UNIDAD 10: Paradigma de programación funcional

Introducción

El paradigma de programación funcional se basa en funciones matemáticas.
En los programas escritos en lenguaje imperativo, una de las características fundamentales es el estado que tienen, lo que cambia a lo largo de la ejecución. Este estado está representado por las variables del programa. El autor y todos los lectores del programa deben entender los usos de sus variables y cómo el estado del programa cambia a través de la ejecución. Para un programa amplio, esto es una tarea desalentadora. Este es un problema con los programas escritos en un lenguaje imperativo que no está presente en un programa escrito en un lenguaje funcional puro, ya que no tienen ni variables ni estado.

LISP comenzó como un lenguaje funcional puro, pero pronto adquirió cierta importancia con características imperativas con el fin de aumentar su eficiencia en la ejecución. Es aún el más importante de los lenguajes funcionales, al menos en el sentido de que es el único que ha alcanzado un uso generalizado.
Domina en las áreas de la representación del conocimiento, aprendizaje de las máquinas, los sistemas de formación inteligentes, y el modelado de habla. Common Lisp es una combinación de varios principios de LISP de 1980.
Scheme es un pequeño dialecto de LISP. Scheme ha sido ampliamente utilizado para enseñar programación funcional. También se utiliza en algunas universidades para dictar cursos de introducción a la programación.

Funciones matemáticas

Una función matemática es un mapeo de los miembros de un conjunto (llamado el *dominio* del grupo), a otro conjunto (llamado *rango* del grupo). Una definición de función especifica el dominio y rango, ya sea explícita o implícitamente, junto con el mapeo.
El mapeo se describe mediante una expresión.
Las funciones se aplican a menudo a un elemento particular del conjunto de dominio, dado como un parámetro a la función.
Una función devuelve un elemento del rango del grupo.

Una de las características fundamentales de las funciones matemáticas es que el orden de evaluación de sus expresiones de mapeo es controlada por la recursividad y expresiones condicionales, en lugar de por la secuenciación y la iteración que son comunes en los lenguajes de programación imperativos.

Otra característica importante de las funciones matemáticas es que no tienen efectos secundarios y no pueden depender de los valores externos, siempre asignan un elemento en particular del dominio para el mismo elemento del rango.

Sin embargo, un subprograma en un lenguaje imperativo puede depender de los valores de varias variables no locales o globales. Esto hace que sea difícil determinar estáticamente lo que el subprograma producirá y qué efectos secundarios tendrá en una ejecución en particular.

En matemáticas, no existe el concepto del estado de una función.

Una función matemática mapea su parámetro (s) a un valor (o valores), en vez de especificar una secuencia de operaciones en los valores en la memoria para producir un valor.

Funciones simples

Las definiciones de funciones se suelen escribir con un nombre de función, seguido de una lista de parámetros entre paréntesis, seguido de la expresión de mapeo. Por ejemplo:

cubo (x) ≡ x * x * x, donde x es un número real.

En esta definición, el dominio y el rango son números reales. El símbolo ≡ significa "se define como". El parámetro x puede representar cualquier miembro del conjunto de dominio, pero se fija para representar un elemento específico durante la evaluación de la función. Esta es una diferencia que existe entre los parámetros de funciones matemáticas y las variables en los lenguajes imperativos.

Las aplicaciones de funciones se especifican con el nombre de la función y un elemento particular del conjunto de dominio. El elemento de rango se obtiene de la evaluación de la expresión de la función de mapeo con el elemento de dominio sustituido para las ocurrencias del parámetro.

El valor de x se fija durante la ejecución de la función.

Notación Lambda

Proporciona un método para definir funciones sin nombre. La expresión lambda especifica los parámetros y la expresión de asociación. La expresión lambda es la misma función, que no tiene nombre. Por ejemplo, veamos la siguiente expresión lambda:

λ(x) x * x * x

Como se dijo anteriormente, un parámetro representa cualquier miembro del conjunto de dominio, pero durante la evaluación se enlaza a un miembro en particular.

Cuando se evalúa una expresión lambda para un parámetro dado, la expresión se dice que es aplicada a ese parámetro. La mecánica de dicha solicitud son los mismos que para cualquier evaluación de la función. Por ejemplo:

(λ (x) x * x * x) (2)

lo que resulta en el valor 8.

Las expresiones lambda, al igual que otras definiciones de funciones, pueden tener más de un parámetro.

Forma funcional

Una función de orden superior, o de forma funcional, es aquella que tiene una o más funciones como parámetros o produce una función como resultado, o ambas.

Un tipo común de forma funcional es la composición de funciones, que tiene dos funciones como parámetros y produce una función cuyo valor es el primer parámetro función aplicado al resultado del segundo.

La **composición de funciones** se escribe como una expresión, usando ° como un operador:

$$h \equiv f \circ g$$

Por ejemplo, si:

$f(x) \equiv x + 2$

$g(x) \equiv 3 * x$

entonces h es definida así:

$$h(x) \equiv f(g(x)), \text{ o } h(x) \equiv (3 * x) + 2$$

Aplicar a todo es una forma funcional que tiene una única función como parámetro. Si se aplica a una lista de parámetros, *aplicar a todo* aplica su parámetro funcional a cada uno de los valores en la lista y devuelve los resultados en una lista o secuencia. Aplicar-a-todo se denota por α.

Veamos el siguiente ejemplo:

$$h(x) \equiv x * x$$

Luego:

$$\alpha\ (h\ (2, 3, 4))\ \text{genera}\ (4, 9, 16)$$

Fundamentos de los lenguajes de programación funcionales

El objetivo del diseño de un lenguaje de programación funcional es imitar las funciones matemáticas en la mayor medida posible. Esto resulta en un enfoque para la resolución de problemas que es fundamentalmente diferente de los enfoques que se utilizan en los lenguajes imperativos. En un lenguaje imperativo, una expresión es evaluada y el resultado se almacena en una ubicación de memoria, que se representa como una variable en un programa. Este es el propósito de las sentencias de asignación.

Esta utilización necesaria de las celdas de memoria, cuyos valores representan el estado del programa, se traduce en una metodología de programación relativamente de bajo nivel.

Un programa en un lenguaje ensamblador, a menudo también debe almacenar los resultados de evaluaciones parciales de expresiones. Por ejemplo, para evaluar:

$$(x + y) / (a - b)$$

el valor de $(x + y)$ se calcula en primer lugar. Ese valor debe entonces ser almacenado, mientras se evalúa (a - b). El compilador maneja el almacenamiento de resultados intermedios de las evaluaciones de expresiones en lenguajes de alto nivel. El almacenamiento de los resultados intermedios sigue siendo necesario, pero los detalles se ocultan al programador.

Un lenguaje de programación puramente funcional no utiliza las variables o sentencias de asignación, liberando así al programador de las preocupaciones relacionadas con las celdas de memoria, o estado del programa. Sin variables, las estructuras iterativas no son posibles, porque ellas son controladas por variables. La repetición debe ser especificada con recursión en lugar de con iteración.
Los programas son definiciones de funciones y especificaciones de la aplicación de las funciones, y las ejecuciones consisten en la evaluación de las aplicaciones de funciones. Sin variables, la ejecución de un programa puramente funcional no tiene un estado en el sentido de semántica operacional y denotacional. La ejecución de una función siempre produce el mismo resultado cuando toma los mismos parámetros. Esta característica se denomina **transparencia referencial**. Eso hace que la semántica de los lenguajes puramente funcionales sea mucho más simple que la semántica de los lenguajes imperativos (y los lenguajes funcionales que incluyen características imperativas). También hace la prueba más fácil, ya que cada función puede ser probada por separado, sin interesar el contexto en que se encuentra.

Un lenguaje funcional proporciona un conjunto de funciones primitivas, un conjunto de formas funcionales para construir funciones complejas a partir de esas funciones primitivas, una operación de aplicación de la función, y alguna estructura o estructuras de representación de datos. Estas estructuras se utilizan para representar los parámetros y valores calculados por funciones. Si un lenguaje funcional está bien definido, requiere de un número, relativamente pequeño, de funciones primitivas.
El primer lenguaje de programación funcional, LISP, utiliza una forma sintáctica, tanto para los datos y el código, que es muy diferente a la de los lenguajes imperativos.
Aunque hay algunos lenguajes puramente funcionales, por ejemplo, Haskell, la mayoría de los lenguajes que se llaman funcionales, incluyen algunas características imperativas, por ejemplo, las variables mutables y construcciones que actúan como instrucciones de asignación.

El primer lenguaje de programación funcional: LISP

Tipos de datos y estructuras

Sólo había dos categorías de datos en el LISP original: átomos y listas.

Los elementos de una lista son pares, donde la primera parte son los datos del elemento, que es un puntero a un átomo o a una lista anidada.

La segunda parte del par puede ser un puntero a un átomo, un puntero a otro elemento, o la lista vacía.

Los elementos están enlazados entre sí en las listas mediante las segundas partes. Los átomos y las listas no son tipos en el sentido de que los lenguajes imperativos tienen tipos. De hecho, el LISP original era un lenguaje sin tipo. Los átomos son o símbolos, en la forma de identificadores, o literales numéricos.

Las listas en LISP se especifican a través de la delimitación de sus elementos con paréntesis.

Los elementos de las listas simples se limitan a átomos, como en:

(A B C D)

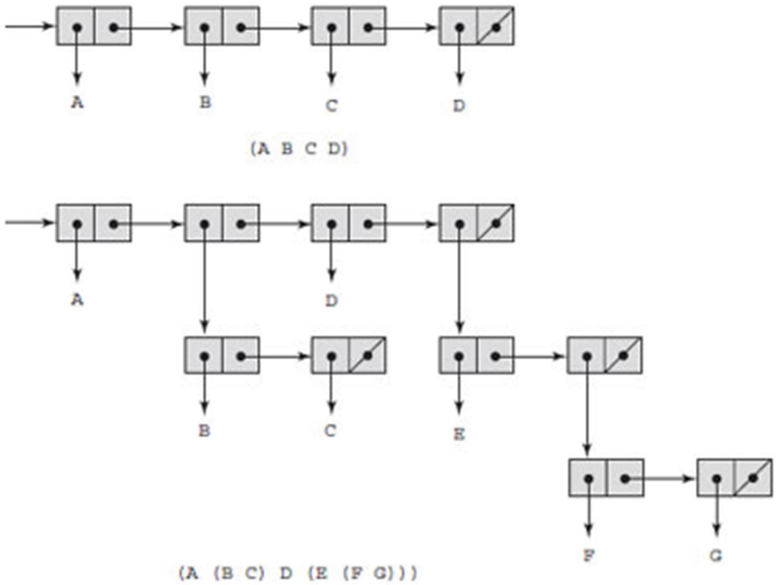
Las listas anidadas también se especifican entre paréntesis. Por ejemplo, la lista:

(A (B C) D (E (F G)))

es una lista de cuatro elementos. El primero es el átomo A, el segundo es la sublista (B C), el tercero es el átomo D y el cuarto es la sublista (E (F G)), que tiene como segundo elemento la sublista (F G).

Internamente, una lista por lo general se almacena en forma de estructura de lista enlazada en la que cada nodo tiene dos punteros, uno para hacer referencia a los datos del nodo y el otro para formar la lista enlazada. Una lista es referenciada por un puntero a su primer elemento.

La siguiente figura muestra las representaciones internas de las dos listas del ejemplo:



Los elementos de una lista se muestran horizontalmente. El último elemento de una lista no tiene sucesor, por lo que su relación es nula. Las sublistas se muestran con la misma estructura.

El primer intérprete LISP

Uno de los requisitos comunes del estudio de la computación es que uno debe ser capaz de demostrar ciertas características de computabilidad.

A partir de este concepto surgió la idea de construir una función LISP universal que pudiera evaluar cualquier otra función en LISP.

El primer requisito para la función LISP universal fue una notación que permita expresar funciones en la misma forma en que se expresan los datos.

La notación de lista entre paréntesis ya había sido adoptada para los datos en LISP, por lo que se decidió inventar convenciones para la definición de función y llamadas a funciones que también podrían ser expresadas en notación lista.

Las llamadas a funciones se especifican en una forma de lista con prefijos como se hace a continuación:

(nombre_función argumento1 … argumentoN)

Por ejemplo, si + es una función que toma dos o más parámetros numéricos, las siguientes dos expresiones dan como resultado 12 y 20, respectivamente:

(+ 5 7)

(+ 3 4 7 6)

La función EVAL es una función universal que podría evaluar cualquier otra función. Dos de las personas que estaban desarrollando LISP, Stephen B. Russell y Daniel J. Edwards, notaron que una implementación de EVAL podría servir como un intérprete de LISP.

Scheme

El lenguaje Scheme, que es un dialecto de LISP, fue desarrollado a mediados de la década de 1970. Se caracteriza por su pequeño tamaño y su uso exclusivo del alcance estático.
Como Scheme es un lenguaje esencialmente pequeño con una sintaxis y una semántica muy simples, es muy adecuado para aplicaciones educativas.

Funciones numéricas primitivas

Scheme incluye funciones primitivas para las operaciones aritméticas básicas. Estas son: +, -, * y /, para sumar, restar, multiplicar y dividir.

* y + pueden tener cero o más parámetros.

Si no tienen ningún parámetro:

* devuelve 1;

+ devuelve 0.

+ suma todos sus parámetros juntos. * multiplica todos sus parámetros juntos. / y - pueden tener dos o más parámetros. En el caso de la resta, todos menos el primer parámetro se restan al primero. La división es similar a la resta.

Algunos ejemplos son:

Expresión	Valor
42	42
(* 3 7)	21
(+ 5 7 8)	20
(− 5 6)	-1
(− 15 7 2)	6
(− 24 (* 4 3))	12

Hay un gran número de otras funciones numéricas en Scheme, como MODULO, ROUND, MAX, MIN, LOG, SIN y SQRT. SQRT devuelve la raíz cuadrada de su parámetro numérico, si el valor del parámetro no es negativo. Si el parámetro es negativo, SQRT produce un número complejo.

En Scheme, se utilizan las letras mayúsculas para todas las palabras reservadas y funciones predefinidas. La definición oficial del lenguaje especifica que es case sensitive, o sea que distingue entre mayúsculas y minúsculas. Sin embargo, algunas implementaciones, por ejemplo, el DrRacket, requieren minúsculas para las palabras reservadas y funciones predefinidas.

Si una función tiene un número fijo de parámetros, tales como SQRT, el número de los parámetros de la llamada debe coincidir con ese número. Si no, el intérprete producirá un mensaje de error.

Definición de funciones

Un programa en Scheme es un conjunto de definición de funciones. En consecuencia, es necesario saber cómo definir estas funciones para poder escribir un programa más simple.

En Scheme, una función sin nombre, incluye la palabra LAMBDA, y se denomina expresión Lambda. Por ejemplo:

(LAMBDA (x) (* x x))

es una función sin nombre que devuelve el cuadrado de su parámetro numérico dado.
Esta función se puede aplicar de la misma manera que las funciones con nombre: colocándola en el principio de una lista que contiene los parámetros actuales. Por ejemplo, la siguiente expresión devuelve 49:

((LAMBDA (x) (* x x)) 7)

En esta expresión, x es denominada **variable enlazada** dentro de la expresión lambda.
Durante la evaluación de esta expresión, x será 7. Una variable enlazada nunca cambia en la expresión después de haber sido unida a un valor de parámetro actual.
Las expresiones lambda pueden tener cualquier número de parámetros. Por ejemplo, podría tener:

(LAMBDA (a b c x) (+ (* a x x) (* b x) c))

La función DEFINE sirve para dos necesidades fundamentales: para enlazar un nombre a un valor y para enlazar un nombre a una expresión lambda.
DEFINE se invoca de una forma especial porque es interpretado (por EVAL) de una manera diferente que las primitivas normales como las funciones aritméticas.
La forma más simple de una función DEFINE se utiliza para enlazar un nombre al valor de una expresión. Esta forma es:

(DEFINE símbolo expresión)

Por ejemplo:
(DEFINE pii 3.14159)
(DEFINE dos_pi (* 2 pii))

Si estas dos expresiones se han introducido al intérprete de Scheme y luego se escribe pii, se mostrará el número 3,14159. Cuando se tipea dos_pi, se mostrará

6,28318. En ambos casos, los números mostrados pueden tener más dígitos que los que se muestran aquí.
Esta forma de utilizar DEFINE es similar a una declaración de una constante en un lenguaje imperativo. Por ejemplo, en Java, los equivalentes a los anteriores nombres definidos son los siguientes:

```
final float PI = 3.14159;
final float TWO_PI = 2.0 * PI;
```

Los nombres en Scheme pueden contener letras, dígitos y caracteres especiales excepto paréntesis, son sensibles a mayúsculas y no deben empezar con un dígito.

El segundo uso de la función DEFINE es vincular una expresión lambda a un nombre.
Para enlazar un nombre a una expresión lambda, DEFINE toma dos listas como parámetros. El primer parámetro es el prototipo de una llamada a la función, con el nombre de la función seguido de los parámetros formales, juntos en una lista. La segunda lista contiene una expresión que lleva el nombre que se va a enlazar.

La forma general de DEFINE es:

```
(DEFINE (nombreFuncion parámetros)
        (expresión)
)
```

Por ejemplo, aquí DEFINE une el nombre *cuadrado* a una expresión funcional que toma un parámetro:

```
(DEFINE (cuadrado nro) (* nro nro))
```

Después de que el intérprete evalúa esta función, se puede utilizar, como:

```
(cuadrado 5)
```

que devuelve 25.

A continuación se presenta otro ejemplo de una función. Se calcula la longitud de la hipotenusa de un triángulo rectángulo, dadas las longitudes de los catetos.

```
(DEFINE (hipotenusa cat1 cat2)
        (SQRT (+ (cuadrado cat1) (cuadrado cat2)))
)
```

Tenga en cuenta que la hipotenusa utiliza la función *cuadrado*, que se definió anteriormente.

Funciones predicado numéricas

Una función de predicado es aquella que devuelve un valor booleano. Scheme incluye una colección de funciones predicados para datos numéricos. Entre ellas se encuentran las siguientes:

Función	**Significado**
=	Igual
<>	Distinto
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
EVEN?	¿Es un nro. par?
ODD?	¿Es un nro. impar?
ZERO?	¿Es cero?

Observe que los nombres de todas las funciones de predicados predefinidas que tienen palabras para los nombres terminan con signos de interrogación. En Scheme, los dos valores booleanos son #T y #F (o #t y #f), aunque algunas implementaciones utilizan la lista vacía para false.

Cuando una lista se interpreta como un valor booleano, cualquier lista no vacía se evalúa como verdadera, la lista vacía como falsa. Esto es similar a la interpretación de enteros en C como valores booleanos; evaluando el cero como resultado falso y cualquier valor distinto de cero como verdadero.

Control de Flujo

Scheme utiliza tres construcciones diferentes para el control de flujo:

- una similar al constructor de selección de los lenguajes imperativos y
- dos en función de la evaluación utilizada en funciones matemáticas.

La función del selector de dos caminos, llamada IF, tiene tres parámetros:

- una expresión de predicado,
- una expresión *then* y
- una expresión *else*.

Una llamada a IF tiene la forma:

(IF predicado expression_then expression_else)

Por ejemplo:

```
(DEFINE (factorial n)
        (IF (<= n 1)
            1
            (* n (factorial (− n 1)))
))
```

Recordemos que la selección múltiple de Scheme, COND, la estudiamos en la Unidad 6.

La tercera estructura de control de Scheme es la recursión, que se utiliza, como en las matemáticas, para especificar la repetición.

Funciones de lista

Los programas escritos en Scheme son interpretados por la función EVAL. Cuando se aplica a una función primitiva, EVAL evalúa primero los parámetros de la función dada.

En algunas llamadas, sin embargo, los parámetros son elementos de datos en lugar de referencias a la función. Cuando un parámetro no es una referencia a una función, no debe ser evaluado.

Supongamos que tenemos una función que tiene dos parámetros, un átomo y una lista, y el propósito de la función es la de determinar si el átomo dado se encuentra en la lista dada. Ni el átomo ni la lista deben ser evaluados, son datos literales a examinar. Para evitar la evaluación de un parámetro, éste se pasa como parámetro a la función primitiva QUOTE, que simplemente lo devuelve sin cambios. Los siguientes ejemplos ilustran el uso de QUOTE:

(QUOTE A) devuelve A
(QUOTE (A B C)) devuelve (A B C)

La abreviatura común de la llamada a QUOTE, se realiza precediendo a la expresión con un apóstrofe ('). Por lo tanto, en lugar de (QUOTE (A B)), se usará '(A B).

La necesidad de QUOTE surge debido a la naturaleza fundamental de Scheme (y los otros lenguajes basados en LISP): los datos y el código tienen el mismo formato.

Las funciones CAR, CDR, y CONS las estudiamos en la unidad 4. A continuación algunos ejemplos adicionales de las operaciones de CAR y CDR:

- (CAR '(A B C)) devuelve A
- (CAR '((A B) C D)) devuelve (A B)
- (CAR 'A) es un error porque A no es una lista
- (CAR '(A)) devuelve A

- (CAR '()) es un error
- (CDR '(A B C)) devuelve (B C)
- (CDR '((A B) C D)) devuelve (C D)
- (CDR 'A) es un error
- (CDR '(A)) devuelve ()
- (CDR '()) es un error

Veamos otro ejemplo de una función simple:

(DEFINE (second unaLista) (CAR (CDR unaLista)))

Una vez que se evalúa esta función, se puede utilizar, como en:

(second '(A B C))

que devuelve B.

Algunas de las composiciones funcionales más comúnmente utilizados en Scheme son construidas como funciones individuales. Por ejemplo:

- (CAAR x) es equivalente a (CAR (CAR x))
- (CADR x) es equivalente a (CAR (CDR x)), y
- (CADDAR x) es equivalente a (CAR (CDR (CDR (CAR x)))).

Cualquier combinación de letras A y D, hasta cuatro, son legales entre la "C" y la "R" en el nombre de la función.

Por ejemplo, considere la siguiente evaluación de CADDAR:

(CADDAR '((A B (C) D) E)) =
(CAR (CDR (CDR (CAR '((A B (C) D) E))))) =
(CAR (CDR (CDR '(A B (C) D)))) =
(CAR (CDR '(B (C) D))) =
(CAR '((C) D)) =
(C)

Los siguientes son ejemplos de llamadas a CONS:

(CONS 'A '()) devuelve (A)

(CONS 'A '(B C)) devuelve (A B C)

(CONS '() '(A B)) devuelve (() A B)

(CONS '(A B) '(C D)) devuelve ((A B) C D)

Nota que CONS es, en cierto sentido, la inversa de CAR y CDR. CAR y CDR tienen una lista aparte, y CONS construye una nueva lista a partir de las listas dadas.

Por lo tanto, si tenemos una lista listaA, a continuación:

(CONS (CAR listaA) (CDR listaA))

devuelve una lista con la misma estructura y mismos elementos que listaA.

LIST es una función que construye una lista a partir de un número variable de parámetros.

Es una versión abreviada de funciones CONS anidadas:

(LIST 'apple 'orange 'grape)

devuelve

(apple orange grape)

Usando CONS, la llamada anterior a LIST se escribiría de la siguiente manera:

(CONS 'apple (CONS 'orange (CONS 'grape '())))

Funciones de predicado para átomos y listas

Scheme tiene tres funciones de predicado fundamentales, EQ?, NULL?, y LIST?, para los átomos y las listas.

La función EQ? toma dos expresiones como parámetros, aunque por lo general se utiliza con dos parámetros de átomos simbólicos. Se devuelve #T si ambos parámetros tienen el mismo valor de puntero, es decir, apuntan al mismo átomo o lista; de otra manera, devuelve #F. Si los dos parámetros son átomos simbólicos, EQ? devuelve #T si son los mismos símbolos; de lo contrario devuelve #F.

Por ejemplo:

(EQ? 'A 'A) devuelve #T
(EQ? 'A 'B) devuelve #F
(EQ? 'A '(A B)) devuelve #F
(EQ? '(A B) '(A B)) devuelve #F o #T
(EQ? 3.4 (+ 3 0.4)) devuelve #F o #T

Como se ve en el cuarto ejemplo, el resultado de la comparación de las listas con EQ? no es consistente. La razón de esto es que dos listas que son exactamente iguales no son duplicadas en memoria. En el momento que Scheme crea una lista, verifica si ya hay una lista de este tipo. Si la hay, la nueva lista no es nada más que un puntero a la lista existente. En estos casos, las dos listas serán evaluadas como iguales por la función EQ?. Sin embargo, en algunos casos, puede ser difícil detectar la presencia de una lista idéntica, en cuyo caso se crea una nueva lista. En este caso, EQ? devuelve # F.

El último caso muestra que la suma puede producir un nuevo valor, en el cual no sería igual (con EQ?) a 3,4, o puede reconocer que ya tiene el valor 3,4 y usarlo, en cuyo caso EQ? utilizará el puntero a la antigua 3,4 y devolvería #T.

Como hemos visto, EQ? funciona para átomos simbólicos, pero no necesariamente para átomos numéricos. El = funciona para los átomos numéricos, pero no para los átomos simbólicos. Como se mostró previamente, EQ? tampoco funciona de forma fiable para parámetros de una lista.

A veces es conveniente poder probar la igualdad de dos átomos cuando no se sabe si son simbólicos o numéricos. Para este propósito, Scheme tiene un predicado diferente, EQV?, que funciona tanto para átomos numéricos como simbólicos. Veamos los siguientes ejemplos:

(EQV? 'A 'A) devuelve #T
(EQV? 'A 'B) devuelve #F
(EQV? 3 3) devuelve #T
(EQV? 'A 3) devuelve #F
(EQV? 3.4 (+ 3 0.4)) devuelve #T
(EQV? 3.0 3) devuelve #F

En el último ejemplo demuestra que los valores de punto flotante son diferentes de los valores enteros. EQV? no es una comparación de punteros, es una comparación de valor.
La razón principal para usar EQ? o = en lugar de EQV? cuando sea posible es que EQ? y = son más rápidos que EQV?.

La función predicado LIST? devuelve #T si su único argumento es una lista y #F de otro modo. Por ejemplo:

(LIST? '(X Y)) devuelve #T
(LIST? 'X) devuelve #F
(LIST? '()) devuelve #T

La función NULL? verifica su parámetro para determinar si es una lista vacía y devuelve #T si lo es. Por ejemplo:

(NULL? '(A B)) devuelve #F
(NULL? '()) devuelve #T
(NULL? 'A) devuelve #F
(NULL? '(())) devuelve #F

En el último ejemplo devuelve #F porque el parámetro no es la lista vacía, es una lista que contiene un solo elemento, una lista vacía.

Ejemplos de funciones en Scheme

Veamos el problema de la pertenencia de un átomo dado en una lista teniendo en cuenta que la misma no incluye sublistas. Dicha lista se denomina *lista simple*. Si la función es llamada *member*, podría ser utilizada de la siguiente manera:

(member 'B '(A B C)) devuelve #T
(member 'B '(A C D E)) devuelve #F

Pensando en términos de iteración, el problema de miembros es simplemente comparar el átomo dado y los elementos individuales de la lista dada, uno a la vez

en algún orden, hasta que se encuentre una coincidencia o no haya más elementos en la lista para comparar. Un proceso similar se puede lograr usando la recursión.

La función puede comparar el átomo dado con el CAR de la lista. Si coinciden, se devuelve el valor #T. Si no coinciden, el CAR de la lista debe ser ignorado y la búsqueda continúa en el CDR de la lista. Esto se puede hacer por que la función se llama a sí misma con el CDR de la lista como el parámetro de lista y devuelve el resultado de esta llamada recursiva. Este proceso terminará si el átomo dado se encuentra en la lista. Si el átomo no está en la lista, la función será eventualmente llamada (por sí misma) con una lista null como el parámetro actual. Allí devolvería #F.

En este proceso, hay dos maneras de salir de la recursión: o bien la lista es vacía en alguna llamada (en el que se devuelve el caso #F), o se encuentra una coincidencia y se devuelve #T.

En total, hay tres casos que deben ser manejados en la función: una lista de entrada vacía, una coincidencia entre el átomo y el CAR de la lista, o una falta de coincidencia entre el átomo y el CAR de la lista, lo que hace que la llamada sea recursiva. Estos tres son los tres parámetros a COND, siendo el último, el caso por defecto que se desencadena por un predicado ELSE.

La función completa sería:

```
(DEFINE (member atomo lista)
        (COND
                ((NULL? lista) #F)
                ((EQ? atomo (CAR lista)) #T)
                (ELSE (member atomo (CDR lista)))
))
```

Esta forma es típica de las funciones de procesamiento de listas en Scheme. En tales funciones, los elementos de las listas se procesan uno a la vez. El elemento

individual se obtiene con el CAR, y el proceso se continúa utilizando la recursividad en el CDR de la lista.
Se debe tener en cuenta que la prueba nula debe preceder a la prueba igual, debido a que la aplicación de CAR a una lista vacía da error.

Segundo ejemplo, consideremos el problema de determinar si dos listas dadas son iguales. Si las dos listas son simples, la solución es relativamente fácil. Una función de predicado, listasiguales, para comparar dos listas simples es ésta:

```
(DEFINE (listasiguales lista1 lista2)
        (COND
                ((NULL? lista1) (NULL? lista2))
                ((NULL? lista2) #F)
                ((EQ? (CAR lista1) (CAR lista2))
                        (listasiguales (CDR lista1) (CDR lista2)))
                (ELSE #F)
))
```

El primer caso, el cual es manejado por el primer parámetro a COND, es para cuando el primer parámetro es la lista vacía. Esto puede ocurrir en una llamada externa si el primer parámetro lista1, está vacía inicialmente. Debido a que una llamada recursiva utiliza las CDR de las dos listas de parámetros como sus parámetros, el primer parámetro lista puede estar vacía en una llamada de este tipo. Cuando la primera lista está vacía, debe verificarse el segundo parámetro para ver si también es una lista vacía. Si es así, son iguales (ya sea por primera vez o las CAR eran iguales en todas las llamadas recursivas anteriores), y NULL? devuelve #T correctamente. Si la segunda lista no está vacía, es mayor que la primera lista de parámetros y debe devolver #F.

El siguiente caso se refiere a cuando la segunda lista está vacía y la primera lista no. Esta situación se produce sólo cuando la primera lista es más larga que la segunda.

El tercer caso es el paso recursivo que pone a prueba la igualdad entre dos elementos correspondientes de las dos listas. Esto se hace mediante la comparación de las CAR de las dos listas no vacías. Si son iguales, entonces las dos listas son iguales hasta ese punto, por lo que la recursividad se utiliza en las CDR de ambas. Este caso falla cuando se encuentran dos átomos desiguales. Cuando esto ocurre, el proceso no necesita continuar y devuelve # F.
Tenga en cuenta que *listasiguales* espera listas como parámetros, y no funcionará correctamente si uno o ambos parámetros son átomos.

El problema de la comparación de listas generales es ligeramente más complejo que esto, porque deben evaluarse las sublistas por completo en el proceso de comparación.
En esta situación, la recursión es muy apropiada, porque la forma de sublistas es la misma que la de las listas dadas. Cuando los elementos correspondientes de las dos listas dadas son listas, se separan en dos partes, CAR y CDR, y se utiliza la recursividad. Si los elementos correspondientes de las dos listas que figuran son átomos, pueden simplemente compararse utilizando EQ?.

La definición de la función completa sería:

```
(DEFINE (equal list1 list2)
        (COND
                ((NOT (LIST? list1)) (EQ? list1 list2))
                ((NOT (LIST? list2)) #F)
                ((NULL? list1) (NULL? list2))
                ((NULL? list2) #F)
                ((equal (CAR list1) (CAR list2))
                        (equal (CDR list1) (CDR list2)))
                (ELSE #F)
))
```

Los dos primeros casos de la COND manejan la situación en la que uno de los parámetros es un átomo, en lugar de una lista. El tercer y cuarto casos manejan la situación donde una o ambas listas están vacías. Estos casos también evitan

casos posteriores de intentar aplicar CAR a una lista vacía. El quinto caso de COND es el más interesante.
El predicado es una llamada recursiva con los CAR de las listas como parámetros. Si esta llamada recursiva devuelve #T, a continuación, la recursividad se utiliza de nuevo en las CDR de las listas.
Este algoritmo permite que las dos listas incluyan listas secundarias a cualquier profundidad.
Esta definición de *equal* trabaja con cualquier par de expresiones, no sólo listas.
Equal es equivalente a la función de EQUAL?. Tenga en cuenta que EQUAL? sólo debe utilizarse cuando sea indispensable, porque es mucho más lento que EQ? y EQV?.

Otra operación de lista comúnmente necesaria es la de construir una nueva lista que contiene todos los elementos de dos argumentos de lista dados. Esto es generalmente implementado como una función de Scheme llamada *append*. La lista de resultados se puede construir por el uso repetido de CONS, colocando los elementos de la primera lista en la segunda, que se convierte en la lista de resultados. Veamos los siguientes ejemplos:

(append '(A B) '(C D R)) devuelve (A B C D R)
(append '((A B) C) '(D (E F))) devuelve ((A B) C D (E F))

La definición de *append* es:

```
(DEFINE (append list1 list2)
        (COND
                ((NULL? list1) list2)
                (ELSE (CONS (CAR list1) (append (CDR list1) list2)))
))
```

El primer caso de COND se utiliza para terminar el proceso recursivo cuando la primera lista está vacía, devolviendo la segunda lista.

En el segundo caso (el ELSE), el CAR de la primera lista de parámetros está incluido en el resultado devuelto por la llamada recursiva, que pasa a la CDR de la primera lista como su primer parámetro.

Veamos la siguiente función Scheme, llamada *guess*, que utiliza la función *member* que vimos anteriormente:

```
(DEFINE (guess list1 list2)
        (COND
                ((NULL? list1) '())
                ((member (CAR list1) list2)
                        (CONS (CAR list1) (guess (CDR list1) list2)))
                (ELSE (guess (CDR list1) list2))
))
```

Guess produce una lista simple que contiene los elementos comunes a las dos listas. Por lo tanto, si las listas representan conjuntos, *guess* genera una lista que representa la intersección de los dos conjuntos.

Función LET

LET es una función que crea un ámbito local, en la que los nombres se enlazan temporalmente a los valores de las expresiones. A menudo se utiliza para factorizar las subexpresiones comunes de expresiones más complejas. Estos nombres se pueden utilizar en la evaluación de otra expresión, pero no pueden ser nuevamente enlazados a nuevos valores en LET.

El siguiente ejemplo ilustra el uso de LET. Se calcula las raíces de una ecuación de segundo grado dada, suponiendo que las raíces son reales. La definición matemática de las raíces reales (a diferencia de las complejas) de la ecuación cuadrática: $ax^2 + bx + c$

son las siguientes:

raiz1 = (-b + sqrt (b^2 - 4ac)) / 2a

raiz2 = (-b - sqrt (b^2 - 4ac)) / 2a

```
(DEFINE (quadratic_roots a b c)
        (LET (
                (root_part_over_2a (/ (SQRT (− (* b b) (* 4 a c))) (* 2 a)))
                (minus_b_over_2a (/ (− 0 b) (* 2 a)))
                )
        (LIST (+ minus_b_over_2a root_part_over_2a)
                (− minus_b_over_2a root_part_over_2a))
))
```

En este ejemplo se utiliza LIST para crear la lista de los dos valores que componen el resultado.
Debido a que los nombres enlazados en la primera parte del LET no pueden estar modificados en la siguiente expresión, no son los mismos que las variables locales en un bloque en un lenguaje imperativo.
LET es en realidad la abreviatura de una expresión lambda aplicada a un parámetro.

Los siguientes dos expresiones son equivalentes:

```
(LET ((alpha 7)) (* 5 alpha))
((LAMBDA (alpha) (* 5 alpha)) 7)
```

En la primera expresión, 7 está enlazado a alfa con LET; en la segunda, 7 está enlazado a alfa mediante el parámetro de la expresión lambda.

Funciones que construyen código

Recordemos que el intérprete de Scheme usa una función llamada EVAL. EVAL se aplica a toda expresión tipeada, ya sea un símbolo de Scheme en el intérprete o parte de un programa que se está interpretando. La función EVAL también se puede llamar directamente por los programas de Scheme.
Esto proporciona la posibilidad de que un programa cree expresiones y llamando a EVAL las evalúe.

Uno de los ejemplos más simples de este proceso consiste en átomos numéricos. Scheme incluye una función llamada +, que toma cualquier número de átomos numéricos como argumentos y devuelve su suma. Por ejemplo, (+ 3 7 10 2) devuelve 22.

Ahora supongamos que en un programa tenemos una lista de átomos numéricos y la necesidad de sumarlos. No podemos aplicar + directamente en la lista, porque + puede tomar sólo los parámetros atómicos, no una lista de átomos numéricos. Pero podríamos, por supuesto, escribir una función que añade varias veces el CAR de la lista a la suma de su CDR, utilizando la recursividad para recorrer la lista.

Tal función sería:

```
(DEFINE (adder a_list)
    (COND
        ((NULL? a_list) 0)
        (ELSE (+ (CAR a_list) (adder (CDR a_list))))
))
```

A continuación se presenta un ejemplo de llamada a *adder*, junto con las llamadas y devoluciones recursivas:

```
(adder '(3 4 5))
(+ 3 (adder (4 5)))
(+ 3 (+ 4 (adder (5))))
(+ 3 (+ 4 (+ 5 (adder ()))))
(+ 3 (+ 4 (+ 5 0)))
(+ 3 (+ 4 5))
(+ 3 9)
(12)
```

Una solución alternativa al problema es escribir una función que construye una llamada a + con las formas de los parámetros adecuados. Esto se puede hacer

mediante el uso de CONS para construir una nueva lista que es idéntica a la lista de parámetros, excepto que tiene el átomo + agregado al comienzo. Esta nueva lista puede entonces ser sometida a EVAL para su evaluación, como en el siguiente ejemplo:

```
(DEFINE (adder a_list)
        (COND
                ((NULL? a_list) 0)
                (ELSE (EVAL (CONS '+ a_list)))
))
```

Tenga en cuenta que el nombre de la función + es precedida con Quote para prevenir que EVAL la evalúe en la evaluación de CONS. A continuación se presenta un ejemplo de llamada a esta nueva versión de *adder*, junto con la llamada de EVAL y el valor de retorno:

```
(adder '(3 4 5))
(EVAL (+ 3 4 5)
(12)
```

En todas las versiones anteriores de Scheme, la función EVAL evalúa su expresión en el ámbito de aplicación más externa del programa. Las versiones posteriores de Scheme, a partir de la 4, requieren un segundo parámetro para EVAL que especifica el alcance de la expresión que se va a evaluar. Por razones de simplicidad, dejamos el alcance fuera de nuestro ejemplo, y no hablamos de nombres de ámbito aquí.

Haskell

Haskell es un lenguaje fuertemente tipado, las funciones en Haskell se pueden sobrecargar y es un lenguaje de programación funcional puro, lo que significa que no tiene expresiones o declaraciones que tengan efectos laterales.

Veamos la siguiente definición de la función factorial:

```
fact 0 = 1
fact 1 = 1
fact n = n * fact (n – 1)
```

Aquí, en primer lugar, no existe una palabra reservada para introducir la definición de una función. En segundo lugar, las definiciones alternativas de funciones (con diferentes parámetros formales) tienen todas el mismo aspecto.

Las listas se escriben entre corchetes, como por ejemplo:

```
colores = ["blue", "green", "red", "yellow"]
```

Haskell incluye una colección de operadores de lista. Por ejemplo, las listas pueden ser concatenadas con ++, : sirve como una versión infija de CONS, y .. se utiliza para especificar una serie aritmética en una lista.

Por ejemplo:

```
5 : [2, 7, 9] devuelve [5, 2, 7, 9]
[1, 3, 5] ++ [2, 4, 6] devuelve  [1, 3, 5, 2, 4, 6]
```

Haskell incluye un constructor **let**. Por ejemplo, se podría escribir:

```
quadratic_root a b c =
    let
        minus_b_over_2a = − b / (2.0 * a)
        root_part_over_2a = sqrt(b ^ 2 − 4.0 * a * c) / (2.0 * a)
    in
        minus_b_over_2a − root_part_over_2a,
        minus_b_over_2a + root_part_over_2a
```

Listas por comprensión:

Veamos el siguiente ejemplo:

```
[n * n * n | n <− [1..50]]
```

Esto define una lista de los cubos de los números del 1 al 50. Se lee como "una lista de todos los n * n * n de modo que n es tomada en el rango de 1 a 50".

Evaluación perezosa

En Scheme los parámetros de una función son completamente evaluados antes de que la función sea llamada. Mediante la *evaluación perezosa,* un parámetro actual se evalúa sólo cuando su valor es necesario para evaluar la función. Por lo tanto, si una función tiene dos parámetros, pero en una ejecución particular de la función no se utiliza el primer parámetro, el parámetro actual no será evaluado. Además, si sólo una parte de un parámetro actual debe ser evaluado para una ejecución de la función, el resto se deja sin evaluar. Finalmente, los parámetros actuales se evalúan solamente una vez, en todo caso, incluso si el mismo parámetro actual aparece más de una vez en una llamada de función.

La evaluación perezosa permite definir estructuras de datos infinitas.

Por ejemplo:

```
positives = [0..]
evens = [2, 4..]
squares = [n * n | n <− [0..]]
```

Por ejemplo, si se quiere saber si un número en particular es un cuadrado perfecto, podríamos comprobar la lista *squares* con una función de pertenencia. Supongamos que tenemos una función de predicado llamada *member* que determina si un átomo dado se contiene en una lista dada.

Entonces podríamos usarlo como:

```
member 16 squares
```

que devolverá True. La definición de *squares* sería evaluada hasta que el 16 fuese encontrado.

Específicamente, supongamos que la función *member* se define como sigue:

```
member b [] = False
member b (a:x)= (a == b) || member b x
```

La segunda línea de esta definición divide el primer parámetro en cabeza y cola. Su valor de retorno es verdadero si la cabeza coincide con el elemento que está buscando (b), o si la llamada recursiva con la cola de la lista devuelve True.
Esta definición de *member* funcionaría correctamente con los cuadrados sólo si el número dado era un cuadrado perfecto. Si no es así, los cuadrados mantendrían la generación de cuadrados para siempre, o hasta que se alcance una cierta limitación de memoria, en busca del número en la lista.

La evaluación perezosa no está exenta de costos. En este caso, el costo es en un momento la semántica más complicada, lo que resulta en la velocidad mucho más lenta de la ejecución.

Los lenguajes imperativos vs. lenguajes funcionales

Los lenguajes funcionales pueden tener una estructura sintáctica muy simple. La estructura de lista de LISP, que se utiliza tanto para el código como para los datos, lo ilustra claramente.
La sintaxis de los lenguajes imperativos es mucho más compleja. Esto los hace más difíciles de aprender y de utilizar.

La semántica de los lenguajes funcionales es también más simple que la de los lenguajes imperativos. Por ejemplo, en la descripción de la semántica denotacional de un bucle imperativo, el bucle se convierte de una iteración a una recursión.

Cuando los programas funcionales se interpretan, son, por supuesto, mucho más lentos que su contraparte compilada. Sin embargo, ahora hay compiladores para la mayoría de los lenguajes funcionales, por lo que las disparidades entre la velocidad de ejecución de un lenguaje funcional y un lenguaje imperativo compilado, ya no son tan grandes.

Los lenguajes funcionales tienen una ventaja potencial en la legibilidad. En muchos programas imperativos, los detalles de tratar con variables oscurecen la lógica del programa.

Veamos por ejemplo una función que calcula la suma de los cubos de los n primeros números enteros positivos.

En C, una función de este tipo tendría un aspecto similar a:

```
int sum_cubes (int n) {
        int sum = 0;
        for(int index = 1; index <= n; index++)
                sum += index * index * index;
        return sum;
}
```

En Haskell, la función sería:

```
sumCubes n = sum (map (^3) [1..n])
```

Esta versión se limita a indicar tres pasos:

1. Construir la lista de números ([1..n]).
2. Crear una nueva lista mediante la asignación de una función que calcula el cubo de un número a cada número de la lista.
3. Sumar la nueva lista.

Un factor simple que afecta en gran medida la complejidad de los imperativos, es que es necesaria la atención del programador para determinar el estado del programa en cada etapa de su desarrollo. En un programa grande, el estado del programa es un gran número de valores. En la programación funcional pura, no hay ningún estado; por lo tanto, no necesitará dedicar atención del programador.

La ineficiencia de las primeras implementaciones de los lenguajes funcionales era claramente un factor de aquel entonces. Además, la gran mayoría de los programadores

aprenden a programar utilizando los lenguajes imperativos, lo que hace que los programas funcionales parezcan extraños y difíciles de entender.

Bibliografía

Concepts of Programming Languages. 10° edición (Autor: Robert.W. Sebesta – Editorial: Addison – Wesley) – Unidad 15.