**Introduction**

What is Scheme? (Hunk A)

Basic Scheme Features

Code Consists of Expressions

Parenthesized Prefix Expressions

Expressions Return Values, But May Have Side-Effects

Defining Variables and Procedures

Most Operators are Procedures

Definitions vs. Assignments

Special Forms

Control Structures are Expressions

The Boolean Values #t and #f

Some Other Control-Flow Constructs: cond, and, and or

cond

and and or

not is just a procedure

**Comments (Hunk C)**

A Note about Parentheses and Indenting

Let Your Editor Help You

Indenting Procedure Calls and Simple Control Constructs

Indenting cond

Indenting Procedure Definitions

All Values are Pointers to Objects

All Values are Pointers

Most Implementations Optimize Away Many Pointers

Objects on the Heap

Scheme Reclaims Memory Automatically

Objects Have Types, Variables Don't

Dynamic typing

**The Empty List (Hunk E)**

Pairs and Lists

cdr-linked lists

Lists and Quoting

Where the Empty List Got its Name

Some Handy Procedures that Operate on Lists

length

list

append

reverse

member

Recursion Over Lists and Other Data Structures

length

Copying Lists

append and reverse

append

reverse

**Type and Equality Predicates (Hunk G)**

Type Predicates

Equality Predicates

**Choosing Equality Predicates (Hunk I)**

Quoting and Literals

Simple Literals and Self-Evaluation

Local Variables and Lexical Scope

let

Indenting let Expressions

Lexical Scope

Binding Environments and Binding Contours

Block Structure Diagrams for lets

let\*

**Procedures (Hunk K)**

Procedures are First Class

Higher-Order Procedures

Anonymous Procedures and lambda

**lambda and Lexical Scope (Hunk M)**

Local Definitions

Recursive Local Procedures and letrec

Multiple defines are like a letrec

Variable Arity: Procedures that Take a Variable Number of Arguments

apply

Variable Binding Again

Identifiers and Variables

Variables vs. Bindings vs. Values

**Tail Recursion (Hunk O)**

Macros

Continuations

Iteration Constructs

Discussion and Review

Using Scheme (A Tutorial)

**An Interactive Programming Environment**

**(Hunk B)**

Starting Scheme

Making mistakes and recovering from them

Returns and Parentheses

Interrupting Scheme

Exiting (Quitting) Scheme

Trying Out More Expressions

Booleans and Conditionals

Sequencing

Other Flow-of-control Structures

Using cond

Using and and or

**Making Some Objects (Hunk D)**

Lists (Hunk F)

Using Predicates (Hunk H)

Using Type Predicates

Using Equality Predicates

Local Variables, let, and Lexical Scope (Hunk J)

Using First-Class, Higher-Order, and Anonymous Procedures (Hunk L)

First-Class Procedures

Using and Writing Higher-Order Procedures

Interactively Changing a Program (Hunk N)

Replacing Procedure Values

Loading Code from a File

Loading and Running Whole Programs

Some Other Useful Data Types

Strings

Symbols

A Note on Identifiers

Lists Again

Heterogeneous Lists

Operations on Lists

**Basic Programming Examples (Hunk P)**

An Error Signaling Routine

map and for-each

map

for-each

member and assoc, and friends

member, memq, and memv

assoc, assq, and assv

Procedural Abstraction

Procedure Specialization

Procedure Composition

Currying

Discussion and Review

Writing an Interpreter

Interpretation and Compilation

Implementing a Simple Interpreter

The Read-Eval-Print Loop

The Reader

Implementing read

Implementing the read procedure

Comments on the Reader

Recursive Evaluation

Comments on the Arithmetic Evaluator

A Note on Snarfing and Bootstrapping

Snarfing

Bootstrapping and Cross-compiling

Improving the Simple Interpreter

Implementing top-level variable bindings

Running the improved interpreter

Discussion and Review

**Environments and Procedures**

Understanding let and lambda

let

Lambda

Procedures are Closures

Lambda is cheap, and Closures are Fast

An Interpreter with let and lambda

Nested Environments and Recursive Evaluation

Integrated, Extensible Treatment of Special Forms

Interpreting let

Variable References and set!

Interpreting lambda and Procedure Calling

Mutual Recursion Between Eval and Eval-apply

Variants of let: letrec and let\*

Understanding letrec

Using letrec and lambda to Implement Modules

let\*

Iteration Constructs

Named let

Programming with Procedures and Environments

Exercises

Recursion in Scheme

**Subproblems and Reductions (non-tail and tail calls)**

The Continuation Chain

Exploiting Tail Recursion

Passing Intermediate Values as Arguments

Summing a List

Implementing length tail-recursively

reduce

Iteration as Recursion

named let

do

Quasiquotation and Macros

quasiquote

unquote-splicing

**Defining New Special Forms**

Macros vs. Procedures

Implementing More Scheme Special Forms

let

let\*

cond

Discussion

Lisp-style Macros

Ultra-simple Lispish Macros

Better Lisp-style Macros

Problems With Lisp-Style Macros

Ugly Hacks Around Name Conflicts

Implementing Simple Macros and Quasiquote

Implementing Simple Macros

Implementing quasiquote and unquote

Translating backquotes to quasiquote

quasiquote

define-rewriter

define-macro

Procedural Macros vs. Template-filling Macros

Programming Examples Using Macros

**Records and Object Orientation**

Records (Structures)

Using Procedural Abstraction to Implement Data Abstraction

Automating the Construction of Abstract Data Types with Macros

Simple Uses of OOP Objects

Late Binding

Class Definitions and Slot Specifications

Generic Procedures and Methods

Generic Procedures and Classes are First-Class

Implementing the Simple Object System

Implementing define-class

class <<class>>

Implementing define-generic

Implementing define-method

Installing Accessor Methods

Keyword options

Inheritance

Overriding and Refining Inherited Methods

Late Binding and Inheritance

Implementing an Object System with Inheritance

Interfaces and Inheritance

A More Advanced Object System and Implementation

Language Features

Purity

Encapsulation

Multiple Dispatching

Multiple Inheritance

Explictit Subtyping

Control Over Compilation

A Metaobject Protocol

Implementation Improvements

Factoring out Work at Compile Time

Supporting Runtime Changes

Faster Dynamic Dispatching

Compiling Slot Accessors And Methdos Inline

Exploiting Type Information

**Advanced Compilation Techniques**

Some Shortcomings of Standard Scheme for Object System Implementation

Inability to Define Disjoint Types

Lack of Type Objects for Predefined Types

Lack of Weak Tables and Extensible Closure Types.

Standard Macros are Limited

Unspecified Time of Macro Processing

Lack of Type Declarations

Lack of a Standard bound? procedure

Other Useful Features

Special Forms

Input-Output Facilities

read and write

display

Ports

with-input-\dots{} Forms

**Useful Types and Associated Procedures**

Numeric Types

Floating-Point Numbers

Arbitrary-Precision Integers

Ratios

Coercions and Exactness

Vectors

Strings and Characters

call-with-current-continuation

Implementing a Better Read-Eval-Print Loop

Implementing Catch and Throw

Implementing Backtracking

Implementing Coroutines

Implementing Cooperative Multitasking

Caveats about call-with-current-continuation

**A Simple Scheme Compiler**

What is a Compiler?

What is an Interpreter?

OK, so what's a compiler?

What Does a Compiler Generate?

Basic Structure of the Compiler

Data Representations, Calling Convention, etc.

The Registers

The Evaluation Stack (or Eval Stack, for short)

The Continuation Chain

Environments

Closure Representation and Calling

Continuations

Applying a Procedure Doesn't Save the Caller's State

Continuation Saving

An Example

Generating Unique Labels

More on Representations of Environments

Compiling Code for Literals

Compiling Code for Top-Level Variable References

Precomputing Local Variable Lookups using Lexical Scope

Lexical Addressing and Compile-Time Environments

A Detailed Example

Preserving Tail-Recursiveness using Compile-Time Continuations

When Should We Save Continuations?

Compiling Returns

Compiling Top-Level Expressions

Compiling lambda Expressions Inside Procedures

Compiling Top-level Definitions

Interfacing to the Runtime System

Garbage Collection

Safe Points

GC at Any Time

Interrupts

**Advanced Compiler and Runtime System Techniques**

Inlining Small Procedures

Type Declarations and Type Analysis

Using More Hardware Registers

Closure Analysis

Register Allocating Loop Variables for Loops

Conventional Optimizations

Stack Caches