Prólogo

> *Va contra la semilla de la educación moderna enseñar a los niños a programar. Qué diversión hay en hacer planes, adquirir disciplina en organizar pensamientos, dedicar atención al detalle y aprender a ser auto-crítico*
>
> –Alan Perlis, *Epigramas en Programación*

Muchas profesiones requieren alguna forma de programación de computadores. Los contadores programan hojas de cálculo y procesadores de palabra; los fotógrafos programan editores fotográficos; los músicos programan sintetizadores; y los programadores profesionales instruyen computadores llanos. Programar se ha convertido en una habilidad requerida.

Aunque programar es mucho más que sólo una habilidad vocacional. De hecho, *buena programación* es una actividad muy divertida, un desfogue creativo, y una forma de expresar ideas abstractas en una forma tangible. Y diseñar programas enseña una variedad de habilidades que son importantes en todas las clases de profesiones: lectura crítica, pensamiento analítico, síntesis creativa y atención al detalle.

Nosotros por tanto creemos que estudiar diseño de programas merece el mismo rol central en la educación general como las matemáticas y el castellano. O poniéndolo más sucintamente,

Todo el mundo debería aprender como diseñar programas.

Por una parte, el diseño de programas enseña las mismas habilidades de análisis crítico como las matemáticas. Pero, a diferencia de las matemáticas, trabajar con programas en una aproximación activa al aprendizaje. Interactuar con software provee retroalimentación inmediata y así permite la exploración, experimentación y auto-evaluación. Mas aún, diseñar programas produce cosas útiles y divertidas, que incrementan vastamente el sentido de cumplimiento cuando se compara a realizar ejercicios en matemáticas. Por otro lado, el diseño de programas enseña las mismas habilidades de lectura analítica y escritura como el Castellano. Incluso las más pequeñas tareas de programación son formuladas como problemas de palabras. Sin habilidades de lectura crítica, un estudiante no puede diseñar programas que correspondan a la especificación. Recíprocamente los buenos métodos de diseño de programas forzan un estudiante a articular pensamientos sobre los programas en un castellano adecuado.

La Receta de Diseño para las Funciones
Análisis del problema & Definición de Datos
Contrato, Propósito & Declaración del Efecto, Cabecera
Ejemplos
Plantilla de la Función
Definición de la Función
Pruebas

Figura *.1. Los pasos básicos de una receta de diseño de programas.

Este libro es el primer libro sobre programación como el aspecto central de una educación en artes liberales. Su foco principal es el *proceso de diseño* que lleva desde las declaraciones de los problemas a soluciones bien organizadas; desenfatiza el estudio de los detalles del lenguaje de programación, minucial algorítmica, y aplicación a dominios específicos. Nuestro deseo de enfocarnos en el proceso de diseño requiere dos innovaciones radicales para los cursos introductorios. La primera innovación es un conjunto de *líneas guía de diseño explícitas*. Los currículos existentes tienden a proveer sugerencias vagas y debilmente definidas, tales como "diseñe de arriba hacia abajo" o "haga el programa estructural". Hemos en cambio desarrollado líneas guía de diseño que llevan a los estudiantes del enunciado de un problema a una solución computacional en un modo paso a paso con productos intermedios bien definidos. En el proceso ellos aprender a leer, a analizar, a reorganizar, a experimentar, a pensar en una forma sistemática. La segunda innovación es una entorno de programación radicalmente nuevo. En el pasado, los textos sobre programación ignoraron el rol del entorno de programación en el proceso de aprendizaje; ellos simplemente asumían que los estudiantea tenían que acceder a entornos profesionales. Este libro provee un *ambiente de programación para principiantes*. También crece con los estudiantes en la medida en que ellos manetja más y más del material hasta que soporta un lenguaje con todos los adornos para el espectro completo de las tareas de programción: programación a gran escala como también *scripting*[1].

1. Se ha mantenido el termino original en inglés. El término *script* en informática se refiere a un libreto o guión usado para instruir a un computador sobre determinada labor a través de un lenguaje de programación. Sin embargo sus acepciones al castellano aún no están ampliamente difundidas y no son usadas en consenso, por lo cual se ha elegido dejar el término en inglés sin modificación. Para casos similares se obtará por la misma política. N del T.

Nuestras líneas guía son formuladas como un número de *recetas de diseño de programas*[2]. Una receta de diseño guía a un programador principiante a través del proceso entero de solucionar problemas. Con las recetas de diseño, un principiante casi nunca clava la vista de nuevo ante un pedazo de papel en blanco o una pantalla de computador en blanco. En cambio, el estudiante verificará la receta de diseño y usará las líneas guías de pregunta y respuesta para hacer algún progreso.

Creamos las recetas de diseño identificando categorías de problemas. La identificación de una categoría de problema está basada en las clases de datos que son usados para representar la información relevante. Empezando desde la estructura de esta clase de descripción, los estudiante derivan programas con una lista de verficación. La figura *.1 muestra los seis pasos básicos de una lista de verificación de una receta de diseño. Cada paso produce un producto intermedio bien definido:

1. la descripción de la clase de datos del problema;
2. la especificación informal del comportamiento de un programa;
3. la ilustración del comportamiento con ejemplos;
4. el desarrollo de una plantilla de programa o esquema;
5. la transformación de la plantilla en una definición completa; y
6. el descubrimiento de errores a través de la prueba.

La diferencia principal concierne a la relación entre los pasos 1 y 4.

Las recetas de diseño ayudan a los principiantes como a los profesores. Los profesores pueden usar las recetas para inspeccionar las habilidades de solución de problemas de un principiante, para diagnosticar debilidades y sugerir pasos remediales específicos. Después de todo, cada estado de la receta de diseño conlleva a un producto verificable bien definido. Si un principiante está barado, un profesor puede inspeccionar los productos intermedios y determinar que cuál es el problema. Basado en este análisis, el profesor puede luego proveer guía para un paso específico en la recete, encontrar las preguntas adecuadas, y recomendar ejercicios de práctica adicional.

Por qué Todo el Mundo Debería Aprender a Programar

> *Y en la medida en cuerpos de la imaginación emergen*
> *Las formas de las cosas desconocidas, y el lápiz del poeta*
> *Las convierte en figuras, y da a la nada vaporosa*
> *Una habitación local y un nombre.*
> – Shakespeare, *Sueño de una noche de verano* V(i)

Nuestro clamor de que todo el mundo programa o debería aprender a programar podría parecer extraño considerando que, en primera instancia, menos y menos gente parece programar en estos días. En cambio, la mayoría de la gente usa paquetes de aplicación, que no parecen requerir ninguna programación. Incluso los programadores usan "generadores de programas", paquetes que generan programas de, digamos, reglas de negocios. Así por qué debería cada uno aprender a programar?

La respuesta consta de dos partes. Primero, es de hecho cierto que *las formas tradicionales de programación* son útiles para sólo poca gente. Pero, programar *como los autores lo entienden* es útil para todo el mundo: la secretaria administrativa que usa hojas de cálculo como también el programador de alta tecnología. En otras palabras, tenemos una noción más amplia de programación en mente que la tradicional. Explicaremos nuestra noción en un momento. Segundo, enseñamos nuestra idea de programación basados en un principio de intrusión minimal. De aquí que nuestra noción de programación enseña habilidades de análisis de problemas y resolución de problemas *sin* imponer la sobrecarga de las notaciones y herramientas tradicionales de programación.

Para obtener una mejor compresión de la programación moderna, haga una mirada más cercana a las hojas de cálculo, una de los paquetes de aplicación populares de hoy. Un usuario entra fórmulas en una hoja de cálculo. Las fórmulas describen como una celda A depende de una celda B. Entonces, cuando el usuario entra un número en B, la hoja de cálculo automáticamente calcula los contenidos de la celda A. Para hojas de cálculo complicadas, una celda puede depender de muchas otras celdas, no sólo una.

Otra paquetes de aplicación requieren actividades similares. Considere los procesadores de palabra y las hojas de estilo. Una hoja de estilo especifica cómo crear un documento (o una parte de él) desde unas palabras u oraciones aún por ser determinadas. Cuando alguien provee palabras específicas y una hoja de estilo, el procesador de palabra crea el documento reemplazando nombres en la hoja de estilo, con palabras específicas. Similarmente, alguien quien realiza una búsqueda en la Web[3] puede desear especificar cuáles palabras buscar, cuáles palabra palabras deberían estar cerca a otras y cuáles palabras no deben están en la página. En este caso la salida depende de las páginas Web en el caché del motor de búsqueda y de la expresión de búsqueda del usuario.

---

2. Los lectores cuya experiencia está basada exclusivamente en lenguajes de programación como C/C++, Basic y Pascal deben leer "procedimiendo" o "método" donde el prefacio menciona "programa".

3. De nuevo se ha elegido la palabra original en inglés, por su frecuente uso sin ambigüedad en español. También puede usar indistintamente el término Red en lugar de Web. N del T.

Finalmente, usar un generador de programas en muchas formas subyace sobre las mismas características necesarios para los paquetes de aplicaciones. Un generador de programas crea un programa en un lenguaje de programación tradicional, tal como C++ o Java, a partir de descripciones de alto nivel, tales como reglas de negocios o leyes científicas. Tales reglas típicamente relacionan cantidades, ventas y registros de inventarios y así especifican los computos. Las otras partes de programa, especialmente cómo este interactuan con un usuario y cómo almacena datos en los discos de los computadores, son generados sin ninguna intervención humana.

Todas estas actividades ordenan a algún software de computadora hacer algo por nosotros. Algunos usan notación científica, algunos usan un castellano estilizado, algunos una notación concreta de programación. Todos ellos son alguna forma de programación. La esencia de estas actividades se reduce a dos conceptos:

1. relacionar una cantidad con otra cantidad, y
2. evaluar una relación sustituyendo los valores por nombres.

De hecho, los dos conceptos caracterizan la programación al nivel más bajo, el lenguaje nativo del computador, y en lenguajes modernos de moda tales como Java. Un programa relaciona sus entradas a salidas; y, cuando un programa es usado para entradas específicas, la evaluación substituye los valores concretos por nombres.

Nadie puede predecir que clase de paquetes de aplicación existirán en 5 o 10 años a partir de ahora. Pero los paquetes de aplicación requeriran alguna clase de programacion. Para preparar a los estudiantes para esa clase de actividades de programación, las escuelas pueden forzarlos a estudiar álgebra, que es la fundamentación matemática de la programación, o exponerlos a alguna forma de programación. Usando los lenguajes y entornos de programación modernos, las escuelas pueden hacer lo último, ellas puede hacerlo efectivamente, y pueden hacer el álgebra divertida.

Recetas de Diseño

> *Cocinar es tanto un juego de niños como un disfrute de adultos. Y cocinar hecho con amor es un acto de amor.*
> – Craig Claiborne (1920-2000), Editor de Comida, *New York Times*

Aprender a diseñar programas es como aprender a jugar fútbol. Un jugador debe aprender a atrapar el balón, a driblar con una bola, a pasar y disparar el balón. Una vez el jugador conoce estas habilidades básicas, las siguientes metas son aprender a jugar una posición, a jugar ciertas estratégias, a escoger entre estratégias factibles, y, en ocasiones, crear variaciones de una estrategia, porque ninguna de las estrategias existentes se acomoda.

Un programador es también muy similar a un arquitecto, un compositor o un escritor. Ellos son gente creativa que empiezan con ideas en sus cabezas y piezas de papel en blanco. Ellos conciben una idea, forman un bosquejo mental y lo refinan en papel hasta que sus escritos reflejan si imagen mental tanto como sea posible. En la medida en que ellos llevan ideas al papel, ellos emplean habilidades básicas de dibujo, escritura, e instrumentales, para expresar ciertos elementos de estilo de una edificación, describir el caracter de una persona, o formular porciones de una melodía. Ellos pueden practicar su oficio porque han afilado sus habilidades básicas por un largo tiempo y pueden usarlas en un nivel instintivo.

Los programadores también forman bosquejos, los traducen primero en diseños, e iterativamente los refinan hasta que realmente coinciden con la idea original. De hecho, los mejores programadores editan y reescriben sus programas muchas veces hasta que alcanzan ciertos estándares estéticos. Y como los jugadores de fútbol, arquitectos, compositores, o escritores, los programadores deben practicar las habilidades básicas de su oficio por un largo tiempo antes de que ellos puedan ser verdaderamente creativos.

Las recetas de diseño son el equivalente de las técnicas de manejo del balón del fútbol, las técnicas de escritura, las técnicas de disposición y las habilidades de dibujo. Una sencilla receta de diseño representa un punto del espacio en el diseño del programas. Hemos estudiado este espacio y hemos identificado muchas categorías importantes. Este libro selecciona las más fundamentales y más prácticas recetas y las presenta en orden incremental de dificultad[4].

Cerca de la mitad de las recetas de diseño se enfocan sobre la conexión entre los datos de entrada y los programas. Más específicamente, ellas muestra la plantilla de un programa es derivada de la descripción de los datos de entrada. Llamamos a esto diseño *orientado a los datos* y las la más frecuentemente usada forma de diseño. Los diseños orientados a los datos son fáciles de crear, fáciles de entender y fáciles de extender y modificar. Otras recetas de diseño introducen la idea de *recursión generativa*, *acumulación* y *sensibilidad a la historia*. La primera produce programas recursivos que generan nuevas instancias de problemas en la medida en que recurren; los programas del tipo acumulador recolectan datos en la medida en que procesan entradas; y los programas sensitivos a la historio recuerdan información entre aplicaciones sucesivas. Por ultimo, pero no menos importante, también introducimos una receta de diseño para *abstraer* sobre los programas. Abstraer es el acto de generalizar dos (o más) diseños similares en uno y derivar las instancias originales de este.

En muchas ocasiones un problema naturalmente sugiere una receta de diseño. En otras, un programador debe elegir de diferentes posibilidades; cada elección puede producir programas con organizaciones bastamente diferentes. Hacer elecciones es natural para un programador creativo. Pero, a menos que un programador sea concienzudamente familiar con el equipage de recetas de diseño para escoger y entienda las consecuencias de escoger una sobre

4. Nuestras recetas de diseño fueron inspiradas por el trabajo con Daniel P. Friedman sobre recursión estructural, con Robert Harper sobre teoría del diseño, y por el método de diseño de Michael A. Jackson.

la otra, el proceso es necesariamente *ad hoc* y permite caprichosos, malos diseños. Esperamos que a través de la delimitación de una colección de recetas de diseño, podamos ayudar a los programadores a entender qué escoger y cómo escoger.

Ahora que hemos explicado lo que queremos decir por "programación" y "diseño de programas", el lector puede ver como y porque enseñar diseño de programas instila hábitos de pensamiento que son importantes en una variedad de profesiones. Para diseñar un programa apropiadamente, un estudiante debe:

1. analizar el enunciado de un problema, típicamente establecido como un problema de palabras;
2. expresar su esencia, abstractamente y con ejemplos;
3. formular enunciados y comentarios en un lenguaje preciso;
4. evaluar y revisar estas actividades a la luz de verificaciones y pruebas; y
5. poner atención a los detalles.

Todas estas son actividades que son útiles para un hombre de negocios, un abogado, un periodista, un científico, un ingeniero y muchos otros.

Mientras que la programación tradicional requiere esas habilidades, también, los principiantes no entienden esta conexión. El problema es que los lenguajes de programación tradicionales y las formas de programación tradicionales forzan a los estudiantes a realizar grandes cantidades de *trabajo contable* y a memorizar un gran número de hechos específicos del lenguaje. Resumiendo, *el trabajo doméstico disminuye la enseñanza de las habilidades esenciales*. Para evitar este problema, los profesores deben usar un ambiente de programación que imponga tan poca sobre carga como sea posible y que se acomode a los principiantes. Puesto que tales herramientas no existían cuando empezamos, las desarrollamos.

La elección de Scheme y DrScheme

> *Nosotros ascribimos la belleza a aquello que es simple*
> *que no tiene partes superfluas;*
> *que responde exactamente a su fin,*
> *que permanece relacionado a todas las cosas*
> *que es el medio de muchos extremos.*
> –Ralph Waldo Emerson, *El Conducto de la Vida*

Hemos escogido Scheme como el lenguaje de programación para este libro, y hemos diseñado e implementado DrScheme, un entorno de programación para el lenguage con especial asistencia para estudiantes principiantes. El ambiente de programación está libremente disponible en el sitio Web oficial del libro[5].

Sin embargo, el libro no es sobre programación en Scheme. Sólo hemos usado un pequeño número de constructos de Scheme en este libro. Específicamente, usamos seis construcos (definición y aplicación de funciones, expresiones condicionales, definición de estructuras, definiciones locales, y asignaciones) además de una docena o algo así de funciones básicas. Este es un delgado subconjunto del lenguaje y es todo lo que se necesita para enseñar los principios de computación y programación. Alguién que desee usar Scheme como una herramienta necesitará leer material adicional[6].

La escogencia de Scheme para principiantes es natural. Primero, el núcleo de Scheme permite a los programadores enfocarse en sólo aquellos elementos de programación que señalamos al comienzo del prefacio: los programas como relaciones entre cantidades y evaluar programas para entradas específicas. Usando sólo este núcleo del lenguage, los estudiantes puede desarrollar programas completos *durante la primera sesión* con el profesor.

Segundo, Scheme puede fácilmente ser dispuesto como una torre de niveles de lenguaje. Esta propiedad es crucial para los principiantes quienes comenten simples errores notacionales que generan oscuros mensajes de error acerca de las características avanzadas del lenguaje. El resultado es a menudo una búsqueda infructousa y un sentimiento de frustración por parte del estudiante. Para evitar este problema, nuestro ambiente de programación, DrScheme, implementa varios sublenguajes de Scheme cuidadosamente seleccionados. Basados en esta disposición, el entorno puede señalar mensajes de error que son apropiados para el nible de conocimiento del estudiante. Mejor aún, esta estratificación de lenguajes previene muchos errores básicos. Desarrollamos las capas y los modos de protección observando a los principiantes por semanas en el laboratorio de computadores de Rice. En la medida en que los estudiantes aprender más acerca de la programación y el lenguaje, el profesor puede exponer a los estudiantes a capas más ricas del lenguaje que les permite a los estudiantes escribir programas más interesantes y más concisos.

5. Scheme tiene una definición oficial – el Reporte Revisado sobre Scheme, editado por Richard Kelsey William Clinger y Jonathan Rees – y muchas implementaciones. Para una copia del reporte y para una lista alternativa de implementaciones alternativas de Scheme, visite `www.schemers.org` en la Web. Note, sin embargo, que el lenguage de este libro extiende eso del reporte y está personalizado para principiantes.

6. Como cual? [*] Tener la posiblidad de hacer anotaciones al texto!, un estudiante/lector tendría la posibilidad de reportar las lecturas al texto a través de dichas anotaciones, tales como preguntas, hiperenlaces a otros documentos, referencias, etc.

Tercero, el entorno de programación de DrScheme ofrece un evaluador verdaderamente interactivo. Consiste de dos ventanas: una ventana de `Definiciones` donde los estudiantes definen programas y una ventana de `Interacciones`, que actua como una calculadora de bolsillo. Los estudiantes puede entrar expresiones en la última y DrScheme detemina sus valores. En otras palabras, la computacioón inicia con una aritmética de calculadora de bolsillo, que ellos conocen muy bien, y rápidamente procede desde allí a los cálculos con estructuras, listas y árboles ..las clases de dats que los programas de computadora realmente manipula. Más aún, un modo interactivo de evaluación anima al estudiante para experimentar en todas las clases de formas y así estimula su curiosidad.

Finalmente, el uso de un evaluador interactivo con un rico lenguaje de datos permite a los estudiantes enfocarse el las actividades de resolución de problemas y diseño de programas. La clave de la mejora es que la evaluación interactiva muestra una discusión de operaciones de entrada y salida (casi) superflua. Esto tiene varias consecuencias. Primero, las operaciones de entrada y salida requieren memorización. Aprender estas cosas es tedioso y aburridor. Recíprocamente, los estudiantes están en mejor posición aprendiendo habilidades de resolución de problemas y usando soporte [canned] de entrada y salida. Segundo, la *buena* entrada orientada al texto requiere habilidades de programación profundas, que son mejor adquiridas en un curso sobre resolución de problemas computacionales. Enseñar entrada orientada al texto mala es una desperdicio del tiempo de los estudiantes y los profesores. Tercero, el software moderno empela una Interface Gráfica de Usuario (GUI, por su sigla en inglés), en la cual los programadores diseñan con editores y "asistentes" pero no a mano. De nuevo, los estudiantes salen mejor librados aprendiendo el diseño de funciones que están conectadas a las reglas, botones, campos de texto y asó, que memorizando protocolos específicos que las bibliotecas de la GUI actualmente de moda impone. Brevemente, discutir entrada y salida es un desperdicio de valioso tiempo de aprendizaje durante una primera introducción a la programación. Si los estudiantes deciden [to pursue] programación con mayor profundidad, adquirir el conocimiento necesario (de Scheme) acerca de procedimientos de entrada salida es [straightforward].

En resumen, los estudiantes pueden aprender el núcleo de Scheme en un par de horas, aunque el lenguage es tan poderoso como un lenguaje de programación convencional. Como resultado, los estudiantes pueden enfocarse rápidamente en la esencia de la programación, lo cual ensancha grándemente sus habilidades generales de resolución de problemas.

Las partes del libro

El libro consiste en ocho partes y siete intermezzos. Las partes se enfocan en diseño de programas; los intermezzos introducen otros tópicos concernientes a la programación y la computación. La figura *.2 muestra el gráfico de dependencias para las partes del libro. El gráfico demuestra que hay varios caminos a lo largo del libro y que un cubrimiento parcial del material es factible.

Las partes I hasta la III cubren los fundamentos del diseño de programas orientado a los datos. La parte IV introduce la abstracción en los diseños. Las partes V y VI son acerca de recursión generativa y acumulación. Para estas primeras seis partes, el liebor usa una completamente funcinal – o algebráica – forma de programación. Una y la misma expresión siempre evalúa el mismo resultado, no importa cuan a menudo lo evaluemos. Esto propiamente hace fácil diseñar, y razonar acerca de, programas. Para enfrentarnos con las interfaces entre los programas y el resto del mundo, sin embargo, enriquecemos el lenguaje con sentencias de asignación y abandonamos algo de nuestro razonamiento algebráico. Las dos últimas partes muestran lo que esto significa para el diseño de programas. Más precisamente, ellas muestran cómo las recetas de diseño de las primeras seis partese se aplican y porque debemos ser mucho más cuidadosos una vez que las asignaciones son adicionadas.

Los intermezzos introducen tópicos que son importantes para la computación y programación en general pero no para el diseño de programas per se. Algunos introducen la sintaxis y la semántica de nuestro sbuconjuntos escojidos de Scheme sobre una base rigurosa, unos pocos introducen constructos de programación adicionales. El intermezzo 5 es una discusión sore el costo abstracto de la computación (tiempo, espacio, energía) e introduce los vectores. El intermeso 6 contrasta dos formas de representar números y procesarlos.

El cubrimiento de algunos intermezzos puede ser demorado hasta que una necesidad específica arrive. Esto es especialmente cierto en los intermezzos sobre la sintaxis y semántica de Scheme. Pero, considerando el role central del intermezzo 3 en la figura *.2, debería ser cubierto en una forma temprana.

Figura *.2. Las dependencias entre las partes y los intermezzos

Refinamiento Iterativo e Iteración de Tópicos: El diseño de programas sistemático es particularmente interesantes e importante para un gran proyecto. El paso de problemas de una pequeña función simple a pequeños projectos multifunción requiere una idea adicional de diseño: un refinamiento iterativo. La meta es diseñar el núcleo del programa y adicionar funcionalidad a este núclico hasta que el conjunto completo de requerimientos es alcanzado.

Los estudiantes en un primer curso pueden, y deben, obtener su primera degustación del refinamiento iterativo. Por eso, a fin de acostumbrar a los estudiantes con la técnica, hemos incluido ejercicios extendidos. Típicamente, una breve panorámica configura el estado para una colección de ejercicios. Los ejercicios gentilmente guían los estudiantes a través de algunas iteraciones de diseño. En la sección 16, la idea es dicha explícitamente.

Más adelante, el libro revisa ciertos ejercicios y temas de ejemplo de vez en cuando. Por ejemplo, las secciones 6.6, 7.4, 10.3, 21.4, 41.4, y unos pocos ejercicios en el medio de las dos últimas secciones cubren la idea de mover imágenes a través de un lienzo (canvas). Los estudiantes así ven el problema varias veces, cada vez con más y más conocimiento acerca de cómo organizar programas.

Adicionar piezas de funcionalidad a un programa demuestra por qué los deben seguir una disciplina de diseño. Resolver un problema de nuevo le muestra a los estudiantes como escojer de recetas de diseño alternativas. Finalmente, de modo ocasional, el nuevo conocimiento ayuda a los estudiantes a mejorar la organización del programa; en otras palabras, los estudiantes aprender que los programas son están finalizados después de que ellos los trabajan por primera vez sino que, como los ensayos y libros, necesitan edición.

TEACHPACKS: Un segundo aspecto de trabajar en proyectos es que los programadores deben trabajar en grupos. En un contexto instruccional, esto signigica que el programa de un estudiante tiene que acomodarse precisamente al de alguién más. Para simular lo que significa "acomodar la función de uno a la de alguién más", proveemos al DrScheme con paquetes de enseñanaza (teachpack). Diciéndolo gruesamente, un paquete de enseñanza simula un compañero de equipo aunque evita la frustración de trabajar con los componentes de programa defectuosos del compañero. Más técnicamente, el proyecto casi siempre consiste de un componente de programa de una vista y un modelo (en el sentido vista-modelo de la arquitectura de software). En una disposición típica, los estudiantes diseñan el componente modelo. Los paquetes de enseñanza proveen los componentes de vista, a menudo en forma de interfaces (gráficas) de usuario. Así ellos eliminan las porciones de codificación tediosas e irreflexivas. Inclusive, esta separación particular de preocupaciones imita los proyectos del mundo real.

Acomodar los componentes de modelo a los componentes de vista requiere que los estudiantes presten atención a especificaciones precisas de funciones. Esto demuestra la importancia capital de seguir una disciplina de diseño. Es también una habilidad crítica para los programadores y es a menudo sub-enfatizada en los primeros cursos. En la parte IV mostramos cómo construir algunas GUIs (Interfaces Gráficas de Usuario, por sus siglas en inglés) y como los eventos de la GUI disparán la aplicación de las funciones de modelo. El objetivo es explicar que construir GUIs no es un misterio, pero no para gastar mucho tiempo en un tópico que requiere en su mayoría aprendizaje rutinario y poco pensamiento computacional.

CALENDARIO: Cada universidad, preparatoria y escuela tiene sus propias necesidades y debe encontrar un calendario apropiado. En la Universidad de Rice, convencionalmente cubrimos el libro entero más algún material adicional en un sólo semestre. Un instructor en una universidad investigadora debería probablemente matener una marcha similar. Un profesor de preparatoria perseguirá necesariamente una marcha más lenta. Muchos de los profesores de preparatoria que han probado el libro cubrieron las primeras tres partes en un semestre; algunos usaron sólo la primera parte para enseñar resolución algebráica de problemas desde una perspectiva computacional; y aún otros trabajaron a lo largo del libro completo en un año. Para mayor información acerca de calendarios, visite el sitio Web del libro.

EL LIBRO EN LA WEB: El libro viene en dos versiones: una copia de papel y una versión en línea líbremente disponible en

`http://www.htdp.org`

El sitio Web también provee material adicional, especialmente ejercicios expandidos del estilo mencionado arriba. En este momento, la página Web ofrece ejercicios sobre la simulación visual de juegos de pelota y la administración de un sitio Web. Más ejercicios serán adicionados.

Las dos versiones del libro vienen con diferentes clases de pistas. Cada una está marcada con uno de los siguientes tres iconos:


Esta marca se refiere a *pistas de DrScheme*; ellas están disponibles en las dos versiones del libro. El ambiente de programación ha sido diseñado con los estudiantes en mente. Las pistas sugieren cómo usar DrScheme en varios estados del proceso de aprendizaje.


Esta marca se refiere a *pistas del profesor*, que sugiere estrategias sobre cómo presentar una sección, cómo aproximarse a un ejercicio, o cómo complementar algún material.


Esta marca se refiere a *soluciones en línea*. Algunas soluciones están libremente disponibles; otras son accesibles a profesores registrados únicamente. Para encontar más acerca del registro, vea el sitio Web del libro.

TIPOGRAFÍA Y DEFINICIONES: Por legibilidad, los programas de Scheme son impresos usando un pequeño número de fuentes. Las palabras en *itálica* se refieren a nombres de programas y variables. Los items Sans Serif son constantes y operaciones pre-construidas. Las palabras en **negrilla** son palabras reservadas de Scheme.

Las definiciones vienen en tres variedades. Hay aquelos términos que se refieren a los principios de la programación y la computación. El libro lista la primera ocurrencia de tales términos con PEQUEÑAS LETRAS MAYÚSCULAS. Otras definiciones son de una naturaleza más pasajera; ellos introducen términos que son importantes para una sección, un ejemplo, un ejercicio, o alguna otra pequeña parte del libro. El libro usa palabras en *cursiva* para enfatizar tales definiciones. Finalmente, el libro también define clases de datos. La mayoría de las definiciones de datos están encerrados y la primera ocurrencia del nombre definido es también escrita usando palabras en *cursiva*.

Agradecimientos

Cuatro personas merecen gracias especiales: Robert "Corky" Cartwright, quien co-desarrollo un predecesor del curso introductorio de Rice con el primer autor; Daniel P. Friedman, por pedir al primer autor reescribir *The Little LISPer* (también de MIT Press) en 1984, porque eso inició este proyecto; John Clements, quien diseño, implementó y mantiene el stepper de DrScheme; y Paul Stecker, quien fevorosamente soportó el equiepo con contribuciones de nuestra suite de herramientas de programación.

El desarrollo de este libro se benefició de muchos otros amigos y colegas quienes lo usaron en sus cursos y/o dieron comentarios detallados en los primeros bosquejos. Estamos agradecidos a ellos por su ayuda y su paciencia: Ian Barland, John Clements, Bruce Duba, Mike Ernst, Kathi Fisler, Daniel P. Friedman, John Greiner, John Stone, Geraldine Morin, y Valdemar Tamez.

Una docena de estudiantes de *Comp 210* en la Universidad de Rice usaron los primeros borradores del texto y contribuyeron a mejoras en varias formas. En adición, numerosos asistentes a nuestros talleres de trabajo de TeachScheme! usaron los primeros borradores en sus salones de clase. Muchos enviaron comentarios y sugerencias. Como representante de ellos mencionamos a los siguientes contribuyentes activos: Ms. Barbara Adler, Dr. Stephen Bloch, Mr. Jack Clay, Dr. Richard Clemens, Mr. Kyle Gillette, Ms. Karen Buras, Mr. Marvin Hernandez, Mr. Michael Hunt, Ms. Karen North, Mr. Jamie Raymond, y Mr. Robert Reid. Christopher Felleisen pacientemente trabajó hacia las primeras pocas partes del libro con su padre y proveyó comprensiones directas en las visiones de un joven estudiante. Hrvoje Blazevic (navegando, en ese momento, como Maestro del LPG/C Harriette), Joe Zachary (University de Utah) and Daniel P. Friedman (Indiana University) descubrieron numerosos errores tipográficos en la primera impresión, que ahora han sido arreglados. Gracias a cada uno.

Finalmente, Matthias expresa su gratitude a Helga por sus muchos años de paciencia y por crear un hogar para un esposo y padre de mente ausente. Robby está agradecido con Hsing-Huei Huang por el soporte y corage de ella; sin ella, el no hubiera conseguido algo. Matthew agradece a Wen Yuan por su constante soporte y permanente música. Shriram está endeudado con Kathi Fisler por su soporte, paciencia y empuje, y por la participación de ella en este proyecto.

Capítulo 1
Estudiantes, Profesores y Computadores

Aprendemos a computar a una edad temprana. Al comienzo sólo sumamos y restamos números.

> Uno mas uno igual dos. Cinco menos dos es tres.

En la medida en que nos hacemos más viejos aprendemos operaciones matemáticas adicionales, como exponenciación y seno, pero también aprendemos a describir las reglas de la computación.

> Dado un círculo de radio r, su circunferencia es r veces dos veces pi. Un trabajador de salario mínimo que trabaja por N horas gana 5.35 dólares.

La verdad es, que nuestros profesores nos convierten en computadores y nos programan para ejecutar sencillos programas de computadora.

Así, el secreto está revelado. Los programas de computadora son sólo estudiantes muy rápidos. Ellos pueden realizar millones de adiciones mientras nosotros podríamos aún estar varados con la primera. Pero los programas de computadores puede hace más que sólo manipular números. Ellos pueden guiar un aeroplano. Ellos pueden jugar juegos. Ellos pueden buscar el número telefónico de una persona. Ellos pueden imprimir cheques de nomina para enormes corporaciones. En resumen, los computadores procesan toda clase de información.

La gente establece la información y las instrucciones en español.

> La temperatura es $35^o\ C$; convierta esta temperatura a Fahrenheit. Le toma a este carro 35 segundos acelerar desde cero a 100 millas por hora; determine que tan lejos llega el carro en 20 segundos.

Los computadores, sin embargo, escasamente entienden el español básico y ciertamente no pueden entender instrucciones complejas expresadas en español. En su lugar debemos aprender a hablar un lenguaje de computador de modo que podamos comunicar información e instrucciones.

Un lenguaje de computador de instrucciones e información es un *Lenguaje de Programación*. La información expresada en un lenguaje de programación es llamada *datos*. Hay muchos sabores de datos. Los *números* son una clase de datos. Las *series numéricas* pertenence a las clase de *datos compuestos*, porque cada una de esas series está hecha de otras piezas de pequeños trozos de datos, denominadas números. Para contranstar las dos clases de datos, también llamamos a los números *datos atómicos*. Las letras son otros ejemplos de datos atómicos; las familias de árboles son datos compuestos.

Los datos representan información, pero la interpretación concreta es dejada a nosotros. Por ejemplo, un número como 37.51 puede representar una temperatura, un tiempo, o una distancia. Una letra como "A" puede representar un gradod de escuela, un número de calidad para los huevos, o una parte de una dirección.

Como los datos, las instrucciones, también llamadas *operaciones*, vienen en diferentes sabores. Cada clase de datos viene con una serie de *operaciones primitivas*. Para los números, naturalmente tenemos $+$, $-$, $*$, y así sucesivamente. Los programadores componen operaciones primitivas en programas. Así, podemos pensar en las operaciones primitivas como las palabras de un lenguaje y en la programación como en la formación de sentencias en este lenguaje.

Algunos programas son pequeños, como los ensayos. Otros son como conjuntos de enciclopedias. Escribir buenos ensayos y libros requiere de cuidado y planeación, y escribir buenos programas lo requiere, también. Pequeño o grande, un buen programa no puede ser creado mediante el remendado[1] alrededor de él. Tiene que ser cuidadosamente *diseñado*. Cada pieza necesita mucha atención; componer programas en unidades más grander debe serguir una estrategia bien planeada. Diseñar programas apropiadamente debe ser practicado desde nuestro primer día de programación.

En este libro aprenderemos a diseñar programas de computadora y aprenderemos a entender cómo funcionan. Conventirser y ser un programador es devertido, pero no es fácil. La mejor parte de ser un programador es mirar nuestros "productos" crecer y volverser exitosos. Es divertido observar un computador jugar un juego. Es exitante ver un programa de computador ayudar a alguian. Para llegar a este punto, sin embargo, debemos practicar muchas habilidades. Como descubriremos, los lenguajes de programación son primitivos; especialemente, su gramática es restrictiva. E infortunadamente, los computadores son estúpidos. El más pequeño error gramatical en un programa es bloque retenedor fatal para un computador. Peor aún, una vez nuestro programa esta en una forma gramatical apropiada, podría no realizar los computos como se pretendió.

1. Una acepción tradicional en Colombia alrededor de esta creencia del remendado de programas es que se aprende informática "cacharreando" con los computadores. El verbo "cacharrear" viene de la idea de trabajar con cacharros. N. del T.

Programar un computador requiere paciencia y concentración. Sólo la atención a los detalles minuciosos evitará errores gramaticales frustrantes. Sólo la rigurosa planeación y adherencia al plan prevendrá serios errores lógios en nuestros diseños. Pero cuando finalmente dominemos el diseño de programas, habremos aprendido habilidades que son útiles mucho más allá del reino de la programación.

Empecemos!

Capítulo 2

Números, Expresiones y Programas Sencillos

Al comienzo, la gente pensó en los computadores como [*] number cruchers. Y de hecho los computadores son muy buenos trabajando con números. Puesto que los profesores empiezan sus primeros graduandos en computación con números, empezaremos con números, también. Una vez sepamos cómo los computadores tratan con los números, podemos desarrollar programas simples en poco tiempo; sólo traducimos nuestro sentido comnun a nuestra notación de programación. Sin embargo, incluso desarrollar tales programas simples requiere disciplina, y por eso introducimos el esbozo de la receta de diseño más fundamental y la línea guía básica de programación al final de esta sección.

2.1 Números y Aritmética

 Computación [*]

Los números pueden venir en diferentes sabores: enteros positivos y negativos, fracciones (también conocidas como racionales), y los reales que son la clase más ampliamente conocida de números:

```
5          -5          2/3          17/3          #i1.4142135623731
```

El primero es un entero, el segundo es un entero negativo, los otros dos son fracciones y el último es una representación inexacta de un número real.

Como una calculadora de bolsillo[1], que es el más sencillo de los computadores, Scheme permite a los programadores adicionar, sustraer, multiplicar y dividir números:

```
(+ 5 5)      (+ -5 5)      (+ 5 -5)      (- 5 5)      (* 3 4)      (/ 8 12)
```

Las primeras tres piden a Scheme realizar adiciones; los últimos tres demandan una sustracción, una multiplicación, y una división. Todas las expresiones aritméticas están entre paréntesis y mencionan la operación primero; los números siguen la operación y están representados por espacios.

 Steper [*]

Como en aritmética o álgebra, podemos anidar las expresiones:

```
(* (+ 2 2) (/ (* (+ 3 5) (/ 30 10)) 2))
```

Scheme evalua estas expresiones exactamente como nosotros lo hacemos. Primero reduce la expresión entre paréntesis más interior a números, luego lo de la siguiente capa, y así sucesivamente:

```
  (* (+ 2 2) (/ (* (+ 3 5) (/ 30 10)) 2))
= (* 4 (/ (* 8 3) 2))
= (* 4 (/ 24 2))
= (* 4 12)
= 48
```

Porque cada expresión de Scheme tiene la forma

1. Relacionar con la sección "Python como una calculadora" del tutorial de Python por Guido Van Rosum. N del T.

```
(operation A ... B)
```

nunca hay alguna pregunta acerca de cúal parte tiene que ser evaluada primero. Siempre que `A ... B` sean números la expresión puede ser evaluada; de otro modo, `A ... B` son evaluados primero. Contrástelo con esto

$$3 + 4 \cdot 5,$$

que es una expresión que encontramos en el colegio. Sólo una cantidad substancial de práctica garantiza que recordemos evaluar la multiplicación primero y la adición después[2].

Finalmente, Scheme no sólo provee operaciones artiméticas simples, sino un completo rango de operaciones matemáticas avanzadas con números. Aquí están cinco ejemplos:

1. `(sqrt A)` calcula $(A)^{1/2}$;
2. `(expt A B)` calcula A^B;
3. `(remainder A B)` calcula el residuo de la división entera A/B;
4. `(log A)` calcula el logaritmo natural de A; y
5. `(sin A)` calcula el seno de A radianes.

Cuando dude bien sobre la existencia de una operación primitiva o sobre cómo funciona, use DrScheme para probar si una operación está disponible con un simple ejemplo.

Una Nota sobre los Números: Scheme computa con enteros *exactos* y racionales en la medida en que usemos operaciones primitivas que produzcan resultados exactos. Así, mostrará el resultado de `(/ 44 14)` como `22/7`. Desafortunadamente, Scheme y otros lenguajes de programación se comprometen tan pronto como los números reales son involucrados. Por ejemplo, puesto que la raíz cuadrada de `2` no es un número racional sino un número real, Scheme usa un *número inexacto*:

```
  (sqrt 2)
= #i1.4142135623731
```

La notación `#i` advierte al programador que el resultado es una aproximación del número verdadero. Una vez un número inexacto se ha convertido en parte de un cálculo, el proceso continua de una manera aproximada. Es decir:

```
  (- #i1.0 #i0.9)
= #i0.09999999999999998
```

pero

```
  (- #i1000.0 #i999.9)
= #i0.10000000000002274
```

incluso aunque sabemos por las matemáticas que ambas diferencias deberían ser `0.1` e iguales. Una vez que los números son inexactos, la precausión es necesaria.

Esta impresición es debida a la simplificación corriente de escribir números como la raíz cuadrada de 2 o [*] como números racionales. Recuerde que las representaciones decimales de estos números son infinitamente largar (sin repetición). Un computador, sin embargo, tiene un tamaño finito, y por esto sólo puede representar una porción de tal número. Si escogemos rpresentar estos números como racionales con un número fijo de dígitos, la representación es necesariamente inexacta. El intermezzo 6 explicará como los números inexactos trabajan.

Para enfocar nuestros estudios en los conceptos importantes de la computación y no en esos detalles, los lenguajes de enseñanza de DrScheme tratan tanto como es posible los números como números precisos. Cuando escribimos `1.25`, DrScheme interpreta este número como una fracción precisa, no como un número inexacto. Cuando la ventana de *interacciones* de DrScheme muestra un número tal como `1.25` o `22/7`, es el resultado de una computación con racionales precisos y fracciones. Sólo lo numeros precedidos por `#i` son representaciones inexactas.

Ejercicio 2.1. Descubra si DrScheme tiene operaciones para sacar raíz de un número; para computar el seno de un ángulo y para determinar el máximo de dos números.

2. Otra ventaja de la notación de Scheme es que siempre podemos saber donde colcocar un operador o donde encontrarlo: a la derecha inmediata del paréntesis de apertura. Esto es importante en computación porque necesitamos muchos más operadores de los que usamos en aritmética y en algebra.

Ejercicio 2.2. Evalue `(sqrt 4)`, `(sqrt 2)` y `(sqrt -1)` en DrScheme. Luego, descubra si DrScheme sabe una operación para determinar la tangente de un ángulo.

2.2 Variables y Programas

 Programación [*]

En álgebra aprendemos a formular dependencias entre cantidades usando *expresiones variables*. Una variable es un guarda puesto que se establece para una cantidad desconocida. Por ejemplo un cículo de radio r tiene un área aproximada[3]

$$3.14 \cdot r^2$$

En esta expresión r representa cualquier número positivo. Si ahora llegamos con un círculo de radio 5, podemos determinar su área sustituyendo 5 por r en la fórmula de arriba y reduciendo la expresión resultante al número:

$$3.14 \cdot 5^2 = 3.14 \cdot 25 = 78.5$$

Más generalmente las expresiones que contienen variables son reglas que describren como calcular un número *cuando* nos han dado los valores para las variables.

Un programa es tal como una regla. Es una regla que dice al computador y a nosotros como producer datos de algún otro dato. Es por esto importante que los programadores nombren cada regla en la medida en que ellos las escriben. Un buen nombre para nuestra expresión de ejemplo es `area-del-circulo`. Usando este nombre[4], podríamos exprear la regla para clacular el área de un disco como sigue:

```
(define (area-del-circulo r)
  (* 3.14 (* r r)))
```

Las dos líneas dicen que el `area-del-circulo` es una regla, que consume una *entrada* sencilla, llamada `r`, y que el resultado, o salida, va a ser `(* 3.14 (* r r))` una vez que sepamos a cual número representa `r`.

Los programas combinan operaciones básicas. En nuestro ejemplo `area-del-circulo` usa sólo una operación básica, la multiplicación, pero los programas definidos puede usar tantas operaciones como sea necesario. Una vez hemos definido un programa, podemos usarlo como si fuera una instrucción primitiva. Para cada variable listada a la derecha del nombre del programa, debemos proveer una entrada. Esto es, podemos escribir expresiones cuya operación es `area-del-circulo` seguida por un número:

```
(area-de-circulo 5)
```

También decimos que *aplicamos* `area-de-circulo` a 5.

La aplicación de una operación definida (usando el operador `define`) es evaluada compiando la expresión llamada `area-de-circulo` y reemplazando la variable (`r`) con el número que suplimos (`5`):

```
  (area-of-disk 5)
= (* 3.14 (* 5 5))
= (* 3.14 25)
= 78.5
```

Muchos programas consumen más de una entrada. Digamos que deseamos definir un programa que computa el área de un anillo, esto es un círculo con un hoyo en el centro:

3. En este punto el texto original hace alución a lo incorrecto de usar la expresión “area of a circle” pero esto es porque la palabra “circle” se refiere a la palabra española “circunferencia” es decir el borde de un círculo, mientras que el círculo propiamente dicho se refiere a la parte interior, a la cual sí se le puede calcular el área. El uso de círculo y circunferencia en español no genera ninguna ambigüedad en virtud de los significados distintos de cada uno. N del T.

4. Nótese que los nombres de las variables, reglas y procedimientos que se usan no contienen en general caracteres acentuados o propios del castellano, tales como la ñ, pues pueden generar errores en virtud de que los intérpretes para los lenguajes de programación podrían no tener soporte para dichos caracteres. El soporte Unicode eventualmente podría cambiar esta situación, sin embargo se aconseja no usar caracteres con tilde o ñ para cosas distintas a los comentarios dentro de un programa. N. del T.

El área del anillo es aquella del círculo exterior menos el área del círculo interior, lo que signigica que el programa requiere dos cantidades desconocidas: el radio externo e interno. Llamemos a esos número desconocidos `externo` e `interno`. Entonces el programa que calcula el área del anillo está definido como sigue:

```
(define (area-del-anillo enterno interno)
  (- (area-del-circulo externo)
     (area-del-circulo interno)))
```

Estas tres líneas expresan que el `area-del-anillo` es un programa, que el programa acepta dos entradas llamadas `externa` e `interna`, y que el resultado va a ser la diferencia entre `(area-del-circulo externo)` y `(area-del-circulo interno)`. En otras palabras, hemos usado tanto operaciones básicas de Scheme como programas definidos en la definición de `area-del-anillo`.

Cuando deseamos usar `area-del-anillo`, debemos suplir dos entradas:

```
(area-del-anillo 5 3)
```

La expresión es evaluda en la misma manera que `(area-del-circulo 5)`. Copiamos la expresión de la definición del programa y reemplazamos las variables con números que suplimos:

```
  (area-del-anillo 5 3)

= (- (area-del-circulo 5)
     (area-del-circulo 3))

= (- (* 3.14 (* 5 5))
     (* 3.14 (* 3 3)))
= ...
```

El resto es pura aritmética.

Ejercicio 2.3. Defina el programa `Fahrenheit->Celsius`,[5] que consume una temperatura medida en Fahrenheit y produce el equivalente Celsius. Use un libre de química o física para mirar la fórmula de conversión.

 Paquete de enseñanza [*]

Cuando la función esté totalmente desarrollada, pruebela usando el paquete de enseñanza **convert.ss**. El paquete de enseñanza provee tres funciones: `convert-gui`, `convert-repl` y `convert-file`. El primero crea una interface gráfica de usuario. Úselo con

```
(convert-gui Fahrenheit->Celsius)
```

La expresión creará una nueva ventana en la cual los usuarios puede manipular una barra de desplazamiento y unos botones.

El segundo emula la ventana de *interacciones*. A los usuarios se les pide entrar una temperatura Fahrenheit, la cual el programa lee, evalua e imprime. Úselo vía:

```
(convert-repl Fahrenheit->Celsius)
```

La última operación procesa archivos completos. Para usarla, cree un archivo con aquellos números que están para ser convertidos. Separe los números con espacios en blanco o nuevas líneas. La función lee el archivo completo, convierte los números y escribe los resultados en un nuevo archivo. Aquí está la expresión:

```
(convert-file "entrada.dat" Fahrenheit->Celsius "salida.dat")
```

Este asume que el nombre del archivo recien creado es `entrada.dat` y que deseamos que los resultados sean escritos en el archivo `salida.dat`. Para más información use el Help Desk de DrScheme para mirar el paquete de enseñanza **convert.ss**.

5. Una flecha es indicada como un `-` seguido por un `>`.

Ejercicio 2.4. Defina el programa `dolar->euro`, que consume un número de dólares y produce un equivalente en euros. Use una tabla de monedas en el periodico o mire la taza de intercambio actual.

Ejercicio 2.5. Defina el programa `triangulo`. Consume la longitud del lado del triángulo y su altura. El programa produce el área de un triángulo. Use un libro de geometría para mirar la fórmula para computar el área del triángulo.

Ejercicio 2.6. Defina el programa `convert3`. Consume tres dígitos, empezando con el dígito menos significativo, seguido por el siguiente más significativo y así sucesivamente. El programa produce el número correspondiente. Por ejemplo el valor esperado de

```
(convert3 1 2 3)
```

es `321`. Use un libro de álgebra para encontrar cómo tal conversión trabaja.

Ejercicio 2.7. Un típico ejercicio en un libro de álgebra pide al lector evaluar expresiones como

$$\frac{n}{3}+2$$

para $n=2, n=5$ y $n=9$. Usando Scheme, podemos formular tal expresión como un programa y usar el programa tantes veces como sea necesario. Aquí hay un programa que corresponde a la expresión de arriba

```
(define (f n)
  (+ (/ n 3) 2))
```

Primero determine el resultado de la expresión en $n=2, n=5$ y $n=9$ a mano, luego con el stepper de DrScheme.

También formule las siguientes tres expresiones como programas:

1. n^2+10
2. $(1/2)\cdot n^2+20$
3. $2-(1/n)$

Determine sus resultados para $n=2, n=5$ y $n=9$ a mano, luego con DrScheme.

2.3 Problemas de palabras

Los programadores raramente manipulan expresiones matemáticas para convertirlas en programas. En cambio ellos reciben típicamente descripciones informales de problemas que a menudo contienen información irrelevante y algunas veces ambiciosa. La primera tarea de los programadores es extraer la información relevante y luego formular las expresiones apropiadas.

Acá hay un ejemplo típico:

> La compañía XYZ & Co. paga a todos sus empleados US$12 por hora. Un empleado típico trabaja entre 20 y 65 horas por semana. Desarrolle un programa que determine el salario de un empleado del número de horas de trabajo.

La última sentencia es la primera meción a la tarea actual: escribir un programa que determine una cantidad basada en alguna otra cantidad. Más específicamente, el programa consume una cantidad, el número de horas de trabajo, y produce otra, el salario en dólares. La primera sentencia implica cómo computar el resultado, pero no lo establece explícitamente. En este ejemplo particular sin embargo, esto no ocasiona problema. Si un empleado trabaja h horas, el salario es

$$12\cdot h$$

Ahora que tenemos una regla, podemos formular un programa de Scheme:

```
(define (wage h)
  (* 12 h))
```

El programa es llamado `wage`; sus parametro `h` se coloca para las horas que un empleado trabaja; y su resultado es `(* 12 h)`, el salario correspondiente.

Ejercicio 2.8. Los contadores de impuestos de utopia siempre usan progrmas que computan impuestos de ingresos incluso cuando la taza de impuestos es un sólido y nunca cambiente 15%. Defina el programa `impuesto` que determina el impuesto sobre el pago bruto.

También defina `pago-neto`. El programa determina el pago neto de un empleado del número de horas trabajadas. Asuma una taza horaria de US$12.

Ejercicio 2.9. El supermercado local necesita un programa que pueda computar el valor de una bolsa de monedas. Defina el programa `suma-monedas`. Consume cuatro números: el número de monedas de $50, $100, $500 y $1000 en la bolsa; produce la cantidad de dinero en la bolsa.

Ejercicio 2.10. Un teatro de viejo estilo tiene una sencilla función de lucro. Cada cliente paga US$5 por tiquete. Cada realización cuesta al teatro US$20, más US$.50 por asistente. Desarrolle la función `ganacia-total`. Consume el número de asistentes (de una presentación) y produce cuantos ingresos los asistentes producen.

2.4 Errores

 Errores [*]

Cuando escribimos programas en Scheme, debemos seguir unas pocas reglas diseñadas cuidadosamente, que son un compromiso entre las capacidades del computador y el comportamiento humano[6]. Afortunadamente, formar definiciones de Scheme es intuitivo. Las expresiones son bien *atómicas*, esto es, números y variables; o son *compuestas*, en cuyo caso empiezan por "(", seguido por una operación, algunas expresiones más, y terminadas por ")". Cada expresión en una operación compuesta debería ser precedida por al menos un espacio; los saltos de línea son permisibles, y algunas veces incrementela la legibilidad.

Las definiciones tienen la siguiente forma esquemática:

```
(define (f x ... y)
  una-expresion)
```

Esto es, una definición es una secuencia de varias palabras y expresiones "(", la palabra "define", "(", una secuencia no vacia de nombres separados por espacios, ")". una expresión, y un ")" de cierre. Las secuencia embebida de nombres `f x ... y`, introduce el nombre del programa y el nombre de sus parámetros.

Errores Sintácticos:[7] No todas las expresiones entre paréntesis son expresiones Scheme. Por ejemplo, `(10)` es una empresión entre paréntesis, pero Scheme no la acepta como una expresión legal de Scheme porque los números no se soponen para ser incluidos en paréntesis. Similarmente, una secuencia como `(10 + 20)` está también debilmente formada; las reglas de Scheme demandan que el operador sea mencionado primero. Finalmente las siguientes dos expresiones no están bien formadas:

```
(define (P x)
  (+ (x) 10))

(define (Q x)
  x 10)
```

La primera contiene una par extra de paréntesis alrededor de la variable `x`, la cual no es una expresión compuesta; la segunda contiene dos expresiones atómicas, `x` y `10`, en lugar de una.

Cuando hacemos click en el botón Ejecutar de DrScheme, el ambiente de programación primero determina si las definiciones están formadas de acuerdo a las reglas de Scheme. Si alguna parte del programa en la ventana de `Definiciones` están formada debilmente, DrScheme señala un `Error de Sintaxis` con una mensaje de error apropiado y subraya la parte ofensiva. De otro modole permite al usuario evaluar la expresión en la ventana de `Interacciones`.

 Errores[*]

Ejercicio 2.11. Evalue las siguientes expresiones en DrScheme, una a la vez:

```
(+ (10) 20)
(10 + 20)
(+ +)
```

6. Esta sentencia es verdadera para cualquier otro lenguaje de programación también, por ejemplo, lenguajes de hoja de cálculo, C, macros de procesador de palabras. Scheme es más simple que la mayoría de esos y fácil de entender para los computadores. Infortunadamente, para los seres humanos quienes crecieron con expresiones infijas tales como 5 + 4, las expresiones prefijas de Scheme tales como `(+ 5 4)` inicialmente parecen ser complicadas. Un poco de práctica eliminará rápidamente esta concepción errónea.

7. Descubriremos en la sección 8 que por qué tales errores son llamados errores de *sintaxis*.

Lea y comprenda los mensajes de error.

Ejercicio 2.12. Entre las siguientes sentencias, una por una, en la ventana de `Definiciones` de DrScheme y haga click en `Ejecutar` :

```
define (f 1)
  (+ x 10))

(define (g x)
  + x 10)

(define h(x)
  (+ x 10))
```

Lea los mensajes de error, fije la definición ofensiva de modo apropiado y repita hasta que todas las definiciones sean legales.

Errores de ejecución: La evaluación de expresiones Scheme procede de acuerdo a las leyes intuitivas del álgebra y la aritmética. Cuando encontramos nuevas operaciones, entedemos estas leyes, primero intuitivamente y luego, en la sección 8, rigurosamente. Por ahora, es más importante comprender que no todas las expresiones legales de Scheme tienen un resultado. Un ejemplo obvio es `(/ 1 0)`. Similarmente si definimos

```
(define (f n)
  (+ (/ n 3) 2))
```

no podemos pedirle a DrScheme evaluar `(f 5 8)`.

Cuando la evaluación de una expresión legal de Scheme demanda una división por cero o una operación aritmética similarmente sin sentido, o cuando un programa es aplicado al número incorrecto de entradas, DrScheme detiene la evaluación y señana un *error de ejecución*. Típicamente imprime una explicación en la ventana de `Interacciones` y subraya la expresión fallida. La expresión subrayada disparó la señal de error.

Ejercicio 2.13. Evalue las siguientes expresiones Scheme gramaticalmente legales en la ventana de `Interacciones` de DrScheme:

```
(+ 5 (/ 1 0))

(sin 10 20)

(somef 10)
```

Lea los mensajes de error

Ejercicio 2.14. Entre el siguiente programa Scheme gramaticalmente legal en la ventana de `Definiciones` y haga click en el boton `Ejecutar` :

```
(define (somef x)
  (sin x x))
```

Luego en la ventana `Interacciones`, evalue las siguientes expresiones:

```
(somef 10 20)
(somef 10)
```

Errores lógicos: Un buen entorno de programación asiste al programador en encontrar errores sintácticos y de ejecución. Los ejercicios en esta sección ilustran cómo DrScheme atrapa los errores sintácticos y de ejecución. Un programador, sin embargo, puede tambien cometer *errores lógicos*. Un error lógico no dispara ningún mensaje de error; en cambio, el programa calcula resultados incorrectos. Considere el programa `salario` de la sección precedente. Si el programador lo ha definido accidente como

```
(define (salario h)
  (+ 12 h))
```

el programa aún produciría un número cada vez que es usado. De hecho, si evaluamos `(salario 12/11)`, incluso obtendremos el resultado correcto. Un programador puede atrapar tales errores sólo diseñando programas cuidadosa y sistemáticamente.

2.5 Diseñando Programas

Las secciones precedentes muestras que el desarrollo de un programa requiere muchos pasos. Necesitamos determinar qué es relevante en el enunciado del problema y que podemos ignorar. Necesitamos entender qué consume el programa, qué produce, y como se relacionan las entradas con las salidas. Debemos saber, o descubrir, si Scheme provee ciertas operaciones básicas para los datos que nuestro programa está por procesar. Si no, debemos desarrollar programas auxiliares que implementen esas operaciones. Finalmente, una vez tenemos un programa, debemos verificar si de hecho realiza la computación pretendida. Esto podría revelar errores de sintaxis, problemas de ejecución o incluso errores lógicos.

Para brindar algún aroden a este aparente caos, es mejor configurar y seguir una *receta de diseño*, esto es, una prescripción paso a paso de lo que debería hacer y el orden[8] en el cual deberíamos hacer las cosas. Basados en lo que hemos experimentado hasta acá, el desarrollo de un programa requiere al menos las siguientes cuatro actividades:

```
;; Contrato: area-del-anillo : número número -> número

;; Proposito: Calcular el area de un anillo cuyo radio es externo y
;; cuyo agujero tiene un radio interno

;; Ejemplo: (area-del-anillo 5 3) debería producir 50.24

;; Definición: [refina la cabecera]
(define (area-del-anillo externo interno)
   (- (area-del-disco externo)
      (area-del-disco interno)))

;; Tests:
(area-del-anillo 5 3)
;; valor esperado
50.24
```

Figura 2.1. Una receta de diseño: Un ejemplo completo

Entendiendo el Propósito del Programa

El objetivo de diseñar un programa es crear un mecanismo que comsume y produce datos. Por eso empezamos cada desarrollo de programa dando al programa un nombre significativo y estableciendo que clase de información consume y produce. Llamamos esto un *contrato*.

Aquí está como ponemos por escrito un contrato para `area-del-anillo`, uno de nuestros primeros programas[9]:

```
;; area-del-anillo : número número -> número
```

El punto y coma indica que la línea es un *comentario*. El contrato consiste de dos partes. La primera, a la izquierda de los dos puntos establece el nombre del programa. El segundo, a la derecha de los dos puntos, especifica que clase de datos el programa consume y que produce; las entradas están separadas de la salida por una flecha.

Una vez tenemos un contrato, podemos adicionar la *cabecera*. Esta restablece el nombre del programa y da a cada entrada un nombre distinto. Esos nombres son variables (algebráicas) y son referidas como los parámetros del programa[10].

Tomemos un vistazo al contrato y a la cabecera para `area-del-anillo`:

```
;; area-del-anillo : número número -> número
(define (area-del-anillo externo interno) ... )
```

Dice que nos referiremos a la primera entrada como *`externo`* y la segunda como *`interno`*.

Finalmente, usando el contrato y los parámetros, deberíamos formular una corta *declaración de propósito* para el programa, esto es, un breve comentario de *qué* es lo que el programa va a computar. Para la mayoría de nuestro programas, una o dos líneas bastarán. En la medida en que desarrollamos programas más y más grander, podemos necesitar adicionar más información para explicar el propósito del programa.

---

8. Como veremos después, el orden no está completamente fijado. Es posible, y por un número de razones, deseable cambiar el orden de algunos pasos en algunos casos.

9. Una flecha es indicada como un `-` seguido por un `>`.

10. Otros los llaman *argumentos formales* o *variables de entrada*.

Aquí hay un ejemplo del punto de partida de nuestro programa corriente:

```
;; area-del-anillo : número número -> número
;; computa el area de un anillo cuyo radio es externo y
;; cuyo agujero tiene un radio de interno
(define (area-del-anillo externo interno) ...)
```

Pistas: Si el enunciado del problema provee una fórmula matemática, el númeo de distintas variables de la fórmula sugiere cuantas entradas consume el programa.

Para otros problemas de palabras, debemos inspeccionar el problema para separar los hechos dados de lo que es computado. Si algo dado es un número fijo, esto se muestra en el programa. Si es un número desconocido que está para ser fijado por alguién más después, esto es una entrada. La cuestión (o el imperativo) en el enunciado del programa sugiere un nombre para el programa.

Ejemplos de programas:

Para ganar una mejor compresión de lo que un programa debería computar, inventamos ejemplos de entradas y determinamos que salida debería haber. Por ejemplo, `area-del-anillo` debería producir `50.24` para las entradas `5` y `3`, porque es la diferencia entre el área del círculo externo y el área del círculo interno.

Adicionamos ejemplos a la declaración de propósito:

```
;; area-del-anillo : número número -> número
;; computa el area de un anillo cuyo radio es externo y
;; cuyo agujero tiene un radio de interno
;; ejemplo: (area-del-anillo 5 3) debe producir 50.24
(define (area-del-anillo externo interno) ...)
```

Inventar ejemplos – **antes de poner por escrito el cuerpo del programa** – ayuda en muchas formas. Primero, es la única forma segura de descubrir errores con pruebas. Si usamos el programa terminado para inventar ejemplos, estamos tentados a confiar en el programa porque es mucho más fácil ejecutar el rpgorama que predecir lo que hace. Segundo, los ejemplos nos obligan a pensar a través del proceso computacional, que, para los casos complicados que encontraremos después, es crítico para ek desarrollo del cuerpo de la función. Finalmente, los ejemplos ilustran la prosa informal de una declaración de propósito. Los lectores futuros del programa, tales como profesores, colegas o compradores, apreciarán grandemente las ilustraciones de conceptos abstractos.

El cuerpo:

Finalmente, debemos formular el cuerpo del programa. Esto es, **debemos reemplazar el "..." en nuestra cabeza con una expresión**. La expresión computa la respuesta a partir de los parámetros, usando las operaciones básicas de Scheme y programas que ya hemos `definido` o pretendemos `definir`.

Sólamente podemos formular el cuerpo del programa si entendemos cómo el programa calcula[11] la salida partiendo de las entradas dadas. Si la relación entrada-salida es dada como una fórmula matemática, sólo traducimos las matemáticas a Scheme. Si, en cambio, no es dado un problema de palabras, debemos elaborar la expresión cuidadosamente. Para este fin, es útil revisitar los ejemplos del segundo paso y entender *cómo* computamos las salidas para entradas específicas.

En nuestro ejemplo actual, la tarea computacional fue dada vía una fórmula informalmente establecida que reusaba `area-del-circulo`, un programa previamente `definido`. Acá esta la traducción a Scheme:

```
(define (area-del-anillo externo interno)
   (- (area-del-circulo externo)
      (area-del-circulo interno)))
```

Probando:

Después de que hemos completado la definición del programa, aún debemos probar el programa. Al menos, deberíamos asegurar que el programa computa las salidas esperadas para los ejemplos del programa. Para facilitar las pruebas, podemos desear adicionar ejemplos al comienzo de la ventana de `Definiciones` como si ellos fueran ecuaciones. Entonces, cuando hacemos click en el botón `Ejecutar`, ellos son evaluados y vemos si un programa trabaja apropiadamente sobre ellos.

11. Los términos computa y calcla se han usado de modo equivalente como traducciones para el término inglés compute. N del T.

Probar no puede mostrar que un programa produce las salidas correctas para todas las posibles entradas – porque hay típicamente un número infinito de posibles entradas. Pero puede revelar errores de sintaxis, problemas en tiempo de ejecución y errores lógicos.

Para salidas fallidas, debemos poner especial atención a nuestros ejemplos de programas. Es posible que los ejemplos sean incorrectos; que el programa contenga un error lógico; o que tanto los ejemplos como el programa estén incorrectos. En cualquier caso, debemos recorrer pasos a través del desarrollo completo del programa de nuevo.

La figura 2.1 muestra lo que obtenemos cuando desarrollamos un programa de acuerdo a nuestra receta. La figura 2.2 resume la receta en una forma tabular. Debería se consultada siempre que diseñemos un programa.

Fase	Objetivo	Actividad
Propósito del contrato y cabecera	Nombrar la función; especificar sus clases de datos de entrada y sus clases de datos de salida; describir su propósito y formular una cabecera.	Escojer un nombre que satisfage el problema • estudiar el problema para pistas sobre cúantos datos "desconocidos" dados consume la función• seleccionar una variable por entrada; si es posible, usar nombres que son mencionados para los datos "dados" en el enunciado del problema. • describir lo que la función debería producir usando los nombres de las variables escogidas • formular el contrato y la cabecera: ;; *nombre : número ... –> número* ;; computa ... a partir de $x1$ (**define** (*nombre* $x1$...)...)
Ejemplos	Caracterizar la relación entrada-salida vía ejemplos	Buscar el enunciado del problema para los ejemplos • trabajar a través de ejemplos • validar los resultados, si es posible • Inventar ejemplos.
Cuerpo	Definir la función	Formular cómo la función computa sus resultados • desarrollar una expresión de Scheme que use las operaciones primitivas de Scheme, otras funciones, y las variables • traducir las expresiones matemáticas en el enunciado del problema, cuando estén disponibles
Prueba	Descubrir errores ("tipográficos" y lógicos)	Aplicar la función a las entradas de los ejemplos • Verificar que las salidas son como se predijo.

Figura 2.2. La receta de diseño de soslayo

La receta de diseño no es un boleto mágico para los problemas que encontremos durante el diseño de un programa. Provee alguna guía para un proceso que puede a menudo ser abrumante. El paso más creativo y más difícil en nuestra receta se refiere al diseño del cuerpo del programa. En este punto, reposa pesadamente en nuestra habilidad para leer y entender material escrito, en nuestra habilidad para extraer relaciones matemáticas, y en nuestro conocimiento de los hechos básicos. Ninguna de estas habilidades es específica al desarrollo de programas de computadora; el conocimiento que explotamos es específico al dominio de aplicación en el cual estamos trabajando. El resto del libro mostrará qué y cuánta computación puede contribuir a este paso más complicado.

Dominio de Conocimiento: Formular el cuerpo del programa a menudo requiere conocimiento acerca del área, tambien conocida como dominio, desde el cual es problema es esbozado. Esta forma de conocimiento es llamada *dominio de conocimiento*. Puede tener que ser esbozado de las matemáticas simples, tales como la aritmética, de las matemáticas complejas, tales como las ecuaciones diferenciales, o de disciplinas no matemáticas: música, biología, ingeniería civil, arte, y otras por el estilo.

Porque los programadores no pueden conocer todos los dominios de aplicación de la computación, ellos debe estar preparados para entender el lenguaje de una variedad de áreas de aplicación de modo que ellos pueda discutir los problemas con expertos del dominio. El lenguaje es a menudo el de las matemáticas, pero en algunos casos, los programadores deben inventar un lenguaje, especialmente un lenguaje de datos para el área de aplicaicón. Por esa razón, es imperativo que los programadores tenga una sólida comprensión de las posibilidades completas de los lenguajes de programación.