Introducción a LISP

1. Sintaxis (Resumen)

LISP es un acrónimo de *List Processing*. El lenguaje está compuesto escencialmente de dos cosas:

- Átomos:
  - Símbolos (Identificadores): comienzan con un carácter seguido de cualquier otro caracter excepto espacios, comas, puntos o parentesis.
  - Números: las reglas normales de escritura de números se aplican aquí.

  LISP **no es sensible al caso**.

- Listas:

  Lista ::= ‘‘()’’ | ‘‘NIL’’ |
  ‘‘(’’ <atomo$_1$> [<atomo$_2$>] ... [<atomo$_n$>] ‘‘)’’

  Los átomos en una lista deben estar separados por espacios.

Átomos especiales (palabras reservadas).

- Los átomos `T` y `NIL` están reservados para representar el *valor de verdad verdadero* y *falso*, respectivamente.
- `NIL` es al mismo tiempo un átomo y una lista vacía.
- Los nombres de funciones predeterminadas de LISP también son palabras reservadas y no deberían usarse como identificadores.

Ejemplos:

- Átomos: `A, X, B, un-atomo, p+100, atomo*valido-`
- Listas: `(1 2 3 4), (a 3 b un-atomo-valido), (p+100 no es una suma) (pero) (+ p 100) (si lo es)`

2. Semántica

LISP funciona a través de un intérprete. La misión del interprete es evaluar la expresión que se introduce. La evaluación procede de la siguiente forma:

`evalua`(X):

1. si X es un átomo, entonces devolver el valor de la variable X
2. si X es una lista y X_1,...,X_n los elementos de X, hacer
   (a) Y_i = `evalua`(X_i), para $i = 2, .., n$
   (b) regresar X_1(Y_2, Y_3,..., Y_4), es decir, llamar a la función especificada por X_1 con los argumentos Y_2,..,Y_n.
   (si X_1 no es un átomo o la función especificada no existe, se produce un error).

Todas las instrucciones en LISP son funciones y regresan un valor. Además, por la forma como funciona el intérprete, LISP usa notación prefija.

Ejemplos:

Cómo declarar una variable global (el símbolo “>” es el *prompt* de LISP):

```
> X → ERROR
> (setf X 3) → 3
> X → 3
```

La función `setf` es una función predeterminada de LISP que crea una nueva variable y le asigna un valor (si la variable ya existe, cambia su valor anterior).

```
> (a b c) → ERROR  (la función “a” no existe)
> (quote (a b c)) → (a b c)
> '(a b c) → (a b c)
```

En los últimos dos ejemplos se hace uso de una función predeterminada de LISP llamada `quote` o simplemente `'`. Esta función es especial en el sentido de que *evita la evaluación de su único argumento* y devuelve su argumento al evaluarse.

Algunas funciones comunes de LISP:

- `(FIRST <Lista>)` ó `(CAR <Lista>)`: Devuelve el primer elemento de la lista.
- `(REST <Lista>)` ó `(CDR <Lista>)`: Devuelve la lista menos el primer elemento.
- `(CONS <Elem> <Lista>)`: Genera una nueva lista con `Elem` al principio de `Lista`.
- `(APPEND <Lista1> <Lista2>)`: Pega las listas y regresa el resultado.
- `(MEMBER <Elem> <Lista>)`: Si `Elem` está en `Lista`, entonces regresa la lista a partir de ese elemento, en caso contrario regresa `NIL`.
- `(NULL <Lista>)`: Regresa `T` si la lista está vacía.
- `(SETF <Variable> <Valor>)`: Asigna el valor a la variable (y la crea si no existe). Regresa el valor asignado.
- `(REVERSE <Lista>)`: Invierte la lista.

- `(NTH <n> <Lista>)`: Regresa el `n`-ésimo elemento de `Lista` (basado en 0).
- `(NTHCDR <n> <Lista>)`: Elimina los primeros `n` elementos de Lista.
- `(LENGTH <Lista>)`: Regresa la longitud de la lista.
- `(LIST <Elem`$_1$`> <Elem`$_2$`> ... <Elem`$_n$`>)`: Crea una nueva lista cuyos elementos son `Elem`$_1$...`Elem`$_n$
- `(RETURN <Expr>)`: Provoca que la función actual termine y regresa el valor dado por `Expr`.
- `(DOLIST (<Iterador> <Lista> [<Resultado>]) <Expr`$_1$`> ... <Expr`$_n$`>)`: Para cada elemento de `Lista`, se ejecutan las expresiones `Expr`$_1$... `Expr`$_n$. Regresa la expresión `Resultado`.
- `(DOTIMES (<Iterador> <ValorFinal> [<Resultado>]) <Expr`$_1$`> ... <Expr`$_n$`>)`: El iterador toma los valores 0...`ValorFinal`-1 y para cada valor, ejecuta las instrucciones `Expr`$_1$...`Expr`$_n$). Regresa la expresión `Resultado`.

- `(DO ( (<X`$_1$`> <ValInicial`$_1$`> [<Incr`$_1$`>]) ...`
  `(<X`$_n$`> <ValInicial`$_n$`> [<Incr`$_n$`>])`
  `) (<Cond> <Resultado>)`
  `<Expr`$_1$`> ... <Expr`$_n$`> ) :`
  Iteración genérica. Los iteradores X_1...X_n se inicializan e incrementan *en paralelo*. El valor regresado es `Resultado`.
- `(LET ( (<X`$_1$`> <ValInicial`$_1$`>) ...`
  `(<X`$_n$`> <ValInicial`$_n$`>)`
  `) <Expr`$_1$`> ... <Expr`$_n$`> ) :`
  Secuenciación genérica, las variables son creadas e inicializadas *en paralelo*. El valor regresado es `Expr`$_n$.
- `(PROG1 <Expr`$_1$`> ... <Expr`$_n$`>)`: Secuenciación genérica. El valor regresado es `Expr`$_1$.
- `(PROGN <Expr`$_1$`> ... <Expr`$_n$`>)`: Secuenciación genérica. El valor regresado es `Expr`$_n$.

- `(COND (<Cond`$_1$`> <Expr`$_1$`>) ...`
  `(<Cond`$_n$`> <Expr`$_n$`>)`
  `)` : Selección múltple. Si `Cond`$_i$ es verdadera, entonces se ejecuta `Expr`$_i$. El valor regresado es la expresión seleccionada.

- `(DEFUN <Funcion> (<Arg`$_1$`> ... <Arg`$_n$`>)`
  `(<Expr`$_1$`> ... <Expr`$_n$`>)`
  `)` : Define una nueva funcón llamada `Funcion` que recibe los argumentos `Arg`$_1$...`Arg`$_n$. El resultado que regresa la nueva función es `Expr`$_n$.

- Modificadores de argumentos de función:
  - `&KEY <Var`$_1$`> ... <Var`$_n$`>`: Declara argumentos por nombre (argumentos clave). Para poner valores a estos argumentos hay que escribir `:<Var`$_i$`> <Valor`$_i$`>` al llamar a la función. Para inicializar los argumentos se usa la sintaxis `(<Var`$_i$`> <Expr`$_i$`>)`.
  - `&AUX <Var`$_1$`> ... <Var`$_n$`>`: Declara variables locales nuevas. Las nuevas variables se inicializan a `NIL` a menos que se use la sintaxis `(<Var`$_i$`> <Expr`$_i$`>)`.

- `(IF <Cond> <Then> <Else>)`: Selección condicional sencilla. Regresa la expresión que se haya ejecutado.
- LISP usa los operadores aritméticos normales: `+, -, *, /` y los comparadores aritméticos `=,>,<,>=,<=`. La desigualdad se hace con `(NOT (= <Expr`$_1$`> <Expr`$_2$`>))`.
- La división puede dar *números racionales*. Para convertirlos a números de punto flotante se usa la función `float`, para convertirlos a enteros se puede usar `floor` o `round`.
- `(EQ <S`$_1$`> <S`$_2$`>)`: Compara dos símbolos, pero no compara números reales. `EQL` usa `EQ` y regresa `T` si sus argumentos son números del mismo tipo y valor. `EQUAL` hace lo mismo que `EQL` pero también compara listas.
- `AND, OR, NOT`: Conjunción, disyunción y negaciones lógicas, respectivamente. `AND` y `OR` pueden llevar más de un argumento.

Ejercicio: Programe la ordenación de listas por selección, inserción y *quicksort*.

- `Seleccion(L)`: Se extrae el elemento más pequeño de `L` y se pone al final de una lista temporal `Lp`. El algoritmo se repite hasta dejar `L` vacía, en cuyo caso, `Lp` contiene la lista ordenada.
- `Insercion(L)`: Se toma el siguiente elemento de `L` y se inserta de forma ordenada en una lista temporal (ordenada) `Lp`. El algoritmo continúa hasta que todos los elementos de `L` han sido insertados en `Lp`, en cuyo caso, `Lp` contiene la lista ordenada.
- `Quicksort(L)`: `L` se parte en dos listas `L-` y `L+` de acuerdo a algún elemento pivote `p` de `L` de forma que todos los elementos de `L-` son menores (o iguales) que `p` y todos los elementos de `L+` son mayores. Después se ordenan `L-` y `L+` usando `Quicksort` y se fusionan (el pivote debe de ponerse en medio de ambas listas durante la fusión). ¿Cómo podría implementar este algoritmo para que corra en paralelo?

3. Breve nota sobre la programación funcional

La programación funcional está basada en usar las funciones matemáticas con sus propiedades normales. Una de las propiedades clave es que la misma expresión siempre tiene el mismo valor en el mismo contexto. A esto se le llama *transparencia referencial*. Esto quiere decir que no se permiten efectos secundarios (como paso de argumentos por *referencia*).

Por ejemplo, en la expresión `f(sqrt(2), sqrt(2))`, podemos hacer un símbolo nuevo `s = sqrt(2)` y substituir la expresión anterior por `f(s, s)` y tendremos el mismo resultado. Sin embargo, si hacemos `f(getchar(), getchar())`, ya *no* podemos sustituir los argumentos por un sólo símbolo, debido a que `getchar()` produce efectos secundarios.

Esta clase de efectos secundarios no son permitidos en lenguajes funcionales puros. Sin embargo, LISP tiene varias funciones que permiten efectos secundarios (como el `DEFUN`). Pero, por otro lado *no es absolutamente necesario utilizar `DEFUN` para escribir un programa*...

4. Metaprogramación

LISP permite que se genere código *en tiempo de ejecución*. Esto puede lograrse con una variedad de métodos:

- `(EVAL <Expr>)`: Esta función evaúa `Expr` y luego *ejecuta* el resultado.
- `(APPLY <Apuntador-Funcion> <Lista-Argumentos>)`: Esta función ejecuta la función "apuntada por" `Apuntador-Funcion` con la lista de argumentos proporcionada. Ejemplo:

  `> (APPLY #'+ '(3 2))` $\rightarrow$ `5`.

  Note que el operador `#'` se usa para obtener la "dirección" de la función predefinida `+`.

- `(LAMBDA (<Arg`$_1$`> ... <Arg`$_n$`>) <Expr`$_1$`> ... <Expr`$_m$`>)`: Crea una función anónima *in situ* para poder ser usada por funciones como `APPLY`. La nueva función regresa el valor de `Expr`$_m$.

Sabiendo que existe la siguiente función de LISP:

`(MIN <Valor`$_1$`> ... <Valor`$_n$`>)`

Que regresa el mínimo de los valores dados, ¿cómo podría haber usado la metaprogramación y la función `MIN` para programar la ordenación por selección?

Análisis del lenguaje natural

Otro uso posible de la metaprogramación es para el análisis del lenguaje natural. Se puede hacer un programa que reciba como entrada las reglas de producción de alguna oración y luego, en tiempo de ejecución, genere el código que acepta tales oraciones y lo incluya en su programa. Es decir, tendríamos un programa que se modifica y actualiza a sí mismo durante su ejecución. Esto es posible puesto que en LISP, el código y los datos tienen el mismo formato.

¿Cómo podría implementarse esto con lenguajes de programación tradicionales?

Tarea (1): Conversión a notación prefija

Haga un programa que reciba una expresión algebraica en notación infija y la convierta a notación prefija. Ejemplos:

```
> (parse '(a + b / 2 - (4 * a * c - 8) / (8 * x + 4)))
(+ A (- (/ B 2) (/ (- (* 4 (* A C)) 8) (+ (* 8 X) 4))))


> (parse '(- 3 * 4 / - 8 + (7 - 15) * 4))
(+ (* (- 3) (/ 4 (- 8))) (* (- 7 15) 4))

> (+ (* (- 3) (/ 4 (- 8))) (* (- 7 15) 4))
-61/2
```

Los operadores aceptados deben ser: `+`, `-` (binario y unario), `*`, y `/`. Además, note que en el segundo ejemplo, la respuesta se evalúa usando el intérprete de LISP. Esta es una manera sencilla de verificar que su traductor funciona correctamente.

Tarea (2): Ejemplo el poder del procesamiento simbólico

Haga un programa que resuelva, de forma simbólica, una ecuación algebraica cualquiera escrita en sintaxis de LISP (prefija). La restricción es que la incógnita sólo puede aparecer una vez en toda la ecuación. Ejemplo:

```
> (solve 'x '(= (* 3 a) (- y (/ (+ z 8) (^ x k)))))
(= X (^ (/ (+ Z 8) (- Y (* 3 A))) (/ 1 K)))
```

En este ejemplo, la función `solve` recibe dos argumentos: la incógnita con respecto a la cual se va a despejar y la ecuación de la cual se va a despejar. Además, se ha usado el símbolo `^` para simbolizar la exponenciación (la exponenciación genérica se lleva a cabo con la función `EXPT` en LISP).

Los operadores aceptados deben ser: `+`, `-` (binario y unario), `*`, `/`, y `^` (exponenciación). La incógnita a despejar puede aparecer en cualquier lado de la ecuación.