Estructura e Interpretacin de programas computaciones

Segunda Edicin
Formato Texinfo extraoficial 2.neilvandyke4

Harold Abelson y Gerald Jay Sussman
con colaboracin de Julie Sussman
prefacio por Alan J. Perlis

Estructura e Interpretacin de programas computaciones
segunda edicin

Harold Abelson y Gerald Jay Sussman
con colaboracin de Julie Sussman

prefacio por Alan J. Perlis

The MIT Press
Cambridge, Massachusetts
Londres, Inglaterra

McGraw-Hill Book Company
New York, St. Louis, San Francisco
Montreal, Toronto

Este libro es parte de una serie escrita por la facultad de Ingeniera Electrica y el Departamento de Ciencias de la Computacin del Massachusetts Institute of Tecnology (Instituto Tecnolgico de Massachusetts). Editado y producido por la editorial MIT bajo contrato de produccin-distribucin junton con McGraw-Hill Company.

Formato Texinfo no oficial 2.neilvandyke4 (Junio 30,2009)

Short Contents

Table of Contents

Unofficial Texinfo Format

Esta es la segunda edicin del libro SICP, extraido desde un fichero en formato extraoficial Texinfo.

{Probabelmente estas leyendo desde un browser como el modo Info de Emacs. Alternativamente podrias leerlo en tu pantalla o impreso en papel.

UTF es ms facil de buscar que HTML. Tambien es ms accesible a personas que tengan computadoras de gama baja, como equipos 386 donados. Un 386 puede, en teora, correr Linux, Emacs y un interprete de Scheme simultaneamente, aun cuando la mayoria de ellas probablemente no pueda correr al mismo tiempo Netscape y X window sin introducir a los jovenes hackers en el concepto de {thrashing. En UTF cabe perfectamente en un floppy de 1,44MB, el cual viene bien cuando se tiene un ordenador sin acceso a INTERNET.

Se cree que el libro extraoficial en formato Texinfo es mantenido con el espiritu de la versin de distribucin libre en HTML. Aunque nunca se sabe cuando la armada de abogados necesitar algo que hacer, asi que piensalo dos veces antes de usar tu nombre completo.

Peath,

Lytha Ayth

Agregado: Vea tambien las clases SICP por Abelson y Sussman: `http://www.swiss.ai.mit.edu/class`

Dedication

Este libro est dedicado, con respeto y admiracin, al espritu que vive en la computadora.

> “Creo que es extraordinariamente importante que nosotros en las Ciencias de la Computacin, sigamos manteniendo la diversin. Cuando empez, habia mucha diversin. Claro que, de vez en cuando los clientes se quejaba y despues de un tiempo los empezamos a tomar en serio. Comenzamos a sentirnos realmente responsables por el exito y perfeccin en el uso de esas maquinas. No creo que lo seammos. Creo que somos responsables por extenderlas, llevarlas por nuevas direcciones, y de mantener la divercin en casa. Espero que el campo de las Ciencias de la Computacin nunca pierda su sentido del humor. Sobre todo, espero que nunca nos convertiremos en misionarios. No te sientas como si fueras un vendedor de biblias. El mundo ya tiene demasiados. Lo que tu conoces sobre computacin otras personas lo sabrn. No sientas como si la clave para el xito de la computacin est solo en tu manos. Lo que s est en tus manos, creo y espero, es inteligencia: la habilidad de ver que tu fuiste capaz de hacerla ms de lo que te llego a las manos.”
>
> —Alan J. Perlis (Abril 1, 1922 Febrero 7, 1990)

Foreword

Educadores, generales, dietistas, psiclogos, y padres, programan. Ejrcitos, estudiantes, y angulas sociedades son programadas. Un enfrentamiento con los grandes problemas requieren una sucesion de programas, muchos de los cuales Estos programas aparecen con problemas que parecen ser particulares nicamente a los problemas que abordan. Para apreciar la programacin como una actividad intelectual por s misma uno mismo debe sumergirce en la programacin; tienes que leer y escribir los programas – muchos de ellos. No importa mucho de que trata el programa o que aplicaciones tiene. Lo que realmente importa es que tambin lo realiza o cuan bien encaja con otros programas para crear programas incluso ms grandes. El programador debe buscar a la vez perfeccin de las partes y de el conjunto. En este libro el uso de "programa" est enfocado en la creacin, ejecucin, y estudio de programas escritos en un dialecto de lisp para su ejecucin en computadoras digitales. Usando Lisp nos restringimos o limitamos no en que podemos programar, si no nicamente la notacin para la descripcin de nuestros programas.

Nuestra jornada en los temas que trataremos en este libro nos llevar con Los temas que trataremos en ste libro nos envolver principalmente con tres tipos de fenomenos: la mente humana, conjuntos de programas computacionales, y la computadora.

Cada programa computacional es un modelo, trazado en la mente, de un proceso real o mental. Estos procesos, surgen desde los pensamientos y la experiecia humana, son numricamente son inmensos, intrincados en detalle, y al mismo tiempo parcialmente comprendidos.

Ellos estan modelados a nuesta permanente satisfaccin la cual es raramente lograda por nuestros programas.

As que incluso cuando nuestros programas son colecciones descretos de smbolos cuidadosamente colocados como mosaicos de funciones interconectadas, continuamente evolucionan: los cambiamos a la par de nuestra percepcin del modelo se profundiza, crece y generaliza hasta que al final el modelo alcanza un lugar estable dentro de otro modelo con el cual tratamos.

El origen del regocijo que encontramos en las programas computacionales es el continuo crecimiento en la menta y de los mecanismos de la computadora expresados como programas y la explocin de percepciones que ellos generan. Si el arte es un interprete de nuestros sueos, la computadora los ejecuta en la forma de programas!

Con todo su poder, la computadora no es nada amistosa. Sus programas deben correctos, y que todo lo que deseamos decir debe ser dicho con todo detalle. Como toda otra actividad simblica, nos convencemos de la correctes en los programas mediante la argumentacin.

Lisp itself can be assigned a semantics (another model, by the way), and if a program's function can be specified, say, in the predicate calculus, the proof methods of logic can be used to make an acceptable correctness argument.

Desafortunadamente, conforme los programas crecen y se complican, como casi siempre sucede, la adecuacin, consistencia, y correctes de la especificacin misma se vuelve sujeto de duda, dado que argumentos formales sobre la correctes de los programas rara vez los acompaan.

Dado que los programas grandes crecen apartir de los pequeos, es crucial que desarrollemos un arsenal estructuras estandar de aquellas programas que sabemos son correctos

–nosotros los llamamos lenguajes– y aprender a combinarlos en estructuras mayores usando tecnicas de organizacin ha probado ser de gran valor.

Estas tcnicas son tratadas a lo largo de ste libro, y entenderlas, es escencial en la prometedora empresa llamada programacin. Ms que otra cosa, el descubrimiento y dominio de poderosas tcnicas de organizacin aceleran nuestra habilidad para crear programas grandes y complejos. Inversamente, ya que escribir grandes programas es tan agotador, estamos empujados a inventar nuevos metodos para reducir el conjunto de funciones y pormenores a ser puestos en los programas.

A diferencia de los programas, los computadores deben obedecer las leyes de la fsica. Si se desea que ellos actuen ms rapidamente – menos nanosegundos por cambio de estado — deben transmitir electrones a menores distancias. Para lograrlo se necesitara eliminar el calor generado por el gran nmero de dispositivos juntos en tan pequeo espacio.

Se ha desarrollado un esquisito arte para balancear la multitud de funciones y la densidad de los dispositivos. En cualquier momento, el hardware siempre opera a un nivel ms primitivo que aquel en que se encuentran los programas. Los procesos que transforman nuestros programas lisp en programas "maquina" son por si mismos modelos abstractos en que programamos. Su estudio y creacion dan una gran cantidad de conocimiento en la organizacin de estos modelos abstractos en los cuales programamos.

Por supuesto que la computadora por si misma puede ser modelada. Pienselo: el comportamiento de pequeas piezas electrnicas estn modeladas por la mecnica cuntica descrita por ecuaciones diferenciales cuyo comportamiento es capturado detalladamente por aproximaciones representados en programas computaciones ejecutados en en computadoras compuestas de . . . !

No se trata simplemente de una cuestin de conveniencia tctica identificar separadamente cada uno de las tres facetas. Aunque, como algunos dicen, todo est en la cabeza, sta separacin lgica conlleva una aceleracin del trafico simblico entre cada, cuya riqueza, vitalidad, y potencial es superado en experiencia humana nicamente al transcurrir la vida misma.

En el mejor de los casos, las relaciones entre los 3 panoramas son metaestables. Las computadoras nunca son lo sufientemente grandes y rapidas. Cada avance en las tecnologas de hardware nos lleva a empresas mayores en la programcin de software, nuevos principios organizacionales, y a un enriquecimiento de modelos abstractos. Cada lector deberia preguntarce a s mismo de vez en vez, "Con que fin?, hacia que fin? " - aunque no tan seguido como para dejar perder la diversin debido a tales preguntas filosficas.

Entre los programas que escribimos, algunos (aunque nunca son suficientes) desempean alguna funcin matematica muy precisa, como buscar el mximo dentro de una secuencia de nmeros, determinar primaridad, o encontrar la raiz cuadrada. A tales programas los llamamos algoritmos, y con gran detalles son conocidos por su comportamiento optimo, particularmente con respecto de dos parmetros: tiempo de ejecucin y requerimientos del espacio de almacenamiento. Un programador debe adquirir buenos algoritmos e idiomas.

Incluso aunque algunos programas se resistan a una especificacin precisa, es la responsabilidad del programador estimar, y siempre tratar de mejorar, su desempeo.

Lisp es un sobreviviente, habiendo estado en uso cerca de un cuarto de siglo. Entre los lenguajes de programacin en actividad nicamente Fortran tiene una vida mayor. Ambos lenguajes han cubierto las necesidades de programacin en importantes areas de trabajo, por ejemplo Fortran para computacin cienctfica y de ingeniera, Lisp para inteligencia artificial.

stas dos areas continuarn siendo importantes, y sus programadores son tan devotos a stos lenguajes que podrian seguir estando en uso por al menos otro cuarto de siglo.

Lisp cambia. Scheme es un dialecto usado en ste libro el cual a evolucionado apartir del Lisp original y difiere de ste ltimo en varias formas importantes, incluye alcance esttico para la asignacin de variables y permite a las funciones producir funciones como valores de retorno. En su estrctura semntica Scheme se encuentra tan emparentado con Algol 60 como lo estuvieron las primeras versiones de Lisp. Algo 60, nunca volvio a ser un lenguaje activo, sin embargo vive en los genes de Scheme y Pascal. Sera dificil encontrar dos lenguajes que sean la moneda de intercambio de dos culturas tan diferentes que las reunidas alrededor de stos lenguajes. Podramos ver a Pascal como un lenguaje para crear piramides –imponentes, asombrosas, estructuras estticas creadas por ejrcitos que apilan pesados bloques en su lugar. Lisp es para crear organismos –majestuosos, impresionantes, estructuas dinmicas creadas por escuadrones llenando llenando — de organismos ms simples en su lugar. Los principios organizacionales usados son los mismos en ambos casos, excepto por una importante y extraordinaria diferencia: La libertad de exportar funcionalidades confiada al programador en Lisp, es ms de un orden de magnitud mayor que aquel encontrado en los proyectos con pascal. Los programas lisp inflan las librerias con funciones cuya utilidad traciende a la aplicacin que las origin.. Las listas, estructuras de datos nativas de Lisp, es mayormente responsable por dicho crecimiento de utilidad. La aplicabilidad natural de las listas se reflejan en funciones que son sorprendentemente no idiosincrticas. En Pascal la gran cantidad de declaraciones de estructuras de datos inducen a una especializacin por parte de las funciones que las procesan, implicando una inhibicine e incluso penalizacin a la coperacin casual. Es mucho mejor tener 100 funciones que operen sobre una estructura de datos, que tener 10 funciones que operen sobre 10 estructuras de datos. Como resultado tenemos que las pirmides deben permanecer por milenios; los organismos deben evolucionar o morir.

Para ilustrar la diferencia, compara el tratamiento de materiales y ejercicios dentro de este libro con cualquiera de mis primeros cursos usando Pascal. No te dejes llevar por la ilucin de que ste libro es digerible nicamente en el MIT, ste es precisamente lo que un libro serio en programacin en Lisp debe ser, sin importar de donde o quien sea el estudiante.

Nota que este es un libro sobre programcin en general, a diferencia de la mayora de los libros sobre Lisp, las cuales son usados como preparacin para trabajar en inteligencia artificial. Despues de todo, las preocupaciones crticas sobre ingeniera de software e inteligencia artificial tienden a solaparce a medida que el sistema bajo investigacin se vuelve ms grande.

sto explica por que existe un crecimiento del interes en Lisp fuera del campo de inteligencia artificial.

Por sus objetivos, uno podra esperar, que la investigacin en inteligencia artifial genere problemas relevantes en programacin. En otras culturas de programacin sta avalancha de problemas se refleja en la generacin de nuevos lenguajes. De hecho, en cualquier gran proyecto de programacin, principio util de organizacin es controlar e isolar el trfico dentro de las mdulos de las tareas con la invencin de nuevos lenguajes. stos lenguajes tienden a ser menos primitivos a medida que uno se aproxima a los limites del sistema en que nosotros, los humanos, interactuamos ms seguido. Como resultado, tales sistemas contienen replicadas muchas veces,funciones complejas para el tratamiento del lenguaje. Debido a la sintaxis y semntica tan simple de Lisp la tarea de analizarlo resulta una tarea elemental. Debido a lo anterior la tarea del analisis sintctico (parsing) casi no juega ningun rol en la programacion en Lisp, y la construccion de procesadores de lenguaje raramente es un impedimento para

mantener el ritmo de crecimiento y cambios de los grandes sistemas Lisp. Finalmente, sta simplicidad de la sintaxis y semantica es la responsable de las preocupaciones y libertades para todos los programadores en Lisp. Ningun programa en Lisp mayor a unas cuantas lineas puede ser escrito sin ser saturado con funciones descrecionales. Inventa y adapta; ten adaptaciones y reinventa! Brindamos por los programadores que escriben sus pensamientos dentro de parentesis anidados!

Alan J. Perlis
New Haven, Connecticut

Preface to the Second Edition

> Ser posible que no haya nada parecido al software, que est destinado a ser descartado: que siempre tendremos que verlo como una burbuja de jabn? — Alan J. Perlis

El material en presente en ste libro han sido los fundamentos los temas de introduccin en el MIT desde 1980. Hemos estado enseando ste material por cuadro aos cuando la primera edicin fue publicada, y doce aos ms han transcurrido desde la aparicin de sta segunda edicin. stamos contentos de que nuestro trabajo haya sido adoptado e incorporado en otros textos. Hemos visto a nuestros estudiantes tomar las ideas y programas en este libro y crearlos en el ncleo de nuevos sistemas y lenguajes de cmputo. En la realizacin de un antiguo juego de palabras, nuestros estudiantes se han convertido en nuestros constructores. Somos afortunados de tener tan capacez estudiantes y tan consumados constructores.

En la preparacin de sta edicin, hemos incorporado cientos de aclaraciones sugeridas por nuestra propia experiencia en la enseanza y los comentos de colegas en el MIT y en otros lados. Hemos redisado la mayoria de los sistemas de programacin principales en ste libro, incluyendo los sistemas de aritmtica genrica; y hemos reescrito todos los programas de ejemplo para asegurar que cualquier implementacin que cumpla IEE Scheme estndar (!acronymIEEE 1990) podr ejecutar el cdigo.

sta edicin pone nfasis en varios nuevos temas. El ms importante es el rol central que jugado por las diferentes formas de lidias con tiempo y modelos computacionales:l objetos con estado, programacin concurrente, programacin funcional, evaluacin perezosa, y programacin no determinstica. Incluimos varias secciones nuevas sobre concurencia y no determinismo, tratando de integrar stos temas por todo el libro.

La primera edicin de ste libro sigui cuidadosamente el temario de un semestre en el MIT. Con todo el material nuevo en sta segunda edicin, no ser posible curbir todo en nicamente un semestre, por tanto el maestro tendr que escoger que tomar y que no. En nuestra propia experiencia de enseansa, a veces nos saltamos la seccin sobre programacin lgica (section ⟨undefined⟩ [4.4], page ⟨undefined⟩), tenemos estudiantes usando el simulador de la maquina de registros pero no cubrimos su implementacin (section ⟨undefined⟩ [5-2], page ⟨undefined⟩), nicamente damos una rpida vista al tema sobre la construccin del compiladores (⟨undefined⟩ [5-5], page ⟨undefined⟩). Incluso con sto, todavia es un curso intensivo. Algunos instructores tal vez quisieran nicamente cubrir los primeros tres o cuatro captulos, dejando el resto del material para cursos posteriores.

El sitio Web `http://www-mitpress.mit.edu/sicp/` provee soporte para usuarios de ste libro. ste incluye programas del libro, materiales suplementarios e implementaciones ejecutables del dialecto de Lisp, Scheme.

Preface to the First Edition

> Una computadora es como un violin. Puedes imaginar a un novato tratando primero uno fongrafo y despues un violin.El ltimo, dice, suena terrible. ste es el argumento que escuchamos de nuestros humanistas, y la mayora de nuestros cientficos computacionales. Los programas computacionales son buenos, dicen, para propsitos particulares, sin embargo no son flexibles. Tampoco el violin, o una mquina de escribir, hasta que se aprenda a usarlo.
>
> —Marvin Minsky, "Por que la programacin es un buen medio para expresar ideas pobremente comprendidas o no muy bien formuladass"

"La estructura e interpretacin de programas computacionales" es el material de iniciacin en ciencias de la computacin en el Instituto Tecnolgico de Massachusetts. Es requerido que todos los estudiantes de ingenira electrica o ciencias de la computacin tomen ste curso, se concidera como un cuarto del "ncleo curricular comn', el cual tambien incluye temas sobre circuitos y sistemas lineales, adems otro sobre diseo de sistemas digitales. Hemos estado involucrados en el desarrollo de ste asunto desde 1978, y hemos estado enseando ste material tal como se encuenta ahora desde el otoo del 1980 con entre 600 y 700 estudiantes por ao. La mayoria de estos estudiantes han tenido poco o ninguna enseanza formal a priori en computacin., aunque muchos de ellos haya jugado con computadoras un poco y menos han tenido mucha experiencia programando o diseando hardware.

El diseo de este tema de introduccin a las ciencias de la computacin reflaja dos grandes preocupaciones. Primero, que los lenguajes de programacin no son solo una forma de lograr que una computadora realice una tarea, si no que es en realidad una novedosa forma de expresar ideas sobre metodologa. Por tanto, programas deben ser escritos para ser leidos por personas, e incidentalmente para ser ejecutadas por mquinas. Segundo, creemos que el material expuesto por ste tema en ste nivel, no es la sintaxis de las contrucciones sintacticas de un lenguaje de programacin en particular, ni ingeniosos algoritmos para calcular alguna funcion en particular eficientemente, ms bien las tcnicas usadas para controlar la complejidad intelectual de los grandes sistemas de software.

Nuestro objetivo es que los estudiantes que completen ste tema deban tener un buen sentido del gusto del estilo y esttica de la programacin. Ellos deben tener dominio sobre las tcnicas de control de complejidad mayores. Ellos deben ser capaces de leer un programa de 50 hojas de longitud, if est escrito en un estilo ejemplar. Ellos deben saber que no necesitan leer, y que no necesitan entender en cualquier momento. Deben sentirce seguros al modificar un programa, reteniendo el espritu y estilo del autor original.

stas habilidades no son nicas de la programacin de computadoras. Las tcnicas que enseamos son comunes a todo el diseo de ingeniera. Contralamos la complejidad, creando abstracciones que oculten detalles cuando sea apropiado. Controlamos complejidad, estableciendo interfaces que nos permiten construir sistemas, combinando piezas estandar, bien documentadas en una forma "mescla y encaja". Controlamos complejidad al establecer nuevos lenguajes para describir un diseo, cada uno de los cuales enfatiza un particular aspecto del diseo y desenfatiza otros.

Lo que hay por debajo de nuestra aproximacin a ste tema es nuestra conviccin de que "las ciencias de la computacin" no es una ciencia y que su signifado tiene poco que ver con las computadoras. La revolucin informtica, es una revolucin en la manera en que pensamos y

en la forma que expresamos lo que pensamos. La escencia de ste cambio es lo que podra ser llamado *epistemologˆˆeda procedural*—el estudio de la estructura del conociemiento desde un punto de vista imperativo, en oposicin al punto de vista declarativo tomado por las ramas clasicas de las matemticas. Las matemticas proveen un entorno para tratar precisamente con la nocin de "que es" , la computacin provee un entorno para tratar precisamente con la nocin de "como hacerlo."

Enseando nuestro material usamos un dialecto del leguaje de programacin Lisp. Formalmente nunca enseamos el lenguaje, debido a que no necesitamos hacerlo. Solo lo usamos, y los estudiantes lo aprenden luego de unos pocos dias. sta es una de las grandes ventajas de los lenguajes parecidos a Lisp: Tienen muy pocas formas de componer expresiones, y casi ninguna estructura sintctica. Todas las propiedades formales pueden ser cubiertas en una hora, al igual que las reglas de ajedrez. Despues de un corto tiempo nos olvidamos de los detalles de la sintaxis del lenguaje (por que no tiene) y nos topamos con lo que realmente importa —idear que queremos computar, como dividiremos el problema en partes manejables, y como trabajaremos sobre esas partes. Otra ventaja de Lisp es que soporta (aunque no fuerza) ms de una estrategia para descomponer programas de gran escala estrategias para la descomposicin modular de sistemas que cualquier otro lenguaje de programacin que conoscamos soporta. Podemos crear abstraciones de datos y procedurales, podemos usar funciones de orden alto para capturar patrones de uso comun, podemos modelar estado local usando asignacion y mutacin de datos, podemos enlazar partes de un programa con streams y evaluacin retrasada, y podemos implementar facilmente lenguajes empotrados. Todo sto est embebido en un entorno interactivo con excelente soporte para el diseo incremental de programas, su contruccin, prueba y revisin. Agradecemos a todas las generaciones de magos de Lisp, empezando con John McCarthy, quien puso de moda una estupenda herramienta con poder y elegancia sin precedentes.

El dialecto de Lisp que usamos es Scheme, en un intento de lograr juntar el poder y elegancia de Lisp y Algol. De Lisp tomamos el poder metalingustico que deriva de su sintaxis tan simple, la representacin uniforme de programas como objetos de datos, y el colector de basura y colector de datos alojados. De Algol tomamos el alcance lxico y la estructura de bloques, los cuales son legados de los pioneros el diseo de los lenguajes de programacin quienes estuvieron en el comit de Algol. Scheme, the dialect of Lisp that we use, is an attempt to bring together the power and elegance of Lisp and Algol. From Lisp we take the metalinguistic power that derives from the simple syntax, the uniform representation of programs as data objects, and the garbage-collected heap-allocated data. From Algol we take lexical scoping and block structure, which are gifts from the pioneers of programming-language design who were on the Algol committee. We wish to cite John Reynolds and Peter Landin for their insights into the relationship of Church's *[lambda]* calculus to the structure of programming languages. We also recognize our debt to the mathematicians who scouted out this territory decades before computers appeared on the scene. These pioneers include Alonzo Church, Barkley Rosser, Stephen Kleene, and Haskell Curry.

Acknowledgements

Nos gustara agradecer a todas las personas que nos apoyaron a desarrollar ste libro y su curricula.

Nuestra materia es un claro descendiente de “6.231,” un materia sobre ligustica de programacin y el clculo *lambda* enseado en el MIT a finales de 1960 por Jack Wozencraft y Arthur Evans, Jr.

Tenemos una gran deuda con Robert Fano, quien reorganiz la curricula introductoria en Ingeniera Electrica y Ciencias de la Computacin del MIT para enfatizar los principios de la ingeniria de diseo. El nos gui para que empezaramos sta empresa y escribi el primer conjunto de notas de los temas de los cuales se desarroll ste libro.

Mucho del estilo y esttica de la programacin que tratamos de ensear fue desarrollada en conjunto con Guy Lewis Steele Jr. , quien colabor con Gerald Jay Sussman en el desarrollo inicial del lenguaje de programacin Scheme. Adicionalmente, David Turner, Peter Henderson, Dan Friedman, David Wise, y Will Clinger nos han enseado muchas de las tcnicas de la programacin funcional de la comunidad que aparece en ste libro.

Joel Moses nos enseo como estructurar grandes sistemas. Sus experiencia con el sistema Macsyma para computacin simblica nos enseo que uno debe evitar la complejidad en el control y concentrarse en organizar los datos para que reflejen la verdadera estructura del mundo que est siendo modelado.

Marvin Minsky y Seymour Papert formaron muchas de las actitudes sobre la programacin y su lugar en nuestras vidas intelectuales. A ellos les debemos el saber que la computacin provee medios de expresin para explorar ideas de otro modo sera muy complejo de abordar prescisamente. Ellos enfantizan que la habilidad de un estudiante para escribir y modificar programas provee un poderoso medio en el cual explorar se combierte en una actividad natural.

Estamos fuertemente de acuerdo con Alan Perlies que la programacin es muy divertida y que mejor hemos de tener cuidado de apoyar la diversin de la programacin. Parte de que sea divertido se deriva de observar a los grandes maestros trabajando. Somos afortunados de haber sido aprendices de programacino a los pies de Bill Gosper y Richard Greenblat.

Es dificil identificar todas las peronas quienes han contribuido al desarrollo de nuestro curriculum. Agradecemos a los profesores, instructores, y tutores quienes han trabajado con nosotros los pasados quince aos y puesto tantas horas extra en nuestra materia, especialmente Bill Siebert, Albert Meyer, Joe Stoy, Randy Davis, Louis Braida, Eric Grimson, Rod Brooks, Lynn Stein, y Peter Szolovits. Nos gustara agradecer especialmente la sobresaliente contribucin a la enseansa de Franklyn Turbak, ahora en Wellesley; su trabajo en la instruccin de pregrado ha puesto un estandar al cual todos podemos aspirar. Estamos agradecidos a Jerry Saltzer y Jimm Miller por ayudarnos a lidiar con los misterios de la concurrencia, y a Peter Szolovits y David McAllester por sus contribuciones en la exposicin de la evaluacin no determinstica en ⟨undefined⟩ [Chapter 4], page ⟨undefined⟩.

Muchas personas han puesto un esfuerzo significativo para presentar ste material en otras universidades. Alguanas de las personas con que hemos trabajado muy cercanamente son Jacob Katzenelson en el Tecnion, Hardy Mayers en la Universidad de California en Irvine, Joe Stoy en Oxford, Elisha Sacks en Purdue, y Jan Komorowski en la Universidad Norwegian de Ciencia y Tecnologa. Estamos excepcionalmente orgullosos de ste tema en

otras universidades, incluyhendo Kenneth Yip en Yale, Brian Harvey en la Universidad de California en Berkeley, y Dan Huttenlocher en Cornell.

Al Moyé organizo para nosotros la enseanza de ste material a los ingenieros de Hellet-Packard, as como la produccin de cintas de video de esas clases. Nos gustara agradecer a los talentosos instructores, en particular a Jim Miller, Bill Siebert, y Mike Eisenberg, quien han diseado los cursos de educacin continua incorporando stas cintas y enseandolas en universidades e industrias alrededor de todo el mundo.

Muchos educadores en otros paises han puesto un significativo esfuerzo traduciendo la primera edicin. Michel Briand, Perre Chamard, y Andr Pic produjeron una edicin en frances; Sussane Daniels-Herold produjeron una edicin en Aleman; y Fumio Motoyoshi produjeron una edicin en japones. No sabemos quien produjo la edicin en chino, pero consideramos un honor, haber sido seleccionados como el tema una traduccin "no autorizada".

Es dificil enumerar todas las personas quen han hecho contribuciones tcnicas al desarrollo de los sistemas Scheme que usamos para propsitos de enseaza. Adems de Guy Steele, entre los principales magos se han incluido Cris Hanson, Joe Bowbeer, Jim Miller, Guillermo Rozas y Stephen Adams. Otros quienes han puesto un tiempo significativo son Richard Stallman, Alan Bawden, Kent Pitman , Jon Taft, Neil Mayle, John Lamping, Gwyn Osnos, Tracy Larrabee, George Carrette, Soma Chaudhuri, Bill Chiarchiaro, Steven Kirsch, Leigh Klotz, Wayne Noss, Todd Cass, Patrick O'Donnell, Kevin Theobald, Daniel Weise, Kenneth Sinclair, Anthony Courtemanche, Henry M. Wu, Andrew Berlin, and Ruth Shyu.

Nos gustara agradecer, no solo por la implementacin del MIT, a todas las personas quienes trabajaron en el estandard IEEE de Scheme, incluyendo a William Clinger y a Jonathan Rees, quien edit el R^4RS, y a Chris Haynes, David Bartley, Chris Hanson, y Jim Miller, quienes prepararon el estandar IEEE.

Dan Friedman ha sido un viejo lider de la comunidad de Scheme. La mayora del trabajo de la comunidad va desde los problemas en el diseo del lenguaje hasta abarcar incluso inovaciones significativas en el ambito educacional, tal como las las curriculas de bachillerato basadas en el EdScheme de Schemer's Inc., y los maravillosos libros de Mike Eisenberg ,Brian Harve y Matthew Wright.

Apreciamos el trabajo de aquellos quienes contribuyeron a que este libro se hiciera realidad, especialmente a Terry Ehling, Larry Cohen, y Paul Bethge en MIT Press. Ella Mazel encontro la maravillosa imagen de la portada. Estamos particularmente agradecidos a Bernard y Ella Mazel por su ayuda con el diseo del libro en su segunda edicin, tambien al extraordinario mago del TEX David Jones. Tambien estamos en deuda a todos aquellos lectores quienes hicieron perspicaces comentarios en el nuevo borrados: Jacob Katzenelson, Hardy Mayer, Jim Miller, y especialmente Brian Harvey, quien hizo en ste libro lo que Jolie sobre su libro *Simply Scheme*.

Finalmente, nos gustaria reconocer la ayuda de las organizaciones que han apoyado ste trabajo al pasar de los aos, por parte de Hewlett-Packard: Ira Goldstein y Joel Birnbaum, por parte de DARPA: Bob Kahn.