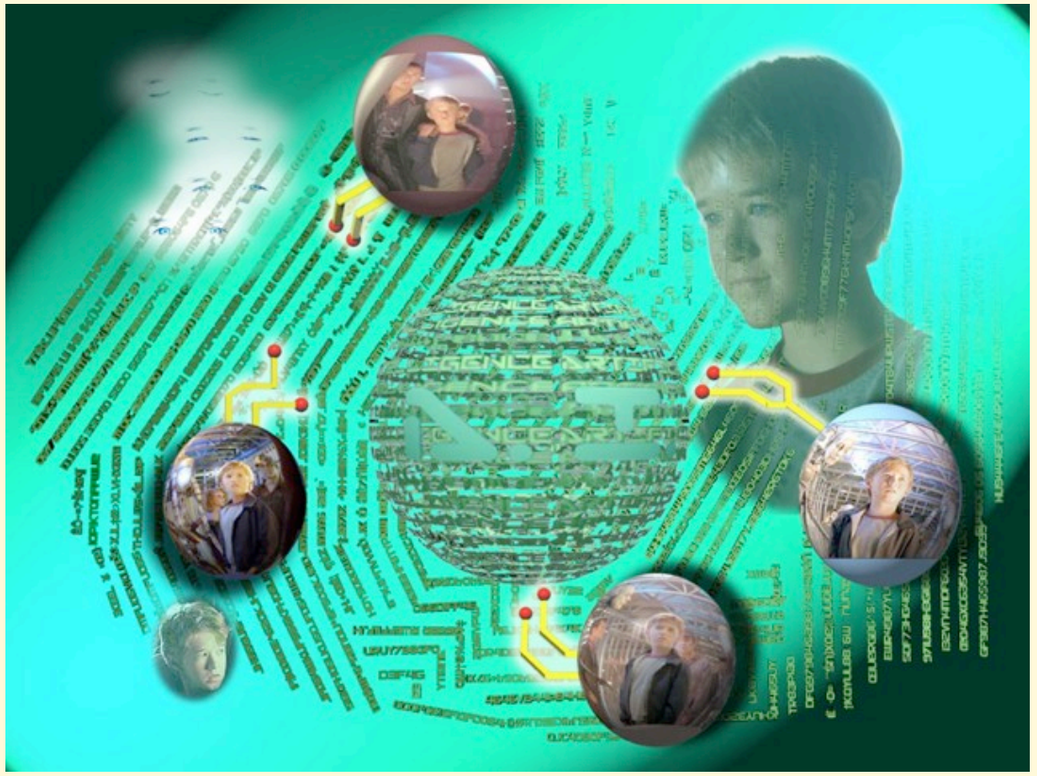
Lenguajes de programación

Abdiel E. Cáceres González
Instituto Tecnológico de Monterrey
Campus Ciudad de México
Verano 2004

Tower of Babel by Gustav Dore 1986

TECNOLÓGICO DE MONTERREY

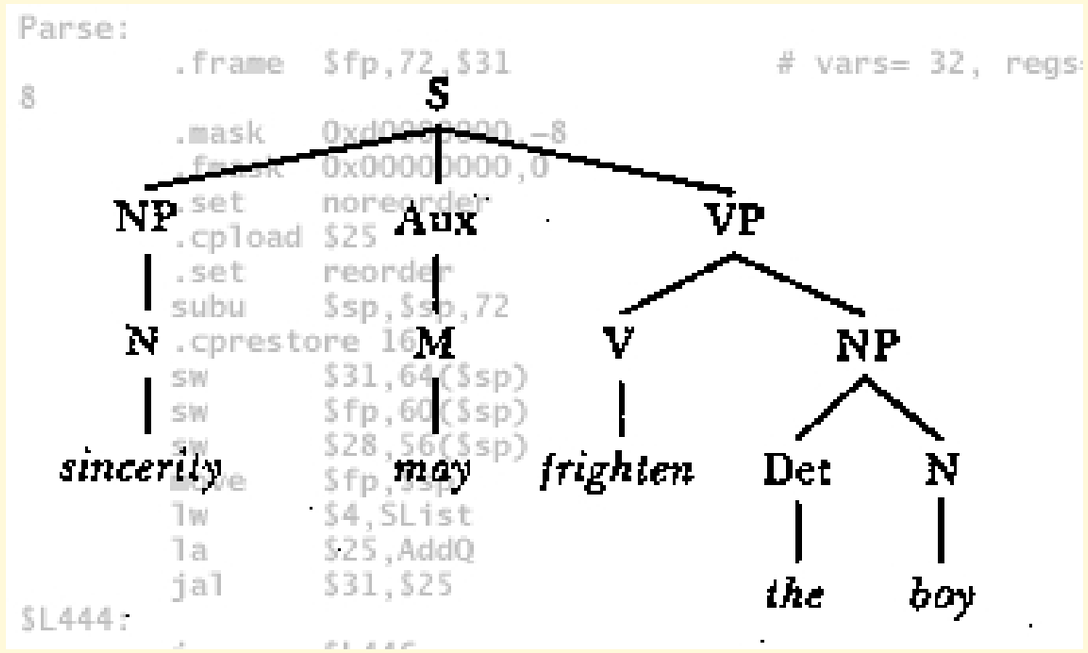
LISP (Antecedentes Históricos)

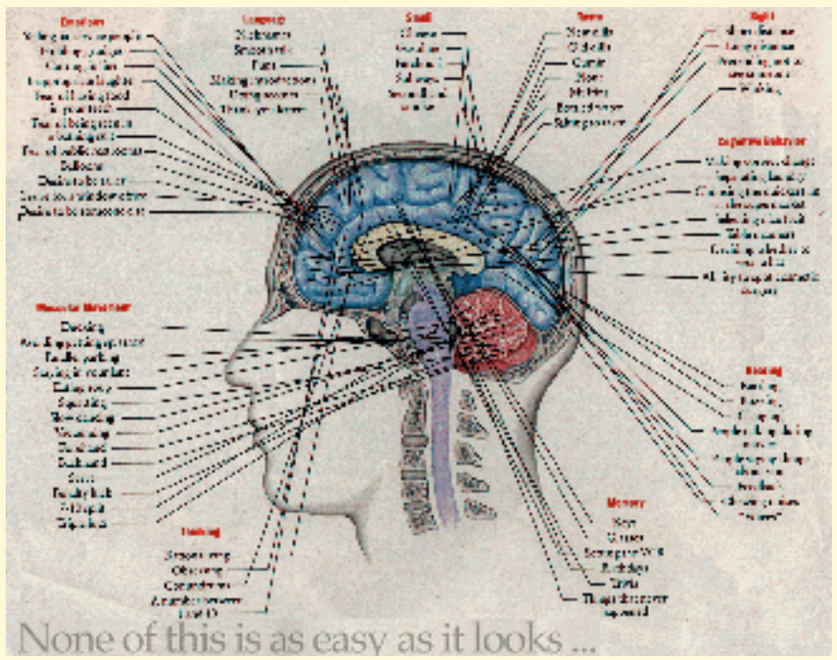


El interés por la inteligencia artificial (IA) comenzó a mediados de los 1950s en diversos lugares alrededor del mundo.

Parte de este interés se derivó de la lingüística, parte de la psicología y parte de las matemáticas.

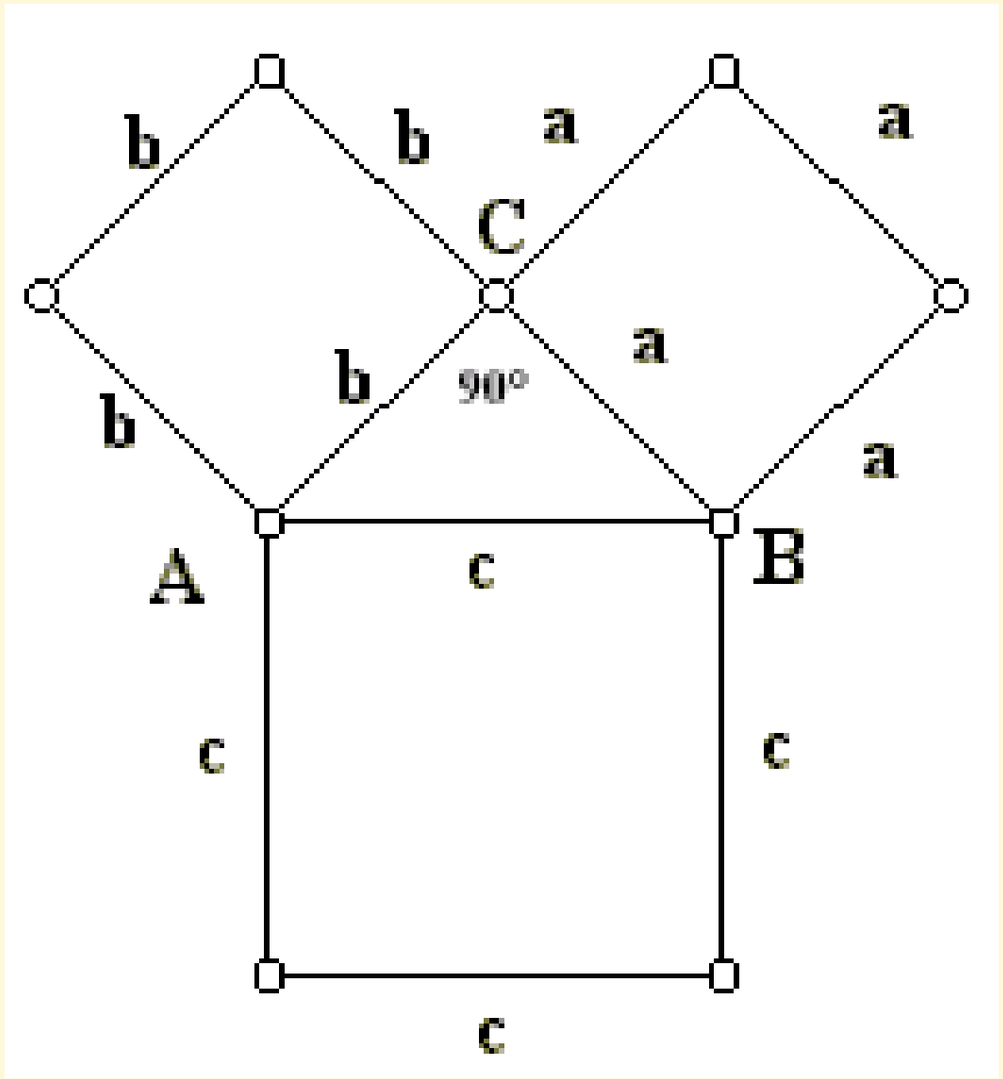
LISP (Antecedentes Históricos)





Los lingüistas estaban interesados en el procesamiento en lenguaje natural. Los psicólogos estaban interesados en modelar el almacenamiento y la recuperación de información de los humanos, junto con otros procesos fundamentales del cerebro.

LISP (Antecedentes Históricos)



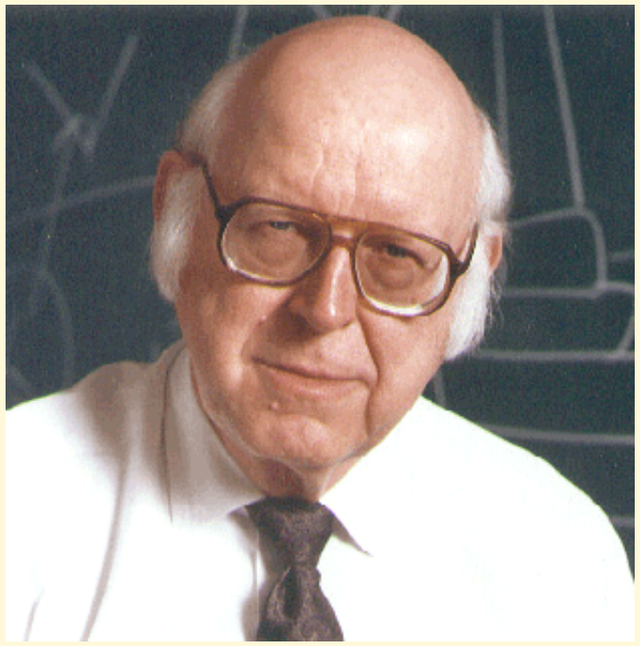
Los matemáticos estaban interesados en mecanizar ciertos procesos inteligentes, tales como la demostración de teoremas.

LISP (Antecedentes Históricos)

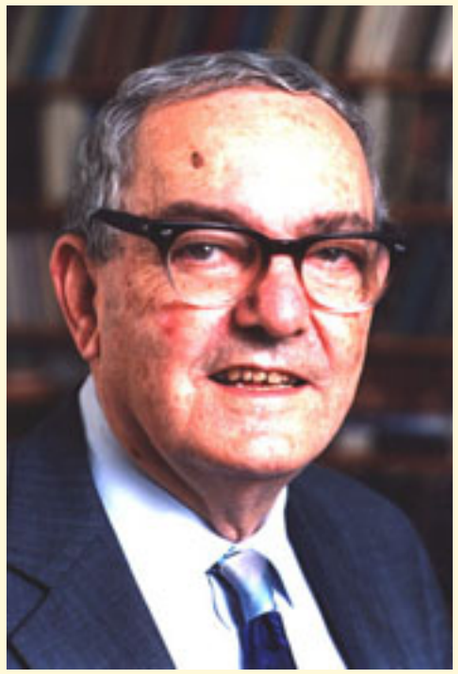
Todos estos esfuerzos arribaron a la misma conclusión: debe desarrollarse algún método para permitir a las computadoras procesar datos simbólicos en listas (colecciones de celdas de memoria no contiguas que se encadenan con apuntadores).

En aquella época, casi todos los procesos computacionales eran con datos numéricos almacenados en estructuras estáticas (arreglos).

LISP (Antecedentes Históricos)



Allen Newell



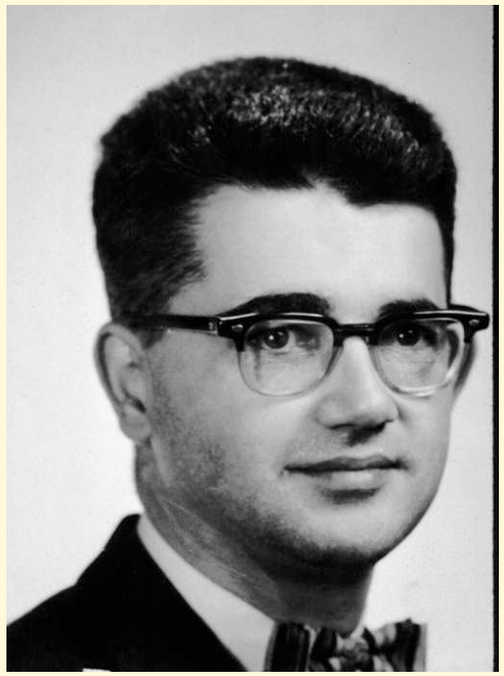
Herbert Simon

El concepto de procesamiento de listas fue desarrollado por Allen Newell, J.Cliff Shaw y Herbert Simon. Este concepto se publicó originalmente en un artículo clásico que describe uno de los primeros programas de IA, el Logic Theorist, así como un lenguaje de programación en el cual podía implementarse.

LISP (Antecedentes Históricos)

El lenguaje, llamado IPL-I (Information Processing Language I), nunca fue implementado. La siguiente versión (el IPL-II), se implementó en una computadora Johniac, de la Rand Corporation.

LISP (Antecedentes Históricos)



John McCarthy

Marvin Minsky

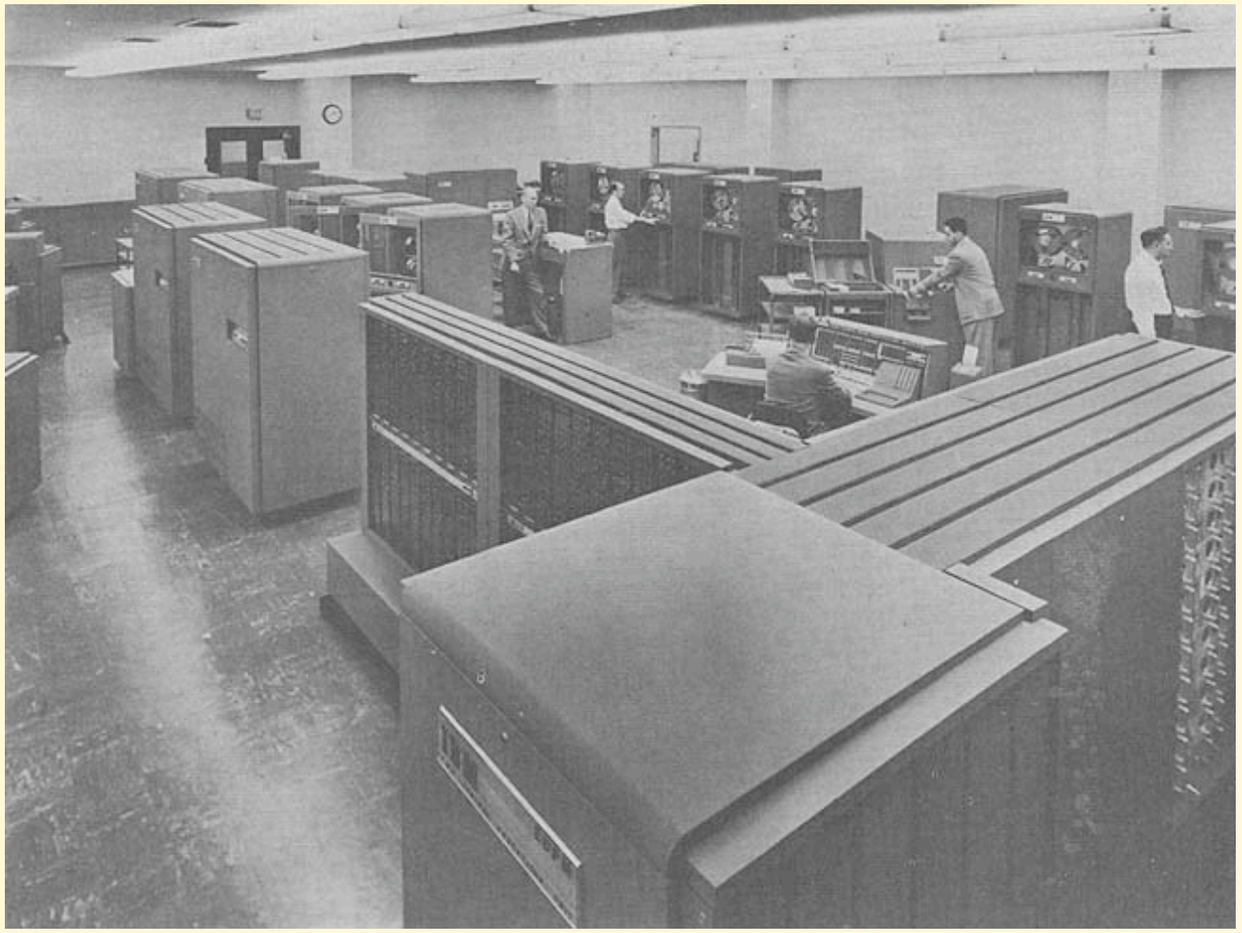
Cuando McCarthy regresó al MIT en el otoño de 1958, él y Marvin Minsky formaron el “MIT AI Project”, con patrocinio del Laboratorio de Investigación en Electrónica. El primer esfuerzo importante del proyecto fue producir un sistema para procesamiento de listas.

LISP (Antecedentes Históricos)

Este sistema para procesamiento de listas se usó inicialmente para implementar un programa propuesto por McCarthy, llamado el “Advice Taker”. Esta aplicación se volvió la mayor motivación para desarrollar el lenguaje de programación LISP.

En el otoño de 1958, comenzó el desarrollo de LISP con un conjunto de subrutinas primitivas para manejo de listas, que luego formarían parte del ambiente de ejecución del lenguaje. La intención original era desarrollar un compilador como FORTRAN.

LISP (Antecedentes Históricos)



Por lo tanto, para adquirir experiencia en la generación de código, se “compilaron” (o sea, se tradujeron a ensamblador) a mano diversos programas en LISP. La versión original de LISP (List Processor) se desarrolló en la IBM 704, y es uno de los lenguajes de programación más antiguos que todavía sigue en uso.

LISP (Antecedentes Históricos)

McCarthy se dio cuenta de que las funciones recursivas para procesamiento de listas acompañadas de expresiones condiciones, formaban una base más fácil de entender para la teoría de la computación que otros formalismos tales como las máquinas de Turing.

En un artículo (ahora clásico) de 1960 titulado “Recursive Functions of Symbolic Expressions and Their Computation by Machine”, McCarthy presentó sus ideas a este respecto.

Cuando McCarthy exploraba algunos de los alcances de su intérprete de LISP, se dio cuenta de que era posible escribir una función universal de LISP la cual podría interpretar cualquier otra función del lenguaje usando el concepto de recursividad.

Puesto que LISP manipula únicamente listas, el poder escribir una función universal requería desarrollar una forma de representar programas en LISP como listas.

LISP (Antecedentes Históricos)

Por ejemplo, la invocación siguiente:

```
f [x+y; u*z]
```

se representaría mediante una lista cuyo primer elemento es "f" y cuyos segundo y tercer elemento serían listas representando "x+y" y "u*z".

En LISP, esta lista se escribe como:

```
(f (+ x y) (* u z))
```

A esta notación similar a la de Algol (p.ej., f[x+y; u*z]) se les llama expresiones-M (por "metalenguaje"), y a la notación de lista se le llama expresiones-S (por lenguaje "simbólico")

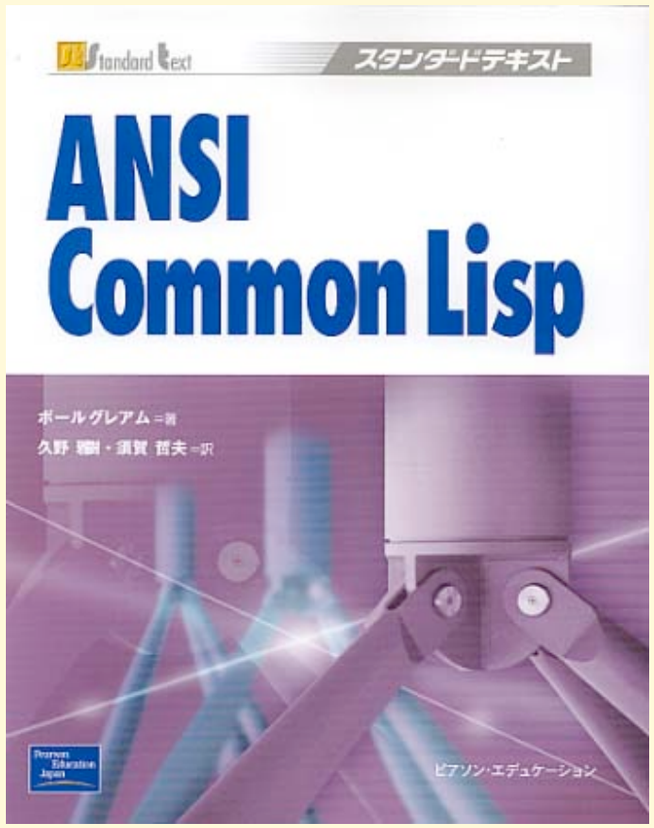
LISP (Antecedentes Históricos)

Una vez que se diseñó la representación de las listas y que se escribió la función universal, uno de los miembros del proyecto se percató de que, en efecto, lo que resultó fue un intérprete del mismo lenguaje.

Por lo tanto, tradujo la función universal a ensamblador y la ligó a las subrutinas para manejo de listas. Así nació la primera implementación de LISP.

Esta implementación requería que los programas se escribieran en forma de expresiones-S, pero esto se vio como un inconveniente temporal. El sistema de LISP 2 (similar a Algol) que se estaba desarrollando en aquel entonces permitiría el uso de expresiones-M.

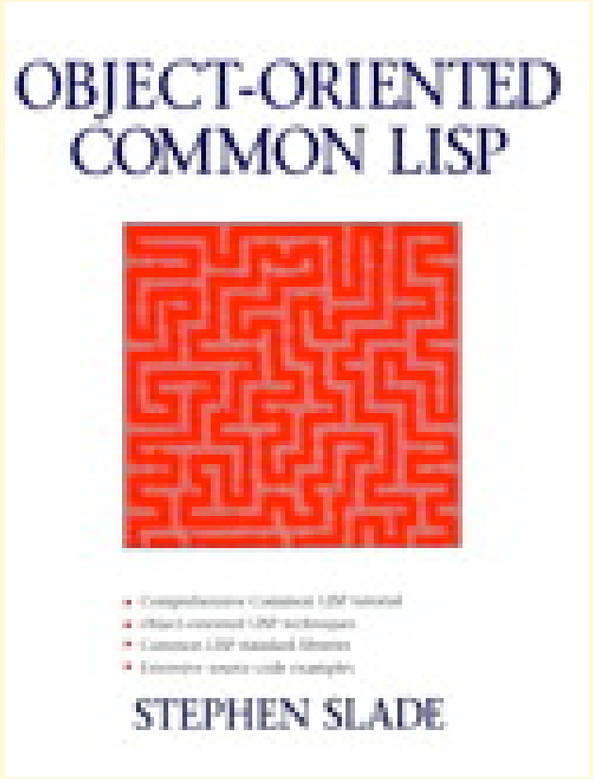
LISP (Antecedentes Históricos)



Sin embargo, este sistema nunca se terminó, y los programadores de LISP, hasta la fecha, siguen escribiendo sus programas usando expresiones-S. Aunque esto fue una cuestión incidental, irónicamente, el uso de expresiones-S se considera como una de las mayores ventajas de LISP.

LISP sigue siendo un lenguaje popular hasta nuestros días, y existen incontables versiones de este lenguaje para todo tipo de plataformas. Un comité ANSI definió un estándar para el denominado COMMON LISP hace varios años.

LISP (Antecedentes Históricos)



El uso de LISP en inteligencia artificial sigue siendo muy extensivo, y se le considera como una piedra angular en el procesamiento en lenguaje natural y otras aplicaciones de IA.

En sus orígenes, sin embargo, el LISP fue severamente criticado por su ineficiencia, que lo hacía impráctico para aplicaciones del mundo real. Sin embargo, con los años se han desarrollado diversos compiladores para el lenguaje, lo que lo ha vuelto más atractivo para aplicaciones comerciales.

LISP (Antecedentes Históricos)

Gerald Jay Sussman

En 1975, Gerald Jay Sussman y Guy Lewis Steele, Jr., eran estudiantes de posgrado en el MIT, interesados en la teoría de actores desarrollada por Carl Hewitt. El modelo de Hewitt estaba fuertemente influenciado por Smalltalk, y los actores eran muy similares a los objetos de este lenguaje.

LISP (Antecedentes Históricos)

Los actores eran capaces de comunicarse entre sí a través de mensajes y contaban con una identidad.

Para realizar sus experimentos, Sussman y Steele decidieron desarrollar un dialecto de LISP que tuviera reglas de entorno estático (siguiendo el modelo de Algol).

LISP (Antecedentes Históricos)

El resultado fue un mini-intérprete al que denominaron “Schemer”. Sin embargo, debido a que el sistema operativo ITS de la máquina donde desarrollaron este intérprete limitaba los nombres de los archivos a seis caracteres, decidieron reducir este apelativo a “Scheme”.

LISP (Antecedentes Históricos)

Con la idea en mente de tener una herramienta más útil para la IA, Sussman y Steele extendieron este lenguaje un poco más y posteriormente publicaron una serie de artículos seminales en los que explicaron la forma en que Scheme era capaz de soportar todos los paradigmas de la época: funcional, imperativo y orientado a objetos.

Desde entonces, Scheme ha sido revisado varias veces, y su uso se ha extendido a través de un sinnúmero de universidades en todo el mundo.

Aspectos de Diseño

LISP usa una sintaxis basada totalmente en paréntesis, y no tiene necesidad de las complejas reglas de precedencia que suelen requerirse en la mayor parte de los lenguajes de programación convencionales.

Esto permite a los estudiantes de este lenguaje concentrarse en las estructuras de datos y no en la sintaxis del lenguaje, como suele ocurrir cuando se programa en otro tipo de paradigmas.

Aspectos de Diseño

LISP usa notación Polaca (llamada también prefija), de acuerdo a la cual un operador debe escribirse antes de sus operandos. Por ejemplo:

```
(+ 3 4)
```

Debe destacarse, sin embargo, que la notación de LISP es un poco más flexible que la notación prefija tradicional, ya que expresiones tales como:

```
(+ 3 4 5 19 2)
```

son perfectamente válidas.

Aspectos de Diseño

Las expresiones condicionales son similares al “if..then..else”:

```
(cond
      ((null x) 0
      ((eq (car x) (car y)) (f (car x)))
      (t (g (car y))) )
```

En este caso, se checa primero si la lista “x” es nula (o sea, si está vacía). Si ese es el caso, la función regresa `()`. De lo contrario, checamos si el “`car`” (o sea, el primer elemento) de la lista es igual a `(car y)`. Si ese es el caso, entonces regresamos `(f (car x))`. De lo contrario, la lista después de la parte “`t`” (true) se ejecuta, lo que significa que se invoca `(g (car y))`.

Aspectos de Diseño

LISP es usualmente un lenguaje interpretado. La mayor parte de los sistemas de LISP de la actualidad son intérpretes interactivos, aunque existen también varios compiladores. De hecho, cabe destacar que hay al menos una computadora cuya arquitectura se inspiró en este lenguaje de programación.

El único constructor de estructuras de datos en LISP es la lista. Puesto que una de las metas de LISP fue el permitir la computación con datos simbólicos, se le permite al programador manipular listas de datos. Algunas versiones de LISP permiten el uso de registros y arreglos, pero el LISP puro sólo lidia con listas.

Aspectos de Diseño

Esto ilustra el Principio de la Simplicidad, ya que el programador no tiene que elegir de entre varios constructores, ya que sólo hay uno disponible.

Las listas son, además, muy fáciles de construir:

```
(list ‘a ‘b ‘c ‘d) ==> ‘(a b c d)
```

Todos los objetos del lenguaje son ciudadanos de primera clase. Esto significa que todos sus objetos pueden pasarse como argumentos, se pueden manipular dentro de una estructura de datos, pueden ser regresados por una función y básicamente nunca “mueren” (hasta que la recolección de basura se hace cargo de ellos).

Aspectos de Diseño

Las aplicaciones de funciones y listas son muy similares. Aun los programas en LISP son listas. Bajo la mayor parte de las circunstancias, una expresión-S se interpreta como una aplicación de una función, lo que significa que se evalúan sus argumentos y se invoca la función.

El hecho de que LISP represente tanto los programas como los datos de la misma manera es de una enorme importancia (y representa una característica casi única en los lenguajes de programación). Esta flexibilidad hace que sea extremadamente fácil escribir un intérprete de LISP en LISP.

Más importante todavía, es el hecho de que esto hace muy conveniente tener un programa en LISP que genere e invoque a otro programa en LISP. También simplifica la escritura de programas en LISP que transformen y manipulen a otros programas en este lenguaje.

Estas características son importantes para las aplicaciones de inteligencia artificial y otros ambientes avanzados de desarrollo de software.

Aspectos de Diseño

Funciones tales como “`eq`” son denominadas “funciones puras” o simplemente “funciones”, debido a que no tienen otro efecto más que el de calcular un valor. Sin embargo, algunas funciones de LISP son pseudo-funciones (o “procedimientos”), porque tienen efectos colaterales además de calcular un cierto valor.

Un ejemplo de procedimiento es “`set`”, que asocia un nombre a un valor:

`set` símbolo valor

Esto permite la alteración del valor de una variable dinámica (especial).

Aspectos de Diseño

"set" hace que la variable dinámica "símbolo" tome el valor definido por "valor".
Puesto que "set" se considera una función "impura".

LISP proporciona otro mecanismo para asociar valores con símbolos, llamado "let", el cual no tiene efectos colaterales:

```
(let ((c 4))
```

Aspectos de Diseño

Otra pseudo-función importante es “defun”, que permite definir una función:

```
(defun fibonacci (n)
(if (or (= n 0) (= n 1))
1
(+ (fibonacci (- n 1))
    (fibonacci (- n 2)))))
```

Aspectos de Diseño

Los dialectos de LISP difieren en pequeños detalles entre sí. En particular, la aplicación de funciones usada para definir funciones es diferente en muchos dialectos, aunque las formas antes presentadas corresponden a COMMON LISP, que es la versión estándar del lenguaje.

Aspectos de Diseño

Las estructuras de datos en LISP pueden clasificarse en dos grupos:

- Primitivas
- Constructores

El constructor es la lista, la cual permite estructuras complicadas a partir de estructuras simples.

Las estructuras primitivas se llaman “átomos” en LISP (“átomo” en griego significa indivisible). Hay al menos dos tipos de átomos: numéricos y no numéricos.

Aspectos de Diseño

Los <u>átomos numéricos</u> tienen la sintaxis de los números (o sea, son dígitos, posiblemente con un punto decimal).

Ejemplos de átomos numéricos:

```
+, -, *, /, sub1, add1, max, min, =, <, >,
sqrt, log, expt, abs, sin, cos, tan, asin, acos, etc.
```

Los <u>átomos no numéricos</u> son cadenas de caracteres que se pretendía originalmente que representaran palabras o símbolos.

Ejemplos de átomos no numéricos:

```
Carlos, x, hola, mi-atomo-unico, etc.
```

Aspectos de Diseño

Con unas pocas excepciones, las únicas operaciones que pueden efectuarse sobre átomos no numéricos son comparaciones de igualdad y desigualdad. Esto se hace con la función "eq":

```
(eq x y)
```

"eq" regresa "t" (cierto) si "x" es igual a "y", y "nil" (falso) de lo contrario.

Algunas versiones de LISP proporcionan tipos adicionales de átomos, tales como las cadenas. En estos casos, se proporcionan también operaciones especiales para manipular estos objetos.

Ejemplos:

```
string=, string<, string>, string <=, string>=, make-
  string, string-upcase, string-downcase, etc.
```

Aspectos de Diseño

El método de estructuración de datos que proporciona LISP es llamado “list”. Las listas se escriben en forma de expresiones-S, rodeando con paréntesis los elementos de las mismas, los cuales se separan mediante espacios en blanco. Las listas pueden tener cero, uno o más elementos, lo que sigue el Principio Cero-Uno-Infinito.

Los elementos de las listas pueden a su vez ser listas también, de forma que este mismo principio se satisface para el nivel de anidamiento de las listas.

Aspectos de Diseño

Por razones históricas, se considera a la lista vacía `'()` como equivalente al átomo `'nil`. Es decir:

```
(eq '() nil)
(null '())
```

regresan, en ambos casos, cierto.

Por esta razón, a la lista vacía se le suele denominar “null list”. Con la excepción de la lista vacía, todas las listas son no atómicas (es decir, no son átomos). Algunas veces se les denomina valores de datos compuestos.

Aspectos de Diseño

Puede averiguarse si algo es un átomo usando el predicado “atom”:

```
(atom 'c)
t
(atom '(()) )
nil
```

Aspectos de Diseño

Puesto que las listas son el método principal de estructuramiento de datos en LISP, el lenguaje proporciona también una serie de constructores (es decir, operaciones usadas para construir listas) y selectores (o sea, operaciones usadas para separar listas).

LISP tiene un constructor (cons) y dos selectores (car y cdr). El primer elemento de una lista se selecciona usando “car”:

```
(car '(mi nombre es Abdiel))

mi
```

Aspectos de Diseño

El primer elemento de una lista puede ser un átomo o una lista, y “car” lo retornará, sin importar su naturaleza:

```
(car '((mi) (nombre) (es) (Abdiel)))
'(mi)
```

Debe advertirse que el primer argumento de “car” debe ser una lista no vacía (de lo contrario, no podría tener un primer elemento y LISP retornaría un mensaje de error). Además, “car” puede regresar un átomo o una lista, dependiendo del argumento que sea el primer elemento de la lista a la que se aplique.

Aspectos de Diseño

La función “cdr” regresa toda la lista, excepto su primer elemento. Por ejemplo:

```
(cdr '(mi nombre es Abdiel))
'(nombre es Abdiel)
```

También “cdr” requiere que su argumento sea una lista no vacía (de lo contrario no podríamos remover su primer elemento, y LISP proporcionaría un mensaje de error). Sin embargo, a diferencia de “car”, “cdr” SIEMPRE regresa una lista, aunque ésta podría ser la lista vacía en caso de que el argumento tenga un solo elemento.

Ejemplo:

```
(cdr '(Abdiel))
'()
```

Aspectos de Diseño

Es importante hacer notar que “car” y “cdr” son funciones puras. Es decir, no modifican su argumento. Puede considerarse que durante su operación utilizan una copia de la lista que se les pasa como argumento. Por ejemplo, “cdr” no borra realmente el primer elemento de una lista sino que regresa una lista idéntica a su argumento, si bien ésta carece del primer elemento.

El único constructor de LISP, “cons”, agrega un nuevo elemento al inicio de una lista.

Por ejemplo:

```
(cons 'mi '(nombre es Abdiel))

'(mi nombre es Abdiel)
```

Aspectos de Diseño

Advierta que “cons” es realmente el inverso de “car” y “cdr”:

```
(car '(my name is Abdiel))   'my
(cdr '(my name is Abdiel))   '(name is Abdiel)
(cons 'my '(name is Abdiel)) '(my name is Abdiel)
```

Aspectos de Diseño

Por lo tanto, cualquier lista que podamos construir, también la podemos separar y viceversa.

Al igual que “car” y “cdr”, “cons” es una función pura.

Esto significa que realmente no agrega nuevos elementos al inicio de su segundo argumento, sino que actúa como si hubiese construido una lista completamente nueva cuyo primer elemento es el argumento pasado a “cons” y el resto de esos elementos se copian de su segundo argumento.

Aspectos de Diseño

Las listas suelen construirse recursivamente. Si hacemos:

```
(cons '(a b) '(c d))
```

el resultado es:

```
'((a b) c d))
```

Esto es debido a que el primer elemento que se agregó es una lista y no un átomo. Sin embargo, probablemente lo que el programador quería hacer era generar la lista

```
‘(a b c d)
```

Aspectos de Diseño

Para hacer eso, necesitaríamos una función llamada “append”:

```
(append '(a b) '(c d))

'(a b c d)
```

Aspectos de Diseño

La definición de “append” es muy sencilla:

```
(defun append (list1 list2)
  (cond
    ((null list1) list2)
    (t (cons (car list1) (append (cdr list1) list2)) )) )
```

Esta solución es muy compacta y elegante; hace uso de recursividad de las primitivas para manejo de listas que vimos anteriormente.

Ejercicio A

- Evalúe las siguientes formas:
  - `(first ‘(p h w))`
  - `(rest ‘(b k p h))`
  - `(first (rest ‘((a b) n (c d))))`
  - `(first (rest (first (rest ‘((a b) (c d) (e f))))))`

Ejercicio B

- Escriba secuencias car y cdr para extraer el símbolo pera de cada una de las siguientes expresiones
  - `(manzana naranja pera uva)`
  - `(((manzana (naranja) (pera) (uva)))`