Lenguajes de Programación I

Listas - Entrada y Salida

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Listas

- Una lista es un tipo recursivo
  - Lista vacía.
  - Un par contienendo un objeto y otra lista.
- Herramientas fundamentales en lenguajes funcionales y lógicos.
  - Por su naturaleza recursiva.
  - Por disponer de funciones especializadas para su manipulación.
- Lisp/Scheme son **homoicónicos**
  - Los programas **son** listas.
  - Pueden transformarse a sí mismos durante su ejecución.
- Funcionan mejor en lenguajes con recolección de basura.
- Muchos lenguajes imperativos con soporte extenso para listas.

Representación

- Según su contenido pueden ser
  - **Homogéneas** – elementos **deben** ser del mismo tipo.
  - **Heterogéneas** – elementos pueden ser de tipos diferentes.
- Esto obliga a representaciones diferentes
  - Homogéneas – listas de bloques del mismo tipo.
  - Heterogéneas
    - Cadena de descriptores con un indicador especial para las listas vacías (`cons` y `nil` en Lisp/Scheme).
    - Bloque de referencias a los objetos contenidos (Perl).

Notación y Operaciones

- Notación explícita para facilitar su comprensión
  - `(a b c d)` – Lisp/Scheme
  - `[a, b, c, d]` – Haskell, Prolog, Erlang, Ruby...
  - `(a, b, c, d)` o bien `qw(a b c d)` – Perl
- Funciones para operar sobre listas
  - Construcción de listas (`cons`ing o *wrapping*).
  - Concatenación de listas – ¡costosa!
  - Extraer y agregar elementos por ambos extremos.
  - Predicados para vacuidad.
  - Calcular la longitud – ¡costosa y usualmente mala idea!
  - Iteración implícita vía recursión – *fold*, *unfold*, *map*, *scan*, *zip*, ...

Operaciones

Lisp/Scheme	Haskell	Perl
`(define lista '(1 2 3))`	`lista = [1 2 3]`	`@lista = qw/1 2 3/;`
`(length lista)`	`length lista`	`length @lista;`
`(car lista)`	`head lista`	`unshift @lista;`
`(cons 0 lista)`	`0 : lista`	`shift 0, @lista;`
`(filter even? lista)`	`filter p lista`	`grep { p($_); } @lista`
`(append lista1 lista2)`	`lista1 ++ lista2`	`( @lista1, @lista2 )`

- Perl permite usarlas como *deques* – `pop` y `push`.
- Perl permite acceder como si fueran arreglos – `$lista[2]`
- Perl permite usarlas como un todo – `@lista[0..2,5]`

Listas por Extensión

Construyendo listas con notación de conjuntos

- *List Comprehensions*
  - Notación de conjuntos original de Miranda – disponible en Haskell, Erlang y Python.
  - Expresión para calcular cada elemento que compone la lista.
  - Uno o más generadores (*enumeradores*) de candidatos.
  - Uno o más filtros booleanos para seleccionar candidatos.
- Lista de los primeros cuadrados de números impares

  ```
  [ i * i | i <- [1..100], i `mod` 2 == 1 ]
  ```
- Producto cartesiano independiente – filtro implícito `true`

  ```
  [ (a,b) | a <- [1..3], b <- [1..2] ]
  ```
- Producto cartesiano dependiente – filtro implícito `true`

  ```
  [ (a,b) | a <- [1..4], b <- [a+1..4] ]
  ```

Listas por Extensión Aplicadas

Expresar el conjunto solución

- **Quicksort**

  ```
  qs []     = []
  qs (x:xs) = qs [ y | y <- xs, y <= x ] ++
              [x] ++
              qs [ y | y <- xs, y > x ]
  ```

  Compacto pero muy ineficiente en espacio – *Quicksort* fue diseñado para lenguajes imperativos
- **Factorización de Enteros**

  ```
  factor n = [ i | i <- [1..n], n `mod` i == 0 ]
  ```
- **Números de Fibonacci (Infinito...)**

  ```
  fib = 1:1:[ a+b | (a,b) <- zip fib ( tail fib ) ]
  ```

Entrada y Salida

De lo contrario, los lenguajes son inútiles

- **Entrada y Salida Interactiva**
  - Interacción con humanos o dispositivos físicos.
  - Entrada y salida dependientes de ejecución e interacción externa.
- **Entrada y Salida hacia Archivos**
  - Interacción con almacén de datos ajeno al espacio del programa.
  - Temporales vs. Persistentes.
- Uno de los elementos más difíciles de diseñar en un lenguaje – se apoya en servicios provistos por el sistema operativo anfitrión.

Implantación

Las preocupaciones fundamentales

- Tipo *builtin* vs. librerías.
  - Pascal tiene `file`, Perl tiene *filehandles*, Haskell tiene `Handle` ...
  - C tiene `FILE`, Java tiene `java.io` ...
- Archivos de texto vs. archivos binarios.
  - Para algunos es indiferente (Linux, MacOSX, cualquier Unix).
  - Para otros es esencial indicarlo (Windows, AmigaOS, VMS).
- Codificación de caracteres.
  - Asumir *bytes* y que el programador sufra – C.
  - Asumir una codificación y "vivir con ella" – Java, Erlang.
  - Ofrecer mecanismos de conversión – Haskell, Perl IO Layer.
- Entrada y salida con formato.

Más complicado si el lenguaje debe ser portátil

Comparación

- Relativamente simple para tipos primitivos escalares – hardware ofrece formas de compararlos rápidamente.
- Para tipos complejos, ¿qué hacer cuando se comparan dos objetos?
  - ¿Ver si son aliases?
  - ¿Ver si ocupan espacios que son idénticos bit a bit?
  - ¿Ver si tienen estructuras isomorfas?
- Más complicado en presencia de referencias
  - **Superficial** – hacen referencia al mismo **objeto**.
  - **Profunda** – los referenciados tienen la misma **estructura y contenido**.

Asignación

- Sólo relevantes en lenguajes imperativos.
- Trivial para tipos primitivos escalares – hardware ofrece formas de trasladarlos rápidamente.
- Para tipos complejos, ¿qué hacer cuando se asigna un objeto a otro?
- Modelo de Referencias
  - **Superficial** – el asignado será otra referencia al original.
  - **Profunda** – se construye una copia del original.
- Modelo de Valores
  - Simplemente se copian los valores – funciona con registros y arreglos.
  - Si es un apuntador, se copia el *valor* apuntador – los contenidos que los copie el programador.
  - Coincide con **Asignación Superficial**.

Bibliografía

- [*Scott*]
  - Secciones 7.8 a 7.10
  - Ejercicios y Exploraciones