Scheme Modeled as a Set of Rewrite Rules: An Efficient Implementation

by

Paul Agustin Van Eyk

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

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

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

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Scheme Modeled as a Set of Rewrite Rules: An Efficient Implementation

by

Paul Agustin Van Eyk

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, another student and I designed and implemented a scheme interpreter which allows the user to see programs being evaluated according to the substitution model. The interpreter itself is tail recursive and maintains explicit continuation and environment information. Along with this, we implemented modules to print this explicit state as a scheme expression. Such an expression represents a step in the evaluation of the original program according to the substitution model. Some sample programs were executed as simple tests of correctness.

Thesis Supervisor: Albert R. Meyer
Title: Hitachi America Professor of Engineering

0.1 Acknowledgments

I would like to thank my partner, Abe, for writing half of the code or more, and for putting up with me for a whole term. Speaking of which, I'd also like to thank my advisor, Albert, for accommodating our schedule and providing an excellent project.

Contents

Chapter 1

Introduction

The computer language Scheme can be used to describe the processes that perform computations. Without a model for evaluating Scheme programs, however, the language would be meaningless. Fortunately, such models exist. The substitution model provides rules for evaluating Scheme expressions, while the environment model is a tool for resolving an occurance of a variable to its value. Understanding these models is a vital step towards understanding the processes that Scheme describes.

The Tail Recursive Evaluator (TREval) was designed as a tool to help students to understand these models. By allowing students to execute programs, and showing them the evaluation of such programs according to the substitution model, the evaluator will help them to clear up confusions and misconceptions about Scheme.

This paper will describe TREval and the accompanying code that allows substitution model evaluation. The rest of this introduction consists of a description of the substitution and environment models, while the later chapters will proceed with the description of the evaluator.

Chapter two will discuss the underlying infrastructure supporting three evaluators: the meta-circular evaluator, the explicit control evaluator, and the tail recursive evaluator. Some key features of the tail recursive evaluator will be discussed, in comparison with its two counterparts.

Chapter three will examine the tail recursive evaluator in detail. The evaluation strategy and data structures of the evaluator will be discussed. Some specific examples

of special forms will be discussed.

Chapter four will show why TREval can be used to show the substitution model evaluation for a given program. This will include an exploration of the relationship between the evaluation strategy of TREval, and the environment and substitution models.

Chapter five will explain how the state of TREval can be printed as a Scheme expression. This is the mechanism which is necessary to see substitution model evaluations.

Chapter six will present some key examples of TREval in action.

Chapter seven will conclude with a summary and some final remarks.

1.1 The Substitution Model

The substitution model consists of a set of rules of evaluation that describe the behavior of Scheme expressions. For every expression type, there is a corresponding rule of evaluation. These rules specify how an expression E may be reduced to a simpler expression. Along the way, the pieces that comprise E may themselves require evaluation, using the rules of the model. When such a subexpression is encountered, it is called a **subproblem**, and when a simpler expression is derived from the whole, that is called a **reduction**. The basic idea of the substitution model is that E may be reduced to its value by repeatedly performing subproblems and reductions on the pieces of E.

For example, suppose that you are on your way to the beach, and you need to decide whether to bring your bathing suit or your umbrella. You might consider the following expression, "If it is raining, then bring my umbrella, otherwise bring my bathing suit." You would then have to decide whether it was raining. That is a subproblem, a piece of the whole expression that you would work on for a while, then come back with an answer. Say it is a bright, sunny day. You could now decide to deal with a simpler problem, "Bring my bathing suit." This is a reduction of the original expression.

Notice the rules that we used in that example. The original problem was, "If it is raining, then bring my umbrella, otherwise bring my bathing suit." To evaluate, we first decided on the weather, and since it was sunny, reduced the entire problem to bringing our bathing suit. This would be the basis for a rule for evaluating such an expression in the model of evaluation for this language. We would also need a rule to determine if it was raining, and one to tell us how to bring an item. Some tasks would be so simple we would not need a rule. These tasks would be the values of our language, the expressions that do not require any further evaluation.

The rest of this section describes the most common types of Scheme expressions, and the rules that correspond to those expressions. It will start by dividing all expressions into four categories, and will go on to discuss some more specific expression types. For a more complete description of the substitution model, refer to the book by Abelson and Sussman [1].

1.1.1 Four Types of Scheme Expressions

Every Scheme expression falls into one of four categories. There are two primitive types of expressions: **constants**, such as 3 and #t, and **variables**, such as foo and bar. There also are two compound types: **combinations**, such as `(+ 2 3)`, and **special forms**, such as `(define x 5)`. This section explains how each of these is handled in the substitution model.

Constants

The first (and simplest) expression type is a constant. As the name implies, these expressions have values that never change with time. The expression `5`, for instance, evaluates to the number 5, always. In the substitution model, these expressions are considered to be values. In other words, the rule for evaluation of these expressions is to stop evaluating.

Variables

The next kind of expression is a variable. A variable in Scheme is any legal identifier. To evaluate a variable, find the value to which that variable is bound, and replace the variable name with that value. For example, if the variable `x` had the value 5, then anywhere the expression `x` was evaluated, it would be replaced with 5. The mechanisms by which variables are bound and looked up will be discussed in the next section on the environment model later in the chapter.

Special Forms

The remaining two expression types fall into the category of compound expressions. They are special forms and combinations. These expressions always start with an open paren, and usually are composed of several pieces. This section will discuss special forms, and the next will discuss combinations.

A special form is any compound expression in which a magic keyword immediately follows the open paren. Examples of these keywords are define and lambda. For a complete list of these magic words, see the Scheme reference manual [2]. For each special form, there is a special rule of evaluation. To evaluate an expression which is a special form, the special rule must be followed. For example, the special form if has a fairly typical rule. An if expression has the form

`(if <predicate> <consequent> <alternative>)`

where the three subforms predicate (P), consequent (C), and alternative (A) are arbitrary Scheme expressions. The rule to evaluate an expression says to evaluate P, and decide whether the value of P is false. If P is false, then the expression reduces to A, otherwise it reduces to C. Notice that the evaluation of P is a subproblem, and that the evaluations of C or A are reductions.

Another interesting special form is lambda. The most common lambda expressions take the form:

`(lambda <parameters> <body>)`

where <parameters> is a list of variable names, and <body> is a sequence of Scheme

expressions. The value of a lambda expression is a Scheme procedure, whose parameters and body are the corresponding parts of the lambda expression. Thus, the rule for lambda in the substitution model is to replace the lambda expression with some written representation of a procedure. In TREval, this representation is just the lambda expression itself, so that the value of `(lambda (x) (+ 2 x))` is just (lambda (x) (+ 2 x)).

There are many other special forms in Scheme. Two notable omissions here are define and set!, and others include and, or, case, let, and cond. Some of these will be discussed in the section on the environment model, as they deal with variables and bindings. For a complete description of the Scheme special forms and their behavior, see the Scheme reference manual [2].

Combinations

The final type of Scheme expression is a combination. A combination looks like:
`(<operator> [<operand> ...])`
where <operator> and <operand> can be arbitrary Scheme expressions, and there are zero or more operands. The rule to evaluate a combination is to evaluate all of the subexpressions, in some arbitrary order, then to "apply" the result of the operator to the results of the operands. The operator expression must evaluate to a procedure, whose parameter list is the same length as the number of operands in the combination. The entire expression reduces to the body of that procedure, with one important change. All free occurances of the variable names present in the parameter list in that body are replaced with the value of the corresponding operand. For example, consider the expression:
`((lambda (x y) (+ x y)) 2 3)`.
The operator evaluates to a procedure, the operands evaluate to 2 and 3. The expression will reduce to `(+ x y)`, but with 2 replacing all the free occurances of `x`, and 3 replacing those of `y`. So, the final reduction is `(+ 2 3)`, which would then evaluate to 5.

Sometimes, the operator will be a Scheme primitive procedure, such as + or cons.

In that case, the rule is to just perform the magic of that procedure, and give back the answer as the reduction.

1.1.2 Conclusion

As a conclusion to this section, I will present a simple example of a Scheme evaluation, step by step using the substitution model. This is something that can be done using TREval, and at this point it may be worthwhile to do so. Consider the expression:
`((lambda (x y) (if (> x y) (- x y) (- y x))) 3 8)`.
We first note that the expression is a combination, since it has the form (`<operator>` `<operand>` `<operand>`). Using the rule for combinations, we will encounter the following 3 subproblems:

```
   (lambda (x y) (if (> x y) (- x y) (- y x)))
3
8
```

We can evaluate these in any order we choose, so let's start with the easy ones. The value of `3` is 3, and the value of `8` is 8. These expressions are constants. The operator is a lambda expression, whose rule is to make a procedure. Thus, the result of the operator is the procedure of two arguments, x and y, whose body is an if expression. Finally, by applying the operator to the operands, we should reduce this expression to the simpler expression:
`(if (> 3 8) (- 3 8) (- 8 3))`

We now have an if special form, which has its own rule. The first step is to evaluate the predicate, which is the combination `(> 3 8)`. A Scheme primitive procedure, >, is the operator. So, we perform the magic (after evaluating all of the subexpressions, of course!) and we come back with false. We now can reduce the entire if expression to the alternative, `(- 8 3)`. When we evaluate this, we get the value, 5. To summarize:

```
Original expression:  ((lambda (x y) (if (> x y) (- x y) (- y x))) 3 8)
Subproblem:           3
Subproblem:           8
Subproblem:           (lambda (x y) (if (> x y) (- x y) (- y x)))
Reduction:            (if (> 3 8) (- 3 8) (- 8 3))
Subproblem:           (> 3 8)
Reduction:            (- 8 3)
Reduction:            5
```

For the sake of simplicity, the subproblems of evaluating the operands in the simple mathematical expressions have been omitted. This example illustrates the key ideas of the substitution model. The rules for if, lambda, combinations, and constants are shown, as well as the dynamics of subproblems and reductions. It is, however, a very simple example, and does not really demonstrate how useful this model can really be.

Mastery over the substitution model is fundamental to understanding Scheme programs and the processes they create. The model not only allows the derivation of the value of a program, but also illustrates how that answer is computed. Such features as stack usage and deferred operations can be brought to light with this model. In short, it is an essential tool for gaining insight into Scheme.

However, this model does have some limitations. You may have noticed that I have cleverly avoided certain issues in this section. Most notably, I have not specified how to determine the value of a variable. While rules for lexical scoping can be used to solve this problem within the framework of the substitution model, there is a more specialized model, called **the environment model**, that people use to handle this issue.

1.2 The Environment Model

The environment model is a tool that keeps track of the bindings of variables to their values. In this model, Scheme expressions are evaluated according to the substitution model, but with some additional rules for building up an environment structure. When an expression is a variable, the value of that expression is computed by finding the binding of that variable with respect to the current frame.

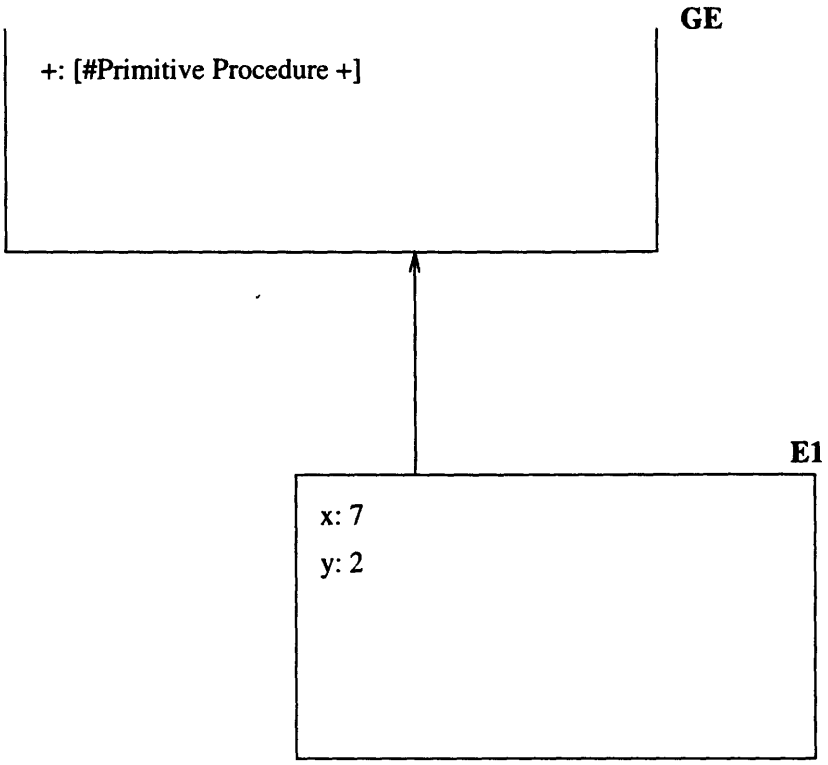

Figure 1: A frame, E1, with two bindings, x and y, and the global environment as its parent.

The main building block of the environment model is called a **frame**. A frame is simply a collection of **bindings**, with a pointer to some parent frame. A binding is an association between a variable name and some value. For instance, some frame may contain a binding between the variable x and the value 3. In this case, if we were to look up the value of x with respect to the frame, we would find the value 3. If we did not find a binding for x, we would recursively look up x with respect to the parent frame, until we either found a binding, or we reached the **Global Environment**, when we would signal an unbound variable.

Another component of the environment model is called a procedure object. This is a pictorial representation of a Scheme procedure (it is often referred to as a "double bubble"). A procedure object has two components: the text of the procedure, and the environment pointer. The text of the procedure keeps track of the parameters

and the body, while the environment pointer keeps track of the frame in which the procedure was defined (ie, where the lambda expression was evaluated).

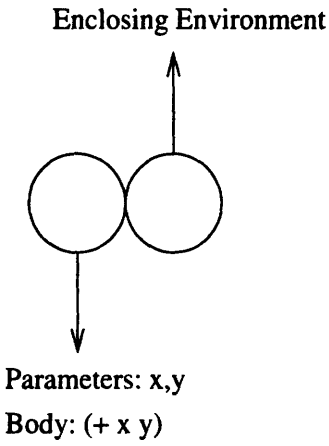

Figure 2: A pictorial representation of a procedure object, with parameters x and y, and body (+ x y).

In the environment model, every expression will be evaluated with respect to some frame. Initially, evaluation starts in a special frame, called the Global Environment (GE for short), which contains bindings for things like +, cons, etc. During the computation, however, new frames may be created, and evaluation may be moved to these new frames. In any case, however, the actual evaluation proceeds **according to the rules of the substitution model.**

The descriptions that follow, then, provide rules for building up the environment structure as an evaluation proceeds. They are not complete evaluation rules, since they all are based on the rules from the underlying model. Rather, they are auxiliary rules that will help to identify the values of variables. The expressions whose rules are discussed are variables, define, set!, let, lambda, and procedure applications.

1.2.1 Variables

The reason for the environment model is to help in the lookup of variables. If a variable expression is evaluated with respect to some environment, it triggers a recursive

traversal of the frames. We look for the variable in the current frame. If it is there, then we are finished. Otherwise, we move to the parent of our frame, where we will again look for a binding. This process will continue until we either find a binding and return the value, or until we try to move to the parent of GE, when we return an unbound variable error. This recursive traversal is a common technique in the environment model, and is an essential part of its mechanics.

1.2.2 Define

Define expressions are used to create a binding between a variable and a value. They take the following form:

`(define <name> <expression>)`.

In the substitution model, the expression is evaluated, and a binding is created between the name and the value of that expression. So, the expression `(define x 5)` would create a binding between x and 5.

In the environment model, there is a rule for dealing with define expressions. When such an expression is evaluated with respect to some frame, a new binding is created, in the current frame, between that name and the value of the given expression. The key difference, here, is that we create a binding **in the current frame**. To show this, people will draw the variable x in the current frame, with an arrow to its value, 5. This way, when another expression uses x, it will find its binding in this frame.

1.2.3 Set!

The special form set! looks and feels very much like define. There is, however, a very fundamental difference: set! **changes** the binding of the variable, while define **creates** a new binding for the variable. In the environment model, this means that a set! will find the binding for the variable in the current environment, and alter that binding to reflect the value in the set!.

For exaple, suppose frame E1, pointing to GE, has no bindings in it, and, in GE, there is a binding of x to 5. If we evaluate `(set! x 1)` with respect to E1, it will

trigger the following process. First, E1 is searched for a binding for x. Since x is not bound in this frame, the parent frame, GE, will be searched. In GE, the binding for x will be found, and changed to be 1. We will be left with E1 still empty, and GE with x bound to 1.

If, on the other hand, if we do `(define x 1)` inside of E1, then a new binding will be created, in E1, between x and 1. We then have two bindings for x, 1 inside of E1 and 5 in GE.

Notice how set! acts like a variable lookup; it must find the binding of the variable in order to change it. In this case, the value of x in the frame E1 was resolved by the binding of x to 5 in GE.

1.2.4 Let

Let is a very useful special form that allows the use of local variables to capture values. The syntax of a let is

`(let ([<binding> ... ]) <body>)`

where there are zero or more bindings, of the form `(<variable> <value>)`. The rule for evaluating a let in some environment E1 is the following. In any order, evaluate the values of the bindings, with respect to E1. Then, create a new frame, whose parent is E1, and whose bindings reflect the results of the first step. Finally, evaluate the body with respect to that new environment. Another way to say this is that the let expression reduces to the body, in a new environment, after the subproblems of evaluating each of the values in the list of bindings.

Let is an example of how a new frame can be created. It also is a good example of the evaluation moving into another frame. However, these mechanisms will be illustrated in the next two sections about lambda expressions and combinations.

1.2.5 Lambda

The value of a lambda expression, as seen before, is a procedure. In the environment model, when a lambda is evaluated, a procedure object is created. This procedure

object has two pieces: text and environment. The text of the procedure can be read directly from the lambda expression; it consists of the parameter list and the body of the procedure. The environment part of the procedure is a pointer to the environment in which the lambda was evaluated. As we shall see in the next section, this environment pointer is very important when the procedure is going to be applied.

As a simple example, consider the expression `(lambda (x) (+ 2 x))`, with respect to GE. This creates a procedure object, whose text is `(x)` for parameters, and `(+ 2 x)` for the body, and whose environment is GE. The next section will describe how such a procedure object is used.

1.2.6 Application

The rule for procedure application is perhaps the most important of all the rules in the environment model, as well as one of the most confusing. There are four steps that should be followed to apply a procedure to the values of the arguments. First, a new frame is created. Second, the parent of this frame is set as a copy of the environment pointer of the procedure object being applied. Third, new bindings are created in the new frame between the names of the parameters of the procedure, and the values of the arguments. Finally, the body of the procedure is evaluated with respect to the new frame. Remember, in the substitution model, an application reduces to the body of the procedure, with the values substituted in for the parameters. Hopefully, that has been clarified by this rule.

There is another point about this rule that is worth mentioning. The step that causes the most confusion is step two, the parent of the new frame. This also happens to be the step which has the most consequences on the scoping rules of Scheme. Scheme is **statically scoped**, meaning that when a procedure is defined, the scope is captured in that procedure. Each time that procedure is run, it will be in that same scope. Some languages are **dynamically scoped**, which means that the scope in which a procedure is run is not determined until the moment that of the call. Therefore, different calls will take place in different scopes. Static scoping is accomplished in Scheme by remembering what frame the procedure is created in, and by linking to

that frame at runtime.

1.2.7 Example

This section will present a representative example of an environment model computation, and will show the completed environment diagram for the example. The expressions that will be evaluated are:

```
(define add-to-one
   (let ((x 1))
      (lambda (n) (+ x n))))

(define x 5)

(add-to-one x)
```

The first expression is a define special form, so a new binding will be created, in GE, between the name add-to-one and the result of evaluating the let special form. To find this value, we follow the rule for a let, which says to drop a new frame, with the current frame as its parent. In the figure, this is frame E1. In E1, the name x is bound to the result of evaluating `1` in GE, which is the number 1. With respect to E1, then, the body of the let is evaluated. It is a lambda special form, so a procedure object (double bubble) is created, that points at the current frame, whose text has the parameter `n` and body `(+ x n)`. Looking at the let expression, we see that this is the result we were looking for, so the name `add-to-one` in GE is bound to this procedure object.

The next expression is also a define special form, so in GE the name `x` is bound to the value 5.

Finally, the last expression is an application. Following the rule, we evaluate x in GE, and come up with 5. We evaluate add-to-one, and come up with a procedure. To apply this procedure to the value 5, we drop a frame E2, whose parent is E1, the same frame that the procedure points at. In E2, the name of the parameter n is

bound to the value 5. The body of add-to-one is then evaluated in E2. It also is a combination, `(+ x n)`. We look up each of the variables in the expression, and we find the procedure that does addition, in the GE, for +, we find the value 1, in E1, for x, and the value 5, in E2, for n. Using these results, we compute the result, 6. The completed diagram for this computation concludes this section.

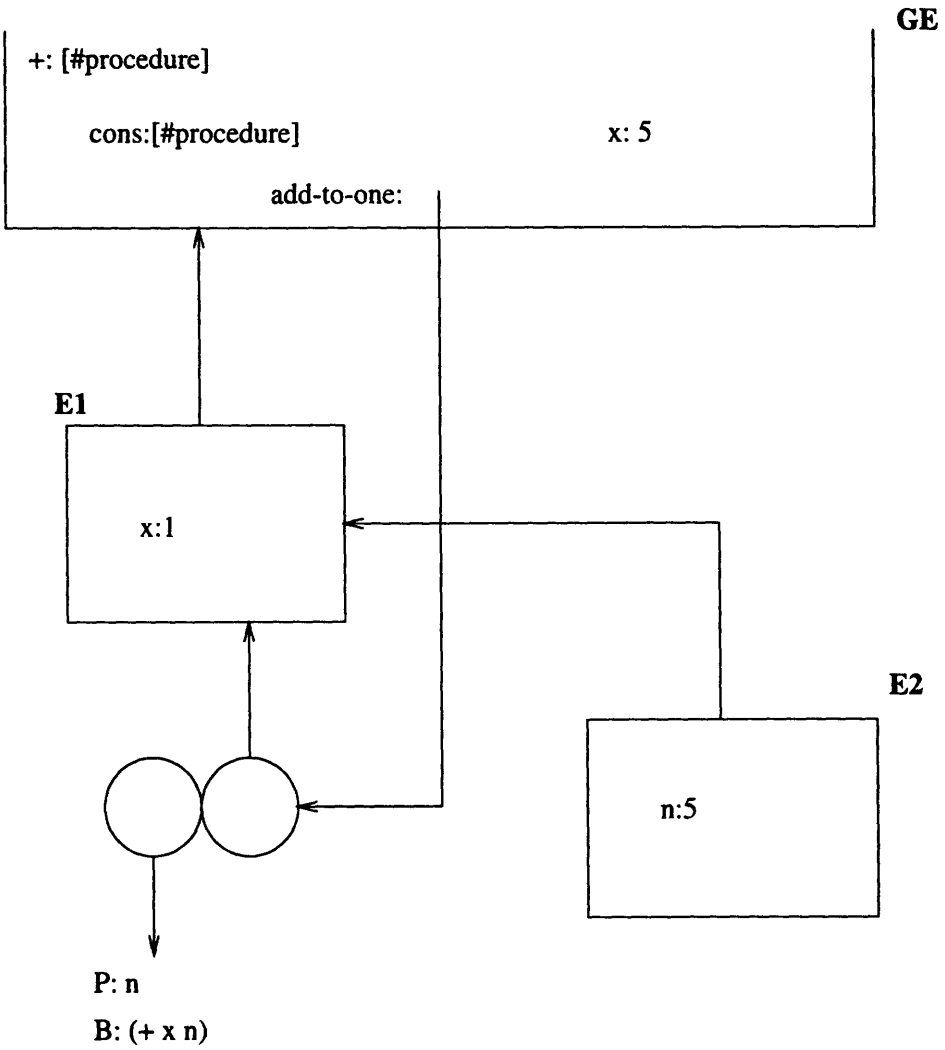

Figure 3: The environment after all three expressions have been evaluated.

1.3 Conclusion

We now have at our disposal two tools for understanding Scheme expressions. The substitution model is a tool for describing how to determine the value of Scheme expressions. It consists of a set of rules, one for each type of expression, that dictate how evaluation of such expressions proceed. By performing subproblems and reductions according to these rules, any Scheme expression may be reduced to its value. The environment model is a tool that helps to decide what the values of variables should be. It must be used in conjunction with the substitution model, and consists of another set of rules for building up and using an environment structure that represents variable bindings. Putting these two models together yields a powerful composite model for understanding the behavior of Scheme.

The purpose of TREval is to illustrate how expressions behave by showing the user some of the intermediate steps in the evaluation, as they might appear if he were performing the substitution model by hand. Each iteration of TREval corresponds to one step of a rule in the substitution and environment models. For instance, it might work on an if expression on one iteration, and on the next iteration it would be working on the predicate. A few iterations later, it may reduce the if into the consequent. Meanwhile, it keeps track of the stack of problems it has been working on, and the environment which is building up. By looking at all of this information, it can display to the user a Scheme expression that shows how far the evaluation has proceeded. This is how TREval brings these models to life.

Chapter 2

Building Scheme Evaluators

A Scheme evaluator is a program that takes Scheme expressions as input and produces Scheme values as output. Like any other program, an evaluator is built using an existing programming system. Often, when people build Scheme evaluators, they use Scheme as that foundation. The meta-circular evaluator (MCEval), for instance, is written in a full-fledged Scheme system.

The decision of what system to use as a foundation has many consequences. The Scheme evaluator built on top of a high level language, for example, may rely on the properties of that language to define the behavior of Scheme, and thus may hide such implementation details from the user. On the other hand, a low level Scheme evaluator may bog the user down in many details, but will explicitly show almost all of the details of Scheme behavior. These tradeoffs are important to think about when deciding what platform to use to write an evaluator.

This chapter will explore the systems on which three Scheme evaluators have been built. The first section will discuss MCEval, which implements Scheme in Scheme. The second will talk about an explicit control evaluator (ECEval), which implements Scheme in a pseudo-assembly language. Finally, the third section will present TREval, which is in Scheme, but chooses not to rely on all of the features of Scheme. The consequences of each decision will be discussed.

2.1 Foundation of MCEval

The meta-circular evaluator is built on top of Scheme. It relies on many of the features of Scheme to provide some of the behavior of the new language, which also happens to be Scheme. As a result, some behavior remains hidden from the user, unless he happens to be familiar with the properties of the underlying Scheme being used. The advantage of MCEval, however, is that it provides a very clear high-level overview of much of the important behavior of Scheme. MCEval is at the high-level end of the tradeoff that was discussed above.

For example, MCEval is very useful for clarifying the Scheme rules of evaluation. It is fairly straightforward to trace through the code of MCEval, especially for the evaluation of simple expressions, such as if expressions. Doing so will illustrate how the predicate is first evaluated, followed by either the consequent or the alternative. In fact, an excellent exercise is to add special forms to this evaluator, while thinking about and understanding the rules for that special form. This exercise will not bog anyone down in very many irrelevant details. This is one strength of MCEval; it allows the user to concentrate on certain aspects of Scheme, without forcing him to think about too many irrelevant details.

However, this strength does not come without a price. Since MCEval relies on underlying Scheme to provide some properties of the new Scheme, it does not shed any light on these properties. The recursive structure of MCEval is a good example of this. The two major procedures, MC-Apply and MC-Eval, are mutually recursive. Because of this, programs for MCEval will inherit certain behavior, that isn't specified in the code for MCEval. Any user of MCEval who does not already understand the mechanisms by which this recursion is working will not be able to decipher those mechanisms and how they work in MCEval programs.

One of these mechanisms is the means by which Scheme values are returned. In Scheme, each expression has a value. Since expressions can be built up from smaller expressions, the value of a given expression may either have to be returned to the user, or to be used in the computation of some other value. This depends on the

context of that expression. Scheme uses a **continuation stack** to keep track of what it has to do with a value when it is found. When a value is computed, the top entry of this stack is consulted, and Scheme follows what it says to do with that value. So, in evaluating the subexpression `(+ 2 3)` in the expression `(* 2 (+ 2 3) 5)`, the top of the continuation would be an entry that said to take the value and multiply it by 2 and 5. In this way, Scheme can keep track of what it was doing when it started working on the current expression, and how to proceed when it is done.

Since MCEval implicitly uses this mechanism, however, it does not show the user that it is happening. MCEval, with a recursive structure, uses the Scheme notion of deferred operations to keep track of what needs to be done, just as any recursive function (such as factorial) would do. These operations are stored on the continuation stack. When examining the code of MCEval, the only available conclusion is that the mechanism that MCEval programs use to return values is the same mechanism by which underlying Scheme returns them. Thus, no knowledge of this can be gleaned from MCEval. This is one example of the drawbacks of this kind of evaluator.

To summarize, MCEval is a Scheme evaluator built on Scheme. The advantage of having a high level basis for MCEval is that many concepts become clear, without examining too many details. The disadvantage is that many concepts are completely obscured, namely those that are inherited from the underlying Scheme. The next section will examine the opposite case, an evaluator built on a low level platform.

2.2 Foundation of ECEval

The explicit control evaluator is built on a low-level, pseudo-assembly language called register machine language. This language is designed as an extended assembly language for a virtual machine, and there is a Scheme program to interpret it. Many of the details of assembly have been eliminated, by allowing the user to declare the register set and the primitive operations. Additionally, registers can hold any data type, like lists or labels, and are not limited to numbers and pointers. Basically, the language is low level enough to see the inner workings of any program, without a lot

of the grunge of real assembly programming.

With such a basis, ECEval is forced to implement virtually everything that MCEval merely inherits from Scheme. Its advantage, therefore, is that these mechanisms are available for the user to see. The disadvantage, though, is that ECEval is not nearly as readable or intuitive to most people as MCEval.

As a comparison to MCEval, it is worthwhile to note the many details that the user needs to wade through in order to understand how if expressions work. For one thing, each time ECEval needs to evaluate a subexpression, it must go through a calling convention to recurse on the predicate. This means saving on the stack, setting the arguments and the return address into registers, etc. The user must also be aware of other conventions, such as the purpose of each of the registers, in order to make heads or tails of anything. To make matters worse, the language of ECEval is not as readable as Scheme. An action which might be simple in Scheme, such as comparing the first element of two lists, could be several lines of register machine language. In general, the higher level details, such as rules of evaluation of the special forms, are very obscure in ECEval.

On the other hand, the low level mechanisms of Scheme are very visible. ECEval manipulates the stack directly, so whenever it defers an operation, it does so explicitly. There is a register named "unev" that, along with the stack, serves as an explicit continuation stack. The Scheme return value mechanism is visible in ECEval because of this. There is code in ECEval that decides what to do with each value as it is computed. So, as opposed to MCEval, ECEval sheds light on this mechanism and how it works in Scheme.

ECEval is a good example of an evaluator built on a low level system. It provides a contrast to MCEval, which is built on a high level system. These evaluators have made tradeoffs involving which mechanisms of the new language are easily learned by a user of the evaluator. MCEval illustrates the high level details very well, but obscures the low level ones. ECEval is the opposite. The next section will examine TREval, which provides a compromise between the two extremes.

2.3 Foundation of TREval

The tail recursive evaluator, like MCEval, is written in Scheme. Like ECEval, though, it chooses to implement some of the low level details of the new language, Scheme. For example, since TREval has a tail recursive, or iterative, structure, it is forced to keep explicit track of the state of computation. Thus, it passes a continuation stack as a parameter of evaluation. Other features of Scheme that TREval does not rely on include higher order procedures, and style and order of evaluation of subexpressions. On the other hand, even though TREval is written in Scheme, it does require more effort to read than MCEval. So, as was the case in ECEval, some of the higher level details are harder to pick out from TREval than from MCEval. The rest of the section will discuss these decisions and how TREval fits in to this tradeoff.

TREval, like ECEval, allows the user to see the mechanisms that Scheme uses to return values. When TREval encounters a subproblem, it merely places a new continuation on the continuation stack for the next iteration. This entry is a reminder for the evaluator to come back to the current problem when it finds a value for this subproblem. Remember that MCEval merely **recursed** on the subexpression, and the problem of remembering where to come back to was dealt with by underlying Scheme. So, the mechanism, invisible to the user of MCEval, is available for the user of TREval to examine.

For example, suppose the user wants to determine how TREval remembers that it is working on the predicate of an if expression. He can look at the code for evaluation of an if, and see that the first thing that happens is a new iteration. This iteration has a new expression, namely the predicate of the if, and a new continuation on top of the stack. This continuation will tell TREval that, when it finishes evaluating the expression, it should check for false and then evaluate either the consequent or alternative. The user's initial question will be answered. Additionally, he can then extend his analysis, and see that when the whole if expression is done, the evaluator will check the top of the stack again, to determine what to do with that value. Eventually, it will find an empty continuation, when it will use the current value as

the result of the entire evaluation. So, with a little bit of extrapolation, the user can see the mechanism that TREval programs use to return values to the correct places.

There are several other features of Scheme that TREval does not depend on for its evaluation. TREval does not allow its programs to inherit the order of evaluation of underlying Scheme, instead it chooses an explicit order. Additionally, it does not allow the style of the underlying evaluator to affect its own style. For instance, if the underlying evaluator were using lazy evaluation, it would make no difference to the style of evaluation for programs of TREval. Additionally, TREval does not rely on Scheme's support for higher order functions. No procedures are passed as parameters inside of the evaluator. As a general description, it could be said that TREval is built upon a relatively small subset of the Scheme language.

The drawbacks of this feature of TREval are similar to those of ECEval, although they occur to a lesser degree. There are simply more details to wade through in order to understand some of the high level details of the program. For example, to understand how if works, the user needs to sift through the details of continuations. This seems sort of strange at first, that the same feature (the presence of continuations) can be seen as both a drawback and a strength. With a moment's thought, however, it makes more sense. The more details present, the more things the user can see, but the more he needs to get through to understand high level details. In any case, TREval is slightly more complex than MCEval in this respect, but not quite as complex as ECEval.

To summarize, TREval is built on a subset of Scheme. As such, it presents to the user many of the details of Scheme's implementation that MCEval leaves to the underlying evaluator. However, MCEval is much clearer to read than TREval in many ways. On the other hand, the Scheme code of TREval is more easily read than the assembly of ECEval, but ECEval leaves very little, if anything, to the underlying evaluator. TREval is a compromise between these two extremes.

2.4 Conclusion

The three Scheme evaluators, meta-circular, explicit control, and tail recursive, are built upon three different foundations. As a consequence of this, they each present Scheme's implementation in a different level of detail. MCEval, written in full-fledged Scheme, gives a very high level flavor of the language. TREval, written on a subset of Scheme, shows some of the low level details, but is also almost as easily understood as the high level MCEval. ECEval is written in assembly code, and gives the most low level picture of Scheme of the three evaluators. They all have their strengths and weaknesses, and all have their place in the learning environment.

Chapter 3

Structure of TREval

The tail recursive evaluator has a structure which is similar in many ways to other Scheme evaluators, such as the meta-circular evaluator. The major difference between these programs is that TREval is iterative where MCEval is recursive. This difference, however, is fundamental, leading to an entire class of structural and strategic differences between the two programs. This chapter will explore the structure of the iterative evaluator, explaining the overall evaluation strategy, data structures, and how some of the special forms are implemented.

Section one will discuss the structure and evaluation strategy of the evaluator. Section two will talk about the environment, and section three will discuss the continuation stack. Finally, section four will talk about how each expression type is implemented by TREval.

3.1 An Iterative Evaluator

The tail recursive evaluator is an iterative Scheme program intended to evaluate Scheme expressions. The main procedure of this evaluator is called TREval (brilliant!). It takes three parameters **exp**, **env**, and **cont**. They represent the expression to be evaluated, the environment in which to evaluate, and the continuation stack. On each iteration, TREval needs to compute the values of these parameters for the next iteration. This section will talk about how that is accomplished.

Any iterative program has a loop invariant, some set of statements about the state of the program that is true each time through the loop. The invariant for TREval consists of the following three statements: 1. **exp** is the expression to be evaluated. 2. **env** is the environment in which to evaluate **exp**. 3. **cont** represents what to do with the value of **exp** when it is computed. On the first iteration, for instance, **exp** is the entire expression, **env** is the global environment, and **cont** is the empty continuation (alternatively, the REPL continuation). TREval will finish evaluation when **exp** is some Scheme value, and **cont** is the empty continuation, at which time it returns **exp**.

The key issue is how TREval maintains this invariant while pushing the evaluation forward towards the final value. The idea is that on each iteration, TREval performs either a subproblem or a reduction. So, if TREval encounters an if expression, with **env** and **cont**, it realizes that the next iteration must be the subproblem of evaluating the predicate. So, the next value of **exp** is the predicate of the if. The next value of **env** is **env**, since the predicate should be evaluated in the current environment. Finally, TREval must remember what it is currently doing, since it must perform the reduction when it computes the value of the predicate. So, the next value of **cont** will be the continuation that says, "Hey, I was doing an if, the next step is the following..." Later, TREval will encounter the value of the predicate. When it does, it will consult **cont**, which will say what the next values should be for **exp**, **env**, and **cont**. This is a reduction. Notice how a subexpression adds something to the continuation stack, while a reduction takes something away.

To look a little more carefully, here is some of the code that implements if expressions. First, the expression is passed into TREval, and it drops into the following code:

```
(cond ((combination? exp) ...)
      .
      .
      .
     ((conditional? exp) (loop (ev-conditional exp env cont))))
```

TREval decides that `exp` is an if expression by testing the predicate `conditional?`, then loops (a fancy Scheme thing) with whatever the procedure `ev-conditional` will return. Let's take a look:

```
(define ev-conditional
  (lambda (exp env cont)
    (make-cep (conditional-predicate exp)
              env
              (make-cont
               'if
               (make-data
                (list
                 (conditional-consequent exp)
                 (conditional-alternative exp))
                '())
               env
               cont))))
```

The procedure make-cep is just a constructor, which allows the three values (the expression, environment, and continuation) that this procedure computes to be packaged into one value to return. The three parameters to `make-cep` are the `exp`, `env`, and `cont` for the next iteration. Notice, the next `exp` will be `(conditional-predicate exp)`, the predicate we must evaluate. So far, so good. The second parameter is `env`, which means the next environment is the same as the current environment. The final parameter is a new continuation, made using the constructor `make-cont`. The details of this will be discussed in a later section; for now, notice the information going into the continuation. The consequent and the alternative are both present, along with a tag, the symbol `if`, to remind TREval what kind of expression it was evaluating. Finally, the old environment and old continuation belong in the new continuation, for reasons that will be discussed later. The important things to realize, for now, are how these three parameters represent the subproblem of evaluating the predicate, and how all of the information we will need to come back to this if is present in the continuation. This function has computed values for `exp`, `env`, and `cont` for the next iteration of TREval.

Now, we can look at what happens when the predicate is finished, and TREval needs to return to the rest of the if. First, TREval detects it is done with the predicate when **exp** is a value, and the continuation is the same one that **ev-conditional** made using **make-cont**. When these conditions are met, TREval loops, using the value returned by another procedure, called **next-cep**, to fill the values of **exp**, **env**, and **cont** for the next iteration. Here are the relevant lines of **next-cep**:

```
(case (cont-tag cont)
   .
   .
   .
  ((if) (next/handle-if val cont)))
```

Next-cep looks at the tag of the contination (remember, the symbol **if**), to decide what type of continuation is on top. It applies the correct procedure for that continuation type. The procedure next/handle-if will return the next values of **exp**, **env**, and **cont**:

```
(define next/handle-if
  (lambda (val cont)
    (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)))))
```

Remember that **make-cep** is a constructor that lets these procedures return three values as one. Also, there are selectors here for continuations, they'll be explained in detail later, but for now: **(cont-next-cont cont)** gives the original continuation that TREval had before the evaluation of the if began, **(car (data-unev**

`(cont-data cont)))` gives the consequent of the if, `(cadr (data-unev (cont-data cont)))` gives the alternative, and `(cont-env cont)` gives the environment in which the whole expression started. Now, we can examine what the code actually does. First, it checks `val`, which is the value of the predicate. If `val` is true, it returns the consequent as the next `exp`, and the old environment and old continuation for `env` and `cont`. If it isn't, the code does everything the same, but uses the alternative as `exp` rather than the consequent. Again, don't worry about the details of the selectors and constructors, but try to see how this procedure is performing the reduction of the if into either the consequent or the alternative.

What we have just seen is an overview of how TREval performs evaluation of Scheme programs. Each iteration is either a reduction or a subproblem, whose task is to set the values of the three parameters for the next iteration. Remember, `exp` is the expression to be evaluated, `env` is the environment to evaluate in, and `cont` is what to do when a value is reached. Those three statements are true each time through the loop of TREval; they comprise the loop invariant. Finally, TREval is finished when `exp` is a value, and `cont` is the empty continuation.

It is time to deliver on promises made about constructors, selectors, and the data structures of TREval. The next section will explain the environment, and the following will explain the continuation.

3.2 The Environment

The environment is the mechanism by which TREval finds the values of variables. The data structure that TREval uses is the environment structure that is built in to Scheme. The constructors and selectors are defined in the code for the 6.001 Scheme evaluator, and are explained in the Scheme reference manual [2]. This section, therefore, will talk about how environments are manipulated inside of TREval.

The operations that TREval performs on environments correspond to rules of the environment model (see chapter 1 for a more thorough description of the environment model). For instance, let expressions create new environments, as do procedure

applications. Lambda expressions capture the current environment as part of the resulting procedure object. Variables are resolved by recursively traversing the chain of frames from the current frame to the global environment. All of the rules for the environment model are present in TREval.

There is one subtlety that takes place when TREval switches the current frame. Consider the following expression:

```
(+ ((lambda (x y) (+ x y)) 2 3) 1)
```

The evaluation of the second subexpression will cause TREval to change the current environment to a frame in which `x` and `y` are bound to 2 and 3, respectively. But, to properly evaluate the first and third subexpressions, they must be done with respect to GE, not that new frame. So, when TREval switches the environment, it places the current environment on the continuation, and passes the new environment as the next value for `env`. When the second subexpression finishes, it pulls the old environment out of the continuation and proceeds with respect to that frame. That is the purpose of the environment in the continuation; it remembers the current environment during subproblems so that TREval can come back to it, regardless of what environment the subexpression may complete in.

Another interesting point is how TREval creates new environments. To create a new environment, TREval first constructs that environment, using the environment constructor `*make-environment`, and specifying the names of the variables which will be bound in the new frame. Then, it creates bindings for those variables. In some cases, TREval knows the values for these bindings in advance, and can specify them at this time. In other cases, though, such as in the middle of a letrec statement, not all of the values are known. In these cases, the variables are left unspecified in the new frame, until the values are known. At that time, the variable is assigned to its value. Any reference to an unassigned value is an error. This strange behavior of TREval is a tribute to the environment constructors which are available, and to the awkward (but useful) semantics of internal define statements and letrecs.

Hopefully, this section has shed a little bit of light on one of the major data structures used by TREval: the environment. The main purpose of the environment

is to provide a mechanism by which the evaluator can look up values for variables. Basically, TREval uses the environment to directly implement the rules given by the environment model. One interesting manipulation that TREval performs is placing the environment on the continuation, preserving the current frame to be restored on the next reduction. Another noteworthy point is the method that TREval actually uses to build new environments. All in all, however, there is little difference between environments in TREval and environments in most other evaluators.

3.3 The Continuation

The continuation is the data structure that reminds TREval of what to do with the value of the current expression when it is computed. It allows TREval to have an iterative structure, by representing the work that remains to be done in the computation. (Every iterative procedure keeps track of what has been done and what has yet to be done.) This section will discuss both the structure of the continuation and how it is used inside of TREval.

The continuation has several pieces. First, every continuation has a tag, which represents the type of expression that spawned the current subproblem. For instance, during the evaluation of the predicate of an if expression, the continuation would be tagged by the symbol `if`. The continuation also has a field called the data, which holds the parts of the outer expression. This data field has two parts, one for evaluated expressions (values), and one for unevaluated expressions (future subproblems or reductions.) The if continuation holds both the consequent and the alternative in the unevaluated part of the data of the continuation. The last two pieces of the continuation are the environment, which was discussed above, and the previous continuation. The previous continuation can be thought of as the reminder of what to do when TREval is finished with the expression that caused the current subproblem. That is a mouthful, but, as always, if is a good example. Assume that TREval is evaluating the predicate of an if expression, with an if continuation. When the predicate is finished, TREval remembers that the next step is to evaluate the consequent

or alternative, in the old environment. But, what should the new continuation be? That is why the if continuation has a continuation of its own. When TREval finishes the predicate, it will take the continuation from `cont` to remind it that when it finishes the if, it must return to the problem of which if was a subproblem. So, one way to view the `(cont-next-cont cont)` is the continuation that was around when the current problem was started.

Another way to view the continuation is as a stack. The top entry is `cont`, the next is `(cont-next-cont cont)`, and so forth. When TREval encounters a subproblem, it pushes an entry on the stack, by creating a new continuation, which contains the old continuation in its next-cont field. When it encounters a value, it pops the stack, by restoring the parameter `cont` to be the next-cont field of the current continuation. This is the view that is taken when `cont` is referred to as the "continuation stack."

Either way you view the continuation, it can be treated as a simple data structure, with its own contract, just like a pair or a list. The constructor is `make-cont`, and the selectors are `cont-type`, `cont-data`, `cont-env`, and `cont-next-cont`. Additionally, the data field has the constructor `make-data`, along with constructors `data-unev` and `data-ev`. The following section will discuss in more detail the types of continuations and how they are used within TREval.

3.3.1 Use of the Continuation

The continuation keeps track of the expressions that have generated subproblems that are in the process of being evaluated. Each time a subproblem is encountered, a new continuation is generated, containing the old continuation in its `cont` field, the current environment as its `env`, the expression type, and the relevant information to reconstruct the current expression when the value of the new subproblem is found. When that occurs, TREval consults the continuation, restores that information and reconstructs the problem as it was just before the subproblem was started.

Continuations generally will pile up into a stack structure during an evaluation. This happens when a subproblem of one expression generates another subproblem, and it reflects the recursive nature of Scheme. For example, consider the evaluation

of the expression `(if (* 2 (+ 3 4)) x y)`. The evaluator, at some point, will be evaluating the expression 3. The continuation at that point will be of type `comb`, for combination, and will indicate that the value of 3 should be added to 4. Below, on the continuation stack, will be another continuation of type `comb`, indicating that some result should be multiplied by 2. Another level down, the continuation is of type `if`, reminding TREval to use the result to decide whether to evaluate `x` or `y`. Finally, the lowest continuation will be the empty continuation, saying that the result should be returned to the user. Notice, however, that if this expression were part of a larger expression, the "bottom" continuation would tell TREval to use the value that it has as part of the larger expression.

The previous two sections have explained something about the structure and dynamics of the two major data structures of TREval: the environment and the continuation. Both are used to keep track of information vital to the evaluation at hand. We now have all of the tools to talk about how Scheme expressions are evaluated in TREval, in fine detail.

3.4 Evaluation of Expression Types

This section will discuss the strategy of evaluation for each type of Scheme expression. It will draw together the ideas from the earlier parts of this chapter. Each subsection will address a different kind of expression, and will explain how the continuation, environment, and expression parameters are manipulated, from the point where the expression type is first encountered, until its value is computed. The relevant code, to follow along, is located in `TREval.scm` and `next-cep.scm`, found in appendix A of this paper.

3.4.1 Combinations

`(<operator> [<operand> ...])`

Upon encountering a combination, TREval will evaluate the subexpressions in reverse order, then apply the value of the operator to the values of the operands. The

continuation during this process will be of type combination, denoted using the symbol `comb` as the tag. The data of the continuation are the subexpressions that are not currently being worked on, whether they have already been finished or not.

TREval uses the continuation as a map to evaluate the combination. At first, a list is created, of all the subexpressions but the last, in reverse order. This is packed into the unevaluated data for the next continuation. The evaluated data is set to the empty list. The `env` and `cont` fields are filled with the current environment and current continuation, and the tag is the symbol `comb`. Since TREval will be evaluating the last subexpression, that is what the next value of `exp` is. Finally, `env` does not change from this iteration to the next.

When a subexpression is evaluated, and a value is reached, TREval will find the `comb` continuation on the stack. At this point, several things happen. The current value is consed to the evaluated list, the car of the unevaluated list becomes the next expression to be evaluated, the cdr of that list becomes the new unevaluated list, and the `env` of the continuation is copied to both the `env` parameter and the next continuation. The next continuation is made up of the updated data lists, along with `env` and `cont` from the old continuation, and the symbol `comb`.

When there are no subexpressions left to be evaluated, (ie, unevaluated data turns out to be the empty list), the result of the last subexpression (the operator) is then applied to the results of the operands, whose values are in the proper order in the evaluated data list in the continuation. If the operator is a primitive procedure, the next `exp` is simply the result of using Scheme's `apply` on it and the argument list, and the next value for `env` is pulled from the current continuation. Otherwise, the next `exp` is the body of the procedure object `val`. The environment will be a frame, whose parent is the frame pointed to by `val`, in which the formal parameters from `val` are bound to the corresponding values in the evaluated data list. This is one of those times when TREval creates a new environment, and knows right away what the values for each of the variables should be. In either the primitive or compound case, the next continuation is the one that is found in the original `comb` continuation, since the result of the combination will be the either value of the body evaluated in

the new environment, or the **exp** parameter itself. This value needs to be given to that original continuation, which is expecting the value of the combination.

Keep in mind that during the evaluation of each of the subexpressions, the comb continuation may very well get buried on the stack. The key idea, though, is that when evaluation of any subexpression has finished, the **comb** continuation will be the one on top of the stack. This is an important point to remember, that any evaluation completes with the same continuation stack that it starts with. That property will allow us to "wish away" the evaluation of subexpressions. By doing so, we can predict the future state of TREval, without really worrying about how it actually manages to get there. Doing so is an important step in truly understanding how these special forms are working.

3.4.2 If Expressions

`(if <predicate> <consequent> <alternative>)`

The if expression generates one subproblem during its evaluation. To evaluate an if, TREval first will evaluate the predicate, then decide to evaluate either the consequent or the alternative.

When **exp** is an if expression, TREval prepares for the next iteration as follows. The next value of **exp** is the predicate of the if. The next **env** is just **env**. The next continuation is tagged with the symbol **if**, and also contains **env** and the current continuation. In the data unevaluated list, the consequent and alternative expressions are placed. When the predicate has been evaluated, TREval pops the if continuation from the stack, and resets **env** and **cont** from it. The next **exp** is the alternative, if the value of the predicate is false, otherwise it is the consequent.

3.4.3 Cond Expressions

`(cond [(<clause> <branch>) ...])`

The evaluator handles cond expressions in two steps. First, it evaluates clauses to decide which of the branches to take. Then, it reduces the expression to that branch.

This evaluation is accomplished using a continuation of type **cond**. When TREval comes across a cond expression, the next expression to be evaluated becomes the first clause. The environment remains the same. The continuation is built from the symbol **cond**, the environment, and the current continuation. (The last two pieces are common to all continuations. Whenever a continuation is built, these pieces are used.) The evaluated data is set to the empty list, and the unevaluated is constructed from the branch corresponding to the current clause and a list of the rest of the clauses.

When the value of a cond branch is computed, and a **cond** continuation is recognized, the following actions take place. The next value of **env** is found in the continuation. To find the next value of **exp**, TREval checks the value that was just computed. If it is not false, then the next expression is the branch of the cond that corresponds to the current clause. If the value is false, then the next **exp** is the next predicate. The two possibilities for **exp** are both located in the current continuation, in its unevaluated data field. If a branch is not taken, the continuation is reset to **cond**, using the branch of the next clause and the list of remaining clauses as the unevaluated data, and keeping the evaluated data empty. On the other hand, if the expression is to be reduced to a branch, the continuation is restored from the **cond** entry to the entry below it on the stack.

If, at any point during this process, the next clause is detected to be the symbol **else**, TREval acts as if a value of true was computed, and the cond expression is reduced to the branch of the else. If TREval runs out of clauses, then the next expression is something called no-value, the continuation is popped, and the environment is reset from the cond continuation.

3.4.4 Case Expressions

`(case <key> [(<tag> <exp>) ...])`

The subproblem that must be finished during the evaluation of a case statement is the evaluation of the key. Therefore, the case continuation type indicates that TREval has finished evaluation of the key, and is ready to reduce the expression into one of the branches.

When TREval sees a case statement, it uses the key as the next value of **exp**. The enviroment remains constant, and the continuation (as stated above) will be of type **case**. Into the unevaluated data will go a list of tag-expression pairs. If the value of the key is present in a given tag, the case statement should reduce to the corresponding expression. These pairs are taken from the case statement itself. The evaluated data is kept empty in this continuation.

Upon finding the value for the key, and recognizing a case continuation, TREval must traverse the list of tag-expression pairs present in the continuation, to find either the symbol **else**, or a match (in the sense of eqv?) between the value of the key and one of the elements of the tags. The case expression reduces to the expression that corresponds to the first tag that satisfies one of those two conditions. In the case where no tag does, no-value is the next expression. Regardless, the continuation is popped, and the environment is reset from the case continuation, as is normal during a reduction.

3.4.5 And/Or Expressions and Sequences

```
(and [<exp> ...])
(or [<exp> ...])
(begin <exp> ...)
```

The special forms or, and, and begin are handled almost identically by TREval. In all three cases, the evaluator traverses the subexpressions, evaluating them in order, as subproblems, until an exit condition is met, or until no subexpressions remain. At this time, the whole expression reduces to the value of the last subproblem. In case of and, the exit condition is a subexpression that evaluates to false. For or, it is a subexpression that doesn't evaluate to false. In a sequence, there is no exit condition. Again, the reduction of these expressions is to the value of the last evaluated subexpression.

To avoid redundancy, this paragraph will discuss the mechanics of and only, and will leave or and begin to the reader as an exercise. When TREval finds an and special form, it takes the first subexpression as the next value for **exp**. The **env**

is held constant. The continuation is set to be of type **and**; it will contain the list of unevaluated subexpressions, in order, as its unevaluated data, and an empty list as evaluated data. When the value of the first subexpression is computed, and the continuation is recognized on top of the stack, TREval looks to see if either the value is false, or if there are no further subexpressions. In either case, the and reduces to the value which was just computed. The continuation is popped from the stack. The environment is taken from the **and** continuation. Otherwise, the new expression is the first element of the list of subexpressions stored in the continuation, and that list is set to its own cdr. In this manner, TREval traverses the list of subexpressions, evaluating them one at a time, until the exit condition is met.

3.4.6 Lambda Expressions

`(lambda <parameters> <body>)`

Lambda expressions are handled by passing the whole expression to the underlying evaluator. This was a design decision, due to the many failed attempts of the authors to find the constructor to make a real Scheme procedure object in the 6.001 system. The next values of the parameters when TREval comes across a lambda expression are very straightforward: next for **exp** is the result of doing **(eval exp env)**, and the environment and continuation are kept the same.

3.4.7 Let, Let*, Letrec Expressions

`(let ([(<name> <exp>) ...]) <body>)`

The let family of special forms allow the user to evaluate an expression in an environment which contains some specified bindings. They all have two pieces: a binding list, which is a list of name-expression pairs, and a body, which is a sequence of scheme expressions. They differ only in the environment in which the expressions of the binding list are evaluated. The strategy that TREval uses to evaluate these expression is to first evaluate each of the expressions in the binding list, in the appropriate environment, then to reduce the expression to its body, in the newly constructed

environment.

For all of these forms, TREval deals identically with the **exp** parameter. The first iteration after the let is encountered, **exp** will be the first expression from the binding list. When that value is computed, **exp** will be set to the next expression of this list, and so forth, until all of the expressions in the binding list have been evaluated. At this time, **exp** will become the body, during the final reduction of the expression.

However, the three forms differ in the way that they handle continuations and environments. Let, for instance, evaluates all of the binding list expressions in the current environment. In the continuation, the data which is unevaluated contains the rest of the bindings of the list, and the body of the expression. The evaluated data holds the name of the variable that will be bound to the current expression, as well as the name-value pairs that have already been computed. When all of the bindings have been evaluated, a new frame is created, containing all of the bindings that have just been computed, whose parent is the environment in the `let` continuation. That new frame is the next value for **env**. The new continuation is the one which is right underneath the current one in the stack. This implements the rule for let, which says that the expressions in the binding list should be evaluated in the current environment, then the new environment should be created, and the body evaluated.

Let* behaves somewhat differently. The rule for let* is that the expressions of the binding list are evaluated sequentially, in an environment in which all of the previous bindings are in effect, but none of the subsequent ones. Therefore, the environment in which the first expression in the binding list is evaluated is the current environment. The continuation has the bindings which have yet to be evaluated in the data unevaluated field, and the name of the current binding in the evaluated data field. When the value is computed, a new environment is made, with the last name-value binding present, and the next expression is evaluated in this new environment. Finally, when no expressions remain, the body is evaluated in a new frame, connected to the original frame through a chain of intermediate frames, in which the last binding is present. Each of the intermediate frames also has one variable bound inside. The continuation type `let*` throws away all of the bindings that are already computed,

since they are present in the environment. So, at any point, the evaluated data of a let* continuation is just the name of the current variable being evaluated.

Letrec has its own rule, different still from the others. The expressions from the binding list are evaluated in an environment in which all of the bindings from the list (past, present, and future) are visible. However, these bindings have undefined values, until all of the expressions of the list have completed. Therefore, no expression is allowed to reference these new bindings, until all of the expressions have been completed. TREval deals with this rule by creating a new frame, in which all of the names in the binding list are bound to bad values. These bad values will cause TREval to halt if, at any time, a name bound to them is looked up. This frame becomes the **env**, for the evaluation of the binding list. **Exp** becomes the first expression of the binding list. The continuation holds the body and the rest of the expression list in unevaluated data, and the current variable name, and previous name-value pairs, in the evaluated data. When there are no expressions left in the binding list, TREval simultaneously assigns each variable to hold its correct value. It then pops the continuation, and evaluates the body, using the newly completed environment and the old continuation.

To summarize, the let family of special forms all proceed by evaluating the expressions in the binding list to form a new environment, then evaluating the body in this new environment. They differ in the exact way that they accomplish this. Let evaluates each binding list expression in the old environment, let* evaluates them in sequence, and letrec evaluates them all in a new, partially completed environment.

3.4.8 Definitions and Assignments

`(define <name> <exp>)`

`(set! <name> <exp>)`

The special forms define and set! are very similar in structure and behavior. In fact, except for the very last step, they are handled in an identical fashion by TREval. First, the expression is evaluated. Then, the name is bound to the value of that expression. In the case of define, a new binding is created in the current frame,

between the name and value. For set!, the current binding for the variable with respect to the current frame is altered. This crucial difference gives these forms very different behavior. However, for the purposes of this evaluator, these expressions are lumped together, since their evaluations are so similar.

When TREval sees these special forms, the next value for **exp** is set to the expression. The environment is kept, and the continuation of type **define** or **assign** is created, with the name of the variable stored in the data. When this evaluation completes, the variable name, environment, and old continuation are retrieved from the current continuation. Next for **exp** is an unspecified value (in the case of define it will be no-value). The expression is reduced to this meaningless value. Before the next iteration, however, TREval uses either `local-assignment` (for define) or `environment-assign!` (for set!) to create or alter the binding of the name with respect to the current frame. These environment modifiers correspond to creating a new binding in the current frame (`local-assignment`), and traversing the environment to find the current binding of the name, in order to change it (`environment-assign!`). In TREval, that is the only difference between the two special forms.

3.4.9 Additional Features and Limitations of TREval

There are some features of Scheme that are implemented in TREval that this chapter has not discussed. These are mostly obscure special forms, or obscure features of well-known special forms, whose discussion would serve mainly to confuse readers. Additionally, there are some features of the Scheme language that are not implemented by TREval for various reasons. This section will attempt to cover some of these features and limitations, very briefly, so that the reader can have a taste of them. For further discussion of these features in TREval, see the thesis of my partner, Abraham Maritime [6].

The following is a brief survey of some of the features of TREval that were not discussed in this chapter. To jump right in, the branch of a cond expression may be a receiver for the value of its corresponding clause, meaning that if the clause is non-false, the receiver will be applied to that value. And and or statements may

have zero subexpressions, in which case the answer is true or false, respectively. Let statements support the construct known as named lets, which is another means of writing an iterative procedure in Scheme. TREval itself is written as a named let. Set! statements may assign variables in different environments, by means of a special construct called an access. Finally, there are four other special forms implemented: quasiquote, the-environment, access, and call-cc.

Some of the special forms of TREval are not completely up to the specification in the Scheme manual [2]. Quasiquote, for instance, has a very complex behavior, and it was deemed not worth the effort to write a complete version. Access yields some problems during the printing phase of this evaluator (which hasn't been discussed yet.) Call-cc is not handled correctly, in the sense that the user will see expressions printed that would not evaluate to the correct value during the course of evaluation. Additionally, named lets interact with the printing code in a strange way, the user is exposed to a "hack" that the author of this paper used to get them working. Each of these problems was considered to be a low priority issue, and may be fixed by the time anyone reads this paper.

3.5 Conclusion

This chapter has hopefully explained a lot about the inner workings of the tail recursive evaluator. It spoke about the iterative nature of the evaluator, and explained the general strategy behind TREval. It discussed the use and form of the two major data structures, the environment and the continuation. Finally, it described how TREval approaches the different kinds of expressions, in an attempt to pull all of this information together. Hopefully, you can now understand most of the code in `TREval.scm` and `next-cep.scm`, at least on a very coarse level. The rest of the thesis will discuss how TREval is used to emulate the substitution model.

Chapter 4

Tail Recursion, Explicit State, and the Models of Computation

This chapter will discuss how TREval can be used to emulate the substitution model. On the surface, TREval is a machine that directly implements the environment model. However, by observing that the environment is merely an aid to the substitution model, TREval can be viewed as a substitution model machine. Notice, then, how iterations of TREval correspond to steps in the substitution model. Finally, see that TREval maintains enough information to print its state as a Scheme expression, which is equivalent to the original expression, and corresponds to a point in the evaluation according to the substitution model.This is how TREval is used to perform examples of the substitution model.

The first impression of TREval is that it is a direct implementation of the environment model. Indeed, it manipulates the environment according to the rules of this model. For example, let special forms all create new environments in which to evaluate their body, lambda expressions capture the current environment in the resulting procedure object, and procedure applications occur in the appropriate environment as well. Taking this view, however, it may be fairly difficult to see the uses of TREval. After all, MCEval, ECEval, and other evaluators can all evaluate Scheme expressions, and all have great educational value. As we will see, however, because of its iterative nature, TREval has an important advantage over these evaluators.

The first step towards realizing a substitution model evaluator is to treat the environment model as merely a convenient extension of the substitution model. The purpose of this convenience is to help the evaluator to look up variables, and to decide which instance of a name is meant by an expression, when there are two separate bindings that are named the same. With that observation, it can be stated that TREval is a machine that follows the rules of the substitution model, except that it uses the environment, rather than scoping or renaming rules, to keep track of variables.

In particular, each iteration of TREval corresponds to a step of the substitution model. You may have noticed the emphasis placed on subproblems and reductions in the previous chapters of this paper. These concepts are important for making a correlation between TREval iterations and substitution model steps. Each step of the substitution model is either a reduction or a subproblem. Each iteration of TREval is also either a subproblem or a reduction. When TREval performs a subproblem, it pushes onto the continuation stack. When it performs a reduction, it pops from the stack. Sometimes, the stack depth is kept constant from one iteration to the next. This happens when a single problem has more than one subproblem, and TREval completes one and moves to the next, popping one entry and pushing another. To use my favorite example, the rule for evaluating an if expression in the substitution model is to evaluate the predicate as a subproblem, then use either the consequent or the alternative as a reduction. In TREval, the first iteration after an if has the predicate as the next expression, with a new continuation on top of the stack to signify a subproblem. When the subproblem is complete, the if continuation is popped, signifying a reduction, and the next expression is either the consequent or the alternative. The parallel is strong.

Finally, we are ready for the key observation of this chapter. Since TREval is iterative, it maintains explicit state at each iteration. This state, therefore, can be observed by code external to the evaluator itself. Additionally, any code that does so can derive a Scheme expression which embodies the current place in the evaluation. By looking at the expression being evaluated, the environment, and the continuation,

the code may print a Scheme expression which has the following properties: it is simpler than the original expression, it evaluates to the same value as the original expression, it embodies the current state of completion of the evaluation, and thus it represents a step along the chain of substitutions that would be made if the evaluation were proceeding according to the substitution model. We can look, for example, at the expressiono `(> 3 4)`, the global environment, and the continuation that says `(if [value here] x y`, and we can write down the corresponding expression `(if (> 3 4) x y)`. The processes that accomplish that mapping are implemented in `print-cep.scm` and `bindings.scm`, and they are described in detail in the next chapter.

This chapter has hopefully helped you make an intuitive leap with your thoughts about TREval. By now, you might be thinking about how to look at the parameters of an iteration of TREval and write them down in another form, as a Scheme expression. Chapter five will explain how the printing code that accompanies TREval accomplishes that task.

Chapter 5

Printing the State of TREval

We have already seen how TREval can be used to emulate the substitution model. The only remaining detail is how the state of TREval is mapped into a Scheme expression. The problem can be broken into three pieces: how the environment, continuation, and current expression can be represented as Scheme expressions. Additionally, these three pieces must be put together to form the solution to the entire puzzle.

The strategy to print an environment is to map the structure into a list of bindings. These bindings can then be placed inside of a letrec statement, the body of which will be the expression that we need to evaluate. This seems like a simple strategy. There is, however, a complication. Since some information of the environment is lost in this transformation (namely, the frame structure), there must be a way to add additional information to the binding list to compensate. The issue becomes what to do when the same name is bound in different frames. To maintain the information which is present in the environment, these variables are suffixed, according to their frame, so that they remain distinguishable when an expression refers to one of them. For example, if there were two different occurances of the name x in the environment, we might leave one of them as x and suffix the other to be x-1. By doing so, we maintain the information of frame structure that would otherwise be lost.

All expressions that refer to x, therefore, also might need to be changed. In other words, an expression `(+ x 1)` may either remain unchanged, or be suffixed to be `(+ x-1 1)`, depending upon which x is being referred to. Therefore, when expressions

are being printed, they must be printed with respect to their environment, so that the code may rename the variables that occur in that expression properly. Additionally, there are some variables that should not be renamed. For example, consider the expression `(lambda (x) (+ x x))`. The variable x should not be renamed in the body of this expression, since doing so would change the meaning of the expression. So, during the printing of an expression, the code maintains a list of variables that are to be left alone.

Except for this renaming issue, printing the expression of TREval is quite straightforward. Perhaps you have noticed that expressions in TREval are kept internally in a different form than the list structure that is valid in Scheme. To print these internal expressions, they merely need to be converted back to that list structure. This is mostly an uninteresting transformation, and will largely be ignored for the rest of this chapter.

To print a continuation, the code starts with the top continuation on the stack, and the expression being evaluated, and prints that much as an S-expression. Then, it looks at the next continuation, using what it just computed as the new "expression being evaluated". Traversing down the continuation stack in this manner, it will build the expression from the inside out. Once it reaches the empty continuation, it is done, and the resulting expression is used as the body of the letrec whose bindings represent the environment.

To summarize, the state of TREval will be printed in the following form:

`(letrec (<environment>) <cont and exp>)`

The next section will detail how the code prints the environment, and the following section will discuss the printing of the continuation.

5.1 How to Print an Environment

This section will explain how the TREval environment is printed. The relevant code for the rest of this chapter can be found in `bindings.scm` and `print-cep.scm`. For this section, the environment refers to all frames that are reachable through either

the environments in the continuation, the current frame, or the values of the variables in any reachable frame. The strategy to print an environment is as follows: first, find all of the reachable bindings and compile them into a list. While that goes on, keep track of name conflicts and assign unique names to each variable. Once this binding list has been compiled, map each of the values of the list into a Scheme expression, using code that prints the expression data structure, and then include the result in the final letrec statement as the binding list.

5.1.1 Traversing the Frames

In order to compile a list of all of the reachable bindings, the printing code (print-cep) must make a traversal of the environment structure. In fact, the traversal as a whole is made up of several mini-traversals, each starting at some reachable frame and finishing at the global environment. Each mini-traversal, therefore, covers the chain of frames that lead from the start to GE. The mini-traversals are done starting from each environment in the continuation, the current environment, and each environment of a compound procedure that is encountered during the course of another traversal.

The first step in this process is to traverse the environments found in the continuation stack. Print-cep looks at the first entry in the continuation stack, gathers the binding list for its environment, then recurses on the rest of the stack. When it reaches the empty continuation, it is finished with this part of the traversal. After each step print-cep appends the result of the current step to the list created by all of the prior steps. Finally, print-cep appends the final list to the binding list generated by traversing the current **env** parameter.

Notice how I haven't mentioned the compound procedures yet. They are handled as they are encountered by print-cep. When a binding has a procedure as its value, print-cep interrupts the current traversal, while it works on the environment of the newfound procedure. In a sense, this is a subproblem of the original mini-traversal. When the procedure's environment has been completed, the current traversal resumes, and the two results are appended together and returned as the result of the entire mini-traversal.

To summarize, print-cep performs the mini-traversals that originate from the environments of the continuation. It then traverses the current environment. At any time, if it encounters a procedure, it stops, traverses the procedure's environment, and resumes. All of the resulting lists are appended to form the final result of this process.

5.1.2 Printing the Bindings of a Frame

In order to reach its goal of computing a binding list that represents the environment, print-cep must take certain actions at each frame it encounters during its traversals. The procedure that performs these actios is called `filter-and-suffix`, and it can be found in the file `bindings.scm`. Filter-and-suffix takes the list of bindings of a frame, and it returns a list of new bindings which are suitable for inclusion in the letrec expression. It does so by recursively traversing the input list, taking each binding and putting it into a new form. In the process, it performs several tests on each binding, and uses the results it decides how to include that binding in the output list.

Filter-and-suffix performs several tests on each binding in its input list. If a binding fails a test, the code moves on to the next binding. The failed binding is not included in the output, and no further computation is performed on it. The steps taken by filter-and-suffix are the following. First, it checks the binding for a value, since some names can be unassigned, especially in a new frame. Then, it computes a suffix for the name. The suffix has some information inside of it, which will tell filter-and-suffix whether this name has already been suffixed in this frame. If that is the case, it implies that some other traversal has already included this binding in the list, so the code leaves it out this time around. Next, filter-and-suffix looks at the value of the binding. If the value is bad-value, the binding is not included. If the binding passes all of these tests, then it will be included in the output list. The suffix is attached to the name to form a new name, and that new name, the value, and the environment are packaged up and added to the output list. Finally, if the value is a compound procedure, filter-and-suffix initiates a mini-traversal of the environment of that procedure, and includes the results in the output list.

To summarize, a variable must pass several tests in order to be included in an output list by filter-and-suffix. It must not already be in the output from a previous traversal. It must be bound to something and that something must not be the bad-value. If these are all true, the binding becomes part of the binding list. Additionally, if the value is a procedure, a new traversal is done, recursively, and the results are included in the binding list. The end result of all of this, is a linear list which contains each reachable binding in the environment exactly once.

5.1.3 Suffixing Variables

The task of renaming variables and the task of keeping track of what variables have been included in the binding list are both handled by the suffixer. There are two different ways that print-cep can suffix variables, this section will discuss minimal numbering, and the unique frame numbering will be left as an exercise.

The suffixer is given as input a variable name and a frame, and its task is to compute a suffix data structure for that variable instance. The suffix data structure has two parts: a value and a tag. The tag will either be the symbol `new` or the symbol `old`. The value will be a number that will be associated with this variable in this frame from now on.

The first time a variable name is suffixed, regardless of which frame, it will be given the number zero. The next time it is suffixed, in a different frame, it will be given one, and so forth. The second time a variable is suffixed in the same frame, the number given will be the same that was given the first time. Additionally, the tag of that suffix will be `old`. To summarize, the value of a suffix is how many frames this name has been suffixed in previously, and the tag is whether the name has been suffixed in this frame in the past.

The procedure make-number-for-variable creates a suffixing procedure, which print-cep binds by the name suffixer. The procedure suffixer has local state which consists of two hash tables. The first table is keyed on variable name, and the second is keyed on both variable name and frame. The first table holds the last number that was given out for each name, and the second table holds the number that was given for

each name-frame combination. When suffixer sees a name-frame pair, it first checks the variable table for the last number given out to that name. If it turns out that suffixer has never seen the name, the suffix returned is a new zero, and entries in both hash tables are created. These entries both have zero as their data, since that is the suffix being handed out. If, on the other hand, there is an entry in the variable table, the data present is called the new number, and it represents the last new suffix that was given to this name. Suffixer then consults the name-frame table to see if this is a fresh combination. If not, then the number found by this lookup is the suffix that was given the last time this combination was encountered. In this case, an old suffix with that number is returned, and no update of the state is required. Otherwise, suffixer returns a new suffix, with the new number plus one as the value. It changes the entry in the variable table to reflect this new number, and it adds an entry to the name-frame table to remember the number in case these same parameters are suffixed again.

In short, the suffixing procedure uses hash tables to keep track of the parameters that they have seen in the past. When called, it consults these hash tables to determine if it has seen the variable yet, and if it has seen the variable in combination with the frame yet. With that information at hand, suffixer can decide what the proper suffix should be.

5.1.4 Conclusion

To summarize, a rather complex series of steps is necessary to print an environment as a list of bindings. A traversal of the structure is made, from all of the frames that are directly reachable from the continuation, and from the current frame, to the global environment. At each frame, a list of bindings is created, consisting of newly encountered, renamed variables and their internal values. The values are then converted into valid Scheme syntax, and the whole list is then included in the binding list of a letrec statement.

5.2 How to Print a Continuation

The task of printing a continuation is slightly less complex than that of printing the environment. The job at hand is to take a continuation, with an "inner expression", and generate an expression representing the entire computation. This will yield a subproblem of the next continuation, and the process can be repeated. By traversing the stack of continuations in this manner, the entire original expression can be reconstructed.

The basic idea about how to transform a continuation into a Scheme expression follows from an observation about the structure of continuations. That observation is that a continuation represents a Scheme expression with a hole somewhere in the middle. For example, a continuation of type if, generated from the expression

```
(if (> 2 3) (+ 1 2) (- 2 1))
```

represents an expression that looks like

```
(if [] (+ 1 2) (- 2 1))
```

where `[]` represents a hole. The hole is the place in the old expression into which the value of the current expression should be inserted. To print this continuation, print-cep first recognizes that it is of type if. Therefore, the result will be an if expression. It then sees that the predicate and alternative are in the data unevaluated field. It constructs an if expression, whose consequent and alternative are found in the continuation. The predicate of the result is the current **exp**, or the inner expression. That is all there is to it.

To give another example, consider a comb continuation

```
(+ 3 [] 7)
```

and say that **exp** is the if expression from above. Once print-cep has reconstructed the if expression **exp** from its continuation and predicate, it will find the comb continuation below it on the stack. Say that the data unevaluated of this continuation held the expressions 3 and +, and the evaluated field held the number 7. Print-cep can see this, and will reconstruct the entire expression, `(+ 3 exp 7)`, from the inner expression **exp** and the continuation. When it is finished, it will use that expression

as the inner expression for the next continuation. When the empty continuation is encountered, the process halts, and `exp` is returned as the result.

In short, the continuation stack is printed by traversing it, top to bottom, and building a Scheme expression from the inside out. At each node of the stack, print-cep computes the Scheme expression with a hole in the middle that the continuation represents, and fills the hole with the inner expression. The procedures that perform that task are called `make-whole-expression` and `cont/handle-*`, with the expression type filled in for the *, and they are found in `print-cep.scm`. When the stack traversal is finished, the resulting expression is used as the body of the letrec expression, and together with the binding list generated by the environment, the result represents the state of the evaluation.

5.3 Conclusion

The task of printing the state of TREval is divided into three parts: printing the expression, environment, and continuation. Printing an expression is very straightforward, except for variables, which must be renamed according to how the environment is printed. Printing a continuation involves a linear traversal of the continuation stack, to build the result from the inside out. Printing an environment requires traversals of the environment structure, that cover all of the reachable frames, at each frame adding to the list of bindings. The end result of this is a letrec expression, that can be shown to the user, that embodies the state of computation, and represents a step in the substitution model evaluation of the original expression.

Chapter 6

Examples of Computation

This chapter will present four very simple examples of an interaction with the tail recursive evaluator and the printing code.

6.1 Running the Evaluator

To run the tail recursive evaluator, load the file load.scm, by typing `(load ''load.scm'')` into the Scheme buffer and hitting ctrl-x ctrl-e. Then, to start the read-eval-print loop, evaluate the expression `(doeval)`. When your interaction is complete, evaluating the expression `(exit-doeval)` at the doeval prompt will exit the tail recursive evaluator. (NOTE: doeval is a name that reflects the eccentricities of the author and his partner, and quite possibly might change at some later time to something more indicative of the structure of the evaluator.)

There are several variables that affect how doeval prints output. The variable `use-counter` is a boolean that tells the evaluator whether to print every certain amount of steps, or whether to use the set of heuristic rules to decide when to print. `Print-every` is the variable that says how often to print, if the counter is being used. `Minimal-numbering` is a flag which says which suffixer to use. By assigning various values to these variables, you can see what kind of effect they have on the output of the evaluator. These may be the topic of future changes, so they will not be explained any further in this paper. The output in the rest of this chapter was generated with

use-counter set to #f and minimal-numbering set to #t.

6.2 Evaluation of a Simple If Expression

The following is a transcript of evaluation for a simple if expression.

```
doeval==> (if (> 2 3) (+ 1 2) (- 2 1))

(letrec #f
  (if #f
      (+ 1 2)
      (- 2 1)))

(letrec #f
  (- 2 1))

(letrec #f
  1)
;Value: 1
```

Notice how the predicate is first evaluated to false, then the expression is reduced to the alternative.

6.3 Evaluation of a Combination

Here is the output from TREval during the evaluation of a combination.

```
doeval==> (+ (* 1 2) (+ 1 2) (if (+ 2 3) 1 2))

(letrec #f
  (+ (* 1 2) (+ 1 2) (if 5 1 2)))

(letrec #f
  (+ (* 1 2) (+ 1 2) 1))

(letrec #f
  (+ (* 1 2) 3 1))
```

```
(letrec #f
  (+ 2 3 1))

(letrec #f
  6)
;Value: 6
```

Here, we can see that TREval uses a right to left order of evaluation. Notice how each of the subexpressions is tackled completely before any of the others are started. Also, notice how the final step is to apply the primitive procedure + to the values of the argumentss, 2, 3 and 1. Applications of primitive procedures are considered "magic" in TREval, and as such are not printed. We will see in the next example how TREval deals with applications of compound procedures.

6.4 Evaluation of Factorial: Recursive

This section and the next will contrast the evaluation of a recursive factorial with that of an iterative factorial. One application of the substitution model, is to illustrate the concept of deferred operations, and order of growth in space of a function. The recursive version presented here has linear growth in space, or a linear amount of deferred operations. Without further ado, here is the edited transcript from the evaluation of `(fact 3)`, recursively.

```
doeval==> (define fact
    (lambda (n)
      (if (= n 1)
  1
  (* n (fact (- n 1))))))

(letrec ((fact (lambda (n) (if (= n 1) 1 (* n (fact (- n 1)))))))
  #[unspecified-return-value])
;Value: #[unspecified-return-value]
doeval==> (fact 3)

(letrec ((fact (lambda (n) (if (= n 1) 1 (* n (fact (- n 1)))))))
  ((lambda (n) (if (= n 1) 1 (* n (fact (- n 1))))) 3))
```

```
(letrec ((n 3) (fact (lambda (n) (if (= n 1) 1 (* n (fact (- n 1))))))) 
  (if (= n 1)
      1
      (* n (fact (- n 1)))))

(letrec ((n 3) (fact (lambda (n) (if (= n 1) 1 (* n (fact (- n 1))))))) 
  (if (= 3 1)
      1
      (* n (fact (- n 1)))))

(letrec ((n 3))
  (if #f
      1
      (* n (fact (- n 1)))))

(letrec ((n 3))
  (* n (fact (- n 1))))

(letrec ((n 3))
  (* n (fact (- 3 1))))

(letrec ((n 3))
  (* n (fact 2)))

(letrec ((n 3))
  (* n ((lambda (n) (if (= n 1) 1 (* n (fact (- n 1))))) 2)))

(letrec ((n-1 2) (n 3))
  (* n (if (= n-1 1) 1 (* n-1 (fact (- n-1 1))))))

.
.
.

(letrec ((n-1 2) (n 3))
  (* n (* n-1 (fact 1))))

(letrec ((n-2 1) (n-1 2) (n 3))
  (* n (* n-1 1)))

(letrec ((n-1 2) (n 3))
  (* n (* 2 1)))

(letrec ((n 3))
  (* n 2))
```

```
(letrec ((n 3))
  (* 3 2))

(letrec ()
  6)
;Value: 6
```

This transcript has been edited for the sake of readibility. The definition of fact has been cut from the binding lists after the first few steps. Also, some of the steps have been left out.

The first thing to notice is how compound procedures applications are handled, as shown by the first few steps. TREval reduces `(fact 3)` to the body of factorial, as shown, with n bound to 3 in the environment. It then proceeds with the evaluation of the body in the new environment. In other words, notice how `(fact 3)` reduces to `(* n (fact 2))`, with n bound to 3 in the environment. The multiplication by n is a deferred operation, it will be performed only after we compute the value of `(fact 2)`. Now, another interesting thing takes place. `(fact 2)` wants to turn into `(* n (fact 1))`, with n bound to 2. But, that would result in a name conflict for the variable n. What happens here, is that in the environment structure, there are two different frames, one with n bound to 3, and one with n bound to 2. When the environment is printed, one of these n's gets renamed to n-1, namely the one bound to 2. Also, when the expression is printed, the two occurances of the variable n are addressed, and one of them is renamed to n-1. The net result of this is `(* n (* n-1 (fact 1)))`, with n bound to 3 and n-1 bound to 2. Since one is the base case for factorial, `(fact 1)` reduces to one. Then, the two deferred operations are completed.

From this evaluation, it can be seen that recursive factorial has a linear amount of deferred operations. That is, the number of leftover multiplications is directly proportional to the size of the argument, n. In the next section, we will see that for an iterative factorial, this is not the case.

6.5 Evaluation of Factorial: Iterative

This section will present the iterative version of factorial. There will be an internal define of a helper function, and there will be no deferred operations.

```
doeval==> (define fact
    (lambda (n)
      (define helper
(lambda (n ans)
  (if (= n 1)
      ans
      (helper (- n 1) (* n ans)))))
      (helper n 1)))

(letrec ((fact
          (lambda (n)
            (begin
             (define helper
               (lambda (n ans)
                 (if (= n 1)
                     ans
                     (helper (- n 1) (* n ans)))))
             (helper n 1)))))
  #[unspecified-return-value])
;Value: #[unspecified-return-value]

doeval==> (fact 3)

(letrec ((fact
          (lambda (n)
            (begin
             (define helper
               (lambda (n ans)
                 (if (= n 1)
                     ans
                     (helper (- n 1) (* n ans)))))
             (helper n 1)))))
  ((lambda (n)
     (begin
      (define helper
        (lambda (n ans)
          (if (= n 1)
              ans
```

```
              (helper (- n 1) (* n ans)))))
      (helper n 1)))
   3))

(letrec ((n 3))
  (begin
   (define helper
     (lambda (n ans)
       (if (= n 1)
           ans
           (helper (- n 1) (* n ans)))))
   (helper n 1)))

(letrec ((helper (lambda (n ans) (if (= n 1) ans (helper (- n 1) (* n ans)))))
         (n 3))
  (begin (helper n 1)))

(letrec ((helper (lambda (n ans) (if (= n 1) ans (helper (- n 1) (* n ans)))))
         (n 3))
  (begin (helper 3 1)))

(letrec ((n-1 3)
         (ans 1)
         (n 3))
  (begin (helper (- n-1 1) (* n-1 ans))))

(letrec ((n-1 3)
         (ans 1)
         (n 3))
  (begin (helper (- n-1 1) (* n-1 1))))

(letrec ((n-1 3)
         (ans 1)
         (n 3))
  (begin (helper 2 3)))

(letrec ((n-1 2)
         (ans 3)
         (n 3))
  (begin (helper 1 6)))

(letrec ((fact
          (lambda (n)
            (begin
             (define helper
```

```
                (lambda (n ans)
                  (if (= n 1)
                      ans
                      (helper (- n 1) (* n ans)))))
              (helper n 1)))))
  6)
;Value: 6
```

This output has also been edited, the helper and fact bindings have been removed after their first appearance or two, and only the key steps have been included. The first interesting thing to observe is how the internal define is turned into a sequence, showing the implicit begin that is always present in the body of a procedure. Then, the define is executed, creating a binding for helper in the environment. The value of this define expression is discarded, since it isn't the last statement of the begin. The helper is called, with parameters n and 1, with n bound to 3 in the environment. This eventually reduces to another call to helper, with different parameters, 2 and 3, and finally, a last call, with 1 and 6. Since 1 is the base case for helper, the application reduces to the answer, 6. Notice, that when helper reached the base case, the answer 6 was just returned as the answer to the entire expression. In the recursive version, remember, there were additional multiplications that needed to complete, before the answer could be returned. This is one reason why iterative procedures are more efficient than recursive procedures, and it is explained very nicely by the substitution model.

6.6 Conclusion

It is difficult to get a real flavor for the uses of this program without trying it out for yourself. Think about a property of the substitution model that is difficult to understand, or some piece of code that has particularly confused you, and try it out to see how it really works. Toy with the parameters explained above, to try to get a clear output. Perhaps, in the future, the evaluator will be smarter about what it prints, but for now, it is worth it to find the key steps in the evaluation and see what

is going on. Who knows, you might learn something!

Chapter 7

Final Remarks

This chapter will address some of the issues that pertain to the tail recursive evaluator that have not yet been discussed by this paper. It will conclude with a summary of the key ideas that have been presented.

7.1 Parsing Expressions

As you may have noticed, TREval does not use list structure to represent Scheme expressions. The major consequence of this is that it must convert the list structure which is initially read from the user into the internal form used for evaluation. For each kind of expression, there is a data abstraction, with a constructor and selectors for each of the various pieces. Converting lists into this form is merely a matter of picking out the relevant data and using the new constructors to make the correct data structures.

This process can be thought of as a pre-evaluation stage, and it serves one other purpose besides data conversion. If the user types a malformed expression, it can be caught in this part of the evaluation, so that the evaluator need only worry about legal Scheme expressions. This is called parse-time error checking.

The data structures that are used to represent expressions are defined using the Scheme macro `define-structure`, which is explained in the Scheme reference manual [2].

7.2 Loose Ends and Future Directions

The goal of this project is to create a production quality substitution model evaluator. While TREval goes a long way towards realizing this, it still lacks in a few areas. The next few paragraphs discuss these areas, and give ideas for possible future work to be done on the interpreter.

First of all, there are some special forms that TREval does not implement up to specifications. Most notably, quasiquote, call/cc, and do have been left out, either entirely or partially. The specification for quasiquote is complex, and although we attempted to implement it, it was getting out of hand. We never got around to attempting to implement do, settling on named lets as a viable compromise. Fluid-let has also been omitted. Some other interesting features of Scheme, such as define-structure, are absent from TREval as well. In the future, this evaluator could be extended to include these and other features.

Additionally, TREval has only a rudimentary user interface. There are only a couple of options available for the user to change how TREval prints its output. In the future, these options may be extended, and the minimal printing heuristics that currently exist may also be extended. Perhaps tools may be created for navigating through the evaluation process, rather than leaving the user with pages of data to sort through. An interesting problem might be how to make TREval very smart about what and how it prints.

There is another issue pertaining to printing that is not quite right with TREval. Shared lists, i.e. lists that are `eq?`, are not printed as such. There is no way to determine between two different cons cells containing the same data and the same cons cell pointed to by two different names. Therefore, if evaluation of set-car! or set-cdr! is performed, with shared lists, not all of the steps printed out may lead to the right answer. TREval will always compute the right answer, it just won't be printed correctly along the way.

Yet another possibility for improvement in TREval is in efficiency. TREval is much more efficient than the previous substitution model evaluator, SMEval, since

SMEval essentially parsed the new expression every step along the path of evaluation. However, it is not as efficient as the production quality interpreters, such as the 6.001 interpreter on which it is based. There may be room for optimizations and improvements in this area.

All in all, there are several open areas of interest for future work on TREval. From the printing code, to the completeness or efficiency of the evaluator itself, there are many problems left with this evaluator, and it is my hope that in the future the work will be continued.

7.3 Summary

The tail recursive evaluator provides the user with the illusion of a substitution model Scheme interpreter. A direct implementation of the environment model, its iterative structure forces it to keep explicit track of the continuation stack. That information, together with the environment, is enough so that a Scheme expression can be printed at any time, which represents the total state of evaluation. By performing this task at key steps, the program presents the user with a picture of how the evaluation of the expression would have proceeded by the rules of the substitution model.

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)
'()
(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 (lst)
   (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))))
  ((exp/variable? exp) (make-variable 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
  (cdr 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))
```

```
      (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))))))
```

```
    (if (list? bindings)
(helper bindings)
#f)))

(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 (cadr exp) bad-value))
  ((> (length exp) 3)
   (error "malformed define: " exp))
  (else
   (make-definition (cadr 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)))))
  (if (list? name)
      (if (exp/access? name)
  (make-set!-statement (parse name) expr)
  (error "Incorrect second subform in set!: " exp))
      (make-set!-statement (parse name) expr))))))


(define parse/handle-sequence
  (lambda (exp)
    (if (> (length exp) 1)
(make-sequence (map parse (cdr 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)))
      ((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 Evlautor

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)))
```

```
 ((sequence? exp)
  (loop (ev-sequence exp env cont)))
 ((quasi? exp)
  (loop (ev-quasi exp env cont)))
 ((the-environment? exp)
  (loop (ev-the-environment exp env cont)))
 ((access? exp)
  (loop (ev-access exp env cont)))
 ((call-cc? exp)
  (loop (ev-call-cc exp env cont)))
 (else
  (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
```

```
      (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-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 #f expr)
   newenv)))
       (local-assignment newenv proc-name letproc)
       (make-cep expr newenv cont)))
    (else
     (make-cep (cadar bindings)
       env
       (make-cont 'let
```

```
  (make-data (list
      (cdr bindings)
      proc-name
      expr)
     (list (caar bindings)))
  env
  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)
  (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
```

```
   cont))))))))

(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
```

```
 (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
```

```
 (make-data (access-name exp) '())
 env
 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)
```

```
    (make-cont 'comb
       (make-data (cdr unev)
  (cons val ev))
       (cont-env cont)
       next-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)
```

```
  (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))
```

```
 (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)))
```

```
   (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
 (make-data (list (cdr bindings)
  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
(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))))
      (let ((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)))))))))

(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
```

```
   (cont-env cont)
   (cont-next-cont cont))))))
      (print-cep 'access newcep)
      newcep)))


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

(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)
  '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))))))
```

```
(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 unparser

;; 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))))
      (let ((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 (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 (lst)
       (list (if (else? (car lst))
 'else
```

```
 (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
```

```
    (cons (list cur-bind inner-exp)
  todo-bind))
    ((print-exp-for-let (cont-env cont) bounds)
     expr)))))))))

(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)))
    (expr (cadr (data-unev (cont-data cont)))))
(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))))
      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
      (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))))
      (if (access? name)
  (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)
'()))))))
      (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))))
       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)
   (list 'quasiquote
 (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)
   (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
```

```
  (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))
```

```
(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 ((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
       (cons 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
 (cons 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-frame-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))))
      (let* ((state 0)
     (new-int (lambda ()
(set! state (+ 1 state))
state)))
(lambda (var-name the-frame)
  (let ((val (hash-table/get the-var-frame-table
     (cons var-name the-frame) #f)))
    (if val
(make-suffix 'old val)
(let ((val (hash-table/get the-frame-table the-frame #f)))
  (if val
      (begin
(hash-table/put! the-var-frame-table
 (cons var-name the-frame) val)
(make-suffix 'new val))
      (let ((suffix (new-int)))
(hash-table/put! the-var-frame-table
 (cons var-name the-frame) suffix)
(hash-table/put! the-frame-table the-frame suffix)
(make-suffix 'new suffix)))))))))))
```

A.4.3 unparse.scm

```
;;;;;  The unparser

(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))
    '())
```

```
   ((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)
```

```
    (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 (letrexp env bound)
    (let ((newbounds (append (map car (letrec-statement-bindings
       letrexp))
     bound)))
      (list 'letrec
    (map (lambda (binding)
```

```
   (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)
```

```
   (list 'unquote (print-expression (unquoted-exp exp) env bound)))
  ((splice? exp)
   (list 'unquote-splicing (print-exprssion
    (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-enviroment)))

(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

[1] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Massachusetts, second edition, 1996.

[2] William Clinger and Jonathan Rees. *Revised Report on the Algorithmic Language Scheme.* Artificial Intelligence Laboratory of the Massachusetts Institute of Technology.

[3] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 1992.

[4] Robert Harper. Computer science 15-212. Class Notes, 1993.

[5] Martin Hoffman. Sound and complete axiomatisations of call-by-value control operators. *Mathematical Structures in Computer Science*, 1995.

[6] Abraham Maritime. An efficient substitution model scheme evaluator. Master's project, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1997.

[7] Albert Meyer. Computability, programming, and logic. Class Notes, 1996.

[8] http://www.cs.rice.edu/CS/PLT/packages.