An Efficient Substitution Model Scheme
Evaluator

by

Abraham Aaron Maritime

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

© Abraham Aaron Maritime, MCMXCVII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author .................................................................
Department of Electrical Engineering and Computer Science May 23, 1997

Certified by...........................................................
Albert Meyer
Hitachi America Professor of Engineering
Thesis Supervisor

Accepted by ..........................................................
Arthur C. Smith
Chairman, Department Committee on Graduate Students
An Efficient Substitution Model Scheme Evaluator

by

Abraham Aaron Maritime

Submitted to the Department of Electrical Engineering and Computer Science on May 23, 1997, in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering

Abstract

In this thesis, I constructed a Scheme evaluator with another student, Paul Van Eyk. The evaluator that we designed emulates substitution model evaluation with renaming. It is actually a tail recursive environment model evaluator that has explicit control of its own stack. We provided a front end to the evaluator that maps its current state to a valid Scheme expression. Our evaluator emulates a substitution model evaluator by performing this mapping at regular steps during the process of interpreting a Scheme expression. The act of writing a Scheme evaluator is a very large project, and while we have implemented one that follows the Scheme reference guidelines, there are many more features we could have added. This thesis is written for the students that may continue our work. It details the work we have done, the motivation behind it, and also what work remains to be done.

Thesis Supervisor: Albert Meyer
Title: Hitachi America Professor of Engineering
Acknowledgments

I would like to thank my partner Paul Van Eyk for not only his help in writing this evaluator, but more for his great sense of humor and his incessant baseball anecdotes. I also would like to thank Albert for allowing us to work on this project and his willingness to set his other work aside to help us when we were stuck.
## Contents

1 Introduction .................................................. 7
   1.1 Motivation for this Evaluator .......................... 8
   1.2 Scheme .................................................. 8
   1.3 Goals of the Evaluator ................................. 9

2 The Substitution Model Evaluator .......................... 10
   2.1 The Parser .............................................. 10
      2.1.1 Variables ......................................... 11
      2.1.2 Conditionals ...................................... 11
      2.1.3 Cond-statements ................................... 11
      2.1.4 Case-statements ................................... 12
      2.1.5 And-statements ..................................... 12
      2.1.6 Or-statements ...................................... 12
      2.1.7 Lambdas ............................................ 12
      2.1.8 Let-statements ..................................... 13
      2.1.9 Let*-statements ................................... 13
      2.1.10 Letrec-statements ................................. 13
      2.1.11 Definitions ....................................... 14
      2.1.12 Set!-statements .................................. 14
      2.1.13 Sequences ......................................... 14
      2.1.14 Quotations ........................................ 14
      2.1.15 Combinations ..................................... 14
   2.2 The Tail Recursive Evaluator ........................... 15
Chapter 1

Introduction

Substitution model Scheme evaluators follow a set of basic rules to evaluate Scheme expressions. Every step in the evaluation involves parsing the expression to determine which subexpression should be evaluated first, and then applying the appropriate rule to the innermost subexpression. A tail recursive interpreter uses a more sophisticated method for evaluating Scheme. By maintaining a continuation stack, the evaluator eliminates the cost of parsing the entire expression at each step. This thesis proposes a method of mapping the current state of a tail recursive evaluator to a Scheme expression (or S-expression). By stopping a tail recursive evaluator at regular intervals and displaying the current S-expression being evaluated, this mapping simulates a substitution model evaluator.

Chapter two describes how the interpreter works, including information on both the tail recursive evaluator and the mapping that converts its state to an S-expression.

Chapter three describes how to use the interpreter. Included in this chapter is an explanation of what parts of Scheme have been implemented and what control settings the interpreter provides to control output display.

Chapter four details the gain in efficiency of the Scheme evaluator presented in this thesis over a more conventional substitution model Scheme evaluator.

Chapter five details the work that has not yet been completed. This is intended to be used by future students to further advance the work started by this thesis.
1.1 Motivation for this Evaluator

Structures and Interpretations of Computer Programs (known as 6.001) is an introductory level class for electrical engineers and computer scientists at the Massachusetts Institute of Technology. It is an introduction to the theory behind computer programming as opposed to the actual practice of memorizing syntax. The class has traditionally been taught in the language Scheme. Scheme is a language with a simple syntax and with simple rules to evaluate an expression.

In 6.001, Scheme is initially introduced as a functional language. The students are taught to evaluate expressions by following a set of rules called the substitution model. As the semester progresses, the class introduces mutation, various means of scoping, and continuations. When all of these factors are considered, it becomes much simpler to build a Scheme interpreter using an environment model evaluator that has explicit control of its own continuation than to attempt to build one using the substitution model.

All production level evaluators are constructed by using tail recursive environment model evaluators, as they are much more efficient than substitution model evaluators. As a result of this, students in 6.001 are taught the substitution model, but are not provided with any evaluator that actually performs it. The task of constructing a substitution model evaluator has been completed by Professor Albert Meyer, MIT [7]. However, that interpreter is very slow and incapable of evaluating expressions that have a large number of steps. Other attempts to produce substitution model evaluators are also being performed at other institutes [8].

1.2 Scheme

This thesis assumes a good understanding of the Scheme language. There are several good texts available to explain any questions regarding the language or how it is evaluated. Before reading this thesis, one should understand Scheme very well, along with the various models to evaluate it. This thesis is based on “6.001 Scheme” and
“MIT Scheme,” both of which have reference manuals available [2] [4]. A less in depth explanation of the evaluator presented in this thesis directed at a student’s level of understanding can be found in Paul Van Eyk’s Master of Engineering Thesis [9]. Information on different models of evaluation can be found in both Van Eyk’s Masters project and in Structures and Interpretations of Computer Programs [1]. Many of the advanced theories that this rewriting model is based on can be found in articles by Martin Hofmann [6] and Matthias Felleisen [3], and in class notes provided by Robert Harper [5] and Meyer [7]. These articles provided a great deal of depth into the subject.

1.3 Goals of the Evaluator

The goal of this thesis is to design a substitution model evaluator that is built on top of a tail recursive evaluator in order to design a more efficient final product. The substitution model evaluator should print out a series of S-expressions. Each S-expression should be derived through a series of steps from the one above it. Every individual line printed out by the evaluator should be a valid expression that if passed to a Scheme evaluator would produce the same output as any line that the evaluator prints out.

The thesis could not possibly undertake the task of implementing a full Scheme evaluator, but the evaluator does implement all of the syntax needed by students taking Structures and Interpretations of Computer Programs with the possible exception of quasiquote.
Chapter 2

The Substitution Model Evaluator

This chapter of the thesis explains the code of the substitution model evaluator. The first section describes the parser. The second section describes the tail recursive evaluator. The third section explains the mapping from the state of the evaluator to a Scheme expression.

The evaluator as a whole works as follows. The read-eval-print loop reads in an expression from the user. The read-eval-print loop is in the file doeval.scm. This expression is passed to parse to put it into the correct syntax. The code for the parser is in the file parse.scm. This syntaxed expression is passed to the tail recursive evaluator. The tail recursive evaluator is broken up into two parts, located in files treval.scm and next-cep.scm. The tail recursive evaluator evaluates the Scheme expression, but stops periodically to print out the state of the evaluator to simulate substitution model evaluation. The procedures to print out the current state are broken up into three parts, located in print-cep.scm, bindings.scm, and unparse.scm.

2.1 The Parser

A parser was created for the evaluator because the existing Scheme syntax is inadequate to perform substitution model evaluation. Scheme’s parser transforms many special forms into conditionals, thus making it impossible to determine whether an expression is an and, a cond, a case, or an if. This ambiguity makes it impossible
to do accurate substitution model evaluation. The added syntax in the evaluator
designed in this thesis causes the dispatch loop of the evaluator to be slower, but the
primary goal of this evaluator is to evaluate Scheme expression using the substitution
model, with the secondary goal of doing it efficiently.

The code for the parser is contained completely in parse.scm. The procedure
parse takes a Scheme expression and converts it into the syntax of the evaluator. The
syntax of this evaluator contains most of syntax of the underlying evaluator, but it
has been expanded to contain many new objects. These will all be detailed in this
section.

The parser works by dispatching on the type of expression. If the expression is
a list, then it calls the appropriate procedure to parse the expression based on what
the first item of that list is. Otherwise, it calls the procedure to make a variable if
the expression is a symbol, or simply returns the expression if it is self-evaluating.

The various syntax objects and procedures to create them are outlined below.

2.1.1 Variables

A variable is part of the underlying Scheme syntax. A variable has only one field, a
name. It is not possible to make a malformed variable.

2.1.2 Conditionals

An expression is considered a conditional if it begins with the symbol if. Conditionals
are part of the built in underlying Scheme syntax. A conditional has either two or three
fields; a predicate, a consequent, and optionally an alternative. To parse a conditional,
the evaluator checks to see if the if has either two or three subexpressions, and either
creates an if by parsing the subexpressions or returns an error.

2.1.3 Cond-statements

An expression is considered a cond if it begins with the symbol cond. Conds are not
part of the underlying Scheme. A cond has only one field, a list of clauses. To parse a
cond expression, the parser ensures that all clauses are well formed and that if there is an else expression that it is the last expression. The parser parses all the clauses and makes elses and receivers (to handle =>) where appropriate, and then forms the cond.

2.1.4 Case-statements

An expression is considered a case if it begins with the symbol case. Cases are not part of the underlying Scheme evaluator. A case has two fields, a key and a list of clauses. To parse a case expression, the parser must check for several possible errors. The case must have at least one subexpression. Every subexpression after the first must be a well-formed clause, and there must be no duplicates in the lists of objects in the clauses. Also, if there is an else, it must be in the last clause. If the case is well formed, then the parser creates a case statement.

2.1.5 And-statements

An expression is considered an and if it begins with the symbol and. Ands are not part of the underlying Scheme evaluator. Ands have one field, a list of subexpressions. It is impossible to make a malformed and, so the parser does not need to check for any errors.

2.1.6 Or-statements

An expression is considered an or if it begins with the symbol or. Ors are not part of the underlying Scheme evaluator. Ors have one field, a list of subexpressions. It is impossible to make a malformed or, so the parser does not need to check for any errors.

2.1.7 Lambdas

An expression is considered a lambda if it begins with the symbol lambda. Lambdas are part of the existing Scheme syntax. A lambda has seven fields; a name, required
parameters, optional parameters, the rest parameter, auxiliary parameters, declarations, and a body. This evaluator only uses the required parameters, rest parameter, and body fields. The evaluator does not use the name field, which is provided for users to aid in debugging. Such a feature is not necessary in a substitution model evaluator. This evaluator also does not handle optional parameters. Auxiliary parameters are for converting internal defines into assignments, which this evaluator does not do, so they are also omitted. Declarations are also not yet supported by this evaluator. To parse a lambda, the parser must check to see if there are at least two subexpressions. It must break up the first one into the various parameter fields, and the remaining ones must be parsed and inserted into the body field.

2.1.8 Let-statements

An expression is considered a let if it begins with the symbol let. Lets are not part of the underlying evaluator. A let has three fields; a name, a list of bindings, and a body. A let expression must have at least two subexpressions. The parser checks to see if a name is provided for the let. It then checks to see that the bindings are well formed, and then packages up the rest of the expressions into the body.

2.1.9 Let*-statements

An expression is considered a let* if it begins with the symbol let*. Let*s are not part of the underlying Scheme evaluator. Let*s are parsed like lets with the exception being that let*s cannot be named.

2.1.10 Letrec-statements

An expression is considered a letrec if it begins with the symbol letrec. Letrecs are not part of the underlying Scheme evaluator. Letrecs are parse identically to lets.
2.1.11 Definitions

An expression is considered a definition if it begins with the symbol `define`. Definitions are part of the underlying Scheme syntax. A definition has two fields, a name and a value. A define must be in one of three forms. It can either have one subexpression, it can have two subexpressions where the first is a symbol, or it can have two or more subexpressions where the first is a list of symbols. The parser parses the three types of defines accordingly.

2.1.12 Set!-statements

An expression is considered a set! if it begins with the symbol `set!`. Set!s are not part of the underlying Scheme evaluator. A set! has two fields, a name and a value. The parser checks to make sure that a set! has two subexpressions, where the first must be a symbol.

2.1.13 Sequences

An expression is considered a sequence if it begins with the symbol `begin`. Sequences are part of the underlying Scheme syntax. A sequence has one field, a list of expressions. A sequence must have at least one subexpression.

2.1.14 Quotations

An expression is considered a quotation if it begins with the symbol `quote`. Quotes are not evaluated, so the parser just returns the text of the quotation unparsed.

2.1.15 Combinations

An expression is considered a combination if it matches none of the above special forms. A combination is part of the underlying Scheme syntax and has two fields, an operator and a list of operands. The only rule for combinations is that there must be at least one subexpression.
2.2 The Tail Recursive Evaluator

The tail recursive evaluator follows the general format of any evaluator that maintains explicit control over its own continuation.

The state of the evaluator is contained in CE-packages. A CE-package represents the current state of the machine. It is the current expression, environment, and continuation. An expression is some piece of syntax. Environments are represented using the underlying Scheme's environment representation. A continuation is a Scheme object with four fields; a tag, a data field, an environment, and a continuation. A data field is itself an object with two fields, an evaluated field and an unevaluated field.

When given a CE-package, treval (the tail recursive evaluator) dispatches to the appropriate procedure which creates a new CE-package. The new expression will be the first subexpression that needs to be evaluated. The new environment will be the environment that subexpression should be evaluated in. The new continuation will contain four things. It will have a tag stating what kind of expression was passed to treval. It will have a data field containing the subexpressions that have and have not been evaluated. It will also save the environment and continuation passed to it. If treval is passed an evaluated expression, then it must refer to the continuation stack to decide what to do with that value. It does this by calling next-cep.

Next-cep (short for next CE-package) takes a CE-package. It refers to the tag of the continuation in the CE-package to decide what to do with the value that it was passed in the CE-package. It creates a new CE-package based on the information provided in the old continuation.

Whenever an expression is evaluated and returns a value, print-cep is called. Print-cep is detailed in the next section.

The following is an explanation as to how the syntax is evaluated.
2.2.1 Combinations

Treval

If the expression in the CE-package passed to treval is a combination, then treval creates a CE-package as follows. A list of the subexpressions in reverse order is made. The expression is the first item of that list. The environment is the environment of the old CE-package. The continuation is a new continuation with the following values. It is tagged \texttt{comb}. Its unevaluated data is the remainder of that list. It has no evaluated data. Its environment and continuation are the same as those from the old CE-package.

Next-cep

If next-cep finds a comb on the continuation of the CE-package passed to it, it forms a CE-package as follows.

If there are no remaining subexpressions in the unevaluated data field in the continuation of the old CE-package; then the evaluated data field of that continuation is the list of operands, and the expression field of the old CE-package is the operator. If the operator is a compound procedure, then a new environment is created whose enclosing environment is the environment of that procedure which binds the parameters of that procedure to the list of operands. The expression of the new CE-package is the body of that compound procedure. The environment is that newly created environment. The continuation is the continuation in the continuation of the old CE-package. If the operator is not a compound procedure, then the expression of the new CE-package is the result of applying the operator to the operands. The environment and continuation are the environment and continuation in the continuation of the old CE-package.

If there are still unevaluated subexpressions in the unevaluated data field in the continuation of the old CE-package, then the expression of the new CE-package is the first unevaluated subexpression. The environment is the environment in the continuation of the old CE-package. The continuation is a new continuation with
the following values. It is tagged comb. Its unevaluated data is the remainder of the unevaluated data. Its evaluated data is the list formed by adding the value in the expression field of the old CE-package onto the list of evaluated data in the continuation of the old CE-package. Its environment and continuation are the same as those from the continuation in the old CE-package.

2.2.2 Variables

Treval

If the expression in the CE-package passed to treval is a variable, then treval makes a new CE-package as follows. The new expression is the lookup of the variable in the environment of the old CE-package. The environment and continuation fields of the new CE-package will be the same as the ones in the old CE-package.

Next-cep

There is no continuation tag associated with variables.

2.2.3 Conditionals

Treval

If the expression in the CE-package passed to treval is a conditional, then treval makes a new CE-package as follows. The new expression is the predicate of the if. The new environment is the same as the environment of the old CE-package. The new continuation is the continuation with the following properties. It is tagged if. Its unevaluated data is the list of the consequent and the alternative. It has no evaluated data. Its environment is the environment of the old CE-package. Its continuation is the continuation of the old CE-package.
Next-cep

If next-cep finds an if on the continuation of the CE-package passed to it, it forms a new CE-package as follows. The new expression is the consequent from the unevaluated data of the old CE-package’s continuation if the value in the expression part of the old CE-package is not false. If the value is false, then the new expression is the alternative from the unevaluated data of the old CE-package. The environment and continuation of the new CE-package are the same as the ones in the continuation of the old CE-package.

2.2.4 Cond-statements

Treval

If the expression in the CE-package passed to treval is a cond, then treval makes a new CE-package as follows.

If the cond has no clauses, then the new CE-package’s expression is the unspecified-return-value. Its environment and continuation are the same as the ones in the old CE-package.

If the cond’s first clause is of the form (else expr), then the new CE-package’s expression is expr. Its environment and continuation are the same as the ones in the old CE-package.

Otherwise, the cond is of the form (cond (expr1 expr2) clause2 clause3 ...). The expression of the new CE-package created for this case is expr1. The environment is the environment of the old CE-package. The continuation is a new continuation with the following properties. It is tagged cond. Its unevaluated data is the list (expr2 clause2 clause3 ...). It has no evaluated data. Its environment and continuation are the same as the ones in the old CE-package.

Next-cep

If next-cep finds a cond on the continuation of the CE-package passed to it, it forms a new CE-package as follows.
If the value of the expression part of the old CE-package is not false, the expression part of the new CE-package is expr2 taken from the unevaluated data field of the old CE-package’s continuation. The environment and continuation of the new CE-package are the same as the environment and continuation from the continuation of the old CE-package.

If the value of the expression part of the old CE-package is false, next-cep acts as if it were treval evaluating the CE-package with the following properties. The expression is (cond clause2 clause3 ...), where (clause2 clause3 ...) are taken from the unevaluated data field of the old CE-package’s continuation. The environment and continuation are the environment and continuation taken from the old CE-package’s continuation.

2.2.5 Case-statements

Treval

If the expression in the CE-package passed to treval is a case, then treval creates a new CE-package as follows. The expression part of the CE-package is the key of the case statement. The environment is the environment of the old CE-package. The continuation is a new continuation created with the following values. The new continuation is tagged case. Its unevaluated data is a list of the clauses of the case statement. It has no evaluated data. Its environment and continuation are the same as those of the old CE-package.

Next-cep

If next-cep finds a case on the continuation of the CE-package passed to it, it forms a CE-package as follows. Next-cep scans through all of the lists of objects to find a match with the value passed to it in the expression part of the CE-package. It uses eqv? to test for the match. If a match is found, the new expression is the expression of the matching clause. If no match is found, the new expression is the unspecified return value. In either case, the new environment and continuation are the same as
the environment and continuation in the continuation of the old CE-package.

2.2.6 And-statements

Treval

If the expression in the CE-package passed to treval is an and, then treval creates a new CE-package as follows.

If the and has no subexpressions, then the expression of the new CE-package is true. The environment and continuation are the same as that of the old CE-package.

If the and has subexpressions, then the expression of the new CE-package is the first subexpression of the and. The environment is the same environment of the old CE-package. The continuation is a new continuation with the following values. The continuation is tagged and. Its unevaluated data is a list of the remaining clauses of the and. It has no evaluated data. Its environment and continuation are the same as that of the old CE-package.

Next-cep

If next-cep finds an and on the continuation of the CE-package passed to it, it forms a CE-package as follows.

If either there are no remaining expressions to evaluate in the unevaluated data field of the old CE-package's continuation or the expression in the old CE-package is the value false; then the expression of the new CE-package is the expression of the old CE-package. The environment and continuation of the new CE-package are the environment and continuation in the continuation of the old CE-package.

Otherwise, the expression for the new CE-package is the first expression in the unevaluated data field of the old CE-package's continuation. The environment of the new CE-package is the environment in the continuation of the old CE-package. The continuation of the new CE-package will be a new continuation with the following values. It will be tagged and. Its unevaluated data field will be the remaining expressions after stripping off the first one. It will have no evaluated data. Its continuation
and environment will be the continuation and environment in the continuation of the old CE-package.

### 2.2.7 Or-statements

**Treval**

If the expression in the CE-package passed to treval is an or, then treval creates a new CE-package as follows.

If the or has no subexpressions, then the expression of the new CE-package is false. The environment and continuation are the same as that of the old CE-package.

If the or has subexpressions, then the expression of the new CE-package is the first subexpression of the or. The environment is the same environment of the old CE-package. The continuation is a new continuation with the following values. The continuation is tagged or. Its unevaluated data is a list of the remaining clauses of the or. It has no evaluated data. Its environment and continuation are the same as that of the old CE-package.

**Next-cep**

If next-cep finds an or on the continuation of the CE-package passed to it, it forms a CE-package as follows.

If either there are no remaining expressions to evaluate in the unevaluated data field of the old CE-package’s continuation or the expression in the old CE-package is not the value false; then the expression of the new CE-package is the expression of the old CE-package. The environment and continuation of the new CE-package are the environment and continuation in the continuation of the old CE-package.

Otherwise, the expression for the new CE-package is the first expression in the unevaluated data field of the old CE-package’s continuation. The environment of the new CE-package is the environment in the continuation of the old CE-package. The continuation of the new CE-package will be a new continuation with the following values. It will be tagged or. Its unevaluated data field will be the remaining expres-
sessions after stripping off the first one. It will have no evaluated data. Its continuation and environment will be the continuation and environment in the continuation of the old CE-package.

2.2.8 Lambdas

Treval

If the expression in the CE-package passed to treval is a lambda, then treval creates a new CE-package whose expression is the procedure returned by evaluating the lambda expression in the environment of the old CE-package. The environment and continuation of the new CE-package are the same as that of the old CE-package.

Next-cep

There is no continuation tag associated with lambdas.

2.2.9 Let-statements

Treval

If the expression in the CE-package passed to treval is a let, then treval creates a new CE-package as follows.

- If it is not a named let, and the let has no bindings; then the expression of the new CE-package is the body of the let. The environment of the new CE-package is a new environment created with the following properties. Its parent environment is the environment of the old CE-package. It has no variables bound. The continuation of the new CE-package is the continuation of the old CE-package.

- If it is a named let, and it has no bindings; then the expression of the new CE-package is the body of the let. The environment of the new CE-package is a new environment created as follows. An empty environment is created whose parent environment is the environment of the old CE-package. A procedure of no arguments whose body is the body of the let is created in that new empty environment. The
name of the let is bound to that procedure in the new empty environment. The continuation of the new CE-package is the continuation of the old CE-package.

If the let has bindings to evaluate, then the expression of the new CE-package is the expression part of the first binding in the let. The environment of the new CE-package is the same environment in the old CE-package. The continuation of the new CE-package is a new continuation with the following properties. It will be tagged let. Its unevaluated data is a list of a list of the remaining bindings, the let’s name (or simply false if it does not have one), and the body of the let. Its evaluated data is the list of the variable name in the first binding. Its environment and continuation are the environment and continuation of the old CE-package.

Next-cep

If next-cep finds a let on the continuation of the CE-package passed to it, it forms a CE-package as follows.

First, the evaluated bindings list is assembled from the evaluated data field of the continuation in the old CE-package and the value in the expression field of the old CE-package. An unevaluated bindings list, a let name, and a let body are also all extracted from the unevaluated data field of the continuation in the old CE-package.

If the unevaluated bindings list is empty and the let name is the value false, then the expression of the new CE-package is the let body. The environment of the new CE-package is a new environment whose enclosing environment is the environment in the continuation in the old CE-package. The new environment binds all of the variables in the evaluated bindings list to their associated values. The continuation of the new CE-package is the continuation contained in the continuation of the old CE-package.

If the unevaluated bindings list is empty and the let name is not the value false, then the expression of the new CE-package is the let body. The environment of the new CE-package is created as follows. A new environment is created binding all of the variables in the evaluated bindings list to their associated values. The enclosing environment of that new environment is the environment in the continuation in the new CE-package.
old CE-package. A procedure whose arguments are the list of variables from the evaluated bindings and whose body is the let body is created in the newly created environment. A binding is then added to the new environment from the name of the let to that newly created procedure. The continuation of the new CE-package is the continuation contained in the continuation of the old CE-package.

If the unevaluated bindings list is not empty, then the expression of the new CE-package is the expression part of the first binding in the unevaluated bindings list. The environment of the new CE-package is the same environment from the continuation in the old CE-package. The continuation is a new continuation with the following properties. It is tagged let. Its unevaluated data field is a list of a list of the remaining bindings, the let name, and the let body. The evaluated data field is a list formed by adding the variable name part of the first unevaluated binding to the list of evaluated bindings. The environment and continuation of the new CE-package will be the environment and continuation of the old CE-package's continuation.

### 2.2.10 Let*-statements

**Treval**

If the expression in the CE-package passed to treval is a let*, then treval creates a new CE-package as follows.

If the let* has no bindings, then the expression of the new CE-package is the body of the let*. The environment of the new CE-package is a new environment created with the following properties. Its parent environment is the environment of the old CE-package. It has no variables bound. The continuation of the new CE-package is the continuation of the old CE-package.

If the let* has bindings to evaluate, then the expression of the new CE-package is the expression part of the first binding in the let*. The environment of the new CE-package is the same environment in the old CE-package. The continuation of the new CE-package is a new continuation with the following properties. It will be tagged let*. Its unevaluated data is a list of a list of the remaining bindings and the body of
the let. Its evaluated data is the variable name in the first binding. Its environment and continuation are the environment and continuation of the old CE-package.

**Next-cep**

If next-cep finds a let* on the continuation of the CE-package passed to it, it forms a CE-package as follows.

First, the evaluated binding is assembled from the evaluated data field of the continuation in the old CE-package and the value in the expression field of the old CE-package. An unevaluated bindings list and a let* body are also both extracted from the unevaluated data field of the continuation in the old CE-package. A new environment is then constructed. Its enclosing environment is the environment in the continuation of the old CE-package. Its only binding is the evaluated binding from above.

If the unevaluated bindings list has no bindings left to be evaluated, then the expression part of the new CE-package is the let* body. The environment is the newly created environment from above. The continuation is the same continuation as the one in the continuation of the old CE-package.

If there are bindings that remain to be evaluated in the unevaluated bindings list, then the expression of the new CE-package is the expression part of the next unevaluated binding. The environment is the newly created environment from above. The continuation of the new CE-package is a new continuation with the following properties. It is tagged let*. Its unevaluated data field is a list of the remaining bindings and the let* body. Its evaluated data field is the variable name in the first binding in the unevaluated bindings list. Its environment is the newly created environment from above. Its continuation is the continuation from the continuation of the old CE-package.
2.2.11 Letrec-statements

Treval

If the expression in the CE-package passed to treval is a letrec, then treval creates a new CE-package as follows.

If the letrec has no bindings, then the expression of the new CE-package is the body of the letrec. The environment of the new CE-package is a new environment created with the following properties. Its parent environment is the environment of the old CE-package. It has no variables bound. The continuation of the new CE-package is the continuation of the old CE-package.

If the letrec has bindings to evaluate, then a new environment is created. The new environment’s enclosing environment is the environment in the continuation of the old CE-package. The new environment has bindings from all of the variable names in the letrec’s binding list to “bad values.” The expression of the new CE-package is the expression part of the first binding in the binding list. The environment is the newly created environment above. The continuation is a new continuation created with the following properties. It is tagged letrec. Its unevaluated data field is a list of the remaining bindings and the body of the letrec. Its evaluated data field is a list containing the variable name of the first binding in the bindings list. Its environment and continuation are the same as the ones in the continuation of the old CE-package.

Next-cep

If next-cep finds a letrec on the continuation of the CE-package passed to it, it forms a CE-package as follows.

First, the evaluated binding is assembled from the evaluated data field of the continuation in the old CE-package and the value in the expression field of the old CE-package. An unevaluated bindings list and a letrec body are also both extracted from the unevaluated data field of the continuation in the old CE-package. The environment in the continuation in the old CE-package is then changed so that the variable in the newly created evaluated binding now points to the value of that binding.
If the unevaluated bindings list is empty, the expression of the new CE-package is the letrec body. The environment of the new CE-package is the environment in the continuation of the old CE-package. The continuation of the new CE-package is the continuation contained in the continuation of the old CE-package.

If the unevaluated bindings list is not empty, then the expression of the new CE-package is the expression part of the first binding in the unevaluated bindings list. The environment of the new CE-package is the same environment from the continuation of the old CE-package. The continuation is a new continuation with the following properties. It is tagged letrec. Its unevaluated data field is a list of a list of the remaining bindings and the let body. The evaluated data field is a list formed by adding the variable name part of the first unevaluated binding to the list of evaluated bindings. The environment and continuation of the new CE-package will be the environment and continuation of the old CE-package’s continuation.

2.2.12 Definitions

Treval

If the expression in the CE-package passed to treval is a define, then treval creates a new CE-package as follows. The expression is the body of the define. The environment is the environment of the old CE-package. The continuation is a new continuation with the following properties. It is tagged define. Its unevaluated data field is the variable of the define. It has no evaluated data. Its environment and continuation are the same as the ones in the old CE-package.

Next-cep

If next-cep finds a define on the continuation of the CE-package passed to it, it forms a CE-package as follows. First, a binding is created from the variable name in the unevaluated data field of the continuation in the old CE-package to the value in the expression field of the old CE-package in the environment in the continuation of the old CE-package. The expression part of the new CE-package is the unspecified return
value. The environment and continuation are the same as the ones in the continuation in the old CE-package.

2.2.13 Set!-statements

Treval

If the expression in the CE-package passed to treval is a set!, then treval creates a new CE-package as follows. The expression is the body of the set!. The environment is the environment of the old CE-package. The continuation is a new continuation with the following properties. It is tagged assign. Its unevaluated data field is the variable of the define. It has no evaluated data. Its environment and continuation are the same as the ones in the old CE-package.

Next-cep

If next-cep finds an assign on the continuation of the CE-package passed to it, it forms a CE-package as follows. First, an assignment is performed in the environment in the continuation of the old CE-package, binding the variable name in the unevaluated data field of the continuation in the old CE-package to the value in the expression field of the old CE-package. The expression part of the new CE-package is the unspecified return value. The environment and continuation are the same as the ones in the continuation in the old CE-package.

2.2.14 Sequences

Treval

If the expression in the CE-package passed to treval is a sequence, then treval creates a new CE-package as follows. The expression is the first expression in the list of sequence actions. The environment is the same environment as the one in the old CE-package. The continuation is a new continuation with the following properties. It is tagged seq. Its unevaluated data field is the list of remaining actions. It has no
evaluated data. Its environment and continuation are the same as the ones in the old CE-package.

**Next-cep**

If next-cep finds a seq on the continuation of the CE-package passed to it, it forms a CE-package as follows.

If there are no remaining actions in the unevaluated data field in the continuation, the expression is the value in the expression field of the old CE-package. The environment and continuation are the same as the ones in the continuation of the old CE-package.

If there are actions left to be evaluated, then the expression is the first of the remaining actions. The environment is the environment in the continuation of the old CE-package. The continuation is a new continuation with the following properties. It is tagged seq. Its unevaluated data field is the list of remaining actions. It has no evaluated data. Its environment and continuation are the same as the ones in the continuation in the old CE-package.

### 2.3 The State to Scheme Expression Mapping

The state of the evaluator at any given time is the current CE-package it is evaluating. The mapping reads through the CE-package and produces an expression of the form:

$$(\texttt{letrec } (\texttt{binding binding ...}) \ \texttt{expression})$$

The method for doing this follows.

1. Scan through all of the environment chains. Start with the environment chain in the current CE-package, and then chase up through the continuations and scan the environment chains in them for all of the variable bindings contained in them. Record every variable and its binding, but do not record the same variable twice. If two variables in different environments have the same name, one of them must be suffixed to distinguish the two.
2. For every binding that was recorded in step one, the expression that the variable is bound to must be unparsed from Scheme syntax back to S-expressions. It is important to use the suffixes that were created in step one. These bindings now form the bindings of the letrec expression.

3. Take the expression in the current CE-package and unparse it from syntax to an S-expression. Once again it is important to be consistent with variable renaming.

4. A continuation can be viewed as a Scheme expression with a section missing. The top level continuation can be written out with the unparsed expression from step three filling in the hole of that continuation. This process can be repeated until the continuation is emptied to form the expression of the letrec expression.

Unparse.scm is responsible for unparsing Scheme syntax to S-expressions. It does not build the tables for suffixing variables, but it consults them in order to correctly unparse syntax.

Bindings.scm is responsible for two tasks. Given an environment, it generates a list of bindings from variables to syntax, both making sure to not repeat variables and generating the suffixing tables. Also, given a list of variables bound to syntax, it uses unparse.scm to turn them into bindings from variables to S-expressions.

Print-cep.scm determines whether or not the current step should be printed. If it determines that the step should be printed, it calls on the file bindings.scm to generate a list of bindings to syntax. It then calls on bindings.scm to convert that list to bindings to S-expressions. Next, it calls on unparse.scm to make the inner expression. It then recursively travels down the continuation to build the entire expression. Finally, the bindings are put together with the expression to build the letrec expression.
2.3.1 The Unparser

The unparser is executed by calling the procedure print-expression. Print-expression takes a piece of Scheme syntax, an environment that the piece of syntax is being evaluated in, and a list of variables that are bound in this piece of syntax by lambdas or let expressions. Print-expression dispatches to the appropriate procedure to print out that expression and possibly make recursive calls to print-expression to print out the various sub-expressions of the syntax. These procedures have access to a procedure suffixer which consults a table to return the suffix of a variable in some environment.

The methods the unparser uses for printing different pieces of syntax follow.

Nil

If the syntax passed to the unparser is the null string, the unparser returns #f.

Symbols

If the syntax passed to the unparser in a symbol, the unparser returns a symbol generated by attaching a ' onto the front of the symbol.

Lists

If the syntax passed to the unparser is a list, the unparser generates a list to return. That list is formed by attaching the symbol list to the result of applying the unparser to all of the elements in the list passed to unparse.

Pairs

If the syntax passed to the unparser is a pair, the unparser generates a list to return. That list has three elements: the symbol cons, the result of applying the unparser to the first element of the pair, and the result of applying the unparser to the second element of the pair.
Primitive Procedures

If the syntax passed to the unparses is a primitive procedure, then the unparses returns the name of that primitive procedure.

Compiled Procedures

If the syntax passed to the unparses is a compiled procedure, then the unparses returns the name of that compiled procedure.

Combinations

If the syntax passed to the unparses is a combination, the unparses generates a list to return. The list is formed by attaching the result of applying the unparses to the operator of the combination to the list formed by applying the unparses to the list of the operands of the combination.

Variables

If the syntax passed to the unparses is a variable, the unparses checks to see if that variable is in the list of bound variables. If it is, it returns that variable, otherwise, it returns the symbol of the suffixed variable. It does this by searching the environment chain to find which environment, if any, the variable is bound in. It then uses the procedure suffix to consult the table to find what suffix that variable should have.

Conditionals

If the syntax passed to the unparses is a conditional, the unparses generates a list to return. If the if has no alternative, that list has three elements: the symbol if, the result of applying the unparses to the predicate of the conditional, and the result of applying the unparses to the consequence. If the if has an alternative, the list has four elements, the additional element being the result of applying the unparses to the alternative.
Cond-statements

If the syntax passed to the unparser is a cond, then the unparser generates a list by attaching the symbol cond to the list generated by unparsing the cond clauses. A cond clause is unparsed by simply unparsing the elements of the clause with a few exceptions. If the test of the clause is an else, that is unparsed to the symbol else. If the expression of the clause is a receiver, the symbol => is inserted between the unparsing of the test and the unparsing of the expression in the receiver.

Case-statements

If the syntax passed to the unparser is a case, then the unparser generates a list by attaching the symbol case and the result of applying the unparser to the key to the list generated by unparsing the case clauses. A case clause is unparsed by leaving the objects untouched and applying the unparser to the expression. If the objects field is an else, then the objects field is replaced with the symbol else.

And-statements

If the syntax passed to the unparser is an and, the unparser generates a list to return. That list is formed by attaching the symbol and to the result of applying the unparser to all of the subexpressions of the and statement.

Or-statements

If the syntax passed to the unparser is an or, the unparser generates a list to return. That list is formed by attaching the symbol or to the result of applying the unparser to all of the subexpressions of the or statement.

Lambdas

If the syntax passed to the unparser is a lambda, the unparser goes through the following process. It extracts the parameter fields from the lambda and assembles an argument list suitable for printing. It then applies the unparser to the body of the
lambda, but with the bound variables extended to also include the parameters of the lambda. It then generates a list of three elements: the symbol lambda, the arguments list, and the unparsed body.

**Let-statements**

If the syntax passed to the unparser is a let, the unpaser goes through the following steps. First it goes through the list of bindings applying the unpaser to the expression part of each binding. Then it applies the unpaser to the body of the let, but with the bound variables extended to also include the variables bound by the let. Then it generates a list of three or four elements: the symbol let, optionally the let’s name, the unparsed bindings, and the unparsed body.

**Let*-statements**

If the syntax passed to the unpaser is a let*, the unpaser goes through the following steps. First it goes through the list of bindings one at a time. It applies the unpaser to the expression part of each binding. It extends the bound variables to include the variable in the previous binding at each step. Then it applies the unpaser to the body of the let*, but with the bound variables extended to also include the variables bound by the let*. Then it generates a list of three elements: the symbol let*, the unparsed bindings, and the unparsed body.

**Letrec-statements**

If the syntax passed to the unpaser is a letrec, the unpaser goes through the following steps. First it goes through the list of bindings applying the unpaser to the expression part of each binding, but with the bound variables extended to also include those variables bound by the letrec. Then it applies the unpaser to the body of the letrec, but with the bound variables extended to also include the variables bound by the letrec. Then it generates a list of three elements: the symbol letrec, the unparsed bindings, and the unparsed body.
Definitions

If the syntax passed to the unparsen is a define, the unparsen generates a list to return. If the define is not binding a variable to a value, then the list has two elements: the symbol define and the result of applying the unparsen to the variable of the definition. If the define is binding a variable to a value, then the list has three expressions, the additional expression being the result of applying the unparsen to the expression of the definition.

Set!-statements

If the syntax passed to the unparsen is a set!, the unparsen generates a list to return. The list has three elements: the symbol set!, the result of applying the unparsen to the variable of the set!, and the result of applying the unparsen to the expression of the set!.

Sequences

If the syntax passed to the unparsen is a sequence, the unparsen generates a list to return. That list is formed by attaching the symbol begin to the result of applying the unparsen to all of the actions in the sequence.

Values

If the syntax passed to the unparsen does not match any of the above cases, then it is returned as is.

2.3.2 Environment to Bindings Converter

An environment chain is converted to a list of bindings of suffixed variables to syntax by the procedure environment->unprinted-letrec-bindings. This procedure scans an environment chain working from the initial-environment backwards up through the chain. At each environment in the chain, the procedure retrieves all of the bindings in that environment. Every time a binding is added to this list a hash table is updated to
reflect that that binding has been printed, and a suffix is generated for that variable and stored in the table. No bindings are inserted into the list twice. Also, no bindings are printed if they are bound to bad values.

To suffix the variables, there are two methods provided in the code. The default method uses as few numbers as possible, always assigning the next available number for a symbol when a new suffix is requested. The other available method assigns all variables in the same environment to the same suffix.

This list of syntaxed bindings is then converted to a list of bindings to Scheme expressions by applying print-expression to the expression fields of all of the bindings.

2.3.3 Whole Expression Generator

The whole expression generator takes a continuation and an inner unparsed expression. The whole expression generator dispatches on the tag of the continuation to determine how the inner expression should be inserted into the continuation. It then calls itself recursively with the newly formed inner expression and the continuation contained inside of that continuation until it reaches the empty continuation.

The following is the behavior of the whole expression generator, make-whole-exp, on the various continuation tags.

Comb

If the continuation passed to make-whole-exp is tagged comb, then make-whole-exp forms a new inner expression as follows. First, it applies the unparses to the reverse of all of the unevaluated subexpressions found in the unevaluated data field of the continuation. Then, it applies the unparses to all of the evaluated subexpressions found in the evaluated data field of the continuation. It then forms a list of the unparsed unevaluated expressions, the inner expression that was passed to it, and the unparse of the evaluated expressions. This new inner expression is then passed to make-whole-exp with the continuation contained inside the continuation originally passed to make-whole-exp.
If

If the continuation passed to make-whole-exp is tagged if, then make-whole-exp forms a new inner expression as follows. First, it applies the unparsers to the consequence and alternative found in the unevaluated data field of the continuation. It then forms a list of the symbol if, the inner expression passed to make-whole-expression, the consequence, and the alternative if there is one. This new inner expression is then passed to make-whole-exp with the continuation contained inside the continuation originally passed to make-whole-exp.

Cond

If the continuation passed to make-whole-exp is tagged cond, then make-whole-exp forms a new inner expression as follows. This new inner expression is the list starting with the symbol cond followed by the unparses of all of the clauses of the cond. The first clause is the list of the inner expression and the unparses of the first expression in the unevaluated data field of the continuation. The list of remaining clauses is the remainder of the list in the unevaluated data field of the continuation. While unparsing the clauses, make-whole-exp ensures that elses and receivers are unparsed correctly. This new inner expression is then passed to make-whole-exp with the continuation contained inside the continuation originally passed to make-whole-exp.

Case

If the continuation passed to make-whole-exp is tagged case, then make-whole-exp forms a new inner expression as follows. This new inner expression is the list starting with the symbol case followed by the inner expression, followed by the unparses of all of the clauses of the case. The clauses are contained in the unevaluated data field of the continuation. This new inner expression is then passed to make-whole-exp with the continuation contained inside the continuation originally passed to make-whole-exp.
And

If the continuation passed to make-whole-exp is tagged **and**, then make-whole-exp forms a new inner expression as follows. This new inner expression is the list starting with the symbol and, followed by the inner expression, followed by the unparse of all of the subexpressions contained in the unevaluated data field of the continuation. This new inner expression is then passed to make-whole-exp with the continuation contained inside the continuation originally passed to make-whole-exp.

Or

If the continuation passed to make-whole-exp is tagged **or**, then make-whole-exp forms a new inner expression as follows. This new inner expression is the list starting with the symbol or, followed by the inner expression, followed by the unparse of all of the subexpressions contained in the unevaluated data field of the continuation. This new inner expression is then passed to make-whole-exp with the continuation contained inside the continuation originally passed to make-whole-exp.

Let

If the continuation passed to make-whole-exp is tagged **let**, then make-whole-exp forms a new inner expression as follows. First, it assembles an unparsed list of bindings by combining the evaluated bindings in the evaluated data field in the continuation with the binding currently being evaluated in the inner expression and the unevaluated bindings in the unevaluated data field of the continuation. The body of the let is then unparsed with all of the variables contained in the bindings marked as bound by the let. These parts are then assembled into a list beginning with the symbol let and optionally the name of the let. This new inner expression is then passed to make-whole-exp with the continuation contained inside the continuation originally passed to make-whole-exp.
Let*

If the continuation passed to make-whole-exp is tagged let*, then make-whole-exp forms a new inner expression as follows. First, it assembles an unparsed list of bindings by combining the evaluated bindings in the evaluated data field in the continuation with the binding currently being evaluated in the inner expression and the unevaluated bindings in the unevaluated data field of the continuation. Each binding is unparsed with all of the previously bound variables marked as bound by the let*. The body of the let* is then unparsed with all of the variables contained in the bindings marked as bound by the let*. These parts are then assembled into a list beginning with the symbol let*. This new inner expression is then passed to make-whole-exp with the continuation contained inside the continuation originally passed to make-whole-exp.

Letrec

If the continuation passed to make-whole-exp is tagged letrec, then make-whole-exp forms a new inner expression as follows. First, it assembles a list of all the variables bound in the bindings of the letrec. Then, it creates an unparsed list of bindings by combining the evaluated bindings in the evaluated data field in the continuation with the binding currently being evaluated in the inner expression and the unevaluated bindings in the unevaluated data field of the continuation. Each binding is unparsed with all of the variables of the letrec marked as bound. The body of the letrec is then unparsed with all of the variables contained in the bindings marked as bound. These parts are then assembled into a list beginning with the symbol letrec. This new inner expression is then passed to make-whole-exp with the continuation contained inside the continuation originally passed to make-whole-exp.

Define

If the continuation passed to make-whole-exp is tagged define, then a list is made of the symbol define, the suffixed variable found in the unevaluated data field of the continuation, and the inner expression. This new inner expression is then passed to
make-whole-exp with the continuation contained inside the continuation originally passed to make-whole-exp.

**Assign**

If the continuation passed to make-whole-exp is tagged `assign`, then a list is made of the symbol set!, the suffixed variable found in the unevaluated data field of the continuation, and the inner expression. This new inner expression is then passed to make-whole-exp with the continuation contained inside the continuation originally passed to make-whole-exp.

**Sequence**

If the continuation passed to make-whole-exp is tagged `seq`, then make-whole-exp forms a new inner expression as follows. This new inner expression is the list starting with the symbol `seq`, followed by the inner expression, followed by the unparse of all of the actions contained in the unevaluated data field of the continuation. This new inner expression is then passed to make-whole-exp with the continuation contained inside the continuation originally passed to make-whole-exp.

### 2.3.4 Printing Controls

There are two methods for determining when to perform the state to S-expression mapping. By default the evaluator uses a simple counter set to print every step. This can be set to display the current state every fixed number of steps. Alternatively, every time `print-cep` is called, the call is tagged to indicate what kind of step was just performed. The code can be set to print out every time a certain rule is evaluated. The print out controls information is stored in the variables `print-every` (an integer indicating which steps to print if using the counter), `use-counter` (a boolean indicating whether to use the counter or to use the rule matcher), and `print-rules` (a list of tags that should be printed).
Chapter 3

User’s Guide for the Evaluator

This chapter of the thesis is provided as a guide to using the Scheme evaluator detailed in chapter 2. The first section of this chapter details exactly what aspects of Scheme have been implemented, and how they behave. The second section of this chapter explain how to use the controls available to control the evaluator.

3.1 Special Forms

The evaluator follows all of the basics of the 6.001 Scheme language. In terms of types, case-sensistivity, scoping, basic syntax, and various other aspects of this sort, this evaluator behaves identical to the 6.001 Scheme evaluator. Information on the 6.001 Scheme evaluator can be found in the Revised Report on the Algorithmis Language Scheme [2]. The special forms implemented in this evaluator do not behave exactly in the same manner as they are explained in a Scheme reference manual, and some special forms are not present at all. The following section details the behavior of special froms in this evaluator.

Every special form in this evaluator is detailed below. Each section begins with the syntax of that special form. Brackets indicate an optional subexpression and “...” indicates that there may be any number of that particular subexpression. That is followed by an explanation of how that special form is evaluated. Finally there are some examples of that expression being evaluated.
3.1.1 If

(if predicate consequence [alternative])

First the predicate of the if is evaluated in the current environment. If the predicate evaluates to a value other than false, the consequence is evaluated in the current environment, and that value is returned. If the predicate evaluates to the value false, then if there is an alternative, it is evaluated in the current environment and that value is returned. If there is no alternative, then the unspecified return value is returned.

doeval==> (if (+ 2 3) (* 5 6) (/ 1 0))

(letrec #f
  (if 5
      (* 5 6)
      (/ 1 0)))

(letrec #f
  (* 5 6))

(letrec #f
  30)
;Value: 30

doeval==> (if (= 2 3) (/ 1 0) (+ 5 7))

(letrec #f
  (if #f
      (/ 1 0)
      (+ 5 7)))

(letrec #f
  (+ 5 7))

(letrec #f
  12)
;Value: 12

doeval==> (if (= 2 3) 1)

(letrec #f
(if #f
  1))

(letrec #f
  #[:unspecified-return-value])
;Value: #:unspecified-return-value

### 3.1.2 Cond

(cond (predicate expression ...) ...)

The predicates of the clauses are evaluated in order from left to right in the current environment. If one of them evaluates to a value other than false, then the expressions of that clause are evaluated in the current environment, and the value of the last expression in that clause is the value returned by the cond. The predicate of the last clause may be an else. If all of the previous predicates evaluate to false, then the expressions of that else clause are evaluated as above.

If the clause whose expressions are to be evaluated has no expressions after the predicate, then the value of the predicate is returned. If all predicates evaluate to the value false and there is no else clause, then the unspecified return value is returned.

A clause may also have the form (predicate => recipient). The recipient must be a procedure of one argument. If the predicate of a clause of that form evaluates to a value other than false, than the value returned by the cond is the result of applying the recipient to the value of the predicate.

doeval==> (cond ((+ 3 3))
  ((* 5 7) => (lambda (x) (* x x)))
  (else (* 2 3) (+ 3 4)))

(letrec #f
  (cond (6)
    ((* 5 7) => (lambda (x) (* x x)))
    (else (begin (* 2 3) (+ 3 4)))))

(letrec #f
  6)
;Value: 6
(letrec #f
  (cond (#f)
    ((= 5 7) => (lambda (x) (* x x)))
    (else (* 2 3) (+ 3 4))))

doeval==> (cond ((= 2 3))
  ((= 5 7) => (lambda (x) (* x x)))
  (else (* 2 3) (+ 3 4)))
(else (begin (* 2 3) (+ 3 4)))

(letrec #f
  (cond (#f => (lambda (x) (* x x)))
       (else (begin (* 2 3) (+ 3 4))))

(letrec #f
  (begin (* 2 3) (+ 3 4)))

(letrec #f
  (begin 6 (+ 3 4)))

(letrec #f
  (begin (+ 3
          (letrec #f
                  (begin 7)))
      #[unspecified-return-value])
  ;Value: 7

  3.1.3 Case

(case key (objects expression ...) ...)

  First the key of the case is evaluated in the current environment. The key is then
compared using eqv? to all of the objects in the object lists. An object can only appear once in the object lists. If a object list is found that contains the key, then the expressions of that clause are evaluated in the current environment, with the value of the last expression becoming the value returned by the case. The objects list of the last clause of the case may be an else. If the key does not match any of the objects listed, then if there is an else, the expressions of the else clause are evaluated, otherwise, the unspecified return value is returned.

doeval==> (case (car '(a c))
   ((a b e) (+ 2 3))
   (else (* 2 3)))

(letrec #f
   (case 'a
      ((a b e) (+ 2 3))
      (else (* 2 3)))))

(letrec #f
   (+ 2 3))

(letrec #f
   5)
;Value: 5

doeval==> (case (cadr '(a c))
   ((a b e) (+ 2 3))
   (else (* 2 3)))

(letrec #f
   (case 'c
      ((a b e) (+ 2 3))
      (else (* 2 3)))))

(letrec #f
   (* 2 3))

(letrec #f
   6)
;Value: 6

doeval==> (case (cadr '(a c))
((a b e) (+ 2 3))

(letrec #f
  (case 'c
    ((a b e) (+ 2 3)))

(letrec #f
  ;[unspecified-return-value])
  ;Value: #[unspecified-return-value]

doeval==> (case (car '(a c))
  ((a b e) (+ 2 3))
  ((p a u l) (* 2 3)))
ERROR: repeated object: ...

3.1.4 And

(and expression ...)

If the and has no subexpressions, then it returns the value true. If it has subexpressions then they are evaluated in left to right order. If any of the subexpressions evaluates to the value false, then evaluation of the and stops, and the value false is returned. If none of the subexpressions evaluate to false, then the value of the last subexpression is returned.

doeval==> (and)
;Value: #t

doeval==> (and (+ 2 3) (= 1 2) (/ 1 0))

(letrec #f
  (and 5
    (= 1 2)
    (/ 1 0)))

(letrec #f
  (and (= 1 2)
    (/ 1 0)))

(letrec #f
  (and #f

47
(letrec #f
    #f)
;Value: #f

doeval==> (and 1 2 3)

(letrec #f
    (and 2
        3))

(letrec #f
    (and 3))

(letrec #f
    3)
;Value: 3

3.1.5 Or

(or expression ...)

If the or has no subexpressions, then it returns the value false. If it has subexpressions then they are evaluated in left to right order. If any of the subexpressions evaluates to a value other than false, then evaluation of the or stops, and the value of that subexpression is returned. If all of the subexpressions evaluate to false, then the value false is returned.

(doeval==> (or)
;Value: #f

doeval==> (or (= 2 3) (+ 2 3) (/ 1 0))

(letrec #f
    (or #f
        (+ 2 3)
        (/ 1 0)))

(letrec #f
    (or (+ 2 3)
(/ 1 0)))

(letrec #f
  (or 5
      (/ 1 0)))

(letrec #f
  5)
;Value: 5

doeval==> (or (= 2 3) (= 1 4))

(letrec #f
  (or #f
      (= 1 4)))

(letrec #f
  (or (= 1 4)))

(letrec #f
  (or #f))

(letrec #f
  #f)
;Value: #f

### 3.1.6 Lambda

(lambda arguments expression ...)

Arguments must be in one of three forms:

1. argument

2. (argument ...)

3. (argument ... argument . argument).

A lambda evaluates to a procedure. The procedure's environment is the current environment. If the arguments are of type 1, then there are no required parameters and the rest parameter is the one argument. If the arguments are of type 2, then there is no rest parameter and the required parameters are that arguments list. If
the arguments are of type 3, then the rest parameter is the final argument of the
arguments list and the required parameters are the list of all but the last argument.
The body of the lambda are the expressions. There must be at least one expression.

doeval==> ((lambda x (car x)) 1 2 3)

(letrec ((x (list 1 2 3)))
  (car x))

(letrec ((x (list 1 2 3)))
  (car (list 1 2 3)))

(letrec #f
  1)
;Value: 1

doeval==> ((lambda (x y z) x) 1 2 3)

(letrec ((x 1) (y 2) (z 3))
  x)

(letrec ((x 1) (y 2) (z 3))
  1)
;Value: 1

doeval==> ((lambda (x . y) x) 1 2 3)

(letrec ((x 1) (y (list 2 3)))
  x)

(letrec ((x 1) (y (list 2 3)))
  1)
;Value: 1

3.1.7 Let

(let [name] (binding ...) expression ...)

A binding is of the form:

(variable expression).

First, all of the expressions in the bindings are evaluated in the current environ-
ment. When they are all values, a new environment, whose parent environment is 
the current environment, is created where all of the variables listed in the bindings 
are bound to the appropriate values. If the let is named, then the name is bound 
to a procedure in that new environment. That procedure's environment is the newly 
created environment. It has no rest parameter. Its required parameters is the list of 
variables bound by the let. Its body is the list of expressions in the let.

Once this new environment is created, the expressions of the let are evaluated in 
this new environment. The value of the last expression is the value returned by the 
let.

doeval==> (define x 1)

(letrec ((x 1))
  ;[unspecified-return-value])
;Value: #[unspecified-return-value]
doeval==> (define y 2)

(letrec ((x 1) (y 2))
  ;[unspecified-return-value])
;Value: #[unspecified-return-value]
doeval==> (let ((x 3) (y x) (z y))
  (+ x y z))

(letrec ((x 1) (y 2))
  (let ((x 3) (y x) (z y))
    (+ x y z)))

(letrec ((x 1) (y 2))
  (let ((x 3) (y 1) (z y))
    (+ x y z)))

(letrec ((x 1) (y 2))
  (let ((x 3) (y 1) (z y))
    (+ x y z)))

(letrec ((x 1) (y 2))
  (let ((x 3) (y 1) (z 2))
    (+ x y z)))

(letrec ((x-1 3) (y-1 1) (z 2) (x 1) (y 2))
(+ x-1 y-1 z))

(letrec ((x-1 3) (y-1 1) (z 2) (x 1) (y 2))
  (+ x-1 y-1 2))

(letrec ((x-1 3) (y-1 1) (z 2) (x 1) (y 2))
  (+ x-1 1 2))

(letrec ((x-1 3) (y-1 1) (z 2) (x 1) (y 2))
  (+ 3 1 2))

(letrec ((x 1) (y 2))
  6)
; Value: 6

[The following is merely an excerpt from the evaluation]

doeval==> (let loop ((answer 1) (n 4))
  (if (= n 1)
    answer
    (loop (* answer n) (- n 1))))
...

(letrec ((loop
    (lambda (answer n)
      (if (= n 1)
        answer
        (loop (* answer n) (- n 1)))))
    (answer 1)
    (n 4))
  (loop 4 3))
...

(letrec ((answer-1 4)
  (n-1 3)
  (loop
    (lambda (answer n)
      (if (= n 1)
        answer
        (loop (* answer n) (- n 1)))))
    (answer 1)
    (n 4))
  (loop 12 2))
...  
(letrec ((answer#1 12)  
(n-1 2)  
(loop  
(lambda (answer n)  
(if (= n 1)  
answer  
(loop (* answer n) (- n 1)))))  
(answer 1)  
(n 4))  
(loop 24 1))
...
;Value: 24

3.1.8 Let*
(let* (binding ...) expression ...)

A binding is of the form:
(variable expression).

The first expression of the first binding of the let* is evaluated in the current environment. A new environment is then created whose parent environment is the current environment, where the variable of the first binding is bound to the value of the expression of the first binding. This process is then repeated with the next binding in this new environment, until a new environment has been added to the chain for every variable in the let*. If the let* has no bindings, then a new empty environment is created whose parent environment is the current environment.

Once this final new environment is created, the expressions of the let* are evaluated in this new environment. The value of the last expression is the value returned by the let*.

doeval==> (define x 1)
(letrec ((x 1))
doeval==>

(let* ((x y z))
  (+ x y z))

(letrec ((x-1 3)
          (let* ((y x-1)
                 (z y z)))
          (x-1 3) (x 1) (y 2))

(letrec ((z 3) (y-1 (z y))
          (+ x-1 y z)))

(letrec ((y-1 3) (x-1 (z y))
          (+ x-1 y z)))

*doeval* => (define y 2)

(letrec ((x 1) (y 2))
  #[unspecified-return-value])

;Value: #[unspecified-return-value]

(let* ((x 3) (y x) (z y))
  (+ x y z))

(letrec ((x-1 3) (x 1) (y 2))
  (let* ((y x-1) (z y))
    (+ x-1 y z)))

(letrec ((y-1 (x 1) (y 2))
          (+ x-1 y z)))

(letrec ((y-1 (x 1) (y 2))
          (let* ((z y-1))
            (+ x-1 y z)))

(letrec ((y-1 (x 1) (y 2))
          (let* ((z y-1))
            (+ x-1 y-1 z)))

(letrec ((y-1 (x 1) (y 2))
          (let* ((z 3))
            (+ x-1 y-1 z)))

(letrec ((y-1 (x 1) (y 2))
          (let* ((z y-1) (x-1 3))
            (+ x-1 y-1 z)))

(letrec ((y-1 (x 1) (y 2))
          (+ x-1 y-1 3)))

(letrec ((y-1 (x 1) (y 2))
          (+ x-1 3 3))

(letrec ((y-1 (x 1) (y 2))
          (+ 3 3 3))

(letrec ((x 1) (y 2))
  9)

;Value: 9
3.1.9 Letrec

(letrec (binding ...) expression ...)

A binding is of the form:

(variable expression).

A new environment is created whose parent environment is the current environment where all the variables of the bindings are bound to nothing. All of the expressions of the bindings are now evaluated in this new environment. When an expression is evaluated, the appropriate variable in the new environment is bound to the value of the expression. When all of the expressions have been evaluated, the expressions of the letrec are evaluated in this new environment. It is possible to define a recursive function in the bindings of a letrec. This is not possible in the bindings of a let or let*.

doeval==> (define x 1)

(letrec ((x 1))
  ;[unspecified-return-value])
;Value: #[unspecified-return-value]

doeval==> (define y 2)

(letrec ((x 1) (y 2))
  ;[unspecified-return-value])
;Value: #[unspecified-return-value]

doeval==> (letrec ((x 3) (y x) (z y))
  (+ x y z))

(letrec ((x-1 3) (x 1) (y 2))
  (letrec ((x 3) (y x-1) (z y))
    (+ x y z)))

(letrec ((x-1 3) (x 1) (y 2))
  (letrec ((x 3) (y 3) (z y))
    (+ x y z)))

(letrec ((x-1 3) (y-1 3) (x 1) (y 2))
  (letrec ((x 3) (y 3) (z y-1)))
(letrec ((x-1 3) (y-1 3) (x 1) (y 2))
  (+ x-1 y-1 z))

(letrec ((x-1 3) (y-1 3) (x 1) (y 2))
  (+ x-1 y-1))

(letrec ((x-1 3) (y-1 3) (x 1) (y 2))
  (+ x-1 3 3))

(letrec ((x-1 3) (y-1 3) (x 1) (y 2))
  (+ 3 3 3))

(letrec ((x 1) (y 2))
  9)
;Value: 9

[The following is merely an excerpt from the evaluation]

doeval==> (letrec ((fact
  (lambda (x)
    (if (zero? x) 1 (* x (fact (- x 1))))))
  (z 2))
  (fact z))
...

(letrec ((x 2) (fact
  (lambda (x) (if (zero? x) 1 (* x (fact (- x 1))))))
  (z 2))
  (if (zero? x)
    1
    (* x (fact (- x 1)))))
...

(letrec ((x 2) (fact
  (lambda (x) (if (zero? x) 1 (* x (fact (- x 1))))))
  (z 2))
  (* x (fact 1))
(letrec ((x-1 1) (x 2))
  (fact
   (lambda (x) (if (zero? x) 1 (* x (fact (- x 1))))))
 (z 2))
(* x (* x-1 (fact (- x-1 1))))

(letrec ((x-2 0) (x-1 1) (x 2) (fact
   (lambda (x) (if (zero? x) 1 (* x (fact (- x 1))))))
 (z 2))
(* x (* x-1 (if (zero? 0) 1 (* x-2 (fact (- x-2 1)))))))

(letrec ((x 2) (fact
   (lambda (x) (if (zero? x) 1 (* x (fact (- x 1))))))
 (z 2))
(* 2 1))

;Value: 2

3.1.10 Define

(define variable [expression])

or

(define procedure-declaration expression ...)

where procedure-declaration is of the following form:

(proc-name argument ... [ . argument]).

If the define is of the first form, then the expression of the define is evaluated in
the current environment. If the variable is bound in the current environment, then
the binding is changed to now point to the new value. If there was no binding, a new
binding is created from the variable to the value.

57
If the define is of the second form, then a binding is created from the proc-name to a procedure in the current environment. If proc-name was already bound in the current environment, then the binding is changed to now point at the new procedure. The new procedure’s environment is the current environment. Its required parameters are the list of arguments excluding the dotted argument. Its rest parameter is the dotted argument, if there is one. Its body is the list of expressions of the define.

A define returns the unspecified return value.

doeval==> (define x 2)

(letrec ((x 2))
  #[unspecified-return-value])
;Value: #[unspecified-return-value]

(doeval==> (define (square x) (* x x))

(letrec ((x 2) (square (lambda (x) (* x x))))
  #[unspecified-return-value])
;Value: #[unspecified-return-value]

(doeval==> (define x 3)

(letrec ((x 3) (square (lambda (x) (* x x))))
  #[unspecified-return-value])
;Value: #[unspecified-return-value]

3.1.11 Set!

(set! variable [expression])

The expression of the set! is evaluated in the current environment. The environment chain is then searched up for the first occurrence of the variable. If one is found, then that binding is changed to point to the value of the expression, and set! returns the unspecified return value. If one is not found, this results in an error.

doeval==> (define x 3)
(letrec ((x 3))
  #[unspecified-return-value])
;Value: #[unspecified-return-value]

doeval==> (let ()
  (set! x 4))

(letrec ((x 4))
  #[unspecified-return-value])
;Value: #[unspecified-return-value]

doeval==> x

(letrec ((x 4))
  4)
;Value: 4

doeval==> (let ()
  (set! xx 5))
ERROR: (Do-assign!, variable not bound: xx)

3.1.12 Begin

(begin action ...)

The actions of the begin are evaluated in order from left to right. The value of
the last expression is returned.

doeval==> (define x 1)

(letrec ((x 1))
  #[unspecified-return-value])
;Value: #[unspecified-return-value]

doeval==> (begin
  (set! x (+ x 1))
  x)

(letrec ((x 1))
  (begin (set! x (+ 1 1)) x))

(letrec ((x 1))
  (begin (set! x 2) x))
3.1.13 Quote

(quote object)

This evaluates to object. There are two ways to quote an object:

- (quote object)

- 'object.

3.2 Evaluator Controls

This section details the various controls and settings of the evaluator.
3.2.1 Starting and Stopping the Evaluator

The read-eval-print loop is started by loading the load file, load.com, and then executing the command (doeval). The loop is exited by executing the command (exit-doeval) from within the doeval read-eval-print loop. Exiting and restarting the loop causes the global environment of the loop to be reset to the empty environment.

An example of these commands follow.

(doeval)
doeval==> (define x 2)

(letrec ((x 2))
  #[unspecified-return-value])
;Value: #[unspecified-return-value]
doeval==> (exit-doeval)
;Value: "Happy Happy Joy Joy"

(doeval)
doeval==> x
ERROR: #[condition 1 unbound-variable]

3.2.2 Choosing Which Lines to Print

There are two mechanisms for deciding at which steps evaluation should be interrupted to display the current state of the machine. The default method is to use a counter to print out every fixed number of steps. A step is defined as a point where the expression that would be printed would change. The default setting is to print every line. By resetting the variable print-every, the user can control the number of steps between print outs. The other mechanism to control print out is to print out every time a certain kind of step or kinds of steps are performed. There is a list print-rules that contains all of the rules that are currently being printed. The list of the rules is:

- comb: a combination is applied
- var: a variable is looked up
• if: the consequent or alternative is chosen
• cond: a clause is chosen or removed
• case: a clause is chosen
• and: a subexpression is thrown out or the and returns a value
• or: a subexpression is thrown out or the or returns a value
• let: a new environment is created
• let*: a new environment is created
• letrec: a binding is added to the environment
• define: a define is evaluated
• assign: a set! is evaluated
• seq: an action is removed from the sequence.

The boolean use-counter determines whether the evaluator should use the counter
or the rule matcher. An example of its usage follows.

doeval==> (define (fact x) (if (zero? x) 1 (* x (fact (- x 1)))))

(letrec (((fact (lambda (x) (if (zero? x) 1 (* x (fact (- x 1)))))
    #unspecified-return-value))
  ;Value: #unspecified-return-value
  doeval==> (set! use-counter #t)

(letrec (((fact (lambda (x) (if (zero? x) 1 (* x (fact (- x 1)))))
    #unspecified-return-value))
  ;Value: #unspecified-return-value
  doeval==> (set! print-every 10)

(letrec (((fact (lambda (x) (if (zero? x) 1 (* x (fact (- x 1)))))
    #unspecified-return-value))
  ;Value: #unspecified-return-value
  doeval==> (fact 4)
(letrec ((fact (lambda (x) (if (zero? x) 1 (* x (fact (- x 1)))))
  ((lambda (x) (if (zero? x) 1 (* x (fact (- x 1)))))) 4))

(letrec ((x-1 3) (x 4)
  (fact
    (lambda (x)
      (if (zero? x) 1 (* x (fact (- x 1)))))))
  (* x (if #f 1 (* x-1 (fact (- x-1 1))))))

(letrec ((x-2 2) (x-1 3) (x 4)
  (fact
    (lambda (x)
      (if (zero? x) 1 (* x (fact (- x 1))))))
  (* x (* x-1 (* x-2 (fact 1))))))

(letrec ((x-3 1) (x-2 2) (x-1 3) (x 4)
  (fact
    (lambda (x)
      (if (zero? x) 1 (* x (fact (- x 1))))))
  (* x
    (* x-1
      (* x-2
        (* x-3 (if (zero? 0) 1 (* x-4 (fact (- x-4 1)))))))))))

(letrec ((fact (lambda (x) (if (zero? x) 1 (* x (fact (- x 1)))))))
  24)
;Value: 24
doeval==> (set! use-counter #f)
;Value: #[unspecified-return-value]
doeval==> (set! print-rules '(if))
;Value: #[unspecified-return-value]
doeval==> (fact 4)

(letrec ((x 4) (fact (lambda (x)
    (if (zero? x) 1 (* x (fact (- x 1)))))))
  (* x (fact (- x 1))))

(letrec ((x-1 3) (x 4)
  (fact
(lambda (x)
  (if (zero? x) 1 (* x (fact (- x 1)))))))
(* x (* x-1 (fact (- x-1 1)))))))

(letrec ((x-2 2) (x-1 3) (x 4)
  (fact
    (lambda (x)
      (if (zero? x) 1 (* x (fact (- x 1)))))))
(* x (* x-1 (* x-2 (fact (- x-2 1)))))))

(letrec ((x-3 1) (x-2 2) (x-1 3) (x 4)
  (fact
    (lambda (x)
      (if (zero? x) 1 (* x (fact (- x 1)))))))
(* x (* x-1 (* x-2 (* x-3 (fact (- x-3 1)))))))

(letrec ((x-4 0) (x-3 1) (x-2 2) (x-1 3) (x 4)
  (fact
    (lambda (x)
      (if (zero? x) 1 (* x (fact (- x 1)))))))
(* x (* x-1 (* x-2 (* x-3 1)))))
;Value: 24

3.2.3 Variable Renaming

There are two options available for variable renaming. Variables can be suffixed in a manner which uses the fewest possible suffixes, or in a manner which distinguishes which variables are in the same frame. If the boolean minimal-numbering is set to true, then the first Scheme is used. If it is set to false, then the second Scheme is used. An example of this follows.

(doeval===> (set! minimal-numbering #t)

(letrec #f
    #*[unspecified-return-value]*)

64
(let ((x-1 (+ x-1 y)))
  (letrec ((x-1 (+ x-1 3)))
    (letrec ((x-1 (+ 2 3)))
      (x 2) (y 3))
    (y 3) (x 1))
  2) (y 3) (x 1))
(letrec ((x 1))
  5)
(letrec ((x 1))
  5)
(letrec ((x-1 1))
  #[unspecified-return-value])
(letrec ((x 1))
  #[unspecified-return-value])
(letrec ((x 1))
  (define x 1))
(letrec ((x 1))
  #[unspecified-return-value])
(letrec ((x 1))
  (set! minimal-numbering #f)
  5)
Chapter 4

Efficiency of the Evaluator

The substitution model evaluator designed in this thesis is designed under the prin-
ciple that the state to S-expression mapping would not be performed every step.
This chapter outlines how efficient the substitution model evaluator designed in this
thesis compares to both the standard Scheme evaluator and the substitution model
evaluator designed by Professor Albert Meyer [7].

The tests were run on the program fibonacci:

```
(define (fib x)
  (if (< x 2)
    x
    (+ (fib (- x 1))
        (fib (- x 2)))))
```

Fibonacci was chosen due to its tree recursion. It allowed tests to be run that
took a very small number of steps and a very large number of steps.

The numbers listed as the efficiency are a measure of the running time of the
machine. Obviously, a lower runtime is better. The runtime is not to be considered
an absolute measure of any statistic, but instead a general guideline to compare
the methods. If a runtime is listed at $\infty$, that means that the procedure takes an
impractical amount of time to perform.

In this chart, the phrase "thesis - n" means the evaluator of this thesis with print-
every set to the number n. The phrase "Scheme" refers to the built in 6.001 Scheme
evaluator [2]. The phrase “smeval” refers to the substitution model Scheme evaluator written by Professor Meyer.

<table>
<thead>
<tr>
<th></th>
<th>Scheme</th>
<th>thesis - ∞</th>
<th>thesis - 10</th>
<th>thesis - 1</th>
<th>smeval</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fib 1)</td>
<td>0</td>
<td>.030</td>
<td>.052</td>
<td>.75</td>
<td>.29</td>
</tr>
<tr>
<td>(fib 5)</td>
<td>.010</td>
<td>.16</td>
<td>1.6</td>
<td>13</td>
<td>7.5</td>
</tr>
<tr>
<td>(fib 10)</td>
<td>.020</td>
<td>.64</td>
<td>26</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>(fib 15)</td>
<td>.22</td>
<td>6.2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

These figures demonstrate that the evaluator presented in this thesis runs at speeds within an order of magnitude of the built in Scheme evaluator when the print-out is turned off. The more important fact is that at very large values, it can return values much more quickly than the substitution model evaluator written by Prof. Meyer that this thesis was designed to improve on, as long as the print out is set to return values every ten steps.
Chapter 5

Extending this Thesis

The evaluator presented in this thesis is not a finished project. The goal of this thesis was to lay out a working framework of a substitution model Scheme evaluator. This section details what work remains to be done so that other students may finish this work in the future.

The aspects of the thesis that are not yet completed are detailed in the following three sections. The first section deals with efficiency issues. The second section discusses additions to the syntax of the evaluator. The final section discusses additions that could be made to the print out controls of the evaluator.

5.1 Efficiency

The evaluator presented in this thesis was designed to be efficient in the manner by which it was implemented. The idea was that a tail recursive evaluator which only displayed its state every fixed number of steps would be more efficient than a substitution model evaluator. The code of this thesis itself has not been optimized to the greatest possible efficiency.

A project that could be undertaken is the task of optimizing the code of this thesis to get the greatest possible efficiency from the evaluator. For example, a deliberate choice made in designing this evaluator was to desugar very little. By not desugaring, the evaluator simulated substitution model evaluation much better than if
it had desugared the lines before evaluating. The actual Scheme interpreter desugars almost everything, including lets, let*s, letrecs, internal defines, ands, conds, cases, and many more special forms. The evaluator took an efficiency hit by not doing this. A possible efficiency improvement would be to allow a setting that set the amount of desugaring the evaluator does.

Also, the various dispatch loops could be optimized so that the most often used types of expressions could be moved up higher in the dispatch loops so that they are dispatched on before the others. This would require some code analysis to determine which special forms are used the most often.

There are many opportunities for efficiency improvements in the code of this thesis, and any improvements in those aspects of the thesis would increase the value of this project.

5.2 Syntax Additions

The evaluator presented in this thesis does not implement the entire Scheme language. There are many special forms present in Scheme that have not been implemented. The most noticeably absent special form is quasiquote. The framework for quasiquote has been started in the code, but the task of getting quasiquote to work on all inputs was not completed.

The special forms access and the-environment are present in the evaluator, but they are not documented as they do not have much meaning in a substitution model evaluator.

Also begun is the special form call-with-current-continuation. This is implemented as a procedure in Scheme, but in the evaluator presented here, it was necessary to implement it as a special form. This code works, but it breaks the requirement of this thesis that every step that is printed out is a valid S-expression. As a result, it is left out of the writeup of this thesis until its implementation is complete.

There are many other special forms present in Scheme that have not been added to this evaluator. Also, some of the special forms, most noticeably lambda, are in-
complete. Lambdas do not allow optional arguments or declarations in this evaluator.

There are many other syntax projects that could be undertaken. For example, this evaluator requires additional work to handle the print out of vectors.

The syntax available in this thesis is adequate to cover the material presented in the class Structures and Interpretations of Computer Programs (with the possible exception of quasiquote), which was the main goal of the syntax of this thesis. It can however be extended to be more complete.

5.3 Print Out Controls

There are many possible additions to the print out controls of the evaluator. The options available are fairly basic. Several ideas are presented below, but there are many possible ideas that could be implemented.

There could be other options of when to print out an expression. One example would be to perform the state to S-expression mapping if the S-expression matched some provided pattern. Other simpler options would be to specify exactly which lines should be printed.

There could also be more control over which procedures are applied by extending the environment and evaluating the body, and which procedures are applied by the underlying Scheme evaluator. The current method for deciding is to use the underlying Scheme evaluator when the procedure is not a compound procedure. This could be changed to specify exactly which compound procedures should be applied by the underlying Scheme evaluator and which should be stepped one step at a time by the substitution model evaluator.
Appendix A

The Scheme Code

A.1 The Read-Eval-Print Loop

A.1.1 doeval.scm

;;; The no-value object used repeatedly throughout the code

(define-structure bad-value)

(define bad-value (make-bad-value))

(define no-value
  (if #f "IThinkThisIsACrazyHackAndShouldBeEradicated")

(define n-bad-values
  (lambda (n)
    (if (= 0 n)
      '(0
        (cons (make-bad-value) (n-bad-values (- n 1))))))

;;; The read eval print loop:

(define doeval
  (lambda ()
    (call-with-current-continuation
      (lambda (p)
        (fluid-let ((exit-doeval (lambda ()
          (p "Happy Happy Joy Joy")))
         (initial-environment (*make-environment
          (procedure-environment doeval)
          (vector 0))))
        (do ()
          (#f #f)
(set! counter (make-counter 0))
(newline)
(display "doeval=> ")
(let ((exp (parse-object (current-input-port)
(current-parser-table))))
(call-with-current-continuation
(lambda (p)
(fluid-let ((error
(lambda args
(define display-in-seq
(lambda (1st)
(cond ((null? lst))
(else (display (car lst))
(display-in-seq (cdr lst)))))))
(newline)
(display "ERROR: ")
(display-in-seq args)
(p . )))
(let ((answer (treval (parse exp)
initial-environment)))
(newline)
(display ";Value: ")
(display answer)))))

;; New error handlers

;; These "placeholders" are used for fluid-lets and set!s in doeval

(define counter)

(define exit-loop)

(define exit-doeval)

(define initial-environment)
A.2 The Parser

A.2.1 parse.scm

;;; The parser

(define parse
  (lambda (exp)
    (cond ((pair? exp)
      (cond ((exp/conditional? exp) (parse/handle-conditional exp))
        ((exp/cond? exp) (parse/handle-cond exp))
        ((exp/case? exp) (parse/handle-case exp))
        ((exp/and? exp) (parse/handle-and exp))
        ((exp/or? exp) (parse/handle-or exp))
        ((exp/lambda? exp) (parse/handle-lambda exp))
        ((exp/let? exp) (parse/handle-let exp))
        ((exp/let*? exp) (parse/handle-let* exp))
        ((exp/letrec? exp) (parse/handle-letrec exp))
        ((exp/define? exp) (parse/handle-define exp))
        ((exp/set!? exp) (parse/handle-set! exp))
        ((exp/sequence? exp) (parse/handle-sequence exp))
        ((exp/quote? exp) (parse/handle-quote exp))
        ((exp/quasiquote? exp) (parse/handle-quasiquote exp))
        ((exp/the-env? exp) (parse/handle-the-env exp))
        ((exp/access? exp) (parse/handle-access exp))
        ((exp/call-cc? exp) (parse/handle-call-cc exp))
        (else (parse/handle-combination exp))))
    (else exp))))

;;; These are used by the parser to aid in the parsing of
;;; lambda, let, and cond

(define parse/handle-conditional
  (lambda (exp)
    (let ((num-of-subexp (length exp)))
      (if (or (= num-of-subexp 3) (= num-of-subexp 4))
        (apply make-conditional (map parse (cdr exp)))
        (error "If takes between 2 and 3 arguments" exp))))

(define parse/handle-cond
  (lambda (exp)
    (define parse-cond-exp
      (lambda (list-of-exp)
        (if (null? list-of-exp)
'()
(let ((cur-clause (car list-of-exp)))
  (if (and (list? cur-clause) (> (length cur-clause) 0))
    (cons (if (eq? (car cur-clause) 'else)
        (if (null? (cdr list-of-exp))
          (list (make-else))
          (cond ((< (length cur-clause) 2)
            (error "Malformed else"
              cur-clause))
              ((= (length cur-clause) 2)
                (parse (cadr cur-clause)))
              (else
                (parse (cons 'begin
                  (cadr cur-clause)))))
          (error "Else must be in last clause"
            exp))
            (list
              (parse (caar list-of-exp))
              (cond ((= (length cur-clause) 1)
                bad-value)
                ((= (length cur-clause) 2)
                  (if (eq? (cadr cur-clause) '=>)
                    (error "Malformed => " cur-clause)
                    (parse (cadr cur-clause))))
                ((and (= (length cur-clause) 3)
                  (eq? (cadr cur-clause) '=>))
                  (make-receiver (parse (caddr cur-clause)))
                  (else
                    (parse (cons 'begin (cdr cur-clause))))))))
        (parse-cond-exp (cdr list-of-exp))
        (error "Malformed clause: " cur-clause)))
      (make-cond-statement (parse-cond-exp (cdr exp)))))))

(define parse/handle-case
  (lambda (exp)
    (define parse-case-exp
      (lambda (list-of-exp)
        (if (null? list-of-exp)
            '()
            (let ((cur-clause (car list-of-exp)))
              (cons (list (if (eq? (car cur-clause) 'else)
                    (if (null? (cdr list-of-exp))
                    (make-else)
                    (error "else must be in last clause: "
                    exp))
                    (parse-case-exp (cadr list-of-exp))))
      (error "Malformed => " cur-clause))
    (else
      (parse-case-exp (cons 'begin (cdr cur-clause))))))))

(define parse-cond-exp
  (lambda (list-of-exp)
    (if (null? list-of-exp)
        '()
        (let ((cur-clause (car list-of-exp)))
          (cons (list (if (eq? (car cur-clause) 'else)
                (if (null? (cdr list-of-exp))
                (make-else)
                (error "else must be in last clause: "
                exp))
                (parse-cond-exp (cadr list-of-exp))))
            (error "Malformed => " cur-clause)))
      (make-cond-statement (parse-cond-exp (cdr exp))))))

74
(car cur-clause))
(if (> (length cur-clause) 2)
   (parse (cons 'begin (cdr cur-clause)))
   (parse (cadr cur-clause)))
   (parse-case-exp (cdr list-of-exp))))))
(define no-copies?
   (lambda (list-of-exp)
   (check-for-copies
   (apply append
   (map (lambda (clause)
       (if (and (list? clause)
       (> (length clause) 1)
       (or (eq? (car clause) 'else)
       (list? (car clause))))
       (if (eq? (car clause) 'else)
       '()
       (car clause))
       (error "malformed clause: " clause)))
       list-of-exp))))
   (define check-for-copies
   (lambda (lst)
   (if (null? lst)
   #t
   (if (memv (car lst) (cdr lst))
   #f
   (check-for-copies (cdr lst)))))))
(if (< (length exp) 2)
(error "malformed case: " exp)
(if (no-copies? (cddr exp))
   (make-case-statement (parse (cadr exp))
   (parse-case-exp (cddr exp)))
   (error "repeated object: " exp))))

(define parse/handle-and
   (lambda (exp)
   (make-and-statement (map parse (cdr exp))))))

(define parse/handle-or
   (lambda (exp)
   (make-or-statement (map parse (cdr exp))))))

(define parse/handle-lambda
   (lambda (exp)
   (if (< (length exp) 3)
   (error "Not enough subforms in lambda: " exp)
(let ((args (cadr exp))
    (body (if (= 3 (length exp))
        (caddr exp)
        (cons 'begin (cddr exp))))
    (make-lambda lambda-tag:unnamed
        (required-args args)
        #f
        (rest-args args)
        #f
        #f
        (parse body))))

(define required-args
    (lambda (args)
        (if (pair? args)
            (cons (car args) (required-args (cdr args)))
            #f)))

(define rest-args
    (lambda (args)
        (if (pair? args)
            (rest-args (cdr args))
            args)))

(define parse/handle-let
    (lambda (exp)
        (apply make-let-statement (parse-let-exp exp))))

(define parse/handle-let*
    (lambda (exp)
        (apply make-let*-statement (parse-let*-exp exp))))

(define parse/handle-letrec
    (lambda (exp)
        (apply make-letrec-statement (parse-letrec-exp exp))))

(define parse-let-exp
    (lambda (exp)
        (if (< (length exp) 3)
            (error "not enough subforms in let: " exp))
            (let ((bindings (if (symbol? (cadr exp))
            (caddr exp)
            (cadr exp)))
            (proc-name (if (symbol? (cadr exp))
            (cadr exp)
#f))
  (let ((body (if proc-name
    (cdddr exp)
    (cddr exp))))
  (if (not (legal-bindings? bindings))
    (error "malformed bindings in let: " exp))
  (list proc-name
    (map (lambda (binding)
      (list (car binding)
        (parse (cadr binding))))
      bindings)
    (parse
      (if (> (length body) 1)
        (cons 'begin body)
        (car body)))))))

(define parse-let*-exp
  (lambda (exp)
    (cond ((< (length exp) 3)
      (error "not enough subforms in let: " exp))
    ((not (legal-bindings? (cadr exp)))
      (error "malformed bindings in let: " exp))
    (else
      (list
        (map (lambda (binding)
          (list
            (car binding)
            (parse (cadr binding))))
          (cadr exp))
        (parse (if (= 3 (length exp))
          (caddr exp)
          (cons 'begin
            (cddr exp))))))))))

(define parse-letrec-exp parse-let*-exp)

(define legal-bindings?
  (lambda (bindings)
    (define helper
      (lambda (bnd-list)
        (if (null? bnd-list)
          #t
          (and (list? (car bnd-list))
            (= 2 (length (car bnd-list)))
            (helper (cdr bnd-list)))))))
(define parse/handle-define
  (lambda (exp)
    (if (< (length exp) 2)
        (error "malformed define: " exp)
      (if (list? (cadr exp))
          (if (and (not (null? (cadr exp)))
                   (not (null? (cddr exp))))
              (make-definition (caadr exp) (parse
                              (cons
                                'lambda
                                (cons (cdadr exp)
                                      (cddr exp))))))
          (error "malformed define: " exp))
      (cond ((= (length exp) 2)
              (make-definition (cadadr exp) bad-value))
            (> (length exp) 3)
            (error "malformed define: " exp))
            (else
             (make-definition (cadadr exp) (parse (caddr exp))))))

(define parse/handle-set!
  (lambda (exp)
    (if (or (< (length exp) 2)
            (> (length exp) 3))
        (error "malformed set!: " exp)
      (let ((name (cadr exp))
             (expr (if (null? (cddr exp))
                      bad-value
                      (parse (caddr exp))))
            (let ((name (parse name)))
              (if (list? name)
                  (if (exp/access? name)
                      (make-set!-statement name expr)
                      (error "Incorrect second subform in set!: " expr))
                  (error "Incorrect second subform in set!: " expr)))))

(define parse/handle-sequence
  (lambda (exp)
    (if (> (length exp) 1)
        (make-sequence (map parse (cadr exp)))
        (error "malformed sequence: " exp)))))
(define parse/handle-quote
  (lambda (exp)
    (if (= (length exp) 2)
        (cadr exp)
        (error "malformed quote: " exp))))

(define parse/handle-quasiquote
  (lambda (exp)
    (if (= (length exp) 2)
        (make-quasi (parse-quasi 1 (cadr exp)))
        (error "malformed quasiquote: " exp))))

(define parse/quasi
  (lambda (level exp)
    (if (list? exp)
        (cond ((null? exp))
            ((eq? (car exp) 'quasiquote)
                (if (= 2 (length exp))
                    (list 'quasiquote (parse-quasi (+ level 1) (cadr exp)))
                    (map (lambda (subexp) (parse-quasi level subexp)) exp)))
            ((eq? (car exp) 'unquote)
                (if (= 2 (length exp))
                    (if (= level 1)
                        (make-unquoted (parse (cadr exp)))
                        (list 'unquote (parse-quasi (- level 1) (cadr exp)))
                        (map (lambda (subexp) (parse-quasi level subexp)) exp)))
                    (map (lambda (subexp) (parse-quasi level subexp)) exp)))
            ((eq? (car exp) 'unquote-splicing)
                (if (= 2 (length exp))
                    (if (= level 1)
                        (make-splice (parse (cadr exp)))
                        (parse-quasi (- level 1) (cadr exp)))
                        (map (lambda (subexp) (parse-quasi level subexp)) exp)))
                    (else (map (lambda (subexp) (parse-quasi level subexp)) exp)))
        exp)))))

(define parse/handle-the-env
  (lambda (exp)
    (if (> (length exp) 1)
        (error "malformed the-environment: " exp)
        (make-the-environment)))

(define parse/handle-access
  (lambda (exp)
    (if (and (= 3 (length exp))))
(symbol? (cadr exp))
(make-access (parse (caddr exp)) (cadr exp))
(error "malformed access: " exp))

(define parse/handle-call-cc
  (lambda (exp)
    (if (= 2 (length exp))
      (make-call-cc (parse (cadr exp)))
      (error "malformed call-with-current-continuation: " exp))))

(define parse/handle-combination
  (lambda (exp)
    (make-combination (parse (car exp)) (map parse (cdr exp)))))

;; These are the procedures parse uses to determine what kind of
;; expression an expression is.

(define exp/variable?
  (lambda (exp)
    (symbol? exp)))

(define exp/value?
  (lambda (exp)
    (not (pair? exp))))

(define exp/conditional?
  (lambda (exp)
    (eq? (car exp) 'if)))

(define exp/sequence?
  (lambda (exp)
    (eq? (car exp) 'begin)))

(define exp/lambda?
  (lambda (exp)
    (eq? (car exp) 'lambda)))

(define exp/define?
  (lambda (exp)
    (eq? (car exp) 'define)))

(define exp/set!?
  (lambda (exp)
    (eq? (car exp) 'set!)))
(define exp/cond?
  (lambda (exp)
    (eq? (car exp) 'cond)))

(define exp/case?
  (lambda (exp)
    (eq? (car exp) 'case)))

(define exp/and?
  (lambda (exp)
    (eq? (car exp) 'and)))

(define exp/or?
  (lambda (exp)
    (eq? (car exp) 'or)))

(define exp/let?
  (lambda (exp)
    (eq? (car exp) 'let)))

(define exp/let*?
  (lambda (exp)
    (eq? (car exp) 'let*)))

(define exp/letrec?
  (lambda (exp)
    (eq? (car exp) 'letrec)))

(define exp/quote?
  (lambda (exp)
    (eq? (car exp) 'quote)))

(define exp/quasiquote?
  (lambda (exp)
    (eq? (car exp) 'quasiquote)))

(define exp/the-env?
  (lambda (exp)
    (eq? (car exp) 'the-environment)))

(define exp/access?
  (lambda (exp)
    (eq? (car exp) 'access)))

(define exp/call-cc?
(lambda (exp)
  (eq? (car exp) 'call-with-current-continuation)))

;; These are the new structures used by the parser
(define-structure case-statement key exp-list)
(define-structure cond-statement exp-list)
(define-structure and-statement exp-list)
(define-structure or-statement exp-list)
(define-structure let-statement proc-name bindings expr)
(define-structure let*-statement bindings expr)
(define-structure letrec-statement bindings expr)
(define-structure set!-statement name expr)
(define-structure quasi exp)
(define-structure else)
(define-structure receiver proc)
(define-structure unquoted exp)
(define-structure splice exp)
(define-structure call-cc proc)
A.3 The Tail Recursive Evaluator

A.3.1 treval.scm

;;; The tail recursive evaluator

(define treval
  (lambda (exp env)
    (let loop ((cep (make-cep exp env (make-empty-continuation env))))
      (let ((exp (cep-exp cep))
        (env (cep-env cep))
        (cont (cep-cont cep)))
        (cond
          ((combination? exp)
            (loop (ev-combination (reverse (cons (combination-operator exp) (combination-operands exp)))
                  env
                  cont)))
          ((variable? exp)
            (loop (ev-variable exp env cont)))
          ((conditional? exp)
            (loop (ev-conditional exp env cont)))
          ((cond-statement? exp)
            (loop (ev-cond-statement exp env cont)))
          ((case-statement? exp)
            (loop (ev-case-statement exp env cont)))
          ((and-statement? exp)
            (loop (ev-and-statement exp env cont)))
          ((or-statement? exp)
            (loop (ev-or-statement exp env cont)))
          ((lambda? exp)
            (loop (ev-lambda exp env cont)))
          ((let-statement? exp)
            (loop (ev-let-statement exp env cont)))
          ((let*-statement? exp)
            (loop (ev-let*-statement exp env cont)))
          ((letrec-statement? exp)
            (loop (ev-letrec-statement exp env cont)))
          ((definition? exp)
            (loop (ev-definition exp env cont)))
          ((set!-statement? exp)
            (loop (ev-set!-statement exp env cont)))
          ((assignment? exp)
            (loop (ev-assignment exp env cont))))
  )
)

83
(define ev-combination
  (lambda (subexp env cont)
    (make-cep (car subexp)
      env
      (make-cont 'comb
        (make-data (cdr subexp)
          '()
          env
          cont))))
)

(define ev-variable
  (lambda (exp env cont)
    (let ((var-value (environment-lookup env (variable-name exp))))
      (let ((newcep
        (if (bad-value? var-value)
          (error "unassigned variable: " (variable-name exp))
          (make-cep var-value
            env
            cont)))
        (if (not (or (primitive-procedure? var-value)
          (compiled-procedure? var-value)))
          (print-cep 'var newcep)
          newcep)))))
)

(define ev-conditional
  (lambda (exp env cont)
    (make-cep (conditional-predicate exp)
      env
      (if (empty-continuation? cont)
        exp
        (loop (next-cep cep)))))))
)

;; ev statements used by treval

(define ev-combination
  (lambda (subexp env cont)
    (make-cep (car subexp)
      env
      (make-cont 'comb
        (make-data (cdr subexp)
          '()
          env
          cont))))
)

(define ev-variable
  (lambda (exp env cont)
    (let ((var-value (environment-lookup env (variable-name exp))))
      (let ((newcep
        (if (bad-value? var-value)
          (error "unassigned variable: " (variable-name exp))
          (make-cep var-value
            env
            cont)))
        (if (not (or (primitive-procedure? var-value)
          (compiled-procedure? var-value)))
          (print-cep 'var newcep)
          newcep)))))
)

(define ev-conditional
  (lambda (exp env cont)
    (make-cep (conditional-predicate exp)
      env
      (if (empty-continuation? cont)
        exp
        (loop (next-cep cep)))))))
)

;; ev statements used by treval

(define ev-combination
  (lambda (subexp env cont)
    (make-cep (car subexp)
      env
      (make-cont 'comb
        (make-data (cdr subexp)
          '()
          env
          cont))))
)

(define ev-variable
  (lambda (exp env cont)
    (let ((var-value (environment-lookup env (variable-name exp))))
      (let ((newcep
        (if (bad-value? var-value)
          (error "unassigned variable: " (variable-name exp))
          (make-cep var-value
            env
            cont)))
        (if (not (or (primitive-procedure? var-value)
          (compiled-procedure? var-value)))
          (print-cep 'var newcep)
          newcep)))))
)

(define ev-conditional
  (lambda (exp env cont)
    (make-cep (conditional-predicate exp)
      env
      (if (empty-continuation? cont)
        exp
        (loop (next-cep cep)))))))
(make-cont
 'if
 (make-data
 (list
 (conditional-consequent exp)
 (conditional-alternative exp))
 '())
   env
 cont))))

(define ev-cond-statement
 (lambda (exp env cont)
   (let ((exp-list (cond-statement-exp-list exp)))
     (if (null? exp-list)
         (make-cep no-value
                   env
                   cont)
         (if (else? (caar exp-list))
             (make-cep (cadar exp-list)
                       env
                       cont)
             (make-cep (caar exp-list)
                       env
                       (make-cont 'cond
                                  (make-data (list (cadar exp-list)
                                                (cdr exp-list))
                                              '())
                                 env
                                 cont))))))

(define ev-case-statement
 (lambda (exp env cont)
   (make-cep (case-statement-key exp)
             env
             (make-cont 'case
                        (make-data (case-statement-exp-list exp)
                                   '())
                        env
                        cont))))

(define ev-and-statement
 (lambda (exp env cont)
   (let ((exp-list (and-statement-exp-list exp)))
     (if (null? exp-list)
         (make-cep #t env cont)
         (make-cont (cadar exp-list)
                    env
                    cont))))
(make-cep (car exp-list)
  env
  (make-cont 'and
    (make-data (cdr exp-list)
      '()
    env
    cont)))))

(define ev-or-statement
  (lambda (exp env cont)
    (let ((exp-list (or-statement-exp-list exp)))
      (if (null? exp-list)
        (make-cep #f env cont)
        (make-cep (car exp-list)
          env
          (make-cont 'or
            (make-data (cdr exp-list)
              '()
            env
            cont))))))

(define ev-lambda
  (lambda (exp env cont)
    (make-cep (eval exp env) env cont)))

(define ev-let-statement
  (lambda (exp env cont)
    (let ((bindings (let-statement-bindings exp))
      (expr (let-statement-expr exp))
      (proc-name (let-statement-proc-name exp)))
      (cond ((and (not proc-name)
          (null? bindings))
        (make-cep expr
          (*make-environment env (vector 0))
          cont))
        ((null? bindings)
          (let* ((newenv (*make-environment env (vector 0)))
            (letproc (eval (make-lambda #f #f #f #f #f expr)
              newenv)))
            (local-assignment newenv proc-name letproc)
            (make-cep expr newenv cont)))))

  (else
    (make-cep (cadar bindings)
      env
      (make-cont 'let
        cont))))
(define ev-let*-statement
  (lambda (exp env cont)
    (let ((bindings (let*-statement-bindings exp))
      (expr (let*-statement-expr exp)))
      (if (null? bindings)
        (make-cep expr
          (*make-environment env (list->vector '0))
          cont)
        (make-cep (cadar bindings)
          env
          (make-cont 'let*
            (make-data (list (cdr bindings) expr)
              (list (caar bindings))
              env
              cont))))))

(define ev-letrec-statement
  (lambda (exp env cont)
    (let ((bindings (letrec-statement-bindings exp))
      (expr (letrec-statement-expr exp)))
      (let ((num-bindings (length bindings)))
        (if (null? bindings)
          (make-cep expr
            (*make-environment env (list->vector '0))
            cont)
          (let ((newenv (apply
                           *make-environment
                           env
                           (list->vector (cons num-bindings
                                           (map car bindings))
                           (n-bad-values num-bindings))))
              (make-cep (cadar bindings)
                newenv
                (make-cont 'letrec
                  (make-data (list (cdr bindings) expr)
                    (list (caar bindings))
                    newenv)))))
    newenv)
(define ev-definition
  (lambda (exp env cont)
    (let ((name (definition-name exp))
          (body (definition-value exp)))
      (make-cep body env
               (make-cont 'define
                          (make-data name '())
                          env
                          cont))))))

(define ev-set!-statement
  (lambda (exp env cont)
    (let ((name (set!-statement-name exp))
          (body (set!-statement-expr exp)))
      (make-cep body env
               (make-cont 'assign
                          (make-data name '())
                          env
                          cont))))))

(define ev-assignment
  (lambda (exp env cont)
    (let ((var (assignment-variable exp))
          (body (assignment-value exp)))
      (make-cep body env
               (make-cont 'assign
                          (make-data var '())
                          env
                          cont))))))

(define ev-sequence
  (lambda (exp env cont)
    (let ((explist (sequence-actions exp)))
      (make-cep (car explist) env
                (make-cont 'seq
                           exp env cont))))
(make-data
cdr explist
')()
env
cont))))

(define ev-quasi
(lambda (exp env cont)
  (let ((quasiexp (quasi-exp exp)))
    (if (list? quasiexp)
      (if (null? quasiexp)
        (make-cep '() env cont)
        (let ((revlist (reverse quasiexp)))
          (if (or (unquoted? (car revlist)) (splice? (car revlist)))
            (make-cep (make-quasi (car revlist))
              env
              (make-cont 'quasi
              (make-data (cdr revlist)
                (list (car revlist)))
              env
              cont))
            (make-cep (make-quasi (car revlist))
              env
              (make-cont 'quasi
              (make-data (cdr revlist) '())
              env
              cont))))))
  (cond ((unquoted? quasiexp)
    (make-cep (unquoted-exp quasiexp) env cont))
    ((splice? quasiexp)
    (make-cep (splice-exp quasiexp) env cont))
    (else
    (make-cep quasiexp env cont)))))))

(define ev-the-environment
(lambda (exp env cont)
  (let ((newcep (make-cep env env cont)))
    (print-cep 'the-env newcep)
    newcep)))

(define ev-access
(lambda (exp env cont)
  (make-cep (access-environment exp)
    env
    (make-cont 'access
    cont))))
(define ev-call-cc
  (lambda (exp env cont)
    (let ((newcep
           (make-cep (make-combination (call-cc-proc exp) (list cont))
                     env cont)))
     (print-cep 'call-cc newcep)
     newcep)))

;; New structures used by treval

(define-structure cont tag data env next-cont)

(define empty-continuation?
  (lambda (cont)
    (eq? (cont-tag cont) 'empty)))

(define make-empty-continuation
  (lambda (env)
    (make-cont 'empty (make-data '() '()) env '())))

(define-structure data unev ev)

(define-structure cep exp env cont)
A.3.2 next-cep.scm

;;;;; Next-cep used for pulling things off the continuation stack

(define next-cep
  (lambda (cep)
    (let ((val (cep-exp cep))
          (cont (cep-cont cep)))
      (case (cont-tag cont)
        ((comb) (next/handle-comb val cont))
        ((if) (next/handle-if val cont))
        ((cond) (next/handle-cond val cont))
        ((case) (next/handle-case val cont))
        ((and) (next/handle-and val cont))
        ((or) (next/handle-or val cont))
        ((let) (next/handle-let val cont))
        ((let*) (next/handle-let* val cont))
        ((letrec) (next/handle-letrec val cont))
        ((define) (next/handle-define val cont))
        ((assign) (next/handle-assign val cont))
        ((seq) (next/handle-seq val cont))
        ((quasi) (next/handle-quasi val cont))
        ((access) (next/handle-access val cont))
        (else (error (list "Next-cep, unknown continuation tag: "
                       (cont-tag cont))))))))

(define next/handle-comb
  (lambda (val cont)
    (let ((unev (data-unev (cont-data cont)))
          (ev (data-ev (cont-data cont)))
          (next-cont (cont-next-cont cont)))
      (if (null? unev)
          (let ((newcep (cond ((compound-procedure? val)
                               (apply-compound val ev cont))
                         ((cont? val)
                          (make-cep (car ev) (cont-env val) val))
                         (else
                          (make-cep
                          (apply val ev)
                          (cont-env next-cont)
                          next-cont))))))
          (print-cep 'comb newcep)
          newcep)
          (make-cep (car unev)
          (cont-env cont))
(define next/handle-if
  (lambda (val cont)
    (let ((newcep
            (let ((next-cont (cont-next-cont cont)))
              (if val
                (make-cep (car (data-unev (cont-data cont)))
                          (cont-env cont)
                          next-cont)
                (make-cep (cadr (data-unev (cont-data cont)))
                          (cont-env cont)
                          next-cont))))
      (print-cep 'if newcep)
      newcep))

(define next/handle-cond
  (lambda (val cont)
    (let ((newcep
            (if val
                (let ((expr (car (data-unev (cont-data cont)))))
                  (cond ((bad-value? expr)
                           (make-cep val
                                      (cont-env cont)
                                      (cont-next-cont cont)))
                           ((receiver? expr)
                            (make-cep (make-combination (receiver-proc expr)
                                        (list val))
                                      (cont-env cont)
                                      (cont-next-cont cont)))
                           (else
                            (make-cep expr
                                      (cont-env cont)
                                      (cont-next-cont cont))))
                 (let ((exp-list (cadr (data-unev (cont-data cont)))))
                   (cond ((null? exp-list)
                           (make-cep no-value
                                      (cont-env cont)
                                      (cont-next-cont cont)))
                           ((else? (caar exp-list))
                            (make-cep (cadar exp-list))))))
      newcep)))
(cont-env cont)
(cont-next-cont cont))

(else
(make-cep (caar exp-list)
(cont-env cont)
(make-cont
'cond
(make-data (list (cadar exp-list)
(cdr exp-list))
'()
(cont-env cont)
(cont-next-cont cont)))))))
(print-cep 'if newcep)
newcep))

(define next/handle-case
(lambda (val cont)

;; Find right clause takes the list of clauses and returns the
;; right expression to be evaluated.

(define find-right-clause
(lambda (clauses)
(if (null? clauses)
no-value
(let ((first-clause (car clauses)))
(if (or (else? (car first-clause))
(memv val (car first-clause)))
(cadr first-clause)
(find-right-clause (cdr clauses))))))
(let ((newcep
(let* ((clauses (data-unev (cont-data cont)))
(new-exp (find-right-clause clauses)))
(make-cep new-exp
(cont-env cont)
(cont-next-cont cont))))
(print-cep 'case newcep)
newcep))

(define next/handle-and
(lambda (val cont)
(let ((newcep
(let ((rest-exp (data-unev (cont-data cont)))
(if (or (null? rest-exp) (not val))
(make-cep val (cont-env cont) (cont-next-cont cont)))))

93
(make-cep (car rest-exp)
(cont-env cont)
(make-cont 'and
(make-data (cdr rest-exp)
'())
(cont-env cont)
(cont-next-cont cont))))
(print-cep 'and newcep)
newcep))

(define next/handle-or
(lambda (val cont)
(let ((newcep
(let (((rest-exp (data-unev (cont-data cont))))
(if (or (null? rest-exp) val)
(make-cep val (cont-env cont) (cont-next-cont cont))
(make-cep (car rest-exp)
(cont-env cont)
(make-cont 'or
(make-data (cdr rest-exp)
'())
(cont-env cont)
(cont-next-cont cont))))))
(print-cep 'or newcep)
newcep))

(define next/handle-let
(lambda (val cont)
(let (((bindings (car (data-unev (cont-data cont))))
(expr (caddr (data-unev (cont-data cont))))
(bound (data-ev (cont-data cont))
(proc-name (cadr (data-unev (cont-data cont))))
(let ((newbound (cons (list (car bound)
val)
(cdr bound)))))
(if (null? bindings)
(let ((newcep
(if (not proc-name)
(apply-let (reverse newbound) expr cont)
(let ((rbound (reverse newbound)))
(let (((boundlist (map car rbound)))
(args (map cadr rbound))
(bllength (length rbound))))
newcep))

94
(let* ((newenv (apply *make-environment
(cont-env cont)
(list->vector
(cons bllength
boundlist))
args))
(let-proc (eval (make-lambda #f
  boundlist
  #f
  #f
  #f
  #f
  expr)
newenv)))
(local-assignment newenv proc-name let-proc)
(make-cep expr
  newenv
  (cont-next-cont cont)))))))
(print-cep 'let newcep)
newcep)
(make-cep (cadar bindings)
  (cont-env cont)
  (make-cont 'let
proc-name
expr)
  (cons (caar bindings)
newbound))
  (cont-env cont)
  (cont-next-cont cont))))))))

(define next/handle-let*
(lambda (val cont)
  (let ((newcep
    (let ((bindings (car (data-unev (cont-data cont))))
      (expr (cadr (data-unev (cont-data cont))))
      (bound (data-ev (cont-data cont)))))
    (let ((newenv (*make-environment (cont-env cont)
      (vector 0 bound)
      val)))
      (if (null? bindings)
    (make-cep expr
      newenv
      (cont-next-cont cont))
    (make-cep (cadar bindings)
newenv
  (make-cont 'let*
  (make-data (list (cdr bindings) expr)
  (caar bindings))
newenv
(new-cont (cont-next-cont cont))))
  (print-cep 'let* newcep)
newcep)))

(define next/handle-letrec
  (lambda (val cont)
    (let ((newcep
      (let ((bindings (car (data-unev (cont-data
  cont))))
      (expr (cadr (data-unev (cont-data
  cont))))
      (binding-var (car (data-ev (cont-data
  cont))))
      (old-bindings (cdr (data-ev (cont-data
  cont))))
      (binding-env (cont-env cont)))
      (environment-assign! binding-env
      binding-var
      val)
      (if (null? bindings)
      (make-cep expr
      binding-env
      (cont-next-cont cont))
      (make-cep (cadar bindings)
      binding-env
      (make-cont 'letrec
      (make-data
      (list (cdr bindings) expr)
      (cons (caar bindings)
      (cons (list
      binding-var val)
      old-bindings))))
      binding-env
      (cont-next-cont cont))))
      (print-cep 'letrec newcep)
newcep)))

(define next/handle-define
  (lambda (val cont)
    (let ((newcep
      (let ((name (data-unev (cont-data cont)))
      (oldenv (cont-env cont)))
      (do-define! name val oldenv)
(make-cep no-value
   oldenv
   (cont-next-cont cont))))
(print-cep 'define newcep)
newcep))

(define next/handle-assign
  (lambda (val cont)
    (let ((newcep
           (let ((name (data-unev (cont-data cont)))
                (oldenv (cont-env cont))
                (possible-value (data-ev (cont-data cont))))
             (if (access? name)
                 (make-cep (access-environment name)
                           oldenv
                           (make-cont 'assign
                                      (make-data (access-name name)
                                                  (cons 'value val))
                           oldenv
                           (cont-next-cont cont)))
             (begin
                 (if (null? possible-value)
                     (do-assign! (variable-name name) val oldenv)
                     (do-assign! name (cdr possible-value)
                                     val))
                 (make-cep no-value
                           oldenv
                           (cont-next-cont cont))))))
(print-cep 'assign newcep)
newcep))

(define next/handle-seq
  (lambda (val cont)
    (let ((newcep
           (let ((unev (data-unev (cont-data cont)))
                (next-cont (cont-next-cont cont))))
            (if (null? unev)
                (make-cep val
                           (cont-env next-cont)
                           next-cont)
                (make-cep (car unev)
                           (cont-env cont)
                           (make-cont 'seq
                                      (make-data
(cdr unev)
',())
(cont-env cont)
(next-cont)))))))
(print-cep 'seq newcep)
newcep)))

(define next/handle-quasi
  (lambda (val cont)
    (let ((unev (data-unev (cont-data cont)))
          (ev (data-ev (cont-data cont)))
          (newev (cond ((null? ev) (list val))
                         ((unquoted? (car ev)) (cons val (cdr ev)))
                         ((splice? (car ev)) (append val (cdr ev)))
                         (else (cons val ev))))))
    (if (null? unev)
        (let ((newcep (make-cep newev
                        (cont-env cont)
                        (cont-next-cont cont))))
          (print-cep 'quasi newcep)
          newcep)
        (if (or (unquoted? (car unev)) (splice? (car unev)))
          (make-cep (make-quasi (car unev))
                    (cont-env cont)
                    (make-cont 'quasi
                    (make-data (cdr unev)
                    (cons (car unev) newev))
                    (cont-env cont)
                    (cont-next-cont cont)))))
      (make-cep (make-quasi (car unev))
                (cont-env cont)
                (make-cont 'quasi
                (make-data (cdr unev) newev)
                (cont-env cont)
                (cont-next-cont cont))))
    (make-cep (make-quasi (car unev))
              (cont-env cont)
              (make-cont 'quasi
              (make-data (cdr unev) newev)
              (cont-env cont)
              (cont-next-cont cont)))))))))

(define next/handle-access
  (lambda (val cont)
    (let ((newcep
      (let ((newval (environment-lookup val
                     (data-unev (cont-data cont)))))
        (if (bad-value? newval)
            (error "unassigned variable: " (data-unev (cont-data cont)))
            (make-cep newval
            (make-cont 'quasi
            (make-data (cdr unev) newev)
            (cont-env cont)
            (cont-next-cont cont))))))))

98
(define bind-parameters!
  (lambda (env args lamb)
    (define bind-helper
      (lambda (args individuals rest)
        (cond ((null? individuals)
           (cond ((and (null? args) (not rest)) 'ok)
                ((not rest) 'error)
                (else
                  (environment-assign! env rest args)
                  'ok)))
           ((null? args) 'error)
           (else
             (environment-assign! env (car individuals) (car args))
             (bind-helper (cdr args) (cdr individuals) rest))))
       (let ((components (lambda-components lamb list)))
         (if (eq? (bind-helper args
cadr components)
          (cadddr components))
           'ok
           (error "Procedure called with " (length args) " arguments.")))))

(define apply-compound
  (lambda (op args cont)
    (let ((openv (procedure-environment op))
      (boundlist (lambda-bound (procedure-lambda op)))
      (body (lambda-body (procedure-lambda op))))
      (let* ((bllength (length boundlist))
        (newenv (apply *make-environment
                       openv
                       (list->vector (cons bllength
                                      boundlist)))
          (n-bad-values bllength)))
       (bind-parameters! newenv args (procedure-lambda op))
       (make-cep body
                  newenv
                  (cont-next-cont cont))))))

;; procedures used by the next/handle...

(define apply-let
  (lambda (bindings expr cont)
    (let ((boundlist (map car bindings))
           (args (map cadr bindings)))
      (make-cep expr
        (apply *make-environment
          (cont-env cont)
          (list->vector (cons (length boundlist) boundlist))
          args)
          (cont-next-cont cont))))

(define do-assign!
  (lambda (name val env)
    (cond ((environment-bound? env name)
              (environment-assign! env name val))
          (else (error (list "Do-assign!, variable not bound: " name))))))

(define do-define!
  (lambda (name val env)
    (local-assignment env name val)))
A.4 The State to Scheme Expression Mapping

A.4.1 print-cep.scm

;;; The continuation unparsers

;;; This procedure determines whether or not to print

(define print-cep
 (lambda (tag cep)
  (if use-counter
   (if (or (= 1 print-every)
        (= 1 (remainder (counter) print-every)))
    (begin (newline)
     (cep->exp cep)))
   (if (memq tag print-rules)
    (begin (newline)
     (cep->exp cep)))))

;;; This will actually do the printing

(define cep->exp
 (lambda (cep)
  (fluid-let ((suffixer (if minimal-numbering
                           (make-number-for-variable)
                           (make-unique-number-for-frame)))
               (exp (cep-exp cep))
               (env (cep-env cep))
               (cont (cep-cont cep)))
   (let* ((temp-bindings (cont->unprinted-bindings cont))
          (bindings (append (environment->unprinted-letrec-bindings env)
                        temp-bindings))
          (printed-bindings (apply-print-expression-to bindings))
          (inner-exp ((print-exp env) exp))
          (whole-exp (make-whole-exp cont inner-exp)))
    (pp (list 'letrec printed-bindings whole-exp)))))))

;;; Traverses down the continuation building up a list of bindings

(define cont->unprinted-bindings
 (lambda (cont)
  (if (empty-continuation? cont)
   ()
   (let ((temp-bindings
            (cont->unprinted-bindings (cont-next-cont cont)))
    ...))
(append (environment->unprinted-letrec-bindings (cont-env cont)) temp-bindings)));

;; Traverses down continuation building up expression

(define make-whole-exp
  (lambda (cont exp)
    (let ((tag (cont-tag cont)))
      (cond ((empty-continuation? cont) exp)
            ((eq? tag 'comb) (cont/handle-combination cont exp))
            ((eq? tag 'if) (cont/handle-conditional cont exp))
            ((eq? tag 'cond) (cont/handle-cond-statement cont exp))
            ((eq? tag 'case) (cont/handle-case-statement cont exp))
            ((eq? tag 'and) (cont/handle-and-statement cont exp))
            ((eq? tag 'or) (cont/handle-or-statement cont exp))
            ((eq? tag 'let) (cont/handle-let-statement cont exp))
            ((eq? tag 'let*) (cont/handle-let*-statement cont exp))
            ((eq? tag 'letrec) (cont/handle-letrec-statement cont exp))
            ((eq? tag 'define) (cont/handle-definition cont exp))
            ((eq? tag 'assign) (cont/handle-set!-statement cont exp))
            ((eq? tag 'seq) (cont/handle-sequence cont exp))
            ((eq? tag 'quasi) (cont/handle-quasi cont exp))
            ((eq? tag 'access) (cont/handle-access cont exp))
            (else (error (list "Make-whole-exp, unrecognized continuation tag: " tag)))))));

;; The cont/handle... procedures called by make-whole-exp

(define cont/handle-combination
  (lambda (cont inner-exp)
    (make-whole-exp (cont-next-cont cont)
      (append (map (print-exp (cont-env cont))
                    (reverse (data-unev (cont-data cont))))
              (cons
               inner-exp
               (map (print-exp (cont-env cont))
                    (data-ev (cont-data cont))))))))

(define cont/handle-conditional
  (lambda (cont inner-exp)
    (if (undefined-value? (cadr (data-unev (cont-data cont))))
      (make-whole-exp (cont-next-cont cont)
        (list 'if
              inner-exp
              ...))))
(((print-exp (cont-env cont))
  (car (data-unev (cont-data cont)))))
(make-whole-exp (cont-next-cont cont)
(append (list 'if inner-exp)
(map (print-exp (cont-env cont))
  (data-unev (cont-data cont)))))

(define cont/handle-cond-statement
  (lambda (cont inner-exp)
    (define print-clause-predicate
      (lambda (pred)
        (if (else? pred)
          'else
          ((print-exp (cont-env cont)) pred))))
    (define print-clause-expr
      (lambda (expr)
        (cond ((bad-value? expr) '')
              ((receiver? expr)
                (list '=> ((print-exp (cont-env cont)) (receiver-proc expr))))
              (else
                (list ((print-exp (cont-env cont)) expr))))
    (let ((printer (print-exp (cont-env cont))))
      (make-whole-exp (cont-next-cont cont)
        (cons 'cond
          (cons inner-exp
            (map (lambda (clause)
            (cons (print-clause-predicate (car clause))
            (print-clause-expr (cadr clause))))
            (cadr (data-unev (cont-data cont))))))))
(define cont/handle-case-statement
  (lambda (cont inner-exp)
    (make-whole-exp (cont-next-cont cont)
      (cons 'case
        (cons inner-exp
          (map (lambda (1st)
            (list (if (else? (car 1st)) 'else 103
            ((print-exp (cont-env cont)) (receiver-proc expr)))))
          (let ((printer (print-exp (cont-env cont))))
            (make-whole-exp (cont-next-cont cont)
              (cons 'cond
                (cons (cons inner-exp
                  (print-clause-expr
                    (car (data-unev (cont-data cont)))))
                (map (lambda (clause)
                (cons (print-clause-predicate (car clause))
                (print-clause-expr (cadr clause))))
                (cadr (data-unev (cont-data cont))))))))}))

(define cont/handle-case-statement
  (lambda (cont inner-exp)
    (make-whole-exp (cont-next-cont cont)
      (cons 'case
        (cons inner-exp
          (map (lambda (1st)
            (list (if (else? (car 1st)) 'else 103
            ((print-exp (cont-env cont)) (receiver-proc expr)))))
            (let ((printer (print-exp (cont-env cont))))
              (make-whole-exp (cont-next-cont cont)
                (cons 'cond
                  (cons (cons inner-exp
                    (print-clause-expr
                      (car (data-unev (cont-data cont)))))
                  (map (lambda (clause)
                  (cons (print-clause-predicate (car clause))
                  (print-clause-expr (cadr clause))))
                  (cadr (data-unev (cont-data cont)))))))))))
(car lst))
  (((print-exp (cont-env cont))
    (cadr lst)))))
  (data-unev (cont-data cont)))))))

(define cont/handle-and-statement
  (lambda (cont inner-exp)
    (make-whole-exp (cont-next-cont cont)
      (cons 'and (cons inner-exp
        (map (print-exp (cont-env cont))
          (data-unev (cont-data cont))))))))))

(define cont/handle-or-statement
  (lambda (cont inner-exp)
    (make-whole-exp (cont-next-cont cont)
      (cons 'or (cons inner-exp
        (map (print-exp (cont-env cont))
          (data-unev (cont-data cont))))))))))

(define cont/handle-let-statement
  (lambda (cont inner-exp)
    (let ((cur-bind (car (data-ev (cont-data cont))))
      (done-bind (map (lambda (binding)
        (list (car binding)
          ((print-exp (cont-env cont))
            (cadr binding)))
          (reverse (cdr (data-ev (cont-data cont))))))))
      (todo-bind (map (lambda (binding)
        (list (car binding)
          ((print-exp (cont-env cont))
            (cadr binding)))
          (car (data-unev (cont-data cont))))))
      (expr (caddr (data-unev (cont-data cont))))
      (proc-name (cadr (data-unev (cont-data cont))))
        (let ((bounds (append (if proc-name (list proc-name) '()))
          (map car done-bind)
            (cons cur-bind (map car todo-bind))))))
      (let ((head (if proc-name
        (list 'let proc-name)
        (list 'let)))
        (make-whole-exp (cont-next-cont cont)
          (append
            head
            (list
              (append done-bind

104
(define cont/handle-let*-statement
  (lambda (cont inner-exp)
    (define make-let*-bindings
      (lambda (bindings bound)
        (if (null? bindings)
            ()
            (cons (list (caar bindings)
                ((print-exp-for-let (cont-env cont) bound)
                (cadar bindings)))
                (make-let*-bindings (cdr bindings)
                (cons (caar bindings) bound))))))
    (let ((cur-bind (data-ev (cont-data cont))))
      (let ((todo-bind (make-let*-bindings
                        (car (data-unev (cont-data cont)))
                        (list cur-bind)))
        (let ((bounds (cons cur-bind (map car todo-bind))))
          (make-whole-exp (cont-next-cont cont)
            (list 'let*
              (cons (list cur-bind inner-exp)
                todo-bind)
              ((print-exp-for-let (cont-env cont) bounds)
                expr))))))))

(define cont/handle-letrec-statement
  (lambda (cont inner-exp)
    (let ((done-bindings (reverse (cdr (data-ev (cont-data cont)))))
      (cur-bind (car (data-ev (cont-data cont))))
      (todo-bindings (car (data-unev (cont-data cont))))
      (let ((bounds (append (map car done-bindings)
        (cons cur-bind (map car todo-bindings))))
        (let ((done-bind (map (lambda (binding)
          (list (car binding)
            ((print-exp-for-let (cont-env cont) bounds)
              (cadr binding))))
          done-bindings))
          (todo-bind (map (lambda (binding)
            (list (car binding)
              ((print-exp-for-let (cont-env cont) bounds)
                (cadr binding))))
              done-bindings))
          (list (car binding)
            ((print-exp-for-let (cont-env cont) bounds)
              (cadr binding))))))
          ((print-exp-for-let (cont-env cont) bounds)
            (cadr binding))))))
    expr))))))))
bounds)
  (cadr binding)))
todo-bindings))
  (expr (cadr (data-unev (cont-data cont))))
(make-whole-exp (cont-next-cont cont)
(list 'letrec
(append done-bind
(cons
(list cur-bind inner-exp)
todo-bind))
((print-exp-for-let (cont-env cont) bounds)
expr)))))))

(define cont/handle-definition
  (lambda (cont inner-exp)
    (make-whole-exp (cont-next-cont cont)
      (let* ((var-name (data-unev (cont-data cont)))
             (suffixed-name (suffix-var var-name
             (suffix-value
              (suffixer var-name
                (lookup-var var-name 106
               106
              (cont-env cont))))))
             (list 'define
             suffixed-name
             inner-exp))))))

(define cont/handle-set!-statement
  (lambda (cont inner-exp)
    (make-whole-exp (cont-next-cont cont)
      (let ((name (data-unev (cont-data cont)))
           (possible-val (data-ev (cont-data cont))))
           (var-name (if (variable? name)
          (variable-name name)
          name))
           (let ((suffixed-name (suffix-var
           var-name
           (suffix-value
           (suffixer
           var-name
           (lookup-var var-name
          (cont-env cont)))))))
           (list 'set!
           ((print-exp (cont-env cont))
           name)
           inner-exp)
           (let ((possible-val (data-ev (cont-data cont)))
           (var-name (if (variable? name)
          (variable-name name)
          name))
           (let ((suffixed-name (suffix-var
           var-name
           (suffix-value
           (suffixer
           var-name
           (lookup-var var-name
          (cont-env cont)))))))
           (list 'set!
           ((print-exp (cont-env cont))
           name)
           inner-exp)
           (let ((possible-val (data-ev (cont-data cont)))
           (var-name (if (variable? name)
          (variable-name name)
          name))
           (let ((suffixed-name (suffix-var
           var-name
           (suffix-value
           (suffixer
           var-name
           (lookup-var var-name
          (cont-env cont)))))))
           (list 'set!
           ((print-exp (cont-env cont))
           name)
           inner-exp)
           (let ((possible-val (data-ev (cont-data cont)))
           (var-name (if (variable? name)
          (variable-name name)
          name))
           (let ((suffixed-name (suffix-var
           var-name
           (suffix-value
           (suffixer
           var-name
           (lookup-var var-name
          (cont-env cont)))))))
           (list 'set!
           ((print-exp (cont-env cont))
           name)
           inner-exp)
           (let ((possible-val (data-ev (cont-data cont)))
           (var-name (if (variable? name)
          (variable-name name)
          name))
           (let ((suffixed-name (suffix-var
           var-name
           (suffix-value
           (suffixer
           var-name
           (lookup-var var-name
          (cont-env cont)))))))
           (list 'set!
           ((print-exp (cont-env cont))
           name)
           inner-exp)
           (let ((possible-val (data-ev (cont-data cont)))
           (var-name (if (variable? name)
          (variable-name name)
          name))
           (let ((suffixed-name (suffix-var
           var-name
           (suffix-value
           (suffixer
           var-name
           (lookup-var var-name
          (cont-env cont)))))))
           (list 'set!
           ((print-exp (cont-env cont))
           name)
           inner-exp))

106
(cont-env cont)
'()())())
    (if (null? possible-val)
        (list 'set!
suffixed-name
inner-exp)
    (list 'set!
(list 'access
    var-name
    inner-exp)
(cdr possible-val)))
)

(define cont/handle-sequence
  (lambda (cont inner-exp)
    (make-whole-exp (cont-next-cont cont)
        (cons 'begin
            (cons inner-exp
                (map (print-exp (cont-env cont))
                    (data-unev (cont-data cont)))))))
)

(define cont/handle-quasi
  (lambda (cont inner-exp)
    (let ((unev (reverse (data-unev (cont-data cont))))
        (ev (data-ev (cont-data cont))))
        (let ((inexp (if (and (not (null? ev))
            (or (splice? (car ev))
            (unquoted? (car ev))
        (undo-listing inner-exp))))
            inner-exp
            (undo-listing inner-exp)))
        (if (eq? (cont-tag (cont-next-cont cont)) 'quasi)
            (cond ((and (not (null? ev)) (unquoted? (car ev))
                (make-whole-exp (cont-next-cont cont)
                    (append (print-unqs (cont-env cont) unev)
                        (cons list 'unquote inexp)
                            (cdr ev)))))
                ((and (not (null? ev)) (splice? (car ev)))
                    (make-whole-exp (cont-next-cont cont)
                        (append (print-unqs (cont-env cont) unev)
                            (cons list 'unquote-splicing inexp)
                                (cdr ev)))))
                (else
                    (make-whole-exp (cont-next-cont cont)
                        (append (print-unqs (cont-env cont) unev)
                            (cons inexp ev)))))))
        (cond ((and (not (null? ev)) (unquoted? (car ev))
                (make-whole-exp (cont-next-cont cont)
                    (append (print-unqs (cont-env cont) unev)
                        (cons list 'unquote-exp inexp)
                            (cdr ev))))))
        (else
            (make-whole-exp (cont-next-cont cont)
                (append (print-unqs (cont-env cont) unev)
                    (cons inexp ev)))))
    (cond ((and (not (null? ev)) (unquoted? (car ev))))

(make-whole-exp (cont-next-cont cont)
  (list 'quasiquote
  (append (print-unqs (cont-env cont)
    unev)
  (cons (list 'unquote inexexp)
    (cdr ev)))))))
((and (not (null? ev)) (splice? (car ev)))
 (make-whole-exp (cont-next-cont cont)
  (list 'quasiquote
  (append (print-unqs (cont-env cont)
    unev)
  (cons (list 'unquote-splicing
      inexp)
    (cdr ev))))))
  (else
    (make-whole-exp (cont-next-cont cont)
  (list 'quasiquote
  (append (print-unqs (cont-env cont)
    unev)
  (cons inexp ev)))))))))))

(define print-unqs
  (lambda (env exp)
    (cond ((list? exp)
      (map (lambda
        (x)
        (print-unqs env x)) exp))
    ((or (splice? exp) (unquoted? exp))
      ((print-exp env) exp))
    (else exp))))

(define undo-listing
  (lambda (exp)
    (cond ((and (list? exp) (not (null? exp)) (eq? (car exp) 'list))
      (map undo-listing (cdr exp)))
    ((and (symbol? exp) (string=? (string-head (symbol->string exp) 1)
        ""))
      (string->symbol (substring (symbol->string exp)
        1
        (string-length (symbol->string exp))))))
    (else
      exp))))

(define cont/handle-access
  (lambda (cont inner-exp)
    (make-whole-exp (cont-next-cont cont)
    (list 'access
        (list 'unquote-splicing
        inexp)
    (cdr ev)))))))

108
(data-unev (cont-data cont)
inner-exp))))

;; Used to determine which lines to print

(define make-counter
(lambda (n)
  (lambda ()
    (set! n (+ n 1))
    n)))

(define use-counter #t)

(define print-every 1)

(define print-rules '(comb var if cond case and or let let* letrec
define assign seq the-env access))
A.4.2 bindings.scm

;;;; Procedures for making and suffixing the list of bindings

(define environment->unprinted-letrec-bindings
  (lambda (env)
    (if (equal? (environment-parent initial-environment) env)
      (),
      (let ((parent-bindings (environment->unprinted-letrec-bindings
                               (environment-parent env))))
        (append (filter-and-suffix (environment-bindings env) env)
                parent-bindings))))))

(define filter-and-suffix
  (lambda (bindings-list env)
    (if (null? bindings-list)
      (),
      (let ((first-bnd (car bindings-list))
            (rest-bnd (cdr bindings-list)))
        (if (null? (cdr first-bnd))
          (filter-and-suffix rest-bnd env)
          (let* ((bnd-var (car first-bnd))
                  (bnd-val (cadr first-bnd))
                  (suffix (suffixer bnd-var env)))
            (if (or (eq? (suffix-tag suffix) 'old)
                    (bad-value? bnd-val))
                (filter-and-suffix rest-bnd env)
                (cons (list (suffix-var bnd-var (suffix-value suffix))
                         bnd-val
                         env)
                      (cond
                       ((compound-procedure? bnd-val)
                        (append (filter-and-suffix rest-bnd env)
                                (environment->unprinted-letrec-bindings
                                 (procedure-environment bnd-val)))
                        (else (filter-and-suffix rest-bnd env))))))))))

(define apply-print-expression-to
  (lambda (unprinted-bindings-list)
    (if (null? unprinted-bindings-list)
      (),
      (let ((first-bnd (car unprinted-bindings-list))
            (rest-bnd (cdr unprinted-bindings-list)))
        (let ((bnd-var (car first-bnd))
              (bnd-val (cadr first-bnd)))
          (bnd-val (cdr first-bnd))))))
(env (caddr first-bnd)))
      
      (cond ((compound-procedure? bnd-val)
            (cons (list bnd-var
                  (print-expression
                  (procedure-lambda bnd-val)
                  (procedure-environment bnd-val)
                  '()))
            (apply-print-expression-to rest-bnd))
            (else
            (cons (list bnd-var
                  (print-expression bnd-val
                  env
                  '()))
            (apply-print-expression-to rest-bnd)))))))

;; The suffixing procedures

(define suffixer)

(define minimal-numbering #t)

(define-structure suffix tag value)

(define suffix-vars
      (lambda (var-list suffix)
      (if (null? var-list)
            '()
            (cons (suffix-var (car var-list) suffix)
                  (suffix-vars (cdr var-list) suffix)))))

(define suffix-var
      (lambda (sym suffix)
      (if (and (number? suffix) (zero? suffix))
            sym
            (string->symbol
            (string-append
            (symbol->string sym)
            "_
            (if (number? suffix)
            (number->string suffix)
            suffix))))))

; This hash table is for minimal numbering.

(define make-number-for-variable

(lambda ()
  (let* ((state 0)
         (new-int (lambda ()
                    (set! state (+ 1 state))))
        (old 112)
        (old-number (cons var env))
        (new-number (cons var env))
        (new-old-number (cons var env))
        (new-old-old-number (cons var env))
        (new-old-old-old-number (cons var env)))
    (begin
      (hash-table/remove! the-var-table var)
      (hash-table/put! the-var-table var (+ new-number 1))
      (hash-table/put! the-var-frame-table var env
                    (+ new-number 1))
      (make-suffix 'new (+ new-number 1)))))
  (begin
    (hash-table/put! the-var-table var 0)
    (hash-table/put! the-var-frame-table var env
                    0)
    (make-suffix 'new 0)))))))))))

; This hash table assigns a suffix to a variable based on what
; frame it is in.

(define make-unique-number-for-frame
  (lambda ()
    (let ((the-var-table ((strong-hash-table/constructor
        (lambda (thing size) (remainder (hash thing) size))
        eq? #t))
           (the-var-frame-table ((strong-hash-table/constructor
            (lambda (thing size)
              (remainder (+ (hash (car thing)) (hash (cdr thing))
                         size))
            (lambda (x y)
              (and (eq? (car x) (car y))
                   (equal? (cdr x) (cdr y))))
            #t))))
      (lambda (var env)
        (let ((new-num (hash-table/get the-var-table var #f)))
          (if new-num
              (let ((old-num (hash-table/get the-var-frame-table
                                             (cons var env)
                                             #f)))
                (if old-num
                    (make-suffix 'old old-num)
                    (begin
                      (hash-table/remove! the-var-table var)
                      (hash-table/put! the-var-table var (+ new-num 1))
                      (hash-table/put! the-var-frame-table var env
                                      (+ new-num 1))
                      (make-suffix 'new (+ new-num 1))))))
              (begin
                (hash-table/put! the-var-table var 0)
                (hash-table/put! the-var-frame-table var env
                                0)
                (make-suffix 'new 0))))))))))
state))
  (my-hash (lambda (k size) (remainder (hash k) size)))
  (the-table ((strong-hash-table/constructor
  my-hash
  equal?
  #t))))
  (lambda (var-name the-frame)
  (let ((val (hash-table/get the-table the-frame #f)))
    (if val
      (make-suffix 'old val)
      (begin
        (let ((suffix (new-int)))
          (hash-table/put! the-table the-frame suffix)
          (make-suffix 'new suffix))))))))
(define print-expression
  (lambda (exp env bound)
    (cond ((null? exp) (handle-null))
          ((symbol? exp) (handle-symbol exp))
          ((list? exp) (handle-list exp env bound))
          ((pair? exp) (handle-pair exp env bound))
          ((primitive-procedure? exp) (handle-primitive-procedure exp))
          ((compiled-procedure? exp) (handle-compiled-procedure exp))
          ((combination? exp) (handle-combination exp env bound))
          ((variable? exp) (handle-variable exp env bound))
          ((conditional? exp) (handle-conditional exp env bound))
          ((cond-statement? exp) (handle-cond-statement exp env bound))
          ((case-statement? exp) (handle-case-statement exp env bound))
          ((and-statement? exp) (handle-and-statement exp env bound))
          ((or-statement? exp) (handle-or-statement exp env bound))
          ((lambda? exp) (handle-lambda exp env bound))
          ((let-statement? exp) (handle-let-statement exp env bound))
          ((let*+statement? exp) (handle-let*-statement exp env bound))
          ((letrec-statement? exp) (handle-letrec-statement exp env bound))
          ((definition? exp) (handle-definition exp env bound))
          ((set!-statement? exp) (handle-set!-statement exp env bound))
          ((assignment? exp) (handle-assignment exp env bound))
          ((sequence? exp) (handle-sequence exp env bound))
          ((quasi? exp) (handle-quasi exp env bound))
          ((unquoted? exp) (handle-unquoted exp env bound))
          ((splice? exp) (handle-splice exp env bound))
          ((access? exp) (handle-access exp env bound))
          ((the-environment? exp) (handle-the-environment))
          ((call-cc? exp) (handle-call-cc exp env bound))
          ((cont? exp) (handle-cont exp env bound))
          (else (handle-value exp)))))

(define print-exp
  (lambda (env)
    (lambda (exp)
      (if (compound-procedure? exp)
          (print-expression
            (procedure-lambda exp)
            (procedure-environment exp)
            '())
          (print-expression exp env '())))))
(define print-exp-for-let
  (lambda (env bound)
    (lambda (exp)
      (if (compound-procedure? exp)
          (print-expression
            (procedure-lambda exp)
            (procedure-environment exp)
            bound)
          (print-expression exp env bound))))

(define handle-null
  (lambda ()
    #f))

(define handle-symbol
  (lambda (sym)
    (string->symbol (string-append "'
      (symbol->string sym)))))

(define handle-list
  (lambda (lst env bound)
    (cons 'list
      (map (lambda (exp) (print-expression exp env bound)) lst))))

(define handle-pair
  (lambda (pair env bound)
    (list 'cons
      (print-expression (car pair) env bound)
      (print-expression (cdr pair) env bound))))

(define handle-primitive-procedure
  (lambda (prim-proc)
    (primitive-procedure-name prim-proc)))

(define handle-compiled-procedure
  (lambda (comp-proc)
    (lambda-name (compiled-procedure/lambda comp-proc))))

(define handle-combination
  (lambda (com env bound)
    (cons (print-expression (combination-operator com) env bound)
      (map (lambda (sub-exp)
        (print-expression sub-exp env bound))
        (combination-operands com))))
(define handle-variable
  (lambda (var env bound)
    (let* ((var-name (variable-name var))
           (frame (lookup-var var-name env bound)))
      (if frame
          (suffix-var var-name (suffix-value (suffixer var-name frame)))
          var-name)))

(define lookup-var
  (lambda (var-name env bound)
    (define lookup-var-helper
      (lambda (var-name env)
        (cond ((equal? (environment-parent initial-environment) env) #f)
              ((memq var-name (environment-bound-names env)) env)
              (else (lookup-var-helper var-name (environment-parent env))))))
    (if (memq var-name bound)
        #f
        (lookup-var-helper var-name env))))

(define handle-conditional
  (lambda (con env bound)
    (let ((alt (conditional-alternative exp)))
      (if (bad-value? alt)
          (list 'if
                (print-expression (conditional-predicate con) env bound)
                (print-expression (conditional-consequent con) env bound))
          (list 'if
                (print-expression (conditional-predicate con) env bound)
                (print-expression (conditional-consequent con) env bound)
                (print-expression (conditional-alternative con) env bound))))))

(define handle-cond-statement
  (lambda (con env bound)
    (cons 'cond (map (lambda (clause)
                      (cons (if (else? (car clause))
                              'else
                              (print-expression (car clause) env bound))
                          (cond ((bad-value? (cadr clause))
                                  '())
                                  (print-expression (cadr clause) env bound)))
                        clause))
            (print-expression (car clause) env bound))))
(((receiver? (cadr clause))
   (list ’=> (print-expression
         (receiver-proc (cadr clause))))))
(else
   (list (print-expression (cadr clause)
      env bound)))))
  (cond-statement-exp-list con)))))

(define handle-case-statement
  (lambda (caseexp env bound)
    (cons ’case (cons (print-expression (case-statement-key caseexp)
       env bound)
      (map (lambda (clause)
            (list (if (else? (car clause))
                   ’else
                   (car clause))
            (print-expression (cadr clause)
               env bound)))
      (case-statement-exp-list caseexp)))))
)

(define handle-and-statement
  (lambda (andexp env bound)
    (cons ’and (map (lambda (exp) (print-expression exp env bound))
        (and-statement-exp-list andexp)))))

(define handle-or-statement
  (lambda (orexp env bound)
    (cons ’or (map (lambda (exp) (print-expression exp env bound))
        (or-statement-exp-list orexp)))))

(define handle-lambda
  (lambda (lam env bound)
    (let ((lambda-comp-list
            (lambda-components lam list)))
      (let ((params (cadr lambda-comp-list))
            (param-list (cadddr lambda-comp-list)))
        (list ’lambda
             (if (null? param-list)
                 (if (null? params)
                     (string->symbol "()")
                     params)
                 (append params (list ’. param-list)))
             (print-expression (lambda-body lam)
               env
               (if (null? param-list)
(define handle-let-statement
  (lambda (letexp env bound)
    (let ((head (if (let-statement-proc-name letexp)
           (list 'let (let-statement-proc-name letexp))
           (list 'let)))
      (append
       head
       (list (map (lambda (binding)
                   (list (car binding)
                         (print-expression (cadr binding) env bound)))
              (let-statement-bindings letexp))
       (print-expression
        (let-statement-expr letexp)
        env
       (append (map car (let-statement-bindings letexp))
               bound)))))))

(define handle-let*-statement
  (lambda (let*exp env bound)
    (define make-let*-bindings
      (lambda (bindings bound)
        (if (null? bindings)
            '()
            (cons (list (caar bindings)
                       (print-expression (cadar bindings) env bound))
                  (make-let*-bindings (cdr bindings) (cons (caar bindings)
                                                          bound))))))
    (list 'let*
           (make-let*-bindings (let*-statement-bindings let*exp) bound)
           (print-expression (let*-statement-expr let*exp)
                              env
           (append (map car (let*-statement-bindings let*exp))
                   bound))))))

(define handle-letrec-statement
  (lambda (letrecexp env bound)
    (let ((newbounds (append (map car (letrec-statement-bindings
               letrecexp))
                          bound))
      (list 'letrec
            (map (lambda (binding)
                   (list (car binding)
                         (print-expression (cadr binding) env bound)))))))

(append params bound)
(append (cons param-list params)
        bound))))))))

(define handle-let-statement
  (lambda (letexp env bound)
    (let ((head (if (let-statement-proc-name letexp)
           (list 'let (let-statement-proc-name letexp))
           (list 'let)))
      (append
       head
       (list (map (lambda (binding)
                   (list (car binding)
                         (print-expression (cadr binding) env bound)))
              (let-statement-bindings letexp))
       (print-expression
        (let-statement-expr letexp)
        env
       (append (map car (let-statement-bindings letexp))
               bound)))))))

(define handle-let*-statement
  (lambda (let*exp env bound)
    (define make-let*-bindings
      (lambda (bindings bound)
        (if (null? bindings)
            '()
            (cons (list (caar bindings)
                       (print-expression (cadar bindings) env bound))
                  (make-let*-bindings (cdr bindings) (cons (caar bindings)
                                                          bound))))))
    (list 'let*
           (make-let*-bindings (let*-statement-bindings let*exp) bound)
           (print-expression (let*-statement-expr let*exp)
                              env
           (append (map car (let*-statement-bindings let*exp))
                   bound))))))

(define handle-letrec-statement
  (lambda (letrecexp env bound)
    (let ((newbounds (append (map car (letrec-statement-bindings
               letrecexp))
                          bound))
      (list 'letrec
            (map (lambda (binding)
                   (list (car binding)
                         (print-expression (cadr binding) env bound)))))))

(append params bound)
(append (cons param-list params)
        bound))))))))

(define handle-let-statement
  (lambda (letexp env bound)
    (let ((head (if (let-statement-proc-name letexp)
           (list 'let (let-statement-proc-name letexp))
           (list 'let)))
      (append
       head
       (list (map (lambda (binding)
                   (list (car binding)
                         (print-expression (cadr binding) env bound)))
              (let-statement-bindings letexp))
       (print-expression
        (let-statement-expr letexp)
        env
       (append (map car (let-statement-bindings letexp))
               bound)))))))

(define handle-let*-statement
  (lambda (let*exp env bound)
    (define make-let*-bindings
      (lambda (bindings bound)
        (if (null? bindings)
            '()
            (cons (list (caar bindings)
                       (print-expression (cadar bindings) env bound))
                  (make-let*-bindings (cdr bindings) (cons (caar bindings)
                                                          bound))))))
    (list 'let*
           (make-let*-bindings (let*-statement-bindings let*exp) bound)
           (print-expression (let*-statement-expr let*exp)
                              env
           (append (map car (let*-statement-bindings let*exp))
                   bound))))))

(define handle-letrec-statement
  (lambda (letrecexp env bound)
    (let ((newbounds (append (map car (letrec-statement-bindings
               letrecexp))
                          bound))
      (list 'letrec
            (map (lambda (binding)
                   (list (car binding)
                         (print-expression (cadr binding) env bound)))))))

(append params bound)
(append (cons param-list params)
        bound))))))))

(define handle-let-statement
  (lambda (letexp env bound)
    (let ((head (if (let-statement-proc-name letexp)
           (list 'let (let-statement-proc-name letexp))
           (list 'let)))
      (append
       head
       (list (map (lambda (binding)
                   (list (car binding)
                         (print-expression (cadr binding) env bound)))
              (let-statement-bindings letexp))
       (print-expression
        (let-statement-expr letexp)
        env
       (append (map car (let-statement-bindings letexp))
               bound)))))))

(define handle-let*-statement
  (lambda (let*exp env bound)
    (define make-let*-bindings
      (lambda (bindings bound)
        (if (null? bindings)
            '()
            (cons (list (caar bindings)
                       (print-expression (cadar bindings) env bound))
                  (make-let*-bindings (cdr bindings) (cons (caar bindings)
                                                          bound))))))
    (list 'let*
           (make-let*-bindings (let*-statement-bindings let*exp) bound)
           (print-expression (let*-statement-expr let*exp)
                              env
           (append (map car (let*-statement-bindings let*exp))
                   bound))))))

(define handle-letrec-statement
  (lambda (letrecexp env bound)
    (let ((newbounds (append (map car (letrec-statement-bindings
               letrecexp))
                          bound))
      (list 'letrec
            (map (lambda (binding)
                   (list (car binding)
                         (print-expression (cadr binding) env bound)))))))
(list (car binding)
  (print-expression
    (cadr binding)
    env
    newbounds))
(letrec-statement-bindings letrexp)
  (print-expression (letrec-statement-expr letrexp)
    env
    newbounds)))))

(define handle-definition
  (lambda (def env bound)
    (if (bad-value? bound)
      (list 'define (definition-name def))
      (list 'define (definition-name def)
        (print-expression (definition-value def) env bound))))))

(define handle-set!-statement
  (lambda (ass env bound)
    (list 'set!
      (print-expression (set!-statement-name ass) env bound)
      (print-expression (set!-statement-expr ass) env bound))))))

(define handle-assignment
  (lambda (ass env bound)
    (list 'set!
      (assignment-name ass)
      (print-expression (assignment-value ass) env bound))))))

(define handle-sequence
  (lambda (seq env bound)
    (cons 'begin
      (map
        (lambda (action) (print-expression action env bound))
        (sequence-actions seq))))))

(define handle-quasi
  (lambda (qua env bound)
    (list 'quasiquote (unparse-quasi (quasi-exp qua) env bound))))))

(define unparse-quasi
  (lambda (exp env bound)
    (cond ((list? exp)
      (map (lambda (x) (unparse-quasi x env bound)) exp))
      ((unquoted? exp)

119
(list 'unquote (print-expression (unquoted-exp exp) env bound)))
((splice? exp)
 (list 'unquote-splicing (print-expression
 (splice exp) env bound)))
(else exp)))))

(define handle-unquoted
 (lambda (unq env bound)
   (list 'unquote (print-expression (unquoted-exp unq) env bound))))

(define handle-splice
 (lambda (spl env bound)
   (list 'unquote-splicing (print-expression
   (unquoted-exp spl) env bound))))

(define handle-access
 (lambda (exp env bound)
   (list 'access
     (access-name exp)
     (print-expression (access-environment exp) env
       bound))))

(define handle-the-environment
 (lambda ()
   (list 'the-environment)))

(define handle-call-cc
 (lambda (exp env bound)
   (list 'call-with-current-continuation
     (print-expression (call-cc-proc exp) env bound))))

(define handle-cont
 (lambda (exp env bound)
   (list '**
     (make-whole-exp exp (string->symbol "[ hole ]") env)
     '**))))

(define handle-value
 (lambda (exp)
   exp))
Bibliography


