la tesis de Church-Turing es verdadera) que, mientras un lenguaje de programación cuente con las características de nuestro lenguaje esencial, permitirá expresar una solución para cualquier problema que pueda resolverse de manera algorítmica.

Un lenguaje de programación esencial. Dado que nuestro lenguaje de programación esencial se usará para calcular funciones recursivas parciales, el único tipo de dato que se requiere es el entero no negativo (como ya hemos señalado, en un ordenador digital moderno todo dato se representa como un entero no negativo, aunque los lenguajes de alto nivel puedan disfrazar esta realidad). Por esta razón, nuestro lenguaje de programación esencial no necesita enunciados de declaración de tipo, sino que los identificadores se declaran automáticamente como de tipo entero no negativo con sólo aparecer una vez en un programa.

Nuestro lenguaje contendrá los dos enunciados de asignación siguientes:

```
incr nombre;          y          decr nombre;
```

El primero incrementa en uno el valor asignado al identificador `nombre`, mientras que el segundo lo decrementa en uno, a menos que el valor a decrementar sea 0, en cuyo caso permanece con dicho valor.

El lenguaje se completa entonces con una estructura de control formada por los dos siguientes enunciados:

```
while nombre ≠ 0 do
     ⋮
end;
```

Esta estructura indica que es necesario repetir los enunciados que se encuentren entre `while` y `end` mientras el valor asignado al identificador `nombre` no sea cero.

Este es entonces nuestro lenguaje de programación esencial. Efectivamente, es tan sencillo que nuestro primer objetivo será el de incorporar algunos enunciados más potentes, pero, eso sí, que puedan simularse con secuencias de enunciados esenciales. Una vez hecho esto, podremos utilizar los enunciados adicionales para escribir programas más claros. Más concretamente, adoptaremos la sintaxis:

```
clear nombre;
```

como versión abreviada de la secuencia:

```
while nombre ≠ 0 do
     decr nombre;
end;
```

cuyo efecto es asignar el valor 0 al identificador `nombre`. Además, utilizaremos

```
nombre1 ← nombre2;
```

para representar el siguiente segmento de programa:

```
clear aux;
clear nombre1;
while nombre2 ≠ 0 do
     incr aux;
     decr nombre2;
end;
while aux ≠ 0 do
     incr nombre1;
     incr nombre2;
     decr aux;
end;
```

el cual asigna el valor de `nombre2` al identificador `nombre1`, asignándolo primero a la variable auxiliar `aux`, y después reasignándolo tanto a `nombre1` como a `nombre2`. Obsérvese que el esfuerzo que implica el uso de `aux` permite evitar el efecto colateral de destruir la asignación original de `nombre2`.

Recursiva parcial implica programable con lo esencial. La siguiente tarea será la de mostrar que para cualquier función recursiva parcial existe un algoritmo que calcula dicha función, el cual puede expresarse con nuestro sencillo lenguaje de programación esencial. Para ello, estableceremos la convención de que para calcular una función de $\mathbb{N}^m$ a $\mathbb{N}^n$ escribiremos un programa con los identificadores $x_1, x_2, \ldots, x_m$ para contener los valores de entrada, y con $z_1, z_2, \ldots, z_n$ para almacenar los valores de salida.

Con nuestro sencillo lenguaje esencial es fácil expresar los programas que corresponden al cálculo de las *funciones iniciales*:

- La función ζ o *función cero* está representada por:

$$\texttt{clear } z_1\texttt{;}$$

- La función σ o *función sucesor* se representa mediante:

$$z_1 \leftarrow x_1\texttt{;}$$
$$\texttt{incr } z_1\texttt{;}$$

- Y las *funciones de proyección* π_j^m pueden calcularse como sigue:

$$z_1 \leftarrow x_j\texttt{;}$$

Seguidamente centraremos nuestra atención sobre la técnica de construcción de los algoritmos correspondientes a las *funciones recursivas parciales*:

- *Combinación.* Si F y G son dos programas que calculan las funciones parciales $f : \mathbb{N}^k \to \mathbb{N}^m$ y $g : \mathbb{N}^k \to \mathbb{N}^n$, respectivamente, entonces la función $f \times g$ se puede calcular concatenando G al final del programa F, modificando F y G para que asignen sus salidas a los identificadores apropiados (F debe asignar su salida a $z_1, \ldots, z_m$ a la vez que G asigna su salida a $z_{m+1}, \ldots, z_{m+n}$) y ajustando F para que no destruya los valores de entrada antes de que G pueda utilizarlos.

  Por supuesto, tanto aquí como en los casos que se presentan más adelante, suponemos que los programas en cuestión no tienen nombres de variables auxiliares comunes que pudieran ocasionar efectos colaterales. En este caso, si los tuvieran, siempre podríamos cambiarlos para obtener espacios de nombres distintos, por ejemplo, haciendo que todos los identificadores del programa F estuvieran precedidos por la letra `F` y todos los del programa G por la letra `G`.

- *Composición.* Si F y G son dos programas que calculan las funciones parciales $f : \mathbb{N}^k \to \mathbb{N}^n$ y $g : \mathbb{N}^n \to \mathbb{N}^m$, respectivamente, entonces la función $g \circ f$ se puede calcular concatenando G al final del programa F y ajustando los identificadores de salida de F para que vayan de acuerdo con los identificadores de entrada de G.

- *Recursividad primitiva.* Ahora supongamos que el programa G calcula la función parcial $g : \mathbb{N}^k \to \mathbb{N}^m$, el programa H calcula $h : \mathbb{N}^{k+m+1} \to \mathbb{N}^m$ y, usando recursividad primitiva, la función $f : \mathbb{N}^{k+1} \to \mathbb{N}^m$ se define como:

$$f(\bar{x}, 0) = g(\bar{x})$$
$$f(\bar{x}, y+1) = h(\bar{x}, y, f(\bar{x}, y))$$

  Entonces se puede calcular f con el siguiente programa:

$$G$$
$$\texttt{aux} \leftarrow x_{k+1}\texttt{;}$$
$$\texttt{clear } x_{k+1}\texttt{;}$$
$$\texttt{while aux} \neq \texttt{0 do}$$
$$\quad x_{k+2} \leftarrow z_1\texttt{;}$$
$$\quad x_{k+3} \leftarrow z_2\texttt{;}$$
$$\quad \vdots$$
$$\quad x_{k+m+1} \leftarrow z_m\texttt{;}$$
$$\quad H$$
$$\quad \texttt{incr } x_{k+1}\texttt{;}$$
$$\quad \texttt{decr aux;}$$
$$\texttt{end;}$$

donde, sin pérdida de generalidad, podemos suponer que G y H no tienen efectos secundarios indeseables.

- *Minimalización.* Si G es un programa que calcula la función parcial $g : \mathbb{N}^{n+1} \to \mathbb{N}$, entonces podemos producir un programa que calcule $\mu y[g(\bar{x}, y) = 0]$ como sigue:

```
clear x_{n+1};
G
while z_1 ≠ 0 do
      incr x_{n+1};
      G
end;
z_1 ← x_{n+1};
```

  Este programa lleva a cabo dicha tarea calculando $g(\bar{x}, 0), g(\bar{x}, 1), g(\bar{x}, 2), \ldots$, hasta que g produce una salida cero para un cierto valor y, almacenado realmente en x_{n+1}, el cual constituirá el valor final de salida del programa. Obsérvese que G se puede diseñar para que no altere las asignaciones originales de sus variables de entrada.

Hasta aquí hemos mostrado entonces que cualquier función recursiva parcial puede calcularse por medio de un programa escrito en nuestro lenguaje de programación esencial. Por tanto, de acuerdo con la tesis de Church-Turing, sabemos que cualquier lenguaje que proporcione el tipo entero no negativo y la capacidad para incrementar un valor, decrementarlo y ejecutar un ciclo `while`, tendrá el poder expresivo suficiente para plantear la solución de cualquier problema que tenga una solución algorítmica. Todas las características adicionales de un lenguaje representan simplemente conveniencias que facilitan las labores de programación.

Programable con lo esencial implica recursiva parcial. Después de haber descubierto el sorprendente poder de nuestro sencillo lenguaje, se podría pensar que ofrece un medio para calcular algo más que las funciones recursivas parciales. Por supuesto, si esta conjetura fuera verdadera, contradiría la tesis de Church-Turing ya que obtendríamos un método para calcular una clase de funciones mayor que la de las funciones recursivas parciales (que son las computables por máquinas de Turing). Por tanto, efectivamente, no constituye ninguna sorpresa el encontrar que cualquier cálculo expresado en nuestro sencillo lenguaje de programación puede modelarse por medio de una función recursiva parcial. La estrategia para demostrar esta afirmación consiste en aplicar razonamientos inductivos sobre el posible número de enunciados de un programa.

Para concluir, señalamos que el estudio de la computabilidad por medio de los lenguajes de programación es otro área de investigación cuyos resultados también apoyan la tesis de Church-Turing. De hecho, no se ha diseñado todavía ningún lenguaje de programación que tenga mayor poder computacional que el sencillo lenguaje presentado aquí, aunque, como ya hemos dicho, los lenguajes de programación mucho más elaborados que se utilizan en la actualidad son obviamente muy superiores en cuanto a legibilidad.

5 Comentarios finales

Reconsiderando de nuevo todo el estudio presentado, vemos que el problema de la computabilidad ha sido atacado desde varias direcciones: máquinas computacionales, gramáticas generativas, teoría de funciones recursivas y lenguajes de programación. En cada caso, hemos descubierto un límite aparente para las capacidades computacionales de cada enfoque, y hemos mostrado que

estos límites coinciden unos con otros. Los lenguajes estructurados por frases son iguales que los lenguajes aceptados por máquinas de Turing, las funciones computables con máquinas de Turing son iguales que las funciones recursivas parciales, y las funciones recursivas parciales son iguales que las funciones computables con el lenguaje de programación esencial.

Por lo tanto, parece que hemos identificado los límites de los procesos computacionales y, específicamente, los límites de los ordenadores. Es decir, hemos encontrado un firme apoyo para la tesis de Church-Turing: si una máquina de Turing no puede resolver un problema, entonces ningún ordenador podrá hacerlo, ya que simplemente no existe un algoritmo para obtener su solución. En otras palabras, las limitaciones que hemos detectado corresponden a los procesos computacionales, no a la tecnología.