1. PROGRAMACION DECLARATIVA

1.1 NUEVOS LENGUAJES. ANTIGUA LÓGICA.

1.1.1 Lambda Cálculo de Church y Deducción Natural de Grentzen

A finales del siglo XIX, los logistas formalizaron una noción ideal de prueba. A lo que les llevó no fue otra cosa que un ávido interés de la verdad. Estas abstracciones matemáticas fueron realizadas a mano. Tardaríamos un siglo en darnos cuenta de que estas pruebas y los programas eran lo mismo.

La lógica moderna empezó con **Gottlob Frege** a finales de 1870. Medio siglo más tarde dos aspectos del trabajo de Frege alcanzaron su punto culminante: La deducción natural de **Grentzen**, que capturó una noción de prueba y el Lambda Cálculo de **Alonzo Church**, que capturó la noción de programa. Ambos se publicaron a principios de 1930.

Hasta finales de 1960 no se descubrió la correspondencia tan precisa que había entre las pruebas de Grentzen y los programas de Church.

Introdujo la deducción natural en 1934. Muchas de las ideas son descendientes directas del trabajo de Frege.

Alonzo Church introdujo el Lambda Cálculo en 1932. Fue entendido como un nuevo camino de formulación lógica.

Hacia 1936, Church se dio cuenta de que los términos lambda podrían ser usados para expresar toda función que podría ser computada por una máquina.

Al mismo tiempo, **Turing** escribió el famoso artículo de la máquina que lleva su nombre. Se reconoció rápidamente que ambas formulaciones eran equivalentes y Turing fue a estudiar a Princeton con Church entre 1936 y 1938.

Church redujo todas las nociones del cálculo de sustitución. Normalmente, un matemático debe definir una función mediante una ecuación. Por ejemplo, si una función f es definida por la ecuación f(x)=t, donde t es algún término que contiene a x, entonces la aplicación f(u) devuelve el valor t[u/x], donde t[u/x] es el término que resulta de sustituir u oir cada aparición de x en t.

Por ejemplo, si f(x)=x*x, entonces f(3)=3*3=9. Church propuso una forma especial (más compacta) de escribir estas funciones. En vez de decir “la función f donde f(x)=t”, él simplemente escribió \x.t.

Para el ejemplo anterior: \x.x*x.

Un término de la forma \x.t se llama “lambda expresión”.

Church introdujo una versión tipada del lambda cálculo en 1940. Como ya hemos dicho había una correspondencia entre los trabajos de Church y Grentzen que fue descubierta logistas como **Haskell Curry** y **W.A. Howard**.

Un artículo de Howard en el que se refleja dicha correspondencia fue escrito en 1969 pero no se publicó hasta 1980 debido a la influencia que podía ejercer.

Church y Curry vivieron para ver el impacto tan importante que tuvieron sus teorías en los lenguajes de programación. Gentzen no fue tan afortunado ya que murió en 1945.

El Lambda Cálculo de Church tuvo un profundo impacto en el campo de los lenguajes de programación.

El concepto de que funciones pueden tomar funciones como argumentos y devolver funciones, fue particularmente importante.

Lisp usó la palabra clave lambda para definir funciones pero su definición de una función difiere sutilmente de la que da el lambda cálculo.

Algunos de los lenguajes que se inspiraron en el lambda cálculo son: Iswin, Scheme (un dialecto de Lisp), Miranda, Haskell...

Iswin y Scheme son no tipados (sin tipos) pero los otros tienen sistemas de tipos basados en los sistemas de Hindley-Milner y Girar-Reynolds.

Standard ML se caracteriza por su exploración en el tipado de módulos, Haskell por el tipado de clases y O'Calm por el tipado de objetos orientados.

1.1.2 El impacto de la lógica.

La programación funcional con tipos tienen también una especial importancia en Java. **Guy Steele** unos de los 3 diseñadores principales de Java, empezó con Scheme. **John Roser** otro programador de Scheme introdujo términos lambda en Java en forma de clases internas y son usadas para implementar llamadas en el GUI.

Una nueva generación de lenguajes distribuidos está ahora en desarrollo, por ejemplo: Cálculo pi de Milner, Ámbito de Cardelli y Gordon, el cálculo de Fournier, Gonthier y otros...

Todos estos lenguajes siguen la tradición del lambda cálculo tipado.

1.1.3 Demostradores de teoremas

Hewlett-Packard aplicó el demostrador de teorema HOL para verificar que ciertos protocolos no sufrían interbloqueo. Con esto se descubrieron errores.

El departamento de defensa y Tecnología Australiano está aplicando el demostrador de teorema Isabelle para verficar las condiciones de seguridad de ciertos misiles.

HOL e Isabelle están implementados en Standard ML, fue desarrollado por **Gordon, Melham** y otros con versiones publicadas en 1988,1990 y 1998; Isabelle fue desarrollado por **Paulson, Nipkow** y otros con versiones publicadas en 1988, 1994, 1998 y 1999.

ML/LCF explotó dos características principales centrales de los lenguajes funcionales:

- Funciones de orden superior.
- Tipos.

Una prueba táctica es una función a la que se le pasa una fórmula meta (final) para ser demostrada y que devuelve una lista de sub-metas emparejadas con una justificación.

Una justificación es una función que va de las pruebas a la prueba final.

Una táctica es una función que transforma tácticas pequeñas en tácticas grandes.

ML/LCF fue desarrollado por Mildner, Gordon y Wadsworth y la descripción completa fue publicada en 1979.

HOL e Isabelle son sólo dos de los muchos demostradores de teoremas que aprovecharon las ideas de LCF, al igual que el estándar ML es sólo uno de los muchos lenguajes que aprovecharon las ideas desarrolladas en ML.

Entre otros, COQ está implementado en CAML, Veritas en Miranda, Yarrow en Haskell y ALF,ELF y LEGO en Estándar ML también.

1.1.4 Confianza y Seguridad.

Cuando ejecutamos código máquina ¿hay algún modo de complementar la confianza de la persona que nos haya mandado ese código con un test que me garantice la seguridad?

La hay:

Lenguaje de tipos ensamblados, es uno de los modos.
El sistema de tipos desarrollado para el lambda cálculo es lo bastante flexible como para ser aplicado a otros lenguajes de programación de otros estilos y puede extenderse, incluso, al lenguaje máquina.

En Java, los código bytes son declarados como tipos y diseñados en TAL **(Greg Morriset 1998)** que son chequeados por el verificador antes de ejecutar el código.

Igualmente, en TAL, el código máquina es declarado con tipos que son chequeados por el chequeador de tipos antes de ejecutar el código.

1.2 INTRODUCCIÓN A LA PROGRAMACIÓN FUNCIONAL

1.2.1 ¿Qué es la Programación Funcional?

C, Java, Pascal, Ada, etc.. son lenguajes imperativos. Son "imperativos" en el sentido de que consisten en una secuencia de comandos que son ejecutados uno tras otro estrictamente.

Un programa funcional es un expresión simple que es ejecutada por evaluación de la expresión. La cuestión está en QUÉ va a ser computado, no en CÓMO va a serlo.

Otro lenguaje muy conocido, casi funcional es el lenguaje de consultas estándar de bases de datos, SQL. Una consulta SQL es una expresión con proyecciones, selecciones y uniones. Una consulta dice qué relación se debe computar sin decir cómo debe computarse. Además, la consulta puede ser evaluada en cualquier orden que sea conveniente

1.2.1.1 Modelo Funcional

El modelo funcional, tiene como objetivo la utilización de funciones matemáticas puras sin efectos.

El esquema del modelo funcional es similar al de una calculadora. Se establece una sesión interactiva entre sistema y usuario: el usuario introduce una expresión inicial y el sistema la evalúa mediante un proceso de reducción. En este procesos se utilizan las definiciones de funciones realizadas por el programador hasta obtener un valor no reducible.

Programador ✍✍ Usuario

Evaluación

Escrito (def. de función) ✍✍ Evaluador(Intérprete/Compilador)

El valor que devuelve una función está únicamente determinado por el valor de sus argumentos consiguiendo que una misma expresión tenga siempre el mismo valor. Es más sencillo demostrar la corrección de los programas ya que se cumplen propiedades matemáticas tradicionales como la propiedad conmutativa, asociativa, etc.

El programador se encarga de definir un conjunto de funciones sin preocuparse de los métodos de evaluación que posteriormente utilice el sistema. Estas funciones no tienen efectos laterales y no dependen de una arquitectura concreta.

Este modelo promueve la utilización de una serie de características como las funciones de orden superior, los sistemas de inferencia de tipos, el polimorfismo, la evaluación perezosa, etc.

1.2.1.2 Funciones de orden superior

Un lenguaje utiliza funciones de orden superior cuando permite que las funciones sean tratadas como valores de 1ª clase, permitiendo que sean almacenadas en estructuras de datos, que sean pasadas como argumentos de funciones y que sean devueltas como resultados.

La utilización de funciones de orden superior proporciona una mayor flexibilidad al programador, siendo una de las características mas sobresalientes de los lenguajes funcionales.

1.2.1.3 Sistemas de Inferencia de Tipos y Polimorfismo

Muchos lenguajes funcionales han adoptado un sistema de inferencia de tipos que consiste en:

- El programador no está obligado a declarar el tipo de las expresiones.
- El compilador contiene un algoritmo que infiere el tipo de las expresiones.
- Si el programador declara el tipo de alguna expresión, el sistema chequea que el tipo declarado coincide con el tipo inferido.

Los sistemas de inferencia de tipos permiten una mayor seguridad evitando errores de tipo en tiempo de ejecución y una mayor eficiencia, evitando realizar comprobaciones de tipos en tiempo de ejecución.

Los sistemas de inferencia de tipos aumentan su flexibilidad mediante la utilización de polimorfismo.

El polimorfismo permite que el tipo de una función dependa de un parámetro. Por ejemplo, si se define una función que calcule la longitud de una lista, una posible definición sería:

```
long ls = if vacia(L) then 0
          else 1 + long(cola (L))

long :: [x] ✍ Integer
```

El sistema de inferencia de tipos inferiría el tipo: long::[x]✍Integer, indicando que tiene como argumento una lista de elementos de un tipo a cualquiera y que devuelve un entero.

En un lenguaje sin polimorfismo sería necesario definir una función long para cada tipo de lista que necesitase. El polimorfismo permite una mayor reutilización de código ya que no es necesario repetir algoritmos para estructuras similares.

1.2.1.4 Evaluación Perezosa

Los lenguajes tradicionales, evalúan todos los argumentos de una función antes de conocer si estos serán utilizados.

Por ejemplo:

```
g (x:Integer):Integer          f (x:Integer; y:Integer):Integer
```

```
begin                              begin
  (* bucle infinito *)                 return (x+3);
while true do                      end;
   x:=x
end;

-- Programa Principal
begin
  write (f(4,g(5)));
end;
```

Con el sistema de evaluación tradicional, el programador no devolvería nada, puesto que al intentar evaluar g(5) el sistema entraría en un bucle infinito. Dicha técnica de evaluación se conoce como evaluación ansiosa porque evalúa todos los argumentos de una función antes de conocer si son necesarios.

Por otra parte, en ciertos lenguajes funcionales se utiliza evaluación perezosa que consiste en no evaluar un argumento hasta que no se necesita. En el ejemplo anterior, si se utilizase evaluación perezosa, el sistema escribiría 7.

Estos son algunas de las características más importantes de los lenguajes funcionales.

1.2.2 ¿Qué tienen de bueno los lenguajes funcionales?

Spreadsheets y SQL son lenguajes bastante especializados. Los lenguajes funcionales toman las mismas ideas y se mueven dentro del campo de la programación de propósito general.

Examinemos algunos de los beneficios de la programación funcional:

1. La brevedad de los programas funcionales hace que sean mucho más concisos que su copia imperativa.

2. Facilidad de comprensión de los programas funcionales. Deberíamos ser capaces de entender el programa sin ningún conocimiento previo del Haskell. No podemos decir lo mismo de un programa en C. Nos lleva bastante tiempo comprenderlo y, cuando lo hemos entendido, es muy fácil cometer un pequeño fallo y tener un programa incorrecto.

3. No hay ficheros “core”.
   La mayoría de los lenguajes funcionales y Haskell en particular son fuertemente tipados y eliminan una gran cantidad de clases que se crean en tiempo de compilación con las que se pueden cometer errores. O sea, fuertemente tipados significa que no hay ficheros “core**”.** No hay posibilidad de tratar un puntero como un entero o un entero como un puntero nulo.

4. Reutilización de código.

Los tipos fuertes están, por supuesto, disponibles en muchos lenguajes imperativos como Ada o Pascal. Sin embargo el sistema de tipos de Haskell es mucho menos restrictivo que, por ejemplo el de Pascal porque usa polimorfismo. Por ejemplo, el algoritmo Quicksort se puede implementar de la misma manera en Haskell para listas de enteros, de caracteres, listas de listas... mientras que la versión en C es sólo para arrays de enteros.

5. Plegado.

   Los lenguajes funcionales no estrictos tienen otra característica, la evaluación perezosa.

   Los lenguajes funcionales no estrictos llevan exactamente esta clase de evaluación. Las estructuras de datos son evaluadas justo en el momento en el que se necesita una respuesta y puede que haya parte de estas estructuras que no se evalúen

6. Abstracciones potentes.

   Generalmente, los lenguajes funcionales ofrecen nuevas formas para encapsular abstracciones. Una abstracción permite definir un objeto cuyo trabajo interno está oculto. Por ejemplo, un procedimiento en C es una abstracción. Las abstracciones son la clave para construir programas con módulos y de fácil mantenimiento. Son tan importantes que la pregunta para todo nuevo lenguaje es: "¿De qué mecanismos de abstracción dispone?".

   Un mecanismo de abstracción muy potente que está disponible en los lenguajes funcionales son las funciones de alto orden. En Haskell una función es un "ciudadano de primera clase": pueden pasarse tranquilamente a otras funciones, ser devueltas como el resultado de otra función, ser incluídas en una estructura de datos, etc. Esto nos quiere decir que el buen uso de estas funciones de alto orden puede mejorar sustancialmente la estructura y modularidad de muchos programas.

7. Manejo de direcciones de memoria.

   Muchos programas sofisticados necesitan asignar memoria dinámica desde una pila. En C, esto se hace con una llamada a "malloc", seguida de un código para inicializar la memoria. El programador es el responsable de liberar memoria cuando ya no se necesite más. Esto produce, muchas veces, errores del tipo "dangling-pointer" (punteros colgados).

   Cada lenguaje funcional libera al programador del manejo de este almacenamiento. La memoria es asignado e inicializado implícitamente y es recogido por el recolector de basura. La tecnología de asignación del store y la recolección de basura está muy bien desarrollada y los costes son bastante insignificantes.

1.2.3 Funcional contra imperativo.

Hay pocos programas que requieran ejecución a cualquier coste. Después de todo dejamos de escribir programas en lenguaje máquina, excepto, quizá para bucles de clave interna, hace mucho tiempo.
Los beneficios de tener un modelo de lenguaje de programación con más soportes (un número arbitrario de llamadas, variables locales en vez de un número fijo de registros, por ejemplo) vale más que los modestos costes de ejecución.

Los lenguajes funcionales están dando un gran paso hacia el modelo de programación de alto nivel.

Los programas son más fáciles de diseñar, de escribir y de mantener pero este tipo de lenguajes ofrece menos control sobre la máquina. Para la mayoría de los programas el resultado es perfectamente aceptable.

1.2.4 Preguntas sobre Programación Funcional.

¿Es difícil aprender la programación funcional?

La programación requiere un cambio de perspectiva y esta resulta difícil para los programadores. Pero la experiencia de Ericsson en la enseñanza de Erlang a sus programadores es que la mayoría encuentran esta transición fácil.

¿Son lentos los programas funcionales?

Solían serlo pero los compiladores de ahora han mejorado. Los programas en Haskell van bastante rápido para casi todas las aplicaciones que se demandan hoy en día.

Tengo muchas aplicaciones en C y C++, ¿puedo beneficiarme de la programación funcional sin tener que re-escribir todo el sistema entero?

Haskell ha sido integrado con éxito en muchas aplicaciones de diversas formas. HaskellDirect es un IDL (Interface Description Language) que permite a los programas Haskell trabajar con componentes software.

Los interfaces de bajo nivel C/C++ pueden ser generados con Green Card, permitiendo una estrecha integración entre Haskell y C. Estas herramientas han sido utilizadas exitosamente para construir un montón de sistemas de lenguajes mixtos.

1.2.5 Otro aspecto: La Crisis del Software

A principios de la década de los setenta aparecieron los 1ºs síntomas de lo que se ha denomindado crisis del software. Los programadores que se enfrentan a la construcción de grandes sistemas de software observan que sus productos no son fiables. La alta tasa de errores conocidos o por conocer pone en peligro la confianza que los usuarios depositan en sus sistemas.

Cuando los programadores quieren corregir los errores detectados se enfrentan a una dura tarea de mantenimiento. Cuando se intenta corregir un error detectado, una pequeña modificación trae consigo una serie de efectos no deseados sobre otras partes del sistema que, en la mayoría de las ocasiones, empeora la situación inicial.

La raíz del problema radica en la dificultad de demostrar que el sistema cumple los requisitos especificados. La verificación formal de programas es una técnica costosa que en raras ocasiones se aplica.

Por otro lado, el incremento en la potencia de procesamiento lleva consigo un incremento en la complejidad del software. Los ususarios exigen cada vez mayores prestaciones cuyo cumplimiento sigue poniendo en evidencia las limitaciones de recursos y la fragilidad del sistema.

Las posibles soluciones podrían englobarse en las siguientes líneas de investigación:

?? Ofrecer nuevos desarrollos de la Ingeniería del Software que permitan solucionar el problema del Análisis y Diseño de grandes Proyectos Informáticos. Actualmente las Metodologías Orientadas a Objetos han aportado una evolución importante dentra de este campo.
?? Proporcionas Sistemas de Prueba y Verificación de programas cuya utilización no sea costosa.
?? Construir técnicas de Síntesis de Programas que permitan obtener, a partir de unas especificaciones formales, código ejecutable.
?? Diseñas nuevas Arquitecturas de Computadoras, en particular, técnicas de procesamiento en paralelo.
?? **Proponer un modelo de computación diferente al modelo imperativo tradicional.**

Esta última solución se basa en la idea de que los problemas mencionados son inherentes al modelo computacional utilizado y su solución no se encontrará a menos que se utilice un modelo diferente. Entre los modelos propuestos se encuentra la **programación funcional**, cuyo objetivo es describir los problemas mediante funciones matemáticas puras sin efectos laterales, y la **programación lógica,** que describe los problemas mediante relaciones entre objetos.

2. HASKELL (Basado en un artículo de Simon Peyton Jones)

2.1 INTRODUCCIÓN

Haskell es un lenguaje de programación. En particular, es un lenguaje de tipos polimórficos, de evaluación perezosa, puramente funcional, muy diferente de la mayoría de los otros lenguajes de programación.

El nombre del lenguaje se debe a Haskell Brooks Curry.

Haskell se basa en el lambda cálculo, por eso se usa lamda como un logo.

2.2 ¿QUE ES HASKELL?

Haskell es un lenguaje de programación moderno, estándar, no estricto, puramente funcional.

Posee todas las características explicadas anteriormente, incluyendo polimorfismo de tipos, evaluación perezosa y funciones de alto orden. También es un tipo de sistema que soporta una forma sistemática de sobrecarga y un sistema modular.
Está específicamente diseñado para manejar un ancho rango de aplicaciones, tanto numéricas como simbólicas. Para este fin, Haskell tiene una sintaxis expresiva y una gran variedad de constructores de tipos, a parte de los tipos convencionales (enteros, punto flotante y booleanos).

Hay disponible un gran número de implementaciones. Todas son gratis. Los primero usuarios, tal vez, deban empezar con Hugs, un intérprete pequeño y portable de Haskell.

2.3 ¿POR QUE USAR HASKELL?

Escribir en grandes sistemas software para trabajar es difícil y caro. El mantenimiento de estos sistemas es aún más caro y difícil. Los lenguajes funcionales, como Haskell pueden hacer esto de manera más barata y más fácil.

Haskell, un lenguaje puramente funcional ofrece:

1. Un incremento sustacial de la productividad de los programas.
2. Código más claro y más corto y con un mantenimiento mejor.
3. Una “semántica de huecos” más pequeña entre el programador y el lenguaje.
4. Tiempos de computación más cortos.

Haskell es un lenguaje de amplio espectro, apropiado para una gran variedad de aplicaciones. Es particularmente apropiado para programas que necesitan ser altamente modificados y mantenidos.

La vida de muchos productos software se basa en la especificación, el diseño y el mantenimiento y no en la programación.

Los lenguajes funcionales son idóneos para escribir especificaciones que actualmente son ejecutadas (y, por lo tanto, probadas y depuradas).Tal especificación es, por tanto, el primer prototipo del programa final.

Los programas funcionales son también relativamente fáciles de mantener porque el código es más corto, más claro y el control riguroso de los efectos laterales elimina gran cantidad de interacciones imprevistas.

2.4 DESARROLLO FUTURO DEL HASKELL

2.4.1 Introducción

Haskell 98 está completo. Es la definición oficial y actual de Haskell. Se espera que este lenguaje pueda seguir funcionando aunque se añadan nuevas extensiones y los compiladores seguirán apoyándose en Haskell 98.

No obstante, han sido propuestas muchas extensiones que han sido implementadas en algunos sistemas Haskell; por ejemplo el diseño de guardas, clases de tipos con multiparámetros, cuantificaciones locales y universales, etc.

La lista de correo de Haskell es un forum de discusión sobre las características de los nuevos lenguajes. La gente propone una nueva característica de algún lenguaje y trata de convencer a los investigadores sobre la conveniencia de desarrollarla en algún sistema Haskell. Al final, la gente que está interesada de verdad define esta nueva característica en Haskell 98 y b presenta en un documento que es añadido a esta lista de correo.

Un buen ejemplo de la utilidad de este proceso es la creación de FFI (Foreing Function Interface) en Haskell.

Haskell II ha sido desarrollado durante mucho tiempo por un comité liderado por John Launchbury. Obviamente las extensiones que hayan sido bien descritas y comprobadas tendrán más probabilidad de ser aceptadas.

2.4.2 Extensiones de Haskell

La Lista Haskell Wish (The Haskell Wish List) recoge todas las extensiones, librerías e implementaciones específicas propuestas para el lenguaje.

FFI

Aunque todavía no hay una estándar para las FFI, sí hay una propuesta muy actual implementada en GHC y NHC98, y pronto en Hughs. Este es un buen candidato para un estándar de FFI en Haskell II, pero sólo ofrece lo mínimo. Esta propuesta ofrece dos módulos compañeros de FFI y algunos cambios en las librerías que hay ahora.

Hay una propuesta de Simon para implementar clases con tipos multi-parámetros en Haskell 2.

Views permite múltiples constructores lógicos en contra del diseño de los constructores reales. Hay una propuesta que se va a tener en cuenta en futuros artículos de Haskell.

El diseño de Guardas de Simon Peyton-Jones es una propuesta para el uso general de guardas en la definición de funciones.

Gofer.

Gofer es un pequeño intérprete desarrollado por Mark Jones que sirve de soporte al Haskell 1.2. Gofer está pensado como un lenguaje experimental sobre todo cuando hay clases de tipos. Aunque Haskell ha adoptado un montón de ideas de Gofer el sistema de clases de tipos de Gofer es más flexible que el de Haskell. Está disponible para todas la plataformas Unix incluyendo Linux y Mac.

Hugs es el sucesor de Gofer y Gofer está en proceso de desaparecer.

GPH (Glasgow Parallel Haskell).

Es una extensión de Haskell para programación paralela. Añade dos primitivas nuevas al lenguaje:

- par: una forma de composición paralela.
- seq: un composición secuencial.

Haciendo un uso correcto de estas dos primitivas, es posible que un programa sea evaluado en paralelo.

PH (Parallel Haskell).

El lenguaje pH es una variante de Haskell paralela, con evaluación impaciente con estructuras sintácticas para bucles, barreras y estructuras de almacenamiento I y M. El modelo de evaluación impaciente de pH es igual que el anterior.

Goffin.

Es una extensión de Haskell para programación paralela y distribuída.

Mondrian.

Modrian está evolucionando desde ser sólo un lenguaje de Internet a un lenguaje funcional para entornos Orientados a Objetos. Mondrian es un lenguaje funcional no estricto con hebras y excepciones, es como un Haskell “light” con extras.

O'Haskell.

O'Haskell es una extensión de objetos orientados para Haskell desarrollado en Chalmers.

O'Haskell añade dos nuevas características a la esencia de Haskell: una mónada de concurrencia, objetos reactivos con estado encapsulado y sistema de tipos con subtipos entre registros que funcionan como tipos de datos. Hay una implementación disponible, O'Hughs que es una derivación de Hughs.

Eden.

Eden es un lenguaje paralelo funcional que da una nueva visión de la programación paralela. Da a los programadores el control suficiente como para implementar los algoritmos de manera eficiente y, al mismo tiempo, los libera de tener que controlar los detalles relacionados con el bajo nivel como por ejemplo la gestión de memoria.

Eden es explícito entre procesos y la entrada y salida de datos, pero abstrae la transferencia entre estos datos y su sincronización. Eden es una extensión de Haskell pero anula la evaluación perezosa que es necesaria para el soportar el paralelismo.

3. PROLOG (A. COLMERAUER Y P.ROUSSEL 1992)

3.1. INTRODUCCION

El nombre de Prolog fue inventado en Marsella en 1972. Philippe Roussel eligió el nombre de la abreviación Programación en Lógica. Se puede decir que Prolog fue el nacimiento de una unión exitosa entre el procesamiento del lenguaje natural y la demostración automática de teoremas. La idea de usar un lenguaje natural como el Francés para razonar y comunicarse directamente con una computadora parecía una locura, éstas fueron las bases del proyecto propuesto por Alain Comerauer en el verano del 70. Alain tenía cierta experiencia con el procesamiento en la computadora del lenguaje natural y quiso ampliar sus conocimientos.

3.2. LA HISTORIA

A principio de Julio del 70, Robert Pasero y Philippe llegaron a Montreal. Ellos fueron invitados por Alain que dirigía el proyecto de traducción automática de la Universidad de Montreal.

Durante sus estancias en Montreal, Robert y Philippe se familiarizaron con el procesamiento del lenguaje natural. Ellos escribieron varios analizadores de contexto libre no deterministas en Algol 60 y un generador de parafrases francés usando A-System, el lenguaje de programación que Alain había desarrollado para el proyecto de traducción.

Al mismo tiempo, Jean Trudel, había desarrollado un demostrador completo de teoremas en el cual la unificación fue escrita en un estilo de programación actual.

3.2.1. Los Primeros Pasos (1971)¡Error! Marcador no definido.

A principios de 1971 cada uno volvió de Marsella. El proyecto de Alain tenía como objetivo hacer deducciónes basadas en textos escritos en francés.

Este trabajo se dividió de la siguiente manera:

Jean Trudel y Philippe trabajaron sobre la parte de deducción. Robert Pasero y Alain sobre la parte del lenguaje natural.

Trabajamos sobre un IBM360-44.

Jean mejoró su demostrador, Philippe produjo una serie de domostradores escritos en Algol-W. La interfaz entre las fórmulas lógicas y el francés consistieron el reglas 50Q-System para la entrada y reglas 17Q-System para la salida. La parte del razonamiento fue implementada a través de los demostradores de Philippe. Fue posible tener la siguiente conversación con la computadora:

USER

Cats kill mice
Tom is a cat who does not like mice who eat cheese
Jerry is a mouse who eats cheese
Max is not a mouse
¿What does Tom do?

COMPUTER

Tom does not like mice who eat cheese
Tom kills mice

USER

Who is a cat?

COMPUTER

Tom

USER

What does Jerry eat?

COMPUTER

Cheese

USER

Who does not like mice who eat cheese?

COMPUTER

Tom

USER

What does Tom eat?

COMPUTER

What cats who do not like mice who eat cheese eat??????

La fórmula lógica creada hizo uso de:

1. Constantes representando elementos:
   Tom,Jerry,Max,Cheese

2. Constantes representando conjuntos
   Cats,Mice,MiceWhoEatCheese,CatsWhoDoNotLikeMiceWhoEatCheese

3. Constantes representando relaciones binarias entre conjuntos
   Kill,DoesNotLike,Eat

4. Un símbolo funcional de aridad 1 y dos símbolos relacionales de aridad 2 y 3
   The,Subset,True

Un término de la forma **The(a)** fue tomado para representar el conjunto que consistía solamente en el elemento a. Una fórmula de la forma **Subset(x,y)** expresó la inclusión de conjunto 'x' en conjunto 'y' y una fórmula de la forma **True(r,x,y)** expresaba que los conjuntos 'x' e 'y' estaban en la relación 'r'. A las cláusulas que codificaban las sentencias, Jean Trudel añadió cuatro cláusulas relacionadas con los símbolos The,Subset,True:

(? **x)[Subset(x,x)]**
(? **x)(**? **y)(**? **z)[Subset(x,y)** ? **Subset(y,z)** ✍ **Subset(x,z)]**
(? **x)(**? **b)[Subset(The(a),The(b))** ✍ **Subset(The(b),The(a))]**
(? **x) (**? **y) (**? **r) (**? **x') (**? **y')[True(r,x,y)** ? **Subset(x,x')** ? **Subset(y,y')** ✍ **True (r, x',y')]**

Mientras continuaba su investigación sobre demostración automática de teoremas, Jean Trudel encontró un muy interesante método: SL-Resolución. El nos persuadió a invitar a uno de sus creadores, Robert Kowalski. Fue la primera vez que hablábamos a un especialista en demostradores de teoremas completos y era capaz de explicar el principio de resolución.

3.2.2. La Aplicación que creó Prolog (1972)

El año 1972 fue el más fructífero. En febrero obtuvimos una subvención de $20,000 por un período de 18 meses del Instituto de Investigación de Informática y Autómatas. Esto hizo posible la compra de un terminal y conectarlo al IBM360-67. También pudimos contratar una secretaria y un investigador, Henry Kanoui. Al final, Kowalski consiguió una financiación de la OTAN para numerosos intercambios entre Edimburgo y Marsella.

Después de los sistemas de resolución implementados por Philippe, el SL-Resolución de Kowalski y Kuehner parecían ser lo más interesante. SL-Resolución llegó a ser el foco de la tesis de Philippe sobre el procesamiento de igualdad formal en demostradores automáticos. La igualdad formal es menos expresiva que la estándar pero puede ser procesada más eficientemente. La tesis de Philippe llevaría a la introducción del predicado “dif” (para <>) en la 1ª versión de Prolog.

De nuevo invitamos a Robert Kowalski. Juntos tuvimos más conocimientos de demostradores automáticos. Supimos como automatizar pequeños problemas.

Después de la marcha de Robert. Alain encontró una forma de desarrollar analizadores poderosos. El asoció un predicado binario N(x,y) con cada símbolo no terminal N de la gramática, significando que “x” e “y” son cadenas terminales para la cual la cadena “u” definida por ”x=uy” existe y puede ser derivada de N. Al representar “x” e “y” por listas, cada regla gramatical puede ser codificada por una cláusula teniendo exactamente el mismo número de literales como ocurrencias de símbolos no terminales. Esto fue además posible hacerlo sin concatenación de listas (listas diferenciales). Alain también introdujo parámetros adicionales en cada no terminal para propagar y computar información. Como en Q-Systems, el analizador no sólo verificó que la sentencia era correcta si no también extraía una fórmula que representaba la información que contenía.

Se tomó una decisión: elegimos resolución lineal con unificación solamente entre las cabezas de las cláusulas.

Al final de 1972, el primer sistema Prolog fue implementado por Philippe en lenguaje Algol-W (Niklaus Wirt); a la paz, Alain y Robert Pasero crearon el sistema de comunicación interactivo en francés.

El sistema de comunicación interactivo fue el primer programa largo en Prolog alguna vez escrito. Tuvo 610 clausulas.

Este es el texto original y las tres respuestas a las tres preguntas finales.

TOUT PSICHIATRE EST UNE PERSONNE.
CHAQUE PERSONNE QU'IL ANALYSE, EST MALADE.
JACQUES EST UN PSYCHIATRE A MARSEILLE.

EST-CE QUE JACQUES EST UNE PERSONNE?
OU EST JACQUES?
EST-CE QUE JACQUES EST MALADE?

QUI.
A MARSEILLE.
JE NE SAIS PAS.

En Noviembre, junto con Robert Pasero, visitamos varios laboratorios americanos. Nos llevamos un informe preliminar sobre sistemas de comunicación en lenguaje natural y nuestro primer Prolog. Dejamos copias del informe en casi todos los lugares. Jacques Cohen nos dio la bienvenida a Boston y nos presentó a MIT. Fuimos a Standford, visitamos el SRI y el Laboratorio de Inteligencia Artificial de John McCarthy.

3.2.3. El definitivo Prolog (1973)

A principios de 1973, para ser exactos, en Abril, nuestro grupo alcanzó un estado oficial. La CNRS nos reconoció como un equipo de investigación llamado "Máquina dialogo interactivo en lenguaje natural" y nos financió con la cantidad de $6500 el primer año. Más tarde en octubre IRIA nos dio $50000 para renovar el contrato por dos años y medios.

Los usuarios de la versión preliminar de Prolog habían hecho suficientes programas para que su experiencia sirviera para una 2ª versión de Prolog, una versión firmemente orientada hacia un lenguaje de programación y no solo una clase de sistema automático deductivo.

Entre Febrero y Abril de 1973, Philippe visitó la Escuela de Inteligencia Artificial en la Universidad de Edinburgo, Philippe conoció Roger Boyer and Jay Moore. Ellos habían construido una implementación de resolución usando un método extremadamente ingenioso basado en una técnica de estructura compartida para representar las formulas lógicas generadas durante una deducción. El resultado de esta visita y las necesidades del laboratorio de un verdadero lenguaje de programación impulsaron nuestra decisión de un 2º Prolog.

En Mayo y Junio de 1973, teníamos las principales líneas del lenguaje, en particular, la elección de la sintaxis, primitivas básicas y los métodos de computación del intérprete, lo que tendió a una simplificación de la versión inicial. Desde Junio hasta el final del año, Gérard Battani,Henry Méloni y René Bazzoli, escribieron el interprete en Fortran y su supervisor en Prolog.

3.2.4. La Distribución de Prolog (1974 y 1975)

David Warren lo usó para escribir su sistema de generación de planes.

Henry Kanoui y Marc Bergman lo usó para desarrollar un sistema de manipulación simbólica.

Gérard Battani y Henry Meloni lo usó para desarrollar un sistema de reconocimiento de habla.

A principios de 1975, Alain Colmerauer había reescrito completamente el supervisor manteniendo las operaciones infijas.

René Bazzoli, usó un analizador top-down para leer las reglas del Prolog. David Warren incluyó más tarde reglas de gramática de esta clase en su versión compilada de Prolog y junto a Fernando Pereira rebautizó una veriante simplificada de gramáticas "metamorfosis" con el nombre de "gramáticas de clausula definitiva". Gramáticas metamorfosis permitían reglas gramáticas parametrizadas ser escritas directamente como ellas estaban en Q-Systems. El supervisor compiló estas reglas dentro del Prolog. Para probar la eficiencia y expresividad de las gramáticas metamorfosis, Alain escribió un pequeño modelo de compilador de un lenguaje Algol a un lenguaje máquina ficticio y un sistema completo de diálogo en francés con deducciones automáticas.

Gerard Battani y Henri Meloni estuvieron muy ocupados con la distribución del Prolog. Lo mandaron a Budapest, Varsovia, Toronto, Waterloo. Un estudiant b instaló en la Universidad de Montreal. Michel Van Caneghem hizo lo mimo en la IRIA en París antes de entrar a trabajar con nosotros. Finalmente, Maurice Bruynooghe llevó el Prolog to Leuven.

Como David Warren había apuntado, Prolog creció mucho más por el interés de las personas sacando copias.

3.3. UN ANTECESOR DE PROLOG, LOS Q-SYSTEMS

La historia del nacimiento de Prolog termina al final de 1975. Ahora iremos a un aspecto más técnico, primero de todo, describir el Q-Systems, el resultado de un primer intento: desarrollar un lenguaje de programación de muy alto nivel. Este intento y la experiencia adquirida inplementando el Q-system fue determinando para el segundo intento: Prolog.

3.3.1. Unificación One-way

El Q-system consiste en un conjunto de reglas reescritas formadas por secuencias de símpolos complejos separados por el signo '+'. Cada regla fiene la forma:

$$e1+e2+....+em \leftarrow f1+f2+....+fn$$

y su significado es que en la secuencia que nosotros manipulamos, una subsecuencia de la forma e1+e2+...+em puede ser reemplazado por la subsecuencia f1+f2+....+fn. Las

e's y laas f's son expresiones entre paréntesis que representan árboles, con una fuerte semejanza a los términos Prolog pero usando tres tipos de variables. Dependiendo si la variable empieza con una letra en el conjunto {(A,B,C,D,E,F},{I,J,K,L,M,N} o {U,V,W,X,Y,Z] denota si es una etiqueta, un arbol o usa secuencia de árboles separados por comas. Por ejemplo, la regla

P + A*(X*,I*,Y*) ✍ I* + A*(X*,Y*)

aplicada a la secuencia

P + Q(R,S,T) + P

produce tres posibles secuencias

R + Q(S,T) + P,
S + Q(R,T) + P,
T + Q(R,S) + P.

El concepto de unificación estaba ya presente pero en un solo sentido, las variables aparecian en las reglas pero nunca en la secuencia de árboles que estaba siendo transformada.

3.3.2. Estrategia de aplicación de reglas

Por ejemplo la secuencia

A+A+B+B+C+C

está representada por el grafo

A A B B C C
---

y la aplicación de las 4 reglas

A+B+C✍S
A+S+X+C ✍S
X+C✍C+X
B+B✍B+X

produce el grafo

Uno contiene todos los caminos que van desde el punto de entrada al de salida y no contiene ninguna flecha usada en la producción de otras flechas. Ademas contiene una única flecha

S

Esto es la secuencia reducida al símbolo s.

Este procedimiento es relativamente eficiente, desde que mantiene el máximo número de partes comunes en todas las secuencias. Otro aspecto del Q-Systems es que pueden ser aplicados uno después del otro. Cada uno coje la entrada al grafo resultante del sistema previo. Esta técnica es ampliamente usada en el proyecto de traducción automática, donde una sentencia inglesa pasa por no menos de 15 Q-Systems antes de ser traducida al francés.

Describamos la reversibilidad de los Q-Systems. El signo que hemos representado por ✍ era de hecho escrito == y dependiendo de que opción especificada era elegida al principio del programa era interpretado de izquierda a derecha o de derecha a izquierda. Estp es el mismo programa podía ser usado para describir una transformación y su inversa.

Es interesante destacar que en contraste a los analizadores escritos en Prolog que usaban una estrategia top-down, los analizadores escritos en Q-Systems usaban una estrategia bottom-up.

3.3.3. Implementación

Los Q-Systems fueron escrito en Algol por Alain Colmerauer y fueron operacionales en Octubre'69. Michel Van Caneghen y Francois Stellin desarrollaron más tarde una versión Fortran y Gilles Steward desarrolló una versión ultra rápida en lenguaje máquina para la computadora CDC 6400 de la Universitad de Montreal.

Estos Q-Systems fueron usados por el proyecto TAUM para la traducción Inglés-Francés.

Los Q-Systems también fueron usados unos pocos años más tard para escribir METEO system, una versión industrial que producía diariamente traducciones de las previsiones del clima en Canadá del inglés al francés.

3.4. EL PRELIMINAR PROLOG

Vayamos ahora a la versión preliminar de Prolog la cual fue creada a final de 1972 en conexión con el sistema de comunicación interactivo.

Nuestro objetivo era extremadamente ambicioso, tener una herramienta para el análisis sintáctico y semántico del lenguaje natural considerando la lógica de 1er orden no sólo como un lenguaje de programación si no como un conocimiento de representación de lenguaje. Con esto se crean las bases para tal lenguaje.

3.4.1. Razones para la elección del método de resolución

Los éxitos del proyecto dependían de la decisión concerniente a la elección del sistema lógico y el mecanismo de inferencia que fuera adoptado.

Más que demostrar un teorema por reducción al absurdo nosotros queríamos calcular un interesante conjunto de clausulas que fueran deducibles de un conjunto dado de clausulas.

La elección final fue una adaptación del método de resolución, pero incluyendo algunos elementos nuevos. Cada ejecución se llevaba a cabo con un conjunto de clausulas que constituían el "programa" y un conjunto de clausulas que constituían la "pregunta". Ambas producían un conjunto de clausulas que constituían la "respuesta". Los literales de las clausulas estaban ordenados de izquierda a derecha y la resolución era hecha entre la cabeza del literal del resolvente y la cabeza del literal de una de las clausulas del programa. La novedad residía en el hecho que en cada clausula una parte de los literales separadas por el signo "/" no era procesada durante la prueba. En vez de eso, eran acumuladas para producir una de las clausulas respuestas al final de la deducción. Además, ciertos predicados eran procesados por evaluación tardía y podía también ser transmitida como una respuesta. Finalmente, se decidió que no determinismo debería ser procesador por backtracking.

Regla inicialización de la deducción

Cuestión: L1...Lm / R1.....Rn

Resolvente: L1...Lm / R1....Rn

Regla de deducción básica

Resolvente: L0 L1....Lm / R1....Rn clausula elegida L'0L'1....L'm / R'1.....R'n

Resolvente: (L'1)...(L'm') (L1)....(Lm) / (R'1)..... (R'n') (R1).....(Rn)

Fin de la regla de deducción

Resolvente: / R1....Rn
Respuesta: R1....Rn

Donde naturalmente, L0 y L'0 son literales completamente unificables y ¿? es la sustitución más general que los unifica.

La 1ª razón para elegir esta técnica de resolución lineal con una selección de orden predefinido era su simplicidad y el hecho que nosotros podíamos producir clausular que fueran logicamente deducibles del programa el cual además garantizaba en cierta forma la validez de los resultados.

La elección del tratamiento de no determinismo fue básicamente un problema de eficiencia. Al programar un cierto número de métodos, Philippe había mostrado que uno de los problemas cruciales fue la explosión combinatorial y una consecuente perdida de memoria. Backtracking fue seleccionado pronto para el manejo de no determinismo.

3.4.2. Características del Prolog preliminar

Aparte del mecanismo de deducción que ya hemos discutido, un número de predicados construidos fueron añadidos al sistema: predicados para trazar una ejecución, COPY para copiar un término, BOUM para dividir un identificador a una lista de caracteres o reconstituirlo y DIF para procesar la igualdad simbólica. Debería decirse que nosotros reusamos incluir predicados entrada-salida en esta lista porque eran considerados demasiado lejanos de la lógica. Entrada-salida, especificación de resolventes iniciales, y encadenamiento entre programas eran especificados en un comando de lenguaje aplicado a conjuntos de cláusulas (read,copy,write,merge,prove,etc.). Este lenguaje, sin ninguna instrucción de control, permitió al encadenamiento ser definido solamente estaticamente. La 1ª versión ya incluía evaluación perezosa de ciertos predicados. El predicado DIF fue abandonado en la siguiente versión pero reapareció en Prologs modernos. Los operadores de control fueron situados al final de las cláusulas como marcas de puntuación, y su función era hacer cortes en el espacio buscado.

Las anotaciones:

.. ejecutaba un corte después de la cabeza de la clausula
.; ejecutaba un corte después de la ejecución de la regla entera.
;. Ejecutaba un corte después de la producción de al menos una respuesta
;; no tenía efecto

Estos operadores extra-lógicos eran extraños y fueron por lo tanto abandonados.

En el nivel sintáctico, los términos fueron escritos es forma funcional aunque fue posible introducir operadores unarios y binarios definidos por precedentes y operadores binarios infijos que podían ser representados por una ausencia de signo (como un producto en matemáticas).

```
READ
   RULES
      +DESC (*X,*Y) –CHILD (*X,*Y);;
      +DESC (*X,*Z) –CHILD (*X,*Y)  –DESC(*X,*Z);;
      +BROTHERSISTER  (*X,*Y)    --CHILD  (*Z,*X)  –CHILD(*Z,*Y)  --
DIF(*X.*Y);;
      AMEN

READ
   FACTS
      +CHILD(PAUL,MARIE);;
      +CHILD(PAUL,PIERRE);;
      +CHILD(PAUL,JEAN);;
      +CHILD(PIERRE,ALAIN);;
      +CHILD(PIERRE,PHILIPPE);;
      +CHILD(ALAIN,SOPHIE);;
      +CHILD(PHILIPPE,ROBERT);;
   AMEN

 READ
   QUESTION
  --BROTHERSISTER(MARIE,*X)–DESC(*X,*Y)/+GREATNEPHEW(*Y)--
MASC(*Y)..
   AMEN

CONCATENATE(FAMILYTIES, RULES,FACTS)

PROVE(FAMILYTIES,QUESTION,ANSWER)

WRITE(ANSWER)

AMEN
```

La salida de este programa son el siguiente conjunto de cláusulas binarias:

```
+GREATNEPHEW (SOPHIE)  --MASC(SOPHIE);..
+GREATNEPHEW(ROBERT) –MASC(ROBERT);.
```

El comando READ lee un conjunto de cláusulas precedidas por un nombre ‘x’ y finalizada con AMEN y lo asignó al nombre x. El comando CONCATENATE(y,x1,....,xn) computó la unión ‘y’ de los conjuntos de cláusulas x1,...,xn. El comando PROVE(x,y,z) fue el más importante; comenzó un procedimiento donde el programa es ‘x’, la resolvente inicial ‘y’ y el conjunto de respuestas es ‘x’. Finalmente, el comando WRITE(x) mostró el conjunto de cláusulas ‘x’.

3.4.3. Implementación del Prolog preliminar

El interprete fue implementado por Philippe en Algol-W, sobre un IBM 360-67 de la Universidad.Grenoble, equipado con el sistema operativo CP-CMS. Nosotros estábamos conectados a esta máquina via una línea especial. La máquina tenía 2 únicas características desconocidas en su mayoría para este tiempo, características que eran especiales para nuestro trabajo: Memoria virtual de 1Mb y nos permitía escribir programas interactivos. La elección del Algol-W nos fue impuesta, ya que era el único lenguaje de alto nivel que teníamos disponible para crear objetos estructurados dinamicamente.

Las bases para la implementación de la resolución fue ***an encoding of the clauses into inter-pointing*** structures con copia anticipada de cada regla usada en una deducción. El no determinismo se dirigió con una pila de backtracking y las sustituciones fueron ejecutadas creando cadenas de punteros. El analizador de cláusulas fue escrito también en Algol-W.

3.5. EL PROLOG DEFINITIVO (1973)

Nuestra mayor preocupación después de la versión preliminar fue el reforzamieto del aspecto de este lenguaje de programación, minimizando conceptos y mejorando su capacidad interactiva. Prolog estaba convirtiendose en un lenguaje basado en el principio de resolución dado un conjunto de predicados. Este conjunto era concevido como un mínimo conjunto facilitando al usuario a:

- Crear y modificar programas en memoria.
- Leer programas fuentes, analizarlos y cargarlos en memoria
- Interpretar cuestiones dinamicamente con una estructura análoga a otros elementos del lenguaje.
- Tener acceso dinamicamente a las estructura y los elementos de una deducción.
- Control de la ejecución del programa tan simple como fuera posible.

3.5.1. Estrategia de resolución

La experiencia desde la 1ª versión nos llevó a usar una versión simplificada de su estrategia de resolución. La decisión fue basada no solamente sobre sugerencias de los primeros programadores sino también en criterios de eficiencia y la elección de Fortran para programar el interprete lo cual nos forzó a usar el espacio de memoria. Las diferencias esenciales con la versión anterior fueron:

- No mas evaluación retrasada.
- Reemplazamiento del predicado BOUM por el predicado general UNIV.
- Las operaciones ASSERT Y RETRACT, en ese tiempo escrito AJOUT y SUPP.
- Un operador simple para el manejo del backtracking, el operador corte "!", en ese tiempo escritro "/".
- El concepto metallamada para usar una variable en vez de un literal.
- Uso de predicados ANCESTOR y STATE, los cuales han desaparecido en el presente Prolog, para acceder a literales antecesores y el común resolvente, para programadores que desean definir su propio mecanismo de resolución.

- El backtracking y el ordenamiento del conjunto de cláusulas que definen un predicado fueron los elementos básicos conservados como técnica para gestionar el no determinismo. La versión preliminar de Prolog fue bastante satisfactoria en este aspecto.

Además, después de una visita a Edinburgh, Philippe tuvo en mente las bases para una arquitectura que es extremadamente simple de implementar desde el punto de vista de la gestión de memoria y mucho más eficiente en términos de tiempo y espacio, si mantenemos la filosofía de gestionar el no determinismo por backtracking. Al final, todas las experiencias programando habían mostrado que esta técnica permitía a nuestros usuarios incorporar no determinismo facilmente como una dimensión añadida al control de la ejecución de predicados.

El procesamiento de "and" y "or" fueron justificadas por los objetivos requeridos:

- Emplear una estrategia simple y predecible que el usuario pueda controlar, permitiendo algun predicado extralógico (como E/S) para ser dada una definición operacional.

- Proveer un intérprete capaz de procesar deducciones con miles o diez miles de pasos (un objetivo imposible en los sistemas deductivos existentes en ese tiempo).

3.5.2. Sintaxis y Primitivas

En su mayoría, la sintaxis era la misma de la de la versión preliminar. En el nivel léxico, la sintaxis del identificador era la misma que en la mayoría de lenguajes y por eso las letras minúsculas no podían ser usadas (los teclados y sistemas operativos en ese tiempo no lo permitían). Entre las primitivas básicas para procesar problemas morfológicos, una primitiva simple UNIV fue usada para crear dinamicamente un átomo a partir de una secuencia de caracteres, para construir un objeto estructurado a partir de sus elementos y a la inversa. Esta primitiva era una de las herramientas básicas usadas para crear programas dinamicamente y manipular objetos cuyas estructuras son desconocidas antes de la ejecución de el programa.

Permitir al usuario definir sus propios operadores unarios y binarios especificando prioridad numércia fue muy útil y flexible aunque complicó los analizadores de cláusulas.

En la versión preliminar de Prolog, fue posible crear cláusulas que eran logicamente deducibles de otras cláusulas. Nuestra experiencia con esta versión nos había mostrado que algunas veces era necesario manipular cláusulas para propuestas ***very far removed*** desde lógica de 1er. Orden. Nos dimos cuenta que mucha investigación sería necesaria en el área de semánticas para modelar los problemas de actualización de conjuntos de cláusulas. Desde aquí tomamos la decisión pragmática de introducir primitivas extra lógicas (ADD, DELETE).

Uno de los rasgos echados en falta en el Prolog preliminar era una mecanismo que pudiera computar un término que pudiera entonces ser cogido como un literal para ser resuelto. Esto es una función esencial para la meta programación tal como un intérprete de comando.

3.5.3. Implementación del intérprete

El sistema de resolución en el cual en no determinismo era gestionado por Backtracking fue implementado usando un método muy novedoso para representar cláusulas, a medias entre la técnica basada sobre estructuras compartidas usada por Robert Boyer y Jay Moore y entre el backtracking usado en la versión preliminar de Prolog. Philippe llegó a esta solución durante su estancia en Edimburgo después de muchas discusiones con Robert Boyer. Esta técnica tenía muchas ventajas comparada a los demostradores automáticos de teoremas usados normalmente:

- En todos los sistemas conocidos, la unificación era hecha en tiempos, los cuales eran, el lo mejor, lineal en relación al tamaño de los términos unificados. En nuestro sistema, la mayoría de las unificaciones podían ser hechas en tiempo constante, determinado en ver de por el tamaño del dato por los templates traidos por las cláusulas de los programas llamados. Como resultado, la concatenación de dos listas era hecha en tiempo lineal correspondiente al tamaño de la 1ª y no en tiempo cuadrático como en todos los otros sistemas basados en técnicas de copia.

- En el mismo sistema, el espacio de memoria requerido por un paso en una deducción no está en fución del dato, sino de la cláusula del programa usada. La concatenación de dos listas usaba sólo una cantidad de memoria proporcional al tamaño de la 1ª lista.

- La implementación del no determinismo no requirió un recolector de basura sofisticado en la 1ª aproximación simplemente el uso de varias pilas sincronizadas sobre el backtracking.

En relación a la representación de los templates en memoria, decidimos usar representación prefija. El sistema del actual interprete consistía en: un cargador para leer cláusulas en una sintaxis restringida y un supervisor escrito en Prolog. Entre otras cosas, este supervisor contenía una evaluador de preguntas, un analizador que aceptaba sintaxis extendida y predicados de entrada-salida de gran nivel.

A Alain como a todos nosotros no le gustraba el Fortran, ***succeeded nonetheless*** en persuadir al equipo a programar el intérprete en este lenguaje. La elección fue basada en el hecho de que Fortran estaba ampliamente distribuido en todas las máquinas y que la máquina a la que teníamos acceso no soportaba otro lenguaje. Nosotros teníamos esperanza en tener un sistema portable, una predicción que fue bastante correcta.

Bajo la supervisión de Philippe, Gérard Battani y Henri Meloni desarrollaron el intérprete actual entre Juni 1973 y Octubre 1973 sobre una CII 10070 (variante de SIGMA 7) mientras que a René Bazzoli, bajo la dirección de Alain, se le dio la tarea de escribir el supervisor en el lenguaje Prolog. El programa consistió en aproximadamente 2000 instrucciones.

La máquina tenía un sistema operativo sin posibilidad de interacción vía un terminal. Por lo tanto, los datos y programas fueron introducidos por medio de tarjetas perforadas. El intérprete fue terminado finalmente en Diciembre 1973 por Gérard Battani y Henry Meloni después lo llevaron dentro de la máquina de Grenoble IBM

360-67. Philippe Roussel escribió el manual del usuario para este Prolog dos años más tarde.

3.6. CONCLUSION

Después de todas estas vicisitudes y todos los detalles técnicos, es interesante dar un paso atrás e irnos al nacimiento de Prolog en una perspectiva más amplia. El articulo publicado por Alan Robinson en Enero 1965 "Una máquina de orientación lógica basada en el principio de resolución", contenía las semillas del lenguaje Prolog. Este artículo fue la fuente de un caudal de trabajos sobre demostradores de teoremas automáticos.

Nuestra contribución fue transformar ese demostrador de teorema a un lenguaje de programación.

4. LISP (ARTICULO DE JOHN MCCARTHY 26/07/1996)

4.1 INTRODUCCION

Este artículo contiene las ideas básicas del desarrollo y distingue dos períodos:

Desde verano 1956 hasta verano 1958, cuando la mayoría de las ideas claves fueron desarrolladas (algunas de las cuales fueron desarrolladas en FORTRAN basado en FLPL) y final de 1958 hasta 1962 cuando el lenguaje de programación fue implementado y aplicado a problemas de inteligencia artificial.

Como lenguaje de programación, LISP está caracterizado por las siguientes ideas:

- ?? Computación con expresiones simbólicas más que con números.
- ?? Representación de expresiones simbólicas y otra información con estructuras de listas en memoria.
- ?? Un pequeño conjunto de operaciones selectoras y constructoras expresadas como funciones.
- ?? Composición de funciones como una herramienta para formar funciones más complejas.
- ?? El uso recursivo de expresiones condicionales como una herramienta suficiente para construir funciones computables.
- ?? El uso de expresiones para nombrar funciones.
- ?? La representación de programas LISP como datos LISP.
- ?? La interpretación de expresiones condicionales de conectivas booleanas.
- ?? La función “eval” que sirve como definición formal del lenguaje y como un intérprete.
- ?? Recolección de basura como un medio de manejar el problema del borrado.

Algunas de estas ideas se tomaron de otros lenguajes, pero la mayoría eran nuevas.

4.1.1 Algo viejo, algo nuevo (Martin Heller 12/02/2001)

LISP (Lenguaje de procesamiento de listas) data de 1959 (la fecha del interprete LISP 1). Sus orígenes se remontan a 1955, si se cuenta la primera investigación de McCarthy en la Inteligencia Artificial.

De los lenguajes de programación supervivientes, sólo FORTRAN es más antiguo que LISP, aunque COBOL anda cerca (la especificación CODASYL para COBOL fue realizada en 1960).

Evolución del LISP

LISP divergió en varios dialectos alrededor del año 1962. Éstos fueron mezclados en un único dialecto, Common Lisp, al principio de los 80, y el libro "Common Lisp: the language", escrito por Guy Steele, fue publicado en 1984. CLOS (Common LISP Object System) añadió orientación a objetos a LISP en los 80;

CL/CLOS se convirtió en estándar ANSI a finales de 1994. Otros dialectos de LISP aun sobreviven, como Elisp (para Emacs), AutoLisp (para AutoCAD) y Scheme.

Durante años, LISP tuvo una bien merecida reputación de lenguaje demasiado grande y lento (y con demasiados paréntesis) para el trabajo real. A finales de los 60, el PDP-10 era la plataforma dominante para LISP, así como para la investigación de la Inteligencia Artificial. Mas tarde, hubo un movimiento para construir maquinas especializadas de alto rendimiento para LISP, que engendraron Symbolics, LMI (Lisp Machines Inc.) y Thinking Machines Corporation (TMC). Symbolics y LMI desaparecieron hace mucho. TMC duró algo mas debido a las aplicaciones científicas de la ¿"maquina de conexión masiva en paralelo de series"?, también de algún software sobre tratamiento de datos que de vez en cuando iba a parar a manos de Oracle.

Actualmente hay muchas implementaciones de LISP disponibles, desde software gratuito de dominio publico con el código fuente hasta sistemas de desarrollo comercial de miles de dólares.

Las implementaciones mas corrientes de LISP pueden ejecutarse correctamente en un PC normal con Windows, MacIntosh o en estaciones de trabajo Unix.

La primera vez que me enfrenté a LISP era un estudiante graduado en Física en los 70. La mejor manera por aquel entonces de resolver complejos problemas algebraicos y de cálculo de la teoría cuántica era conectando con el sistema Macsyma (Project MAC symbolic algebra) en el MIT, que estaba escrito en MacLisp y "hosted-almacenada-albergado" en un PDP-10. Tuve otra experiencia con Lisp al principio de los 90, cuando ayudé a Symbolics a implementar una versión PC mucho más madura de Macsyma.

4.2 LA PREHISTORIA DEL LISP: VERANO '56 AL VERANO '58

Mi deseo por un lenguaje que procesara listas algebraicas para trabajar en el campo de la Inteligencia Artificial en el computador IBM 704 surgió en el verano de 1956, durante el proyecto de investigación en Inteligencia Artificial en Dartmouth (primer estudio organizado de Inteligencia Artificial).

En esta reunión, Newel, Shaw y Simon describieron el IPL2, un lenguaje de procesamiento de listas para el computador JOHNIAC de la Rand's Corporation, en el que implementaron su programa de lógica teórica. Existió la tentación de copiar el IPL, porque su forma estaba basada en un cargador JOHNIAC que tenían disponible, y porque la idea de FORTRAN de escribir programas algebraicamente resultaba atractiva.

Inmediatamente estuvo claro que subexpresiones arbitrarias de expresiones simbólicas podían ser obtenidas mediante la composición de funciones que extrajeran subexpresiones inmediatas, y esto pareció razón suficiente para llegar a un lenguaje algebraico.

Había dos motivaciones principales para desarrollar un lenguaje para el IBM 704. La primera, es que IBM estaba estableciendo un centro de computación en Nueva Inglaterra, en el M.I.T., que Dartmouth usaría. La segunda motivación es que IBM

quería desarrollar un programa que demostrara teoremas en geometría plana (basado en una idea de Marvin Minski), y yo iba a ser consultor en ese proyecto.

En esa época, IBM aprecia una buena apuesta para impulsar la investigación de la Inteligencia Artificial vigorosamente, y se esperaba que surgieran nuevos proyectos.

No estaba claro si el proyecto de IBM sobre FORTRAN llevaría a un lenguaje en el que el procesamiento de listas pudiera ser sacado convenientemente, o si un nuevo lenguaje seria requerido. Sin embargo, muchas consideraciones eran independientes de cómo eso podría cambiar. Aparte de mi trabajo como consultor en el programa de geometría, mi propia investigación en el campo de la Inteligencia Artificial estaba siguiendo las pautas que llevaron a la propuesta "Advice taker" en 1958. Ésta trataba de la representación de información sobre el mundo mediante sentencias en un lenguaje formal apropiado y un programa deductivo que decidiría que hacer mediante inferencias lógicas. Representar sentencias usando la estructura de listas parecía apropiado (aun lo es), y un lenguaje de procesamiento de listas también parecía apropiado para programas las operaciones involucradas en la deducción (y aun lo es).

Esta representación interna de información simbólica abandona la familiar notación infija a favor de una notación que simplifica la tarea de programar las computaciones sustantivas (por ejemplo, la deducción lógica o la simplificación algebraica, diferenciación o integración).

Si las notaciones tradicionales iban a ser usadas externamente, programas de traducción deberían ser escritos. Muchos programas LISP usan notación prefijara las expresiones algebraicas, porque generalmente deben determinar la conectiva principal antes de decidir lo siguiente que hay que hacer. En esto, LISP difiere de casi todos los otros sistemas de computaciones simbólicas. Los programas de COMIT, FORMAC y formula Algol expresan las computaciones como operaciones en alguna aproximación a la tradicional forma de representar expresiones simbólicas. SNOBOL opera con cadenas de caracteres, pero es neutral en como se usan éstas cadenas de caracteres para representar información simbólica. Esta característica probablemente influyo en el éxito de LISP frente a los lenguajes anteriormente mencionados, sobre todo cuando se deben escribir grandes programas. La ventaja es como la que tiene los computadores binarios frente a los decimales, pero aun mayor.

El primer problema fue como hacer la estructura de lista en el IBM 704. Este ordenador tiene palabras de 36 bits, y dos partes de 15 bits llamadas *dirección* y *decremento* se distinguían por usar instrucciones especiales para mover su contenido (el de las 2 partes de 15 bits) desde y hacia los registros índice de 15 bits. La dirección de la maquina tenia 15 bits, por lo que estaba claro que la estructura de lista debería usar punteros de 15 bits.

Por consiguiente, era natural considerar la palabra dividida en 4 partes:

- La parte de dirección,
- la parte de decremento,
- la parte prefija y
- La parte del tag.

Las 2 ultimas partes eran de 3 bits cada una y separadas la una de la otra por el decremento, así no podían ser fácilmente combinadas en una única parte de 6 bits.

En este punto surgieron algunas dudas acerca de cuales deberían ser los operadores básicos, porque la operación de extraer una parte de la palabra usando mascaras era considerada separadamente de la operación de tomar el contenido de una palabra en memoria como una función de su dirección. Por el momento, parecía dudoso considerar esta última operación como una función, ya que su valor dependía del contenido en memoria en el momento en que la operación fuera ejecutada, por lo que no actuaba realmente como una función matemática. Sin embargo, las ventajas de tratarla gramaticalmente como una función, tal que pudiera ser compuesta también era evidente.

Por consiguiente, el conjunto inicial de funciones propuesto incluía *cwr* , que devolvía el "contenido de la palabra en el registro numero XXX" y 4 funciones que extraían las partes de la palabra y las desplazaba a una posición estándar a la derecha de la palabra. Una función adicional de 3 argumentos que también extraería una secuencia arbitraria de bits fue propuesta.

Pronto nos dimos cuenta de que la extracción de una subexpresion implicaba la composición de la extracción de la parte de dirección con *cwr* y que continuar a lo largo de la lista implicaba componer la extracción de la parte de decremento con *cwr*. Por consiguiente, se crea *car*, que devuelve el "contenido de la parte de dirección del registro numero XXX", y sus análogos *cdr, cpr y ctr* son definidos también. La principal motivación para implementar *car* y *cdr* separadamente fue fortalecida por el simple (y vulgar) hecho de que el IBM 704 tenía instrucciones (relativas a la indexación) que hacia esas operaciones fáciles de implementar.

Una operación constructora que sacara una palabra de la lista de almacenamiento y la 'rellenara' con un cierto contenido dado era también obviamente requerida. En algún momento *cons(a,d,p,t)* fue definida, pero era considerada como una subrutina y no como una función con un valor. Este trabajo se desarrolló en Dartmouth, pero no en una computadora ya que el centro de computación de Nueva Inglaterra no esperaba sus IBM 704 hasta el año siguiente.

En relación con el proyecto de IBM sobre geometría plana, Nathaniel Rochester y Herbert Gelernter (aconsejados por McCarthy) decidieron implementar un lenguaje de procesamiento de listas en FORTRAN, ya que esto parecía ser el método más fácil para comenzar, y, en esos días, se pensaba que escribir un compilador para un lenguaje nuevo llevaba muchos años. Este trabajo fue emprendido por Herbert Gelernter y Carl Gerberich en IBM, y condujo a FLPL, que quiere decir "FORTRAN List Processing Language". Gelernter y Gerberich se dieron cuenta de que *cons* debería ser una función, no solo una subrutina, y que su valor debería ser la localización de la palabra que había sido sacada de la lista de almacenamiento libre. Esto permitió construir nuevas expresiones sin nada que ver con las subexpresiones, sino componiendo ocurrencias de *cons*.

Mientras que las expresiones podían ser manejadas fácilmente en FLPL, y fue usado con éxito en el programa de geometría, no tenia ni expresiones condicionales ni recursión, y borrar la estructura de la lista era llevado a cabo explícitamente por el programa. Yo inventé las expresiones condicionales relacionándolas con un conjunto de

rutinas de movimientos legales de ajedrez que escribí en FORTRAN para el IBM 704 en el MIT durante 1957-1958. Este programa no usaba procesamiento de listas. La sentencia IF que proveía FORTRAN 1 y FORTRAN 2 era muy difícil de usar, por lo que fue natural el inventar una función *XIF(M,N1,N2)* cuyo valor devuelto era bien *N1*, bien *N2*, según si la expresión *M* valía 0 o no. Esta función acortó muchos programas y los hizo más fáciles de entender, pero tenia que ser usada poco, porque los 3 argumentos tenían que ser evaluados antes de que el XIF fuera introducido, ya que el XIF era llamado como una función normal de FORTRAN pero escrita en lenguaje maquina. Esto condujo a la invención de una verdadera expresión condicional que evaluara sólo uno de los 2, bien N1, bien N2, dependiendo de que M fuera verdadero o falso y del deseo de un lenguaje de programación que permitiría su uso.

Un informe definiendo las expresiones condicionales y proponiendo su uso en Algol fue enviado a "Communications of the ACM", pero fue arbitrariamente degradado a carta al editor, porque era muy corto.

Pasé el verano del 58 en el departamento de Información e Investigación de IBM, invitado por Nathaniel Rochester y elegí la diferenciación de expresiones algebraicas como un programa prueba. Esta investigación llevó a las siguientes innovaciones mas allá de FLPL:

1. Escritura de definiciones de funciones recursivas usando expresiones condicionales. La idea de diferenciación es claramente recursiva, y las expresiones condicionales permitieron combinar los casos en una única formula.

2. La función *maplist* que crea una lista con las aplicaciones de un argumento funcional a los elementos de una lista. Esta función era deseada para diferenciación de sumas de (arbitrariamente) muchos términos, y con una leve modificación, podía ser aplicado para la diferenciación de productos. (Es lo que ahora llamamos *mapcar*).

3. Para usar funciones como argumentos, se necesita una notación para funciones, y lo normal era usar la notación de Church (1941). No entendí el resto de su libro, por lo que estuve tentado de intentar implementar sus mecanismos más generales para definir funciones. Church usaba funciones de orden superior en vez de usar expresiones condicionales. Las expresiones condicionales son mucho más fáciles de implementar en los computadores.

4. La definición recursiva de la diferenciación no hacia nada respecto a la eliminación de estructuras de lista abandonadas. Parecía no haber ninguna solución clara en ese momento, pero la idea de complicar la elegante definición de diferenciación con un borrado explícito era muy poco atractiva.

Ni que decir tiene que la idea de esta investigación no era el programa de diferenciación en sí (varios ya habían sido escritos anteriormente), sino la clarificación de las operaciones implicadas en la computación simbólica. De hecho, el programa de

diferenciación no fue implementado ese verano, ya que FLPL no permite ni expresiones condicionales ni el uso recursivo de subrutinas.

En este punto, un nuevo lenguaje era necesario, debido tanto a las dificultades técnicas como políticas de tratar vanamente de reparar el FORTRAN, y ni las expresiones condicionales ni la recursión podían ser implementadas con funciones FORTRAN en lenguaje máquina--ni siquiera con funciones que modificaran el código que las llama.

Por otra parte, el grupo IBM parecía satisfecho con el FLPL tal y como estaba, y no quería hacer los drásticos cambios requeridos para permitir expresiones condicionales y definiciones recursivas. Tal y como lo recuerdo, argumentaron que eran cambios innecesarios.

4.3 LA IMPLEMENTACIÓN DE LISP

A finales de 1958, John McCarthy ,profesor de Ciencias de la Comunicación en M.I.T. y Marvin Minsky , profesor del Departamento de Matemáticas empezaron un proyecto de Inteligencia Artificial. El proyecto fue apoyado por el Laboratorio de investigación de Electrónica de M.I.T. Ellos pidieron, una habitación, dos programadores, una secretaria y una tarjeta perforada.

La implementación de Lisp empezó a finales de 1958. La idea original era producir un compilador y para ello necesitaban alguna experiencia para obtener convenciones buenas para linkar subrutinas , pila manual y borrado. Antes de esto, empezaron compilando a mano varias funciones en ensamblador y escribiendo subrutinas para probar un 'entorno' de LISP.

Esto incluyó programas para leer e imprimir estructuras de lista. McCarthy no recuerda si se tomó la decisión de usar paréntesis tanto en listas como en los datos de Lisp o si ya había sido usada en algún articulo.

Los programas que se compilaban a mano fueron escritos con una notación informal llamada M-expresiones deseando que fueran lo más parecido posible a Fortran. Además el lenguaje Fortran permitió expresiones condicionales y funciones básicas de Lisp. Permitiendo definiciones de funciones recursivas que no requerían una nueva notación de las definiciones de función permitidas en Fortran I

La M-notación también usaba corchetes en vez de paréntesis para encerrar los argumentos de las funciones para así reservar los paréntesis para las listas. Esto se deseó para compilar aproximaciones de la M-notación , pero la M-notación nunca fue completamente definida , porque la representación de funciones Lisp por listas de Lisp llegó a ser el lenguaje de programación dominante cuando más tarde el interprete estuvo disponible.

Una máquina capaz de leer M-notaciones habría requerido redefinición , porque la M-notación con lápiz y papel usaba caracteres que no estaban disponibles en los IBM de tarjeta perforada 026.

Los programas de 'lectura' e 'impresión' provocó una notación estándar externa para información de símbolos , por ejemplo representando x + 3y + z por (PLUS X (TIMES 3 Y) Z) y por (ALL (X) (OR (P X) (Q X Y))). Otras notaciones requerían necesariamente una programación especial , porque las notaciones estándares de matemáticas tratan sintácticamente diferentes operadores de diferentes formas. Esta notación más tarde se llamó 'Cambridge Polish' , porque esta era similar a la notación prefija de Lukasiewicz.

El problema del borrado también tuvo que ser considerado y fue claramente poco estético usar el borrado explícitamente como hizo IPL. Había dos alternativas. La primera fue borrar el contenido antiguo de un programa cuando era actualizado. Desde que las operaciones 'car' y 'cdr'no eran para copiar estructuras, la listas se mezclarían y el borrado requeriría un sistema para contar referencias. Desde que había solamente 6 bits a la izquierda de la palabra , en partes separadas , las cuentas de referencia parecían imposibles de realizar sin un cambio drástico en la manera en que las listas eran representadas (Un esquema de manejo de listas usando cuentas de referencia , fue usado en 1960 por Collins en un ordenador CDC de 48 bits).

La segunda alternativa es la recolección de basura en la cual el almacenaje se abandona hasta que la lista de almacenaje libre esta exhausta , el almacenaje accesible desde las variables del programa y la pila se marca , y lo almacenado sin marcar se mete en una nueva lista libre.

Se decidió usar recolección de basura. Al mismo tiempo también se decidió usar rutinas SAVE y UNSAVE que usan un simple array de pila publico para salvar los valores de las variables y direcciones devueltas de las subrutinas en la implementación de subrutinas recursivas. IPL construyó pilas como estructura de listas y su uso tenia que ser programado explícitamente.

Otra decisión fue renunciar al prefijo y etiquetas de las palabras , para abandonar 'cwr' y hacer una función 'cons' de dos argumentos.

Estas simplificaciones hicieron de Lisp una forma de describir funciones computables mucho más ingeniosas que las máquinas de Turing o que la definición general de recursividad usada en teoría de funciones recursivas. El hecho de que las máquinas de Turing constituyen un lenguaje de programación difícil no preocupaba mucho a los teóricos de las funciones recursivas ya que ellos nunca tuvieron una razón para escribir definiciones recursivas particulares , desde que la teoría tiene que ver con las funciones recursivas en general. Ellos a menudo tenían razón para probar que las funciones recursivas con propiedades especificas existen ,pero esto se puede hacer con un argumento informal sin tener que apuntarlos explícitamente. En los comienzos de la informática , algunas personas desarrollaron lenguajes de programación basados en máquinas de Turing , quizás esto se veía más científico. De todas maneras ,McCarthy decidió escribir un articulo describiendo Lisp como un lenguaje de programación y como un formalismo para hacer funciones recursivas. El articulo fue "Funciones Recursivas de expresiones de símbolos y su computación por máquina, (McCarthy 1960)".

Un consideración matemática que influenció a Lisp fue expresar programas como expresiones aplicadas construidas de variables y constantes usando funciones.

McCarthy consideró esto importante para hacer que estas expresiones obedezcan las leyes matemáticas permitiendo reemplazar expresiones por expresiones dando el mismo valor. El motivo fue permitir pruebas de propiedades usando ordinariamente métodos matemáticos. Esto es sólo posible si los efectos laterales pueden ser evitados. El Lisp puro esta libre de efectos laterales y (Cartwright 1976) y (Cartwright y McCarthy 1978) muestran como representar programas en Lisp puro con sentencias y esquemas en lógica de primer orden.

Otra forma para mostrar que Lisp era más hábil que las máquinas de Turing fue escribir una función universal de Lisp y mostrar que es mucho más corto y más comprensible que la descripción de una máquina universal de Turing. Esta fue la función eval(e,a) , la cual computa el valor de una expresión Lisp 'e' -- el segundo argumento 'a' es una lista de asignaciones de valores para las variables ('a' se necesita para hacer la recursión).

La escritura de 'eval' requirió inventar una notación representando funciones Lisp como datos Lisp. Se rechazó esa notación porque no se consideraba adecuada para representar programas Lisp en la práctica. La completitud lógica requirió que la notación usada para expresar argumentos funcionales como funciones , se extendiese a las funciones recursivas , y la notación LABEL fue inventada por Nathaniel Rochester para éste propósito. Park señalo que LABEL no fue lógicamente necesario desde que el resultado se pudo conseguir usando sólo LAMBDA.

Russell notó que 'eval' podía servir como interprete para Lisp , rápidamente se codificó a mano, y tuvimos un lenguaje de programación con un interprete.

La inesperada aparición de un interprete tendió a congelar la forma del lenguaje y algunas de las decisiones hicieron que se alejara bastante de las funciones recursivas. Estas incluyeron la notación de COND para expresiones condicionales que conduce a una innecesaria profundidad de paréntesis, y el uso del número cero para denotar la lista vacía NIL y el valor de verdad FALSE.

Otra razón para apoyar la dificultad de Lisp es que esperamos cambiar esta forma de escritura a M-expresiones. El proyecto de definir M-expresiones y compilarlas o al menos traducirlas en S-expresiones no fue terminado aunque tampoco se abandonó.

4.4 DEL LISP 1 AL LISP 1.5

?? Propiedades de las listas

La idea de proveer cada átomo con una lista de propiedades fue presentada en la primera asamblea de implementación de lenguajes. También fue una de las ideas teóricas de la Advice Taker . Los programas de lectura (READ) e impresión (PRINT) exigieron que fueran accesibles los nombres de los átomos y que pronto la definición de funciones también. Esto fue necesario para indicar si una función era una SUBR en código máquina o era una EXPR representada por una estructura de lista. Varias funciones que trataron con listas de propiedades fueron también hechas para programas de aplicación que hacían duro el uso de estas.

?? Inserción y eliminación de elementos en una lista.

Una de las virtudes originales anunciada para la Inteligencia Artificial del procesamiento de listas fue la habilidad para insertar y borrar elementos de las listas. Desafortunadamente , esta facilidad coexistía con la dificultad de compartir estructuras de listas. Además , las operaciones de insertar y borrar no tienen una buena representación como funciones. Lisp las tiene en forma de pseudo funciones rplaca y rplacd, pero los programas que las usan no pueden ser convenientemente representados en lógica.

?? Los números

Muchas computaciones requieren ambos , números y símbolos de expresión. Originalmente los números fueron implementados en LISP I como una lista de átomos , y esto era demasiado lento para todas las computaciones , excepto para las más simples. Una eficiente implementación de los números como átomos en S-expresiones se hizo en LISP 1.5 , pero aun así , las computaciones numéricas todavía eran de 10 a 100 veces más lentas que en Fortran. Una computación numérica eficiente requiere algunas formas de tipos en los lenguajes fuente y una distinción entre tratar los números como tales o como elementos de una S-expresión. Algunas versiones recientes de LISP permiten distinguir tipos.

?? Variables libres

James R. Slagle programó la siguiente definición de función de LISP.

El objetivo de esta función era encontrar una subexpresión de X satisfaciendo p(X) y devolviendo f(X). Si la búsqueda era insatisfacible , entonces la función u sin argumentos se computa y se devuelve su valor. La dificultad era que cuando había una recursión interna , el valor buscado por car(X) estaba fuera , pero el valor interno estaba actualmente usado. En terminología moderna se buscaba un ámbito léxico y se obtuvo un ámbito dinámico.

Debo confesar que consideré esta dificultad como minúscula y expresé la confidencia de que Steve Russell debería arreglarlo pronto. Lo arregló pero para inventar el mecanismo FUNARG. Dificultades similares se mostraron más tarde en Algol 60 , y fue Russell el que aportó la solución más comprensiva. Mientras todo iba bien en el interprete , la comprensión y velocidad se vieron opuestas al código compilado , lo que llevó a una sucesión de compromisos.

Desafortunadamente , el tiempo no permitió escribir un apéndice de la historia del problema.

?? La característica del programa

Además de composición de funciones y expresiones condicionales , LISP también permite programas secuenciales escritos con instrucciones de asignación y go to's. La idea de tener programas secuenciales antecede a la definición de funciones recursivas.

?? Una vez programado el interprete "eval" y disponible para el programador , fue especialmente fácil de usar porque los programas interpretes de Lisp estaban expresados como datos Lisp. En particular, hizo posible FEXPRS y FSUBRS las cuales son 'funciones' que no dan sus argumentos actuales si no que dan las expresiones que evalúan a los argumentos y deben llamarse así mismas cuando quieren evaluar las expresiones. La principal aplicación de esta facilidad es para funciones que no siempre evalúan todos sus argumentos , sólo evalúan algunos del principio y entonces deciden los próximos a evaluar. Esta facilidad se parece a la de ALGOL sólo que es más flexible porque esta explícitamente disponible.

Puesto que Lisp trabaja con listas, fue también conveniente proveer a funciones con un número variable de argumentos ,suministrándolos con una lista de argumentos más bien que con los argumentos separados.

Desafortunadamente , ninguna de estas características ha llegado a dar una comprensiva y clara semántica matemática en conexión con Lisp u otros lenguajes de programación. El mejor intento de conexión con Lisp fue el de Michael Gordon en 1973, pero fue demasiado complicado.

?? El primer intento de hacer un compilador fue realizado por Robert Brayton , pero no tuvo éxito. El primer compilador de Lisp con éxito fue realizado por Timothy Hart y Michael Levin , fue escrito en Lisp pretendiendo ser el primer compilador escrito en el lenguaje que se compila. Participó mucha gente en el desarrollo inicial de Lisp ,algunos de ellos fueron Paul Abrahams , Robert Brayton , Daniel Edwards , Patrick Fischer , Phyllis Fox , Rochester de IBM , Steve Russell,etc.

4.5 MÁS ALLÁ DEL LISP 1.5

Como lenguaje de programación Lisp tenia muchas limitaciones. Las más evidentes a principios de 1960 fueron: computaciones numéricas muy lentas , imposibilidad de representar objetos por bloques de registros y recolección de basura, y falta de un buen sistema de entrada-salida en notaciones convencionales.

Todos estos problemas y otros fueron arreglados en Lisp 2. Mientras tanto nos teníamos que manejar con el Lisp 1.5 desarrollado por M.I.T. el cual corregía sólo las deficiencias más grandes.

El proyecto de Lisp 2 fue una colaboración de Corporación de Desarrollo de Sistemas y de Información Internacional Inc. y fue planeado inicialmente para los computadores Q32, los cuales fueron construidos por IBM para propósitos militares.

Cuando quedó claro que los Q32 tenían muy poca memoria se decidió desarrollar el lenguaje para los IBM 360/67 y Equipos Digitales PDP-6. SDC estaba adquiriendo la versión antigua mientras que III y M.I.T. y Stanford prefirieron la última.

El proyecto resultó más caro de lo que se esperaba y por eso LISP 2 se perdió.

Desde el punto de vista de los años 70, esto fue lamentable porque se gastó mucho dinero para el desarrollo del Lisp con muy pocos resultados. Sin embargo no se sabia entonces que la máquina dominante para la investigación en Inteligencia Artificial debería ser la PDP-10, sucesora de la PDP-6. La existencia de un interprete y la ausencia de declaraciones hizo particularmente natural el uso de Lisp en un entorno de tiempo compartido. Esto es conveniente para definir funciones, probarlas y re-editarlas sin salir del interprete del Lisp.

En 1960 se hizo una demostración de Lisp en un prototipo de tiempo-compartido en IBM 704. L.Peter Deutsch implementó el primer Lisp interactivo en un computador PDP-1 en 1963 pero los PDP-1 tenían muy poca memoria para computaciones simbólicas. Las más importantes implementaciones de Lisp resultaron ser estas , para los computadores PDP-6 y su sucesor el PDP-10 hecha por Corporación de Equipos Digitales de Maynard, Massachusetts.

El temprano desarrollo de Lisp en M.I.T. para esta línea de máquinas y su subsecuente desarrollo de INTERLISP y MACLISP también contribuyó para hacer de estas máquinas , las máquinas elegidas para la investigación de la Inteligencia Artificial.

El Lisp de IBM 704 fue extendido al IBM 7090 , para más tarde llevar Lisps a los IBM 360 y 370.

Las primeras publicaciones de Lisp fueron en el Quarterly Progress Reports del Laboratorio de Investigación de electrónica de M.I.T. McCarthy 1960 fue la primera publicación con su Manual del Programador de LISP 1.5.

Después de la publicación de McCarthy y Levin en 1962 , se hicieron muchas implementaciones de Lisp para numerosos computadores. Sin embargo , en contraste con la situación de los lenguajes de programación más usados , ninguna organización ha intentado jamas propagar el Lisp y nunca ha habido un intento de estandarizarlo , aunque recientemente A.C. Hearn ha desarrollado un Lisp estándar conjuntamente con Marty Griss&Griss en 1978 , el cual se ejecuta en un número de ordenadores para apoyar al Sistema REDUCE usado para la computación con expresiones algebraicas.

4.6 CONCLUSIONES

LISP es el segundo lenguaje de programación más viejo en uso expansivo (después de FORTRAN y sin contar APT, el cual no es usado para programar por si mismo).

Su longevidad se debe a dos hechos:

?? El primero, su esencia en el espacio de los lenguajes de programación. El uso recursivo de expresiones condicionales, representación externa de información simbólica por listas e interna por estructuras de listas.

?? Segundo, LISP todavía tiene rasgos operacionales que no tienen otros lenguajes y que lo hacen un vehículo conveniente para sistemas de más alto nivel para computación simbólica y para inteligencia artificial. Estos incluyen su sistema ***run-time*** que ofrece buen acceso al host de la máquina y a su sistema operativo, su lenguaje interno de estructura de lista que lo hace un buen objetivo para compilar desde lenguajes de más alto nivel, su compatibilidad con sistemas que producen programas binarios o de nivel ensamblador y la ventaja de su intérprete como una orden de lenguaje para llevar otros programas.

LISP llegará a ser obsoleto cuando alguien haga un lenguaje más comprensivo que el de LISP.

5. LA EVOLUCIÓN DE LISP, Por Guy Steele y Richard Gabriel

5.1. INTRODUCCIÓN

Algo increíble ha ocurrido a Lisp en los últimos 30 años. Se ha hecho imposible tratar todo lo de interés coherentemente en un único pase lineal a lo largo de la materia, cronológicamente o del modo en que se haga. Los proyectos y dialectos emergen, se separan, se unen y mueren de muy diversas y complejas maneras; las trayectorias de distintas personas son llevadas por estas conexiones de modos a veces paralelos, pero más frecuentemente ortogonales. Las ideas brincan de un proyecto a otro, de una persona a otra. Hemos elegido presentar una serie de tajadas a través del asunto principal. Esta organización nos va a llevar inevitablemente a alguna redundancia en la presentación.

Además, hemos tenido que omitir una gran cantidad de material debido a la falta de espacio.

La siguiente sección trata sobre la historia del Lisp en términos de proyectos y personas, desde el punto en que McCarthy lo dejó.

5.2. CRONOLOGIA DE LA IMPLEMENTACIÓN DE PROYECTOS

Las primeras ideas sobre un lenguaje que finalmente se convirtió en Lisp comenzaron en 1956, con John McCarthy en la "Dartmouth Summer Research Project on Artificial Intelligence". La implementación actual comenzó en el otoño de 1958. En 1978 McCarthy contó los inicios del lenguaje hasta justo después del LISP 1.5. Comenzaremos nuestra historia donde McCarthy la dejó

5.2.1. Del Lisp 1.5 al Lisp del PDP-6: 1960-1965

Durante este periodo, Lisp se extendió rápidamente a una gran variedad de computadoras, bien haciendo más robusto un Lisp ya existente en otro ordenador o mediante una nueva implementación. En casi todos los casos, el dialecto Lisp era pequeño y simple, y la implementación sencilla. Hubo muy pocos cambios sobre el lenguaje original.

A principio de los 60, Timothy Hart y Thomas Evans implementaron Lisp 1.5 en el Univac M 460, una versión militar del Univac 490. Fue re-utilizado y robustecida desde el Lisp 1.5 del IBM 7090 usando un "cross-compiler" y una pequeña cantidad código máquina para los niveles más bajos de la implementación del Lisp.

Robert Saunders y sus colegas de la System Development Corporation implementaron el Lisp 1.5 en el computador AN/FSQ-32/v, construido por IBM y llamado simplemente Q-32. La implementación fue re-utilizada y robustecida del IBM 7090 y de los computadores PDP-1 de la universidad de Stanford (cuyo Lisp fue implementado por John McCarthy y Steve Russel)

En 1963, L. Peter Deutsch (en ese momento un estudiante de instituto), implementó un Lisp similar al Lisp 1.5 del PDP-1 en Bolt Beranek and Newman. Ese Lisp fue llamado Basic PDP-1 Lisp.

En 1964, una versión de Lisp 1.5 estaba ejecutándose en el departamento de Ingeniería eléctrica del MIT, en un IBM 7094, ejecutando el sistema compatible de compartición de tiempo (CTSS). Este Lisp y el Basic PDP-1 Lisp fueron las principales influencias en el Lisp del PDP-6, implementado por DEC y algunos miembros del "MIT's Tech Model Railroad Club" en la primavera de 1964. Este Lisp fue el primer programa escrito en el PDP-6. También fue el ancestro de MacLisp, el Lisp escrito para ejecutarse en el sistema incompatible de compartición de tiempo (ITS) en el MIT, inicialmente en el PDP-6 y posteriormente en el PDP-10.

En BBN, un sucesor del Basic PDP-1 Lisp fue implementado en el PDP-1, y una actualización, diseñada después del Lisp 1.5 en el sistema CTSS del MIT, fue implementado el Scientific Data Systems por Daniel Bobrow y Murphy. Una nueva actualización fue escrita para el PDP-10 por Alice Hartley y Murphy, y este Lisp fue llamado BBN Lisp. En 1973, no mucho después de que SDS fuera adquirida por Xerox y renombrada Xerox Data Systems, el mantenimiento del BBN Lisp fue compartido por BBN y Xerox Palo Alto Research Center y el nombre de Lisp fue cambiado a InterLisp.

Los computadores PDP-6 y PDP-10 eran, debido a su diseño, especialmente amoldables a Lisp, con palabras de 36 bits y direcciones d e18. Esto permitió que una celda *CONS* (un par de punteros o direcciones) fuera almacenada eficientemente en una única palabra. Existían instrucciones de media palabra que hicieron que la manipulación de las celdas *CONS* con *CAR* y *CDR* fuera muy rápida. Los PDP-6 y los PDP-10 también tenían una rápida y muy poderosa pila de instrucciones que habilitó la llamada rápida a funciones en Lisp.

En 1965, prácticamente todos los Lisp existentes eran idénticos o diferían solo en pequeñas cosas. Después de 1965 (o siendo más preciso, después de que MacLisp y BBN Lisp divergieran el uno del otro) surgieron muchos dialectos de Lisp.

Durante este periodo había pocos fondos para el trabajo sobre lenguajes, los grupos estaban separados unos de otros, y cada grupo estaba dirigido principalmente a servir las necesidades de un grupo de usuarios local, que generalmente consistía en un puñado de investigadores. Se experimento mucho con estrategias de implementación. Había poco pensamiento de consolidación, en parte por el sentimiento de ser pioneros en algo que cada laboratorio tenia.

Una excepción a todo esto fue el proyecto LISP 2, el desarrollo de un lenguaje concertado que fue fundado por ARPA y representó una salida radical del Lisp 1.5. Existía la esperanza de que este diseño superaría con creces al Lisp 1.5 y un procesamiento simbólico más cercano al Algol 60. Lisp 2 fue implementado para los computadores Q-32 pero nunca consiguió una gran aceptación. Jean Sammet comento sobre el Lisp 2:

"...en contraste con muchos lenguajes, en los que primero se realiza el diseño y luego se implementa... se dijo del Lisp 2 que era una implementación en busca de un lenguaje."

Los primeros Lisps estándares fueron MacLisp e InterLisp; como tales se merecen alguna atención.

5.2.2. MacLisp

MacLisp fue el principal dialecto Lisp en los laboratorios de Inteligencia Artificial del MIT desde finales de los 60 hasta principios de los 80. Otros trabajos importantes sobre Lisp realizados en dicho laboratorio fueron el Lisp de las máquinas Lisp (mas tarde llamado ZetaLisp) y Scheme. MacLisp se identifica generalmente con el computador PDP-10, pero MacLisp también s ejecutaba en otra máquina, la HoneyWell 6180, bajo el sistema operativo Multics.

5.2.2.1. Inicios de MacLisp

La principal característica de la era del MacLisp/InterLisp es la atención a la calidad de la producción. Durante este periodo se vio una consolidación de las técnicas de implementación, con una gran cantidad de detalles.

Una diferencia clave entre MacLisp e InterLisp fue la aproximación a la sintaxis. MacLisp favoreció el estilo de lista pura, usando *EVAL* como el nivel máximo. InterLisp, junto con el Lisp 1.5, usaba *EVALQUOTE*.

Para concatenar las listas (BOAT AIRPLANE SKATEBOARD) y (CAR TRUCK) en MacLisp, uno pasaría la siguiente expresión a *EVAL:*

(APPEND (QUOTE (BOAT AIRPLANE SKATEBOARD) (QUOTE (CAR TRUCK)))

Y el resultado sería, por supuesto, (BOAT AIRPLANE SKATEBOARD CAR TRUCK). Sin embargo, en Lisp 1.5, uno pasaría una expresión como la siguiente (en realidad, 2 expresiones) al *EVALQUOTE*:

APPEND ((BOAT AIRPLANE SKATEBOARD) (CAR TRUCK))

La primera expresión denota una función, y la segunda una lista de argumentos. El 'quote' de *EVALQUOTE* significa que hay un 'quote' implícito en los argumentos dentro de la función aplicada. MacLisp se bifurcó y usó *EVAL* como un interfaz de nivel máximo: BBN Lisp (e InterLisp) se acomodaron a las 2 formas; si la primera línea de entrada contenía una forma completa y al menos un carácter de una segunda forma, el BBN Lisp finalizaba leyendo la segunda forma y usando la interface del *EVALQUOTE* para esa interacción; si no, leía exactamente una forma y usaba la interface del *EVAL* para esa interacción.

McCarthy dijo que el Lisp original era considerado como una maquina de Turing: podía realizar cualquier computación, dado un conjunto de instrucciones (una función) y la entrada inicial en su cinta (argumentos).

MacLisp introdujo *LEXPR*, que es un tipo de función que tiene un numero cualquiera de argumentos y los mete en la pila; el único parámetro de la función depende del numero de argumentos que se le pasen. La necesidad del *LEXPR* venia del

deseo de tener funciones de aridad variable tales como el +. Aunque no hay necesidad semántica de un + n-ario, es conveniente para los programadores poder escribir (+ A B C D) en vez de su equivalente (+ A (+ B (+ C D))).

La simple pero poderosa macro en la que DEFMACRO esta basada fue introducida en MacLisp a mediados de los 60.

Otra gran mejora respecto al Lisp 1.5 fueron los arrays; la modificación de simples predicados (tales como *MEMBER*) para que fueran funciones que devolvieran valores útiles; y la introducción de la función *ERR*, que permitía usar código para señalizar un error.

En Lisp 1.5, ciertas funciones pre-construidas podían señalar errores, cuando se introducían argumentos incorrectos, por ejemplo. La señalización de un error normalmente tenía lugar bien al finalizar la ejecución del programa, bien cuando se invocaba el debugger. Lisp 1.5 también tenia la función *ERRSET*, que era útil para una ejecución controlada de código que podría tener errores.
La instrucción especial

(ERRSET form)

evalúa *form* en un contexto en el que los errores que se produzcan no terminen la ejecución del programa ni entre en el debugger. Si *form* no provoca error, *ERRSET* devuelve una lista singleton del valor. Si la ejecución del *form* causa un error, entonces *ERRSET* devuelve tranquilamente un NIL.

MacLisp añadió la función *ERR*, que señaliza un error. Si *ERR* es invocado en el contexto dinámico de un *ERRSET*, entonces el valor que se devuelve al *ERR* es el valor que devuelve el *ERRSET*.

Los programadores pronto comenzaron a usar ERRSET y ERR no solo para atrapar y señalizar errores, sino para propósitos de control más generales (salidas no locales dinámicas). Desafortunadamente, también atrapaba errores no esperados, haciendo el programa más difícil de depurar. Un par mas de primitivas, CATCH y THROW fueron introducidas en MacLisp, por lo que ERRSET podía ser reservado para su uso, el atrapar errores.

La siguiente fase del desarrollo de MacLisp comenzó cuando los desarrolladores de MacLisp se dieron cuenta de influencia de un grupo de usuarios que emergía: el proyecto MAC y el grupo de Mathlab/Macsyma. La idea del desarrollo de MacLisp se orientó a satisfacer las necesidades de comunidad de usuarios mas que a preocuparse de realizar el diseño y la implementación del lenguaje como tal.

5.2.2.2. Lo último de MacLisp

Durante la ultima parte de su ciclo de vida, MacLisp adoptó características de otros dialectos Lisp y de otros lenguajes, y fueron inventadas algunas cosas novedosas.

El desarrollo más significativo para el MacLisp tuvo lugar a principio de los 70, cuando las técnicas en el prototipo "compilador aritmético rápido" LISCOM fueron incorporadas al compilador MacLisp por John White, quien ya había sido el principal 'mantenedor' y desarrollador de MacLisp durante varios años.

Steele, quien a los 17 ya llevaba colaborando con el MIT varios años y había implementado un sistema Lisp para el IBM 1130, fue contratado como hacker para el Lisp por el grupo Mathlab del MIT, cuyo jefe era Joel Moses. Steele pronto tuvo la responsabilidad del mantenimiento del interprete de MacLisp y del sistema de tiempo de ejecución, permitiendo a John White concentrarse casi todo el tiempo en mejorar el compilador.

El nuevo compilador MacLisp resultante, NCOMPLR, se convirtió en un estándar en contra de que el resto de compiladores Lisp estaban medidos en términos de la velocidad de ejecución del código. Inspirados en las necesidades del laboratorio de Inteligencia Artificial del MIT, cuyas necesidades incluían las computaciones numéricas realizadas en visión y robótica, aparecieron nuevas maneras de representar y compilar código numérico debido a la actuación conjunta del MacLisp compilado y diversos compiladores FORTRAN.

Los BigNums (enteros con precisión arbitraria) fueron añadidos alrededor de 1971 debido a las necesidades de los usuarios de Macsyma. El código era una transcripción mas o menos fiel de los algoritmos que se encontraban en [Knuth, 1969]. Mas tarde, Bill Gosper sugirió algunas mejoras, a destacar una versión del GCD que combinaba las buenas características del algoritmo GCD binario con el método de Lehmer para incrementar la velocidad de la división entera de BigNums.

En 1973 y 1974, David Moon lideró un proyecto para implementar MacLisp en el HoneyWell 6180 bajo Multics. Como parte del proyecto, escribió el primer manual de referencia verdaderamente entendible para el MacLisp, que se conoció familiarmente como el "Moonual".

Richard Greenblatt comenzó el proyecto de la maquina de Lisp para el MIT en 1974; Moon, Stallman y otros muchos hackers de Lisp del laboratorio de Inteligencia Artificial del MIT se unieron eventualmente a este proyecto. Conforme este proyecto progresaba, las características del lenguaje se iban ajustando al MacLisp del PDP-10 ya que los 2 proyectos crecieron juntos.

Las lambda-listas complejas surgieron por la influencia del Muddle (mas tarde llamado MDL), que era un lenguaje usado por el grupo de simulación dinámica en el MIT. Se ejecutaba en un PDP-10 situado en la misma habitación en la que se encontraban las máquinas de Inteligencia Artificial y Mathlab. La austera sintaxis del Lisp 1.5 no era lo suficientemente potente para expresar claramente los diferentes papeles de los argumentos de una función. Las lambda-listas complejas aparecieron como una solución a este problema, y fueron ampliamente aceptadas; el problema es que complicaron terriblemente el elegante Common Lisp Object System.
MacLisp introduce la noción de 'read tables'. Una 'read table' provee una sintaxis de entrada programable para los programas y datos. Cuando un carácter es introducido, se consulta la 'read table' para determinar las características sintácticas del carácter para usarlo en juntar tokens. Por ejemplo, la tabla se usa para determinar que caracteres

denotan el espacio en blanco. Adicionalmente, se pueden asociar funciones a caracteres, y de este modo una función es invocada siempre que un carácter determinado sea leído. Este poderoso recurso, hizo fácil el experimentar con sintaxis de entrada alternativas para Lisp.

MacLisp adoptó muy pocas de las características de otros dialectos Lisp. En 1974, unas doce personas (entre implementadores de MacLisp e InterLisp) se reunieron en el MIT. Había una esperanza inicial de encontrar algún punto común entre ambos dialectos, pero la reunión sólo sirvió para mostrar el gran abismo q separaba a ambos grupos, en todo, desde los detalles de implementación hasta la filosofía del diseño. Al final, sólo se produjo un intercambio trivial de características en la 'gran cumbre MacLisp/InterLisp':

> MacLisp adoptó de InterLisp el comportamiento "(CAR NIL)✍NIL
> y (CDR NIL)✍NIL", e InterLisp adoptó el concepto de 'read table'.

A mediados de los 70 se hizo mas patente que la limitación de espacio del PDP-10 (sobre un megabyte), se estaba convirtiendo en una limitación ya que el tamaño de los programas Lisp estaba creciendo. Por ese tiempo MacLisp ya había disfrutado de 10 años de un gran uso y aceptación dentro de la reducida pero muy influyente comunidad de usuarios.

Para muchos, el periodo en que MacLisp era estable, fue una época dorada en la que todo lo referente al mundo de Lisp era correcto. Este mismo periodo es recordado también por algunos nostálgicos como la época dorada de la Inteligencia Artificial. Alrededor de 1980 la comunidad de usuarios estaba en declive debido a problemas monetarios, a la vez que los laboratorios de Inteligencia Artificial tenían el mismo problema.

5.2.3. InterLisp

InterLisp (y el Lisp de BBN antes) introdujo muchas ideas radicales a la metodología y al estilo de programación en Lisp. Las mas conocidas se encuentran en herramientas de programación, y algunas son el corrector ortográfico, el empaquetamiento de archivos, DWIM, CLISP, el editor de estructuras y MASTERSCOPE.

El inicio de estas ideas se puede encontrar en el proyecto de doctorado de Warren Teitelman sobre la simbiosis del hombre y la maquina. En particular, contiene las raíces de la edición de estructuras, puntos de ruptura, avisos y CLISP.

El corrector ortográfico y DWIM fueron diseñados para compensar flaquezas humanas. Cuando un símbolo no tiene valor (o definición de función), el corrector ortográfico de InterLisp era invocado, porque el símbolo podía estar mal escrito. El corrector comparaba el símbolo que quizás estuviera mal escrito con una lista de palabras conocidas. El usuario tenia opciones para controlar el comportamiento del sistema con respecto a la corrección ortográfica. El sistema haría una de estas tres cosas:

1. - Corregir automáticamente,

2. - parar y preguntar si una corrección propuesta era aceptable o
3. - dar simplemente una señal de error.

El corrector ortográfico se encontraba bajo el control de un programa mucho mas grande, el DWIM (Do What I Mean). Cuando un error era detectado por el sistema InterLisp, DWIM era invocado para determinar la acción mas apropiada. DWIM era capaz de arreglar sentencias con un numero de paréntesis incorrectos, que junto con la incorrección ortográfica de los identificadores eran los problemas más comunes que tenían los usuarios.

DWIM encaja bien con la filosofía de trabajo del InterLisp. El modelo de InterLisp estaba para emular una sesión de conexión infinita. En InterLisp, el programador trabajaba con código fuente presentado por un editor de estructura, que operaba con código fuente con la forma de estructura de datos Lisp residente en memoria. Cualesquiera cambios en el código eran salvados en un fichero, que servia como almacén para el código del programador. Los cambios de DWIM también eran salvados. La estructura residente en memoria era considerada la representación primaria del programa; el fichero era simplemente una copia de seguridad. (El modelo de MacLisp, como contraste, era para el programador trabajar con ficheros ASCII que representaban el problema, usando un editor orientado a caracteres. Aquí el fichero era considerado la representación primaria del programa).

CLISP(Conversational Lisp) era una mezcla de las sintaxis de ALGOL e Ingles metida en la sintaxis normal de InterLisp. CLISP también dependía del mecanismo genérico DWIM. Hay que darse cuenta de que no solo debe arreglar los tokens e insertar paréntesis, sino que también debe separar átomos tales como ‘N=0’ y ‘N*’ en varios tokens apropiados, pero el usuario no tiene porque poner espacios alrededor de los operadores infijos.

CLISP definió un conjunto muy útil de constructores de iteración. Veamos un ejemplo sobre estos constructores. Este ejemplo imprime todos los números primos ‘p’ entre ‘m’ y ‘n':

(FOR P FROM M TO N DO (PRINT P) WHILE (PRIMEP P))

MASTERSCOPE era una utilidad para descubrir información sobre las funciones en un sistema muy grande. MASTERSCOPE podía analizar el cuerpo del código, construir una base de datos y contestar preguntas interactivamente.

InterLisp introdujo a la comunidad Lisp el concepto de compilación por bloque, en la que múltiples funciones podían ser compiladas como un solo bloque; esto llevo a una llamada de funciones más rápida de lo que hubiera sido posible en InterLisp.

InterLisp funcionaba en PDP-10’s, Vaxen (plural de VAX) y en varios maquinas Lisp desarrolladas por XEROX y BBN. Las máquinas InterLisp mas comúnmente usadas eran la Dolphin, el Dorado y la Dandelion (o D-máquinas). El Dorado era la más rápida de las 3, y las D-máquinas las más usadas. Es interesante observar que las diferentes implementaciones de InterLisp usaban distintas técnicas para manejar variables especiales. Por ejemplo, InterLisp-10 (para PDP-10) usaba búsqueda en anchura, mientras que InterLisp-D (para D-máquinas) usaba en profundidad. Estas

distintas técnicas hacían que un mismo programa tardase 10 veces mas en ejecutarse en un tipo de maquina que en otra.

Como MacLisp, InterLisp extendió los mecanismos de llamada de funciones de Lisp 1.5 con respecto a como los argumentos puedan ser pasado a una función. Las definiciones de funciones de InterLisp especificaban los argumentos como el producto cruzado de 2 atributos: LAMBDA vs. NLAMBDA y expandido vs. no-expandido.

Las funciones lambda evalúan cada uno de sus argumentos; las nlambda ninguno. Las funciones expandidas requieren un numero fijo de argumentos; las no-expandidas aceptan un numero variable de argumentos. Estos atributos no eran demasiado ortogonales, ya que el parámetro de una nlambda no-expandida estaba ligado a una lista de los argumentos no evaluados, y el parámetro de una lambda no-expandida estaba ligado al numero de argumentos pasados y la función ARG era usada para coger el valor actual del argumento. Había unas equivalencias bastante significativas entre los mecanismos de InterLisp y MacLisp:

InterLisp	**MacLisp**
LAMBDA expandida	EXPR
LAMBDA no-expandida	LEXPR
NLAMBDA expandida	Sin equivalencia
NLAMBDA no-expandida	FEXPR

Existía otra diferencia importante entre MacLisp e InterLisp. En MacLisp, “numero de argumentos fijados” tenia un significado demasiado rígido; una función definida con 3 argumentos, tenia que ser llamada con 3 argumentos, ni más, ni menos. En InterLisp, cualquier función podía ser llamada con un numero cualquiera de argumentos; los que sobraban se evaluaban y sus valores descartados, y los que faltaban eran sustituidos por NIL. Esta fue una de las principales diferencias irreconciliables que separaba los dos grupos en esa reunión del 1974. Los fieles a MacLisp decían del InterLisp que era indisciplinado y los fans del InterLisp decían del MacLisp que era demasiado inflexible, ya que ellos tenían la opción de introducir argumentos opcionales, opción que no estuvo disponible en MacLisp hasta la llegada del "&optional" y otras sintaxis complejas de lambda-listas.

Una de las más innovadoras extensiones del lenguaje que introdujo InterLisp fue la pila de ‘spaghettis’. El problema de retenciones (por cierres) del entorno de definición de funciones dinámicas en presencia de variables especiales nunca fue completamente resuelto hasta la invención de las pilas de “spaghettis”.

La idea tras las pilas de “spaghettis” es generalizar la estructura de las pilas para que sea mas como un árbol, con varias ramas del árbol sujetas a retención siempre que un puntero a esa rama este detenido.

MacLisp e InterLisp nacieron a la vez, y duraron mas o menos lo mismo. Diferían en sus grupos de usuarios, aunque descripciones genéricas de ambos grupos no los distinguirían: ambos grupos estaban formados por investigadores de laboratorios de Inteligencia Artificial patrocinados principalmente por ARPA (mas tarde DARPA), y estos investigadores fueron educados por el MIT, CMU y la universidad de Stanford.

Las implementaciones principales se ejecutaban en las mismas máquinas. Las diferencias principales vinieron de sus distintas aproximaciones filosóficas al problema de la programación. También había distintas necesidades en cada grupo: los usuarios de MacLisp (sobre todo el grupo de Mathlab), deseaban usar un entorno de programación menos integrado para conseguir un compilador mas optimo y tener una mayor porción del espacio de direcciones del PDP-10 para su uso propio. Los usuarios de InterLisp preferían concentrarse en la tarea de codificación usando un entorno de desarrollo mucho más completo e integrado.

5.2.4 Principio de los 70

Aunque MacLisp e InterLisp dominaron los 70, hubo otros dialectos Lisp en uso durante estos años. La mayoría eran mas parecidos a MacLisp que a InterLisp. Los dos mas extendidos fueron Standard Lisp y Portable Standard Lisp. Standard Lisp fue definido por Hearn y Griss junto con sus estudiantes y colegas. Su motivación era definir un subconjunto de Lisp 1.5 y otros dialectos Lisp que pudieran servir como medio para portar programas Lisp, mas concretamente el sistema de álgebra simbólica REDUCE.

Mas tarde, Eran y sus colegas se dieron cuenta de que para conseguir un buen rendimiento, necesitaban mas control sobre el entorno y el compilador, por lo que Portable Standard Lisp nació. PSL fue una nueva y completa implementación de Lisp, y supuso un esfuerzo pionero en la evolución de la tecnología de compilación de Lisp. A finales de los 70, PSL se ejecutaba, y muy correctamente, en mas de una docena de tipos de computadores diferentes.

PSL fue implementado usando 2 técnicas. La primera, usando un lenguaje de implementación de sistemas llamado SYSLISP, que se usaba para codificar operaciones 'a pelo', representaciones sin tipo. Segundo, uso un conjunto parametrizado de macros de traducción a nivel ensamblador, llamadas c-macros. El Portable Lisp Compiler (PLC) compilaba código Lisp en un lenguaje ensamblador abstracto.

De la segunda mitad de los 70 y hasta mediados de los 80, el entorno PSL fue mejorado adaptando editores y otras herramientas. En particular, un editor multi-ventanas basado en Emacs llamado Emode fue desarrollado de tal manera que permitía una edición inteligente y el pasar información del editor al Lisp y viceversa. Mas tarde, una versión ampliada de Emode llamada Nmode fue desarrollada por Griss y sus colegas en Hewlett-Packard en California. Esta versión de PSL y Nmode fue comercializada por HP a mediados de los 80.

En Stanford, en los 60, una versión temprana de MacLisp fue adaptada para sus PDP-6; este Lisp se llamó Lisp 1.6. Fue reescrita por Allen y Lynn Quam; mas tarde, Diffie introdujo mejoras en el compilador. Lisp 1.6 desapareció a mediados de los 70, siendo uno de los últimos remanentes de la era del Lisp 1.5.

UCI Lisp fue una versión extendida de Lisp 1.6 en la que se introdujeron como mejoras un editor del estilo del de InterLisp y otros entornos de programación. UCI Lisp fue usado por algunas personas en Stanford a principios de la segunda mitad de los 70, al igual que en otras instituciones.

En 1976, la versión de MacLisp del MIT fue ‘portada’ al sistema operativo WAITS por Gabriel, en el laboratorio de Inteligencia Artificial de Stanford (SAIL), que era dirigido en esa época por John McCarthy.

5.2.5 La muerte de los PDP-10

A mediados de los 70 se hizo evidente que el espacio de direcciones de 18 bits de los PDP-10 no proveía suficiente espacio de trabajo a los programas de Inteligencia Artificial. La gama de computadores PDP-10 (KL-10's y DEC-20's) fue alterada para permitir un esquema extendido de direccionamiento, en el que múltiples espacios de direcciones de 18 bits pudieran ser direccionados mediante indexación relativa de registros base de 30 bits.

Sin embargo, esta adición no era una sencilla expansión a la arquitectura en lo que al implementador de Lisp se refería; el cambio de 2 punteros por palabra a 1 puntero por palabra requería rediseñar completamente casi todas las estructuras internas de datos. Solamente 2 Lisps estaban diseñados para soportar direccionamiento expandido: Elisp y PSL.

Una propuesta al problema de espacio de direcciones fue construir maquinas Lisp de propósito especial(máquinas Lisp, Sección 2.6). La otra propuesta fue usar computadores comerciales con un mayor espacio de direcciones; el primero de estos fue el VAX (de DEC).

Los VAX presentaban pros y contras para los implementadores de Lisp. El conjunto de instrucciones del VAX era bastante bueno para implementar las primitivas de nivel bajo de Lisp eficientemente, pero requería un diseño muy inteligente (quizás demasiado) de las estructuras de datos. Sin embargo, la llamada a funciones de Lisp no podía ser correctamente implementada por las instrucciones de llamada a funciones de las que disponía el VAX. Sin embargo, el VAX, a pesar de su 'teóricamente grande' espacio de direcciones, estaba aparentemente diseñado para ser usado por muchos programas pequeños, no por varios grandes. Las tablas de paginas ocupaban demasiada memoria; La paginación en memoria de grandes programas Lisp fue un problema nunca resuelto completamente en los VAX. Finalmente, estaba el mayor problema de todos; mas de una de las mayores implementaciones de Lisp de ese momento tenían una gran base en lenguaje ensamblador que era muy difícil de portar.

Los principales dialectos Lisp para VAX desarrollados a finales de los 70 fueron VAX InterLisp, PSL (portado al VAX), Franz Lisp y NIL.

Franz Lisp fue escrito para habilitar la investigación de la álgebra simbólica en la universidad de California, en Berkeley, bajo la supervisión de Fateman, quien fue uno de los principales implementadores de Macsyma en el MIT. Fateman y sus estudiantes comenzaron con una versión del Lisp del PDP-11 escrita en Harvard, y la extendieron a un Lisp similar al MacLisp que se estuvo ejecutando en casi todos los computadores con Unix, gracias a que Franz Lisp estaba escrito casi completamente en C.
NIL, el cual intento ser un sucesor del MacLisp, fue diseñado por White, Steele y otros en el MIT, bajo la influencia del Lisp de las máquinas Lisp, también desarrolladas en el MIT. NIL es un acrónimo de "New Implementation of Lisp", y causó gran confusión

debido al papel tan importante que juega en el Lisp el símbolo NIL. NIL era un gran Lisp, y pronto su implementación se centró en el lenguaje ensamblador del VAX.

En 1978, Gabriel y Steele empezaron a implementar NIL en el Mark S-1 IIA, una supercomputadora que había sido diseñada e implementada por el "Lawrence Livermore National Laboratory". Estas personas cooperaron mucho en este proyecto en parte porque Steele alquilo una habitación en la casa de Gabriel. Este Lisp nunca fue completamente funcional, pero sirvió para hacer pruebas para adaptar técnicas avanzadas de compiladores a la implementación de Lisp. En particular, el trabajo generalizó las técnicas de computación numérica del compilador de MacLisp y las unificó con estrategias de localización de registros.

En Francia, a mediados de los 70, Greussay desarrollo un Lisp basado en un interprete, llamado VLisp. A nivel del dialecto base InterLisp, introdujo un par de conceptos bastante interesantes, tales como la cronología, que es una especie de entorno dinámico para la implementación de interrupciones y funciones de entorno como *trace* y *step*, creando diferentes encarnaciones del evaluador. El énfasis de VLisp estaba en tener un interprete rápido. El concepto era proveer una maquina virtual que fuese usada para transportar el evaluador. Esta maquina virtual estaba al nivel de lenguaje ensamblador y fue diseñada para un fácil porte y una ejecución eficiente. El interprete obtuvo una buena parte de su velocidad de 2 cosas: un rápido despachador de funciones que usaba un espacio de tipo de funciones que distinguía varias funciones de distinta aridad y la eliminación de la recursión de cola. (VLisp fue el primer Lisp que eliminó la recursión de cola. Otros dialectos de la época, incluyendo a MacLisp, eliminaban la recursión de cola sólo en ciertas situaciones, de una manera no predecible). VLisp fue el precursor de Le_Lisp, uno de los más importantes dialectos de Lisp en Francia y Europa durante los 80; aunque los dialectos eran distintos, compartían algunas técnicas de implementación.

A finales de los 70, no había en el horizonte nuevas maquinas comerciales apropiadas para Lisp; parecía que el VAX era lo único que había. A pesar de los años de valioso apoyo por parte de Glenn Burke, VAX NIL nunca consiguió una gran aceptación. Las estaciones de propósito general (aquellas diseñadas para ejecutar otros lenguajes que no fueran Lisp) y los computadores personales no habían aparecido todavía. Para muchos implementadores y usuarios de Lisp, la situación comercial del hardware lo tenia crudo.

Pero de 1974 hacia delante ha habido muchas investigaciones y proyectos de prototipos para máquinas Lisp, y a finales de la década parecía que las maquinas Lisp eran el futuro.

5.2.6. Máquinas LISP

Aunque las ideas para hacer una maquina Lisp habían sido discutidas informalmente antes, Peter Deutsch parece ser que fue quien publicó la primera propuesta concreta. Deutsch ideó una visión básica de una minicomputadora mono-usuario que estaría microcodificada especialmente para ejecutar Lisp y soportar un entorno de desarrollo Lisp. Las 2 ideas clave del articulo de Deutsch que han tenido un impacto duradero en Lisp son:

1) dualidad del acceso a carga y almacenamiento basado en funciones y
2) la representación compacta de listas lineales usando el CDR.

Todos los dialectos Lisp de esa época tenían una función CAR para leer el primer componente de un par y una función interna RPLACA para escribir ese mismo componente. Deutsch propuso que funciones como CAR deberían tener un modo de carga y otro de almacenamiento a la vez. Sí (f a1..an) es llamada en modo carga, debería devolver un valor; si es llamado en modo almacenamiento, como en (f a1..an v), el nuevo valor v debería ser almacenado en la posición que será accedida por la versión de carga. Deutsch indicó que debería haber 2 funciones internas asociadas con cada función que accede a memoria, una para cargar y otra para almacenar, y que la función de almacenamiento debería ser llamada cuando la función es mencionada en algunas instrucciones especiales.

5.2.6.1. Las Máquinas Lisp del MIT: 1974-1978

Richard Greenblatt comenzó el proyecto de las máquinas Lisp en el MIT en 1974; su propuesta cita al documento de Deutsch. El proyecto también incluía a Thomas Knight, Jack Holloway y Pitts Jarvis. La maquina que diseñaron fue llamada CONS, y su diseño estaba basado en ideas del microprocesador Xerox PARC Alto, el PDP-11/40 de DEC, las extensiones al PDP-11/40 hechas CMU, y algunas ideas de modificación de instrucciones sugeridas por Sam Fuller en DEC.

Esta maquina fue diseñada para tener una buena actuación soportando una versión de Lisp compatible con MacLisp, pero aumentado con la sintaxis de declaración de argumentos de "Muddle-Conniver". Sus metas incluían un coste no demasiado alto (menos de 70.000 dólares por maquina), operación mono-usuario, un lenguaje objetivo común con la estandarización de llamadas a procedimientos, un almacenamiento 3 veces mejor que el del PDP-10 para lenguajes compilados, apoyo del hardware para comprobación de tipos y recolección de basura, una implementación más grande en Lisp y un entorno de programación mejorado que explotara el mapeado de bits.

La maquina CONS fue construida; entonces una versión mejorada llamada CADR (un chiste, CADR significa 'el segundo' en Lisp) fue diseñada y unas docenas de estas maquinas fueron construidas. Se formaron dos compañías que manufacturaron clónicos del CADR, LISP Machine, Inc. (LMI) y Symbolics. Poco después, Symbolics introdujo su gama de 3600's, que pronto se convirtió en la industria líder en máquinas Lisp durante los siguientes 5 años.

Mientras que Greenblatt había prestado especial atención a proveer de mecanismos hardware que apoyaran la rápida recolección de basura, Las maquinas Lisp del MIT no implementaron un recolector de basura hasta pasados bastante años; pero incluso cuando los recolectores de basura aparecieron, muchos usuarios prefirieron desconectarlo.

El primer dialecto Lisp de las maquinas Lisp del MIT era muy parecido al MacLisp. Consiguió su meta de soportar programas MacLisp con un mínimo esfuerzo de portabilidad. Las extensiones respecto al MacLisp más importantes fueron:

?? Un entorno de programación mejorado, consistente principalmente en un compilador residente, recursos para el 'debugging' y un editor de texto. El editor de texto era un clon del EMACS.
?? Listas lambda complejas, que incluían &optional, &key, &rest y &aux.
?? Localizadores, que proveían de la habilidad que tiene el C de apuntar a la mitad de una estructura.
?? DEFMACRO, para definir macros de una manera mucho más conveniente.
?? Backquote, una sintaxis para construir estructuras de datos rellenando un template (patrón).
?? Grupos de pilas, que dieron recursos de co-rutinas.
?? Valores múltiples, la posibilidad de pasar mas de un valor de vuelta desde la invocación de una función sin tener que construir una lista.
?? DEFSTRUCT, recurso para definir registros.
?? Cierres sobre variables especiales. Estos cierres no eran como los que tenia Scheme; las variables capturadas por el entorno debían ser explícitamente listadas y la invocación del cierre requería el 'emparejamiento' de variables SPECIAL a los valores salvados.
?? Flavors, sistema de programación no jerárquico y orientado a objetos fue diseñado por Cannon y Moon e integraron en él partes del entorno de programación de las maquinas Lisp (el sistema de ventanas, en particular, fue escrito usando Flavors)
?? SETF, recurso para variables generalizadas.

El uso de SETF hasta el Common Lisp (el ultimo y más popular de los dialectos Lisp) puede ser trazado desde el ZetaLisp de Symbolics y el MacLisp hasta la influencia de las maquinas Lisp del MIT y entonces volver a la propuesta de Greenblatt, a Peter Deutsch y a Alan Kay.

El tratamiento uniforme de los accesos (lectura y escritura de estados) ha hecho al Common Lisp mas uniforme de lo que seria en otro caso. Si la instrucción general de una operación de lectura es (f ...), entonces la instrucción de la escritura es (setf(f ...) nuevo_valor), y eso es todo lo que el programador necesita saber sobre la lectura y escritura de datos. Se abandona el tener que recordar tanto las funciones de lectura como las de lectura, y el orden de aparición de los argumentos.

Mas tarde, en el Common Lisp Object System (CLOS), esta idea fue extendida a métodos. Si un método M especificado para actuar como un lector, y es invocado como (M objeto), entonces es posible definir un método escritor que sea invocado (setf (M objeto) nuevo_valor).

Que CLOS se ajuste tan bien a este lenguaje no es sorprendente. Si Alan Kay era la inspiración de la idea alrededor de 1973, estaba en medio de su compromiso con SmallTalk, un lenguaje orientado a objetos. Lectura y escritura son ambos métodos que un objeto puede soportar, y la adaptación de CLOS de la versión de Lisp de la visión de Kay fue una simple reinvención de la génesis orientada a objetos de la idea.

5.2.6.2. Las máquinas Lisp de Xerox: 1973-1980

El Alto era una maquina microcodificable desarrollada en 1973 y usada para la experimentación de computación personal en Xerox, usando InterLisp y otros lenguajes tales como Mesa. La versión Alto del entorno InterLisp primero fue usada en el XEROX PARC y en la universidad de Stanford alrededor de 1975.

El Alto tenia un equipamiento estándar de 64k palabras de memoria de 16 bits ampliables a 256k palabras, lo que era bastante para una computadora mono-usuario pero sólo la mitad de memoria de un PDP-10. La máquina demostró ser de baja potencia para el entorno InterLisp.

El Alto fue también usado para construir el primer entorno de SmallTalk (el 'interim Dynabook') y en esto si tuvo un éxito relativo.
En 1976 el Xerox PARC emprendió el diseño de una máquina llamada el Dorado(o Xerox 1132), que era una máquina ECL('Emiter Coupled Logic', una tecnología de implementación de lógica digital rápida para esa época), diseñada para reemplazar al Alto. Un prototipo disponible en 1978 ejecutó todo el software de Alto. Un rediseño fue completado en 1979 y algunos fueron construidos para ser usados en Xerox y en ciertos experimentos de la universidad de Stanford. El Dorado estaba específicamente diseñado para interpretar códigos byte producidos por compiladores y así es como el Dorado ejecutó software de Alto. El dorado era básicamente una máquina de emulación.

InterLisp fue portado a esta maquina usando el modelo de maquina virtual de InterLisp. El Dorado era más rápido ejecutando Lisp que un KL-10 ejecutando InterLisp mono-usuario y habría demostrado ser una maquina Lisp muy buena en caso de haber sido ampliamente comercializado.

InterLisp fue similarmente portado a una maquina más pequeña y barata llamada Dolphin (1100), que fue comercializada como máquina Lisp a finales de los 70. La actuación del Dolphin fue mejor que la del Alto, pero suficientemente mala para que la máquina nunca tuviera éxito como 'motor' de Lisp.

A principio de los 80, Xerox construyó otra máquina llamada Dadelion (1108), que era considerablemente más rápida que el Dolphin, pero no tan rápida como el Dorado. Debido a que el nombre de estas 3 máquinas comenzaba por D, se las comenzó a conocer colectivamente como máquinas-D.

Todas las máquinas Lisp de Xerox usaban un recolector de basura que era incremental. Se ejecutaban unos pasos de la recolección de basura cada vez que lo que se tenia que almacenar se ubicara en su posición de memoria. Por lo tanto se hacía una pequeña cantidad de recolección de basura por unidad de tiempo.

A finales de los 70, BBN construyó también una maquina, la Jericho, que era usada como un motor InterLisp. Permaneció interna a BBN, no se comercializó.

5.2.6.3. Comentarios a los inicios de la historia de las máquinas Lisp

Liberadas de los problemas del espacio de direcciones que tenían las arquitecturas previas, todas las compañías de maquinas Lisp produjeron

implementaciones de Lisp muy expandidas, añadiendo gráficos, capacidades de ventanas e interacción del ratón a sus entornos de programación. El mismo lenguaje Lisp, particularmente en la maquina de Lisp del MIT, también creció en el numero y complejidad de sus características. Aunque algunas de estas ideas se originaron en otro contexto, la comunidad de Lisp las adoptó y sus maquinas tuvieron mucho éxito.

5.2.7 Los LISPs de IBM: LISP360 y LISP 370

Aunque los primeros Lisps fueron implementados en computadores IBM, IBM desapareció de la escena de Lisp a finales de los 60 por dos razones: una mejor cooperación entre el MIT y DEC, y una continua disputa entre el MIT e IBM.

A principio de los 60, DEC (Digital Equipment Corporation) discutió con el MIT las necesidades que éste tenia respecto a los computadores y las características fueron añadidas para ayudar a las implementaciones de Lisp.

A principios de los 60, IBM y el MIT discutieron sobre quien había inventado la memoria del núcleo, e IBM insistía en hacer prevalecer su patente frente a la del MIT. El MIT contestó declinando usar equipos IBM tan extensamente como había hecho en el pasado. Esto dio mas ímpetu a usar equipos DEC, especialmente para Lisp e Inteligencia Artificial.

No obstante, Lisp fue implementado por IBM para el IBM 360 y llamado Lisp360. Cuando el IBM 370 apareció, la implementación del Lisp370 (mas tarde llamado Lisp/VM) comenzó.

Lisp360 era básicamente una colección Lisp, y fue usado extensamente para enseñar en las universidades.

Lisp370 comenzó con la definición de un núcleo Lisp basado en una semántica formal expresada en el modelo SECD. El proyecto de Lisp370 estaba bajo la dirección de Fred Blair(quien desarrolló la definición de SECD) en el centro de investigación de IBM Thomas J. Watson, en Yorktown Heigths (New York).

Lisp370 soportaba tanto ligaduras especiales como ligaduras léxicas, así como cierres sobre variables léxicas y especiales, usando una técnica similar a la de pilas "spaguetti". Jon L. White pasó todo 1977 trabajando en Yorktown Heights en el Lisp370, y luego volvió al MIT; su experiencia en IBM tuvo alguna influencia en el posterior desarrollo de MacLisp y en el dialecto NIL.

Lisp370 tenia un entorno de programación parecido al InterLisp escrito para operar tanto en 'hardcopy terminal' como en los 'IBM 3270 character-display terminals'.

Mas tarde aparecieron otros Lisps en la gama de computadores 360/370, incluyendo varios Common Lisps. El primer Common Lisp que apareció en el IBM 370 fue escrito por Intermetrics, incluyendo una buena tecnología de compilación. Sin embargo, el Lisp no estaba bien construido y nunca tuvo mucho impacto el mundo Lisp. Algunos años después, Lucid portó su Common Lisp a los 370, bajo contrato con IBM. IBM abandonó su apoyo al Lisp370 a finales de los 80 a favor del Common Lisp.

5.2.8 SCHEME: 1975-1980

El dialecto de Lisp conocido como Scheme fue originalmente un intento de Sussman y Steele durante el otoño del 75 de explicarse a ellos mismos algunos aspectos de la teoría de actores de Carl Hewitt como un modelo de computación. El modelo de Hewitt estaba orientado a objetos (e influenciado por SmallTalk); cada objeto era una entidad computacionalmente activa capaz de recibir y reaccionar ante mensajes. Los objetos eran llamados actores, y los mismos mensajes eran también actores. Un actor podía saber sobre otros actores y enviarles mensajes: uno podía enviar a un actor llamado 'factorial' el numero 5 y otro actor al que enviar el valor de la computación realizada.

Sussman y Steele tenían problemas en entender algunas de las consecuencias del modelo de Hewitt y del diseño del lenguaje, por lo que decidieron construir una implementación de juguete de un lenguaje actor para experimentar con ella. Usando MacLisp como entorno de trabajo, escribieron un pequeño interprete de Lisp y le añadieron los mecanismos necesarios para crear actores y que se pudieran mandar mensajes. El Lisp de juguete proveería de las primitivas necesarias para implementar el comportamiento interno de las primitivas 'actores'.

El paso de mensajes podía ser expresado sintácticamente como una invocación de función. La diferencia entre un actor y una función seria detectada en una parte del interprete conocida como 'apply'. Una función devolvería un valor, pero un actor nunca lo devolvería; en vez de esto, invocaría una 'continuación', otro actor que sabe sobre eso.

Sussman y Steele quedaron muy contentos con su implementación de juguete de actores y la llamaron 'Schemer', deseando que pudiera ser desarrollado como otro lenguaje de Inteligencia Artificial, siguiendo la tradición de Planner y Conniver. Sin embargo, el sistema operativo ITS tenia una limitación de 6 caracteres en los nombres de ficheros, por lo que el nombre fue truncado a 'Scheme', y así se ha quedado.

Se produjo un descubrimiento crucial: el hecho de que las funciones debieran devolver valores y que los actores no, no implicaba ninguna diferencia en su implementación. La diferencia radicaba únicamente en las primitivas usadas para codificar sus cuerpos. Sussman y Steele concluyeron que los cierres y los actores eran el mismo concepto.

En 1976 Sussman y Steele publicaron dos artículos mas que exploraban la semántica de los lenguajes de programación usando Scheme. "Lambda: The Ultimate Imperative" demostró que un gran numero de estructuras de control podían ser modeladas en Scheme. "Lambda: The Ultimate Declarative" se concentró en la naturaleza de lambda como un constructor que renombrara; también comparó Scheme a l PLASMA de Hewitt.

Steele y Sussman siguieron publicando e investigando el Scheme.

Se trabajó mas extensamente en implementaciones de Scheme en Yale y más tarde en el MIT por Jonathan Rees, Norman Adams y otros. Esto llevó al dialecto de Scheme conocido como T; este nombre también tenía mucha gracia ya que T era a

Scheme lo que NIL a MacLisp. La meta era hacer un dialecto simple con una implementación especialmente eficiente.

El diseño de T, sin embargo, representó una ruptura consciente no sólo con la tradición de Lisp, sino también con las primeras versiones de Scheme. T fue inicialmente orientado a los VAX (bajo Unix y VMS) y a las estaciones de trabajo Apollo. Casi todo el sistema estaba escrito en T y hecho más robusto por el compilador de T, TC. El evaluador y el recolector de basura, en particular, estaban escritos en T y no en lenguaje máquina. El proyecto de T comenzó con una versión del compilador Lisp S-1, y realizó mejoras substanciales; en el transcurso de su trabajo identificaron varios fallos en el compilador S-1 y en el informe original en RABBIT.

Una segunda generación del compilador T, llamada ORBIT implementó estrategias de optimización específicas de Lisp.

5.2.9 Preludio a COMMON LISP: 1980-1984

En la primavera de 1981, la situación era como se describe. 2 compañías de máquinas Lisp habían surgido del proyecto de máquina Lisp del MIT: Lisp Machine Inc. (LMI) y Symbolics, Inc. La primera fue fundada principalmente por Greenblatt y la última por un gran grupo del que destacaba Moon. Inicialmente ambas produjeron el CADR, la segunda máquina Lisp del MIT, y obtuvieron la licencia del software de la máquina Lisp del MIT bajo un arreglo que incluía dar las mejoras hechas por las compañías. Symbolics pronto se embarcó en diseñar y construir una máquina Lisp, la 3600. El lenguaje Lisp de la máquina Lisp había evolucionado mucho desde la primera definición publicada en 1978, adquiriendo una gran variedad de características, sobre una extensión orientada a objetos llamada Flavors.

Las máquinas de Lisp de Xerox (Dolphin y Dorado) ejecutaban InterLisp y estaban siendo usadas en laboratorios de investigación situados en la costa oeste. BBN estaba construyendo su máquina InterLisp, la Jericho, y un porte de InterLisp a VAX estaba en camino.

En el MIT un proyecto había comenzado a definir e implementar un descendiente de MacLisp llamado NIL en computadores VAX y S-1.

En CMU, Scott Fahlman y sus colegas y estudiantes estaban definiendo e implementando un dialecto parecido a MacLisp llamado Spice Lisp, para ser implementado en las máquinas SPICE (Scientific Personal Integrated Computing Environment).

5.2.10 Primer COMMON LISP

Si no hubiera habido una consolidación en la comunidad Lisp en este punto, Lisp habría muerto. ARPA no estaba interesada en proporcionar fondos a una serie de proyectos Lisp que competían innecesariamente entre sí.

5.2.10.1. Fuera del caos de MacLisp

En abril de 1981, ARPA preparó una reunión para la comunidad Lisp, en la que los grupos implementadores se juntaron para discutir el futuro de Lisp.

El día antes de la reunión de ARPA, parte de la comunidad InterLisp se juntó para discutir como presentar una comunidad de InterLisp sana y funcionando en una gran variedad de máquinas. La idea era potenciar una visión de un lenguaje estándar (InterLisp) y un entorno estándar existente en un cada vez mayor numero de computadores distintos.

El día de la reunión, la comunidad InterLisp se presentó exitosamente como un grupo coherente con una meta y una misión.

Como se encontraba la comunidad MacLisp puede comentarse con una anécdota: Scott Fahlman dijo "La comunidad MacLisp no se encuentra en un estado caótico. Consiste en 4 grupos bien diferenciados que se mueven en 4 direcciones distintas".

Gabriel presenció tanto la exposición de InterLisp como el 'espectáculo' de MacLisp, y no podía creerse que las diferencias entre los grupos de MacLisp fueran insuperables, por lo que comenzó a intentar vender la idea de alguna clase de cooperación entre los grupos.

Primero se acercó a White. Después, Gabriel y White se acercaron a Steele, luego a CMU y se afiliaron al proyecto SPICE Lisp. Los 3 estaban asociados de un modo u otro con el proyecto S-1 NIL. Unos meses después, Gabriel, Steele, White, Fahlman, William Scherlis y Rodney Brooks se reunieron en el CMU y se discutieron algunos de los detalles técnicos del nuevo Lisp. El nuevo dialecto debía tener las siguientes características:

- ?? Alcance léxico, incluyendo cierres completos
- ?? Múltiples valores, como los del Lisp de las máquinas Lisp, pero quizás con algunas modificaciones para situaciones en que se fuerce un único valor.
- ?? Valor y elementos de función separados(un Lisp 2)
- ?? DEFSTRUCT
- ?? SETF
- ?? Números en punto flotante atractivos, incluyendo números complejos y racionales (esta fue la principal influencia del Lisp de S-1)
- ?? Declaraciones de lambda listas complejas, parecidas a las del Lisp de las máquinas Lisp
- ?? Cierres no dinámicos

Después de un día y medio de discusiones técnicas, este grupo se fue a un bar a comer. Durante y después del almuerzo, el tema del nombre para el Lisp surgió, y nombres tan obvios como NIL y Spice Lisp se propusieron y fueron rechazados (como si se le diera mucha importancia a un grupo y no la suficiente a los otros), y nombres no tan obvios como Yu-Ilsiang Lisp también fueron propuestos y también fueron rechazados.

El nombre que parecía mejor era Standard Lisp, pero otro dialecto era conocido por ese nombre en ese momento. En la búsqueda de palabras similares, el nombre Common Lisp surgió. Gabriel señaló que no era un buen nombre porque estaban intentado definir un Lisp elitista y Common Lisp sonaba demasiado a "Lisp para el hombre vulgar".

La discusión por el nombre continuó en la cena, en el Pleasure Bar (restaurante italiano en otro barrio de Pittsburgh), pero no se sacó nada en claro.

Mas tarde, por e-mail, Moon se refirió a 'whatever we call this common Lisp', y esta vez, con la tristeza y consternación de que no encontrarían un nombre mejor, fue seleccionado.

El siguiente paso era contactar con mas grupos. La clave eran las compañías de máquinas Lisp, a las que se acercarían por ultimo. Además, Gabriel se presentó voluntario para visitar el grupo de Franz Lisp en Berkeley y al grupo de PSL en Salt Lake City.

El grupo PSL no se unió al completo al grupo de Common Lisp, y el grupo de Franz no se unió en absoluto. El grupo de Lisp370 no fue invitado a unirse. La comunidad de InterLisp envió un observador. ARPA apoyó el esfuerzo.

Casi todo el trabajo se hacía por medio de correo electrónico, que era automáticamente archivado. De hecho, este fue el primer lenguaje cuyo esfuerzo por la estandarización se hizo casi completamente por e-mail.

Una reunión con Symbolics y LMI tuvo lugar en Symbolics en Junio del 81. Steele y Gabriel condujeron desde Pittsburgh hasta Cambridge para asistir a la reunión. La reunión alternó entre una profunda discusión técnica de lo que debería estar en el dialecto y una discusión política sobre porqué el nuevo dialecto era bueno. Desde el punto de vista de las compañías de máquinas Lisp, la acción estaba en las máquinas Lisp, y el interés en un mismo dialecto ejecutándose en muchos lugares parecía académico. Por supuesto, había razones de negocios para tener el mismo dialecto ejecutándose en muchos lugares, pero la gente con sentido comercial no fue a la reunión.

Al final, ambas compañías de máquinas Lisp decidieron unir sus esfuerzos, y el grupo de Common Lisp fue formado por:

Bawden, Brooks, Bryan, Burke, Cannon, Carrette, Dill, Fahlman, Fateman, Feinberg, Foderaro, Gabriel, Greenblatt, Griss, McCarthy, Moon, Steele, White y muchos mas.

Como compromiso se acordó que merecía la pena definir una familia de lenguajes de tal forma que cualquier programa escrito en el lenguaje definido se pudiera ejecutar en cualquier lenguaje de la familia. Así, un subconjunto iba a ser definido, aunque no estaba claro si alguien implementaría el subconjunto directamente.

Algunas de las características de las máquinas Lisp que se dejaron caer fueron Flavors, sistemas de ventanas, multiprocesamiento (incluyendo multitareas), gráficos y localizadores.

Durante el verano de 1981 Steele trabajó en un manual de Common Lisp basado en el manual del Spice Lisp. Al principio de su trabajo fue ayudado por Brooks, Scherlis y Gabriel. Scherlis ayudó específicamente en el sistema de tipos, sobre todo dándole avisos informales a Steele. Gabriel y Steele discutían regularmente algunos temas porque Gabriel estuvo viviendo en la casa de Steele ese verano.

El borrador, llamado la Edición de queso suizo (porque estaba lleno de agujeros) era en parte una votación en la que varias alternativas y preguntas de si/no fueron propuestas. A través de un proceso de discusión y votación por medio de e-mails, las primeras decisiones clave fueron tomadas. Esto fue seguido por una reunión cara a cara en Noviembre de 1981, donde las decisiones finales de las cuestiones más complejas fueron fijadas.

Esto llevó a otra ronda de refinamiento con otras versiones y votaciones.

Las discusiones por e-mail eran a menudo propuestas, discusiones y contrapropuestas. Ejemplos de código de software existente o propuestas de nuevas sintaxis eran a menudo intercambiadas. Todos los e-mails eran archivados en línea, por lo que todo estaba disponible para un repaso rápido para la gente que lo deseara.

Éste estilo tenía algunos problemas. El principal era que no era posible observar las reacciones del resto de la gente, por ejemplo, para ver si algún punto les enfadaba, lo que significaba que ese punto les resultaba importante. No había manera de ver si un argumento había llegado muy lejos o tenia poco apoyo. Esto significaba que un cierto tiempo había sido malgastado y que argumentos escritos cuidadosamente eran requeridos para hacer algo.

Una vez que el proceso comenzó, la aproximación al problema cambió de ser solo una consolidación de los dialectos existentes, que era la dirección obvia que tenía que tomar, a intentar diseñar ‘ The Right Thing’. Algunas personas tenían la opinión de que esa era la época idónea de replantearse algunos temas importantes y abandonar la meta de la compatibilidad estricta con MacLisp, que era tan importante para el diseño del Lisp de máquinas Lisp. Algunos temas tales como si NIL era tanto un símbolo como una parte de CONS, no fueron repensadas, aunque se llegó al acuerdo de que lo debería ser.

Un tema que surgió de los primeros y que merece la pena comentar, porque es el corazón de uno de los ataques al Common Lisp, que surgió durante el trabajo de ISO en Lisp. Es el tema de la modularización, que tenía 2 aspectos:

1)si el Common Lisp debería ser dividido en un lenguaje central y varios módulos, y

2)si debería haber una división entre las llamada páginas amarillas, blancas y rojas. Estos temas parecen haber sido mezclados en la discusión.

Las páginas blancas se referían al manual propiamente dicho, y cualquier cosa que se encuentre en las páginas blancas debe estar implementado de alguna manera por un Lisp cuyos desarrolladores lo llamen Common Lisp. Las páginas amarillas se refieren a paquetes de implementación independientes que pueden ser cargados, tales como Trace y paquetes de subrutinas científicas. Las páginas rojas se pretendía que describieran la implementación de rutinas dependientes, tales como controladores de dispositivos.

Common Lisp no fue estructurado como un lenguaje central y varios módulos, ni se materializaron las páginas rojas, blancas ni amarillas.

Cuatro escritos más se realizaron, llamados 'The Colander Edition (Con más agujeros que antes, pero son más pequeños!!)', 'The Laser Edition (se supone que es completamente coherente)', 'The Excelsior Edition (Excelsior no es sólo una exhortación, sino que también se le llama así al papel que se usa para envolver cosas)' y 'The Mary Poppins Edition (Prácticamente perfecta en todo!!)'

Prácticamente todas las decisiones técnicas fueron completadas a principios del 83, pero hasta el año siguiente no se publicó el libro 'Common Lisp: The Language', a pesar de lo rápido que trabajaron en Digital Press.

Las metas declaradas del grupo de Common Lisp eran:

- **Generalidad** : Common Lisp se originó en un intento de enfocar el trabajo de varios grupos de implementación, cada uno de los cuales estaba construyendo sucesivas implementaciones de MacLisp para distintas computadoras. Mientras que las diferencias entre las distintas implementaciones continuamente forzarían incompatibilidades, Common Lisp debería servir como un dialecto común para estas implementaciones.
- **Portabilidad**: Common Lisp debería excluir características que no puedan ser fácilmente implementadas en una ancha gama de computadores. Esto debería servir para excluir características que requirieran microcódigo o hardware por un lado y características generalmente requeridas para 'Stock hardware' por otro, como las declaraciones.
- **Consistencia**: el interprete y el compilador deberían exhibir la misma semántica.
- **Expresividad**: Common Lisp debería tomar lo mejor de una gran variedad de dialectos, no solo de MacLisp o InterLisp.
- **Compatibilidad**: Common Lisp debería ser compatible con ZetaLisp, MacLisp e InterLisp, en ese orden.
- **Eficiencia**: debería ser posible escribir un compilador que optimizara para Common Lisp
- **Poder**: Common Lisp debería ser un buen lenguaje constructor de sistemas, adaptable para escribir paquetes a nivel de usuario parecidos a InterLisp, pero no proveerá de esos paquetes.
- **Estabilidad**: Common Lisp debería evolucionar lentamente pero sin pausa.

5.2.10.2. - Retumbos tempranos

El proceso de definición de Common Lisp no era un camino de rosas. Existía el sentimiento entre algunos de que estaban siendo encarrilados o de que las cosas no estaban yendo bien. El grupo de InterLisp no estaba muy contento con el modo en que estaban saliendo las cosas, opinaban que Common Lisp no iba a aportar muchas características a favor de InterLisp.

Parte del problema era la fuerza que tenían las compañías de máquinas Lisp y la necesidad del grupo de Common Lisp de mantenerlos interesados, lo que les otorgaba mucho poder.

5.2.10.3. Crítica al Common Lisp

En el simposio ACM de programación Lisp y funcional de 1984, Brooks y Gabriel rompieron los esquemas y presentaron el articulo 'Crítica al Common Lisp'. Esto fue muy sorprendente, ya que Gabriel y Brooks fueron fundadores de la compañía que se proponía convertir en la primera con Common Lisp. Fahlman, escuchando el discurso de Gabriel, le llamó traidor.

Este artículo fue sólo el primero de una larga serie de críticas al Common Lisp; muchas de estas críticas apuntaban a la primera. Los puntos principales de este artículo desvelaban una serie de problemas que persiguieron a Common Lisp a lo largo de esa década.

Este tema reapareció en la historia del Common Lisp debido al surgimiento de algunos 'UnCommon Lisps'. Cada uno de estos 'UnCommon Lisp' proclamaba sus mejores aproximaciones a algunos de los acercamientos de Common Lisp. Algunos de los dialectos Lisp que se han declarado tanto oficial como no oficialmente 'UnCommon Lisps' en una época o en otra son Lisp370, Scheme, EuLisp y muLisp. Esto era un claro intento de distanciarse de las bases del Common Lisp.

Mas que cualquier otro fenómeno, este comportamiento demuestra una de las claves de la diversificación de Lisp: una extrema, casi juvenil, rivalidad entre los grupos de dialectos.

Ya hemos visto Lisp370 y Scheme. EuLisp es la respuesta europea al Common Lisp, desarrollada como la 'lingua franca' para Lisp en Europa. Sus características principales son que es un Lisp-1 (no tiene separados el espacio de nombres de las variables y el de las funciones), tiene un sistema orientado a objetos de tipo función genérica del estilo del CLOS integrado desde sus inicios, tiene un sistema de construcción de módulos, y está definido en capas para promocionar el uso de Lisp en pequeño e inmerso hardware y máquinas educativas. Sin embargo, EuLisp es parecido al Common Lisp. La definición de EuLisp llevó mucho tiempo; comenzó en 1986, y hasta 1990 la primera implementación no estaba disponible (sólo el interprete) junto con una completa especificación del lenguaje. Sin embargo, la definición por capas, el sistema de módulos y la orientación a objetos del principio demostraron que nuevas lecciones pueden ser aprendidas en el mundo Lisp.

5.2.11 Otros Dialectos LISP: 1980-1984

El resto del mundo no se quedó parado mientras que Common Lisp era desarrollado, aunque Common Lisp era el foco de mucha atención.

5.2.11.1 Dialectos Lisp en el 'stock' hardware

Portable Standard Lisp se extendió a los VAX, DECsystem-20, una variedad de máquinas MC68000 y al Cray-1. Su entorno Emode (mas tarde Nmode) se probó pidiendo ayuda a HP, que lo 'produjo' debido a la creciente presencia del Common Lisp.
Franz Lisp fue portado a muchos sistemas y se convirtió en el mulo de carga del 'stock hardware' de Lisp por los años liderando la capacidad general de Common Lisp en 1985-1986.

InterLisp/VAX hizo su aparición, pero será recordado como un fallo al que contribuyeron 3 causas:

- ?? Proveía compatibilidad con InterLisp-10, la rama d la familia InterLisp maldita por la muerte eventual de los PDP-10
- ?? Proveía sólo de un 'stop-and-copy' recolector de basura, que tenia una malísima actuación en los VAX
- ?? Como el resto del mundo de Lisp, incluyendo el mundo InterLisp, congregado alrededor de máquinas personales Lisp, el VAX nunca fue tomado en serio para propósito Lisp mas que por un pequeño número de negocios.

En Francia, Chailloux y sus colegas desarrollaron un nuevo dialecto Lisp llamado Le_Lisp. El dialecto tenía reminiscencias de MacLisp y se enfocaba en la portabilidad y la eficiencia. Necesitaba ser portable porque la situación de los computadores para Lisp en Europa no estaba tan clara como en los Estados Unidos. Las máquinas Lisp eran dominantes para Lisp en Estados Unidos, pero en Europa estas máquinas no se encontraban tan disponibles y a menudo tenían un precio prohibitivo. Los laboratorios de investigación adquirían, o les daban, unas máquinas muy extrañas (desde la perspectiva de los Estados Unidos). Por eso, la portabilidad era un deber.

La experiencia con VLisp enseñó a Chailloux que la actuación y la portabilidad podían ir juntas, y que extendiendo algunas de las técnicas de VLisp, su grupo era capaz de alcanzar sus metas. En 1984 su dialecto se ejecutaba en 10 máquinas diferentes, y demostraba una actuación mucho mejor que las de Franz Lisp, la alternativa más comparable. En un VAX-11/780, Le_Lisp actuaba tan bien como el ZetaLisp en un Symbolics 3600.

Además, Le_Lisp proveía de un entorno de programación muy completo llamado Ceyx. Ceyx tenia un conjunto completo de ayudas para 'debug', un editor de estructura multi-ventana a pantalla completa, y una extensión de programación orientada a objetos también llamada Ceyx.

La comunidad de Scheme creció de un puñado de aficionados a un grupo mucho más grande, caracterizado por un interés en los aspectos matemáticos de los lenguajes de programación. El pequeño tamaño de Scheme, sus raíces del lambda cálculo y su

apuntalamiento en una semántica generalmente compacta comenzó a hacerlo popular como un vehículo para la investigación y la enseñanza.

En el MIT, bajo la guía de Sussman y Abelson, Scheme fue adoptado para enseñanza de computación a estudiantes no graduados. El libro "Structure and Interpretation of Computer Programs" se convirtió en un clásico y catapultó a Scheme a la notoriedad en una gran comunidad.

Varias compañías surgieron de esa implementación comercial de Scheme. Cadence Research Systems fue fundada por Dybvig, y su Chez Scheme se ejecutó en varias estaciones de trabajo. En Semantic MicroSystems, Clinger, Hartheimer y Ulrich produjeron MacScheme, para el Macintosh de Apple. PC Scheme, de la Texas Instruments, se ejecutó en los PC d IBM y clónicos como el T1, construido y vendido.

El original " Revised Report on Scheme" fue tomado como un modelo para futuras definiciones de Scheme, y un grupo auto-seleccionado de autores de Scheme (ellos mismos se llamaron así) se metieron en el papel de evolucionar Scheme. La regla que adoptaron fue que las características podían ser añadidas sólo por consentimiento unánime.

Emergieron una serie de 'Revised Reports', llamados 'The Revised,... , Revised Report on Scheme'. A finales de 1991, 'The Revised, Revised, Revised, Revised, Report on Scheme' fue escrito y aprobado. Se le llamó 'R^4RS'.

Muchos de los que más tarde se convirtieron en miembros del grupo de Common Lisp proclamaron su gran amor a Scheme y su deseo de ver algo como el Scheme convertirse en el siguiente Lisp estándar. Sin embargo, partes de las comunidades de Scheme y Common Lisp se convirtieron a veces en amargos rivales en la última parte de la década.

5.2.11.2 ZetaLisp

ZetaLisp era el nombre de la versión de Symbolics del Lisp de la máquina Lisp. Debido a que el 3600 (la máquina Lisp de segunda generación de Symbolics, descrita un poco mas abajo) fue programado casi enteramente en Lisp, ZetaLisp vino a requerir un conjunto significante de capacidades no visto en ningún Lisp antes de este punto. ZetaLisp no fue sólo usado para programación ordinaria, sino que el sistema operativo, el editor, el compilador, el servidor de red, el recolector de basura y el sistema de ventanas fueron todos programados en ZetaLisp, y como el primer Lisp de máquinas Lisp no estaba preparado para realizar estas tareas, ZetaLisp fue expandido para manejarlas.

La principal adición al Lisp de la máquina Lisp fue Flavors, un lenguaje no jerárquico orientado a objetos, un sistema de paso de mensajes y herencia múltiple desarrollado a partir de unas ideas de Howard Cannon. Convertir las ideas en un sistema coherente fue en gran parte debido a Moon, aunque Cannon continuó ejerciendo un papel clave.

Las características de Flavors fueron dirigidas por las necesidades del sistema de ventanas de la máquina Lisp, que, por un largo tiempo, fue considerado como el único sistema cuya programación requería herencia múltiple.

Otras adiciones que merecen la pena comentar fueron FORMAT y SETF, arrays complejos y lambda listas complejas para argumentos opcionales y claves. FORMAT es un mecanismo para producir salida de cadenas de caracteres convenientemente, tomando una cadena predeterminada con comillas y sustituyendo valores computados o cadenas por esas comillas (aunque se transformó en algo mucho mas complejo que esto porque las comillas incluían primitivas de iteración para producir listas de resultados, plurales y otras cosas exóticas).

Uno de los factores en la aceptación e importancia de ZetaLisp fue la aceptación de las máquinas Lisp, lo que es discutido en la siguiente sección. Debido a que las máquinas Lisp (sobre todo las de Symbolics) eran los vehículos más populares para trabajo real en Lisp en un entorno comercial, crecería para ser una creencia explícita, fomentado por el mismo Symbolics, ZetaLisp fue el dialecto principal para Lisp. Sin embargo, los chicos de Symbolics fueron tomados muy en serio como una fuerte fuerza política y un aliado político necesario para el éxito de un estándar Lisp más ancho.

5.2.11.3 Las primeras compañías de máquinas Lisp

Existían cinco compañías de máquinas Lisp principales: Symbolics; Lisp Machine, Inc. (LMI); Three Rivers Computer, mas tarde llamado PERQ después de su principal producto; Xerox; y Texas Instruments (TI).

De estas, Symbolics, LMI y TI usaban básicamente el mismo software registrado del MIT como la base de sus ofertas. El software incluía la implementación del Lisp, el sistema operativo, el editor, el sistema de ventanas, el software de red y todas las utilidades. Había un acuerdo en el que el software estaría disponible de manera gratuita (o muy barato) mientras que las mejoras fueran entregadas al MIT. Sin embargo, las compañías competían básicamente en las bases de actuación del hardware, pero secundariamente en la disponibilidad de los avances en el software base común antes de que el software pasare a ser parte del código común. Algunas de las compañías también produjeron extensiones propias, tales como implementaciones C y FORTRAN de Symbolics.

Xerox produjeron las D-máquinas, que ejecutaban InterLisp-D.

Three Rivers vendió una máquina, la PERQ, que ejecutaba tanto un sistema operativo y un lenguaje basados en Pascal como Spice Lisp(y más tarde Common Lisp basado en Spice Lisp).

Así, todas las compañías de máquinas Lisp comenzaron con software existente y todas las compañías de máquinas Lisp derivadas del MacLisp registraron su software de una universidad.

De estas compañías, Symbolics fue la que tuvo mas éxito (medido por el número de instalaciones al final de la era pre-Common Lisp), seguida por Xerox y TI, aunque TI

posiblemente podría haber clamado que ellos eran los que más máquinas habían instalado en base a las compras de 1 ó 2 grandes compañías.

Symbolics es la más interesante de estas compañías debido a la extrema influencia de Symbolics en la dirección del Common Lisp; sin embargo, no queremos decir que las otras compañías no tenían una gran importancia. La popularidad de ZetaLisp (o al menos su aparente influencia en la otra gente del grupo Common Lisp) provino en gran parte de la popularidad del 3600.

El 3600 era una máquina Lisp de segunda generación. Symbolics y LMI comenzaron sus negocios produciendo esencialmente el CADR. Sin embargo, el plan de negocios de Symbolics era producir una mucho más rápida máquina Lisp e introducir las estaciones de trabajo 'sweepstakes'.

A veces es fácil olvidar que, a principio de los 80, las estaciones de trabajo eran una rareza, y las estaciones de trabajo tenían generalmente tan poca fuerza computacional que la gente no se las tomaba en serio. El PDP-10 aún ofrecía una actuación mejor que las estaciones de trabajo y simplemente no se veía posible que los ingenieros, en algún momento, se calentasen con ellas o que formaran un nuevo mercado. Además, no estaba claro que una estación de trabajo basada en Unix y C fuera necesariamente la ganadora, ya que se pensaba que las aplicaciones conducirían al mercado mas allá del desarrollo de software. Sin embargo, no fue estúpido por parte de Symbolics hacer el plan de negocios que hicieron.

Symbolics y LMI fueron fundadas por rivales en el laboratorio de Inteligencia Artificial del MIT, con Greenblatt fundando el LMI y casi todos los demás fundaron Symbolics, en especial Moon, Weinreb, Cannon y Knight.

Uno de los factores en la adopción de máquinas Lisp Symbolics y ZetaLisp fue el hecho de que los primeros 3600 no tenían recolector de basura. Originalmente, el 3600 debía tener un recolector incremental 'stop-and-copy' del estilo de Baker, pero debido a que el espacio de direcciones era tan grande, programas normales no agotaban la memoria en varios días, y los programas intensivos podían ejecutarse durante 8 horas antes de que se agotase.

El recolector de basura incremental fue estrenado varios años después de los primeros 3600. Probó tener una actuación relativamente mala, posiblemente debido a un problema de paginación. Por otro lado, Moon desarrolló un efímero recolector de basura, parecido al usado en SmallTalk. La idea es que un objeto se convertirá en basura poco después de su creación, por lo que si puedes mirar la edad de los objetos y concentrarte sólo en los mas jóvenes, conseguirás casi toda la basura y como es muy poco trabajo, la actuación de la paginación es buena.

El efímero recolector de basura resultó efectivo, y varios años después de que el primer 3600 fuera vendido, un recolector de basura efectivo estaba operativo. Llevó un cierto tiempo que los usuarios se acostumbraran a las diferencias de actuación, pero por ellos el 3600 estaba ya establecido y la posición de los 3600 firmemente implantada.

PERQ entró en el mercado con una máquina micro codificada con una pobre actuación que había sido usada como un computador para preparar documentos.

Fahlman estaba implicado con la compañía, y cuando Common Lisp hizo su debut, el código del Spice Lisp era un Common Lisp sumiso, por lo que el de PERQ fue el primer Common Lisp disponible en una máquina Lisp.

El PERQ original y sus últimas versiones nunca tuvieron mucho impacto comercial fuera del área de Pittsburgh.

La temprana historia de las D-máquinas de Xerox es discutida mas arriba. Antes de la era Common Lisp, su uso se extendió mucho en círculos de InterLisp-10.

Texas Instruments comenzó a entrar en el negocio de las máquinas Lisp justo antes de que se publicara el CLTL1, a principios de 1984. Comenzaron con el proyecto Viking (sin relación con su implementación actual de la arquitectura del microprocesador SPARC), que ejecutaba Spice Lisp en un microprocesador Motorola MC68020. Mas tarde, decidieron ponerse en el camino de las máquinas de Lisp puras e introdujeron el Explorer, una máquina micro codificada que también ejecutaba el software del MIT. Mas tarde, TI se unió con la LMI par explotar alguna tecnología, que parecía tener un efecto mínimo en las fortunas de los negocios de cada una de las compañías excepto la inyección de capital en LMI, prolongando su existencia. El Explorer tenia una buena relación calidad / precio. El hecho de que TI fuera mantenida por el departamento de defensa resultó en buenas ventas para TI durante la temprana era Common Lisp.

5.2.11.4 MacLisp en declive

Aunque MacLisp aún se usaba mucho y se extendía un poco, su desarrollo se paró a inicios de los 80. En ese momento, MacLisp se ejecutaba en ITS, Multics, Tenex, TOPS-10, TOPS-20 y WAITS (todos excepto Multics eran diversos sistemas operativos para el PDP-10).

Los fondos para el desarrollo de MacLisp habían sido dados por el grupo Macsyma, porque los principales clientes del MacLisp, desde el punto de vista del MIT, eran las personas que usaban Macsyma (el consorcio Macsyma). Desde el punto de vista del resto del mundo, sin embargo, mientras que Macsyma era una interesante aplicación de Lisp, MacLisp era más interesante como herramienta para la investigación y desarrollo de la Inteligencia Artificial. Sin embargo, ninguno de estos grupos fue ayudado con fondos y, en ningún evento, nadie encontró razón alguna para hacer otra cosa que aceptar el uso de un MacLisp de distribución gratuita, como el MIT distribuyó.

Los fondos para MacLisp fueron sustituidos por fondos para NIL en VAX por el departamento de energía. El departamento de energía imaginó cosas tales como investigación y desarrollo de armas nucleares además de sus proyectos más benignos como la energía de paisano. Sin embargo, el DOE era una fuente alternativa de fondos de defensa, y otorgó fondos a proyectos como el Lisp S-1.

En general, cada sitio que usaba MacLisp tenia un mago local que era capaz de manejar casi todos los problemas encontrados, posiblemente consultando a White. En al menos un caso, los fondos estaban disponibles para el MIT para hacer algún trabajo de costumbre. Por ejemplo, la versión de MacLisp de segmento único en WAITS fue

pagada por el laboratorio de Inteligencia Artificial de Stanford, y el trabajo fue hecho por Cannon.

MacLisp era el anfitrión para una gran variedad de desarrollos de lenguajes y características a lo largo de los años, incluyendo MicroPlanner, Conniver, Scheme, Flavors, Frames, Extends, Qlisp y varias características de procesamiento de imágenes. La ultima pieza de investigación en MacLisp fue el entorno de programación multi-programas hecho por Frost y Gabriel en Stanford. Este entorno definía un protocolo que permitía MacLisp y E, el editor de Stanford que tenía un soporte del sistema operativo, comunicarse con un mecanismo de sistema operativo parecido a los buzones de correos. Sin embargo, en los primeros días del grupo Common Lisp, los fondos para NIL en VAX por la DOE y el consorcio de Macsyma pararon, quizás por la creencia de que las máquinas Lisp proveerían la base común para Macsyma.

Por esa época los fondos del DOE pararon, Symbolics comenzó un grupo Macsyma para vender Macsyma basado en máquinas Lisp. Este grupo permaneció dando beneficios hasta su disolución a finales de los 80, principios de los 90.

Con la parada de los fondos para NIL, White se unió a Xerox para trabajar en InterLisp. Una situación mas extraña no se puede imaginar. Primero, White tenia una personalidad tan del Este, que todo el aura del estilo de vida californiano parecía demasiado extraño para él. Segundo, la intensa rivalidad entre MacLisp e InterLisp a lo largo de los años podría hacer pensar que nunca trabajarían juntos.

5.2.12 Desarrollo de Estándares: 1984-1992

El periodo de justo después de la aparición de 'Common Lisp: The Language' marcó el comienzo de una era de popularidad del Lisp sin precedentes. En gran medida esta popularidad estaba unida con la popularidad de la Inteligencia Artificial, pero no completamente. Miremos los ingredientes:

- ?? Por primera vez había un estándar comúnmente acordado de Lisp.
- ?? La Inteligencia Artificial estaba en alza, y el Lisp era el lenguaje de la Inteligencia Artificial.
- ?? Parecía haber un recién nacido mercado de estaciones de trabajo, y la actuación de las estaciones de trabajo en Lisp no estaba demasiado lejos de las máquinas de Lisp.
- ?? La arriesgada comunidad central estaba buscando el éxito de compañías como Sun, muy reverenciada desde la perspectiva de la Inteligencia Artificial, y tenía un montón de dinero como resultado del boom de la economía en la primera mitad de la presidencia de Reagan.

Artículos sobre Lisp estaban siendo escritos por revistas muy populares, peticiones de Common Lisp estaban llegando a sitios como el CMU (Fahlman) y Stanford (Gabriel), y por otro lado gente que solo tenia que ver con lo académico era llamada para participar en conferencias industriales para hablar sobre la Inteligencia Artificial y Lisp, y eran considerados como sabios de tendencias futuras.

El ímpetu clave tras el interés por la industria en el Lisp y la Inteligencia Artificial era que los problemas de hardware parecían bajo control y la bestia rabiosa que era el software estaba a punto de ser domada. Métodos más tradicionales parecían inadecuados y siempre estaba el sentimiento de que la nueva cosa, la cosa radical tendría mas efecto que la cosa vieja y tradicional.

En 1984, el año en que CLTL1 fue publicado, varias compañías fueron fundadas para comercializar Common Lisp. Algunas de estas compañías eran Franz Inc. Y Lucid, Inc. Otras compañías al borde de Lisp se unieron a la estela del Lisp con Common Lisp o con Lisps que estaban en el camino hacia Common Lisp. Algunas de estas ultimas incluían a Three Rivers (PERQ) y TI. Algunos 'mainstream' manufacturadores de computadoras se unieron al negocio de Lisp, como DEC, HP, Sun, Apollo, Prime e IBM. También algunas compañías europeas se unieron, como Siemens y HoneyWell Bull. Y los viejos jugadores comenzaron a trabajar con Common Lisp, como las compañías de máquinas Lisp y Xerox. Un nuevo jugador de Japón, Kyoto Common Lisp (KCL) metió un poco de caña: KCL tiene un compilador que compila a C, que es compilado por el compilador de C. Este Lisp se registraba de manera esencialmente gratuita, y las compañías de Common Lisp de repente tuvieron un sorprendente competidor. (Sorprendente porque apareció justo cuando se publicó CLTL1[la implementación estaba basada en la versión Mary Poppins]. KCL también era notable porque estaba implementada por extranjeros, Taiichi Yuasa y Masami Hagiya, exclusivamente en las bases de la especificación. Este esfuerzo expuso un numero bastante grande de agujeros y errores en la especificación que habían pasado desapercibidos por aquellos que habían participado en el histórico desarrollo de Common Lisp).

Aunque uno podría pensar que un producto de buena calidad y gratuito fácilmente ganaría a un producto de mas calidad pero más caro, esto resulto ser falso, y las compañías de Common Lisp florecieron a pesar de sus competidores sin coste. Resultó que una mejor actuación, mas calidad, el compromiso por parte de los desarrolladores de avanzar con el estándar y un mejor servicio eran más importantes que una etiqueta sin precio.

5.2.12.1 Compañías de Common Lisp

Franz Inc. estaba ya en el negocio vendiendo Franz Lisp, el dialecto Lisp parecido al MacLisp usado para transportar una versión de Macsyma llamada Vaxima. Franz Lisp era el dialecto de Lisp más popular en el VAX hasta que el Common Lisp apareció. Franz decidió meterse en el mercado de Common Lisp, dando fondos para este esfuerzo con las ventas de su Franz Lisp. Los principales fundadores de Franz son Kunze, Foderaro y Fateman. Kunze era un estudiante de doctorado de Fateman en la universidad de California de Berkeley en el departamento de matemáticas; Foderaro, habiendo ya obtenido su doctorado con Fateman, se convirtió en el principal arquitecto e implementador de los diversos Lisps ofertados por Franz Inc. Fateman, uno de los primeros implementadores de Macsyma en el MIT, llevó la antorcha del MacLisp / Lisp a Berkeley; él era responsable del portado de Macsyma a VAX. Franz adoptó una estrategia de venta directa, en la que la compañía seleccionaba clientes y les vendía directamente a ellos.

Gold Hill era una división de una compañía padre llamada Apiary Inc, que fue fundada por Hewitt y su estudiante Barber. Barber se había pasado el año anterior fundando Apiary / Gold Hill en INRIA, en Francia, dnde escribió un Lisp parecido al MacLisp / ZetaLisp para los PCs de IBM. Este trabajo fue en parte financiado por INRIA. Cuando volvió, estaba tan cerca al Common Lisp que Hewitt y Barber pensaron que podrían vender su Lisp como un Lisp a punto de convertirse en Common Lisp. Debido a que el PC era considerado una máquina muy importante para la Inteligencia Artificial, parecía ser un plan muy bueno, en el que muchos inversores de la costa Este invertirían. El Lisp de Gold Hill no era un Common Lisp, y en sus primeros años la compañía recibió muchas críticas por falsa publicidad; peor aun, cuando el Lisp se transformó en Common Lisp, su calidad bajó. Al mismo tiempo, el invierno de la Inteligencia Artificial golpeó, y Gold Hill no fue capaz de sobrevivir al nivel que lo había hecho una vez. Fue abandonada por sus inversores capitalistas, echaron a todos los empleados y continua siendo hoy en día una compañía de 2 hombres. Gold Hill también vendía directamente al cliente.

Invierno de la Inteligencia Artificial es el termino usado por primera vez en 1988 para describir el desafortunado destino comercial de la Inteligencia Artificial. Desde finales de los 70 hasta mediados de los 80, la Inteligencia Artificial era una parte importante del negocio de los computadores (muchas compañías comenzaron con el entonces abundante capital disponible para comienzos de alta tecnología) Alrededor de 1988 se hizo evidente para los analistas de negocios que la Inteligencia Artificial no experimentaría un meteórico crecimiento y trataron muy mal tanto a la Inteligencia Artificial como a Lisp en el aspecto comercial. Las compañías de Inteligencia Artificial comenzaron a tener dificultades económicas sustanciales, y también las de Lisp.

Lucid Inc fue fundada por Gabriel (Stanford), Brooks (MIT), Benson (Utah / PSL), Fahlman (CMU) y otros pocos. Apoyados por un capital, Lucid adoptó una estrategia diferente al resto de compañías Common Lisp. En vez de empezar con el código fuente del Spice Lisp, Lucid escribió una implementación del Common Lisp partiendo de cero; mas aún, adoptó una estrategia OEM (hacer un arreglo con una compañía de computadores para comercializar Lisp bajo su mismo nombre. Sin embargo, el Lisp es implementado y mantenido por otra compañía, en este caso, Lucid, que se lleva los royalties.) Lucid rápidamente hizo tratos OEM con Sun, Apollo y Prime. Esto fue posible porque Lucid jugaba con la fuerza de los nombres de sus fundadores y por el hecho de que estaba escribiendo un Common Lisp desde cero, por lo que sería el primer Common Lisp verdadero.

Eventualmente Lucid portó su Lisp y estableció arreglos OEM con IBM, DEC y HP. Aunque los royalties eran relativamente pequeños por copia, la estrategia OEM estableció a Lucid como la principal compañía 'stock-hardware' Lisp. Debido a que las compañías de hardware estaban entusiasmadas con respecto a las oportunidades de negocios para el Lisp y la Inteligencia Artificial, invirtieron un montón en ese negocio. Conseguir a Sun como un OEM fue la clave para la supervivencia de Lucid, ya que las estaciones de trabajo de Sun eran el sello de calidad que era necesario para atraer clientes al Lisp. Sun siempre fue considerada como una líder, por lo que la gente interesada en tecnología líder en Inteligencia Artificial acudía primero a Sun. Sun también contrató un numero de ingenieros que hicieron su propio desarrollo de Lisp, sobre todo en el área de entornos de programación. Antes de que el invierno de la

Inteligencia Artificial golpeara, Lucid comenzó a diversificarse en otros lenguajes (C y C++) y entornos de programación.

5.2.12.2 Grandes compañías con sus propios Lisps

DEC y HP implementaron sus propios Lisps. DEC comenzó con el código fuente de Spice Lisp y HP con PSL. Cada compañía creía que la Inteligencia Artificial despegaría y que tener un Lisp era un ingrediente esencial para el éxito en el negocio de la Inteligencia Artificial. Ambos, DEC y HP, hicieron arreglos con los implementadores originales de esos Lisps, contratando estudiantes que habían trabajado en ellos y consultándoles mucho. Ambos, DEC y HP, tuvieron otros muchos negocios aparte de estos grupos de Lisp- en la cumbre del Lisp, en el ultimo cuarto de los 80, los principales jugadores en el negocio del Lisp con beneficios eran Symbolics, DEC, HP, Sun y Lucid.

DEC y HP se esforzaron mucho en sus ofertas de Lisps, principalmente en el área de entornos, pero también en la actuación del sistema Lisp en sí mismo.

En el ultimo cuarto de los 80, HP se dio cuanta de que PSL no era el ganador y de que necesitaban proporcionar un Common Lisp. Eligieron a Lucid para proporcionarlo y reducieron su plantilla de ingenieros, eligiendo enfocarse mas en el marketing. Como el invierno de la Inteligencia Artificial estaba justo sobre ellos cuando tomaron la decisión, no esta claro si su percepción de esta situación les forzó a recortar su inversión en Lisp.

A principio de los 90, en medio del invierno de la Inteligencia Artificial, DEC también decidió abandonar sus esfuerzos y también eligió Lucid.

IBM tenía varias plataformas adaptables para Lisp; el PC, el 'mainframe' y el RT. De estos, IBM decidió poner un Common Lisp sólo en el RT. IBM subvencionó un programa piloto para poner Spice Lisp (Common Lisp) en el RT, que se iba a convertir en la primera entrada real de IBM en el mercado de las estaciones de trabajo. Debido a la relación de Fahlman con Lucid(uno de los fundadores), un contrato fue eventualmente escrito para que Lucid portara su Lisp al RT de IBM.

Mas tarde, IBM re-entró en el mercado de las estaciones de trabajo con su RS6000, que tiene una buena actuación, y Lucid hizo el Common Lisp para él. IBM también firmó un contrato con Lucid para que proveyera el mismo Lisp en los 370 y los PS-2 ejecutando AIX, una versión de Unix.

Xerox produjo para InterLisp un paquete de compatibilidad con Common Lisp. Este paquete nunca fue un gran éxito para Xerox, que a finales de los 80 se salió del negocio de Lisp, licenciando su software de Lisp a un 'spinoff' iniciado por Xerox llamado Envos en 1989.

Envos sacó una implementación real de Common Lisp y vendió el entorno InterLisp-D. Pero Envos salió del negocio 3 años después de que fuera fundado. Lo que quedó de Envos se convirtió en la compañía Venue, que sencillamente continuaba con el marketing del mismo software que Envos, pero sin fondos directos. Los fondos eran proporcionados por trabajar para el servicio de mantenimiento del Lisp de Xerox.

5.2.12.3 DARPA y las Mailing Lists SAIL

Justo después de que CLTL1 fuera publicado en 1984, ARPA (renombrado DARPA) tomó un interés real en Common Lisp. Esponsorizaron una reunión de comunidad y promovieron un mayor desarrollo de Common Lisp aplicado un sistema de desarrollo completo, (incluyendo una extensión de orientación a objetos), multitareas, sistemas de ventanas, gráficos, interfaces de funciones extrañas 'foreign' e iteración. Parecía que DARPA desease un resurgimiento de Lisp y quería proporcionar fondos para ayudar a esto.

Después de la reunión, nuevas mailing lists fueron creadas en SAIL para discutir esos temas y, aunque casi todas esas listas eran bastante tranquilas, algunas fueron testigo de discusiones muy interesantes.

5.2.12.4 El comienzo del comité técnico ANSI X3J13

En una continuación de la reunión un año después (diciembre 1985 en Boston), una reunión técnica aparentemente benigna fue interrumpida por un anuncio sorprendente: Common Lisp debía ser estandarizado, anunció DARPA, y Robert Mathis, del grupo de trabajo ISO del lenguaje de programación ADA iba a liderar este esfuerzo.

La razón para esta repentina necesidad fue que la comunidad europea de Lisp estaba planeando lanzar su propia estandarización para frenar la expansión de Common Lisp. Mathis, con su gran experiencia internacional, parecía a DARPA la elección natural para liderar esa responsabilidad, y el grupo de Common Lisp poco podía hacer excepto aceptarlo.

El periodo comprendido entre la primavera de 1986 y la de 1992 fue una combinación de riñas políticas y de un desarrollo interesante de Lisp. Como era normal, el ímpetu tras el desarrollo de Lisp era incrementar el expresivo poder de Lisp. Las riñas políticas se centraron en dos cosas: en Common Lisp, cada individuo luchaba por dar su propia marca en el lenguaje; fuera del Common Lisp, varios grupos intentaron minimizar el tamaño de Lisp para garantizar su supervivencia, tanto académica como comercial.

Unos pocos meses después de la reunión de Diciembre de 1985 hubo una reunión en la CBEMA (la Computer and Business Equipment Manufacturers Association) en Washington D.C. (CBEMA servía como secretaria de X3, un comité estándar acreditado para sistemas de procesamiento de información que operaba siguiendo los procedimientos de ANSI, el American National Standards Institute. Entre los comités técnicos mas conocidos englobados por X3 estaban X3H3 (gráficos en computadores, incluyendo PHIGS), X3J3 (lenguaje de programación FORTRAN), X3J4 (lenguaje de programación COBOL) y X3J11 (lenguaje de programación C)). Las metas de la estandarización fueron discutidas, y el temas mas importante, propuesto por DARPA, era si mezclarse con las actividades de Scheme. Las metas del nuevo grupo, que pronto se convertiría en el comité técnico X3J13 para el lenguaje de programación Common Lisp, también fueron discutidas.

El punto sobre el espacio de nombres es importante para entender el debate entre los ‘proponents’ que proponían el dialecto Lisp. Un espacio de nombre es un mapeado

entre un identificador (cadena de caracteres) y su significado. En Common Lisp hay un cierto numero de espacio de nombres (variables, funciones, tipos, etiquetas, bloques y capturadores de etiquetas, entre otros). Si un Lisp tiene espacio de nombres separados para variables y funciones, los usuarios pueden dar a variables nombres que son también nombre de función, porque las reglas de evaluación especifica el espacio de nombres en el que mirar el significado. En un Lisp con un único espacio de nombres, el usuario debe tener cuidado cuando cree variables para evitar 'dar sombra, ocultar, pisar' el nombre de una función. Esto es importante para las macros, que en efecto debe decidir lo que se supone que significa una variable libre. Aunque hay muchos espacios de nombres en Lisp, porque el problema recae en la práctica en el espacio de nombres de variables y funciones, por lo que esto reduce la cuestión a si los nombres de variables y funciones pertenecen a uno o a dos espacios de nombres separados. A menudo se refiere a esta disputa como el debate Lisp-1 versus Lisp-2. Un Lisp-1 es un Lisp en el que las funciones y variables estan en el mismo espacio de nombres, y un Lisp-2 es cuando estan en distintos espacios de nombres.

El esfuerzo para mezclar las comunidades de Scheme y Common Lisp fue lanzado por 2 'frentes-fronts'. Uno quería conseguir una solución al problema con las macros que el Lisp-1 causaba. Este problema es que con solo un espacio de nombres es relativamente mas facil caer en conflicto de nombres que llevaría a un código incorrecto. Los principales lideres de Common Lisp creían que si este problema pudiera ser solucionado, Common Lisp podría sobrevivir a una transición del Lisp-2 al Lisp-1. El otro 'frente-front' intentaba convencer a la comunidad de Scheme que esto era una buena idea.

En el primer 'frente-front', Gabriel y Pitman hicieron un informe detallando los detalles técnicos de las macros. Varias soluciones técnicas surgieron al mismo tiempo, la mas prometedora descrita en la disertación de Kohlbecker.

En el segundo, Gabriel y Clinger se acercaron a la comunidad Scheme, que rechazó cualquier asociación con la comunidad Common Lisp. (Lamentablemente, aunque ha habido otros intentos de juntar las dos comunidades, nunca ha habido un dialogo serio entre los dos grupos).

A finales de 1986, estaba claro que la comunidad europea de Lisp iba a producir un nuevo dialecto Lisp, que sería informalmente llamado EuLisp, y que la misma comunidad iba a comenzar un 'ISO effort – esfuerzo ISO' para estandarizar este dialecto.

EuLisp es un dialecto de Lisp definido por capas, con un lenguaje de núcleo muy pequeño, e incrementándose su tamaño conforme vamos subiendo de capa, siendo la meta tener un Common Lisp estructurado en capas. EuLisp tiene un recurso para la orientación a objetos, modulos, multitarea y un sistema de condición. Es un Lisp-1. Un sistema de condición es un recurso para definir y manejar excepciones de usuario y manejar excepciones de sistema. En Common Lisp, el recurso provee un mecanismo para ejecutar codigo definido por el usuario en el contexto dinámico del error.

El desarrollo técnico mas importante durante este periodo fue el Common Lisp Object system (CLOS). En 1986, 4 grupos comenzaron a competir por definir la parte

de programación orientada a objetos de Common Lisp: New Flavors (Symbolics), CommonLoops (Xerox), Object Lisp (LMI) y Common Objects (HP). Tras una batalla de 6 meses, se formó un grupo para escribir el estándar para CLOS basado en CommonLoops y en New Flavors. Este grupo fue formado por Moon (Symbolics), Bobrow (Xerox), Kiczales (Xerox), Keene (Symbolics), DeMichiel (Lucid) y Gabriel (Lucid). Otros tambien contribuyeron informalmente al grupo: Dussud, Kempf y White. Realizar la especificación de CLOS llevó 2 años, y dicha especificación fue adoptada en junio de 1988 sin cambios.

CLOS tiene las siguientes características:

- Herencia múltiple usando un algoritmo de linealización para resolver conflictos y ordenas métodos. La herencia múltiple provee de un mecanismo para construir nuevas clases combinando 'mizins', que son clases que proveen alguna estructura y comportamiento. La programación con herencia múltiple habilita al diseñador para combinar el comportamiento deseado sin tener que seleccionar la clase existente mas cercana ni modificarla o comenzar una nueva cadena de herencia simple.
- Funciones genéricas cuyos métodos son seleccionados basándose en las clases de todos los argumentos requeridos. Esto está en contraste con el modelo de paso de mensajes en el que un mensaje es enviado a un objeto simple cuya clase selecciona el método a invocar.
- Combinación de metodos, que provee el mecanismo para coger comportamientos de partes de los componentes y mezclarlos todos juntos. La combinación de metodos es un aspecto muy importante de la herencia múltiple, porque cada clase combinada puede proveer parte del comportamiento que se necesite, y el programador no tiene necesidad de codificar un método que combine los métodos existentes.
- Metaclases, cuyas instancias son clases y que son usadas para controlar la representación de instancias de clases.
- Metaobjetos, que controlan el comportamiento del mismo CLOS. CLOS puede ser visto como un programa escrito en CLOS. Ya que cualquier programa CLOS puede ser personalizado, también pude serlo el CLOS.
- Una creación de objetos elaborada y un protocolo de inicialización que puede ser usado para proveer al usuario de la posibilidad de pesonalizar los procesos de creación de instancias, cambio de clases, reinicialización y redefinición de clases.

Durante las deliberaciones de X3J13, se le añadieron otras cosas al Common Lisp: un recurso para la iteración, un sistema de condición, una mejor especificación de las semánticas de compilación y evaluación y varios cientos de pequeñas tonterías.

El proceso X3J13 no se parecia al de Scheme. El proceso de Scheme permitía a cualquier persona vetar una adición, mientras que el proceso X3J13 se basaba en el voto

de la mayoría. Esto hizo que muchos comites estuviera impacientes por dejar su marca en Common Lisp.

El recurso de iteración, llamado LOOP, consiste en una única macro que tiene una sintaxis parecida al inglés o al COBOL. El debate sobre este recurso era a veces intenso, especialmente cuando Fahlman aún estaba activo en Common Lisp. Debido a que su sintaxis no se parecía a la del Lisp, era facil de ridiculizar.

El sistema de condición llevó mas lejos el desarrollo de las posibilidades del manejo de excepciones de Common Lisp introduciendo condiciones de primera clase y mecanismos para definir como las condiciones de ciertas clases tenían que ser manejadas (bien automáticamente, bien por la intervención humana). La adopción de este recurso fue mas fácil debido a la adopción de CLOS, que pavimentó el camino para llegar a una formulación mas limpia de los mecanismos básicos. Sin embargo, el sistema de condición no estaba completamente 'CLOSificado' y arreglado. Por ejemplo, las clausulas que sintácticamente parecen ser definiciones de métodos y por lo tanto deberían ser seleccionadas basándose en la especificidad de las clases, son actualmente tratadas como clausulas COND.

En 1987 ISO creó un grupo de trabajo llamado WG16 para comenzar el proceso de estandarización de Lisp a nivel internacional. Los dos principales competidores eran EuLisp y Common Lisp.

La meta política de EuLisp era echar a Common Lisp de Europa. Debido a que los estándares de los Estados Unidos tenían una gran influencia en Europa y porque la única organización de estándares con verdadera fuerza en Europa era la ISO, esto fue lo que se intentó.

La meta intelectual era un dialecto Lisp limpio, con calidad comercial y estructurado en capas para el futuro. EuLisp parece haber conseguido sus metas y muchos lo consideran una de las definiciones de Lisp mas agradables, aunque no hay todavía implementaciones comerciales de ella.

Durante cinco años, los Estados Unidos se las arreglaron para mantener los progresos apartados del comité ISO hasta que, en 1992, se alcanzó un compromiso en el que esencialmente un subconjunto de Common Lisp y de CLOS crearían las bases de un dialecto Lisp de nucleo.

En 1988, DARPA convocó otra reunión de Lisp para discutir el juntar las comunidades Scheme y Lisp pero, como en intentos anteriores, fallaron principalmente porque la comunidad Scheme no quería tener nada que ver con Common Lisp. Scherlis, Squines, Gabriel, Bobrow, Sussman y Fahlman encabezaron esta reunión.

En 1989, scheme comenzó un proceso IEEE de estandarización, que culminó en 1991 con estándares IEEE y ANSI, éste último después de un periodo de revisión pública no anunciada.

También en 1989, los primeros recolectores de basura no intrusos aparecieron en las compañías Lucid y Franz. El recolector de Lucid es un recolector de basura efímero

basado en una combinación de ideas de la generación 'basurero' de SmallTalk y el recolector de basura efímero de Symbolics.

La aparición de estos recolectores pareció tener el efecto de incrementar la legitimidad de las compañías de máquinas Lisp 'stock-hardware' al mismo o mas alto nivel que las compañías de máquinas Lisp. Esto fue porque las compañias de máquinas Lisp promovieron la creencia de que los 'stock-hardware' lisps nunca podrían tener la actuación (particularmente en la recoleccion de basura) que los computadores Lisp de propósito específico podrían tener. Cuando se comprobó que esto era falso, las compañías de máquinas Lisp sufrieron.

En 1991, el R^4 Report incluyó una especificación de macros higiénicos, el primer recurso para macros parcialmente estandarizado en Scheme.

Conforme C3J13 progresaba y el poder de las estaciones de trabajo de propósito general se incrementaba (en gran parte debido al desarrollo de rápidos procesadores RISC), las compañías Lisp 'stock-hardware' dominaron, forzando a las compañías de máquinas Lisp bien a abandonar el negocio, bien a dejar su posición de liderazgo. También, el deterioro del mercado Lisp y el declive general de la economía en los Estados Unidos se combinaron para hacer posible la supervivencia de las pequeñas compañías dedicadas al software (los clientes no deseaban comprar caras computadoras dedicadas y después gastarse mucho dinero en su mantenimiento).

En Abril de 1992, X3J13 entregó a X3 SPARC (parte autorizada de X3J13, nada que ver con la arquitectura del microprocesador del mismo nombre) su versión de Common Lisp. Al mismo tiempo, ISO WG16 produjo la primera versión de su Lisp de núcleo.

La versión ANSI de Common Lisp es increiblemente larga (sobre 1000 páginas). Alguien nos dijo que la primera vez que un oficial de X· SPARC lo vió dijo que era mucho más grande que el COBOL.

Aunque esto pueda parecer divertido, muestra como un proceso de creciente deseo de expresividad intensificó la atención para conseguir detalles correctos (incluso detalles que apenas importaban), la necesidad de los individuos de dejar su propia marca, y una aparentemente deliberada ceguera hacia las realidades comerciales puede derivar en un resultado no pretendido (un gran e inmanejable lenguaje que pocos pueden entender completamente).

5.3 EVOLUCIÓN DE ALGUNAS CARACTERÍSTICAS ESPECÍFICAS DEL LENGUAJE

En esta sección discutimos la evolución de algunas características que son o únicas de Lisp, o únicamente manejadas por Lisp.

Pero debido a la falta de espacio, no podemos tratar todos los aspectos que nos gustarían, tales como editores de estructura, procesamiento paralelo, programación orientada a objetos, recoleccion de basura, estructuras de registros, cierres (tanto léxicos como dinámicos)... Cada uno de estos temas tiene una historia que abarca décadas e

interacciones con el desarrollo de la teoría del lenguaje y otros lenguajes de programación. Aquí debemos contentarnos con unos pocos aspectos que son representativos de lo que concierne a la comunidad de Lisp.

5.3.1 El tratamiento de NIL (y T)

Casi desde el principio, Lisp ha usado el símbolo NIL como el símbolo distinguido que indica el final de una lista (y que es en sí mismo la lista vacia); este mismo objeto también se usa como el valor 'false' devuelto por los predicados. McCarthy ha comentado que estas decisiones fueron hechas 'muy a la ligera' y 'mas tarde se ha probado que fueron desafortunadas'. Además, las primeras implementaciones establecieron una tradición de usar la dirección 0 como la representación de NIL; McCarthy también comentó que "además de promover la programación pornográfica, dar una interpretación especial a la dirección 0 ha causado dificultades en todas las implementaciones posteriores"

La ventaja de usar la dirección 0 como la representación de NIL es que la mayoría de las máquinas tienen una instrucción 'salta si cero' o una equivalente, permitiendo una rápida y compacta prueba para ver si es fin de lista. Como un ejemplo de las dificultades de implementación, sin embargo, considerad la arquitectura PDP-10, que tenía 16 registros, o acumuladores, que eran también direccionables como posiciones de memoria. La posición de memoria 0 era por consiguiente el registro 0. Debido a que la dirección 0 era NIL, la representación estándar de los símbolos dictaba que la mitad dercha del registro 0 contenía la propia lista para el símbolo NIL (y la mitad izquierda contenía la dirección de otra información como la cadena de caracteres para el nombre "NIL"). La tradición en la implementación así resultó de atar un registro a un valor en una arquitectura donde los registros eran un bien escaso.

Mas tarde, cuando MacLisp adoptó de InterLisp la convención de que (CAR NIL) = (CDR NIL) = NIL, el registro 0 seguía reservado; sus 2 mitades contenían el valor 0, por lo que las operaciones car y cdr no necesitaban un caso especial de NIL.

Hoy en dia algunas implementaciones de Common lisp usan un sistema complejo de 'offset' representación de datos para evitar casos especiales tanto para CONs como para símbolos; cada símbolo es representado de tal modo que los datos para el símbolo no comienza en la palabra de memoria direccionada por un símbolo puntero, sino 2 palabras despues de la palabra direccionada. Una celda CONS consiste en la palabra direccionada (el CDR) y la palabra de despues (el CAR). De este modo el mismo puntero sirve tanto para NIL (símbolo) como para ( ), la lista vacía cuyo CAR y CDR son ambos NIL.

Hay un peligro en usar un test rápido para comprobar el final de una lista; una lista podría cambiar para ser algo impropio, esto es, acabar en un objeto que no es ni la lista vacía ni una celda CONS. Interlisp separa la diferencia, dando al programador la opción de elegir entre velocidad o seguridad.

> *Aunque casi todas las listas terminan en* NIL, *la lista ocasional que acabe en un átomo, por ejemplo (A B . C), o peor aún, un número o cadena, podría causar efectos muy*

extraños. De acuerdo con esto, hemos tomado las siguientes decisiones de implementación:

Todas las funciones que iteran a lo largo de una lista, por ejemplo, member, length,... *terminaran por un chequeo* nlistp, *en vez de con el convencional* nullcheck, *como precaución de seguridad por si se encuentran tipos de datos que podría causar bucles CDR infinitos.*

Para los usuarios con una aplicación que requiera extrema eficiencia (chequeo de NIL *requiere una sola instrucción, mientras que* nlistp *requiere unas 8), les hemos provisto versiones rápidas de* member, last, nth... *que se compilan abiertamente con chequeo de* NILs...

Black comentó en 1964 sobre la diferencia entre NIL y ( ) que era una cuestión de estilo de programación. A lo largo de los años se ha discutido si dejar aparte los 3 papeles de la lista vacia, l valor 'false' y el simbolo cuyo nombre es NIL; esa discusión fue particularmente intensa en la comunidad Scheme, donde muchos de los que la constituyen estan interesados en la elegancia y la claridad. Ellos consideran la construcción:

```
(if (car x) (+ (car x) 1))
```

como un mal juego, prefiriendo la más explicita

```
(if (not (null (car x))) (+ (car x) 1))
```

El "Revised Revised Report on Scheme" definió 3 distintos valores para *nil*; ( ), la lista vacía; y $!false, el valor booleano para falso (junto con $!true, el valor booleano para verdadero). Sin embargo, en un interesante compromiso, todos los sitios del lenguaje que probaban valores true/false consideraban ambos ( ) y $!*false* como falso, y los demas objetos como verdaderos. El Report comenta:

La lista vacía cuenta como falso por razones históricas exclusivamente, y los programas no deberían confiar en esto porque futuras versiones de Scheme probablemente eliminarán esta tontería.

Los programadores acostumbrados a otros dialectos de Lisp deberían tener en cuenta que Scheme ya ha eliminado la tontería de identificar la lista vacía con el símbolo nil.

El Revised[3] Report en el *Algorithmic Language Scheme* acortó *$!false* y *$!true* a *$f* y *$t*, y lo remarcaron mas refinadamente (en los dos sentidos):

La lista vacia cuenta como falso por compatibilidad con programas e implementaciones existentes que asumen que éste es el caso.

Los programadores acostumbrados a otros dialectos de Lisp deberían tener en cuenta que Scheme distingue false *y lalista vacía del símbolo* nil.

El recientemente aprobado estándar IEEE para Scheme especifica que $f y $t son los valores estándar para falso y verdadero (respectivamente), y que todos los valores excepto *$f* cuentan como veerdaderos, incluyendo $t, lalista vacia, símbolos, numeros, cadenas, vectores y procedimientos. Por esto, la comunidad Scheme ha vencido la larga tradición y separado completamente las 3 nociones del valor *false*, la lista vacía y el símbolo NIL. Sin embargo la cuestión continua siendo debatida.

La cuestión del NIL tambien fue debatida en el diseño del Common lisp, y al menos una de las implementaciones que contribuyeron directamente, NIL (el dialecto),ya había tomado la decisión de que la lista vacia ( ) no sería lo mismo que el símbolo NIL (hay una broma circulando que dice que NIL -- New Implementation of Lisp-- liberó a NIL--símbolo-- de su papel conmo la lista vacía y así podría servir libremente como el nombre del lenguaje).

Merece la pena comentar que los implementadores de Lisp no han estado tentados de identificar NIL con el número 0, con una notable excepción, un sistema de Lisp para el PDP-11 escrito en los 70 por Stallman, en el que el número 0, junto con el símbolo NIL era usado como la lista vacía y como *false*. Compara esto con el uso del 0 y el 1 como falso y verdadero en APL, o el uso del 0 como falso y como el puntero nul en C. Estos 2 lenguajes jhan provocado la misma clase de comentarios sobre los juegos de palabras y la programación tan poco práctica que McCarthy hizo sobre Lisp.

Esto puede parecer al lector ser una gran discusión y un desperdicio de papel para un punto tan pequeño del diseño de un lenguaje. Sin embargo, el espacio tomado aquí refleja exactamente la proporción de tiempo y energía desperdiciados en un debate sobre este tema por la comunidad de Lisp a lo largo de los años. Es un debate sobre la expresividad vs. Limpieza y sobre las diferentes nociones de claridad. Incluso si Lisp no impone un tipado mas fuerte, algunos programadores prefieren mantener una disciplina de tipo en sus códigos, escribiendo

(if (not (null (car x))) ...) en vez de (if (car x) ...).

Otros sostienen que tales excesos son perjudiciales para la claridad en vez de mejorarla.

5.3.2 Iteración

Lisp, acordando a su P.R., ha usado tradicionalmente funciones condicionales y recursivas como el principal medio de expresar estructuras de control.

En algunos casos esto fue llevado por el deseo de emular estilos de programación encontrados en ALGOL y por el hecho de que algunos compiladores optimizaran algunas veces las llamadas recursivas de cola, el programador no podía confiar en esto hasta la era del buen Scheme y los compiladores de Common Lisp de los 80.

Quizás la iteración más simple construída es ejemplificada por el 1er. “do loop” introducido en MacLisp.

(do var init stet text . body)

significa lo mismo que

```
(let ((var init))
        (block (when test (return))
                (progn . body)
                (setq var step)))
```

Además el Fortran "do loop"

```
      DO 10 J=1, 100
      IF (A(J) .GT. 0) SUM=SUM+A(J)
   10 CONTINUE
```

Podía ser expresado como

```
(DO J 1 (+ J 1)(> J 100)
        (WHEN (PLUSP (A J))
                (SETQ TOTAL (+ TOTAL (AREF A J)))))
```

excepto, naturalmente, los array Lisp son usualmente 0-origin en vez de 1-origin, así que

```
(DO J 0 (+ J 1)(= J 100)
        (WHEN (PLUSP (A J))
                (SETQ TOTAL (+ TOTAL (AREF A J)))))
```

Este "viejo estilo" MacLisp "do loop" ***was by no means the earliest iteration syntax*** presentada a Lisp; la mencionamos solo porque es la más simple. La declaración iterativa de InterLisp CLISP fue el ejemplo más temprano del estílo más típico que habubiera sido inventado alguna vez.

```
(FOR J← 0 TO 99 SUM (A J) WHEN (PLUSP (A J)))
```

Esto devuelve la suma como su valor más que acumularlo dentro de la variable TOTAL por efecto lateral.

Las Macros u otras facilidades de transformación de código tales como CLISP hacen esta clase de extensiones particularmente fácil. Mientras que el esfuerzo en InterLisp estaba centralizado, en otros dialectos Lisp han sido repetidos instancias de un asistente local y otra sintaxis para procesos iterativos en Lisp. Usualmente es caracterizado por la clase de Keywords pseudo-inglés encontradas en Algol, aunque una versión en Standford dio más confianza al conjunto de caracteres ASCII extendido disposible en sus teclados ***home-grown***. Esto llevó a una proliferación de sintaxis cercanamente afines, tipicamente por la keyword FOR o LOOP que han atraido a muchos programadores pero que desagrada a otros.

Por estas reacciones estéticas fuertes y diferentes a la sintaxis de iteración, la pregunta de si incluir una macro loop llegó a ser una batalla en el temprano diseño de Common Lisp, con la máquina Lisp actuó generalmente a favor de su adopción y Scott

Fahman en contra fuertemente y de forma más débil Steele. El resultado fue un compromiso. La 1ª definición de Common Lisp incluyo una macro loop con funcionalidad absolutamente mínima: ello permitió keywords no especiales y era bueno solamente para expresar repetición infinita de una secuencia de subformas. Ello era comprendido ser un ***placeholder***, reservando el nombre loop para una posible extensión a alguna sintaxis de iteración abierta. ANSI comité X3J13 estuvo de acuerdo y adopto una ligera versión de loop basado en el usado en MIT y en la máquina Lisp.

En el proceso X3J13 también se consideró otras 2 aproximaciones a iteración que había

.....

```
(collect-sum (choose-if #'plusp
                        (#M(lambda (j) (a j))
                        (scan-range :start 0 :below 100)))
```

La llamada a scan-range genera una serie de enteros del 0(inclusive) al 100(exclusive). La notación #M significa "map" la siguiente función es aplicada a vada elemento de una series, produciendo una nueva serie. La función choose-if es un filtro, y collect-sum devuelve la suma de todos los elementos de una serie. Además las series son usadas en un estilo funcional, reminiscencia de APL:

```
+/(T>0)/T✍A[ 100]
```

La definición de estas series primitivas y sus composiciones permitidas es inteligentemente ***constrained*** para que estas puedan siempre ser compiladas a un código iterativo eficiente sin necesidad de grandes estructuras de datos intermedias en tiempo de ejecución. Los generadores y ***gatherers*** son un metodo de encapsulamiento de series para que parezcas como ***streams*** de entrada o salida, respectivamente, que sirve como fuente de ***sinks*** de valores sucesivos por efecto lateral. Además

```
(generator (scan-range :start 0 :below 100))
```

produce un objeto que envía sucesivos enteros del 0 al 99 cuando dados repetidamente a la función de extración next-in.

```
(let ((num (generator (scan-range :start 0 :below 100))))
     (gathering (result collect-sum))
          (loop (let ((j (next-in num (return result))))
                    (if (plusp j) (next-out result j))))))
```

Esto recuerda a una de las posibilidades de las listas de Conniver or de los generadores de Alphard, aunque no hay conexión directa. Los generadores y ***gatherers*** enfatizan el uso de estructuras de control mas que las relaciones funcionales.

Después de muchos debates, X3J13 aplaudió el desarrollo de series y generadores pero los rechazó como propuestas de estandarización, prefiriendo dejar pasar un poco de tiempo para ver como funcionaban.

5.3.3 Macros

Las Macros parecen haber sido introducidas en LISP por Timoth P. Hart.

En Lisp 1.5, las formas especiales son usadas para tres propuestos lógicos:

1. Buscar la ***alist***
2. Permitir a las funciones tener un número indefinido de argumentos
3. Conservar los argumentos que son evaluados

Los intérpretes del nuevo LISP pueden facilmente satisfacer las necesidades (1) haciendo la ***alist*** un tipo especial o una entidad de tipo APVAL. (2) y (3) pueden ser reemplazadas por incorporar una MACRO ***expander in define.***

La lista de propiedades de una definición macro tendrá el indicador MACRO seguido por una función de un argumento, una forma que empieza con el nombre de la macro y cuyo valor reemplazará la forma original en todas las definiciones de la función.

La función **macro** definirá macros como **define** funciones.

Define será modificado para hacer expansiones macro.

Ejemplos:

1. El existente **FEXPR csetq** puede ser reemplazado por la definición macro:

```
MACRO ((
(CSETQ (LAMBDA (FORM) (LIST (QUOTE CSET) (LIST (QUOTE QUOTE)
(CADR FORM)) (CADDR FORM))))
))
```

2. Una nueva **macro stash** generará la forma encontrada frecuentemente en PROG's:

   x:= cons[form;x]

   Usando la macro stash, uno podría escribir:

   (STASH FORM X)

   Stash puede ser definido por:

```
MACRO ((
(STASH (LAMBDA (FORM) (LIST (QUOTE SETQ) (CADAR FORM)
(LIST (CONS (CADR FORM) (CADAR FORM))) )))
))
```

3. Nuevas macros pueden ser definidas. **Enter** es una macro para añadir una nueva entrada a la tabla almacenada como el valor de una variable de programa.

```
                MACRO
enter[form]  ?  list[STASH;list[CONS;cadr[form];
caddr[form];cadddr[form]]
```

La macro definiendo una función macro es facilmente definida:

```
macro[1]  ?  deflist [1;MACRO]
```

El nuevo define es un poco más duro:

```
define [1]  ?  deflist[mdef[1];EXPR]
mdef[1] ? [
    atom[1] ✍ 1;
    eq[car[1];QUOTE] ✍ 1;
    member[car[1];(LAMBDA LABEL PROG)] ✍
            cons [car[1];cons[cadr[1];mdef[caddr[1]]]];
    get[car[1];MACRO] ✍ mdef[get[car[1];MACRO][1]];
    T ✍ maplist[1;?[[j];mdef[car[j]]]]]
```

4. La macro para **select** ilustra el uso de macros como un medio de permitir funciones con un número arbitrario de argumentos.

```
    select[form]  ?  ?[[g];
            list[list[LAMBDA;list[g];cons[COND;
            maplist[cddr[form];?[[1];
                    [null[cdr[1]] ✍ list[T;car[1]];
                    T ✍ list[list[EQ;g;caar[1]];cadar[1]]]]]
    ]];cadr[form]]][gensym[]]
```

La macro STASH es la equivalente de la macro PUSH encontrada en Interlisp y más tarde en Common Lisp. Hay dos fallos menores en la definición de mdef: 1º PROG falla al procesar todos las declaraciones en una forma PROG. 2º COND puede errar si una variable tiene el mismo nombre que una macro y la variable es uada como la parte test de una cláusula COND. Finalmente, Hart obtuvo un increible aumento en poder expresivo con un cambio simple al lenguaje, animando al usuario a explotar el poder de Lisp para servir como su propio metalenguaje.

Las macros de Hart fueron usadas en el sistema Lisp para el Q-32. Inspección e la función MDEF en el compilador revela que el error al procesar las declaraciones PROG había sido reparadas: mdef[caddr[1]] fue reemplazado por mdef[cddr[1]]. Desafortunadamente, el uso de mdef[cddr[1]] tiene sus propios problemas: una variable cuyo valor es devuelto por una expresión landa o una etiqueta en la cabeza de un PROG podía ser incorrectamente reconocida como el nombre de una macro, con eso de tratar el cuerpo de la forma LAMBDA o PROG como una llamada macro.

Macros de esta clase fueron una parte integral del diseño del Lisp 2.

Una clase similar de macro computada apareció en el MIT PDP-6 Lisp, pero las llamadas macros fueron expandidas al encontrarse con el compilador del intérprete. En el caso del intérprete, si una función nombrada explicitamente cambiara a una llamada de función para tener una propiedad MACRO o su lista de propiedades entonces la definición de la función era dada a la llamada macro original como un argumento y era se esperaba que devolviera otra forma para ser evaluada en lugar de la llamada. El ejemplo dado en la PDP-6 Lisp fue:

```
(DEFPROP CONCONS
        (LAMBDA (A)
                (COND ((NULL (CDDR A)) (CADR A))
                        ((LIST (QUOTE CONS)
                        (CADR A)
                        (CONS (CAR A)
                        (CDDR A)))))
         MACRO)
```

Esto definió una macro equivalente en efecto a la función Common Lisp **list***.

Una ventaja de este esquema era que uno podía definir algunas funciones que usan macros y más tarde la macro. Un inconveniente fue que el intérprete debía constantemente reexpandir la misma llamada macro cada vez que esta era repetida, reduciendo la velocidad de ejecución. Un dispositivo llamado "macros displacing" pronto llegó a ser común entre los usuarios de MacLisp; esto implicó la función DISPLACE:

```
(DEFUN DISPLACE (OLD NEW)
        (RPLACA OLD (CAR NEW))
        (RPLACD OLD (CDR NEW))
        OLD)
```

Uno entonces escribiría

```
(DEFPROP CONSCONS
        (LAMBDA (A)
                (DISPLACE A
                        (COND ....)))
        MACRO)
```

El efecto fue alterar la estructura de la lista original de la llamada macro asi como reemplazar la con su expansión.

Esto tuvo sus propios inconvenientes. Primero, fallaba si una macro necesitaba expandir a un átomo; los escritores de macros aprendieron a producir (PROGN FOO) en vez de FOO. Segundo, si una macro era redefinida después de una llamada a ella hubiera sido hecha, las ejecuciones subsecuentes de la llamada no usaría la nueva definición. Tercero, el código era modificado; *pretty-printing* código que originalmente

contenía una llamada macro mostraría la expansión, no la llamada a la macro original. Esta última desventaja era tolerable solamente porque el entorno MacLisp estaba firmemente basado en ficheros***:*** desplazando macros modificaba sólo la copia *in-core* de un programa; no afectaba a la definición master, la cual se considerada que estaba en el texto de algún archivo. Desplazar macros de esta clase habría sido intolerable en Interlisp.

Alrededor de 1978, máquina Lisp introdujo una mejora técnica:

```
(defun displace (old new)
        (rplacd new (list (cons (car old) (cdr old)) new))
        (rplaca old 'si:displaced)
        new)

(defmacro si:displaced (old new) new)
```

La idea es que la llamada macro es desplazada por una lista (si:displaced macro-call expansion). La macro **si:displaced** devuelve su segundo argumento, así todo se comporta como si la expansión misma hubiera reemplazado la llamada macro. Durante la expansión de la llamada a **si:displaced** no está libre, es presumiblemente más barato que reexpandir continuamente la llamada macro origina.

BBN Lisp tenía tres clases de macro: open, computed y substitution. Una definición macro fue almacenada en la lista de propiedades de su nombre bajo la propiedad MACRO; la forma del valor de la propiedad determinó cual de los tres tipos de macro era. Originariamente los tres tipos eran efectivos solamente en código compilado. Eventualmente, después de que BBN Lisp llegara a ser Interlisp, un DWIM llamado MACROTRAN fue añadido eso hizo las tres tipos de macros efectivas en código interpretado.

Una macro open BBN-Lisp causó que la macro name fuera reemplazada por una expresión lambda, causando que la función fuera compilada "open". Aquí tenemos una definición de macro open para ABS:

```
(LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X))))
```

Naturalmente esto tiene exactamente la misma forma que una definición de función.

Una macro computada en BBN-Lisp era similar a la clase en MIT PDP-6 Lisp, excepto que la función expandedora recibía el CDR de la llamada a la macro mas que la llamada macro entera. Aquí tenemos una macro computada para LIST:

```
(X (LIST (QUOTE CONS)
        (CAR X)
        (AND (CDR X)
                (CONS (QUOTE LIST)
                        (CDR X]
```

La 1ª X es el nombre de una variable para ser ligada al CR de la llamada macro. Destacar también el uso de un corchete de cierre en la definición.

Una macro substitution en BBN-Lisp consistía en un patrón simple y un patrón de substitución; las subformas de la llamada macro eran substituidas por ocurrencias en el patrón de los correspondientes parametros de nombres. Una macro substitución para ABS sería así:

((X) (COND ((GREATERP X 0) X) (T (MINUS X))))

Sin embargo, la llamada (ABS (FOO Z)) sería expandida a

(COND ((GREATERP (FOO Z) 0) (FOO Z)) (T (MINUS (FOO Z))))

llevando a las evaluaciones múltiples de (FOO Z), la cual sería desafortunado si (FOO Z) fuera una computación cara o tuviera efectos laterales. En contraste, con una macro open la llamada (ABS (FOO Z)) sería expandida a

((LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X)))) (FOO Z))

la cual evaluaría (FOO Z) exactamente una vez.

Muddle tenía unos recursos de macros muy parecidos a los del *PDP-6 Lisp*, con una ligera diferencia. La función macro expansión, más que ser llamada con la forma macro como su argumento, era aplicada al CDR de la forma. Esto permitió a la lista de argumentos complejos keywords (***complex argument-list keywords***) actuar, permitiendo ciertos clases simples de patrones ajustando como para las macros substitution de Interlisp.

<DEFMAC INC (ATM “OPTIONAL” (N 1))
<FORM SET .ATM <FORM + <FORM LVAL .ATM> .N>>>

Esto no fue necesario construir el resultado para una macro computed. El resultado de expandir <INC X> sería <SET X <+ .X 1>>.

Como MacLisp creció de PDP-6 Lisp, la comunidad de MacLisp se diversificó, produciendo una variedad de metodologías para definir macros.

Las distintas localizaciones de MacLisp desarrollaron docenas de definiciones macros similares pero no lo bastante compatibles. No era raro para cada paquetes estar en uso en la misma localización, con los adherentes de cada orden usando todo lo que su wizard había desarrollado. Tales paquetes usualmente incluían herramientas para la destrucción de formas de argumentos y la construcción de formas resultantes.

Las macros dieron un paso mayor con el Lisp de las máquinas Lisp, la cual consolidó las técnicas varias de definiciones de macro dentro de dos rasgos estandarizados que fueron adoptados por la comunidad de MacLisp y eventualmente por la de Common Lisp.

Aquí tenemos la definición de la macro FOR usando DEFMACRO:

```
(defmacro for (var lower upper . body)
        (cons ‘do
                (cons var
                        (cons lower
                                (cons (list ‘+ var 1)
                                        (cons (list ‘> var upper)
                                                body))))))
```

La sintaxis del backquote era particularmente poderosa cuando estaba anidado. Esto ocurría primordialmente dentro de las macro-definiciones; porque tales eran codificadas por wizards. Alan Bawden de MIT adquirió una reputación particular como ***backquote-meister*** en los principios de la máquina Lisp.

Backquote y DEFMACRO tenían una gran diferencia. Esta destacaba en poder expresivo, hizo posible una forma standard, empezó un nuevo extensión del lenguaje porque era mucho más fácil definir construcciones del nuevo lenguaje en un standard, formas portables para que dialectos experimentales pudieran ser compartidos. Algunos, incluyendo David Moon, habían opinado que el éxito de Lisp como un kit de diseñador de lenguaje era debido a la habilidad del usuario de definir macros que usaban Lisp como lenguaje de procesamiento y la estructura de lista como la representación de programa, haciendo fácil extender la sintaxis del lenguaje y las semánticas. En 1980 Ken Pitman escribió un buen sumario de las ventajas de macros sobre FEXPR’s definiendo sintaxis de nuevo lenguaje.

Sin embargo, no era fácil expresar cada macro en su nuevo formato. Considerando la macro INC. En Noviembre 1978, la DEFMACRO de la máquna Lisp no era lo bastante rica para llevar argumentos “optional”:

```
(DEFUN INC MACRO (VAR . REST)
        ‘(SETQ ,VAR (+ ,VAR ,(IF REST (CAR REST) 1))))
```

La parte opcional debía ser llevada con una condicional programada explicitamente, esta deficiencia fue pronto notada y rapidamente remediada permitiendo a DEFMACRO aceptar la misma sintaxis de lambda listas complejas como DEFUN:

```
(DEFUN INC MACRO (VAR &OPTIONAL (N 1))
        ‘(SETQ ,VAR (+ ,VAR ,N)))
```

Esto ocurrió en Enero 1979, en el cual MacLisp absorbió DEFMACRO y DEFUN con &-keywords de Lisp-Machine Lisp.

Un problema adicional fue esa sintaxis repetitiva, de la clase que debía ser expresada en BNF extendida con una estrella de Kleene, no fue capturado por este armazón y tenía que ser programado explicitamente. Veamos esta simple definición de LET:

```
(defmacro let (bindings . body)
        ‘((lambda , (mapcar ‘car bindings) , body)
        , (mapcar ‘cadr bindings)))
```

Nota el uso de MAPCAR para procesamiento iterativo de los ligamientos. Esta dificultad no fue abordada por el Lisp de las máquinas Lisp o Common Lisp, en esa comunidad DEFMACRO con &-keywords está el estado del arte hoy. Common Lisp lo hizo, sin embargo, generalizar DEFMACRO permitir anidamiento recursivo de tales listas lambdas.

Una forma práctica para permitir tales problemas es una macro de escritura para intentar elegir nombres que al usuario no le gusta encontrarse. Sin embargo recoge nombres extraños tales como %%foo%% , usando múltiples arrays o paquetes para permitir nombres de clases. Sin embargo ninguna de estas clases provee una buena garantía. Steele señalo que el uso cuidadoso de 'thunks' podía eliminar el problema probablemente, pero no en todos los casos.

Los propósitos del esquema recordaron todos estos arreglos como también los defectos para que Scheme los adopte. El resultado fue que Scheme divergió en los años 80. Después cada implementación tuvo alguna clase de macro (pero dos no gustaron). Después todo el mundo estuvo de acuerdo en que los recursos de las macros fueron inestimables en principio y en la practica pero mirando sobre cada instancia particular como una clase de vergüenza secreta de familia. ¡¡Si sólo la cosa secreta pudo ser encontrada!!. Esta cuestión llegó a ser más precisa como la posibilidad de desarrollar un Scheme estándar.

A mitad de los 80 dos nuevas clases de propósitos fueron lanzados: macros higiénicas y cierres sintácticos. Ambas aprovechan envolver el uso de especial de entornos sintácticos para asegurar que las referencias encajan con las definiciones. Una relación de líneas de trabajo permite al programador controlar procesos de expansión, rodeando explícitamente y manipulando funciones de expansión. Todo esto se intentó como una macro de recursos para Scheme, los métodos previos fueron recordados también como incorrectos para su adopción en lenguajes elegantes.

Las macros higiénicas se desarrollaron en 1986 por Eugene Kohlbecker con la ayuda de Daniel Friedman , Matthias Felleisen y Bruce Duba. La idea es etiquetar las ocurrencias de las variables con una etiqueta indicando si apareció en el código fuente original o fue introducida como resultado de la expansión de la macro. Si sucede que múltiples macros se expanden, la etiqueta debe indicar en que paso de la expansión estuvo envuelta. La técnica renombra variables así que una variable referenciada no puede referirse a una introducida en un paso diferente.

La disertación de Kohbecker llevó este paso más lejos, proponiendo ajustar un patrón y un lenguaje de sustitución de patrones para definir macros. El mecanismo usó macros higiénicas de expansión para permitir nombres de clases. El lenguaje de definición de las macros fue suficientemente rico para expresar una gran variedad de macros útiles, pero no suministraron recursos para la ejecución de código Lisp especificados por el usuario arbitrariamente; esta restricción se pensó que era necesaria para subversiones de la garantía de buena higiene.

La primera lista que se da a una sintaxis extendida es una lista de palabras claves que son parte de la sintaxis de la macro y no son etiquetadas como posibles variables referenciadas. La segunda lista menciona variables que pueden ser

introducidas por la expansión de la macro pero son propuestas para interactuar con las formas arguméntales. Por ejemplo, consideremos una implementación (usando la primitivas de llamada concurrente de Scheme) de una pequeña generalización del bucle atribuido a Dahl. Este ejecuta instrucciones repetidamente hasta que falle la cláusula del While o hasta que se use Exit :

```
(extend-syntax (loop while repeat) (exit)
        ((loop e1 e2 ...repeat)
         (call/cc (lambda (exit)
                        ((label foo
                                (lambda () e1 e2...(foo)))))))
        ((loop e1...while p e2...repeat)
         (call/cc (lambda (exit)
                        ((label foo
                                (lambda () e1...
                                                (unless p (exist #f))
                                                        e2...
                                                (foo))))))))
```

En este ejemplo ‘loop’, ’while’ y ‘repeat’ son palabras claves y no deberían ser confundidas con referencias de variables. ‘Exit’ es para salir de la macro, pero se propuso para usarla en las formas arguméntales de la llamada de la macro. El nombre ‘foo’ no se propuso para tal uso, y la expansión de la macro higiénica lo renombrará si necesariamente permite nombres de clases. Nótese el uso del idioma ‘e1 e2...’ requiere que al menos una forma este presente si no hay cláusula ‘while’.

Los cierres sintácticos fueron propuestos en 1988 por Alan Bawden y Jonattan Rees. Su idea tiene un fuerte parecido a la técnica de expansión-pasajera de Dybuing, Friedman y Haynes pero es más general. Los contextos sintácticos son representados no por manejo de etiquetas de la expansión de la macro higiénica sino por objetos del entorno; una forma de cerrar un trozo de código con respecto a un entorno sintáctico, dando el control explícito de la escritura de la macro sobre la correspondencia entre una ocurrencia de un símbolo y otro.

Los cierres sintácticos proveen gran poder y flexibilidad pero ponen la carga en el programador para que los use propiamente.

En 1990, William Clinger unió sus fuerzas con Rees para proponer una gran síntesis que combina los beneficios las macros higiénicas y los cierres sintácticos con la ventaja añadida de correr linealmente mejor que cuadráticamente. Su técnica se llamo ‘La macro que trabaja’. Su perspicacia se puede explicar por analogía de reducción del lambda calculo. A veces la regla de la ?-conversión debe ser aplicada para renombrar variables en una expresión de lambda-calculo así que una subsecuente ?-reducción no producirá un nombre de clase. Algunas no pueden hacer tal renombramiento de una vez; Esto es necesario para entremezclar los renombramientos con las ?-reducciones, porque una ?-reducción puede hacer dos copias de una lambda-expresión y trae la obligación de uno en conflicto con la del otro. Lo mismo pasa con las macros: esto es necesario para entremezclar el renombramiento con la expansión de la macro.

El Estándar de Scheme (IEEE,1991) fue adoptado sin una macro de recursos, así que la confusión aun reina en este punto. Las macros permanecen en una activa investigación.

¿Por qué son las macros tan importantes para los programadores de Lisp?

No solamente por la conveniencia sintáctica que ellas proveen, sino porque ellas son programas que manipulan otros programas, las cuales han sido siempre el tema central de la comunidad de Lisp. Si Fortran es el lenguaje basado en números y C es el lenguaje basado en caracteres y punteros, entonces Lisp es el lenguaje basado en los programas. Su estructura de datos es útil para representar y manipular código de programas. La macro es el ejemplo más inmediato de programa escrito en un metalenguaje. Porque Lisp es su propio metalenguaje, el poder de los lenguajes de programación completos puede ser traído para llevar la tarea de transformar código.
Por comparación, el procesador de C está completamente anémico. El lenguaje de la macro consiste meramente en sustitución y concatenación de tokens. No hay ni recursión ni metarecursión, lo cual dice que una macro de C no puede ni invocarse a si misma ni definir a otra macro. Los usuarios de Lisp encuentran esto ridículo. Hay mucha preocupación con los procesos de programación como un objeto de discurso y un objeto de computación y ellos insisten en tener los mejores significados de expresión para sus propósitos.

5.3.4 Recursos numéricos

Aun tenemos un pequeño desconcierto sobre el origen de los Bignums (un tipo de dato que usa una cantidad variable de almacenaje así como para representar arbitrariamente grandes valores de enteros, sujeto al tamaño total del montón, que donde los Bignums son almacenados). Estos habían aparecido en MacLisp y el Lisp 1.6 de Stanford aproximadamente al mismo tiempo y quizás también en el Standard Lisp. Se necesitaron para programas de símbolos algebraicos tal como REDUCE y MACSYMA. Hoy en día el manejo de Bignums es una característica de distinción del Lisp. Ambos, el Standar Scheme y el Common Lisp los requirieron.

Jon. L. White escribió un articulo sobre los conjuntos de primitivas que permite uno de los códigos más eficientes de aritmética de Bignum en Lisp, en vez de tener que codificar todo en lenguaje ensamblador.

Hay también una literatura en aritmética BigFloat. Esta ha sido usada en sistemas de álgebra simbólica. Lisp es a menudo usado como plataforma para esta clase de investigación porque teniendo Bignums tenemos 2/3 del camino recorrido.

Las funciones del MacLisp HAULONG y HAIPART fueron introducidas para soportar la aritmética BIGFLOAT de Macsyma; se convirtieron en las funciones INTEGER-LENGTH y LDB del Common Lisp.

En los 80 los desarrolladores del Common Lisp se ajustaron a la introducción del estándar de punto flotante de IEEE.

Aunque Lisp no es considerado como un lenguaje de programación numérico, hubo tres grandes influencias en esta dirección: MACSYMA, el proyecto S-1 y Gerald Sussman.

El primer compilador numérico de Lisp fue desarrollado por MACSYMA. El resultado fue un compilador de Lisp que estaba compitiendo con el compilador de Fortran.

La S-1 fue inicialmente desarrollado para ser un procesador rápido de señales. Una de las aplicaciones fue la detección de submarinos, la cual parecía requerir una mezcla de procesamiento de señales numéricas y técnicas de I.A. El proyecto recibió asesoramiento de Kahan en el diseño de su aritmética en punto flotante, así que al final llegó a ser similar al estándar de IEEE. Esto pareció apropiado para refinar las técnicas del compilador del MacLisp para producir buen código numérico en S-1 Lisp.
La S-1 ofreció cuatro diferentes formatos de punto flotante (18,36,72,144 bits). Influenciado por el S-1 Lisp, el Common Lisp provee un sistema expandido de datos de tipo punto flotante para acomodar tal variación arquitectural.

Gerald Sussman se interesó en aplicaciones numéricas y en el uso de Lisp para transformar y generar programas numéricos. A petición de Sussman la función '/' fue definida para devolver racionales cuando fuera necesario por lo que (/ 10 4) en Common Lisp devuelve 5/2 (Esto no se consideró como un cambio radical al lenguaje, ya que los racionales ya se usaban en sistemas de álgebra simbólica. Los desarrolladores de Common Lisp simplemente estaban integrando en el lenguaje funciones que eran frecuentemente requeridas por sus clientes).

5.3.5 Algunos fallos notables

A pesar de la tendencia de Lisp para absorber nuevas características, tanto de otros lenguajes de programación como de la experiencia de la Comunidad Lisp , hay unas pocas ideas que han sido probadas repetidamente pero que por alguna razón no han cuajado en la Comunidad Lisp. De esta ideas destacan: sintaxis parecida al Algol, múltiples valores generalizados y programación lógica con unificación de variables.

5.4 LISP COMO UN LENGUAJE DE LABORATORIO

Un aspecto interesante de la cultura de Lisp, en contraste con la mayoría de los lenguajes de programación que hay alrededor es que sus "dialectos" son tenidos en consideración. Lisp ha sido usado durante su historia como un lenguaje de laboratorio. Es trivial añadir a Lisp un conjunto de funciones que faciliten su uso.

Dadas macros, como CLISP o alguna equivalente es bastante fácil añadir estructuras de control nuevas u otros constructores sintácticos.

Si esto falla, es cosa de hora y media escribir en Lisp un intérprete completo para el nuevo dialecto.

Lisp no tiene una sintaxis especial (como operadores infijos) para usar operaciones constructoras de manera que las funciones definidas por el usuario parecen funciones definidas por el sistema para el que las llama.

Hay una gran tradición en experimentar con argumentaciones de Lisp, desde "vamos a añadir una característica nueva" a inventar un lenguaje nuevo usando Lisp como lenguaje de implementación. Esta actividad fue más intensa en MIT a finales de los 60 y principios de los 70 y en sitios como la Universidad de Stanford, la de Cornegie-Mellon e Indiana.

Uno de los primeros lenguajes basados en Lisp fue METEOR (Bobrow, 1964), una versión de COMMIT con la sintaxis de Lisp. COMMIT fue un lenguaje que relacionaba un conjunto de reglas con los contenidos de un conjunto de tokens simbólicos; fue el precursor de SNOBOL y el antecesor de lenguajes como OPS5.

METEOR fue absorbido por el sistema MIT Lisp 1.5 que se ejecutaba en IBM 7090.

Otro lenguaje construido sobre Lisp fue CONVERT (Guzman, 1966). Mientras que METEOR era una representación de COMIT representado con estructuras Lisp, COVERT nació de las características de COMIT que tenían que ver con las estructuras recursivas de Lisp permitiendo la asignación de diseños recursivos a estructuras arbitrarias de Lisp.

Carl Hewitt diseñó un ambicioso lenguaje para demostradores de teoremas, llamado Planner.

Su primera contribución consistió en avances en el diseño de las llamadas directas y el uso de backtraking para la búsqueda directa.

Nunca fue implementado como se pensó originalmente pero sirvió para otros tres descubrimientos en la historia de Lisp: Micro-Planner, Muddle y Conniver.

Gerald Jay Sussman, Drew McDermott y Eugene Charniak implementaron un subconjunto de Planner, llamado Micro-Planner, que fue absorbido por el sistema Lisp 6 MIT PDP que empezó MacLisp. La semántica del lenguaje implementado no estaba completamente formalizada. Las técnicas de implementación eran, más bien, "ad-hoc" y no funcionaban correctamente para casos complicados.

También se implementó una versión de Planner en POP-2 (Davies,1984).

El lenguaje Muddle (el posterior MDL) fue una versión extendida de Lisp y en algunos aspecto un competidor, diseñado y utilizado por el Dynamic Modeling Group de MIT, que estaba separado del laboratorio de Inteligencia Artificial (AI) del MIT. Esto comenzó en 1970 con Gerald Jay Sussman, Carl Hewitt, Chris Reeve y David Cressey y más tarde siguieron Bruce Daniels, Greg Pfister y Stu Galley.

Hasta cierto punto, la competición entre Muddle y Lisp, resultó beneficiosa. La E/S (entrada/salida), el manejo de interrupciones y la multiprogramación hizo que Muddle estuviese mucho más avanzado que el MacLisp de aquel tiempo. Muddle tenía

un recolector de basura más completo que el que nunca tuvo PDP-10 MacLisp. También tenía una librería de funciones de aplicación de subrutinas, especialmente para gráficos muy grande. Muddle introdujo los términos relacionados con las lambda-listas, OPTIONAL, REST y AUX, que más tarde fueron adoptados por Conniver, la máquina de Lisp y Common Lisp.

El lenguaje Conniver fue diseñado por Drew McDermott y Gerald Jay Sussman en1972, previendo las limitaciones de Micro-Planner y, en particular, del control de sus estructuras.

El diseño de Conniver puso el flujo de control en manos de los programadores de una forma mucho más explícita.

Este diseño estuvo fuertemente influenciado por el modelo de pila de spaghettis introducido por Daniel Borrow y Ben Wegbreit e implementado en BBN-Lisp (más tarde sería conocido como Interlisp). Como la pila de spaghettis, Conniver ofrecía nociones separadas de entornos de datos y entornos de control y la posibilidad de crear cierres sobre cualquier estructura. Conniver difería de la pila de spaghettis en las consideraciones sobre la implementación. El principal punto de Conniver era la generalidad y la facilidad de implementación; estaba escrito en Lisp y representaba los entornos de datos y control como estructuras de Lisp, permitiendo al recolector de basura de Lisp manejar las reclamaciones de entornos abandonados.

Por otra parte, la implementación de las pilas de spaghettis contenía cambios estructurales para el nivel más bajo de Lisp. El direccionamiento de memoria era muy eficiente porque permitían asignar y liberar posiciones de la pila cada vez que se daba el caso.

Hewitt y sus alumnos desarrollaron e implementaron en MacLisp un lenguaje nuevo. Este lenguaje se llamó primero Planner-73 pero después se cambió y pasó a llamarse PLASMA (PLAnner-like System Modeled Actor).

La sintaxis de PLASMA se parecía a la de Lisp, usaba muchas clases de paréntesis y corchetes y otros caracteres especiales. PLASMA usa operadores aritméticos infijos.

El desarrollo de Lisp en MIT tomó dos caminos distintos durante los 70. En el primer camino, MacLisp fue la herramienta de trabajo, en código máquina (ensamblado) para tener una máxima eficiencia y compactación, sirviendo las necesidades del laboratorio de Inteligencia Artificial y las del grupo MACSYMA. El segundo camino consistió en un diálogo/competición/discusión entre Hewitt (y sus estudiantes) y Sussman (y sus estudiantes). Consistió en proponer cada conjunto de ideas en forma de un lenguaje nuevo, implementado normalmente parecido a Lisp (MacLisp o Muddle) para experimentarlo y probarlo.

La siguiente disputa entre Hewitt y Sussman fue, por supuesto Scheme. Pero observamos que esta disputa terminó en diálogo. Empezando por Lisp, trataron de explicar la búsqueda, el control de las estructuras, los modelos de computación y, al final, simplemente volvieron al Lisp pero con una diferencia: era necesaria una extensión léxica (cierres) para hacer el Lisp compatible con el lambda cálculo, no tanto

para la sintaxis como para la semántica, por esto, contactaron con varios investigadores de lógica matemática y consiguieron relacionar a la comunidad del Lisp con investigadores en programación funcional.

Scheme era mucho más simple que Lisp 1.5. Esto permitió experimentar de una manera muy rápida con ideas sobre implementación y lenguajes. Sussman y Steele estuvieron probando y midiendo así como diez intérpretes a la semana. Un punto interesante es la comparación entre llamadas por nombre y llamadas por valor.

Scheme sirvió también como implementación base para otros lenguajes prototipos. Otra técnica popular para explicar el significado de un lenguaje es la de dar una semántica denotacional, que describe el significado de cada construcción en términos de sus partes y las relaciones entre ellas. El lambda cálculo es la base de esta notación.

Mientras que Scheme mostraba que un buen diseño de Lisp daba toda la flexibilidad necesaria para el manejo de las estructuras de control y de paso de mensajes, no resolvía otra propiedad del desarrollo de los lenguajes de Inteligencia Artificial basados en Lisp: el manejo automático de la búsqueda directa o los demostradores de teoremas. Después de Scheme, algunos lenguajes basados en Lisp se desarrollaron en esta dirección por Sussman y sus estudiantes.

El desarrollo de lenguajes para aplicaciones en Inteligencia Artificial continuó por otros caminos y Lisp fue el vehículo para elegir cómo implementarlas. Durante el comienzo de los años 80, empezó a ser otra alternativa, sobre todo en asuntos relacionados con la eficiencia. Las mejoras en las técnicas de implementación de Lisp, particularmente en compilación y recolección de basura surgieron.

Durante el boom de la Inteligencia Artificial al principio de los años 80, los sistemas experto fueron "lo más", fueron utilizados para los sistemas basados en reglas basados en lenguajes parecidos a METEOR o CONVERT. OPS5 fue uno de los lenguajes basados en reglas más conocidos de este periodo; XCON (un sistema experto para instalaciones de configuraciones VAX, desarrollados por la Universidad para la Corporación de equipamiento digital de Carnegie-Mellon) fue su primera aplicación exitosa. OPS5 fue implementado primero en Lisp, después, por eficiencia, fue recodificado en BLISS.

MultiLisp fue el trabajo de Bert Halstead y sus estudiantes en MIT. Se basan en Scheme. MultiLisp propuso una construcción *pcall*, esencialmente es una llamada de función que evalúa los argumentos de forma concurrente (y completa) antes de invocar a la función. Esta pcall propone una disciplina para el uso de "futuros" que es adecuado para muchos propósitos. MultiLisp se ejecutó en el multiprocesador Concert, una colección de procesadores Motorola 68000. MultiScheme, un descendiente de MultiLisp, fue implementado, más tarde por la BBN Butterfly.

Butterfly PSL fue una implementación del Portable Standard Lisp en la BBN Butterfly. Fue utilizado para el desarrollo de características de procesos paralelos.

Qlisp fue desarrollado por Richard Gabriel y John McCarthy en Stanford. Extendió el Common Lisp con algunas estructuras paralelas de control que funcionaban

como estructuras de control como qlet, qlambda y qcatch. El modelo computacional incluía un conjunto de procesos y una explicación de cómo crear procesos y controlar sus interacciones y consumo de recursos. Por ejemplo, qlambda podía producir tres clases de funciones: normales, como las que produce lambda, activas (eager) que crearían un proceso a parte cuando fuesen creadas; y con retraso, que crearían un proceso a parte cuando fueses invocadas (llamadas). Qlisp fue implementado en el FX8 Alliant y fue la primera implementación de Lisp compilada de forma paralela.

La máquina "Connection" de Lisp fue un dialecto extendido de Common Lisp con una nueva estructura de datos, el *xapping*, que intentó soportar paralelismo de datos. Un *xapping* era un híbrido entre array, tabla hash y asociaciones de listas, semánticamente es un conjunto de pares de índices ordenados. Las primitivas del lenguaje eran creadas con el objetivo de poder procesar todos los valores del *xapping* concurrentemente, emparejando valores de distintos *xappings* por sus índices asociados. La idea era que los índices eran etiquetas para procesos virtuales.

Recapitulando: Lisp es un excelente laboratorio para experimentar con lenguajes por dos razones:

1. Se puede elegir un subconjunto muy pequeño, con una docena de primitivas, como mucho, que es fácilmente reconocible por miembros de los lenguajes como Lisp.

2. Es particularmente fácil (cosa de una hora o así) robustecer un dialecto nuevo dentro de una implementación de Lisp que ya exista. Incluso si la implementación del lenguaje "padre" difiere en aspectos fundamentales, puede ofrecer operaciones primitivas como por ejemplo aritméticas o de E/S.

6. BIBLIOGRAFIA

Para la realización de este trabajo han sido necesarias distintas fuentes de información, éstas han sido:

PROGRAMACIÓN FUNCIONAL CON HASKELL
Blas C. Ruiz – José E. Gallardo – Francisco Gutiérrez – Pablo Guerrero
Universidad de Málaga

LOGICA PARA LA COMPUTACION I
Inmaculada Pérez de Guzmán Molina – Gabriel Aguilera Venegas
Ed. Agora. Universidad de Málaga

APUNTES DE LA ASIGNATURA PROGRAMACIÓN DECLARATIVA II
Blas C. Ruiz
Universidad de Málaga

FOUNDATIONS OF LOGIC PROGRAMMING
J.W.Lloyd
Springer-Verlag

LOGIC. A FOUNDATION FOR COMPUTER SCIENCE
V. Sperschneider & G. Antoniou
Addison-Wesley

Además de la recopilación de información obtenida a través de diversas direcciones de Internet, algunas de ellas proporcionadas por Mr. John McCarthy.