Lenguajes de Programación

Dr. Carlos A. Coello Coello
Departamento de Computación
CINVESTAV-IPN
ccoello@cs.cinvestav.mx

Representación de los Registros de Activación

- Se usa un sistema de coordenadas en dos dimensiones para identificar instrucciones:
  - *Un apuntador al objeto* identifica el método-objeto que contiene todas las instrucciones de un método.
  - *Un desplazamiento relativo* que identifica la instrucción en particular dentro del par método-objeto.

Representación de los Registros de Activación

- Este direccionamiento en coordenadas bidimensionales es necesario porque el direccionamiento de instrucciones pasa a través del manejador de almacenamiento.

Representación de los Registros de Activación

- Esto último indica que nos adherimos al **Principio de Ocultamiento de Información**.

- La parte del ambiente debe proporcionar acceso tanto al ambiente local como al no local.

Representación de los Registros de Activación

- El ambiente local incluye espacio para los parámetros del método y para las variables temporales.
- Esta parte del registro de activación debe proporcionar espacio para variables temporales ocultas tales como los resultados intermedios de las expresiones.

Representación de los Registros de Activación

- El ambiente no local incluye a todas las otras variables visibles (es decir, las variables instanciadas y las variables de la clase).
- Las variables instanciadas se almacenan en la representación del objeto que recibió el mensaje.
- Por lo tanto, un simple apuntador a este objeto las vuelve accesibles.

Representación de los Registros de Activación

- La representación del objeto contiene un apuntador a la clase de la cual es una instancia el objeto; por lo tanto, las variables de la clase también son accesibles vía la referencia al objeto.

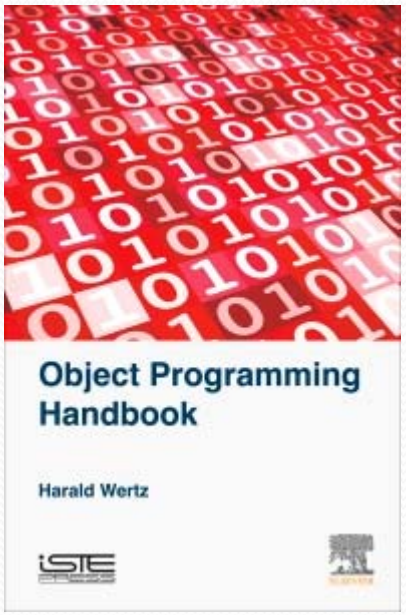
- Finalmente, puesto que la representación de la clase también contiene un apuntador a su superclase, las variables de la superclase son también accesibles.

Representación de los Registros de Activación

- Adviertan que el enfoque usado por Smalltalk para acceder ambientes no locales es muy similar a la cadena estática que estudiamos anteriormente.

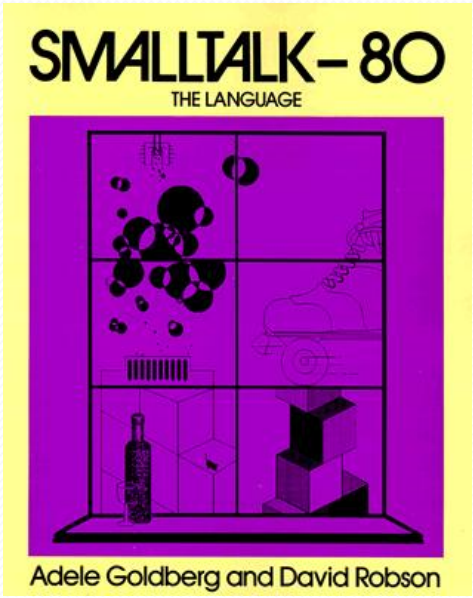
- En esa técnica, la cadena estática nos llevaba del ambiente activo más anidado al ambiente activo más exterior.

Representación de los Registros de Activación



- En el caso de Smalltalk, la cadena estática nos lleva del método activo al objeto que recibió el mensaje, y de ahí hacia arriba, a través de la jerarquía de clases, hasta terminar en la clase "**object**" (la metaclase).

Representación de los Registros de Activación



- Al igual que en los lenguajes estructurados, el acceso a las variables requiere conocer la distancia estática a la variable, de manera que se salte ese número de ligas de la cadena estática para llegar al ambiente que define la variable.

Paso y Regreso de Mensajes

- Cuando se envía un mensaje a un objeto, deben efectuarse los siguientes pasos:

  1) Crear un registro de activación para el receptor ("*callee*", en inglés).

Paso y Regreso de Mensajes

2) Identificar el método que está siendo invocado extrayendo el patrón (*template*) del mensaje y buscándolo en el diccionario de mensajes de la clase del objeto receptor o en sus superclases.

3) Transmitir los parámetros al registro de activación del receptor.

Paso y Regreso de Mensajes

4) Suspender al remitente (invocador) almacenando su estado en su registro de activación

5) Establecer una ruta (liga dinámica) del receptor al remitente, y establecer como activo el registro de activación del receptor.

Paso y Regreso de Mensajes

- Como es de esperarse, regresar de un método debe revertir este proceso:

  1) Transmitir el objeto retornado (si hay alguno) del receptor al remitente

  2) Continuar la ejecución del remitente, restaurando su estado a partir de su registro de activación

Paso y Regreso de Mensajes

- Advierta que los registros de activación no son liberados de la memoria explícitamente, sino que se usa recolección de basura.

- El Manejador de Almacenamiento crea el espacio libre y reclama espacio que un objeto ya no está usando de vez en cuando.

Paso y Regreso de Mensajes

- Este enfoque es muy diferente del usado en los lenguajes estructurados, en los cuales los registros de activación ocupan posiciones contiguas en una pila.
- El enfoque de Smalltalk es menos eficiente, pero es más regular y más simple.

Paso y Regreso de Mensajes

- La otra razón para no usar una pila tiene que ver con la concurrencia.
- Las pilas sólo son apropiadas si estamos lidiando con algo que siga una estructura LIFO (*Last-In, First-Out*).

Paso y Regreso de Mensajes

- Los procedimientos en un lenguaje procedural siguen esta estructura, pero los procedimientos en un lenguaje concurrente no.
- Por lo tanto, necesitamos un modelo diferente para lidiar con las tareas concurrentes en Smalltalk.

Puntos Finales sobre Smalltalk

- Smalltalk es un buen ejemplo de lo que puede hacerse con un lenguaje pequeño, simple y extremadamente regular.

- Smalltalk ha sido utilizado en un sinnúmero de aplicaciones: juegos, simulaciones, gráficos, inteligencia artificial, etc.

Puntos Finales sobre Smalltalk

- Dado que Smalltalk no tiene tipos, hablar de polimorfismo no es muy adecuado, pero sí podemos mencionar que la sobrecarga se da de manera natural.

- No hay ninguna razón por la que objetos de diferentes clases no puedan responder a un mismo mensaje.

Puntos Finales sobre Smalltalk

- El diseño de Smalltalk ha sido guiado por varios principios muy valiosos: Simplicidad, Regularidad, Abstracción, Seguridad y Ocultamiento de Información.

- En general, ilustra un principio de MacLennan que no habíamos mencionado: el de la Elegancia.

Extensiones Orientadas a Objetos

- Varios lenguajes han sido extendidos para soportar el paradigma de orientación a objetos.

- Por ejemplo: LISP, C y Ada-83.

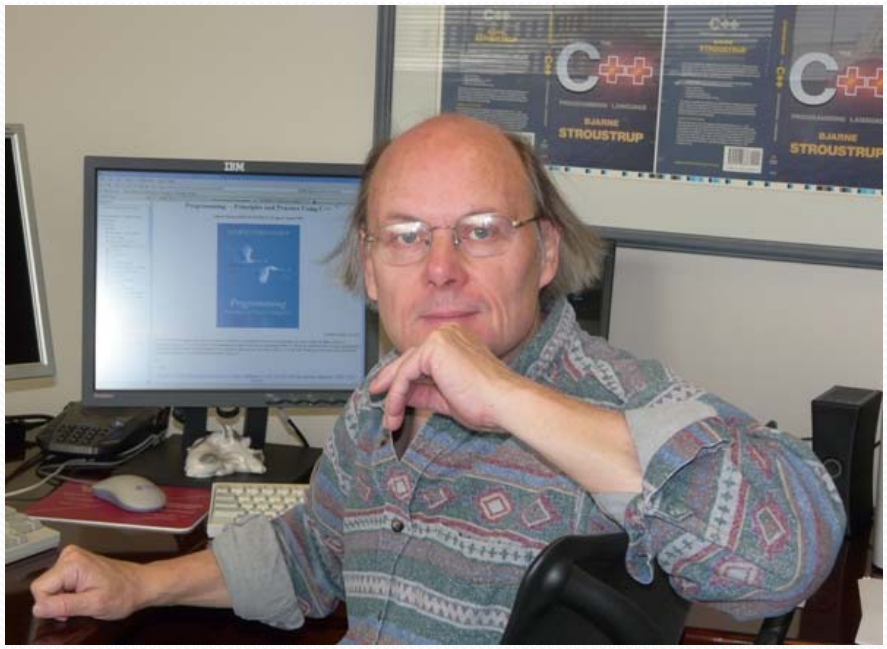
Extensiones Orientadas a Objetos

- Ada 95 posee nuevos mecanismos que hacen más flexible su manejo de clases y tipos.
- De hecho, se manejan clases como tipos especiales (restringidos) que pueden pasarse como parámetros pero no pueden usarse como tipos de variables.

Extensiones Orientadas a Objetos

- Sin embargo, el soporte de orientación a objetos en Ada es realmente muy complejo y costoso.
- Esto se debe a que se pretende agregar un nuevo mecanismo que implica tipos dinámicos sin sacrificar la estructura de un sistema de tipos estáticos.

Extensiones Orientadas a Objetos



- C++ es una extensión orientada a objetos del popular lenguaje C.

- C++ fue desarrollado también en Laboratorios Bell y es, en gran parte, responsabilidad de Bjarne Stroupstrup.

Extensiones Orientadas a Objetos

- Las clases en C++ son una generalización del los registros (records) de C. C++ soporta objetos representados internamente a través de la declaración de clases.
- Nótese, sin embargo, que C++ permite herencia múltiple.
- Esto se debe, en buena medida, a que usa un sistema de tipos estático.

Extensiones Orientadas a Objetos

- Los componentes de una clase en C++ pueden ser de tipo "**public**" (visibles a todo mundo), "**protected**" (visibles en la clase y sus subclases) o "**private**" (visibles sólo dentro de la clase).

- La programación orientada a objetos de C++ depende en buena medida de las funciones virtuales.

Extensiones Orientadas a Objetos

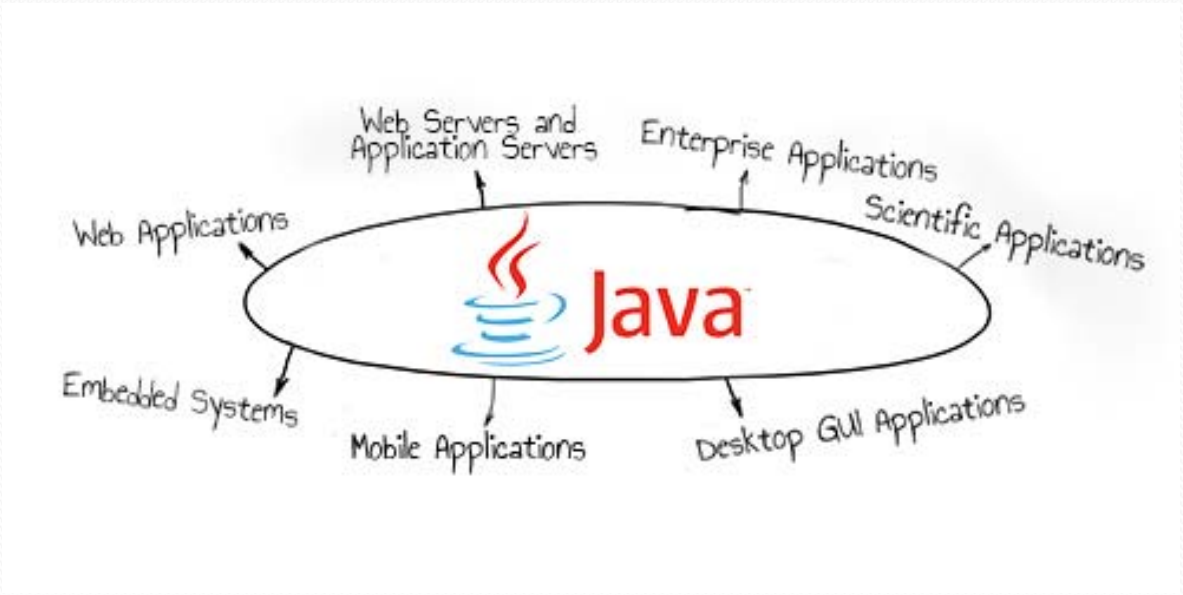
- C++ se vale de los “templates” para implementar un mecanismo similar a los paquetes genéricos de Ada.

- La mayor crítica al C++ se debe a que retiene las características estáticas (de los tipos, sobre todo) de C a fin de poder compilar ambos lenguajes.

Extensiones Orientadas a Objetos

- Java ha sido foco de atención más recientemente y puede verse como una variante del C++.
- En Java se han enfatizado aspectos de seguridad, robustez, portabilidad y asociaciones tardías.
- El enfoque principal del lenguaje es la programación en ambientes de red y sistemas distribuidos.

Extensiones Orientadas a Objetos



- Java omite varias funciones de C++, como por ejemplo sobrecarga de operadores, apuntadores, coerciones excesivas y herencia múltiple (sólo se permite de manera más restringida), pero a cambio agrega otras tales como recolección de basura automática y un chequeo seguro de tipos.

Extensiones Orientadas a Objetos

- CLOS (*Common LISP Object System*) es una extensión bastante popular del LISP que permite que éste soporte orientación a objetos.
- CLOS proporciona objetos representados externamente dado que los métodos se implementan como funciones genéricas.
- Dado que LISP tiene tipos dinámicos, éstos se mantienen en la orientación a objetos.

Extensiones Orientadas a Objetos

- Un detalle interesante de CLOS es que sí soporta herencia múltiple, pero el mecanismo para realizarla dista de lo trivial.

- Las reglas de la herencia múltiple deben expresarse en la forma de un algoritmo a fin de determinar la lista de precedencia de clases que se requiere.

LISP (Antecedentes Históricos)

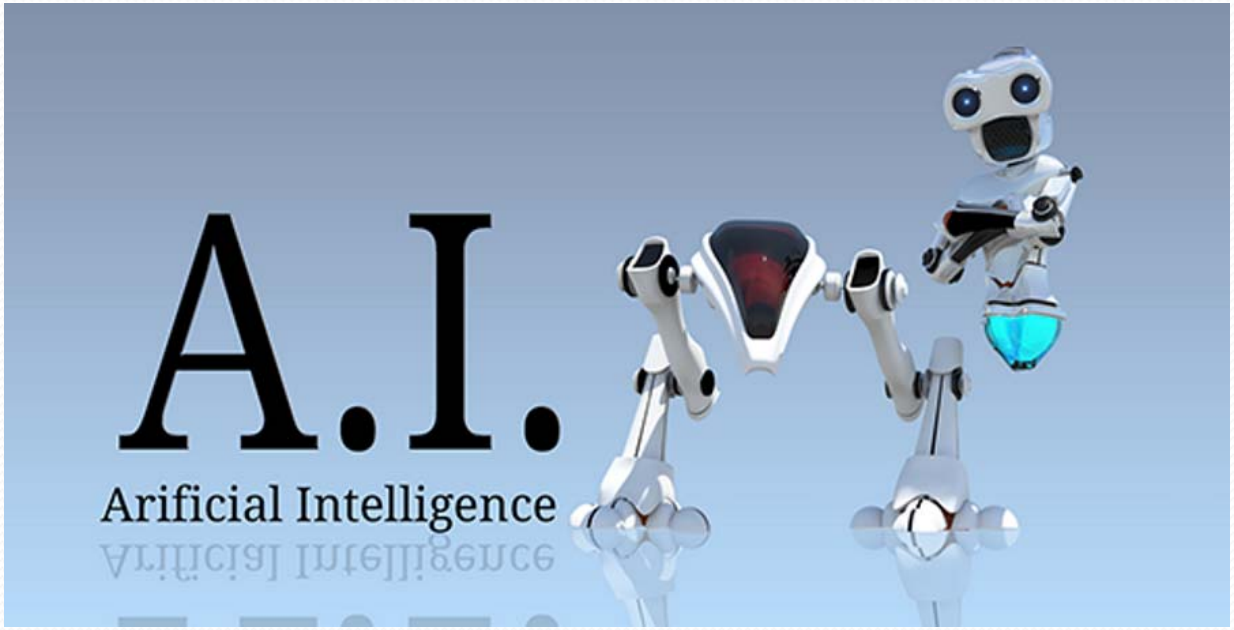
- El interés por la inteligencia artificial (IA) comenzó a mediados de los 1950s en diversos lugares alrededor del mundo.

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

LISP (Antecedentes Históricos)

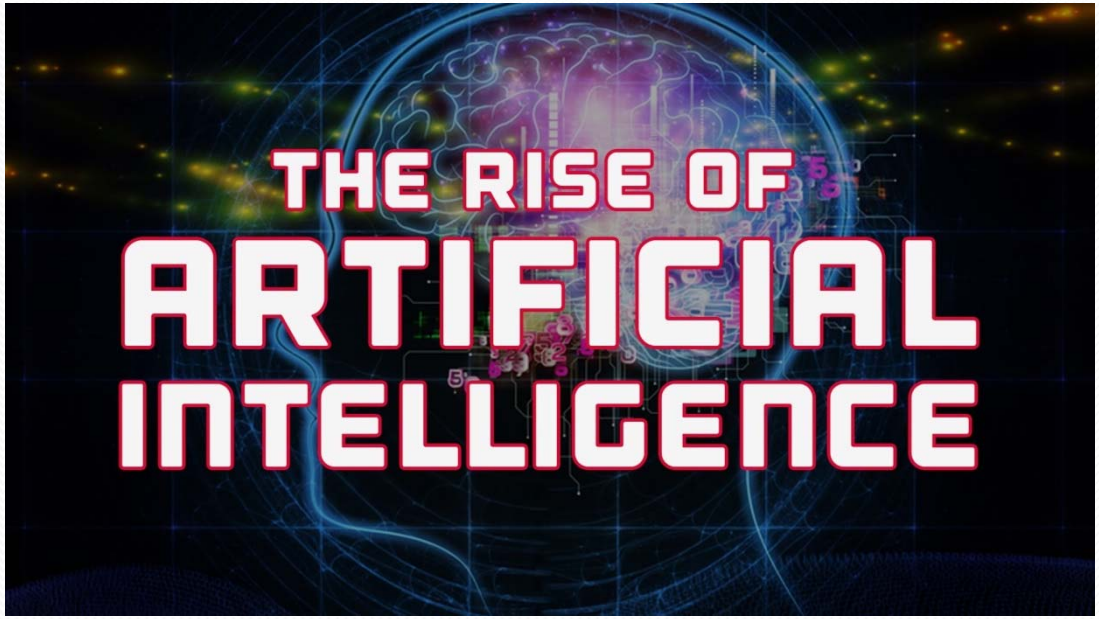
- Los lingüistas estaban interesados en el procesamiento en lenguaje natural.
- Los psicólogos estaban interesados en modelar el almacenamiento y la recuperación de información de los humanos, junto con otros procesos fundamentales del cerebro.

LISP (Antecedentes Históricos)



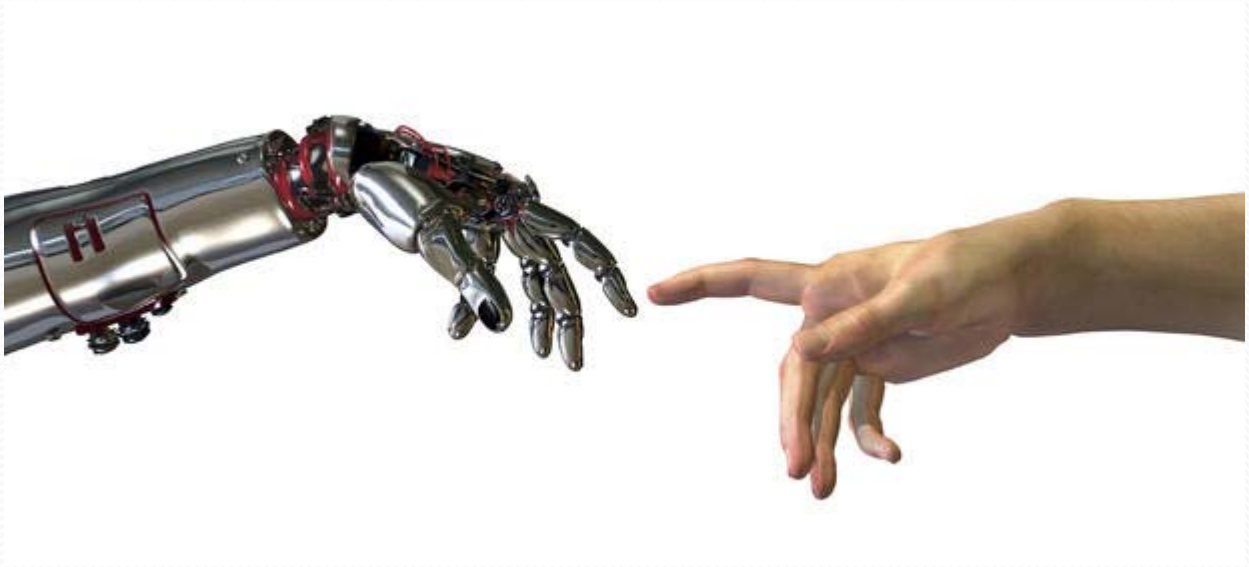
- Los matemáticos estaban interesados en mecanizar ciertos procesos inteligentes, tales como la demostración de teoremas.

LISP (Antecedentes Históricos)



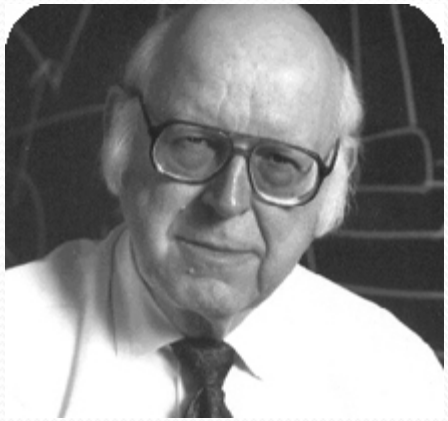
- Todos estos esfuerzos arribaron a la misma conclusión: debe desarrollarse algún método para permitir a las computadoras procesar datos simbólicos en listas (colecciones de celdas de memoria no contiguas que se encadenan con apuntadores).

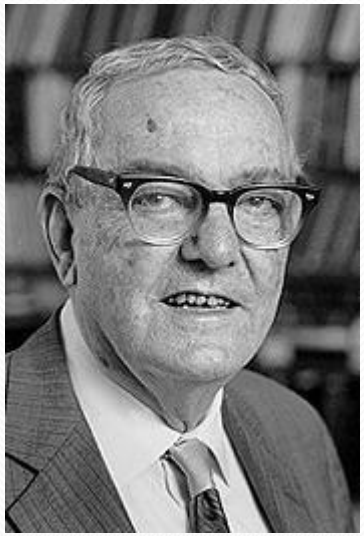
LISP (Antecedentes Históricos)



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

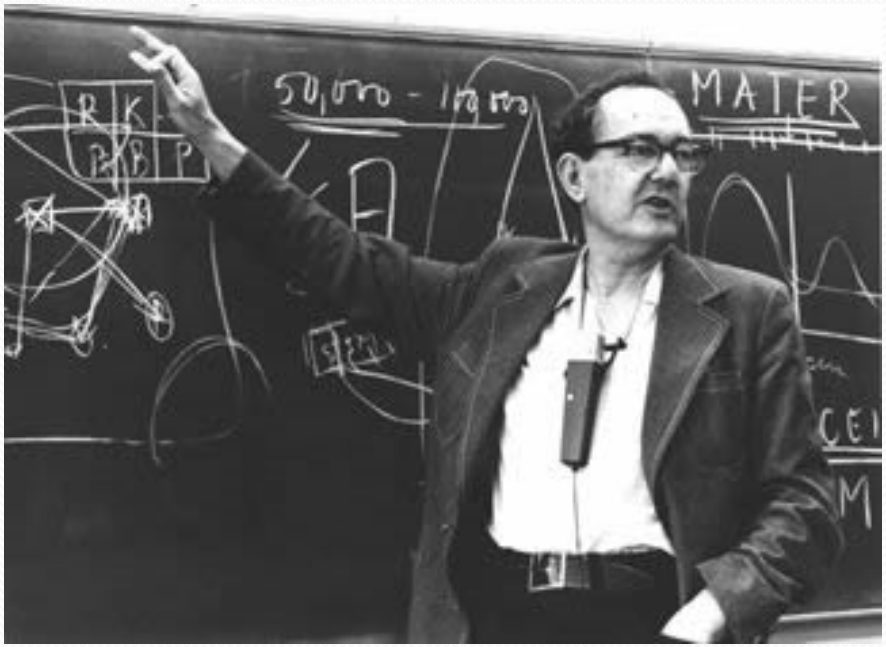
LISP (Antecedentes Históricos)





- El concepto de procesamiento de listas fue desarrollado por Allen Newell, John Clifford Shaw y Herbert Simon.

LISP (Antecedentes Históricos)



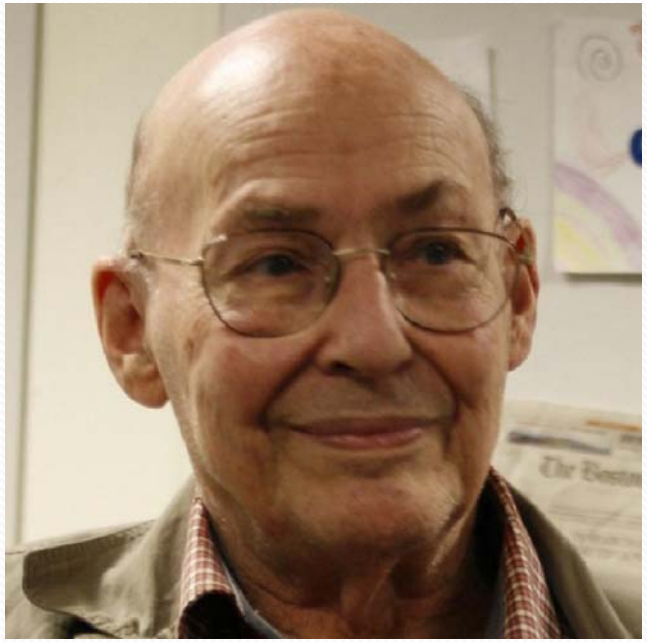
- Este concepto se publicó originalmente en un artículo clásico que describe uno de los primeros programas de IA, el *Logic Theorist*, así como un lenguaje de programación en el cual podía implementarse.

LISP (Antecedentes Históricos)

- El lenguaje, llamado IPL-I (*Information Processing Language I*), nunca fue implementado.

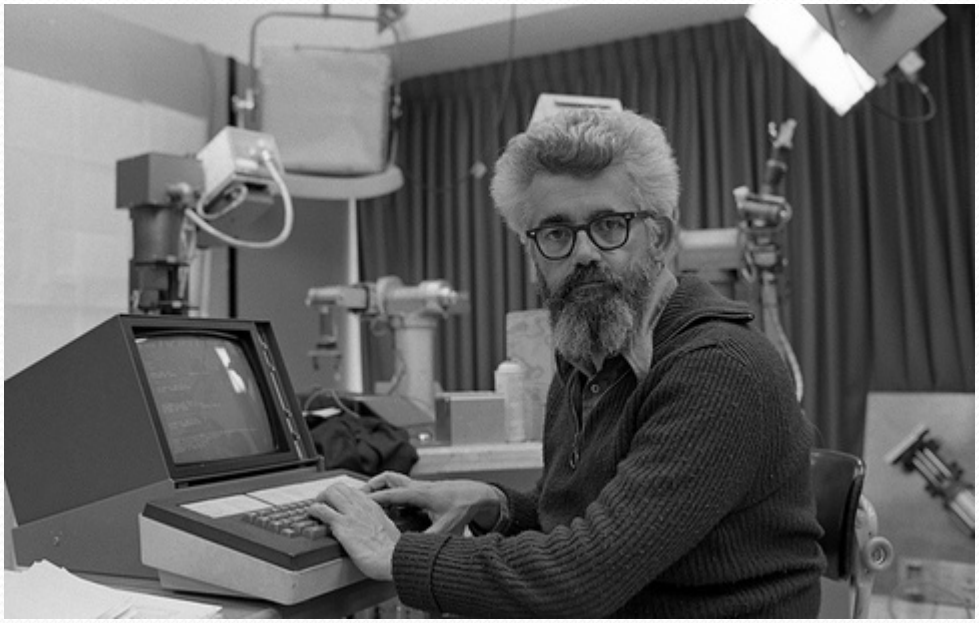
- La siguiente versión (el IPL-II), se implementó en una computadora Johniac, de la *Rand Corporation*.

LISP (Antecedentes Históricos)



- 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.

LISP (Antecedentes Históricos)



- 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”.

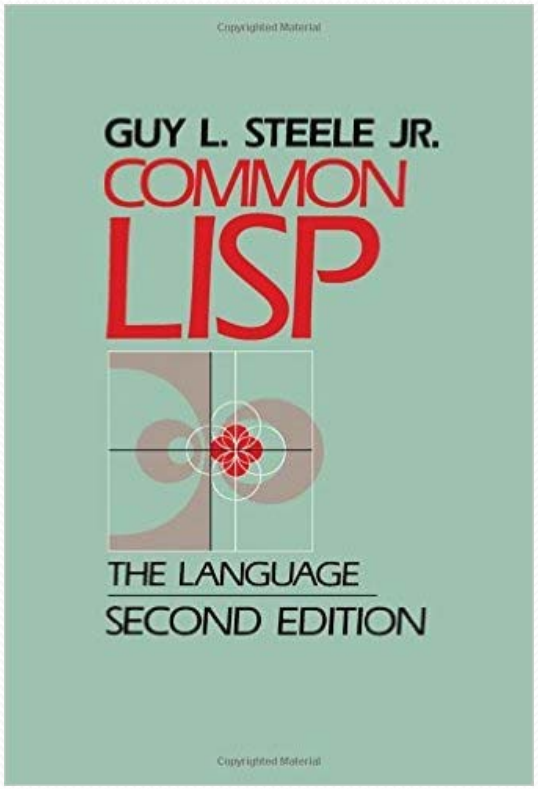
- Esta aplicación se volvió la mayor motivación para desarrollar el lenguaje de programación LISP.

LISP (Antecedentes Históricos)

- 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)

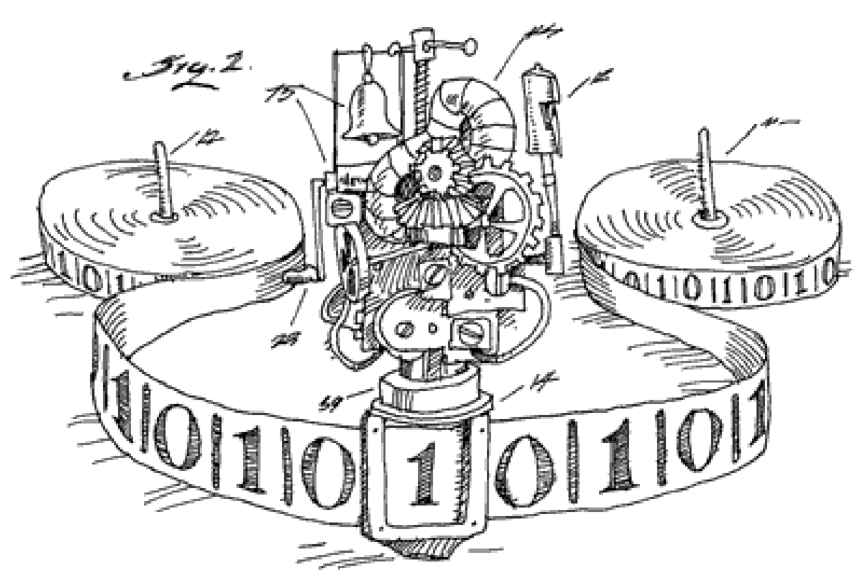


- Por lo tanto, para adquirir experiencia en la generación de código, se "compilaron" (o sea, se tradujeron a ensamblador) a mano diversos programas en LISP.

LISP (Antecedentes Históricos)

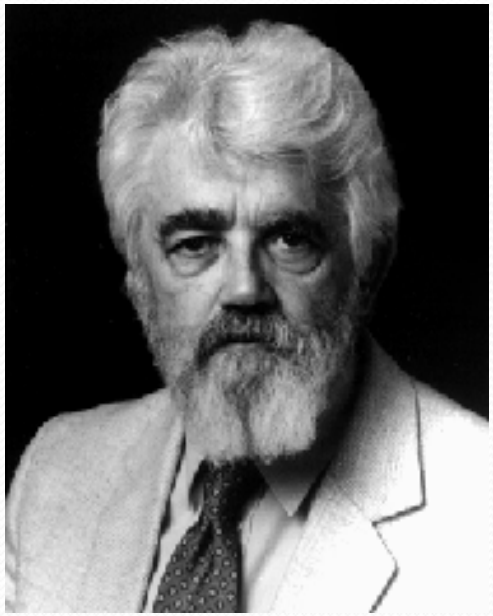
- La versión original de LISP (*List Processor*) se desarrolló en la IBM 704, y es uno de los lenguajes de programación más antiguos que todavía sigue en uso.

LISP (Antecedentes Históricos)



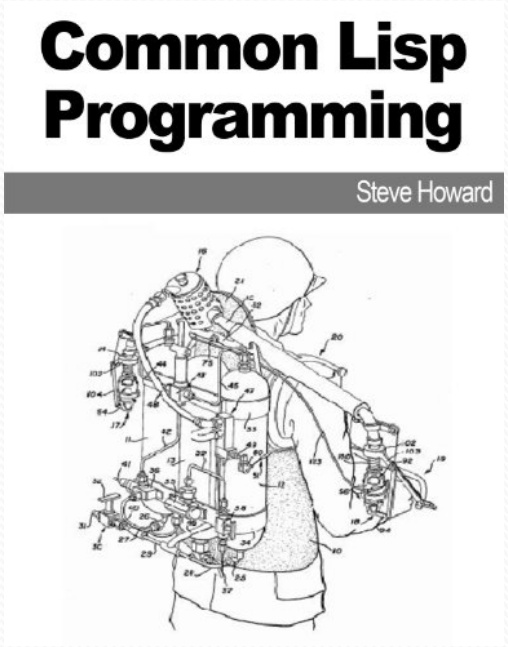
- McCarthy se dio cuenta de que las funciones recursivas para procesamiento de listas acompañadas de expresiones condiciones, formaban una base más fácil de entender para la teoría de la computación que otros formalismos tales como las máquinas de Turing.

LISP (Antecedentes Históricos)



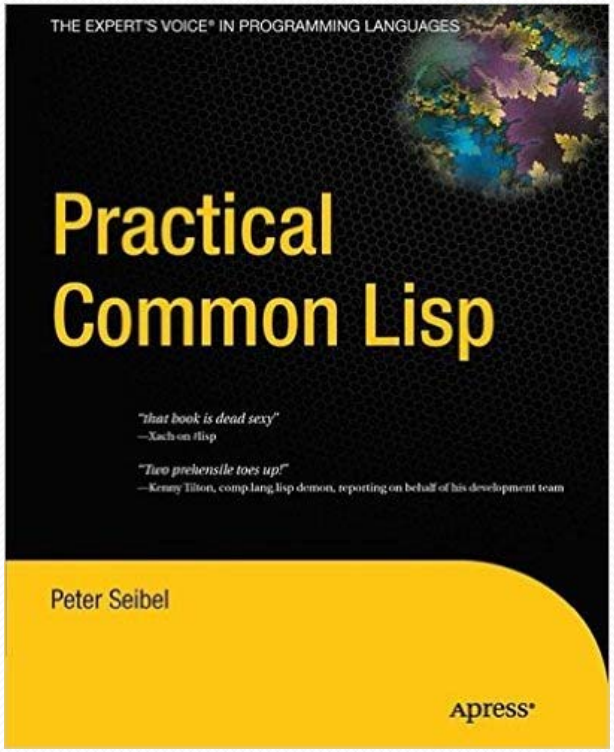
- En un artículo (ahora clásico) de 1960 titulado "*Recursive Functions of Symbolic Expressions and Their Computation by Machine*", McCarthy presentó sus ideas a este respecto.

LISP (Antecedentes Históricos)



- 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.

LISP (Antecedentes Históricos)



- 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”.

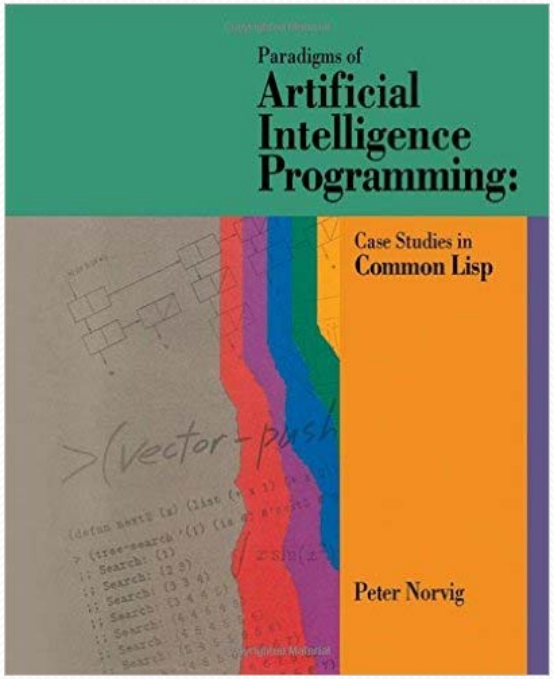
LISP (Antecedentes Históricos)

- En LISP, esta lista se escribe como:

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

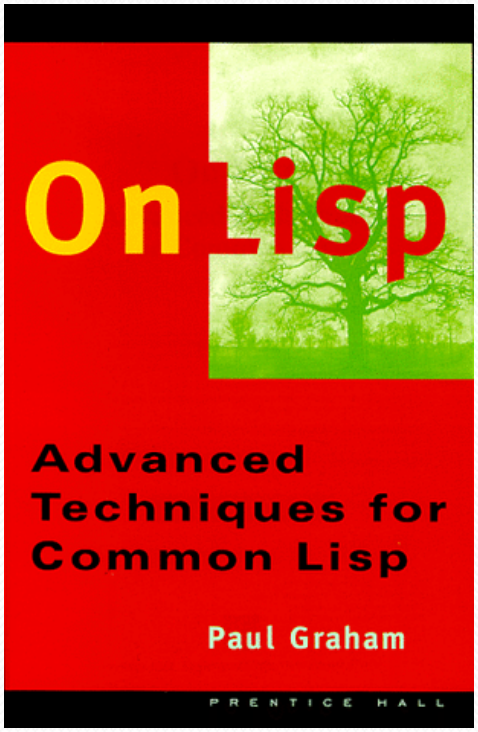
- A esta notación similar a la de Algol (p.ej., f[x+y; u*z]) se les llama expresiones-M (por “metalenguaje”), y a la notación de lista se le llama expresiones-S (por lenguaje “simbólico”)

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.

LISP (Antecedentes Históricos)



- 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.

LISP (Antecedentes Históricos)

- 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 (Antecedentes Históricos)

- 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 en 1987.
- Scheme, que es quizás el dialecto más famoso de LISP, se propuso originalmente en 1975.

LISP (Antecedentes Históricos)

- **Clojure** es una versión de LISP que corre en una máquina virtual de Java y que maneja procesos concurrentes muy bien.

- Varios dialectos de LISP se han usado también para diversas aplicaciones.

- De entre ellos destaca el Emacs Lisp, que viene en el editor Emacs, así como AutoLisp y su versión posterior, Visual Lisp, que se proporcionan en AutoCAD.

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.

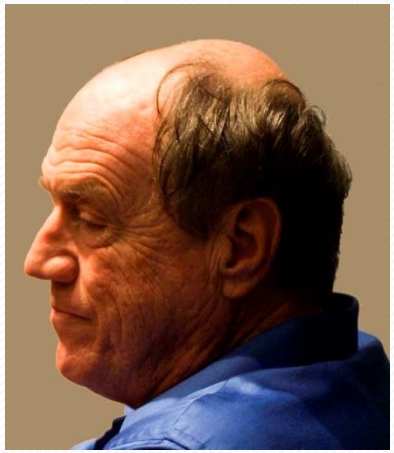
LISP (Antecedentes Históricos)

- 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.

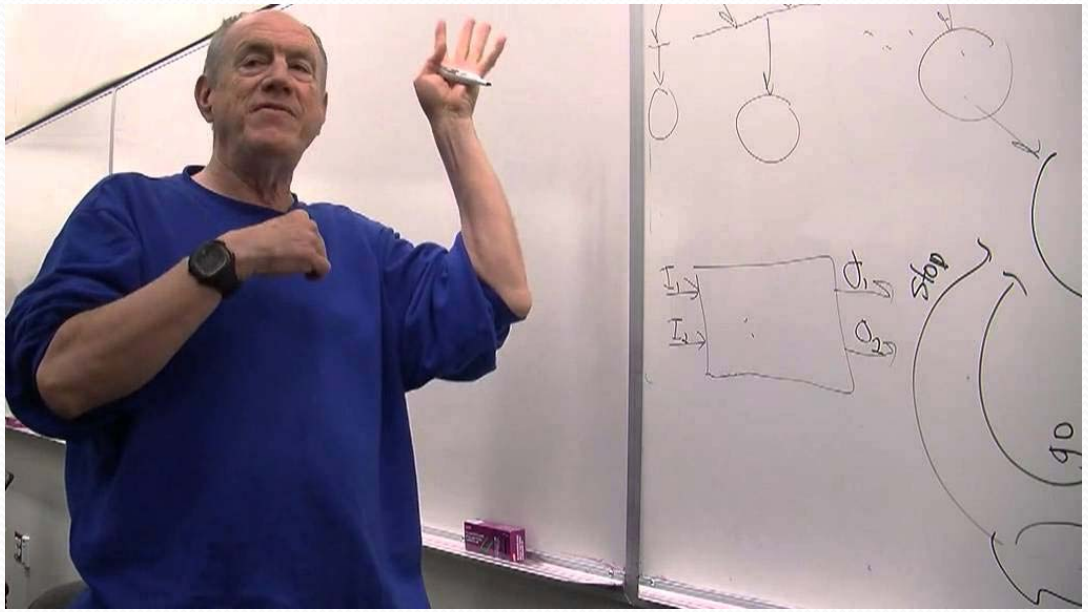
LISP (Antecedentes Históricos)





- 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.

LISP (Antecedentes Históricos)



- 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**".

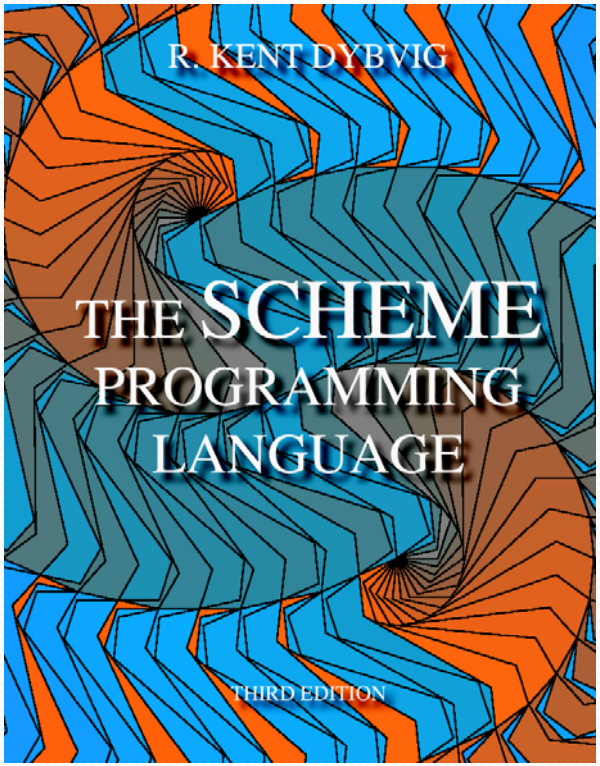
- Sin embargo, debido a que el sistema operativo ITS de la máquina donde desarrollaron este intérprete limitaba los nombres de los archivos a seis caracteres, decidieron reducir este apelativo a "**Scheme**".

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.

- 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.

LISP (Antecedentes Históricos)

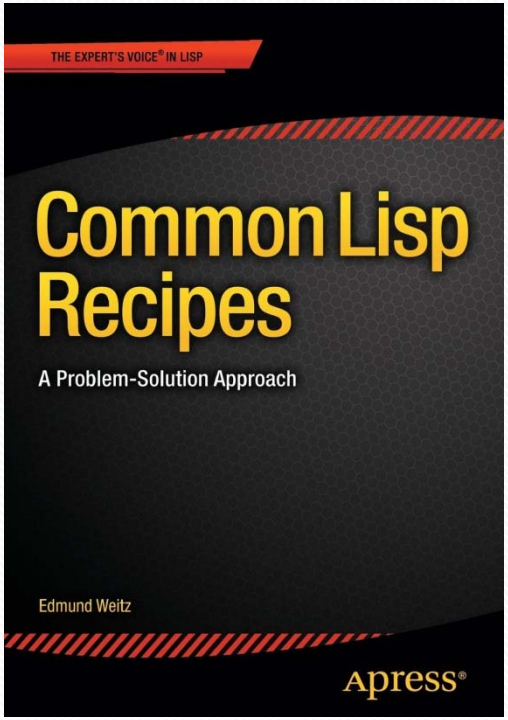


- Desde entonces, Scheme ha sido revisado varias veces, y su uso se ha extendido a través de un sinnúmero de universidades en todo el mundo.

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.

Aspectos de Diseño



- 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)

Aspectos de Diseño

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

(+ 3 4 5 19 2)

- es perfectamente válida.

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))) )
```

Aspectos de Diseño

- 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 (la *LISP Machine*).

Aspectos de Diseño

- 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)

Aspectos de Diseño

- 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.

Aspectos de Diseño

- 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).

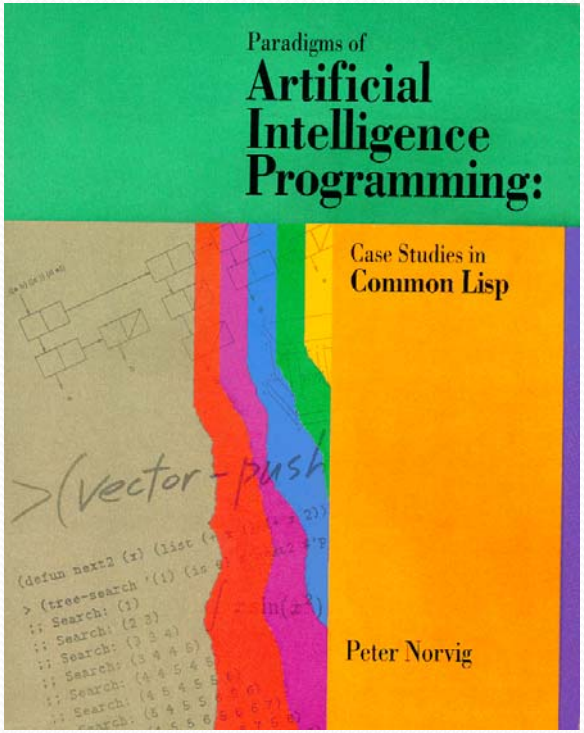
- Esta flexibilidad hace que sea extremadamente fácil escribir un intérprete de LISP en LISP.

Aspectos de Diseño

- 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.

Aspectos de Diseño



- 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.

Aspectos de Diseño

- 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:

1) Primitivas

2) Constructores

Aspectos de Diseño

- 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 **átomos numéricos** tienen la sintaxis de los números (o sea, son dígitos, posiblemente con un punto decimal).

- Los **átomos no numéricos** son cadenas de caracteres que se pretendía originalmente que representaran palabras o símbolos.

Aspectos de Diseño

- Ejemplos de átomos numéricos:

+, -, *, /, sub1, add1, max, min, =, <, >,

sqrt, log, expt, abs, sin, cos, tan, asin, acos, etc.

Aspectos de Diseño

- 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)

Aspectos de Diseño

- "**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.