Capítulo 1

Introducción

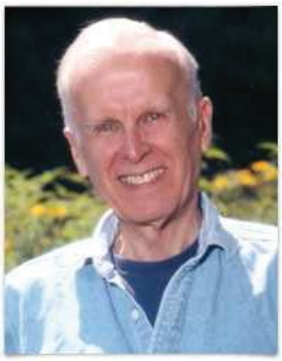

Figura 1.1: John Backus (1924-2007)

El matemático John Backus (ver figura 1.1) (creador del FORTRAN) nos advertía ya en 1977 [2] sobre los peligros de hacer evolucionar los lenguajes de programación mediante una simple acumulación injustificada e incoherente de comandos. Aunque su crítica la dirigió al tristemente célebre PL/1, las ideas de Backus bien pueden aplicarse a la mayoría de los compiladores actuales, e incluso a mucho software dirigido a usuarios finales. Backus decía que el "cuello de botella de von Neumann" de las computadoras de su época había generado un cuello de botella intelectual que hacía que los desarrolladores de lenguajes pensaran con la restricción impuesta por tal arquitectura, que permite transmitir sólo una palabra de información a la vez entre la unidad central de proceso y la unidad de almacenamiento. En su tan

memorable plática con motivo de la recepción del prestigioso premio Turing otorgado por la ACM (*Association for Computing Machinery*), Backus atacó sin piedad el uso de variables para imitar las celdas de almacenamiento de las computadoras de von Neumann, el uso de sentencias de control para emular sus instrucciones de salto y prueba, y las sentencias de asignación para imitar los mecanismos de almacenamiento, transferencia y manipulación aritmética de dichas computadoras. Específicamente, Backus dirigió sus argumentos contra la sentencia de asignación, atribuyéndole el nada envidiable honor de ser la causante principal de la auto-censura a la que se habían visto limitados los diseñadores de lenguajes de su época.

Backus respaldó abiertamente el uso de la programación funcional como una alternativa viable a la programación procedural tradicional, carente de "propiedades matemáticas útiles que permitan razonar acerca de los programas" [2] y de mecanismos poderosos que permitan combinar programas existentes para crear otros. De tal forma, la sentencia de asignación y el uso de variables se vuelven innecesarios, la recursividad (funciones que se llaman a sí mismas) sustituye a las sentencias de control, el cálculo lambda proporciona el tan deseado modelo matemático que permita razonar acerca de los programas y las funciones de orden superior (funciones que toman como argumentos a otras funciones) pasan a ser el mecanismo clave para la reutilización del código previamente escrito. Esto permite un estilo de programación no sólo más elegante, sino también más abstracto que el de los lenguajes tradicionales como Pascal y C.

En la programación funcional pura [28] el "valor de una expresión depende sólo de los valores de sus subexpresiones", sin que hayan efectos secundarios de ningún tipo. En la programación funcional pura no se permite (ni se requiere) realizar asignaciones. En la práctica, la mayor parte de los lenguajes funcionales son impuros, porque permiten realizar asignaciones de variables. Sin embargo, su estilo de programación está dominado por la parte pura del lenguaje. Scheme es un ejemplo típico de un lenguaje funcional simple pero poderoso.

La programación funcional no es la única alternativa a la programación procedural; la programación lógica es otro serio contrincante. Si quisiéramos generalizar las ideas de John Backus, debiéramos referirnos a los llamados **lenguajes declarativos**, en los que un programa establece explícitamente las propiedades que el resultado necesita tener, pero no establece cómo debe obtenerse; es decir, estos lenguajes carecen de estado implícito y se ocupan de conceptos estáticos más que de dinámicos (es decir, se ocupan más del

"qué" que del "cómo"). Esto trae como consecuencia que estos lenguajes no dependan de una noción innata de orden y que no exista un concepto de flujo de control ni sentencias de asignación; además, en ellos se enfatiza la programación mediante expresiones (o términos) que permitan caracterizar el resultado deseado. Usando este concepto, la programación funcional puede considerarse una instancia de la programación declarativa en la que el modelo de computación usado como base son las *funciones* (contrastando, por ejemplo, con las *relaciones*, que son el modelo utilizado por la programación lógica).

El objetivo principal de estas notas es enseñar los conceptos básicos de programación en Scheme, un lenguaje funcional caracterizado por su extrema simplicidad que, sin embargo, no compromete su elegancia ni su poder. Comenzaremos por hacer un recorrido histórico de la evolución de Scheme, para posteriormente hablar brevemente de las versiones más importantes del lenguaje disponibles en la actualidad. Después iremos abordando diversos temas de programación, yendo de los más simples a los más complejos, incluyendo capítulos dedicados a temas especiales tales como conjuntos, archivos, búsqueda con retroceso (*backtracking*), números aleatorios, streams, continuaciones y búsqueda y diseño de juegos. Adicionalmente, se incluye un capítulo en el que se revisan los conceptos básicos del cálculo lambda, y otro que resume el abundante trabajo que se ha realizado en torno a técnicas de recolección de basura.

El material principal fue diseñado para cubrirse en un semestre, y cada capítulo incluye un buen número de ejercicios de programación que pueden asignarse como tareas tras cubrir cada tema. Aunque no tiene que seguirse estrictamente el orden de los temas utilizado en este texto, los primeros 10 capítulos están enlazados, y se recomienda estudiarlos en la secuencia presentada, a menos que se tengan conocimientos preliminares de Scheme, en cuyo caso pueden saltarse los capítulos introductorios y consultarse sólamente los avanzados.

1.1 Un Poco de Historia

El desarrollo de los lenguajes funcionales ha sido influenciado a lo largo de la historia por muchas fuentes, pero ninguna tan grande ni fundamental como el trabajo de Alonzo Church (ver figura 1.2) en el cálculo lambda [4]. De hecho, el cálculo lambda suele considerarse como el primer lenguaje

Figura 1.2: Alonzo Church (1903-1995)

funcional de la historia en algunos textos, aunque no se le consideró como tal en la época en que se desarrolló, ni Church lo vio como un nuevo paradigma de programación, sino más bien como una herramienta teórica que resolvería los profundos enigmas de la computabilidad. Las dificultades para hacer acopio de una computadora en plena Segunda Guerra Mundial tornaban la investigación en lenguajes de programación en un lujo excesivo. Sin embargo, no resulta erróneo asumir que los lenguajes funcionales hoy en día no son más que embellecimientos (no triviales) del cálculo lambda.

Hay quienes creen que el cálculo lambda fue asimismo el fundamento en el que se basó el desarrollo del Lisp, que fue el primer lenguaje funcional de la historia (al menos, el primero que se implementó en una computadora). Sin embargo, el mismo John McCarthy (su creador, ver figura 1.3), ha negado ese hecho [21]. El impacto del cálculo lambda en el desarrollo inicial del Lisp fue mínimo, y no ha sido sino hasta recientemente que el (ahora) popular lenguaje ha comenzado a evolucionar de acuerdo a los ideales de esta teoría matemática. Por otra parte, Lisp ha tenido una enorme influencia en el desarrollo de los lenguajes funcionales que le siguieron.

La motivación original de McCarthy para desarrollar Lisp fue el deseo de contar con un lenguaje para procesamiento de listas algebraicas que pudiera usarse para hacer investigación en inteligencia artificial. Aunque la noción de procesamiento simbólico era una idea muy radical en aquella época, las metas de McCarthy eran sumamente pragmáticas. Uno de los primeros intentos por desarrollar un lenguaje de tal naturaleza fue llevado a cabo por McCarthy en los 1950s, produciéndose el llamado FLPL (*FORTRAN-compiled List Pro-*

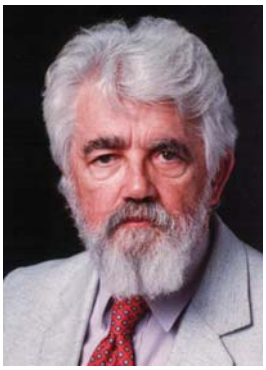

Figura 1.3: John McCarthy (1927-2011)

cessing Language), que se implementó en el FORTRAN con que contaba la IBM 704 en 1958. Durante los años subsiguientes McCarthy diseñó, refinó e implementó Lisp (*List Processor*), en parte porque FLPL no contaba con recursividad ni con condicionales dentro de las expresiones. La elegancia matemática vendría después.

Lisp no es sólo uno de los lenguajes más viejos que existen (su primera implementación data de 1958), sino también el primero en proporcionar recursividad, funciones como ciudadanos de primera clase, recolección de basura y una definición formal del lenguaje (escrita asimismo en Lisp). Las diversas implementaciones de Lisp desarrolladas a través de los años han sido también pioneras en cuanto al uso de ambientes integrados de programación, los cuales combinan editores, intérpretes y depuradores [20].

Las primeras implementaciones de Lisp tuvieron, sin embargo, algunos problemas que las hicieron perder popularidad, como por ejemplo su tremenda lentitud para efectuar cálculos numéricos, su sintaxis basada por completo en paréntesis que suele causar tremenda confusión entre los usuarios novatos y su carencia de tipos que hace difícil la detección de errores y el desarrollo de compiladores. Además, problemas adicionales tales como la carencia de verdaderas funciones que sean tratadas como ciudadanos de primera clase y el uso de reglas de ámbito dinámico, bajo las cuales el valor de una variable libre se toma del ambiente de activación, hicieron que Lisp se mantuviera durante un buen tiempo como un lenguaje restringido a los laboratorios de investigación, lejos del alcance de un número significativo de usuarios. El mismo McCarthy llegó a decir que su lenguaje no era apropiado "para los progra-

madores novatos o los neófitos en la materia", pues se requería una cierta cantidad de "conocimientos sofisticados para apreciar y usar el lenguaje de manera efectiva" [21].

A partir de 1962, el desarrollo de Lisp divergió en un gran número de dialectos, entre los que destacan MacLisp y ZetaLisp (desarrollados en el MIT—Instituto Tecnológico de Massachusetts), Franz Lisp (desarrollado en la Universidad de California en Berkeley), ICI-Lisp (desarrollado en Stanford), y el InterLisp (un producto comercial desarrollado por el laboratorio privado de investigación Bolt, Boranek y Newman). Todas estas versiones fueron desarrolladas en los Estados Unidos, y se puede hallar más información sobre ellas, así como más detalles sobre la evolución de Lisp en la referencia [33].

En el otoño de 1975 Gerald Jay Sussman y Guy Lewis Steele, Jr. se encontraban estudiando la teoría de los actores como un modelo de computación desarrollado por Carl Hewitt [15] en el MIT. El modelo de Hewitt era orientado a objetos, con una fuerte influencia del lenguaje de programación Smalltalk. De acuerdo a este modelo, cada objeto era una entidad computacionalmente activa capaz de recibir y de reaccionar a mensajes. A estos objetos y a los mensajes que intercambiaban, Hewitt los llamó actores. Un actor podía tener un número arbitrario de *conocidos*; es decir, podía "saber" acerca de la existencia de otros actores y enviarles mensajes. Este mecanismo de intercambio de mensajes era la única forma de interacción proporcionada por el lenguaje de Hewitt, y se utilizaban *continuaciones* para modelar funciones.

Dado que Sussman y Steele estaban teniendo dificultades para entender algunos aspectos teóricos y prácticos del trabajo de Hewitt, decidieron construir un pequeño intérprete del lenguaje de Hewitt usando MacLisp, a fin de poder experimentar con él. La implementación de este intérprete comenzó con una extensión del MacLisp que incluía los mecanismos necesarios para crear actores y enviar mensajes, proporcionando las primitivas necesarias para estudiar el comportamiento interno de los actores.

Como Sussman había estado estudiando Algol en aquella época, le sugirió a Steele comenzar con un dialecto de Lisp que tuviera reglas de ámbito estático (es decir, el valor de una variable libre se toma de su ambiente de definición). Algunos de los aspectos relevantes y los mecanismos necesarios para crear un intérprete de Lisp con reglas de ámbito estático ya habían sido delineados por Joel Moses (ver figura 1.4) [22]. Al parecer, este tipo de mecanismo se requeriría de cualquier manera para llevar un registro de

los *conocidos* de cada actor. Esta decisión les permitió crear actores con la misma facilidad con que se crean las funciones en Lisp (y usando casi los mismos mecanismos). El paso de mensajes se podría entonces expresar sintácticamente en la misma forma en que se invoca una función. La única diferencia entre un actor y una función era que la segunda regresaba un valor, mientras que la primera no regresaba nada, sino que más bien invocaba una continuación, o sea otro actor que sabía de su existencia.

Figura 1.4: Joel Moses (nacido en 1941)

Sussman y Steele se sintieron tan satisfechos con su mini-intérprete que decidieron llamarlo "Schemer", pensando que con el tiempo se convertiría en otro lenguaje que se podría utilizar en Inteligencia Artificial, tal y como PLANNER [14] (el lenguaje desarrollado por Hewitt) y Conniver [37] (un lenguaje desarrollado por Drew McDermott y Gerald Jay Sussman en 1972, como una reacción a las limitaciones de PLANNER). Sin embargo, el sistema operativo ITS (en el que Sussman y Steele desarrollaron originalmente Schemer) limitaba los nombres a 6 letras, por lo que el apelativo del intérprete hubo de ser truncado a **Scheme**, que es como se le conoce hoy en día.

Pensando que su intérprete parecía capturar muchas de las ideas que circulaban en aquellos días acerca del manejo de datos y estructuras de control, Sussman y Steele vieron en Scheme el siguiente lenguaje de la Inteligencia Artificial. El reporte original de Scheme [38] describe a un lenguaje sumamente austero, con un mínimo de primitivas (una por concepto), pero marcó el inicio de lo que se convertiría en un lenguaje de programación sumamente popular en las esferas académicas.

En 1976, Sussman y Steele publicaron dos artículos en los que se hablaba sobre semántica de los lenguajes de programación usando Scheme: "Lambda: The Ultimate Imperative" [34] y "Lambda: The Ultimate Declarative" [31], en los cuales se enfatizó el hecho de que Scheme podía soportar eficientemente

los principales paradigmas de programación actuales (es decir, imperativa, funcional y orientada a objetos). Esta última publicación se convirtió de hecho en la propuesta de tesis de Maestría de Steele, que culminó en el desarrollo de un compilador de Scheme llamado RABBIT [32].

Figura 1.5: Mitchell Wand

Mitchell Wand (ver figura 1.5) y Daniel Friedman (ver figura 1.6) se encontraban realizando trabajo similar al de Steele en la Universidad de Indiana [39] y mantuvieron abierta una línea de comunicación durante ese período que resultó muy provechosa para la evolución de Scheme.

Figura 1.6: Daniel Friedman

A raíz del trabajo de Steele con RABBIT, éste y Sussman publicaron un reporte revisado sobre Scheme en 1978 [36], cuyo título pretendió rendir tributo al legendario Algol, pero que, de acuerdo al propio Steele, propició una serie de "títulos cada vez más tontos" [7, 8, 26]. Poco después, escribieron una monografía en varias partes titulada "El Arte del Intérprete" [35], que

ilustraba numerosos mini-intérpretes de Lisp con diversas variaciones. Esta monografía nunca se terminó, y sólo las partes Cero, Uno y Dos fueron publicadas. La parte Cero introdujo un pequeño dialecto de Lisp de primer orden modelado con base en ecuaciones recursivas. La parte Uno discutía el uso de procedimientos como datos y exploraba las asociaciones léxicas y dinámicas. La parte Dos se refería a la descomposición del estado y el significado de los efectos secundarios. La parte Tres iba a cubrir el orden de evaluación (llamadas por valor vs. llamadas por nombre) y la parte Cuatro abarcaría los metalenguajes, macro procesadores y compiladores. Aunque estas últimas dos partes no se escribieron, otros investigadores abordaron posteriormente estos temas, por lo que la pérdida no se considera cuantiosa [33]. Curiosamente, pese a que "El Arte del Intérprete" logró cierta notoriedad en el inframundo de los aficionados a Scheme, fue rechazado por una revista de la ACM [33].

Históricamente, Scheme contribuyó a estrechar la brecha entre los teóricos (que estudiaban el cálculo lambda y el modelo de actores) y los prácticos (implementadores y usuarios de Lisp) en el área de lenguajes de programación. Además, Scheme hizo la semántica denotacional mucho más accesible a los programadores y proporcionó una plataforma operacional que permitiera a los teóricos realizar sus experimentos. Por su reducido tamaño, no había necesidad de tener una versión centralizada del lenguaje que tuviera que soportar un gran número de plataformas, como sucedía con Lisp. De tal forma, brotaron por todas partes implementaciones y dialectos de Scheme hacia inicios de los 1980s. Un ejemplo es el Scheme 311 [13], desarrollado en la universidad de Indiana varios años antes que alguien hiciera un intento por producir una implementación aislada del lenguaje (es decir, que no tuviera que montarse sobre Lisp).

En los años subsiguientes se llevó a cabo mucho trabajo sobre diversos aspectos de la implementación de Scheme en Yale y posteriormente en el MIT, por conducto de Jonathan Rees, Norman Adams y otros. Estos esfuerzos produjeron el dialecto de Scheme llamado "T", cuyo diseño estaba guiado por la eficiencia [25] y que modeló en gran medida el estándar que se conoce hoy en día. El nombre "T" fue una broma, pues ese caracter representa el valor Booleano *True* (Cierto), que es retornado típicamente por las funciones de Scheme, y que se eligió en contraposición a *NIL* (un dialecto de MacLisp), que significa *Nulo* o *Falso* en el contexto de Lisp.

El diseño de "T" fue no sólo un evento separado de la tradición de Lisp, sino también de otras versiones anteriores de Scheme, pues fue una imple-

mentación más limpia del lenguaje, con mayor regularidad en sus primitivas y una mayor pureza estilística [25]. "T" fue diseñado originalmente para la VAX (bajo Unix y VMS) y las estaciones de trabajo *Apollo*. La mayor parte del sistema se escribió en "T" y se convirtió a código ejecutable con el compilador de "T" llamado "TC". Particularmente, cabe destacar que el evaluador de expresiones y el recolector de basura se escribieron en "T" y no en lenguaje máquina. El proyecto "T" empezó con una versión del compilador de Lisp S-1 [3] y logró realizar mejoras sustanciales, identificando y corrigiendo de paso algunos errores del compilador S-1 y del reporte original de RABBIT [36]. Al igual que el compilador S-1, el compilador TC hizo uso intensivo de las estrategias de optimización de compiladores existentes en la literatura de aquel entonces [33], basándose sobre todo en el trabajo de William Wulf (ver figura 1.7) y otros en el compilador BLISS-11 [40].

Figura 1.7: William Wulf (nacido en 1939)

Una segunda generación del compilador "T", llamada ORBIT [18] integró una pléyade de estrategias de optimización generales y específicas para Lisp, produciendo un ambiente de desarrollo para Scheme de muy alta calidad [33].

Poco a poco, la comunidad de usuarios de Scheme creció de unos cuantos aficionados a un grupo mucho más grande que se caracterizaba por tener un interés común en los aspectos matemáticos de los lenguajes de programación. El pequeño tamaño de Scheme, sus raíces en el cálculo lambda y su semántica tan compacta, lo hicieron un vehículo idóneo para la investigación y la enseñanza. Los grupos más fuertes de soporte de Scheme aparecieron en el MIT, la universidad de Indiana y la universidad Rice. La mayoría de estos grupos fueron iniciados por egresados del MIT o la universidad de Indiana

que después se volvieron profesores de las universidades antes mencionadas.

En el MIT, Yale y la universidad de Indiana, Scheme comenzó a adoptarse como un lenguaje de enseñanza para cursos de su licenciatura en ciencias de la computación en 1981 y 1982.

Tras la publicación del libro "Estructura e interpretación de programas de computadora" en 1985 [1], Scheme se volvió un clásico y su uso se propagó rápidamente a un sinnúmero de universidades en los Estados Unidos y otras partes del mundo. Consecuentemente, varias compañías comenzaron a desarrollar versiones comerciales de Scheme. Por ejemplo, la empresa *Cadence Research Systems* fue creada por R. Kent Dybvig; su implementación, llamada *Chez Scheme*, corría en varios tipos de estaciones de trabajo. La empresa *Semantic Microsystems*, fundada por William Clinger, Anne Hartheimer y John Ulrich, produjo *MacScheme* para la Macintosh de Apple. La *Texas Instruments* desarrolló el *PC Scheme* para la IBM PC y compatibles.

El "Reporte Revisado de Scheme" original [32] fue adoptado como modelo para las definiciones posteriores del lenguaje. En octubre de 1984, quince representantes de las implementaciones de Scheme más importantes de ese entonces se reunieron para desarrollar un estándar del lenguaje, y se hicieron cargo de la consecuente evolución del mismo. Los resultados de este esfuerzo conjunto se publicaron como reportes técnicos en el MIT y la universidad de Indiana en el verano de 1985 [11]. Este comité adoptó como su regla básica el agregar características extra al lenguaje únicamente mediante consenso unánime. De tal forma, cualquier cosa agregada a Scheme en subsecuentes revisiones ha resultado ser adecuada y planeada cuidadosamente. La tercera revisión del estándar de Scheme se efectuó en la primavera de 1986, dando lugar a otro reporte [26]. El 2 de noviembre de 1991 se publicó la cuarta revisión del reporte [6], producto de los esfuerzos efectuados en la reunión que precedió a la Conferencia de la ACM (*Association for Computing Machinery*) sobre Lisp y Programación Funcional de 1988. El 25 de junio de 1992 se efectuó otra reunión en el Centro de Investigación de Xerox en Palo Alto, California, a fin de trabajar los detalles técnicos de la quinta versión del reporte, y los principales temas discutidos fueron publicados por Jonathan Rees [24].

Curiosamente, los comités encargados de la evolución de Lisp nunca han podido colaborar con los de Scheme, y el proceso de estandarización de ambos lenguajes se ha dado por vías distintas. El esfuerzo por estandarizar Scheme se inició en 1989, y culminó en 1991 con un estándar del IEEE y de ANSI [29].

1.2 Versiones de Scheme

Actualmente existe un gran número de versiones de Scheme disponibles en forma comercial y gratuita, muchas de ellas a través de Internet. Las más populares son las siguientes [9]:

- **MIT Scheme**[1]: Ambiente completo de programación que incluye editores, documentación y programas de ejemplo. Está disponible para Windows, Linux y Mac OS X y es gratuito.

- **MacGambit Scheme**[2]: Esta es una popular versión de Scheme disponible de forma gratuita para Mac OS X (32 y 64 bits), Windows (32 y 64 bits), iPhone, iPod touch y iPad. Su código fuente está disponible en C/C++ y cuenta con documentación y un depurador de código. Adicionalmente, existe una herramienta llamada Gambit-C que permite generar código en C, a partir de código escrito en Gambit Scheme, de tal forma que puedan producirse ejecutables independientes a partir de programas escritos en Scheme.

- **Scheme 48**[3]: Esta versión se basa en la arquitectura de una máquina virtual, a fin de permitir una gran portabilidad del intérprete. El código es fácilmente transportable a cualquier computadora que tenga POSIX y un compilador que soporte el C de ANSI (*American National Standards Institute*). Sigue muy de cerca el estándar más reciente [6] y soporta multitareas, manejo de excepciones, arreglos, tablas de direcciones calculadas (*hash tables*), etc. La versión más reciente cuenta con instaladores para Linux y Windows. Esta es la versión de Scheme con la cual se elaboraron los ejercicios proporcionados en estas notas.

1.3 ¿Por qué Scheme?

La primera pregunta que uno se hace cuando se pretende estudiar un nuevo lenguaje de programación es: ¿para qué hacerlo? Si el lenguaje o lenguajes que ya conocemos puede(n) resolver la tarea en cuestión, parece un lujo

[1] Puede descargarse de: `https://www.gnu.org/software/mit-scheme/`
[2] Puede descargarse de: `http://dynamo.iro.umontreal.ca/wiki/index.php/Main_Page`
[3] Puede descargarse de: `http://s48.org/`

innecesario utilizar un lenguaje del que lo único que sabemos es que posiblemente permita escribir una solución más elegante o más simple. Scheme es un caso extraño a este respecto, pues pese a ser muy pequeño y simple (comparado con C o Lisp), es muy poderoso. Guy L. Steele, Jr. lo llama "la navaja suiza" de los lenguajes de programación [30] porque proporciona las herramientas básicas para sacarnos de cualquier apuro, aunque nada tan aparatoso como para construir una casa), y posee innumerables atributos que lo hacen idóneo para la enseñanza, permitiendo a los estudiantes enfocarse más en el diseño de algoritmos (y programas) que en la sintaxis (como suele suceder con los lenguajes de programación tradicionales, como Pascal o C). Scheme permite programar en modo imperativo, funcional y orientado a objetos, y su extrema simplicidad hace muy fácil para un estudiante poder aprenderlo bien en un semestre.

Algunas de las características de Scheme son las siguientes [9, 10]:

- Todos los objetos son ciudadanos de primera clase : Todos los objetos de Scheme tienen los siguientes derechos inalienables [5]:

  1. Tienen el derecho a permanecer anónimos.
  2. Tienen una identidad que es independiente de cualquier nombre por el que se les identifique.
  3. Pueden almacenarse en variables y en estructuras de datos sin perder su identidad.
  4. Pueden ser retornados como el resultado de la invocación de un procedimiento.
  5. Nunca se mueren.

  En la mayoría de los lenguajes de programación los objetos que son ciudadanos de primera clase son aquellos que pueden acomodarse fácilmente en un registro de la computadora, como por ejemplo los números, los valores Booleanos y los apuntadores. Asimismo, en la mayoría de los lenguajes de programación los números pueden usarse sin que se les asocien nombres, pero no puede hacerse lo mismo con los procedimientos; los caracteres y los apuntadores pueden almacenarse en un solo registro, pero no puede hacerse lo mismo con las cadenas; los Booleanos viven para siempre, pero los vectores pueden morir cuando el procedimiento

que los creó concluye su ejecución. De tal forma, dado que Scheme reune todas las características antes mencionadas, se constituye como un lenguaje muy diferente a la mayoría de los lenguajes de programación tradicionales. La razón por la que a las implementaciones de Scheme normalmente no se les acaba el espacio de almacenamiento de manera inmediata, es porque se les permite reclamar el espacio ocupado por un determinado objeto si puede probarse que dicho objeto no volverá a ser utilizado en ninguna operación futura (a este proceso se le llama *recolección de basura*).

- Manejo implícito de almacenamiento : Tal y como se mencionó en el punto anterior, en Scheme no tenemos que preocuparnos acerca del manejo de almacenamiento para nuestros datos, porque tal espacio es asignado conforme se requiere. Cualquier región de memoria que se vuelve inaccesible se reasigna automáticamente en vez de que el programador lo tenga que hacer de manera manual.

  La ausencia de código explícito para reasignar memoria hace que los programas en Scheme sean más simples y más cortos (no se necesitan comandes tales como **dispose** en Pascal o **free** en C para reasignar memoria). La consecuencia natural de este esquema es que la implementación del lenguaje debe realizar la recolección de basura antes mencionada para reclamar el espacio que se ha vuelto inaccesible.

1.4 Conceptos Básicos

Para poder empezar a escribir programas en Scheme debemos estar familiarizados con los siguientes conceptos:

- **Abstracción**: Se refiere al ocultamiento de detalles innecesarios y permite que los patrones recurrentes se puedan expresar de manera concisa.
- **Caracteres Especiales**: En Scheme los caracteres **( ) [ ] { } ; , “ ” ‘ ’ #** tienen un significado especial y no pueden aparecer en los símbolos. De manera análoga, los símbolos +, - y . se usan en números y pueden aparecer en cualquier posición de un símbolo, excepto como el primer carácter.
- **Números**: Estos no son considerados como símbolos; forman una categoría aparte.

- **Variables**: Son símbolos usados para representar algún valor.

- **Expresiones**: Pueden consistir de un solo símbolo, o número, o una lista.

- **Lista**: Consiste de un paréntesis izquierdo seguido por expresiones separadas por espacios en blanco y concluye con un paréntesisis derecho.

- **Lista vacía**: Es una lista sin elementos. Se denota mediante (). En algunos intérpretes se retorna la lista vacía también cuando se evalúa como falsa una expresión booleana.

- **Programa**: En general, se refiere a cualquier expresión (o conjunto de expresiones) que puede introducirse como respuesta a la petición de acciones de la línea de comandos (el denominado *prompt* del sistema) `<RETURN>`.

- **Ciclo Lee-Evalúa**: Este es el nombre de la secuencia de eventos que Scheme hace en sucesión: lee, evalúa e imprime. Si durante cualquiera de estas etapas ocurre algo inadecuado se accionará un mensaje de error.

- **Expresión de definición**: Es una forma de expresión identificada por un símbolo especial llamado **palabra clave** la cual es, en este caso, **define**. Su formato general es:

  (**define** *var expr*)

  Por ejemplo:
  (**define** cinco 5)

- **Encomillar**: Esta operación (llamada *quoting* en inglés) se usa para indicar al intérprete que queremos usar el valor literal de un símbolo. Se encomilla un símbolo encerrándolo entre paréntesis y usando la palabra clave **quote**. Por ejemplo: (**quote** simbolo). No es necesario encomillar números puesto que el valor de un número es el número mismo. Podemos también encomillar un símbolo precediéndolo de un apóstrofe (').

- **Constante**: Un objeto cuyo valor es el mismo que el objeto mismo.

- **Símbolo**: Es un objeto de datos con dos características muy importantes.

  - Se representan simplemente mediante sus nombres.
  - Dos símbolos son idénticos sólo si sus nombres se escriben de la misma forma.

- **Notación prefija**: En esta notación, el operador precede a los operandos. Por ejemplo + **3 4**. Scheme usa notación prefija. Por ejemplo:
  $>$ (+ **3 4**)
  7

- **Preservación de mayúsculas**: Se dice que un lenguaje tiene el mecanismo de preservación de mayúsculas (llamado *case preserving* en inglés) cuando distingue entre mayúsculas y minúsculas. La mayoría de las versiones de Scheme no tiene este mecanismo, ya que todos lo objetos suelen convertirse a minúsculas, excepto por los caracteres contenidos en comillas dobles (las cadenas). Es sólo en este último caso que se preservan los caracteres en mayúsculas y minúsculas.

- **Operadores aritméticos**: Scheme tiene todos los operadores aritméticos básicos: * para la multiplicación, + para la suma, - para la resta y / para la división.

- **Procedimientos**: El procedimiento es un mecanismo fundamental para la abstracción en Scheme. La mayoría de los intérpretes en Scheme tiene dos clases de objetos procedurales:

  1. **Cierre Léxico**: Es un objeto procedural que encapsula un proceso computacional a ejecutarse y define el ambiente en el cual se ejecutará.
  2. **Continuaciones**: Es un objeto de datos de primera clase que representa el estado futuro de un proceso computacional. Se usa con mayor frecuencia para implementar procedimientos de escape pero es, sin embargo, un mecanismo mucho más general que puede usarse para implementar una amplia variedad de constructores de control.

Los procedimientos de Scheme son objetos por derecho propio (es decir, son ciudadanos de primera clase). Los procedimientos pueden ser creados dinámicamente, ser almacenados en estructuras de datos, ser retornados como resultado de un procedimiento, y así sucesivamente. Los argumentos que se transmiten a un procedimieto en Scheme siempre se pasan por valor, lo que significa que las expresiones que conforman un argumento se evalúan antes de que se pase el control de la ejecución al procedimiento invocado. Esto se hace independientemente de si el procedimiento necesita o no el resultado de la evaluación.

Scheme fue uno de los primeros lenguajes de programación en incorporar procedimientos como ciudadanos de primera clase, tal y como el cálculo lambda. Con ello se proporcionan las ventajas de las reglas de entorno estático y de las estructuras de bloques en un lenguaje con tipos dinámicos. Scheme fue el primer dialecto importante de Lisp en distinguir los procedimientos de las expresiones lambda y los símbolos. También fue el primero en usar un solo ambiente léxico para todas las variables y en evaluar la posición de un operador en una invocación procedural en la misma forma que la posición de un operando. Al basarse completamente en las invocaciones procedurales para efectuar iteraciones, Scheme enfatiza el hecho de que las invocaciones a procedimientos con recursión de cola son esencialmente goto's (o sea, saltos absolutos) que permiten paso de argumentos. Scheme fue el primer lenguaje de uso extendido en adoptar procedimientos de escape como ciudadanos de primera clase, desde los cuales pueden sintetizarse todas las estructuras de control secuenciales que se conocían previamente.

1.4.1 Problemas resueltos

1. ¿Cuál es el resultado de introducir cada una de las siguientes expresiones en la línea de comandos de Scheme? Verifique sus respuestas efectuando los experimentos correspondientes en la computadora (las respuestas se muestran en **negritas**).

   a. (- 10 (- 8 (- 6 4)))
   4

   b. (/ 40 (* 5 20))
   2/5

c. (+ (* 0.1 20) (/ 4 -3))
0.6666666666

2. Escriba las expresiones en Scheme que denoten el mismo cálculo que las siguientes expresiones aritméticas. Verifique sus respuestas probando directamente en la computadora las expresiones propuestas.

   a. (4 × 7) - (13 + 5)

   En Scheme: **(- (* 4 7) (+ 13 5))**
   Resultado: **10**

   b. (3 × (4 + (-5 - -3)))

   En Scheme: **(* 3 (+ 4 (- -5 -3)))**
   Resultado: **6**

   c. (2.5 ÷ (5 × (1 ÷ 10)))

   En Scheme: **(/ 2.5 (* 5 (/ 1 10)))**
   Resultado: **5.0**

   d. 5 × ((537 × (98.3 + (375 - (2.5 × 153)))) + 255)

   En Scheme: **(* 5 (+ (* 537 (+ 98.3 (- 375 (* 2.5 153)))) 255))**
   Resultado: **245073**

1.5 Construyendo listas

Scheme proporciona un procedimiento para construir listas a razón de un elemento a la vez. Su nombre es **cons**, que es una abreviacion de *constructor*. Considere los siguientes ejemplos:

>(**cons** 7 '())
(7)

>(**define** ls1 (cons 7 '()))
>ls1
(7)

>(**define** ls2 (cons 9 ls1))
>ls2
(9 7)

Ahora intentemos construir la lista **((2 1) tres 2 1)**, para lo cual usaremos:

>(**cons** (**cons** 2 (**cons** 1 '())) (**cons** 'tres (**cons** 2 (**cons** 1 '()))))

Para hacer que la lista sea tomada literalmente, se usa el símbolo de encomillado simple.

>'((2 1) tres 2 1)
((2 1) tres 2 1)

Los elementos de la lista previa se tomarán literalmente (es decir, sin interpretarse).

1.5.1 Problemas resueltos

1. Usando los símbolos **uno** y **dos** y el procedimiento **cons** podemos construir la lista (**uno dos**), escribiendo **(cons 'uno (cons 'dos'()))**. Usando los símbolos **uno, dos, tres** y **cuatro** y el procedimiento **cons**, construya las listas siguientes sin usar listas encomilladas (puede usar el símbolo de encomillado simple y la lista vacía):

   a. (uno dos tres cuatro)
   (cons 'uno (cons 'dos (cons 'tres (cons 'cuatro '()))))

   b. (uno (dos tres cuatro))
   (cons 'uno (cons (cons 'dos (cons 'tres
   (cons 'cuatro '()))) '()))

c. (uno (dos tres) cuatro)
(cons 'uno (cons (cons 'dos (cons 'tres '()))
(cons 'cuatro '()))))

d. ((uno dos) (tres cuatro))
(cons (cons 'uno (cons 'dos '()))
(cons (cons 'tres (cons 'cuatro '())) '()))

e. (((uno)))
(cons (cons (cons 'uno '()) '()) '())

1.6 Separando Listas

Hay dos procedimientos selectores en Scheme a los que se denomina **car** y **cdr**. Sus nombres se originaron de la forma en que se implementó LISP en la IBM 704. En esta computdora se podían accesar las partes de la **dirección** y el **decremento** de una posición de memoria. Por lo tanto, **car** es el acrónimo de *contents of address register* (contenido del registro de dirección) y **cdr** es el acrónimo de *contents of decrement register* (contenido del registro de decremento). Ambos selectores suponen que su argumento será una lista no vacía. Si se les aplica a una lista vacía generarán un mensaje de error.

> (**car** '(1 2 3 4))
1

> (**car** '((1) (2) (3) (4)))
(1)

car regresa el primer elemento de alto nivel de una lista.

> (**cdr** '(1 2 3 4))
'(2 3 4)

> (**cdr** '(()))
()

> (**cdr** '((1 2)))
()

cdr regresa una lista que contiene todos los elementos de la lista original excepto por el primero (el **car** de la lista). Podemos combinar **cons, car** y **cdr** para manipular listas de cualquier forma que queramos. Las aplicaciones sucesivas de **car** y **cdr** se facilitan si usamos los procedimientos **caar, cadr, caddr, ..., cddddr**. El número de **a**s y **d**s entre la **c** y la **r** nos indican cuántas veces se aplicará **car** o **cdr** de derecha a izquierda. Por ejemplo:

> (**cadr** '(a b c))
b

es equivalente a:

> (**car** (**cdr** '(a b c)))
b

1.6.1 Pares Punteados

Normalmente presuponemos que el segundo argumento de **cons**, es una lista, pero ese no tiene que ser el caso. Sin embargo, si este argumento no es una lista, el resultado de aplicar **cons** será un **par punteado**. Un par punteado se escribe como una pareja de objetos separados por un punto y encerrados en paréntesis. El primer objeto en el par punteado es el **car** del par punteado, y el segundo objeto es su **cdr**. Considere los siguientes ejemplos:

> $\bar{(}$**cons** 'a 'b) $\Longrightarrow$ **(a . b)**

> $\bar{(}$**car** '(a . b)) $\Longrightarrow$ **a**

> $\bar{(}$**cdr** '(a . b)) $\Longrightarrow$ **b**

> $\bar{)}$(a . ()) $\Longrightarrow$ **(a)**

> $\bar{)}$(a . (b c)) $\Longrightarrow$ **(a b c)**

Una cadena de parejas que no termine con la lista vacía se denomina **lista impropia**. Advierta que una lista impropia realmente no es una lista. Puede combinarse la notación de lista con la de par punteado para representar listas impropias.

(a b c . d)

es equivalente a

(a . (b . (c . d)))

Cualquier elemento construido con **cons** es denominado un **par**. Recuerde también que los procedimientos **car, cdr** y **cons** no alteran a sus operandos.

1.7 Predicados

Un **predicado** toma un argumento y regresa cierto **#t** o falso (**#f** o **()**) dependiendo del resultado de la expresión que se le pida evaluar. Algunos de los predicados existentes son los siguientes:

(**number?** *arg*) :	Evalúa si *arg* es un número
(**symbol?** *arg*) :	Evalúa si *arg* es un símbolo
(**boolean?** *arg*) :	Evalúa si *arg* es un número booleano
(**pair?** *arg*) :	Evalúa si *arg* es un par
(**null?** *arg*) :	Evalúa si *arg* es una lista vacía
(**procedure?** *arg*) :	Evalúa si *arg* es un procedimiento

1.8 Checando igualdad

Hay varios procedimientos para checar igualdades en Scheme y cada uno de ellos está destinado a un cierto tipo de dato, aunque la mayor parte de la gente suele usar **equal?** en la práctica, ya que este procedimiento es el más general y trabaja con todo tipo de datos. Los otros procedimientos son los siguientes:

=	Compara dos números
eq?	Evalúa igualdad de símbolos
eqv?	Evalúa igualdad de números, símbolos y booleanos

Tanto **eq?** y **eqv?** tienen problemas cuando se aplican a pares (o listas). Cada aplicación de **cons** produce un par nuevo y distinto. Por lo tanto:

> (**eq?** (**cons** 1 '(2 3)) (**cons** 1 '(2 3)))
#f

> (**eqv?** (**cons** 1 '(2 3)) (**cons** 1 '(2 3)))
#f

Como se mencionó antes, el predicado de igualdad universal es **equal?** y sirve para determinar igualdad de números, símbolo, booleanos, procedimientos, listas, cadenas, caracteres y vectores. La razón por la que hay varios predicados de igualdad es la eficiencia. Para los símbolos, el uso de = es más eficiente que **equal?**, **eq?** y **eqv?**.

1.8.1 Problemas Resueltos

1. Evalúe cada una de las expresiones siguientes (el resultado se muestra en **negritas**).

   a. (cdr '((a (b c) d))) $\Longrightarrow$ **()**
   b. (car (cdr (cdr '(a (b c) (d e))))) $\Longrightarrow$ **(d e)**
   c. (car (cdr '((1 2) (3 4) (5 6)))) $\Longrightarrow$ **(3 4)**
   d. (cdr (car '((1 2) (3 4) (5 6)))) $\Longrightarrow$ **(2)**
   e. (car (cdr (car '((gato perro gallina))))) $\Longrightarrow$ **perro**
   f. (cadr '(a b c d)) $\Longrightarrow$ **b**
   g. (cadar '((a b) (c d) (e f))) $\Longrightarrow$ **b**

2. Por cada una de las listas siguientes, escriba una expresión **car** y **cdr** que permita extraer el símbolo **a**:

   a. (b c a d)
      Respuesta: **(car (cdr (cdr '(b c a d))))**
   b. ((b a) (c d))
      Respuesta: **(car (cdr (car '((b a) (c d)))))**
   c. ((d c) (a) b)
      Respuesta: **(car (car (cdr '((d c) (a) b))))**
   d. (((a)))
      Respuesta: **(car (car (car '(((a))))))**

3. Decida si las siguientes expresiones son ciertas o falsas:

a. (symbol? (car '(gato raton))) $\Longrightarrow$ **#t**
b. (symbol? (cdr '((gato raton)))) $\Longrightarrow$ **#f**
c. (symbol? (cdr '(gato raton))) $\Longrightarrow$ **#f**
d. (pair? (cons 'hound '(perro))) $\Longrightarrow$ **#t**
e. (pair? (car '(cheshire gato))) $\Longrightarrow$ **#f**
f. (pair? (cons '() '())) $\Longrightarrow$ **#t**

4. Decida si las expresiones siguientes son ciertas o falsas:

a. (eqv? (car '(a b)) (car (cdr '(b a)))) $\Longrightarrow$ **#t**
b. (eqv? 'flea (car (cdr '(dog flea)))) $\Longrightarrow$ **#t**
c. (eq? (cons 'a '(b c)) (cons 'a '(b c))) $\Longrightarrow$ **#f**
d. (eqv? (cons 'a '(b c)) (cons 'a '(b c))) $\Longrightarrow$ **#f**
e. (equal? (cons 'a '(b c)) (cons 'a '(b c))) $\Longrightarrow$ **#t**
f. (null? (cdr (cdr '((a b c) d)))) $\Longrightarrow$ **#t**
g. (null? (car '(()))) $\Longrightarrow$ **#t**
h. (null? (car '((())))) $\Longrightarrow$ **#f**

1.8.2 Problemas resueltos adicionales

a. (pair? '()) $\Longrightarrow$ **#f**
b. (pair? '(a b)) $\Longrightarrow$ **#t**
c. (pair? '1) $\Longrightarrow$ **#f**
d. (symbol? (car '(cat mouse))) $\Longrightarrow$ **#t**
e. (number? 3) $\Longrightarrow$ **#t**
f. (number? 'a) $\Longrightarrow$ **#f**
g. (symbol? '(a)) $\Longrightarrow$ **#f**
h. (boolean? (number? '(b))) $\Longrightarrow$ **#t**
i. (boolean? '(f)) $\Longrightarrow$ **#f**
j. (null? '()) $\Longrightarrow$ **#t**
k. (null? (cdr '(a))) $\Longrightarrow$ **#t**
l. (procedure? cons) $\Longrightarrow$ **#t**
m. (procedure? +) $\Longrightarrow$ **#t**
n. (procedure? 'cons) $\Longrightarrow$ **#f**
o. (= 3 (+ 2 1)) $\Longrightarrow$ **#t**

símbolo	**lista**
tom	'(kevin () ((y (mi (primo (tom ())) estan))) aqui)
15	'(() 34 (ley 45) (90 10) ((12)) () 15)
Harry	'(((((harry) come) ()) manzanas))
bueno	'(este es realmente (((11 3 5 muy) muy)) bueno)

Tabla 1.1: Símbolos y listas requeridas para el ejercicio propuesto No. 5.

p. (= 8 (- 9 2)) $\Longrightarrow$ **#f**
q. (eq? 'cat 'dog) $\Longrightarrow$ **#f**
r. (eq? 'cat 'cat) $\Longrightarrow$ **#t**
s. (eqv? 5 'five) $\Longrightarrow$ **#f**
t. (eqv? 6 (+ 4 2)) $\Longrightarrow$ **#t**

1.9 Ejercicios Propuestos

1. ¿Cual es el resultado entero de cada una de las siguientes expresiones dentro de Scheme?

   a. (/ (* (- 2 3) (/ (+ 9 8) 5)) (+ 7 (* 4 12)))
   b. (* (+ (+ 2 -9)) (/ (- -5 7) -8) (* (* 3 (* 15 (+ 6 1)) 14) (- 65 42)))
   c. (+ (/ (* 9 (/ 7 5)) (- 8 -7)) (+ (/ (* 6 7) (/ 2 3)) 17))
   d. (* 4 (/ (* (* (* (* (* (* (* 2 4) 4) 6) 6) 8) 8) 10) (* (* (* (* (* (* 3 3) 5) 5) 7) 7) 9) 9)))

2. Escriba una expresión en Scheme que denote los mismos cálculos que la siguientes expresiones aritméticas:

   a.

$$\frac{(5+9)\times(8-7)-(-6\times(-4))}{(62+15+17-90)\times(18-12)}$$

b.

$$8 \times \left(\frac{3}{2} - \frac{1}{5}\right) + \frac{7}{4} + \left(\frac{1}{2} \times \frac{3}{7}\right) + (-2)$$

c.

$$\frac{15}{\frac{9+\frac{1}{2}-\frac{5}{3}}{\frac{2}{7}-\frac{1}{8}} + \frac{\frac{8}{3}+\frac{4}{9}}{\frac{3}{5}\times\frac{6}{13}}}$$

d.

$$4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13}\right)$$

3. Indique si cada una de las siguientes expresiones devuelve #t o #f:
   a. (**symbol?** (**car** '(juan ramos)))
   b. (**pair?** (**car** '(() ())))
   c. (**number?** (**cdr** '(12.4 (+ 6 7))))
   d. (**eqv?** (**car** (**cons** 'a '())) (**car** '(a)))

4. Evalúe las siguientes expresiones usando la computadora:
   a. (**car** (**car** (**cdr** (**car** '(((a Mary ((le gusta ()) (comer))) (manzanas ()))))))))
   b. (**car** (**car** (**cdr** (**cdr** '((45) (13) (- 6 7) ((((23) () d) 3) ()) 5)))))
   c. (**cons** (**car** (**cdr** '((12 (5)) a b (c d)))) '())
   d. (**caddr** '(George (no (es)) un (buen () ((tipo)))))

5. Escriba las expresiones en Scheme para extraer el símbolo indicado en la primera columna de la tabla 1.1 de la lista mostrada en la segunda columna de la misma tabla.