Scheme: lo pequeño es bello[1]

Un lenguaje de programación ideal para la enseñanza

Este artículo pretende dar una perspectiva general acerca del lenguaje de programación Scheme, abordando su evolución histórica, sus principales características y su utilidad como herramienta para la enseñanza de la programación y para el desarrollo de software en diferentes áreas. Su diseño simple pero poderoso constituye uno de los mejores ejemplos de cómo debe diseñarse un lenguaje de programación, pues su semántica es sumamente clara y flexible como para permitir la incorporación de los paradigmas funcional, imperativo y orientado a objetos.

Introducción

John Backus nos advertía ya en 1977 [1] 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 elaborar 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" [1] 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 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 pasan a ser el mecanismo clave para la reutilización de código previamente escrito.

Claro que la programación funcional no es la única solución a la programación procedural (la programación lógica es otro serio contrincante). Si quisiéramos generalizar las ideas de 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

---

[1] Publicado en 2 partes:
Coello Coello, Carlos A. "Scheme Lo pequeño es bello (Primera Parte)", *Soluciones Avanzadas. Tecnologías de Información y Estrategias de Negocios*. Año 4, No. 39, 15 de noviembre de 1996, pp. 27-34.
Coello Coello, Carlos A. "Scheme: Lo pequeño es bello (Segunda Parte)", *Soluciones Avanzadas. Tecnologías de Información y Estrategias de Negocios*. Año 5, No. 43, 15 de marzo de 1997, pp. 58-64.

estáticos más que de dinámicos (i.e., 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).

En este artículo se abordará la importancia de la programación funcional y específicamente la de un popular dialecto de Lisp llamado Scheme, cuya simpleza no sacrifica elegancia ni poder, y cuya popularidad como lenguaje de instrucción para no programadores y estudiantes de ciencias de la computación de primer ingreso, crece a pasos agigantados alrededor del mundo.

Antecedentes Históricos

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 en el *cálculo lambda* [2]. De hecho, el cálculo lambda suele considerarse como el primer lenguaje funcional de la historia, aunque no se le consideró como tal en la época en que se desarrolló debido a la carencia de computadoras de que se adolecía en aquel entonces. Sin embargo, podemos decir, sin temor a equivocarnos, que los lenguajes funcionales de 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, pero el mismo John McCarthy ha negado ese hecho [3]. El impacto del cálculo lambda en el desarrollo inicial del Lisp fue mínimo y no ha sido sino hasta recientemente que el popular lenguaje ha comenzado a evolucionar de acuerdo a los ideales del mismo. Por otra parte, Lisp ha tenido una gran 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, produciéndose el llamado FLPL (*FORTRAN-compiled List Processing 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, 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.

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 como ciudadanos de primera clase y el uso de reglas de ámbito dinámicas, 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 programadores novatos o los no programadores", pues se requería una cierta cantidad de "conocimientos sofisticados para apreciar y usar el lenguaje efectivamente" [3].

A partir de 1962, el desarrollo del Lisp divergió en un gran número de dialectos, entre los que destacan MacLisp y ZetaLisp en el Instituto Tecnológico de Massachusetts (MIT), Franz Lisp en la Universidad de California en Berkeley, ICI-Lisp en la Universidad de Stanford, y el InterLisp, un producto comercial desarrollado por Bolt, Boranek y Newman (un laboratorio de investigación privado de los Estados Unidos).

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 desarrollada por Carl Hewitt [4] en MIT. El modelo de Hewitt era orientado a objetos y con una fuerte influencia del Smalltalk. De acuerdo a él, cada objeto era una entidad computacionalmente activa capaz de recibir y de reaccionar a los mensajes. A estos objetos y a los mensajes que intercambiaban, Hewitt los llamó actores. Dado que Sussman y Steele estaban teniendo dificultades para entender algunos aspectos teóricos y prácticos del trabajo de Hewitt, decidieron construir un pequeño intérprete de este lenguaje usando MacLisp, a fin de poder experimentar con él. 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 (i.e., el valor de una variable libre se toma de su ambiente de definición). 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 una función regresaba un valor y un actor no regresaba nada, sino que más bien invocaba una *continuación*, o sea otro actor que sabía de su existencia.

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 pudiera utilizar en inteligencia artificial, tal y como PLANNER, el lenguaje desarrollado por Hewitt. Sin embargo, el sistema operativo ITS 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.

Con la idea en mente de que su intérprete parecía capturar muchas de las ideas que circulaban en aquellos días sobre lenguajes de programación, Sussman y Steele decidieron publicar la definición de Scheme en la forma de un memo del laboratorio de inteligencia artificial del MIT [5]. Esta primera definición del lenguaje era sumamente austera, 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" [6] y "Lambda: The Ultimate Declarative" [7], en los cuales los autores enfatizarían el hecho de que Scheme podía soportar eficientemente los principales paradigmas de programación actuales (i.e., 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 [8].

Mitchell Wand y Daniel Friedman se encontraban en aquel entonces desarrollando un trabajo similar al de Sussman y Steele en la universidad de Indiana, por lo que mantuvieron abierta una línea de comunicación durante ese período que resultó muy provechosa para la evolución de Scheme.

A raíz del trabajo de Steele con RABBIT, se publicó un reporte revisado sobre Scheme en 1978 [9], cuyo título pretendió rendir tributo al legendario ALGOL, pero que, de acuerdo al propio Steele, propició una serie de "títulos cada vez más tontos" [10].

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 80s. Un ejemplo, es el Scheme 311 [11], desarrollado en la universidad de Indiana varios años antes de que alguien hiciera un intento por producir una implementación aislada del lenguaje (i.e., 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 MIT, por conducto de Jonathan Rees, Normal Adams y otros. Estos esfuerzos produjeron el dialecto de Scheme llamado "T", cuyo diseño estaba guiado por la eficiencia y que modeló en gran medida el estándar que se conoce hoy en día. Como parte de este trabajo, Scheme empezó a utilizarse en cursos de licenciatura en MIT, Yale y la Universidad de Indiana en 1981 y 1982.

En 1984, dos profesores del MIT y un programador de Bolt, Beranek y Newman publicaron un libro de texto basado en Scheme [12] que se ha utilizado en cursos introductorios de la licenciatura en ciencias de la computación en MIT y varias otras universidades desde entonces.

Conforme Scheme se fue volviendo un lenguaje de uso más extendido, se empezaron a desarrollar diferentes versiones hasta llegar al punto de saturación babélica en que los estudiantes e investigadores tuvieron que pensar en la estandarización del lenguaje. En octubre de 1984, quince representantes de las implementaciones de Scheme más importantes de ese entonces se reunieron para desarrollar dicho estándar. Los resultados de este esfuerzo conjunto se publicaron como reportes técnicos en MIT y la universidad de Indiana en el verano de 1985 [13]. Una revisión posterior tuvo lugar en la primavera de 1986, dando lugar a otro reporte [14]. El 2 de noviembre de 1991 se publicó la cuarta revisión del reporte [15], producto de los esfuerzos efectuados en la reunión que precedió la Conferencia de la ACM 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 [16] y se encuentra disponible, al igual que los reportes antes mencionados, en formato *PostScript* en el Internet[2].

Versiones de Scheme

Actualmente existe un gran número de versiones de Scheme disponibles en forma comercial y gratuita, a través del Internet. Algunas de las más populares son las siguientes:

• MIT Scheme[3] : ambiente completo de programación que incluye editores, documentación y programas de ejemplo. Corre en las siguientes máquinas y sistemas operativos: DEC Alpha (OSF/1), DEC MIPS (Ultrix), HP 68000 y PA (HP-UX 9), IBM-RS6000 (AIX), NeXT, SGI MIPS (Irix 4), Sun 3, SPARC (SunOS 4.1), y PCs bajo Microsoft Windows 3.1, Windows 95 y Windows NT. Existe una versión para DOS, pero requiere de una contribución de $95 dólares, para cubrir costos de distribución.

• PC Scheme[4] : ambiente de programación desarrollado por Texas Instruments para las IBM PCs y compatibles 286 y 386 con al menos 512K de RAM. Aunque existe una versión comercial distribuída a un costo de $95 dólares, incluyendo la guía del usuario y el manual de referencia del lenguaje, es posible conseguir el intérprete de forma gratuita en el Internet, y posteriormente comprar por separado la guía del usuario [17] y un texto introductorio de programación editado por Texas Instruments [18]. Existe una versión llamada PCS/Geneva[5] que soporta los microprocesadores 486, la interfaz gráfica de Borland™ (llamada BGI), cuenta con facilidades para edición en línea, soporte de memoria extendida y soporte de ratón, entre otras cosas. Esta versión corre tanto en DOS como en OS/2, e incluso en el HP95LX de Hewlett Packard.

• MacGambit[6] : Esta es una popular versión de Scheme disponible de forma gratuita para las máquinas basadas en el microprocesador M68000 y sus sucesores, incluyendo las siguientes computadoras: Sun3, Apollo, HP9000, Amiga, Next y Apple Macintosh. El código fuente del intérprete (escrito en THINK-C 5.0) es proporcionado junto con todas las utilerías necesarias para instalar el sistema completo. La versión para la Macintosh goza de gran popularidad, sobre todo por su manejo de gráficos (a través de las primitivas internas proporcionada por el sistema operativo) y su excelente sistema de ayuda en línea. El autor pide se le envíe a la universidad de Montreal una cantidad que fluctúa entre $25 y $50 dólares como donación voluntaria si uno se encuentra satisfecho con su software.

---

[2] Por ejemplo, pueden conseguirse en los siguientes sitios Web:

http://www.cs.indiana.edu/scheme-repository/home.html
http://www-swiss.ai.mit.edu/scheme-home.html

[3] Disponible en: **http://swiss-ftp.ai.mit.edu/scheme-home.html**

[4] Disponible en: **http://www.cs.indiana.edu/scheme-repository/imp.html**

[5] Disponible por ftp '*anonymous*' en: **cui.unige.ch** en el directorio **/PUBLIC/pcs/**

[6] Disponible por ftp '*anonymous*' en: **ftp.iro.umontreal.ca** en el directorio **/pub/parallele/gambit**

• Scheme 48[7]: 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 máquina de 32-bits que tenga POSIX y un compilador que soporte el C de ANSI (American National Standards Institute). Sigue muy de cerca el estándar más reciente [15] y soporta multitareas, manejo de excepciones, arreglos, tablas de direcciones calculadas (*hash tables*), etc. Esta versión de Scheme ha sido utilizada ampliamente para implementar algoritmos de planeación de movimientos en robots móviles en la universidad de Cornell.

¿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 pueden resolver la tarea en cuestión, parece un lujo innecesario utilizar un lenguaje del que lo único que sabemos es que posiblemente permita escribir una solución más elegante o simple. Scheme es un caso extraño a este respecto, pues pese a ser muy pequeño y simple, es asimismo sumamente poderoso, a tal grado que se ha llegado a utilizar como herramienta seria para el desarrollo de aplicaciones y software de sistemas, y muchas de sus ideas fueron incorporadas en el famoso Common Lisp. Procedamos pues a enumerar algunas de las características de Scheme:

• Todos los objetos son ciudadanos de primera clase: Todos los objetos de Scheme tienen los siguientes derechos inalienables:

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 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 máquina, como por ejemplo los números, los booleanos y los punteros. 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 punteros pueden almacenarse, 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, es porque se les permite reclamar el espacio ocupado por un determinado

[7] Disponible por ftp '*anonymous*' en **ftp.cs.indiana.edu** en el directorio **/pub/scheme-repository/imp**

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*).

• Su sintaxis es muy simple y regular: Los programas en Scheme consisten de definiciones de variables y procedimientos mezcladas con expresiones de nivel superior que efectúan inicialización y comienzan la ejecución de un programa. La sintaxis de Scheme emplea una notación prefija completamente rodeada por paréntesis para los programas y (otros) datos; la gramática de Scheme genera un subconjunto del lenguaje usado para los datos. Una consecuencia importante de esta representación tan simple y uniforme es la susceptibilidad de los programas y datos en Scheme al tratamiento uniforme por parte de otros programas [15]. Sólamente existen 6 tipos básicos de expresiones cuya semántica debe ser explicada: constantes, variables, asignaciones, invocaciones de procedimientos, condicionales y expresiones lambda. Todos estos tipos de expresiones, excepto por las lambda, suelen encontrarse en los lenguajes de programación convencionales. El poder de Scheme se deriva de lo extremadamente simple de su semántica, que se encuentra libre de las restricciones artificiales que complican y estropean a los lenguajes convencionales.

• Es un lenguaje orientado a las expresiones: La mayoría de los lenguajes de programación están orientados a las sentencias (o instrucciones); esto implica que tienen que distinguir entre los fragmentos de programa que se comportan como expresiones matemáticas (*expresiones*) de los que cambian el estado de un programa (*sentencias* o *instrucciones*). Este objetivo tiene sentido, pero desgraciadamente resulta difícil de conseguir en la práctica, porque un efecto secundario (cambio del estado de un programa) puede producirse en el cuerpo de una función. De hecho, la mayoría de los lenguajes de programación proporcionan funciones estándar (por ejemplo, la función para generar números aleatorios) que producen efectos secundarios. Scheme evita esta paradoja haciendo indistinguibles las expresiones de las sentencias, a la vez que impone, en la más pura tradición de los lenguajes orientados a las sentencias, un estilo de programación en el que las expresiones se comporten realmente como expresiones matemáticas. Sin embargo, el hecho de que en Scheme usemos la palabra procedimiento (**procedure**, en inglés) en vez de la palabra función, es un recordatorio de que algunos de ellos tienen efectos secundarios.

• Es un lenguaje con tipos dinámicos: Eso significa que no tienen que declararse las variables que se van a utilizar en un programa, porque el tipo es una propiedad de los objetos y no una propiedad de las variables que nombran a los objetos. Los lenguajes con tipos estáticos (i.e., aquellos en que se asocian tipos con las variables) suelen ser más restrictivos, aunque existen excepciones notables tales como ML [19], en el que los tipos de las variables no tienen que declararse porque el compilador usualmente los infiere observando su uso. Los sistemas de tipos dinámicos pueden considerarse como una forma extremedamente simple y general (aunque algo cruda) de polimorfismo [20]. La mayor desventaja de los lenguajes que cuentan con tales sistemas es que resulta imposible capturar los errores de tipos en tiempo de compilación.

• Su mecanismo de paso de parámetros es por valor: Existen varios mecanismos de paso de parámetros, de los cuales los principales son:

1) Paso por valor: Se pasa el valor del parámetro real. Este es el mecanismo adoptado por omisión en lenguajes tales como C, ML y APL.

2) Paso por referencia: Se pasa la dirección de la posición de memoria donde el parámetro real está almacenado. Es el mecanismo utilizado en FORTRAN, en el cual un procedimiento podía cambiar el valor de un parámetro pasado como argumento.

3) Paso por nombre: El parámetro real se revalúa cada vez que resulte necesario durante la ejecución del procedimiento. Es el mecanismo utilizado por Algol 60, y que se utiliza hoy en día en la mayoría de los lenguajes de programación modernos que carecen de efectos secundarios, porque es potencialmente más eficiente con respecto al tiempo y tiene una propiedad de substitución muy general que resulta muy útil cuando se piensa acerca de los programas.

Scheme pasa sus argumentos por valor porque eso le otorga al programador más control sobre los compromisos entre espacio y tiempo y sobre el orden en que se ocurren los efectos secundarios [20].

• Tiene reglas de ámbito estáticas (o léxicas): Eso significa que las funciones pueden accesar cualquier vinculación (*binding*, o sea asociaciones de un valor con un objeto) que aparezca en su ambiente léxico, pero no otras. El ambiente léxico de una función consiste de sus vinculaciones locales, más aquellos de cualquier forma que contenga textualmente la función. En otras palabras, si un lenguaje tiene reglas de ámbito estáticas, puede determinarse cualquier vinculación viendo simplemente el código que rodea al símbolo en cuestión. Si se usan reglas de ámbito dinámico, entonces los símbolos se consideran vinculados hasta que el procedimiento que los encierra regresa (i.e., concluye su ejecución), y debe verse al orden en el que los procedimientos son evaluados para saber las vinculaciones de un determinado símbolo. Un ejemplo que muestra la diferencia entre reglas de ámbito estático y dinámico es el siguiente[8]:

```
(let ((z 0))
  (let ((f (lambda (y) (* y z))))    ; ambiente de definición
    (let ((z 1))
          (f 2))))                    ; ambiente de activación
```

Scheme regresará **0** cuando **(f 2)** se invoque, mientras que cualquier dialecto de Lisp que use reglas de ámbito dinámico regresará **2**. La diferencia se debe al hecho de que Scheme tomará el valor de **z** del ambiente de definición (señalado con comentarios en el código de arriba), mientras que los lenguajes con reglas de ámbito dinámico tomarán el valor de **z** del ambiente de activación.

• Todas las implementaciones del lenguaje deben usar recursión de cola (*tail recursion*): Existen dos formas básicas de implementar recursión en un lenguaje de programación:

[8] Todos los ejemplos incluídos en este artículo fueron probados en PC Scheme, y algunos de ellos pudieran no funcionar en una diferente versión de Scheme. Las palabras reservadas de cada fragmento de código se presentarán, para mayor claridad, con caractereres **remarcados**.

1) Recursión lineal: Se dice que la definición de una función *f* es recursiva lineal si una activación *f(a)* de *f* puede iniciar cuando mucho una nueva activación de *f*. Consideremos la siguiente función para calcular factoriales que emplea recursión lineal:

```
(define fact                    ; versión que usa recursión lineal
        (lambda (n)
          (if (zero? n)
              1
             (* n (fact (- n 1))))))
```

Una activación de **fact** pasando como argumento *n*, regresa 1 si *n* es cero; de lo contrario, inicia una nueva activación de **fact** usando como argumento *n* - 1. De tal forma, cada activación de **fact** puede iniciar cuando mucho una nueva activación de **fact**.
La evaluación de una función recursiva lineal tiene dos fases:

1. Una fase en la cual se inician nuevas activaciones, y
2. Una fase subsecuente en la cual se retorna el control desde las activaciones en una forma LIFO (*Last In First Out*, o sea la última en entrar es la primera en salir).

2) Recursión de cola: Se dice que una función *f* usa recursión de cola si regresa un valor sin necesidad de recursión, o si simplemente retorna el resultado de una activación recursiva. Consideremos la siguiente definición de nuestra función para calcular factoriales que hace uso de la recursión de cola:

```
(define fact          ; versión que usa recursión de cola
  (lambda (n)
    (letrec ((fact-it
              (lambda (val acc)
                (if (zero? val)
                    acc
                   (fact-it (- val 1) (* acc val))))))
      (fact-it n 1))))
```

Todo el trabajo de una función como ésta, en que se hace uso de la recursión de cola, se efectúa en una única fase de iniciación de activaciones. La segunda etapa mencionada anteriormente se vuelve trivial porque el valor calculado mediante la activación final se vuelve el resultado de toda la activación. De tal forma, el mecanismo de iteración puede implementarse muy eficientemente haciendo uso de este tipo de recursión[9], en vez de recurrir a los tradicionales ciclos de los lenguajes procedurales tradicionales.

---

[9] Scheme también proporciona una forma especial llamada **do** para efectuar iteraciones. Un ejemplo de su uso es el siguiente:

```
(do  ((i 1 (+ i 1)))
     ((> i 10) i)
   (display " ")
   (display i))
```

• Soporte de diferentes paradigmas de programación: Scheme es un lenguaje procedural, pero es suficientemente flexible y poderoso como para permitir programación funcional, imperativa y orientada a objetos. Este punto es de vital importancia para entender porqué Scheme es una herramienta ideal para enseñar programación a estudiantes de primer ingreso. El lenguaje es suficientemente poderoso como para cubrir los paradigmas básicos de la programación, pero a la vez es lo suficientemente simple como para poder aprenderlo en un semestre. Por ejemplo, el siguiente programa interactivo para calcular el día de la semana está escrito en un estilo imperativo, aunque haciendo uso de la elengancia que proporciona la recursividad:

```
(define semana-aux
        (lambda (m d y)
        (let ((c (quotient y 100)))
        (let ((y (remainder y 100)))
        (let ((a (quotient (sub1 (* 13 m)) 5)))
        (let ((b (quotient y 4)))
        (let ((e (quotient c 4)))
        (let ((f (+ a b e d y (- (* 2 c)))))
        (let ((r (modulo f 7)))
        (string-append " es " (convertir r
        '("Domingo" "Lunes" "Martes" "Miércoles" "Jueves"
          "Viernes" "Sábado") 0)))))))))))

(define convertir
        (lambda (elem ls base)
        (cond
        ((equal? elem base) (car ls))
        (else (convertir elem (cdr ls) (add1 base))))))

(define semana
        (lambda ()
        (display "Introduzca el mes (1..12): ")
        (let ((m (read)))
        (display "Introduzca el día (1..31): ")
        (let ((d (read)))
        (display "Introduzca el año: ")
        (let ((y (read)))
        (begin
        (if (< m 3)
                (begin
                (writeln d "/" m "/" y (semana-aux (+ m 10) d (- y 1)))
                (newline))
           (begin
                (writeln d "/" m "/" y (semana-aux (- m 2) d y))
                (newline)))
```

---

que produce: **1 2 3 4 5 6 7 8 9 10**

Esta forma simplifica la enseñanza de programación imperativa, pero su uso se vuelve innecesario si se utiliza recursión de cola (cuando se enfatiza el estilo de programación funcional).

```
(display "¿Desea proporcionar otra fecha? ")
        (let ((respuesta (read)))
        (newline)
          (if (eq? respuesta 'no)
            (writeln "Hasta la vista.")
            (semana)))))))))
```

• Procedimientos de escape tratados como ciudadanos de primera clase: En Scheme existe un mecanismo llamado *continuaciones* (*continuations*) que consiste básicamente en la representación de una computación futura. Cada vez que se evalúa una expresión en Scheme hay una continuación esperando el resultado de la misma. La mayor parte del tiempo una continuación incluye acciones especificadas mediante código proporcionado por el usuario (por ejemplo, imprimir un resultado o pasarlo como argumento a otro procedimiento), y sus maquinaciones suelen ser desconocidas y carentes de interés para el programador. Sin embargo, en algunas ocasiones se requiere lidiar explícitamente con las continuaciones, por lo que Scheme proporciona un poderoso procedimiento llamado **call-with-current-continuation**, el cual crea un procedimiento que actúa tal y como la continuación actual, a fin de permitir al programador redefinirla. Las continuaciones pueden usarse como procedimientos de escape, similares a **exit**, **return** o **goto** de los lenguajes tradicionales, pero con la peculiaridad de que en Scheme este mecanismo es tratado como un ciudadano de primera clase, lo que permite almacenar una continuación en una variable o una estructura de datos, además de poder llamarlas tantas veces como se desee y de poder retornarlas y pasarlas como argumento a un procedimiento, como cualquier otro objeto en Scheme. Una breve reseña histórica de las continuaciones puede encontrarse en la referencia [15], partiendo de la invención del operador J por Peter Landin en 1965 y la definición de la forma especial *catch* descrita por Sussman y Steele en su definición de Scheme en 1975 emulando el constructor inventado por Reynolds en 1972 que resultó ser igualmente poderoso que el de Landin, pero mucho más simple. Varios implementadores de Scheme notaron que un procedimiento podría emular todo el poder del constructor *catch*, por lo que decidieron incorporarlo en la definición del lenguaje en 1982, marcando así el nacimiento de **call-with-current-continuation**. Varias personas prefieren utilizar la abreviación **call/cc**, pese a que el nombre original resulta más descriptivo, aunque tal vez más difícil de recordar.

El uso más simple de las continuaciones es para implementar salidas no locales en el caso de detectarse condiciones de error. Por ejemplo, el siguiente procedimiento muestra una aplicación en la que un procedimiento que calcula la suma de los recíprocos de una lista de números regresa un mensaje de error cuando se detecta un cero en la lista dada, deteniendo las operaciones aritméticas y anticipándose al sistema de detección de errores del intérprete:

```
(define division
  (lambda (numeros)
    (call/cc
      (lambda (termina)
        (let ciclo ((numeros numeros))
          (if (null? numeros) 0
            (if (zero? (car numeros))
              (termina "División entre cero")
              (+ (/ 1 (car numeros)) (ciclo (cdr numeros))))))))))
```

Las principales aplicaciones de las continuaciones son: simulación de eventos, búsqueda de retroceso (*backtracking*) [21], multiprocesamiento [22] y corutinas [23].

• Los *streams* permiten implementar la evaluación concisa (*lazy evaluation*): Un *stream* es una secuencia de objetos similar a una lista o un vector, pero con la peculiaridad de que sus componentes son evaluados cuando se accesan y no cuando se almacenan. Este mecanismo, conocido como evaluación concisa, permite la representación de estructuras de datos de gran tamaño, e incluso infinitas, en un espacio finito.

La implementación práctica de *streams* involucra el concepto de evaluación postpuesta, el cual fue implementado por primera vez en el método de paso de parámetros llamado paso por nombre, con que contaba Algol 60. La forma especial **delay** de Scheme permite postponer la evaluación de una expresión, retornando entonces una promesa que se utilizará en el futuro para producir el resultado deseado. Posteriormente, el procedimiento **force** se utiliza para forzar la evaluación de la promesa previamente producida. Consideremos el siguiente ejemplo [17]:

```
(define criba-de-eratostenes
  (lambda (stream)
    (cons-stream
      (head stream)
      (criba-de-eratostenes (filtro
                        (lambda (x) (not (divisible? x (head stream))))
                        (tail stream))))))

(define divisible?
  (lambda (x y)
    (= (remainder x y) 0)))

(define filtro
  (lambda (predicado stream)
    (cond ((empty-stream? stream) the-empty-stream)
          ((predicado (head stream))
           (cons-stream (head stream)
                        (filtro predicado (tail stream))))
          (else (filtro predicado (tail stream))))))

(define extraer
  (lambda (n s)
    (cond
      ((zero? n) (head s))
      (else (extraer (- n 1) (tail s))))))

(define enteros-a-partir-de
  (lambda (n)
    (cons-stream n (enteros-a-partir-de (+ n 1)))))

(define primos (criba-de-eratostenes (enteros-a-partir-de 2)))
```

Si ahora escribimos en el intérprete:

(extraer 50 primos)

la respuesta será *233*, pues ese es el cincuenteavo número primo. Lo interesante es que **primos** es realmente una lista infinita, pero dado el mecanismo de evaluación concisa que incorporan los *streams*, sus elementos no tienen que conocerse ni almacenarse al momento de definirse en el lenguaje. Esto contrasta con los mecanismos de evaluación de adentro hacia afuera con que cuentan la mayoría de los lenguajes tradicionales tales como Pascal y C, que impedirían una definición de esta naturaleza. La definición clave en este ejemplo es la de la forma especial **cons-stream**:

(**cons-stream** *expresión stream*) ≡ (**cons** *expresión* (**delay** *stream*))

No debemos sucumbir a la tentación de tratar de escribir esta equivalencia usando la primitiva **define** de nuestro intérprete (asumiendo que carezca de **cons-stream**), pues eso no funcionará dado que de acuerdo al modelo de computación que sigue Scheme, la parte correspondiente al stream sería evaluada inmediatamente, propiciando un error de desborde de memoria en el intérprete. Por ello es que se requiere que **cons-stream** sea una forma especial, que suele implementarse usando las técnicas descritas por Springer y Friedman [28].
Debe mencionarse también que los *streams* son sumamente utilizados para implementar funciones para el manejo de archivos, para realizar simulaciones y para programación orientada a objetos [17][28].

• Los *motores* (*engines*) permiten abstraer la apropiación (*preemption*) de recursos con respecto al tiempo: Este tipo de objetos fueron introducidos por Texas Instruments en PC Scheme, pero no han sido adoptados por el estándar. Estos objetos, combinados con las continuaciones, permiten que Scheme sea extendido con una gran variedad de facilidades para abstraer procesos, que resultan de gran utilidad en sistemas operativos.

Metafóricamente, un *motor* (*engine*) representa una computación que se lleva a cabo dándole una cierta cantidad de combustible. Si el motor completa su computación antes de que se le agote el combustible, retorna el resultado de la misma y la cantidad de combustible que le queda. Si se le acaba el combustible, se retorna un nuevo motor que, al ejecutarse, continúa la computación.

El combustible de un motor se mide en tictacs (*ticks*), los cuales suelen asociarse con las interrupciones del reloj interno de la computadora (como en PC Scheme).

Los motores se crean usando el procedimiento **make-engine**, el cual toma como argumento un procedimiento sin argumentos (llamado *thunk*), cuyo cuerpo contiene la computación a llevarse a cabo, la cual se termina mediante la invocación del procedimiento **engine-return** con el valor que va a ser regresado por la computación en cuestión. El motor regresado por **make-engine** es un proedimiento de tres argumentos: un entero positivo que especifica el número de tictacs durante el cual va a correrse la computación, un procedimiento a ejecutarse en caso de completarse satisfactoriamente la ejecución (lo llamaremos procedimiento de éxito) y otro a ejecutarse en caso de ejecución fallida (lo llamaremos procedimiento de falla). Si se invoca **engine-return** antes de que se haya cumplido con el número de tictacs asignados,

entonces el procedimiento de éxito es invocado con el valor de la computación y el número de tictacs restantes. Si al motor se le agotan los tictacs antes de que **engine-return** sea invocado, entonces el procedimiento de falla es invocado con un nuevo motor que continúa la computación del motor original cuando se le ejecuta. Cualquier valor retornado por cualquiera de estos dos procedimientos (el de éxito o falla) se vuelve el valor de la invocación del motor. Consideremos el siguiente ejemplo, en el cual queremos calcular el factorial de un número muy grande, siempre y cuando no tome más de cierta cantidad de tiempo:

```
(define factorial
  (lambda (n)
    (cond
      ((zero? n) 1)
      (else (* n (factorial (- n 1)))))))

(define exito
  (lambda (valor ticks) valor))

(define falla
  (lambda (nuevo-motor) nuevo-motor))

(define motor
  (make-engine
    (lambda () (engine-return (factorial 500)))))
```

Si ahora cargamos estas definiciones en el intérprete y realizamos la siguiente invocación:

```
(begin (define motor (motor 1 exito falla)) motor)
```

Scheme responderá con el mensaje:

```
#<PROCEDURE MAKE-ENGINE>
```

indicando que no tuvo tiempo suficiente para realizar la tarea deseada. Si su intérprete es muy rápido, o simplemente incrementa el límite de tiempo asignado al motor (1 en este caso), en vez de este mensaje obtendrá el número correspondiente al factorial de 500. Por ejemplo, el PC Scheme es capaz de calcular tal valor con sólo 5 tictacs de tiempo.

Los motores pueden ser usados para toda una gama de aplicaciones dentro de sistemas operativos, tales como simuladores de tiempo discretos y la implementación, vía tiempo compartido, de abstracciones procedurales tradicionales. Algunos ejemplos pueden encontrarse en la referencia [24].

Proporciona poderosas representaciones numéricas: Aunque algunas versiones de Scheme tienen algunas debilidades en cuanto al manejo de números de punto flotante (por ejemplo Scheme 48), la mayoría de las implementaciones cuentan con una rica variedad de formatos y representaciones numéricas, que incluye números enteros, de punto fijo y flotante, complejos, racionales, etc., si bien sólo unas cuantas de ellas son requeridas por el estándar de lenguaje.

Los números en Scheme pueden ser *exactos* o *inexactos*. Un número es exacto si se deriva de números exactos usando únicamente operaciones exactas. Un número es inexacto si modela una cantidad que se conoce sólo aproximadamente, si se derivó usando componentes inexactos, o si se ha derivado usando operaciones inexactas. De tal forma, la inexactitud es una propiedad contagiosa. Algunas operaciones, tales como la raíz cuadrada (de números no cuadrados), son inherentemente inexactas debido a la precisión finita de la representación. Otros números pueden ser inexactos debido a requerimientos de la implementación. Sin embargo, debe enfatizarse que la exactitud no depende del tipo de número de que se trate. De tal forma, podemos tener enteros inexactos y reales exactos. El poder numérico de Scheme puede apreciarse en el siguiente programa que calcula con cuantos decimales como se quiera:

```
(define (suma-parcial i n e ee base)
  (- (quotient base (* i e))
     (quotient base (* (+ 2 i) ee))))

(define (a n base)
  (do ((i 1 (+ 4 i))
       (delta 1 (suma-parcial i n e (* e n n) base))
       (e n (* e n n n n))
       (suma 0 (+ suma delta)))
      ((zero? delta) suma)))

(define (calc-pi base)
  (- (* 32 (a 10 base))
     (* 16 (a 515 base))
     (* 4 (a 239 base))))

(define (ejecutar)
  (display "¿Cuántos dígitos de pi desea? ")
  (let ((numero (read)))
    (if (and (not (eof-object? numero)) (integer? numero) (positive? numero))
        (let* ((extra (+ 5 (truncate (log numero))))
               (base (expt 10 (+ numero extra)))
               (pi (calc-pi base)))
          (display (quotient pi base))
          (display ".")
          (display (quotient (remainder pi base) (expt 10 extra)))
          (newline)))))

(ejecutar)
```

Scheme como un instrumento para la enseñanza

Aunque Lisp fue pensado originalmente para computación simbólica y su uso estaba restringido a investigadores del área de Inteligencia Artificial (IA) que lo adoptaron rápidamente como una herramienta flexible y poderosa para desarrollar sus proyectos, hoy en día el rango de aplicaciones de este lenguaje y sus dialectos es verdaderamente extenso.

Scheme es un dialecto de Lisp sumamente poderoso y flexible, pero a la vez increíblemente pequeño (su definición ocupa sólamente unas 50 páginas, en comparación con las

220 del C y las más de 1000 del Common Lisp). El diseño de Scheme sigue más de cerca el cálculo Lambda de Alonzo Church, a diferencia de Lisp, y de ahí su tremenda riqueza semántica. Eso ha convertido a Scheme en un vehículo idóneo para conducir investigación en el área de lenguajes de programación, tanto en aspectos netamente teóricos como en otros más pragmáticos. Por ejemplo, Eugene Kohlbecker escribió su tesis doctoral sobre la expansión higiénica de macros [25], que es un sofisticado mecanismo que permite extender de manera confiable un lenguaje estructurado, y para su estudio usó Scheme como modelo base.

Las extensiones de Scheme tales como los motores mencionados anteriormente vuelven a este lenguaje apto para aplicaciones del área de sistemas operativos, y en varias escuelas en los Estados Unidos se le ha usado de esa forma. Asimismo, existen versiones del lenguaje que se utilizan ampliamente para la programación de robots móbiles (por ejemplo, Scheme 48).

Por otro lado, Scheme ha sido utilizado durante varios años como primer lenguaje de programación para estudiantes de licenciaturas en ciencias de la computación (normalmente usando como texto el libro de Abelson y Sussman [12]) en varias universidades de los Estados Unidos.

El argumento principal para utilizar Scheme en la enseñanza es que su diseño y simpleza permiten que un alumno pueda manejar con destreza sus conceptos básicos en sólo un semestre. Una de las experiencias más interesantes para un alumno con conocimientos de programación que aprende Scheme, es que lo que tiene que aprender sobre sintaxis del lenguaje es mínimo (básicamente tiene que entender dónde colocar los paréntesis necesarios), mientras que en un curso de C o Pascal la mayor parte del semestre se dedica a memorizar aspectos sintácticos del lenguaje sin enfocarse realmente en cómo abstraer ideas para elaborar un programa. Eso produce la impresión errónea de que la programación involucra mucha memorización, cuando en realidad lo que se debiera requerir es simplemente mucha creatividad.

Adicionalmente, algunos profesores han propuesto su uso como primer lenguaje de programación para el público en general, de entre los que destacan los esfuerzos de Harvey y Wright de la Universidad de California en Berkeley, que desarrollaron extensiones de Scheme que hacen al lenguaje todavía más accesible y fácil de utilizar (asemejándolo un poco a Logo). Su texto [26] resume su curso de introducción a la programación, e incluye el código necesario para incorporar las extensiones antes mencionadas.

Para quien cree que el texto de Abelson y Sussman es demasiado denso, por su enfoque matemático típico de MIT, entonces es sumamente recomendable el peculiar libro titulado "The Little LISPer" [27], que es ampliamente utilizado en los Estados Unidos como texto introductorio en cursos de Lisp. Sin embargo, para un curso formal de programación en Scheme, es más recomendable el texto de Springer y Friedman [28], que cubre prácticamente todos los aspectos del lenguaje, e incluye suficiente código y ejercicios como para mantener un curso de al menos un semestre.

Scheme fue diseñado de tal forma que resultase fácil incorporar diferentes paradigmas existentes y nuevos en materia de lenguajes de programación. Su manejo de funciones de orden superior (funciones que toman como argumentos otras funciones) y sus poderosas primitivas para manejo de listas lo ponen en la lista de lenguajes funcionales apropiados para la enseñanza de este paradigma. Por otro lado, su conjunto de formas especiales para control y secuenciamiento, tales como **do**, **begin**, **if** y **cond**, lo vuelven una herramienta suficientemente flexible como para enseñar programación imperativa. Finalmente, su concepto tan general de objeto, su conjunto de

métodos (operaciones que se pueden ejecutar sobre un objeto) y sus mecanismos para paso de mensajes, lo hacen un candidato idóneo para la enseñanza de programación orientada a objetos.

Como información adicional, cabe mencionar que así como Lisp inspiró el desarrollo de una computadora simbólica derivada de un intérprete abstracto del lenguaje, Robert G. Burger hizo lo propio con Scheme [29].

Conclusiones

Scheme es un lenguaje simple, pero poderoso; pequeño, pero flexible. Su naturaleza lo hace ideal para la enseñanza, incluso como primer lenguaje de programación, por su tremenda facilidad para incorporar diversos paradigmas con sólo un puñado de primitivas. Como lenguaje funcional, es más accesible que el mismo Lisp, y menos intrincado. Como lenguaje imperativo, conserva la pureza y elegancia que le proporcionan la recursividad y las funciones de orden superior, superando así a muchos de los lenguajes que gozan de gran popularidad hoy en día, tales como Pascal y C. Como lenguaje orientado a objetos, hace alarde de un sistema de paso de mensajes y de manipulación de objetos equiparable únicamente al de Smalltalk. Su mecanismo de continuaciones proporciona procedimientos de escape como ciudadanos de primer orden, condenando a la obsolescencia a los mecanismos similares con que cuentan otros lenguajes. El uso de *streams* permite implementar la evaluación concisa, convirtiendo al lenguaje en una herramienta idónea para cursos avanzados de lenguajes de programación. Una de sus pocas desventajas estriba en su carencia de tipos, aunque existen versiones de Scheme que incorporan un sistema similar al de ML (por ejemplo, Scheme 48).

Desde una perspectiva más pragmática, podemos decir que su sintaxis es extremadamente simple, lo que permite que el lenguaje pueda dominarse fácilmente en sólo un semestre. Los estudiantes suelen preferirlo, sobre todo, cuando se trata del primer lenguaje de programación que aprenden, y los instructores lo elogian porque facilita la enseñanza de ideas de abstracción y diseño de algoritmos, tan útiles para formar buenos programadores. Con la sabia filosofía de proporcionar sólo una primitiva que haga lo que queremos en vez de varias, como en otros lenguajes, Scheme se erige en la cima de los lenguajes preferidos por las nuevas generaciones, ya no sólo como una mera curiosidad científica, como se vió a Lisp en otra época, sino como una herramienta efectiva que sirve para aprender a programar y a resolver problemas del mundo real.

Referencias

[1] Backus, John, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", en *Communications of the ACM*, agosto 1978, Vol. 21, No. 8, pp. 613-641.

[2] Church, Alonzo, "The Calculi of Lambda Conversion", en *Annals of Mathematical Studies*, No. 6, Princeton University Press, Princeton, N. J. 1941.

[3] McCarthy, John, "History of Lisp", en *SIGPLAN Notices*, Vol. 13, 1978, pp. 217-223.

[4] Hewitt, Carl, "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot", *PhD Thesis*, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1972, MIT Artificial Intelligence Laboratory TR-258.

[5] Sussman, Gerald Jay & Steele Jr., Guy Lewis, "Scheme: an interpreter for extended lambda calculus", *MIT Artificial Intelligence Memo 349*, Diciembre 1975.

[6] Steele Jr., Guy Lewis & Sussman, Gerald Jay, "LAMBDA: The Ultimate Imperative", *AI Memo 353*, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, March 1976.

[7] Steele Jr., Guy Lewis, "LAMBDA: The Ultimate Declarative", *AI Memo 379*, MIT Artificial Intelligece Laboratory, Cambridge, Massachusetts, November 1976.

[8] Steele Jr., Guy Lewis, "RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization)", *Technical Report 474*, MIT Artificial Intelligence Laboratory, May 1978.

[9] Steele Jr., Guy Lewis & Sussman, Gerald Jay, "The revised report on Scheme, a dialect of Lisp", *MIT Artificial Intelligence Memo 452*, Enero 1978.

[10] Steele Jr., Guy Lewis & Gabriel, Richard P., "The Evolution of Lisp", en *ACM SIGPLAN NOTICES*, Volume 28, Number 3, pp. 231-270, March 1993.

[11] Fessenden, Carol; Clinger, William; Friedman, Daniel P. & Haynes, Christopher, "Scheme 311 Version 4. Reference Manual", *Technical Report 137*, Indiana University, February 1983.

[12] Abelson, Harold & Sussman, Gerald Jay with Sussman, Julie, "Structure and Interpretation of Computer Programs", MIT Press, Cambridge, Massachusetts, 1985.

[13] Clinger, William (Editor), "The revised report on Scheme, or an uncommon Lisp", *MIT Artificial Intelligence Memo 848*, August 1985.

[14] Rees, Jonathan & Clinger, William (Editors), "The revised^3 report on the algorithmic language Scheme", en *ACM SIGPLAN Notices* 21(12), pp. 37-79, December 1986.

[15] Clinger, Willian & Rees, Jonathan, "The revised^4 report on the algorithmic language Scheme", November, 1991.

[16] Rees, Jonathan, "The Scheme of Things: The June 1992 Meeting", en *Lisp Pointers*, Volume V(4), October-December, 1992.

[17] Texas Instruments, "PC Scheme. User´s Guide & Language Reference Manual", Trade Edition, The MIT Press, Cambridge, Massachusetts, 1990.

[18] Eisenberg, Michael, "Programming in Scheme", The MIT Press, Cambridge, Massachusetts, 1990.

[19] Milner, Robin; Tofte, Mads & Harper, Robert, "The Definition of Standard ML", The MIT Press, Cambridge, Massachusetts, 1990.

[20] Clinger, William, "Semantics of Scheme", en *BYTE*, February, pp. 221-227, 1988.

[21] Haynes, Christopher; Friedman, Daniel & Kohlbecker, Eugene, "Programming with continuations", Indiana University Computer Science Department, *Technical Report No. 151*, November, 1983.

[22] Wand, Mitchell, "Continuation-based Multiprocessing", en *Conference Record of the 1980 LISP Conference*, August, pp. 19-28, 1980.

[23] Haynes, Christopher; Friedman, Daniel & Wand, Mitchell, "Continuations and Coroutines", en *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, August, pp. 293-298, 1984.

[24] Haynes, Christopher & Friedman, Daniel, "An Abstraction of Timed Preemption", Indiana University Computer Science Department, *Technical Report No. 178*, August, 1985.

[25] Kohlbecker Jr., Eugene Edmund, "Syntactic Extensions in the Programming Language Lisp", *PhD Thesis*, Indiana University, August, 1986.

[26] Harvey, Brian & Wright, Matthew, "Simply Scheme: Introducing Computer Science", The MIT Press, 1994.

[27] Friedman, Daniel & Felleisen, Matthias, "The Little LISPer", Third Edition, Macmillan Publishing Company, 1989.

[28] Springer, George & Friedman, Daniel, "Scheme and the Art of Programming", The MIT Press, 1990.

[29] Burger, Robert, "The Scheme Machine", Indiana University Computer Science Department, *Technical Report # 413*, August, 1994.